HAL
open science

Effects and handlers in natural language

Jiří Maršík

To cite this version:

Jiří Maršík. Effects and handlers in natural language. Computation and Language [cs.CL]. Université de Lorraine, 2016. English. NNT : 2016LORR0322 . tel-01417467v3

HAL Id: tel-01417467

https://inria.hal.science/tel-01417467v3

Submitted on 22 Jun 2017

HAL is a multi-disciplinary open access archive for the deposit and dissemination of scientific research documents, whether they are published or not. The documents may come from teaching and research institutions in France or abroad, or from public or private research centers.

L'archive ouverte pluridisciplinaire **HAL**, est destinée au dépôt et à la diffusion de documents scientifiques de niveau recherche, publiés ou non, émanant des établissements d'enseignement et de recherche français ou étrangers, des laboratoires publics ou privés.

UNIVERSITÉ DE LORRAINE

AVERTISSEMENT

Ce document est le fruit d'un long travail approuvé par le jury de soutenance et mis à disposition de l'ensemble de la communauté universitaire élargie.

Il est soumis à la propriété intellectuelle de l'auteur. Ceci implique une obligation de citation et de référencement lors de l'utilisation de ce document.

D'autre part, toute contrefaçon, plagiat, reproduction illicite encourt une poursuite pénale.

Contact : ddoc-theses-contact@univ-lorraine.fr

LIENS

Code de la Propriété Intellectuelle. articles L 122. 4
Code de la Propriété Intellectuelle. articles L 335.2- L 335.10
http://www.cfcopies.com/V2/leg/leg_droi.php
http://www.culture.gouv.fr/culture/infos-pratiques/droits/protection.htm

UNIVERSITÉ DE LORRAINE

École doctorale IAEM Lorraine

Les effects et les handlers dans le langage naturel

Effects and Handlers in Natural Language

THÈSE

présentée et soutenue publiquement le 9 décembre 2016

pour l'obtention du

Doctorat de l'Université de Lorraine

(mention informatique)

par

Jiří Maršík

Composition du jury

Président :	VIGNERON Laurent	Université de Lorraine
Directeur de thèse :	DE GROOTE Philippe	Inria Nancy - Grand Est
Co-directeur de thèse :	AMBLARD Maxime	Université de Lorraine
Rapporteurs :	BARKER Chris HERBELIN Hugo	New York University Inria Paris - Rocquencourt
Examinateurs :	QUATRINI Myriam UNGER Christina	Institut de Mathématiques de Luminy Universität Bielefeld

Laboratoire Lorrain de Recherche en Informatique et ses Applications — UMR 7503

Acknowledgments

I would like to thank my supervisors, whose contributions have been essential to my work. Maxime Amblard was always eager to spend his time listening to all the developments in my work, including numerous dead ends, and providing help and support every step of the way. Philippe de Groote had a large impact on my work by sharing his insight and experience with the lambda calculus, leading to some of the most important developments in the first part of this thesis. I would also like to thank the other members of the jury, who have all gone above and beyond their duty and devoted their time to studying my work: Chris Barker, Hugo Herbelin, Myriam Quatrini, Christina Unger, Laurent Vigneron.

The research presented in this manuscript was carried out within the Sémagramme team at Inria. One of the key factors behind the quality of this work was the friendly atmosphere at the Sémagramme team fostered by its researchers, engineers, PhD students and interns: Clément Beysson, Sarah Blind, Vu Anh Duc, Bruno Guillaume, Nicolas Lefebvre, Pierre Ludmann, Aleksandre Maskharashvili, Guy Perrier, Sylvain Pogodalla, Sai Qian, Stéphane Tiv. My gratitude also extends to all the people that made the day-to-day working conditions at Loria very pleasant, notably the excellent staff at the restaurant, the administrative teams handling all the paperwork (Aurélie Aubry, Céline Simon, François Thaveau) and the many friends and colleagues that I got to play music with or chat during lunch breaks: Mihai Andries, Benoît Chappet de Vangel, Émilie Colin, Sylvain Contassot-Vivier, Baldwin Dumortier, Mariia Fedotenkova, Iñaki Fernández Pérez, Francesco Giovannini, Arseniy Gorin, Meysam Hashemi, Adrien Krähenbühl, Cecilia Lindig-Leon, Jérémy Miranda, Aybüke Özgün, Motaz Saad, Yann Salaun, Evangelia Tsiontsiou, Jan Volec.

My decision to stay in Nancy and do a PhD with the Sémagramme team was in one part due to an excellent introduction into the fields of formal grammar and computational semantics that I had during my Master studies there, taught by Maxime Amblard, Philippe de Groote, Bruno Guillaume, Guy Perrier and Sylvain Pogodalla. The other part that made me enjoy my Master studies in Nancy so much were my amazing classmates, talented individuals who were always eager to go to bars on weekdays and organize study group sessions on weekends: Bruno Andriamiarina Miharimanana, Georgy Boytchev, Caroline Ceballos, Tatiana Ekeinhor-Komi, Nicolas Feuillet, Natalia Korchagina, Imane Nkairi, Gabin Personeni, Wei Qiu, Camille Sauder.

During my PhD studies, I drew a lot of inspiration and motivation from my regular visits to the European Summer School of Logic, Language and Information. Among the many bright minds that I have had the time to discuss and spend time with, I would like to mention Lasha Abzianidze, Eleri Aedmaa, Pepijn Kokke and Stellan Petersson.

Before ending this section, I would also like to thank the people who have enriched my life outside of the office. I am grateful to Marie-George Alle-Hans for her great teaching skills during the university dancing classes. I am forever in debt to Nelly Madigou for carefully teaching me the basics of cello playing. I was very lucky to find and join the Orchestre Symphonique Universitaire de Lorraine, where I got to discover the joys of orchestral playing. I also wish to thank the many friends and flatmates who have kept me company during my stay in Nancy: Aurélie Boizard, Orianne Cabaret, Marie Gantet, Marine Larrière, Jordan Medic, Dorine Petit, Gautier Vacheron.

Finally, I wish to send the biggest thanks to my parents and family who have supported me during my long studies abroad.

Tuto práci věnuji
mým rodičům

Contents

Introduction **1**

I Calculus of Effects and Handlers **5**

1 Definitions **7**
1.1 Sketching Out the Calculus . . . 7
1.2 Terms . . . 9
1.3 Types and Typing Rules . . . 10
1.4 Reduction Rules . . . 15
1.5 Sums and Products . . . 20
1.5.1 New Terms . . . 20
1.5.2 New Types . . . 21
1.5.3 New Reduction Rules . . . 21
1.5.4 Booleans . . . 21
1.6 Common Combinators . . . 23
1.6.1 Composing Functions and Computations . . . 23
1.6.2 Operations and Handlers . . . 25

2 Examples **27**
2.1 Introducing Our Running Example . . . 27
2.2 Adding Errors . . . 28
2.2.1 Raising the Type of exp . . . 29
2.2.2 Refactoring with Monads . . . 29
2.2.3 Interpreting Expressions as Computations . . . 30
2.2.4 Handling Errors . . . 30
2.2.5 Examples with Errors . . . 31
2.3 Enriching the Context with Variables . . . 33
2.3.1 Example with Variables . . . 34
2.3.2 Treating Variables without Computations . . . 36
2.4 Summary . . . 38

3 Properties **39**
3.1 Derived Rules . . . 39
3.1.1 Function Composition ($\circ$) . . . 40
3.1.2 Monadic Bind ($\gg\!=$) . . . 40
3.1.3 Closed Handlers ($(\!|\ |\!)$) . . . 41
3.2 Type Soundness . . . 42
3.2.1 Subject Reduction . . . 43
3.2.2 Progress . . . 46
3.3 Algebraic Properties . . . 47
3.3.1 Denotational Semantics . . . 48
3.3.2 Category . . . 50
3.3.3 The Three Laws . . . 51
3.3.4 Functor . . . 52
3.3.5 Applicative Functor . . . 53
3.3.6 Monad . . . 56
3.3.7 Free Monad . . . 57
3.4 Confluence . . . 58
3.4.1 Combinatory Reduction Systems . . . 58
3.4.2 $(\!|\lambda|\!)$ as a CRS . . . 60
3.4.3 Orthogonal CRSs . . . 61
3.4.4 Putting η Back in $(\!|\lambda|\!)$ . . . 63
3.5 Termination . . . 64
3.5.1 Inductive Data Type Systems . . . 65
3.5.2 $(\!|\lambda|\!)$ as an IDTS . . . 66
3.5.3 Termination for IDTSs . . . 70
3.5.4 Higher-Order Semantic Labelling . . . 73
3.5.5 Putting η Back in $(\!|\lambda|\!)$ . . . 81

4 Continuations **85**
4.1 Introducing Call-by-Value . . . 86
4.2 Simulating Call-by-Value . . . 86
4.3 Introducing Control Operators . . . 88
4.4 Simulating `shift0` and `reset0` . . . 89
4.5 Turning to `shift` and `reset` . . . 92
4.6 Considering Types . . . 94
4.7 Other Control Operators . . . 101

II Effects and Handlers in Natural Language **103**

5 Introduction to Formal Semantics **105**
5.1 Montague Semantics . . . 105
5.1.1 Syntax . . . 106
5.1.2 Semantics . . . 108
5.1.3 Summary . . . 111
5.2 Abstract Categorial Grammars . . . 111
5.2.1 Definition . . . 111
5.2.2 Syntax . . . 113
5.2.3 Semantics . . . 114
5.2.4 Summary . . . 116
5.3 Discourse Representation Theory . . . 116
5.3.1 Discourse Representation Structures . . . 118
5.3.2 DRSs as Contents . . . 119
5.3.3 DRSs as Contexts . . . 120
5.3.4 Removing Implication and Disjunction . . . 121
5.4 Type-Theoretic Dynamic Logic . . . 121
5.4.1 Logic . . . 123
5.4.2 Monadic Structure . . . 123
5.4.3 Example Lexical Entries . . . 124

6 Introducing the Effects **127**
6.1 Lifting Semantics into Computations . . . 127
6.2 Deixis . . . 131
6.2.1 Quotations . . . 132
6.2.2 Algebraic Considerations . . . 133
6.3 Conventional Implicature . . . 134
6.3.1 Algebraic Considerations . . . 136
6.4 Quantification . . . 137
6.4.1 Quantifier Ambiguity . . . 139
6.4.2 Algebraic Considerations . . . 142
6.5 Methodology . . . 142
6.5.1 Using Computation Types . . . 143
6.5.2 Digression: Call-by-Name and Call-by-Value . . . 144
6.5.3 Choosing an Effect Signature . . . 146
6.5.4 When (Not) to Use Effects . . . 150

7 Dynamic Semantics in $(\!|\lambda|\!)$ **153**
7.1 DRT as a Programming Language . . . 154
7.2 $(\!|\lambda|\!)$ Analysis . . . 159
7.2.1 Example . . . 162

7.2.2 Handler for Dynamics . . . 163
7.2.3 Negation . . . 164
7.2.4 Truth Conditions as Side Effects . . . 166
7.2.5 Algebraic Considerations . . . 167
7.3 Presuppositions . . . 169
7.3.1 Revising the Dynamic Handler . . . 171
7.3.2 Presupposition in Action . . . 174
7.3.3 Cancelling Presuppositions . . . 176
7.3.4 Ambiguous Accommodation . . . 181
7.3.5 Comparison with TTDL . . . 193
7.4 Double Negation . . . 194
7.4.1 Double Negation as an Effect . . . 195
7.4.2 DN-TTDL is Not Monadic . . . 196
7.4.3 DN-TTDL is Not Functorial . . . 197
7.5 Summary . . . 198

8 Composing the Effects 201
8.1 Dynamic Kernel . . . 202
8.2 Adding Presuppositions . . . 206
8.3 Adding Conventional Implicature . . . 208
8.3.1 Connection to the Standalone Theory . . . 210
8.3.2 Connection to Layered DRT and Projective DRT . . . 211
8.4 Adding Deixis . . . 212
8.5 Adding Quantification . . . 214
8.5.1 Quantifier Raising — Inverse Scope and Crossover . . . 215
8.6 Considering Restrictive Relative Clauses . . . 219
8.6.1 Different Interpretation for Nouns . . . 221
8.6.2 Relative Clauses and Presuppositions . . . 222
8.7 Summary . . . 227
8.7.1 Note on Conservativity . . . 228

9 Conclusion 229
9.1 Summary of Results . . . 229
9.2 Comparison with Existing Work . . . 231
9.2.1 Calculus . . . 231
9.2.2 Linguistic Modelling . . . 234
9.2.3 Combining Linguistic Effects . . . 236
9.3 Future Work . . . 240
9.3.1 Future Work on the Calculus . . . 240
9.3.2 Future Work on the Linguistic Applications . . . 242

Bibliography **245**

Appendices **253**

A List of Examples **253**

B Example from the Final Fragment **255**

C Computer Mechanization of the Calculus **261**

Résumé en français **277**

Introduction

The motivation behind this thesis is to advance the automatic process of translating natural language into logical representations. The interest behind this translation procedure is two-fold: linguists use it to give a formal semantics to natural languages and computer scientists use it to perform automated reasoning on natural language data.

There has been considerable research into translating parts of English and other natural languages. When looking at a sentence of English, we can identify many of the problems inherent in the translation and point to papers that have proposed solutions.

(1) She might still be dating that idiot.

1. We have anaphoric expressions, such as the pronoun *she*. We know that we can translate sentences with anaphora into Discourse Representation Theory (DRT) structures [64], Dynamic Predicate Logic (DPL) formulas [53] or continuation-passing style λ-terms [38].

2. We have the modal auxiliary *might* that we can translate into an operator of modal logic or directly to some existential quantification over possible worlds.

3. We have the progressive present tense in *be dating*. Similarly to the modal, we can translate this into an operator of temporal logic or directly posit the existence of some interval of time during which the dating takes place and assert that the time of utterance (possibly) lies in that interval.

4. We have the presupposition trigger *still* that tells us that the two subjects must have been dating in the past already. We will have to possess some mechanism to project this contribution outside the scope of any of the logical operators present.[1] We can adopt the strategy of Lebedeva [80] and raise exceptions to perform the projection.

5. We have the expressive epithet *idiot*. Following the theory of conventional implicatures of Potts [108], elaborated by Gutzmann [54], we know that we should introduce a second dimension into which we tuck away the speaker's negative attitude towards the date-ee so that it does not interfere with the at-issue content.

All of the advice given above seems sound. We could follow these guidelines to intuitively arrive at some reasonable logical representation. Now how do we write down this joint translation process? Most of the theories mentioned above come with their own language: their own definitions, notation and operations.

DRT introduces its own encoding of logical formulas and states an algorithm that builds them up incrementally from the input sentence [64]. Potts's logic of conventional implicatures introduces two-dimensional logical formulas and defines new modes of combination to compute with them [108]. Compositional treatments of intensionality or tense tend to use the simply-typed lambda calculus [18, 40], as is also the case in de Groote's treatment of anaphora [38]. In studying presuppositions, Lebedeva uses a modified version of de Groote's calculus which includes exceptions [80].

It seems clear that in order to arrive at a precise notion of what it means do all of 1–5, we will first have to be able to express the theories behind 1–5 using a common formal language.

[1] While in the present, we can only infer that they *might* be dating, by accommodating the presupposition, we can infer that sometime in the past, they *must have been* dating.

Enter Monads

We will base our universal language on the λ-calculus. Thanks to Richard Montague's hugely influential work [98], the λ-calculus is already a very popular formalism in formal compositional semantics.[2] Many phenomena are analyzed using the λ-calculus and the rest tend to get translated to λ-calculus as well (see, for example, λ-DRT [79] or de Groote's continuation-based approach to dynamics [38]).

However, even though we have several theories which are all formalised in λ-calculus, it does not necessarily mean that they are compatible or that we know how to combine them together. A theory of intensionality might state that sentences ought to be translated to terms that have the type $\sigma \to o$, the type of functions from possible worlds to truth values. On the other hand, an account of expressives would argue that sentences ought to correspond to terms of type $o \times \epsilon$, the type of pairs of truth values (propositional content) and expressive markers (expressive content). The two theories would be compatible at the calculus-level but not at the term-level. A function operating with intensional propositions would not be directly applicable to an expressive proposition.

To continue our quest for uniformity and compatibility of semantic operations, we will look at the terms and types used by the different λ-theoretic treatments of semantics and try to find a common structure underneath. We notice that all such approaches share at least the following structure:

1. The types of some of the denotations get expanded. For example, when dealing with quantifiers, the type of denotations of noun phrases goes from ι (single individuals) to $(\iota \to o) \to o$ (generalized quantifiers over individuals); in intensional semantics, the type of denotations of sentences goes from o (truth values) to $\sigma \to o$ (functions from possible worlds to truth values, i.e. sets of possible worlds); and with expressives, the type of denotations of sentences goes from o to $o \times \epsilon$ (truth values coupled with expressive markers).

2. There is a process that can lift denotations of the old type into denotations of the new type. In the quantifier example, this is the famous type raising operation. In the intensional example, this is the **K** combinator that turns a truth value into a constant function that assigns that truth value to all worlds, a rigid intension. In the expressive example, this is the function that couples a proposition with a neutral/empty expressive marker.

3. Then there are other inhabitants of the expanded type that are not found by using the lifting function described above; those are the ones for which we expanded the type. Quantificational noun phrases such as *everyone* are not the result of type raising any specific individual. Intensional propositions such as *Hespherus is Phosphorus* have extensions that vary from world to world. Expressives such as the diminutive *Wolfie* will point to some individual and also carry some expressive marker that conveys the speaker's attitude towards the referent.

4. Finally, these approaches also have some general way of composing smaller denotations into larger ones and dealing with the added complexity caused by the more elaborate types. When applying a transitive verb to a quantificational subject and object, we let one (often the subject) first take scope and then we let the other take scope. When applying the verb to intensional arguments, we pass the world at which we are evaluating the sentence down to both the subject and the object. When applying it to expressive arguments, we apply the verb to the referents of both the subject and the object and on the side we collect the expressive content of both.

This kind of structure is very commonly seen in functional programming and in denotational semantics of programming languages. It is the structure of an *applicative functor* [91] (or *strong lax monoidal functor*, for the categorically-inclined). The above examples are also instances of a more special structure called a *monad* [95].

We will not go into the minutiae of the definition of a monad here but we will give a rough sketch nonetheless. A monad is a triple $(T, \eta, \gg\!=)$ where T is a function on types (the type expansion we saw

[2] Frege's compositionality principle states that the meanings of complex expressions should be determined by (i.e. be functions of) the meanings of its constituents. If complex meanings are to be seen as functions of other meanings, it makes sense to use a calculus of functions, i.e. the λ-calculus.

in 1), η is some way of lifting simple values into expanded values (the lifting functions in 2) and $\gg\!=$ gives us a general way of combining values of this expanded type (similar to the examples given in 4).[3] The triple also has to satisfy some algebraic properties which guarantee that composing functions of expanded types is associative and that the lifting function serves as a unit for this composition operator.

The examples given above are all instances of monads. The prevalence of monads in natural language semantics has been already discovered by Shan in [113]. However, the challenge lies in trying to use several monads at the same time.

Linguistic Side Effects

Monads often appear in denotational semantics of programming languages to account for notions of computation commonly referred to as *side effects* [95]. We can base ourselves on this correspondence and regard the monadic structure in natural language as linguistic side effects. This analogy was pursued by Shan [115, 116] and Kiselyov [67] and is present in recent work on monads in natural language semantics [48, 27]. However, the idea itself stretches back before monads were even introduced to computer science. In their 1977 paper [57], Hobbs and Rosenschein take a computational perspective on the intensional logic of Montague [98]: intensions correspond to programs and extensions correspond to values. A program can read the value of global variables that describe the state of the world.[4] The operators $\uparrow$ and $\downarrow$, which map between extension-denoting expressions and intension-denoting expressions, then correspond to the Lisp-style operators `quote` and `eval` respectively.

The idea of treating linguistic expressions as effectful actions or programs is also very relevant to dynamic semantics, which treats the meanings of sentences as instructions to update some common ground or other linguistic context.[5] Dynamic semantics and anaphora are sometimes classified as belonging to both semantics and pragmatics. This is also the case for other phenomena that we will treat as side effects in our dissertation: deixis, presupposition, conventional implicature. Pragmatics studies the way a language fits into the community of its users, i.e. how it is actually used by its speakers to achieve their goals. It might then come as no surprise that pragmatics align well with side effects in programming languages since side effects themselves concern the ways that programs can interact with the world of their users (e.g. by making things appear on screen or by listening for the user's input).

Effects and Handlers

By looking at the different monadic structures of natural language semantics as side effects, we can apply theories that combine side effects to find a formalism that can talk about all the aspects of language at the same time. One such theoretical framework are effects and handlers. In this framework, programs are interpreted as sequences of instructions (or more generally as decision trees).[6] The instructions are symbols called *operations*, which stand for the different effects, the different ways that programs can interact with their contexts. In our application to natural language semantics, here are some examples of operations that will feature in our demonstrations, along with their intended semantics:[7]

- `introduce` introduces a new discourse referent to the context. This is the kind of operation used by noun phrases such as the indefinite *a man*.

- `presuppose` P presupposes the existence of an entity satisfying the predicate P. This is used by definite descriptions *the* P and by proper nouns.

[3] This way of presenting a monad (a type constructor, η and $\gg\!=$) is particular to functional programming. Note that this presentation differs from the category-theoretical one which replaces $\gg\!=$ with a natural transformation μ [88].

[4] Dependence on an environment of some type σ is a side effect that can be described using the reader monad. This monad lifts the type α to the type $\sigma \to \alpha$. This is exactly the change of types that is prescribed by theories of intensionalization [18, 40].

[5] The use of monads to encode dynamic effects (anaphora) dates back to 2009 and the work of Giorgolo and Unger [52, 129].

[6] More precisely, we are interpreting programs in a free monad [123].

[7] The operations are just symbols and so have no inherent meaning.

- `implicate` i states that i is a conventional implicature. This operation is used by appositive constructions such as *John, who is my neighbor*.
- `speaker` asks the context for the identity of the speaker. This is used by the first-person pronoun to find its referent.

The process of calculating the denotation of a linguistic expression is broken down to these operations. When expressions combine to form sentences and discourses, these operations end up being concatenated into a large program which will want to perform a series of interactions with its context. This is when handlers come into play. A *handler* is an interpreter that gives a definition to the operation symbols in a program. Handlers can be made modular [8] so that the interpreter for our vocabulary of context interactions can be defined as the composition of several smaller handlers, each treating a different aspect of language (dynamicity, implicatures, deixis…).

When using effects and handlers, we therefore start by enumerating the set of interactions that programs (i.e. linguistic expressions in our application) can have with their contexts. Then, we can interpret linguistic expressions as sequences of such instructions. Finally, we write handlers which implement these instructions and produce a suitable semantic representation. This approach thus closely follows the mantra given by Lewis:

> In order to say what a meaning is, we may first ask what a meaning does and then find something that does that.
>
> General Semantics, David Lewis [83]

We can trace the origins of effects and handlers to two strands of work. One is Cartwright and Felleisen's work on Extensible Denotational Language Specifications [24], in which a technique for building semantics is developed such that when a (programming) language is being extended with new constructions (and new side effects), the existing denotations remain compatible and can be reused. The other precursor is Hyland's, Plotkin's and Power's work on algebraic effects [60], a categorical technique for studying effectful computations, which was later extended by Plotkin and Pretnar to include handlers [103, 110, 104]. The technique has gained in popularity in recent years (2012 and onward). It finds applications both in the encoding of effects in pure functional programming languages [71, 70, 63, 22] and in the design of programming languages [16, 87, 41, 73, 56]. Our thesis will explore the applicability of effects and handlers to natural language semantics.

Plan

The manuscript of this thesis is split into two parts. In the first, we develop a formal calculus that extends the simply-typed lambda calculus with a free monad for effects and handlers. We prove some key properties, such as strong normalization, and we show how the calculus relates to other similar notions such as continuations or monads. In the second part, we analyze some of the aspects of linguistic meaning as side effects: deixis, conventional implicature, quantification, anaphora and presupposition. We then incrementally build up a fragment that features all of those features and demonstrates some of their interactions.

[8] In a similar way that monads can be turned into monad transformers (monad morphisms) and then composed [113, 134].

Part I

Calculus of Effects and Handlers

We will present a calculus with special constructions for dealing with effects and handlers, which we will then apply to the problem of natural language semantics in the second part of the thesis. Our calculus, which we will call $(\!| \lambda |\!)$, is an extension of the simply-typed λ-calculus (STLC). We enrich STLC with a type for representing effectful computations alongside with operations to create and process values of this type.

1

Definitions

We are tempted to start by first giving the formal definitions of all the essential components of $(\!| \lambda |\!)$:

- the syntax of the terms in $(\!| \lambda |\!)$
- the syntax of the types in $(\!| \lambda |\!)$
- the judgments that relate types to terms
- the reduction semantics

However, before we do so, we will briefly sketch the ideas behind $(\!| \lambda |\!)$ so you can start building an intuition about the meaning of the symbols that we will be introducing below.

Contents

1.1 Sketching Out the Calculus **7**
1.2 Terms **9**
1.3 Types and Typing Rules **10**
1.4 Reduction Rules **15**
1.5 Sums and Products **20**
1.5.1 New Terms 20
1.5.2 New Types 21
1.5.3 New Reduction Rules 21
1.5.4 Booleans 21
1.6 Common Combinators **23**
1.6.1 Composing Functions and Computations 23
1.6.2 Operations and Handlers 25

1.1 Sketching Out the Calculus

We will be adding a new type constructor, $\mathcal{F}$, into our language. The type $\mathcal{F}(\alpha)$ will correspond to effectful *computations* that produce values of type α. The idea comes from the programming language Haskell and its use of monads [95, 133, 62]. Our type constructor $\mathcal{F}$ will also stand in for a monad, one that has been already encoded in Haskell in several ways [71, 63]. The motivation behind $(\!| \lambda |\!)$ is to build a minimal language which directly gives us the primitive operations for working with this particular monad. This way, we end up with a language that:

- is smaller than Haskell (and thus more mananageable to analyse),
- is closer to the STLC favored by semanticists,

- and which makes more evident the features that our proposal relies on.

The transition from type α to a type $\mathcal{F}(\alpha)$ is meant as a generalization of a scheme often seen in semantics when novel forms of meaning are studied (e.g., type raising [98], dynamization [80], intensionalization [40]). The distinction between the type α and the type $\mathcal{F}(\alpha)$ will, in different analyses, align with dichotomies such as the following:

- reference/sense
- static/dynamic meaning
- extension/intension
- semantics/pragmatics

The question then is, what form should our general $\mathcal{F}$ type constructor take? We want to have a construction that can combine all the existing ones. One can do the combining at the level of monads with the use of monad transformers, a technique pioneered by Moggi and very well-established in the Haskell programming community [95]. Simon Charlow has made the case that this technique can be exploited to great benefit in natural language semantics as well [27].

However, a competing technique has emerged in recent years and it is the goal of this thesis to introduce it to semanticists and verify its applicability to the study of natural language. The technique goes by many names, "algebraic effects and handlers" and "extensible effects" being the most commonly used ones. This is in part due to the fact that it lies at the confluence of several research programs. This fact will allow us to present the theory from two different perspectives so you can be equipped with two different intuitive models.

Algebraic Effects and Handlers

Hyland, Power and Plotkin have studied the problem of deriving denotational semantics of programming languages that combine different side effects [60]. In their approach, rather then modeling the individual effects using monads and combining the monads, every effect is expressed in terms of *operators* on computations. Computations thus become algebraic expressions with effects as operations and values as part of the generator set.

Let us take the example of nondeterminism. In the monadic framework, this effect is analyzed by shifting the type of denotations from α to the powerset $\mathcal{P}(\alpha)$. In the algebraic framework, a binary operator $+$ is introduced and is given meaning through a set of equations. In this case, these are the equations of a semilattice (stating the operator's associativity, commutativity and idempotence).

When the time comes to combine two effects, their signatures are summed together and their theories are combined through either a sum or a tensor (tensor differs from sum in that it adds commutativity laws for operators coming from the two different effects).

In order to fit exception handlers into their theory, Plotkin and Pretnar enriched the theory with a general notion of a *handler* [104]. A handler's purpose is to replace occurrences of an operator within a computation by another expression. This notion was shown to be very useful. Since using a handler on a computation is similar to interpreting its algebraic expression in a particular algebra, in many practical applications, the use of handlers has replaced equational theories altogether [16, 63, 22].

Extensible Effects

In the early 90's, Cartwright and Felleisen were working on the following problem. Imagine you have a simple programming language along with some denotational semantics or some other interpretation. In your simple language, numerical expressions might be interpreted as numbers. In that case, the literal number 3 would denote the number 3 and the application of the sum operator to two numerical expressions would denote the sum of their interpretations. Now imagine that you want to add mutable variables to your language. Numerical expressions no longer denote specific numbers, but rather functions from states of the variable store to both a number and an updated variable store (since expressions

can now both read from and write to variables). The number 3 is thus no longer interpreted as the number 3 but as a combination of a constant function yielding the number 3 and an identity function. The addition operator now has to take care to thread the state of the memory through the evaluation of both of its arguments. In short, we are forced to give new interpretations for the entire language.

Cartwright and Felleisen proposed a solution to this problem [24]. In their system, an expression can either yield a value or produce an effect. If it produces an effect, the effect percolates through the program all the way to the top, with the context that the effect projected from stored as a continuation. The effect and the continuation are then passed to an external "authority" that handles the effect, often by producing some output and passing it back to the continuation. When a new feature is added to the language, it often suffices to add a new kind of effect and introduce a new clause into the central "authority". The central authority then ends up being a collection of small modular interpreters for the various effect types. Denotation-wise, every expression can thus have a stable denotation which is either a pure value or an effect request coupled with a continuation.

Later on, this project was picked up by Kiselyov, Sabry and Swords, who, following Plotkin and Pretnar's work on handlers, proposed to break down the "authority" into the smaller constituent interpreters and have them be part of the language themselves [71].

Synthesis

In our language, values of type $\mathcal{F}(\alpha)$ can be seen either as algebraic expressions or as programs. Under the algebraic perspective, an expression is either a variable or an operator applied to some other expressions, whereas under the "extensible effects" perspective, a program is either a value or a request for an effect followed by some other programs (the continuation).

Our calculus will also have a special form for defining handlers. In the "algebraic effects and handlers" frame of mind, these can be thought of as algebras that interpret the operations within an algebraic expression. On the other hand, with "extensible effects", the intuition is more similar to that of an exception handler which intercepts requests of a certain type and decides how the computation should continue.

1.2 Terms

Having sketched the idea behind our calculus, we will now turn our attention to the specifics. We start by defining the syntactic constructions used to build the terms of our language.

Without further ado, we give the syntax of the expressions of our language. First off, let $\mathcal{X}$ be a set of variables, Σ a typed signature and $\mathcal{E}$ a set of operation symbols.

The expressions of our language are comprised of the following:

variable x, where x is a variable from $\mathcal{X}$

constant c, where c is a constant from Σ

abstraction $\lambda x.\, M$, where x is a variable from $\mathcal{X}$ and M is an expression

application $M\, N$, where M and N are expressions

injection $\eta\, M$, where M is an expression

operation $\mathtt{op}\, M_{\mathrm{p}}\, (\lambda x.\, M_{\mathrm{c}})$, where $\mathtt{op}$ is an operator from $\mathcal{E}$, x is a variable from $\mathcal{X}$ and M_{p} and M_{c} are expressions

handler $(\!|\, \mathtt{op}_1 \colon M_1,\ \ldots,\ \mathtt{op}_n \colon M_n,\ \eta \colon M_\eta \,|\!)$ where $\mathtt{op}_i$ are operators from $\mathcal{E}$ and M_i and M_η are expressions

extraction $\downarrow$

exchange $\mathcal{C}$

The first four constructions — variables, constants, abstractions and applications — come directly from STLC with constants.

The next four deal with the algebraic expressions used to encode computations. Let us sketch the behaviors of these four kinds of expressions under the two readings outlined above.

Algebraic Expressions – The Denotational View

The set of algebraic expressions is generated by closing some generator set over the operations of the algebra. The η function serves to inject values from the generator set into the set of algebraic expressions. It is the constructor for the atomic algebraic expressions.

Next, for every symbol $\mathtt{op}$ in $\mathcal{E}$, we have a corresponding constructor $\mathtt{op}$ in our calculus. $\mathtt{op}$ is a constructor for algebraic expressions whose topmost operation is $\mathtt{op}$. The $\mathtt{op}$ constructor takes as argument a function that provides its operands, which are further algebraic expressions.

The banana brackets $(\!|\, \mathtt{op}_1 \colon M_1,\ \ldots,\ \mathtt{op}_n \colon M_n,\ \eta \colon M_\eta \,|\!)$ contain algebras: interpretations of operators and constants. These components are combined into a catamorphism that can interpret algebraic expressions (hence the use of banana brackets [93])[9].

The extraction function $\downarrow_{\circ}$, pronounced "cherry", takes an atomic algebraic expression (the kind produced by η) and projects out the element of the generator set.

Effectful Computations – The Operational View

We will now explain these constructions from the computational point of view.

The η function "returns" a given value. The result of applying it to a value x is a computation that immediately terminates and produces the value x.

The symbols from $\mathcal{E}$ become something like system calls. A computation can interrupt its execution and throw an exception with a request to perform a system-level operation. For every symbol $\mathtt{op}$ in $\mathcal{E}$, there is a constructor $\mathtt{op}$ that produces a computation which issues a request to perform the operation $\mathtt{op}$. This constructor takes as an argument a continuation which yields the computation that should be pursued after the system-level operation $\mathtt{op}$ has been performed.

The banana brackets $(\!|\, \mathtt{op}_1 \colon M_1,\ \ldots,\ \mathtt{op}_n \colon M_n,\ \eta \colon M_\eta \,|\!)$ describe handlers: they contain clauses for different kinds of interrupts (operation requests) and for successful computations (clause η). They behave very much like handlers in languages with resumable exceptions such as Common Lisp or Dylan.

Finally, the cherry function $\downarrow_{\circ}$ can take a computation that is guaranteed to be free of side effects and run it to capture its result.

The 9th construction in our calculus is the $\mathcal{C}$ operator. $\mathcal{C}$ serves as a link between the function type discussed by STLC (constructions 1–4) and the computation type introduced in our calculus (constructions 5–8). $\mathcal{C}$ is a (partial) function that takes a computation that produces a function and returns a function that yields computations. In a way, $\mathcal{C}$ makes abstracting over a variable and performing an operation commute together[10].

We will see the utility of $\mathcal{C}$ later on. The idea came to us from a paper by Philippe de Groote [39] which tried to solve a similar problem. The name comes from the **C** combinator, which reorders the order of abstractions in a λ-term.

1.3 Types and Typing Rules

We now give a syntax for the types of $(\!|\, \lambda \,|\!)$ alongside with a typing relation. In the grammar below, ν ranges over atomic types from a set $\mathcal{T}$.

[9] Since the banana brackets can contain an arbitrary number of operator clauses, we adopt the syntax of named parameters/records used in languages such as Ruby, Python or JavaScript.

[10] This is *very* reminiscent of the idea behind Paul Blain Levy's call-by-push-value calculus [82], which treats abstracting over a variable as an effectful operation of popping a value from a stack. Using call-by-push-value could prove to be a rewarding way to refine our approach.

The types of our language consist of:

function $\alpha \to \beta$, where α and β are types

atom ν, where ν is an atomic type from $\mathcal{T}$

computation $\mathcal{F}_E(\alpha)$, where α is a type and E is an effect signature (defined next)

The only novelty here is the $\mathcal{F}_E(\alpha)$ computation[11] type. This type will be inhabited by effectful computations that have permission to perform the effects described in E and yield values of type α. The representation will be that of an algebraic expression with operators taken from the signature E and generators of type α.

In giving the typing rules, we will rely on the standard notion of a *context*. For us, specifically, a context is a partial mapping from the variables in $\mathcal{X}$ to the types defined above. We commonly write $\Gamma, x : \alpha$ for a context that assigns to x the type α and to other variables y the type $\Gamma(y)$. We also write $x : \alpha \in \Gamma$ to say that the context maps x to α. Note, however, that for $\Delta = \Gamma, x : \alpha, x : \beta$, we have $x : \beta \in \Delta$ while $x : \alpha \notin \Delta$.

Effect signatures are very much like contexts. They are partial mappings from the set of operation symbols $\mathcal{E}$ to pairs of types. We will write the elements of effect signatures the following way: $\mathtt{op} : \alpha \rightarrowtail \beta \in E$ means that E maps $\mathtt{op}$ to the pair of types α and β.[12] When dealing with effect signatures, we will often make use of the disjoint union operator $\uplus$. The term $E_1 \uplus E_2$ serves as a constraint demanding that the domains of E_1 and E_2 be disjoint and at the same time it denotes the effect signature that is the union of E_1 and E_2.

The last kind of dictionary used by the type system is a standard *higher-order signature* for the constants (a map from names of constants to types). For those, we adopt the same conventions.

In our typing judgments, contexts will appear to the left of the turnstile and they will hold information about the *statically* (lexically) bound variables, as in STLC. Effect signatures will appear as indices of computation types and they will hold information about the operations that are *dynamically* bound by handlers. Finally, there will be a single higher-order signature that will globally characterize all the available constants.

The typing judgments are presented in Figure 1.1. Metavariables M, N... stand for expressions, α, β, γ... stand for types, Γ, Δ... stand for contexts, $\mathtt{op}$, $\mathtt{op}_i$ stand for operation symbols and E, E'... stand for effect signatures. Σ refers to the higher-order signature giving types to constants.

The typing rules mirror the syntax of expressions. Again, the first four rules come from STLC. The next four deal with introducing pure computations, enriching them with effectful operations, handling those operations away and finally eliminating pure computations. The $\mathcal{C}$ rule lets us start to see what we meant by saying that the $\mathcal{C}$ operator lets the function type and the computation type commute.

Let us ponder the types of the new constructions so as to get a grip on the interface that the calculus provides us for dealing with computations.

[η]

First off, we have the η operator. It takes a value of type α and injects it into the type $\mathcal{F}_E(\alpha)$. The metavariable E is free, meaning η can take values of type α to type $\mathcal{F}_E(\alpha)$ for any E. The algebraic intuition would say that elements of the generator set are valid algebraic expressions independent of the choice of signature. Computationally, returning a value is always an option, independently of the available permissions.

[$\mathtt{op}$]

More complicated computations can be built up by extending existing computations using the *operation* construction. Let us have an effect signature E such that $\mathtt{op} : \alpha \rightarrowtail \beta \in E$. To use $\mathtt{op}$, we first apply it

[11] Throughout this manuscript, we will be using the term *computation* to mean values of type $\mathcal{F}_E(\alpha)$. Programs written in $(\!|\lambda|\!)$ are simply called terms and their normal forms are called values. To break it down, in $(\!|\lambda|\!)$, terms evaluate to values, some of which can be computations (those of an $\mathcal{F}$ type).

[12] The two types α and β are to be seen as the operation's *input* and *output* types, respectively.

$$\frac{x : \alpha \in \Gamma}{\Gamma \vdash x : \alpha}\ [\text{var}] \qquad \frac{c : \alpha \in \Sigma}{\Gamma \vdash c : \alpha}\ [\text{const}]$$

$$\frac{\Gamma, x : \alpha \vdash M : \beta}{\Gamma \vdash \lambda x.\, M : \alpha \to \beta}\ [\text{abs}] \qquad \frac{\Gamma \vdash M : \alpha \to \beta \qquad \Gamma \vdash N : \alpha}{\Gamma \vdash M\, N : \beta}\ [\text{app}]$$

$$\frac{\Gamma \vdash M : \alpha}{\Gamma \vdash \eta\, M : \mathcal{F}_E(\alpha)}\ [\eta] \qquad \frac{\begin{array}{c}\Gamma \vdash M_p : \alpha \qquad \Gamma, x : \beta \vdash M_c : \mathcal{F}_E(\gamma) \\ \mathtt{op} : \alpha \rightarrowtail \beta \in E\end{array}}{\Gamma \vdash \mathtt{op}\, M_p\, (\lambda x.\, M_c) : \mathcal{F}_E(\gamma)}\ [\mathtt{op}]$$

$$\frac{\Gamma \vdash M : \mathcal{F}_\emptyset(\alpha)}{\Gamma \vdash \downarrow_{\circ} M : \alpha}\ [\downarrow_{\circ}] \qquad \frac{\begin{array}{c}E = \{\mathtt{op}_i : \alpha_i \rightarrowtail \beta_i\}_{i \in I} \uplus E_f \\ E' = E'' \uplus E_f \\ [\Gamma \vdash M_i : \alpha_i \to (\beta_i \to \mathcal{F}_{E'}(\delta)) \to \mathcal{F}_{E'}(\delta)]_{i \in I} \\ \Gamma \vdash M_\eta : \gamma \to \mathcal{F}_{E'}(\delta) \\ \Gamma \vdash N : \mathcal{F}_E(\gamma)\end{array}}{\Gamma \vdash (\!|\, (\mathtt{op}_i \colon M_i)_{i \in I},\ \eta \colon M_\eta \,|\!)\, N : \mathcal{F}_{E'}(\delta)}\ [(\!|\ |\!)]$$

$$\frac{\Gamma \vdash M : \alpha \to \mathcal{F}_E(\beta)}{\Gamma \vdash \mathcal{C}\, M : \mathcal{F}_E(\alpha \to \beta)}\ [\mathcal{C}]$$

Figure 1.1: The typing rules for $(\!|\, \lambda \,|\!)$.

to a value of the input type α and to a continuation. The continuation is a function of type $\beta \to \mathcal{F}_E(\gamma)$ that accepts a value of the output type β (the result of performing the operation) and chooses in return a computation that should be pursued next. The return type of our new computation will thus be the return type γ of the computation provided by the continuation. The continuation's computation and the new extended computation will also share the same effect signature E. This means that all uses of the operation op within the created computation have the same input and output types.[13]

There is a parallel between the [var] rule and the [op] rule. The [var] rule lets use a symbol x with type α provided $x : \alpha \in \Gamma$. The [op] rule lets use a symbol op with type $\alpha \to (\beta \to \mathcal{F}_E(\gamma)) \to \mathcal{F}_E(\gamma)$ provided $\mathtt{op} : \alpha \rightarrowtail \beta \in E$. The crucial difference is that contexts (Γ) are components of judgments whereas effect signatures (E) are components of types. The meaning of a variable is determined by inspecting the expression in which it occurs and finding the λ that binds it (this is known as *lexical* or *static binding*). On the other hand, the meaning of an operation in a computation is determined by evaluating the term in which the computation appears until the computation becomes the argument of a handler. This handler will then give meaning to the operation symbol by substituting it with a suitable interpretation (this kind of *late* binding is known as *dynamic binding*).

We have now seen how to construct pure computations using η and extend them by adding operations. However, before we go on and start talking about handlers, we would like to give the algebraic intuition behind [op], as the algebraic point of view makes explaining the handler rule [$(\!|\ |\!)$] easier.

We can see the effect signature as an algebraic signature. For every $\mathtt{op} : \alpha \rightarrowtail \beta \in E$, we have an α-indexed family of operators of arity β. Let's unpack this statement.

- First, there is the matter of having an indexed family of operators. A common example of these is the case of scalar multiplication in the algebra of a vector space. A *single-sorted algebraic signature* is a set of operation symbols, each of which is given an arity (a natural number). For vector addition, the arity is 2, since vector addition acts on two vectors (two elements of the domain). Scalar multiplication acts on one scalar and one vector. However, neither arity 1 nor arity 2 adequately express this. We can get around the limitations of a single-sorted signature by introducing for every scalar k an operation of arity 1 that corresponds to multiplying the vector by k. Scalar multiplication is

[13] In general, the same operation symbol can be used with different input and output types, in computations whose types are indexed by different effect signatures.

therefore not a single operator but a scalar-indexed family of operators.

The very same strategy is applied here as well. A single operation symbol doesn't need to map to a single operator but can instead map to (possibly infinitely) many operators indexed by values of some type α. For example, writing messages to the program's output ($\texttt{print} : \mathit{string} \rightarrowtail 1$) can be seen as a string-indexed family of unary operators on computations. For every string s, we get an operator that maps computations c to computations that first print s and then continue as c.

- Next, we were speaking about operators of arity β. The use of a type in place of a numerical arity is due to a certain generalization. In set theory, natural numbers become sets that have the same cardinality as the number they represent ($|N| = N$). We can therefore conservatively generalize the idea of arity to a set by saying that an operator of arity X takes one operand per each element of the set X. It's a short step from there to using types as arities, wherein an operator of arity β takes one operand per possible value of type β.

  This will come in very handy in our system. We want our operator $\texttt{op}$ to have as many operands as there are possible values in the output type β. Therefore, we simply say that the operator has arity β.

  How do we write down the application of an operator of arity β to its operands? We can no longer just list out all the operands, since types in $(\!| \lambda |\!)$ may have an unbounded number of inhabitants. We will organize operands in *operand clusters*,[14] arity-indexed families of operands. We will write them down as functions, using λ-abstraction, from the arity type β to some operand type, e.g., $\mathcal{F}_E(\gamma)$.

Now we can understand what it means to say that $\texttt{op} : \alpha \rightarrowtail \beta \in E$ gives rise to an α-indexed family of operators of arity β. We apply to $\texttt{op}$ an index of type α to get an operator and then we apply that operator to an operand cluster of type $\beta \to \mathcal{F}_E(\gamma)$ to get a new expression of type $\mathcal{F}_E(\gamma)$.

We suggest visualizing these algebraic expressions as trees (see Section 2 of [86] for the original idea). Trees of type $\mathcal{F}_E(\alpha)$ consist of leafs containing values of type α and internal nodes labelled with operations and their parameters. Every internal node is labelled with some $\texttt{op} : \alpha \rightarrowtail \beta \in E$ and with a parameter of type α and it has a cluster of children indexed by β.

[$(\!| \;|\!)$]

Now we are ready to explain the handler rule. The typing rule for $(\!| \;|\!)$ is repeated in Figure 1.2.

To illustrate the constraints on the types of the components, M_i and M_η, of a handler, we will examine its semantics. The handler processes the algebraic expression N by recursive induction. Depending on the shape of the expression, one of the following will happen:

- If $N = \eta\, N'$, then N' is of type γ. This is where the M_η function comes in. It must take a value of type γ and produce a new tree of type $\mathcal{F}_{E'}(\delta)$, hence the fourth hypothesis of the $(\!| \;|\!)$ rule.

- If $N = \texttt{op}_i\, N_\mathrm{p}\, (\lambda x.\, N_\mathrm{c})$ for some $i \in I$, then N_p must be of type α_i. Furthermore, for every $x : \beta_i$, we have an operand $N_\mathrm{c} : \mathcal{F}_E(\gamma)$. We know this since N is of type $\mathcal{F}_E(\gamma)$ and the first hypothesis tells us that $\texttt{op}_i : \alpha_i \rightarrowtail \beta_i \in E$.

  We will recursively apply our handler to the cluster of operands, changing their type from $\mathcal{F}_E(\gamma)$ to $\mathcal{F}_{E'}(\delta)$. We now need something which takes N_p, whose type is α_i, and the cluster of processed operands, type $\beta_i \to \mathcal{F}_{E'}(\delta)$, which is exactly the function M_i in the third hypothesis of the $(\!| \;|\!)$ rule.

- If $N = \texttt{op}\, N_\mathrm{p}\, (\lambda x.\, N_\mathrm{c})$ and $\texttt{op} : \alpha \rightarrowtail \beta \in E_f$ for some α and β, then we will ignore the node and process only its children. This means that the resulting expression will contain the operation symbol $\texttt{op}$ from E_f[15]. In order for such an expression to be of the desired type $\mathcal{F}_{E'}(\delta)$, E_f must be included in E', which is what the second hypothesis of the $(\!| \;|\!)$ rule guarantees.

[14] Our use of the word *cluster* is synonymous with the mathematical term *family*. We will be using the term *cluster* for families of computations passed to operations and handlers.

[15] The f in E_f stands for *forwarded effects*, since it refers to effects that the handler will not interpret but instead forward to some other interpreter. The notation comes from a similar rule in the λ_{eff} calculus [63].

$$\frac{\begin{array}{c} E = \{\mathtt{op}_i : \alpha_i \rightarrowtail \beta_i\}_{i\in I} \uplus E_f \\ E' = E'' \uplus E_f \\ [\Gamma \vdash M_i : \alpha_i \to (\beta_i \to \mathcal{F}_{E'}(\delta)) \to \mathcal{F}_{E'}(\delta)]_{i\in I} \\ \Gamma \vdash M_\eta : \gamma \to \mathcal{F}_{E'}(\delta) \\ \Gamma \vdash N : \mathcal{F}_E(\gamma) \end{array}}{\Gamma \vdash (\!|\, (\mathtt{op}_i\colon M_i)_{i\in I},\ \eta\colon M_\eta \,|\!)\, N : \mathcal{F}_{E'}(\delta)}\ [(\!|\ |\!)]$$

Figure 1.2: The typing rule for the handler construction.

We have covered the whole $(\!|\ |\!)$ rule, except for the presence of the effect signature E''. It serves two roles.

- First of all, it acts as a "free" variable over effect signatures. This means that we can give any effect signature E' to the type $\mathcal{F}_{E'}(\delta)$ of the resulting computation N' as long as E' contains E_f (E'' represents the relative complement of E_f in E'). This is in analogy to the free effect variable E in the $[\eta]$ and $[\mathtt{op}]$ rules. This freedom of effect variables is a way of implementing the idea that a computation of type $\mathcal{F}_{E_1}(\alpha)$ can be used anywhere that a computation of type $\mathcal{F}_{E_2}(\alpha)$ is needed given that $E_1 \subseteq E_2$.

- In the previous paragraph, why did we put the word "free" in quotation marks? Because the effect variable E'' is not actually free. It is the complement of E_f in E' and E' is constrained by the types of M_i and M_η in the third and fourth hypotheses, respectively. The handler's clauses might themselves introduce new effects, which will in turn translate into constraints on E' and E''. This happens when a handler interprets an operation by making an appeal to some other operation (e.g. a handler could interpret computations using n-ary choice into computations using binary choice).

  As the simplest example, we can take a handler that replaces one operation symbol with another, $(\!|\, \mathtt{old}\colon (\lambda pc.\, \mathtt{new}\, p\, (\lambda y.\, c\, y)),\ \eta\colon (\lambda x.\, \eta\, x) \,|\!)$. The type scheme corresponding to the term is $\mathcal{F}_{\{\mathtt{old}:\alpha\rightarrowtail\beta\}\uplus E_f}(\gamma) \to \mathcal{F}_{\{\mathtt{new}:\alpha\rightarrowtail\beta\}\uplus E^+\uplus E_f}(\gamma)$. In this scheme, α, β and γ are free meta-variables ranging over types and E_f and E^+ range over effect signatures. The E'' of the $(\!|\ |\!)$ rule corresponds to $\{\mathtt{new} : \alpha \rightarrowtail \beta\} \uplus E^+$ (i.e. E'' is not free, it must contain `new`). The handler has eliminated the `old` effect but it has also introduced the `new` effect.

This concludes our exploration of the $(\!|\ |\!)$ rule. We have explained it in terms of algebraic expressions and trees, using the denotational intution. We will develop the operational intuition, which talks about handlers in terms of computations and continuations, in Section 1.4, where we will give the semantics of our language using reduction rules.

[cherry]

Next up is the cherry operator, $\downarrow$. Its type is $\mathcal{F}_\emptyset(\alpha) \to \alpha$ and it serves as a kind of dual to the η opertator, an elimination for the $\mathcal{F}$ type.

The type $\mathcal{F}_\emptyset(\alpha)$ demands that the effect signature be empty. In such a case, the tree has no internal nodes and is composed of just a leaf containing a value of the type α. The $\downarrow$ operator serves to extract that value.

Another way to look at it is to say that a computation of type $\mathcal{F}_\emptyset(\alpha)$ cannot perform any "unsafe" operations and it is therefore always safe to execute it and get the resulting value of type α.

[$\mathcal{C}$]

Finally, we take a look at the $\mathcal{C}$ operator. The type of the operator is $(\alpha \to \mathcal{F}_E(\beta)) \to \mathcal{F}_E(\alpha \to \beta)$. Its input is an α-indexed family of computations and its output is a computation of α-indexed families. The operator applies only in the case when all the computations in the family share the same *internal structure*. By sharing the same internal structure, we mean that the trees can only differ in their leaves. What the

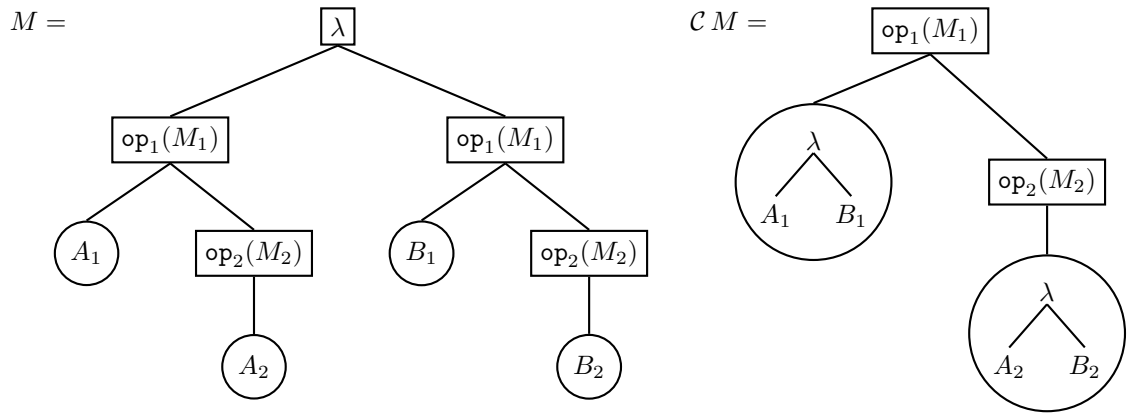


Figure 1.3: Example of applying the $\mathcal{C}$ operator to a term.

$\mathcal{C}$ operator then does is to push the λ-binder down this common internal structure into the leaves. This way, we can evaluate/handle the common operations without commiting to a specific value of type α.

The action of the $\mathcal{C}$ operator is illustrated in Figure 1.3. Here, M is a function of some two-value type. It maps one value to the left subtree of λ (with leaves A_i) and the other value to the right subtree (leaves B_i). Both subtrees correspond to computations, in which a box is an operation and a circle is an atomic expression (i.e. a return value). Furthermore, $\mathtt{op}_1$ has a two-value output type (arity 2) and $\mathtt{op}_2$ has a one-value output type (arity 1).

Since the operations $\mathtt{op}_1$ and $\mathtt{op}_2$ and their arguments M_1 and M_2 are the same in both subtrees, and thus independent of the value passed to the λ, we can apply the $\mathcal{C}$ operator. The $\mathcal{C}$ pulls this common structure out of the λ and gives us a computation that produces a function.

We can also explain the action of $\mathcal{C}$ in operational terms. As in call-by-push-value [82], we can think of abstraction over α as some effectful operation that tries to pop a value $x : \alpha$ off a stack. The input of $\mathcal{C}$ can then be seen as a continuation waiting for this x and wanting to perform some further operations. $\mathcal{C}$ assumes that the continuation performs operations independently of x[16] and it can thus postpone popping x off the stack until after the operations dictated by the continuation have been evaluated. $\mathcal{C}$ is therefore a kind of commutativity law for operations and abstractions (the popping of a value off the operand stack): as long as one does not depend on the other, it does not matter whether we first perform an operation and then abstract over an argument or whether we do so the other way around.[17]

1.4 Reduction Rules

We will now finally give a semantics to $(\!| \lambda |\!)$. The semantics will be given in the form of a reduction relation on terms. Even though the point of the calculus is to talk about effects, the reduction semantics will have no notion of order of evaluation; any reducible subexpression can be reduced in any context.

Before we dive into the reduction rules proper, we will first have to handle some formal paperwork, most of it due to the fact that we use variables and binders in our calculus. In order to quotient out the irrelevant distinction between terms that are the same up to variable names, we will introduce a series of definitions leading up to a notion of α-equivalence.

Definition 1.4.1. *Let M be a term in $(\!| \lambda |\!)$. We define the set of **free variables** of M, written* $\mathrm{FV}(M)$*, using the following set of equations:*

[16]Violating this assumption will yield terms which get stuck during evaluation (we will see the partial reduction rules in Section 1.4). Sam Lindley presented a refined type system for a similar calculus to track the use of variables [86]. A similar refinement should be possible in our case as well but it would obscure the already dense type notation.

[17]The other direction, typed $\mathcal{F}_E(\alpha \to \beta) \to (\alpha \to \mathcal{F}_E(\beta))$, is already possible without introducing a special operator since $\mathcal{F}_E$ is a functor (as we will see in Subsection 3.3.4).

$$
\begin{aligned}
\mathrm{FV}(\lambda x.\, M) &= \mathrm{FV}(M) \setminus \{x\} \\
\mathrm{FV}(M\, N) &= \mathrm{FV}(M) \cup \mathrm{FV}(N) \\
\mathrm{FV}(x) &= \{x\} \\
\mathrm{FV}(c) &= \emptyset \\
\mathrm{FV}(\mathtt{op}\, M_{\mathrm{p}}\, (\lambda x.\, M_{\mathrm{c}})) &= \mathrm{FV}(M_{\mathrm{p}}) \cup (\mathrm{FV}(M_{\mathrm{c}}) \setminus \{x\}) \\
\mathrm{FV}(\eta\, M) &= \mathrm{FV}(M) \\
\mathrm{FV}(\llparenthesis\, (\mathtt{op}_i \colon M_i)_{i \in I},\ \eta \colon M_\eta \,\rrparenthesis\, N) &= \bigcup_{i \in I} \mathrm{FV}(M_i) \cup \mathrm{FV}(M_\eta) \cup \mathrm{FV}(N) \\
\mathrm{FV}(\downarrow M) &= \mathrm{FV}(M) \\
\mathrm{FV}(\mathcal{C}\, M) &= \mathrm{FV}(M)
\end{aligned}
$$

Most often, we will make use of FV indirectly, using the following notion.

Definition 1.4.2. *We say that x is **fresh** for M iff $x \notin \mathrm{FV}(M)$.*

Definition 1.4.3. *Let M and N be terms and x a variable. We define the **capture-avoiding substitution** of N for x in M, written as $M[x \coloneqq N]$, using the following equations:*

$$
\begin{aligned}
(\lambda x.\, M)[x \coloneqq N] &= \lambda x.\, M \\
(\lambda y.\, M)[x \coloneqq N] &= \lambda y.\, (M[x \coloneqq N]) \textit{ given that } y \neq x \textit{ and } y \textit{ is fresh for } N \\
(M\, K)[x \coloneqq N] &= (M[x \coloneqq N])\, (K[x \coloneqq N]) \\
x[x \coloneqq N] &= N \\
y[x \coloneqq N] &= y \textit{ given that } x \neq y \\
c[x \coloneqq N] &= c \\
(\mathtt{op}\, M_{\mathrm{p}}\, (\lambda x.\, M_{\mathrm{c}}))[x \coloneqq N] &= \mathtt{op}\, (M_{\mathrm{p}}[x \coloneqq N])\, (\lambda x.\, M_{\mathrm{c}}) \\
(\mathtt{op}\, M_{\mathrm{p}}\, (\lambda y.\, M_{\mathrm{c}}))[x \coloneqq N] &= \mathtt{op}\, (M_{\mathrm{p}}[x \coloneqq N])\, (\lambda y.\, M_{\mathrm{c}}[x \coloneqq N]) \textit{ given that } y \neq x \textit{ and } y \textit{ is fresh for } N \\
(\eta\, M)[x \coloneqq N] &= \eta\, (M[x \coloneqq N]) \\
(\llparenthesis\, (\mathtt{op}_i \colon M_i)_{i \in I},\ \eta \colon M_\eta \,\rrparenthesis\, N')[x \coloneqq N] &= \llparenthesis\, (\mathtt{op}_i \colon (M_i[x \coloneqq N]))_{i \in I},\ \eta \colon (M_\eta[x \coloneqq N]) \,\rrparenthesis\, (N'[x \coloneqq N]) \\
(\downarrow M)[x \coloneqq N] &= \downarrow (M[x \coloneqq N]) \\
(\mathcal{C}\, M)[x \coloneqq N] &= \mathcal{C}\, (M[x \coloneqq N])
\end{aligned}
$$

Note that it is possible for $M[x \coloneqq N]$ to not be defined by the equations above (e.g. $(\lambda y.\, x)[x \coloneqq y]$). In such cases, we say that the substitution does not exist.

Definition 1.4.4. ***Evaluation contexts** are terms with a hole (written as $[]$) inside. They are formally defined by the following grammar.*

$$
\begin{aligned}
C ::= {} & [] \\
& \mid \lambda x.\, C \\
& \mid C\, N \\
& \mid M\, C \\
& \mid \mathtt{op}\, C\, (\lambda x.\, M_{\mathrm{c}}) \\
& \mid \mathtt{op}\, M_{\mathrm{p}}\, (\lambda x.\, C) \\
& \mid \eta\, C \\
& \mid (\!|\, \mathtt{op}_1 \colon M_1,\ \ldots,\ \mathtt{op}_n \colon M_n,\ \eta \colon M_\eta \,|\!)\, C \\
& \mid (\!|\, \mathtt{op}_1 \colon M_1,\ \ldots,\ \mathtt{op}_i \colon C,\ \ldots,\ \mathtt{op}_n \colon M_n,\ \eta \colon M_\eta \,|\!)\, N \\
& \mid (\!|\, \mathtt{op}_1 \colon M_1,\ \ldots,\ \mathtt{op}_i \colon M_i,\ \ldots,\ \mathtt{op}_n \colon M_n,\ \eta \colon C \,|\!)\, N \\
& \mid \downarrow_{\circ} C \\
& \mid \mathcal{C}\, C
\end{aligned}
$$

Definition 1.4.5. *N is a **subterm** of M if there exists an evaluation context C such that $M = C[N]$.*

Definition 1.4.6. *Let $\sim$ be a binary relation on the terms of $(\!|\, \lambda \,|\!)$. We define the relation $[\sim]$, called the **context closure** of $\sim$, as the smallest relation that contains $\sim$ and satisfies the following closure property for any evaluation context C:*

- *if $M\ [\sim]\ M'$, then $C[M]\ [\sim]\ C[M']$*

We will be defining relations on the terms of $(\!|\, \lambda \,|\!)$ that will correspond to different transformations (such as **swap** and the reduction rules). The notion of context closure will allow us to say that a term can be transformed by transforming any of its parts.

Definition 1.4.7. *Let $\sim$ be a relation. We define the relation $\sim^*$, called the **reflexive-transitive closure** of $\sim$, as the smallest relation that contains $\sim$ and satisfies the following conditions:*

- *$x \sim^* x$ for any x in the domain of R*
- *if $x \sim^* y$ and $y \sim^* z$, then $x \sim^* z$*

Definition 1.4.8. *We define a relation called **swap** on the terms of $(\!|\, \lambda \,|\!)$ as:*

- *$\lambda x.\, M$ **swap** $\lambda y.\, (M[x := y])$ given that $y \notin \mathrm{FV}(M)$ and that $M[x := y]$ exists*
- *$\mathtt{op}\, M_{\mathrm{p}}\, (\lambda x.\, M_{\mathrm{c}})$ **swap** $\mathtt{op}\, M_{\mathrm{p}}\, (\lambda y.\, M_{\mathrm{c}}[x := y])$ given that $y \notin \mathrm{FV}(M_{\mathrm{c}})$ and that $M_{\mathrm{c}}[x := y]$ exists*

Definition 1.4.9. *We define the relation of α-**equivalence**, written as $=_\alpha$, to be $[\textbf{swap}]^*$, i.e. the reflexive-transitive closure of the context closure of the **swap** relation.*

Observation 1.4.10. *$=_\alpha$ is an equivalence relation.*

Proof. **swap** is a symmetric relation and both context closure and reflexive-transitive closure preserve symmetricity. Reflexivity and transitivity are guaranteed by the reflexive-transitive closure. □

The notion of α-equivalence will be very useful. We will use it so much that we will actually quotient the set of terms and start talking about α-equivalence classes instead of specific terms. We will keep using the same notation and so from now on, terms written in the syntax of our calculus no longer designate individual terms but rather α-equivalence classes of terms. The term "term" will itself be freely used to actually mean an α-equivalence class of terms.

However, before we start using individual terms while having them represent entire α-equivalence classes, we should better make sure that the operations that we will perform with these terms are congruent with α-equivalence.[18] Is it the case that the operations we have introduced in this section are congruent with α-equivalence?

FV is congruent: swapping names bound by λ does not change the set of free variables in a term. What about substitution? As we have already seen, capture-avoiding substitution is a partial operation: $(\lambda y.\, x)[x \coloneqq y]$ does not exist. However, $\lambda z.\, x$ is α-equivalent to $\lambda y.\, x$ and $(\lambda z.\, x)[x \coloneqq y]$ exists; it is $\lambda z.\, y$. Therefore, capture-avoiding substitution is not α-congruent.

We will introduce an alternative notion, *capture-resolving substitution*. While capture-avoiding substitution was a partial function on terms, capture-resolving substitution will be a total function on α-equivalence classes of terms. Furthermore, the new function will be an extension of the old one. If capture-avoiding substitution would have mapped A to B, then capture-resolving substitution will map the $=_\alpha$-class of A to the $=_\alpha$-class of B.

In defining capture-resolving substitution, we will demonstrate a technique that is made possible by the transition from single terms to $=_\alpha$ classes of terms. When we have a variable x bound in a term, we can always assume, without loss of generality, that it is distinct from another variable y (since inside our term, we can swap x with any variable from $\mathcal{X}$ which is different from y and end up in the same $=_\alpha$-class). By extension, we can also assume that a variable x bound in some term is fresh in some other term.

Definition 1.4.11. *Let M and N be terms and x a variable. We define the* ***(capture-resolving) substitution*** *of N for x in M, written as $M[x \coloneqq N]$,*[19] *using the following equations:*

$$
\begin{aligned}
(\lambda y.\, M)[x \coloneqq N] &= \lambda y.\, (M[x \coloneqq N]) \textit{ assuming that } y \neq x \textit{ and } y \textit{ is fresh for } N^{20} \\
(M\, K)[x \coloneqq N] &= (M[x \coloneqq N])\, (K[x \coloneqq N]) \\
x[x \coloneqq N] &= N \\
y[x \coloneqq N] &= y \textit{ given that } x \neq y^{21} \\
c[x \coloneqq N] &= c \\
(\mathtt{op}\, M_\mathsf{p}\, (\lambda y.\, M_\mathsf{c}))[x \coloneqq N] &= \mathtt{op}\, (M_\mathsf{p}[x \coloneqq N])\, (\lambda y.\, M_\mathsf{c}[x \coloneqq N]) \textit{ assuming that } y \neq x \textit{ and } y \textit{ is fresh for } N \\
(\eta\, M)[x \coloneqq N] &= \eta\, (M[x \coloneqq N]) \\
(\llparenthesis\, (\mathtt{op}_i \colon M_i)_{i \in I},\ \eta \colon M_\eta \,\rrparenthesis\, N')[x \coloneqq N] &= \llparenthesis\, (\mathtt{op}_i \colon (M_i[x \coloneqq N]))_{i \in I},\ \eta \colon (M_\eta[x \coloneqq N]) \,\rrparenthesis\, (N'[x \coloneqq N]) \\
(\downarrow_{\circ} M)[x \coloneqq N] &= \downarrow_{\circ} (M[x \coloneqq N]) \\
(\mathcal{C}\, M)[x \coloneqq N] &= \mathcal{C}\, (M[x \coloneqq N])
\end{aligned}
$$

We are now at a point where we can easily lay down the reduction rules for $\llparenthesis\, \lambda \,\rrparenthesis$. A reduction rule ξ will be a relation on terms. Most of the time, we will deal with their context closures, $[\xi]$, for which we will also adopt the notation $\to_\xi$. We will also use the notation $\to_{\xi_1,\ldots,\xi_n}$ for the composition $\to_{\xi_n} \circ \ldots \circ \to_{\xi_1}$. The *reduction relation* $\to$ of $\llparenthesis\, \lambda \,\rrparenthesis$ is the union of the $\to_\xi$ relations for every reduction rule ξ. We will also use the symbols $\twoheadrightarrow$ and $\twoheadleftrightarrow$ to stand for the reflexive-transitive and reflexive-symmetric-transitive closures of $\to$, respectively. If $M \twoheadleftrightarrow N$, we will also say that M and N are *convertible*. A term which is not reducible to some other term is said to be in *normal form*.

We will now go through the reduction rules of $\llparenthesis\, \lambda \,\rrparenthesis$, presented in Figure 1.4, one by one.

First off, we have the β and η rules. By no coincidence, they are the same rules as the ones found in STLC.

Next we have the three rules that govern the behavior of handlers. We recognize the three different rules as the three different cases in the informal denotational semantics given in Section 1.3.

[18] Congruence means that equivalent inputs are mapped to equivalent outputs.
[19] From now on, this notation will be used for capture-resolving substitution only.
[20] Here, y is a bound variable and we can simply *assume* that it is different from x and proceed…
[21] …whereas here, y is a free variable and therefore, we have to *examine* whether it is different from x or not.

$$
\begin{array}{ll}
(\lambda x.\, M)\, N \to & \text{rule } \beta \\
M[x \coloneqq N] & \\
\\
\lambda x.\, M\, x \to & \text{rule } \eta \\
M & \text{where } x \notin \mathrm{FV}(M) \\
\\
(\!|\, (\mathtt{op}_i \colon M_i)_{i \in I},\ \eta \colon M_\eta \,|\!)\, (\eta\, N) \to & \text{rule } (\!|\, \eta \,|\!) \\
M_\eta\, N & \\
\\
(\!|\, (\mathtt{op}_i \colon M_i)_{i \in I},\ \eta \colon M_\eta \,|\!)\, (\mathtt{op}_j\, N_\mathrm{p}\, (\lambda x.\, N_\mathrm{c})) \to & \text{rule } (\!|\, \mathtt{op} \,|\!) \\
M_j\, N_\mathrm{p}\, (\lambda x.\, (\!|\, (\mathtt{op}_i \colon M_i)_{i \in I},\ \eta \colon M_\eta \,|\!)\, N_\mathrm{c}) & \text{where } j \in I \\
 & \text{and } x \notin \mathrm{FV}((M_i)_{i \in I}, M_\eta) \\
\\
(\!|\, (\mathtt{op}_i \colon M_i)_{i \in I},\ \eta \colon M_\eta \,|\!)\, (\mathtt{op}_j\, N_\mathrm{p}\, (\lambda x.\, N_\mathrm{c})) \to & \text{rule } (\!|\, \mathtt{op}' \,|\!) \\
\mathtt{op}_j\, N_\mathrm{p}\, (\lambda x.\, (\!|\, (\mathtt{op}_i \colon M_i)_{i \in I},\ \eta \colon M_\eta \,|\!)\, N_\mathrm{c}) & \text{where } j \notin I \\
 & \text{and } x \notin \mathrm{FV}((M_i)_{i \in I}, M_\eta) \\
\\
\mathrel{\underset{\circ}{|}}\, (\eta\, M) \to & \text{rule } \mathrel{\underset{\circ}{|}} \\
M & \\
\\
\mathcal{C}\, (\lambda x.\, \eta\, M) \to & \text{rule } \mathcal{C}_\eta \\
\eta\, (\lambda x.\, M) & \\
\\
\mathcal{C}\, (\lambda x.\, \mathtt{op}\, M_\mathrm{p}\, (\lambda y.\, M_\mathrm{c})) \to & \text{rule } \mathcal{C}_{\mathtt{op}} \\
\mathtt{op}\, M_\mathrm{p}\, (\lambda y.\, \mathcal{C}\, (\lambda x.\, M_\mathrm{c})) & \text{where } x \notin \mathrm{FV}(M_\mathrm{p})
\end{array}
$$

Figure 1.4: The reduction rules of $(\!|\, \lambda \,|\!)$.

- When the expression is just an atom (i.e. $\eta\, N$), rule $(\!|\eta|\!)$ applies the clause M_η to the value N contained within.
- When the expression is an operation $\mathtt{op}_j\, N_\mathrm{p}\, (\lambda x.\, N_\mathrm{c}(x))$ with $j \in I$, we first recursively apply the handler to every child $N_\mathrm{c}(x)$. We then pass the parameter N_p stored in the node along with the cluster of the processed children to the clause M_j.
- When the expression is an operation $\mathtt{op}_j\, N_\mathrm{p}\, (\lambda x.\, N_\mathrm{c}(x))$ but where $j \notin I$, we leave the node as it is and just recurse down on to the subexpressions (effectively using $\mathtt{op}_j$ as the handler clause for $\mathtt{op}_j$).

By looking at these rules, we also notice that all they do is just traverse the continuations (N_c) and replace η with M_η and $\mathtt{op}_i$ with M_i. This justifies thinking of η and the operation symbols as special variables which are bound to a value when being passed through a handler. This substitutability is already hinted at by the types of M_η and M_i in the $(\!|\ |\!)$ typing rule and their correspondence with the typing rules [η] and [op], respectively.

The next rule talks about the cherry operator. It does what we would expect it to do.[22] It expects its argument to always be an atomic algebraic expression, a pure computation, and it extracts the argument that was passed to the η constructor.

Finally, we have the two rules defining the behavior of $\mathcal{C}$. We remind ourselves that the goal of $\mathcal{C}$ is to make computations and abstractions commute by pushing λ below operation symbols and η.

- Rule $\mathcal{C}_\eta$ treats the base case where the computation that we try to push λ through is a pure computation. In that case, we just reorder the λ binder and the η operator.
- Rule $\mathcal{C}_\mathtt{op}$ deals with the case of the λ meeting an operation symbol. The solution is to push the $\mathcal{C}$ operator down through the continuation. The operation $\mathcal{C}\,(\lambda x.\ \ldots)$ is applied recursively to every child $M_\mathrm{c}\,(y)$. However, this strategy is sound only when M_p has no free occurrence of x (which would have been bound by the λ in the redex but would become unbound in the contractum). We therefore have a constraint saying that x must not occur free in M_p. Unlike the other freshness constraints, this one cannot be fixed by a simple renaming of variables. If this constraint is not met, the $\mathcal{C}$ will not be able to reduce.

  When talking about the $\mathcal{C}$ operator in Section 1.3, we talked about how it applies only to families of computations that share the same internal structure (i.e. functions of x where the internal structure does not depend on x). This is reflected in the reduction rules in two ways:

  - Firstly, in order for $\mathcal{C}_\mathtt{op}$ to kick in, the body of the function must have already reduced to something of the form $\mathtt{op}\, M_\mathrm{p}\, M_\mathrm{c}$. This means that the next operation to be performed has already been determined to be op without needing to wait for the value of x.
  - Secondly, the reduction can only proceed if M_p does not contain a free occurrence of x. This means that M_p is independent of x.

1.5 Sums and Products

In the examples throughout this manuscript, we will assume that our calculus has facilities for dealing with sum types and product types (variants and pairs). In this section, we give a brief formal definition of the standard components that will provide these facilities in our calculus.

1.5.1 New Terms

First, we add new expressions into our language:

pair $\langle M, N\rangle$ where M and N are expressions

[22] Based on what we said about it in Section 1.3, not on its name.

unit $\star$

first projection $\pi_1\, M$ where M is an expression

second projection $\pi_2\, M$ where M is an expression

left injection inl M where M is an expression

right injection inr M where M is an expression

case analysis **case** M **of** $\{\text{inl } x \to N_{\text{l}};\ \text{inr } y \to N_{\text{r}}\}$ where M, N_l and N_r are expressions and x and y are variables from $\mathcal{X}$

absurdity **case** M **of** $\{\,\}$ where M is an expression

The first four concern products. We can construct a pair and then we can project out both of its components. We can also create values of the unit type. The other four implement sums. We can construct two disjoint variants and then we can do case analysis on the results. We can also treat the empty type by handling its zero possible cases.

1.5.2 New Types

We also add new types and type operators:

product $\alpha \times \beta$ where α and β are types

unit 1

sum $\alpha + \beta$ where α and β are types

empty 0

The unit type serves as a unit to the product operator insofar as $\alpha \times 1$, $1 \times \alpha$ and α are isomorphic. It will be used as the input type of operations that do not need to be parameterized and as the output type of operations that do not return any interesting value.

Analogously, the empty type serves as a unit to the sum type since $\alpha + 0$, $0 + \alpha$ and α are isomorphic. The empty type is useful as the output type of operations that never return (such as `fail` $: 1 \rightarrowtail 0$, which terminates a computation, signalling failure).

The standard typing rules of the constructions concerning sums and products are given in Figure 1.5. In the typing rules, we can witness the duality of $\times$ and $+$. There is one introduction rule and two elimination rules for pairs, whereas we have two introduction rules and one elimination rule for variants. We also have an introduction rule for the unit type and an elimination for the empty type.

1.5.3 New Reduction Rules

Now that we know how to correctly write terms with sums and products after having introduced the syntax and the typing rules, we will look at how to simplify and evaluate such terms. We will extend the set of reduction rules in $(\!|\lambda|\!)$ with the ones in Figure 1.6. As before, we take the context closure of these rules ξ to get relations $\to_\xi$ and then we include these relations into our reduction relations $\to$ and $\twoheadrightarrow$.

1.5.4 Booleans

The most simple use of a sum type is to serve as a binary Boolean type. Since this comes very handy in examples, we will introduce the typical syntax one expects with Booleans.

We first define the Boolean type 2 to be a type whose value might be either the true element or the false element:

$$2 = 1 + 1$$

$$\frac{\Gamma \vdash M : \alpha \qquad \Gamma \vdash N : \beta}{\Gamma \vdash \langle M, N \rangle : \alpha \times \beta}\ [\times] \qquad \Gamma \vdash \star : 1\ \ [\star]$$

$$\frac{\Gamma \vdash M : \alpha \times \beta}{\Gamma \vdash \pi_1\, M : \alpha}\ [\pi_1] \qquad \frac{\Gamma \vdash M : \alpha \times \beta}{\Gamma \vdash \pi_2\, M : \beta}\ [\pi_2]$$

$$\frac{\Gamma \vdash M : \alpha}{\Gamma \vdash \text{inl}\ M : \alpha + \beta}\ [\text{inl}] \qquad \frac{\Gamma \vdash M : \beta}{\Gamma \vdash \text{inr}\ M : \alpha + \beta}\ [\text{inr}]$$

$$\frac{\Gamma \vdash M : \alpha + \beta \qquad \Gamma, x : \alpha \vdash N_l : \gamma \qquad \Gamma, y : \beta \vdash N_r : \gamma}{\Gamma \vdash \textbf{case}\ M\ \textbf{of}\ \{\text{inl}\ x \to N_\text{l};\ \text{inr}\ y \to N_\text{r}\} : \gamma}\ [\textbf{case}]$$

$$\frac{\Gamma \vdash M : 0}{\Gamma \vdash \textbf{case}\ M\ \textbf{of}\ \{\,\} : \alpha}\ [\text{empty}]$$

Figure 1.5: The typing rules for sums and products.

$\pi_1\ \langle M, N \rangle \to$ M	rule $\beta.\times_1$
$\pi_2\ \langle M, N \rangle \to$ N	rule $\beta.\times_2$
$\textbf{case}\ (\text{inl}\ M)\ \textbf{of}\ \{\text{inl}\ x \to N_\text{l};\ \text{inr}\ y \to N_\text{r}\}$ $N_\text{l}[x := M]$	rule $\beta.+_1$
$\textbf{case}\ (\text{inr}\ M)\ \textbf{of}\ \{\text{inl}\ x \to N_\text{l};\ \text{inr}\ y \to N_\text{r}\}$ $N_\text{r}[y := M]$	rule $\beta.+_2$

Figure 1.6: The reduction rules for sums and products.

True and False are then constructed by injecting $\star$ into 2 using the two different injections:

$$\mathbf{T} = \text{inl}\ \star$$
$$\mathbf{F} = \text{inr}\ \star$$

Finally, we can also simplify the case analysis expression into an if-then-else expression:

$$\textbf{if}\ M\ \textbf{then}\ N_{\mathbf{T}}\ \textbf{else}\ N_{\mathbf{F}} = \textbf{case}\ M\ \textbf{of}\ \{\text{inl}\ x \to N_{\mathbf{T}};\ \text{inr}\ y \to N_{\mathbf{F}}\}$$

where x is fresh for $N_{\mathbf{T}}$ and y is fresh for $N_{\mathbf{F}}$.
This lets us derive the two reduction rules which define the semantics of if-then-else expressions:

$$\textbf{if T then}\ N_{\mathbf{T}}\ \textbf{else}\ N_{\mathbf{F}} \to_{\textbf{if.T}} N_{\mathbf{T}}$$
$$\textbf{if F then}\ N_{\mathbf{T}}\ \textbf{else}\ N_{\mathbf{F}} \to_{\textbf{if.F}} N_{\mathbf{F}}$$

because:

$$\begin{aligned}\textbf{if T then}\ N_{\mathbf{T}}\ \textbf{else}\ N_{\mathbf{F}} &= \textbf{case}\ (\text{inl}\ \star)\ \textbf{of}\ \{\text{inl}\ x \to N_{\mathbf{T}};\ \text{inr}\ y \to N_{\mathbf{F}}\}\\ &\to_{\beta.+_1} N_{\mathbf{T}}\\ \textbf{if F then}\ N_{\mathbf{T}}\ \textbf{else}\ N_{\mathbf{F}} &= \textbf{case}\ (\text{inr}\ \star)\ \textbf{of}\ \{\text{inl}\ x \to N_{\mathbf{T}};\ \text{inr}\ y \to N_{\mathbf{F}}\}\\ &\to_{\beta.+_2} N_{\mathbf{F}}\end{aligned}$$

1.6 Common Combinators

Here we will introduce a collection of useful syntactic shortcuts and combinators for our calculus.

1.6.1 Composing Functions and Computations

First of all, to save some space and write functions in a terse "point-free" style, we introduce the composition operator (known as the **B** combinator in combinatory logic).

$$\begin{aligned}&\_ \circ \_ : (\beta \to \gamma) \to (\alpha \to \beta) \to (\alpha \to \gamma)\\ &f \circ g = \lambda x.\, f\,(g\,x)\end{aligned}$$

We will also "functionalize" our term constructors (i.e. we will write η as a shortcut for $(\lambda x.\, \eta\, x)$). Our motive in not defining our symbols directly as function constants in the core calculus is due to the proofs of confluence and termination. A complete list of functionalized symbols is given below.[23]

[23]The type and effect signature metavariables that appear in the types are bound by the typing constraints of the terms on the right-hand side.

$$\eta : \alpha \to \mathcal{F}_E(\alpha)$$
$$\eta = \lambda x.\, \eta\, x$$
$$\mathtt{op} : \alpha \to (\beta \to \mathcal{F}_E(\gamma)) \to \mathcal{F}_E(\gamma)$$
$$\mathtt{op} = \lambda pc.\, \mathtt{op}\, p\, (\lambda x.\, c\, x)$$
$$(\!|\, (\mathtt{op}_i \colon M_i)_{i \in I},\ \eta \colon M_\eta \,|\!) : \mathcal{F}_E(\gamma) \to \mathcal{F}_{E'}(\delta)$$
$$(\!|\, (\mathtt{op}_i \colon M_i)_{i \in I},\ \eta \colon M_\eta \,|\!) = \lambda x.\, (\!|\, (\mathtt{op}_i \colon M_i)_{i \in I},\ \eta \colon M_\eta \,|\!)\, x$$
$$\downarrow : \mathcal{F}_\emptyset(\alpha) \to \alpha$$
$$\downarrow = \lambda x.\, \downarrow x$$
$$\mathcal{C} : (\alpha \to \mathcal{F}_E(\beta)) \to \mathcal{F}_E(\alpha \to \beta)$$
$$\mathcal{C} = \lambda f.\, \mathcal{C}\, f$$

$$\pi_1 : (\alpha \times \beta) \to \alpha$$
$$\pi_1 = \lambda P.\, \pi_1\, P$$
$$\pi_2 : (\alpha \times \beta) \to \beta$$
$$\pi_2 = \lambda P.\, \pi_2\, P$$
$$\mathrm{inl} : \alpha \to (\alpha + \beta)$$
$$\mathrm{inl} = \lambda x.\, \mathrm{inl}\ x$$
$$\mathrm{inr} : \beta \to (\alpha + \beta)$$
$$\mathrm{inr} = \lambda x.\, \mathrm{inr}\ x$$

Later on, in Section 3.3, we will see that our $\mathcal{F}_E$ is a functor which, combined with some other elements, forms a monad, or equivalently, a Kleisli triple. We use a star to denote the *extension* of a function from values to computations, as in [95], and we use $\gg\!=$ to denote the *bind* of a monad, as in Haskell.[24]

$$\begin{aligned} \_^* &: (\alpha \to \mathcal{F}_E(\beta)) \to (\mathcal{F}_E(\alpha) \to \mathcal{F}_E(\beta)) \\ f^* &= (\!|\, \eta \colon f \,|\!) \\ \_ \gg\!= \_ &: \mathcal{F}_E(\alpha) \to (\alpha \to \mathcal{F}_E(\beta)) \to \mathcal{F}_E(\beta) \\ M \gg\!= N &= N^*\, M \end{aligned}$$

Finally, we will define a notation for applying infix operators to arguments wrapped inside computations. Let $\_ \circledcirc \_$ be an infix operator of type $\alpha \to \beta \to \gamma$. Then we define the following:

$$\begin{aligned} \_ \ll\!\circledcirc \_ &: \mathcal{F}_E(\alpha) \to \beta \to \mathcal{F}_E(\gamma) \\ X \ll\!\circledcirc\, y &= X \gg\!= (\lambda x.\, \eta\, (x \circledcirc y)) \\ \_ \circledcirc\!\gg \_ &: \alpha \to \mathcal{F}_E(\beta) \to \mathcal{F}_E(\gamma) \\ x \circledcirc\!\gg Y &= Y \gg\!= (\lambda y.\, \eta\, (x \circledcirc y)) \\ \_ \ll\!\circledcirc\!\gg \_ &: \mathcal{F}_E(\alpha) \to \mathcal{F}_E(\beta) \to \mathcal{F}_E(\gamma) \\ X \ll\!\circledcirc\!\gg Y &= X \gg\!= (\lambda x.\, Y \gg\!= (\lambda y.\, \eta\, (x \circledcirc y))) \end{aligned}$$

In particular, we will be using this notation for the function application operator:

$$\begin{aligned} \_ \cdot \_ &: (\alpha \to \beta) \to \alpha \to \beta \\ f \cdot x &= f\, x \end{aligned}$$

which will yield the following combinators:

[24] In the types of the operators given below, we use the same effect signature E everywhere. Technically, a more general type could be derived for these terms given our system. However, we will rarely need this extra flexibility and so we stick with these simpler types.

$$
\begin{aligned}
\_ \ll\cdot \_ &: \mathcal{F}_E(\alpha \to \beta) \to \alpha \to \mathcal{F}_E(\beta) \\
F \ll\cdot x &= F \gg\!= (\lambda f.\, \eta\, (f\, x)) \\
\_ \cdot\!\gg \_ &: (\alpha \to \beta) \to \mathcal{F}_E(\alpha) \to \mathcal{F}_E(\beta) \\
f \cdot\!\gg X &= X \gg\!= (\lambda x.\, \eta\, (f\, x)) \\
\_ \ll\!\cdot\!\gg \_ &: \mathcal{F}_E(\alpha \to \beta) \to \mathcal{F}_E(\alpha) \to \mathcal{F}_E(\beta) \\
F \ll\!\cdot\!\gg X &= F \gg\!= (\lambda f.\, X \gg\!= (\lambda x.\, \eta\, (f\, x)))
\end{aligned}
$$

The first of the three, $\ll\cdot$, is the inverse function to $\mathcal{C}$. The second, $\cdot\!\gg$, is the morphism component of the $\mathcal{F}_E$ functor (which we will demonstrate in Section 3.3). $\mathcal{F}_E$ is also an applicative functor and the third operator in the list above, $\ll\!\cdot\!\gg$, is the operator for application within the functor.

1.6.2 Operations and Handlers

Now we will look at syntactic sugar specific to $(\!|\, \lambda \,|\!)$. In 1.6.1, we have seen the bind operator $\gg\!=$ and other ways of composing computations. Since we now have a practical way to compose computations, we can simplify the way we write effectful operations.

$$\mathtt{op}! = \lambda p.\, \mathtt{op}\, p\, (\lambda x.\, \eta\, x)$$

The exclamation mark partially applies an operation by giving it the trivial continuation η. However, we can still recover $\mathtt{op}$ from $\mathtt{op}!$ using $\gg\!=$:

$$
\begin{aligned}
\mathtt{op}!\, p \gg\!= k &= (\lambda p.\, \mathtt{op}\, p\, (\lambda x.\, \eta\, x))\, p \gg\!= k \\
&\to_\beta \mathtt{op}\, p\, (\lambda x.\, \eta\, x) \gg\!= k \\
&= k^*\, (\mathtt{op}\, p\, (\lambda x.\, \eta\, x)) \\
&= (\!|\, \eta \colon k \,|\!)\, (\mathtt{op}\, p\, (\lambda x.\, \eta\, x)) \\
&\to_{(\!|\, \mathtt{op}' \,|\!)} \mathtt{op}\, p\, (\lambda x.\, (\!|\, \eta \colon k \,|\!)\, (\eta\, x)) \\
&\to_{(\!|\, \eta \,|\!)} \mathtt{op}\, p\, (\lambda x.\, k\, x)
\end{aligned}
$$

The exclamation mark streamlines the typing rule for operations, as you can see on the pair of rules below:

$$\frac{\mathtt{op} : \alpha \rightarrowtail \beta \in E}{\Gamma \vdash \mathtt{op} : \alpha \to (\beta \to \mathcal{F}_E(\gamma)) \to \mathcal{F}_E(\gamma)}\ [\mathtt{op}] \qquad \frac{\mathtt{op} : \alpha \rightarrowtail \beta \in E}{\Gamma \vdash \mathtt{op}! : \alpha \to \mathcal{F}_E(\beta)}\ [\mathtt{op}!]$$

We can also see that the $\rightarrowtail$ arrow used in effect signatures gives rise to a Kleisli arrow since $\mathcal{F}_E$ is a monad (as we will see in Subsection 3.3.6).

Handlers

In Section 1.4, we have seen how the reduction rules treat unknown operation symbols: by leaving them intact. With some syntactic sugar, we can extend this behavior to the η operator as well. We will sometimes write a handler and omit giving the η clause. In that case, the η clause is presumed to be just η.[25] Schematically, we can define this piece of new syntax in the following way:

$$(\!|\, (\mathtt{op}_i \colon M_i)_{i \in I} \,|\!) = (\!|\, (\mathtt{op}_i \colon M_i)_{i \in I},\ \eta \colon \eta \,|\!)$$

Finally, we will introduce a special syntax for *closed handlers* [63]. A *closed handler* is a handler that interprets the entire computation that is given as its input (it must have a clause for every operator that

[25] This is not a part of the core calculus as this is only sound when the type of the handler is of the shape $\mathcal{F}_E(\gamma) \to \mathcal{F}_{E'}(\gamma)$ (i.e. when the handler preserves the type γ of values returned by the computation).

appears within). Since all effects are handled and none are forwarded, the codomain of the handler can be something else than a computation. However, if you try writing handlers that want to exploit this possibility, you will find that there is a lot of translating between α types and $\mathcal{F}_\emptyset(\alpha)$ types that needs to be done and that clouds the inherent simplicity of a closed handler. We introduce syntax for closed handlers which takes care of this problem.

$$\llparenthesis (\mathtt{op}_i\colon M_i)_{i\in I},\ \eta\colon M_\eta \rrparenthesis N = \downarrow_\circ (\llparenthesis (\mathtt{op}_i\colon (\lambda xk.\, \eta\, (M_i\, x\, (\downarrow_\circ \circ\, k))))_{i\in I},\ \eta\colon (\lambda x.\, \eta\, (M_\eta\, x))\rrparenthesis N)$$

As we can see, the only material added by the closed handler brackets are the functions η and $\downarrow_\circ$,[26] which simply translate between the types α and $\mathcal{F}_\emptyset(\alpha)$. Rather than closely studying the definition and scrutinizing the etas and the cherries, the idea of a closed handler is better conveyed by giving its typing rule. The following rule will be proven sound in 3.1.3:

$$\frac{\begin{array}{c} E = \{\mathtt{op}_i : \alpha_i \rightarrowtail \beta_i\}_{i\in I} \\ [\Gamma \vdash M_i : \alpha_i \to (\beta_i \to \delta) \to \delta]_{i\in I} \\ \Gamma \vdash M_\eta : \gamma \to \delta \\ \Gamma \vdash N : \mathcal{F}_E(\gamma) \end{array}}{\Gamma \vdash \llparenthesis (\mathtt{op}_i\colon M_i)_{i\in I},\ \eta\colon M_\eta \rrparenthesis N : \delta}\ [\llparenthesis\,\rrparenthesis]$$

The lack of multiple effect signatures to implement effect forwarding makes the $\llparenthesis\,\rrparenthesis$ rule simpler than the one of $\llparenthesis\,\rrparenthesis$:

- M_η gives δ-typed interpretations to the terminal values of type γ
- M_i maps the parameter of type α_i and the δ-typed interpretations of the β_i-indexed family of children to a δ-typed interpretation of an internal node

The η clause in a closed handler is optional. Similarly to open handlers, we will assume that $\gamma = \delta$ and that M_η is the identity function. This is the same as saying that a closed handler without an η clause is translated into an open handler without an η clause.

$$\begin{aligned} \llparenthesis (\mathtt{op}_i\colon M_i)_{i\in I} \rrparenthesis N &= \downarrow_\circ (\llparenthesis (\mathtt{op}_i\colon (\lambda xk.\, \eta\, (M_i\, x\, (\downarrow_\circ \circ\, k))))_{i\in I} \rrparenthesis N) \\ &= \downarrow_\circ (\llparenthesis (\mathtt{op}_i\colon (\lambda xk.\, \eta\, (M_i\, x\, (\downarrow_\circ \circ\, k))))_{i\in I},\ \eta\colon (\lambda x.\, \eta\, x)\rrparenthesis N) \\ &= \downarrow_\circ (\llparenthesis (\mathtt{op}_i\colon (\lambda xk.\, \eta\, (M_i\, x\, (\downarrow_\circ \circ\, k))))_{i\in I},\ \eta\colon (\lambda x.\, \eta\, ((\lambda x.\, x)\, x))\rrparenthesis N) \\ &= \llparenthesis (\mathtt{op}_i\colon M_i)_{i\in I},\ \eta\colon (\lambda x.\, x) \rrparenthesis N \end{aligned}$$

[26] Composing the $\downarrow_\circ$ operator with a $\llparenthesis\,\rrparenthesis$ handler is an identifying characteristic of closed handlers: a closed handler is a banana with a cherry on the top.

2

Examples

We have given a formal definition of $(\!| \lambda |\!)$, our calculus, but we have not shown how to use it to model side effects and what are the benefits. In this chapter, we will present an example problem and as we explore it deeper, we will see emerge some of the properties behind $(\!| \lambda |\!)$.

Contents

2.1 Introducing Our Running Example **27**
2.2 Adding Errors **28**
2.2.1 Raising the Type of exp 29
2.2.2 Refactoring with Monads 29
2.2.3 Interpreting Expressions as Computations 30
2.2.4 Handling Errors 30
2.2.5 Examples with Errors 31
2.3 Enriching the Context with Variables **33**
2.3.1 Example with Variables 34
2.3.2 Treating Variables without Computations 36
2.4 Summary **38**

2.1 Introducing Our Running Example

We will demonstrate our calculus by trying to build a calculator, showing how $(\!| \lambda |\!)$ can be used when defining the interpretation of a simple language of arithmetic expressions, similar to the one on which Wadler demonstrates monads in [133].[27] Given some arithmetic expression, we would like to reduce it to a simple number.

$$\textsc{sum}\,(\textsc{lit}\,1)\,(\textsc{prod}\,(\textsc{lit}\,2)\,(\textsc{lit}\,4)) \quad \rightarrow \quad 9$$

In the above, expressions are formed using the constructors SUM, PROD and LIT which stand for sums, products and literals, respectively. Expressions in our calculator have the type exp, the types of the constructors (formation rules) are given below:[28]

$$\begin{aligned} \textsc{sum} &: exp \rightarrow exp \rightarrow exp \\ \textsc{prod} &: exp \rightarrow exp \rightarrow exp \\ \textsc{lit} &: \mathbb{N} \rightarrow exp \end{aligned}$$

[27] The example we will treat might be too simple to actually justify using our calculus. However, it will allow us to see $(\!| \lambda |\!)$ in action.

[28] The type $\mathbb{N}$ is the type of natural numbers. Since our examples will deal with numbers, we will treat natural numbers as terms of our calculus that have the type $\mathbb{N}$. Their syntax and equational theory are both standard so we will not repeat it here.

The exercise is similar to our enterprise of formal semantics. We have a language (fragment) with a formal grammar and we try to find a systematic way of identifying sentences of this language with their senses or references. Our methodology will also be the same as the one of formal semantics: meanings will be composed in parallel with the syntactic structure.

Before we go on, a quick technical remark about the exp type and its encoding. A common approach is to model exp as an inductive algebraic data type[29] and then have our calculator be a function from exp to $\mathbb{N}$ or some other domain of interpretation. $(\!|\lambda|\!)$ does not directly provide inductive types but the $\mathcal{F}$ type constructor is built on top of them and since it is parameterizable by an effect signature, it can be used to implement a variety of inductive types. In this particular scenario, we could use $\mathcal{F}_{\{\mathtt{sum}:1\rightarrowtail 2,\mathtt{prod}:1\rightarrowtail 2\}}(\mathbb{N})$ as exp. The notion of a closed handler ($(\!|\,|\!)$, see 1.6.2) would then give use the induction principle over these arithmetic expression, i.e. a way to compositionally compute some value from the arithmetic expression.

We will not be using the encoding described above. The point of this chapter is to make the reader familiar with the notion of using the $\mathcal{F}$-types to model computations. We will therefore restrict their use to computations.

We can avoid introducing any kind of type for encoding arithmetic expressions. The reduction step from an arithmetic expression to a simple number can be realized by adding equations that "define" the functions SUM, PROD, LIT and the type exp. We thus take exp to be the name of the type of our desired interpretations and we take SUM, PROD and LIT to be some functions that compute directly with these interpretations.[30] The task of the semanticist is to fix some specific interpretation for the abstract type exp and figuring out the definitions of the abstract functions SUM, PROD and LIT.[31] Within our tiny fragment, this is quite easy to do:

$$\begin{aligned} exp &= \mathbb{N} \\ \textsc{sum} &= \lambda xy.\, x + y \\ \textsc{prod} &= \lambda xy.\, x \times y \\ \textsc{lit} &= \lambda x.\, x \end{aligned}$$

Applying these equations to the term SUM (LIT 1) (PROD (LIT 2) (LIT 4)) and then reducing according to the reduction rules of $(\!|\lambda|\!)$ (alongside with rules/equations for performing arithmetic operations), we get the result, 9.

2.2 Adding Errors

We will now try expanding our fragment by adding integer division.

$$\textsc{div} : exp \to exp \to exp$$

We could be tempted to treat this the same way and simply give this semantics to DIV:

$$\textsc{div} = \lambda xy.\, x/y$$

However, this assumes that we have a division operator that defines the division x/y for all natural numbers x and y. This is not the case when $y = 0$. Defining $x/0$ to be some specific natural number d is not satisfactory since then we cannot distinguish between the cases when x/y is d and when x/y is not defined. Fortunately, this is what sum types are very useful for. We will have the type of x/y be $\mathbb{N} + 1$, $x/0$ will reduce to (inr $\star$) and x/y for $y \neq 0$ will reduce to (inl q) where q is the greatest natural number such that $qy \leq x$.

[29] This would be something called an *initial encoding* of a language.

[30] This is known as a *final encoding* of a language.

[31] As we will see in Chapter 5 when we introduce them, this is exactly the idea behind abstract categorial grammars [35].

2.2.1 Raising the Type of exp

Now the cases when x/y is and is not defined are clearly delimited.[32] Nevertheless, we just made ourselves a new problem. The result of a division is something of type $\mathbb{N}+1$ and so in our interpretation, exp will correspond to $\mathbb{N}+1$. This is in line with our intuition that arithmetic expressions which contain division can sometimes be undefined. However, the implications of having $exp = \mathbb{N}+1$ begin to sting when we consider the cases of SUM and PROD.[33]

$$
\begin{aligned}
exp &= \mathbb{N}+1 \\
\textsc{div} &= \lambda XY.\, \textbf{case}\ X\ \textbf{of}\ \{\text{inl}\ x \to \textbf{case}\ Y\ \textbf{of}\ \{\text{inl}\ y \to x/y;\ \text{inr}\ \_ \to \text{inr}\ \star\};\ \text{inr}\ \_ \to \text{inr}\ \star\} \\
\textsc{sum} &= \lambda XY.\, \textbf{case}\ X\ \textbf{of}\ \{\text{inl}\ x \to \textbf{case}\ Y\ \textbf{of}\ \{\text{inl}\ y \to \text{inl}\,(x+y);\ \text{inr}\ \_ \to \text{inr}\ \star\};\ \text{inr}\ \_ \to \text{inr}\ \star\} \\
\textsc{prod} &= \lambda XY.\, \textbf{case}\ X\ \textbf{of}\ \{\text{inl}\ x \to \textbf{case}\ Y\ \textbf{of}\ \{\text{inl}\ y \to \text{inl}\,(x\times y);\ \text{inr}\ \_ \to \text{inr}\ \star\};\ \text{inr}\ \_ \to \text{inr}\ \star\} \\
\textsc{lit} &= \lambda x.\ \text{inl}\ x
\end{aligned}
$$

We now have to inspect the results of both of the operands and proceed with the calculation only if both of the operands successfully yield a natural number.[34] In SUM, PROD and LIT, we also have to wrap the result in inl in order to go from $\mathbb{N}$ to $\mathbb{N}+1$. In DIV, we do not wrap the result using inl since division already produces a value of type $\mathbb{N}+1$. All this seems a heavy price to pay just to include division.

2.2.2 Refactoring with Monads

We can make this solution look a little bit better. We can introduce a function on types and a pair of combinators that will allow us to be a little less repetitive.

$$
\begin{aligned}
T_\perp(\alpha) &= \alpha + 1 \\
\eta_\perp &: \alpha \to T_\perp(\alpha) \\
\eta_\perp &= \text{inl} \\
(\gg\!=_\perp) &: T_\perp(\alpha) \to (\alpha \to T_\perp(\beta)) \to T_\perp(\beta) \\
(\gg\!=_\perp) &= \lambda Xk.\, \textbf{case}\ X\ \textbf{of}\ \{\text{inl}\ x \to k\,x;\ \text{inr}\ \_ \to \text{inr}\ \star\}
\end{aligned}
$$

With these in our hands, we can now straighten out our interpretation.

$$
\begin{aligned}
exp &= T_\perp(\mathbb{N}) \\
\textsc{div} &= \lambda XY.\, X \gg\!=_\perp (\lambda x.\, Y \gg\!=_\perp (\lambda y.\, x/y)) \\
\textsc{sum} &= \lambda XY.\, X \gg\!=_\perp (\lambda x.\, Y \gg\!=_\perp (\lambda y.\, \eta_\perp\,(x+y))) \\
\textsc{prod} &= \lambda XY.\, X \gg\!=_\perp (\lambda x.\, Y \gg\!=_\perp (\lambda y.\, \eta_\perp\,(x\times y))) \\
\textsc{lit} &= \lambda x.\, \eta_\perp\, x
\end{aligned}
$$

This pattern that we uncovered in the type $\mathbb{N}+1$ and the terms defining DIV, SUM, PROD and LIT is not incidental. The triple $\langle T_\perp, \eta_\perp, \gg\!=_\perp\rangle$ forms a monad. This formulation in terms of monadic operations will allow us to transition more easily into our proposed solution since, as we will show in 3.3.6, $\langle \mathcal{F}_E, \eta, \gg\!=\rangle$ also forms a monad.

[32] Realizing that x/y is not defined for $y=0$ and then changing the type and behavior of the underlying division operator is very much like semanticists taking into account that expressions such as *the King of France* need not have a reference and then changing the underlying model.

[33] As a convention, we use _ as a variable name for variables whose values are never used.

[34] If you are unfamiliar with the notation used in the terms above, consult Section 1.5, where we introduced sums and products into our calculus.

2.2.3 Interpreting Expressions as Computations

We will now explore an alternative solution which takes advantage of the $\mathcal{F}$-types in $(\!|\lambda|\!)$. We will interpret expressions as computations of natural numbers, $exp = \mathcal{F}_E(\mathbb{N})$. Computations can have the ability to fail by using the operation $\texttt{fail} : 1 \rightarrowtail 0$. The input type 1 means that there is only one way to fail (i.e. `fail` does not distinguish failure states) and the output type 0 means that there is no continuation (formally there is a dummy continuation that accepts the impossible type 0). There is a natural way to generalize this approach by using $\texttt{error} : \chi \rightarrowtail 0$ instead. Computations can now terminate by throwing exceptions of type χ, allowing us to distinguish failure states. To identify the division-by-zero failure state, we will introduce $\text{DivisionByZero} : \chi$. A computation that uses the `error` operation to throw a division-by-zero exception would look like the following:

$$\begin{aligned}
&\dot{\frac{}{0}} : \mathcal{F}_{\{\texttt{error}:\chi\rightarrowtail 0\}}(\alpha) \\
&\dot{\frac{}{0}} = \texttt{error}\ \text{DivisionByZero}\,(\lambda o.\,\textbf{case}\ o\ \textbf{of}\ \{\})
\end{aligned}$$

The continuation uses the [empty] rule to turn the o of type 0 into something of type $\mathcal{F}_E(\alpha)$.

$$\begin{aligned}
exp &= \mathcal{F}_{\{\texttt{error}:\chi\rightarrowtail 0\}}(\mathbb{N}) \\
\textsc{div} &= \lambda XY.\,X \gg\!= (\lambda x.\,Y \gg\!= (\lambda y.\,\textbf{case}\ (x/y)\ \textbf{of}\ \{\text{inl}\ z \rightarrow \eta\, z;\ \text{inr}\ _ \rightarrow \dot{\frac{}{0}}\})) \\
\textsc{sum} &= \lambda XY.\,X \gg\!= (\lambda x.\,Y \gg\!= (\lambda y.\,\eta\,(x+y))) \\
\textsc{prod} &= \lambda XY.\,X \gg\!= (\lambda x.\,Y \gg\!= (\lambda y.\,\eta\,(x\times y))) \\
\textsc{lit} &= \lambda x.\,\eta\, x
\end{aligned}$$

Let us compare this with the last set of definitions:

- We replaced the $\langle T_\perp, \eta_\perp, \gg\!=_\perp\rangle$ monad with the $\langle \mathcal{F}_{\{\texttt{error}:\chi\rightarrowtail 0\}}, \eta, \gg\!=\rangle$ monad.
- Since the type of x/y is not the same as the type of our interpretations, we need to translate from the type $\mathbb{N}+1$ to the type $\mathcal{F}_{\{\texttt{error}:\chi\rightarrowtail 0\}}(\mathbb{N})$ by case analysis.

The advantage to using the $\mathcal{F}_E$ monad instead of the $T_\perp$ monad is that the $\mathcal{F}_E$ can be extended to handle other kinds of effects besides exceptions whereas the $T_\perp$ monad would need to be replaced by a different one. We will see an example of this later in 2.3. For now, we still have something more to explore regarding errors.

2.2.4 Handling Errors

We can now use division in our small calculator language. However, division can make the evaluation of an expression fail and yield no useful result. We could thus ask for a way to speculatively evaluate a subexpression and if it fails, recover by providing some default value and carry on evaluating the rest of the expression.

We add a new construction into our language:

$$\textsc{try} : exp \rightarrow exp \rightarrow exp$$

Its intended meaning is to return the value of its second argument. However, if the second argument fails to evaluate, the value of the first argument should be used instead.

$$\textsc{try}\,(\textsc{lit}\,42)\,(\textsc{div}\,(\textsc{lit}\,1)\,(\textsc{lit}\,0)) \quad\rightarrow\quad \eta\,42$$

Our task now is to give a formal semantics to TRY.

$$\textsc{try} = \lambda XY.\, (\!|\, \texttt{error}\colon (\lambda ek.\, X)\, |\!)\, Y$$

Instead of passing Y through the $\gg\!=$ operator, which basically says to do whatever Y would do (i.e. fail whenever Y fails), we apply a handler to Y.[35] This handler is very simple, it replaces any failed computation with the computation X (the backup computation that should yield the default value). It behaves like the exception handlers of common programming languages, whenever the computation Y would throw an `error`, we handle it by running computation X instead.

2.2.5 Examples with Errors

We now have enough interesting material to play around with and see if (and why) it really works as it should. We will take an expression large enough to contain all the features which we added and we will evaluate it piece by piece.

$$\textsc{sum}\,(\textsc{lit}\,5)\,(\textsc{try}\,(\textsc{lit}\,0)\,(\textsc{prod}\,(\textsc{lit}\,3)\,(\textsc{div}\,(\textsc{lit}\,2)\,(\textsc{lit}\,0))))$$

Let's start with $\textsc{div}\,(\textsc{lit}\,2)\,(\textsc{lit}\,0)$.

$$\begin{aligned}
\textsc{div}\,(\textsc{lit}\,2)\,(\textsc{lit}\,0) &= \textsc{div}\,(\eta\,2)\,(\eta\,0) && (1)\\
&= (\lambda XY.\, X \gg\!= (\lambda x.\, Y \gg\!= (\lambda y.\, \textbf{case}\,(x/y)\ \textbf{of}\ \{\text{inl}\ z \to \eta\, z;\ \text{inr}\ _ \to \dot{\tfrac{}{0}}\})))\,(\eta\,2)\,(\eta\,0) && (2)\\
&\to_{\beta,\beta} (\eta\,2) \gg\!= (\lambda x.\, (\eta\,0) \gg\!= (\lambda y.\, \textbf{case}\,(x/y)\ \textbf{of}\ \{\text{inl}\ z \to \eta\, z;\ \text{inr}\ _ \to \dot{\tfrac{}{0}}\})) && (3)\\
&= (\lambda x.\, (\eta\,0) \gg\!= (\lambda y.\, \textbf{case}\,(x/y)\ \textbf{of}\ \{\text{inl}\ z \to \eta\, z;\ \text{inr}\ _ \to \dot{\tfrac{}{0}}\}))^{*}\,(\eta\,2) && (4)\\
&= (\!|\, \eta\colon (\lambda x.\, (\eta\,0) \gg\!= (\lambda y.\, \textbf{case}\,(x/y)\ \textbf{of}\ \{\text{inl}\ z \to \eta\, z;\ \text{inr}\ _ \to \dot{\tfrac{}{0}}\}))\, |\!)\,(\eta\,2) && (5)\\
&\to_{(\!|\eta|\!)} (\lambda x.\, (\eta\,0) \gg\!= (\lambda y.\, \textbf{case}\,(x/y)\ \textbf{of}\ \{\text{inl}\ z \to \eta\, z;\ \text{inr}\ _ \to \dot{\tfrac{}{0}}\}))\,2 && (6)\\
&\to_{\beta} (\eta\,0) \gg\!= (\lambda y.\, \textbf{case}\,(2/y)\ \textbf{of}\ \{\text{inl}\ z \to \eta\, z;\ \text{inr}\ _ \to \dot{\tfrac{}{0}}\}) && (7)\\
&\to_{\ldots} (\lambda y.\, \textbf{case}\,(2/y)\ \textbf{of}\ \{\text{inl}\ z \to \eta\, z;\ \text{inr}\ _ \to \dot{\tfrac{}{0}}\})\,0 && (8)\\
&\to_{\beta} \textbf{case}\,(2/0)\ \textbf{of}\ \{\text{inl}\ z \to \eta\, z;\ \text{inr}\ _ \to \dot{\tfrac{}{0}}\} && (9)\\
&\to_{/} \textbf{case}\,(\text{inr}\ \star)\ \textbf{of}\ \{\text{inl}\ z \to \eta\, z;\ \text{inr}\ _ \to \dot{\tfrac{}{0}}\} && (10)\\
&\to_{\beta.+_2} \dot{\tfrac{}{0}} && (11)
\end{aligned}$$

The reductions state which rule was used to perform the step. The equality on line 2 is due to our interpretation of $\textsc{div}$. Those on lines 4 and 5 come from the definitions of $\gg\!=$ and * in 1.6.1. The step on line 8 is just a repeat of the steps on lines 4 to 6. This sequence of steps will be quite common and we will from now on refer to it as the reduction rule $\eta.\gg\!=$.[36]

The result of this first elaboration was not surprising: dividing 2 by 0 throws a DivisionByZero exception. We will now try to see what happens when this faulty expression appears within another expression.

[35] Feeding it into $\gg\!=$ or applying a handler to it are the two most common ways we will be using computations.

[36] This rule will be formally introduced in Subsection 3.1.2. It says that whenever we have $(\eta\,x) \gg\!= k$, we can reduce it to $k\,x$.

$$
\begin{aligned}
\textsc{prod}\,(\textsc{lit}\,3)\,\dot{\tfrac{}{0}} &= \textsc{prod}\,(\eta\,3)\,\dot{\tfrac{}{0}} && (1)\\
&= (\lambda XY.\,X \gg\!= (\lambda x.\,Y \gg\!= (\lambda y.\,\eta\,(x \times y))))\,(\eta\,3)\,\dot{\tfrac{}{0}} && (2)\\
&\to_{\beta,\beta} (\eta\,3) \gg\!= (\lambda x.\,\dot{\tfrac{}{0}} \gg\!= (\lambda y.\,\eta\,(x \times y))) && (3)\\
&\to_{\eta.\gg=} (\lambda x.\,\dot{\tfrac{}{0}} \gg\!= (\lambda y.\,\eta\,(x \times y)))\,3 && (4)\\
&\to_{\beta} \dot{\tfrac{}{0}} \gg\!= (\lambda y.\,\eta\,(3 \times y)) && (5)\\
&= (\texttt{error}\ \text{DivisionByZero}\,(\lambda o.\,\textbf{case}\ o\ \textbf{of}\ \{\})) \gg\!= (\lambda y.\,\eta\,(3 \times y)) && (6)\\
&= (\lambda y.\,\eta\,(3 \times y))^{*}\,(\texttt{error}\ \text{DivisionByZero}\,(\lambda o.\,\textbf{case}\ o\ \textbf{of}\ \{\})) && (7)\\
&= (\!|\,\eta\colon (\lambda y.\,\eta\,(3 \times y))\,|\!)\,(\texttt{error}\ \text{DivisionByZero}\,(\lambda o.\,\textbf{case}\ o\ \textbf{of}\ \{\})) && (8)\\
&\to_{(\!|\,\texttt{op}'\,|\!)} \texttt{error}\ \text{DivisionByZero}\,(\lambda o.\,(\!|\,\eta\colon (\lambda y.\,\eta\,(3 \times y))\,|\!)\,(\textbf{case}\ o\ \textbf{of}\ \{\})) && (9)\\
&\simeq \dot{\tfrac{}{0}} && (10)
\end{aligned}
$$

Line 2 is due to our interpretation of PROD. Line 4 is due to the $\eta.\gg\!=$ rule that we have demonstrated in the last example. Lines 6, 7 and 8 expand the definitions of $\dot{\tfrac{}{0}}$, $\gg\!=$ and * respectively. On line 10, we equate the term with $\dot{\tfrac{}{0}}$. The terms differ in the continuation but since the continuation can never be called,[37] we can consider them equal.

We have seen the DivisionByZero exception propagate. Now let's see what happens when it hits a TRY construction.

$$
\begin{aligned}
\textsc{try}\,(\textsc{lit}\,0)\,\dot{\tfrac{}{0}} &= \textsc{try}\,(\eta\,0)\,\dot{\tfrac{}{0}} && (1)\\
&= (\lambda XY.\,(\!|\,\texttt{error}\colon (\lambda ek.\,X)\,|\!)\,Y)\,(\eta\,0)\,\dot{\tfrac{}{0}} && (2)\\
&\to_{\beta,\beta} (\!|\,\texttt{error}\colon (\lambda ek.\,\eta\,0)\,|\!)\,\dot{\tfrac{}{0}} && (3)\\
&= (\!|\,\texttt{error}\colon (\lambda ek.\,\eta\,0)\,|\!)\,(\texttt{error}\ \text{DivisionByZero}\,(\lambda o.\,\textbf{case}\ o\ \textbf{of}\ \{\})) && (4)\\
&\to_{(\!|\,\texttt{op}\,|\!)} (\lambda ek.\,\eta\,0)\ \text{DivisionByZero}\,(\lambda o.\,(\!|\,\texttt{error}\colon (\lambda ek.\,\eta\,0)\,|\!)\,(\textbf{case}\ o\ \textbf{of}\ \{\})) && (5)\\
&\to_{\beta,\beta} \eta\,0 && (6)
\end{aligned}
$$

Line 2 is our interpretation of TRY. Line 4 is the definition of $\dot{\tfrac{}{0}}$.

We see that since the embedded expression failed to evaluate (its denotation was $\dot{\tfrac{}{0}}$), we have evaluated the literal 0 instead. We now have an actual number again and we can try feeding it into another operation.

[37] Technical aside: To be precise, we should define $\dot{\tfrac{}{0}}$ to be a class of terms of the shape $\texttt{error}\ \text{DivisionByZero}\,(\lambda o.\,M)$ for any term M. The only operations we will ever perform on $\dot{\tfrac{}{0}}$ in these examples will be congruent with the equivalence relation of our calculus refined by adding equalities between terms of the shape $\texttt{error}\ \text{DivisionByZero}\,(\lambda o.\,M)$. Therefore, so long as we end up removing the $\dot{\tfrac{}{0}}$ terms from our result, we can treat them as equivalent during our calculations.

$$
\begin{aligned}
\textsc{sum}\,(\textsc{lit}\,5)\,(\eta\,0) &= \textsc{sum}\,(\eta\,5)\,(\eta\,0) && (1)\\
&= (\lambda XY.\,X \gg\!= (\lambda x.\,Y \gg\!= (\lambda y.\,\eta\,(x+y))))\,(\eta\,5)\,(\eta\,0) && (2)\\
&\to_{\beta,\beta} (\eta\,5) \gg\!= (\lambda x.\,(\eta\,0) \gg\!= (\lambda y.\,\eta\,(x+y))) && (3)\\
&\to_{\eta.\gg=} (\lambda x.\,(\eta\,0) \gg\!= (\lambda y.\,\eta\,(x+y)))\,5 && (4)\\
&\to_{\beta} (\eta\,0) \gg\!= (\lambda y.\,\eta\,(5+y)) && (5)\\
&\to_{\eta.\gg=} (\lambda y.\,\eta\,(5+y))\,0 && (6)\\
&\to_{\beta} \eta\,(5+0) && (7)\\
&\to_{+} \eta\,5 && (8)
\end{aligned}
$$

Line 2 is our interpretation of SUM. The others are reductions of the kind we have seen before.

Finally, we have seen the simplest case, when an operator just reaches into the results of both operands using $\to_{\eta.\gg=}$ and carries out its operation. This also concludes our evaluation of the expression we have presented at the beginning of this subsection. We can therefore now conclude that:

$$\textsc{sum}\,(\textsc{lit}\,5)\,(\textsc{try}\,(\textsc{lit}\,0)\,(\textsc{prod}\,(\textsc{lit}\,3)\,(\textsc{div}\,(\textsc{lit}\,2)\,(\textsc{lit}\,0)))) \quad\to\quad \eta\,5$$

2.3 Enriching the Context with Variables

Sometimes, when writing down arithmetic expressions, it becomes useful to introduce variables for intermediate expressions and then build up the final result using those. Let's try and add such a facility to our calculator.

$$
\begin{aligned}
\textsc{let} &: \mathit{var} \to \mathit{exp} \to \mathit{exp} \to \mathit{exp}\\
\textsc{var} &: \mathit{var} \to \mathit{exp}\\
\bar{x}, \bar{y}, \bar{z}, \ldots &: \mathit{var}
\end{aligned}
$$

LET binds a variable to the result of the first expression; this variable stays available during the evaluation of the second expression. VAR then lets us use the variable in place of an expression. We also introduce terms for the different variables that can be used in our calculator. They look the same as the variables in $(\!|\,\lambda\,|\!)$ but they have a bar on top.

Here is a term that uses the new constructions:

$$
\begin{aligned}
&\textsc{let}\,\bar{x}\,(\textsc{sum}\,(\textsc{lit}\,2)\,(\textsc{lit}\,3))\\
&\quad(\textsc{let}\,\bar{y}\,(\textsc{prod}\,(\textsc{var}\,\bar{x})\,(\textsc{var}\,\bar{x}))\\
&\qquad(\textsc{prod}\,(\textsc{var}\,\bar{y})\,(\textsc{lit}\,2))) \quad\to\quad \eta\,50
\end{aligned}
$$

In order to give a semantics to LET and VAR, we will augment the set of operations in the computations we use to interpret the expressions. We want computations to be able to ask for the values of variables and so we will introduce $\mathtt{get} : \mathit{var} \rightarrowtail \mathbb{N}$. In our new interpretation, we will have $\mathit{exp} = \mathcal{F}_{\{\mathtt{error}:\chi\rightarrowtail 0,\mathtt{get}:\mathit{var}\rightarrowtail\mathbb{N}\}}(\mathbb{N})$. Though it might seem that we are changing our domain of interpretation and will thus need to redo our existing interpretations, it is actually not the case. The type that we associated with exp before could have been more precisely written as $\mathcal{F}_{\{\mathtt{error}:\chi\rightarrowtail 0\}\uplus E}(\mathbb{N})$, where E is a free variable ranging over effect signatures. What we are doing is simply specifying $E = \{\mathtt{get} : \mathit{var} \rightarrowtail \mathbb{N}\} \uplus E'$, where E' is again free.

$$
\begin{aligned}
\textsc{let} &= \lambda vXY.\,X \gg\!= (\lambda x.\,(\!|\,\mathtt{get}\colon (\lambda uk.\,\textbf{if}\ u = v\ \textbf{then}\ (k\,x)\ \textbf{else}\ (\mathtt{get}\,u\,k))\,|\!)\,Y)\\
\textsc{var} &= \mathtt{get}!
\end{aligned}
$$

In the implementation above, we also rely on the presence of an equality predicate on the var type:[38]

$$(=) : var \to var \to 2$$

And that is all we need. We can put these two combinators side-by-side with the existing ones like DIV and TRY and we will have a semantics for a calculator with exceptions and variables.

2.3.1 Example with Variables

To get some practice with $(\!| \lambda |\!)$, we will evaluate a simple expression with multiple variables, step by step.

$$\textsc{let}\, \bar{x}\, (\textsc{lit}\, 1)\, (\textsc{let}\, \bar{y}\, (\textsc{lit}\, 2)\, (\textsc{sum}\, (\textsc{var}\, \bar{x})\, (\textsc{var}\, \bar{y})))$$

Let us start off with the innermost expression, the sum of $\bar{x}$ and $\bar{y}$.

$$\begin{aligned}
\textsc{sum}\, (\textsc{var}\, \bar{x})\, (\textsc{var}\, \bar{y}) &= \textsc{sum}\, (\mathtt{get!}\, \bar{x})\, (\mathtt{get!}\, \bar{y}) && (1)\\
&= (\lambda XY.\, X \gg\!= (\lambda x.\, Y \gg\!= (\lambda y.\, \eta\, (x+y))))\, (\mathtt{get!}\, \bar{x})\, (\mathtt{get!}\, \bar{y}) && (2)\\
&\to_{\beta,\beta} (\mathtt{get!}\, \bar{x}) \gg\!= (\lambda x.\, (\mathtt{get!}\, \bar{y}) \gg\!= (\lambda y.\, \eta\, (x+y))) && (3)\\
&= ((\lambda p.\, \mathtt{get}\, p\, (\lambda x.\, \eta\, x))\, \bar{x}) \gg\!= (\lambda x.\, (\mathtt{get!}\, \bar{y}) \gg\!= (\lambda y.\, \eta\, (x+y))) && (4)\\
&\to_{\beta} (\mathtt{get}\, \bar{x}\, (\lambda x.\, \eta\, x)) \gg\!= (\lambda x.\, (\mathtt{get!}\, \bar{y}) \gg\!= (\lambda y.\, \eta\, (x+y))) && (5)\\
&= (\!|\, \eta\colon (\lambda x.\, (\mathtt{get!}\, \bar{y}) \gg\!= (\lambda y.\, \eta\, (x+y)))\, |\!)\, (\mathtt{get}\, \bar{x}\, (\lambda x.\, \eta\, x)) && (6)\\
&\to_{(\!|\, \mathtt{op'}\, |\!)} \mathtt{get}\, \bar{x}\, (\lambda x.\, (\!|\, \eta\colon (\lambda x.\, (\mathtt{get!}\, \bar{y}) \gg\!= (\lambda y.\, \eta\, (x+y)))\, |\!)\, (\eta\, x)) && (7)\\
&\to_{(\!|\, \eta\, |\!)} \mathtt{get}\, \bar{x}\, (\lambda x.\, (\lambda x.\, (\mathtt{get!}\, \bar{y}) \gg\!= (\lambda y.\, \eta\, (x+y)))\, x) && (8)\\
&\to_{\beta} \mathtt{get}\, \bar{x}\, (\lambda x.\, (\mathtt{get!}\, \bar{y}) \gg\!= (\lambda y.\, \eta\, (x+y))) && (9)\\
&\to_{\ldots} \mathtt{get}\, \bar{x}\, (\lambda x.\, \mathtt{get}\, \bar{y}\, (\lambda y.\, \eta\, (x+y))) && (10)
\end{aligned}$$

The first three lines are the usual. First, we expand the definitions of the operands (the VAR operator). Then we do the same for the operation SUM and we β reduce.

On line 4, we substitute get! with the definition of the exclamation mark that we provided in 1.6.2. Similarly, on line 6, we expand the definition of $\gg\!=$.[39]

If you compare line 3 and line 9 (the second to last line), you will notice that they are practically identical. The $(\mathtt{get!}\, \bar{x}) \gg\!= (\lambda x.\, \ldots)$ on line 3 has been replaced with $\mathtt{get}\, \bar{x}\, (\lambda x.\, \ldots)$. This is a derivable rule in $(\!|\, \lambda\, |\!)$, which we call op!.$\gg\!=$.[40]

On the last line, we simply repeat the last 6 steps (which we now packaged into the op!.$\gg\!=$ rule) for $\mathtt{get!}\, \bar{y}$.

We have our result, a computation that asks for the values of the variables $\bar{x}$ and $\bar{y}$ to compute their sum. Now we will compute the value of $\textsc{let}\, \bar{y}\, (\textsc{lit}\, 2)$.

$$\begin{aligned}
\textsc{let}\, \bar{y}\, (\textsc{lit}\, 2) &= \textsc{let}\, \bar{y}\, (\eta\, 2) && (1)\\
&= (\lambda vXY.\, X \gg\!= (\lambda x.\, (\!|\, \mathtt{get}\colon (\lambda uk.\, \mathbf{if}\ u = v\ \mathbf{then}\ (k\, x)\ \mathbf{else}\ (\mathtt{get}\, u\, k))\, |\!)\, Y))\, \bar{y}\, (\eta\, 2) && (2)\\
&\to_{\beta,\beta} \lambda Y.\, (\eta\, 2) \gg\!= (\lambda x.\, (\!|\, \mathtt{get}\colon (\lambda uk.\, \mathbf{if}\ u = \bar{y}\ \mathbf{then}\ (k\, x)\ \mathbf{else}\ (\mathtt{get}\, u\, k))\, |\!)\, Y) && (3)\\
&\to_{\eta.\gg\!=} \lambda Y.\, (\lambda x.\, (\!|\, \mathtt{get}\colon (\lambda uk.\, \mathbf{if}\ u = \bar{y}\ \mathbf{then}\ (k\, x)\ \mathbf{else}\ (\mathtt{get}\, u\, k))\, |\!)\, Y)\, 2 && (4)\\
&\to_{\beta} \lambda Y.\, (\!|\, \mathtt{get}\colon (\lambda uk.\, \mathbf{if}\ u = \bar{y}\ \mathbf{then}\ (k\, 2)\ \mathbf{else}\ (\mathtt{get}\, u\, k))\, |\!)\, Y && (5)\\
&\to_{\eta} (\!|\, \mathtt{get}\colon (\lambda uk.\, \mathbf{if}\ u = \bar{y}\ \mathbf{then}\ (k\, 2)\ \mathbf{else}\ (\mathtt{get}\, u\, k))\, |\!) && (6)
\end{aligned}$$

[38] 2 is the Boolean type that we have defined in Section 1.5 as $1 + 1$.

[39] $\gg\!=$ is defined in terms of *, which is itself defined in terms of $(\!|\ |\!)$. Here, we expand both in one go, i.e. $(x \gg\!= f) = ((\!|\, \eta\colon f\, |\!)\, x)$

[40] In Subsection 3.1.2, we will derive a similar, but more general, rule called op.$\gg\!=$, which will cover the steps from line 5 to line 8.

Lines 1 and 2 are our interpretations of LIT and LET and the rest are applications of rules we have seen before. We find out that the semantic action of $\textsc{let}\,\bar{y}\,(\textsc{lit}\,2)$ is a handler. This handler is only interested in get operations that act on the variable $\bar{y}$. It interprets such operations by returning the value 2, otherwise it leaves them untouched.

We will now apply this handler to the meaning of $\textsc{sum}\,(\textsc{var}\,\bar{x})\,(\textsc{var}\,\bar{y})$ to get the meaning of $\textsc{let}\,\bar{y}\,(\textsc{lit}\,2)\,(\textsc{sum}\,(\textsc{var}\,\bar{x})\,(\textsc{var}\,\bar{y}))$.

$$
\begin{aligned}
& \textsc{let}\,\bar{y}\,(\textsc{lit}\,2)\,(\textsc{sum}\,(\textsc{var}\,\bar{x})\,(\textsc{var}\,\bar{y})) && (1)\\
= {} & \textsc{let}\,\bar{y}\,(\textsc{lit}\,2)\,(\mathtt{get}\,\bar{x}\,(\lambda x.\,\mathtt{get}\,\bar{y}\,(\lambda y.\,\eta\,(x+y)))) && (2)\\
= {} & (\!|\,\mathtt{get}\colon (\lambda uk.\,\textbf{if}\ u=\bar{y}\ \textbf{then}\ (k\,2)\ \textbf{else}\ (\mathtt{get}\,u\,k))\,|\!)\,(\mathtt{get}\,\bar{x}\,(\lambda x.\,\mathtt{get}\,\bar{y}\,(\lambda y.\,\eta\,(x+y)))) && (3)\\
\to_{(\!|\,\mathtt{op}\,|\!)} {} & (\lambda uk.\,\textbf{if}\ u=\bar{y}\ \textbf{then}\ (k\,2)\ \textbf{else}\ (\mathtt{get}\,u\,k))\,\bar{x}\,(\lambda x.\,\textsc{let}\,\bar{y}\,(\textsc{lit}\,2)\,(\mathtt{get}\,\bar{y}\,(\lambda y.\,\eta\,(x+y)))) && (4)\\
\to_{\beta} {} & (\lambda k.\,\textbf{if}\ \bar{x}=\bar{y}\ \textbf{then}\ (k\,2)\ \textbf{else}\ (\mathtt{get}\,\bar{x}\,k))\,(\lambda x.\,\textsc{let}\,\bar{y}\,(\textsc{lit}\,2)\,(\mathtt{get}\,\bar{y}\,(\lambda y.\,\eta\,(x+y)))) && (5)\\
\to_{=} {} & (\lambda k.\,\textbf{if}\ \mathbf{F}\ \textbf{then}\ (k\,2)\ \textbf{else}\ (\mathtt{get}\,\bar{x}\,k))\,(\lambda x.\,\textsc{let}\,\bar{y}\,(\textsc{lit}\,2)\,(\mathtt{get}\,\bar{y}\,(\lambda y.\,\eta\,(x+y)))) && (6)\\
\to_{\textbf{if.F}} {} & (\lambda k.\,(\mathtt{get}\,\bar{x}\,k))\,(\lambda x.\,\textsc{let}\,\bar{y}\,(\textsc{lit}\,2)\,(\mathtt{get}\,\bar{y}\,(\lambda y.\,\eta\,(x+y)))) && (7)\\
\to_{\beta} {} & \mathtt{get}\,\bar{x}\,(\lambda x.\,\textsc{let}\,\bar{y}\,(\textsc{lit}\,2)\,(\mathtt{get}\,\bar{y}\,(\lambda y.\,\eta\,(x+y)))) && (8)\\
= {} & \mathtt{get}\,\bar{x}\,(\lambda x.\,(\!|\,\mathtt{get}\colon (\lambda uk.\,\textbf{if}\ u=\bar{y}\ \textbf{then}\ (k\,2)\ \textbf{else}\ (\mathtt{get}\,u\,k))\,|\!)\,(\mathtt{get}\,\bar{y}\,(\lambda y.\,\eta\,(x+y)))) && (9)\\
\to_{(\!|\,\mathtt{op}\,|\!)} {} & \mathtt{get}\,\bar{x}\,(\lambda x.\,((\lambda uk.\,\textbf{if}\ u=\bar{y}\ \textbf{then}\ (k\,2)\ \textbf{else}\ (\mathtt{get}\,u\,k))\,\bar{y}\,(\lambda y.\,\textsc{let}\,\bar{y}\,(\textsc{lit}\,2)\,(\eta\,(x+y))))) && (10)\\
\to_{\beta} {} & \mathtt{get}\,\bar{x}\,(\lambda x.\,((\lambda k.\,\textbf{if}\ \bar{y}=\bar{y}\ \textbf{then}\ (k\,2)\ \textbf{else}\ (\mathtt{get}\,\bar{y}\,k))\,(\lambda y.\,\textsc{let}\,\bar{y}\,(\textsc{lit}\,2)\,(\eta\,(x+y))))) && (11)\\
\to_{=} {} & \mathtt{get}\,\bar{x}\,(\lambda x.\,((\lambda k.\,\textbf{if}\ \mathbf{T}\ \textbf{then}\ (k\,2)\ \textbf{else}\ (\mathtt{get}\,\bar{y}\,k))\,(\lambda y.\,\textsc{let}\,\bar{y}\,(\textsc{lit}\,2)\,(\eta\,(x+y))))) && (12)\\
\to_{\textbf{if.T}} {} & \mathtt{get}\,\bar{x}\,(\lambda x.\,((\lambda k.\,k\,2)\,(\lambda y.\,\textsc{let}\,\bar{y}\,(\textsc{lit}\,2)\,(\eta\,(x+y))))) && (13)\\
\to_{\beta} {} & \mathtt{get}\,\bar{x}\,(\lambda x.\,(\lambda y.\,\textsc{let}\,\bar{y}\,(\textsc{lit}\,2)\,(\eta\,(x+y)))\,2) && (14)\\
\to_{\beta} {} & \mathtt{get}\,\bar{x}\,(\lambda x.\,\textsc{let}\,\bar{y}\,(\textsc{lit}\,2)\,(\eta\,(x+2))) && (15)\\
= {} & \mathtt{get}\,\bar{x}\,(\lambda x.\,(\!|\,\mathtt{get}\colon (\lambda uk.\,\textbf{if}\ u=\bar{y}\ \textbf{then}\ (k\,2)\ \textbf{else}\ (\mathtt{get}\,u\,k))\,|\!)\,(\eta\,(x+2))) && (16)\\
\to_{(\!|\,\eta\,|\!)} {} & \mathtt{get}\,\bar{x}\,(\lambda x.\,\eta\,(x+2)) && (17)
\end{aligned}
$$

This one is a bit more complicated and so we will go through it line by line:

Line 2 (=) We replace the expression $\textsc{sum}\,(\textsc{var}\,\bar{x})\,(\textsc{var}\,\bar{y})$ with its denotation that we computed earlier.

Line 3 (=) We do the same for the construction $\textsc{let}\,\bar{y}\,(\textsc{lit}\,2)$.

Line 4 ($\to_{(\!|\,\mathtt{op}\,|\!)}$) The let-binder is a handler with a clause for get and its argument is a computation that performs get. This means we do two things: we replace get with the clause $(\lambda uk.\ \ldots)$ from the handler and we move the handler inside the continuation. In order to save space, we contract the handler back to $\textsc{let}\,\bar{y}\,(\textsc{lit}\,2)$.

Line 5 (β) Here we apply the handler clause to the operation's argument, the variable $\bar{x}$. We do not do the same for the continuation argument k since we would have to copy it twice and we can avoid that by reducing the conditional expression first.

Line 6 (=) Since $\bar{x}$ and $\bar{y}$ are different variables, we expect the equality predicate to return false (**F**).

Line 7 (if.F) Since the condition evaluated to **F**, we choose the second branch of the conditional. The reduction rule used here was derived in Subsection 1.5.4.

Line 8 (β) We have simplified the handler clause and so we are ready to substitute in the continuation. Remark the similarity between this line and line 2. The only difference is the position of the $\textsc{let}\,\bar{y}\,(\textsc{lit}\,2)$ handler. It has moved from the first get down to the second get.

Line 9 (=) We will proceed as we did from line 3 onward. We expand $\textsc{let}\,\bar{y}\,(\textsc{lit}\,2)$ to reveal the handler.

Line 10 ($(\!|\, \mathtt{op} \,|\!)$) We have another `get` computation and so we pull out the handler clause for `get`. We contract the handler back into $\textsc{let}\, \bar{y}\, (\textsc{lit}\, 2)$ and move it down, applying it to the next step of the computation in the same way as was described at line 4.

Line 11 (β) We pass the variable $\bar{y}$ into the handler clause so we can decide the condition.

Line 12 (=) This time, we are comparing a variable to itself and so the equality predicate will yield true (**T**).

Line 13 (if.T) Since the condition is true, we choose the first branch of the conditional.

Line 14 (β) Having simplified the handler clause, we can pass in the continuation so that we can supply it with the value of the variable $\bar{y}$ that it was looking for.

Line 15 (β) We pass 2 to the continuation.

Line 16 (=) Now we will use the handler one last time so we reveal it first by expanding $\textsc{let}\, \bar{y}\, (\textsc{lit}\, 2)$.

Line 17 ($(\!|\, \eta \,|\!)$) The handler has no clause for η. In 1.6.2, we have declared that this is just a syntactic shortcut for the handler having a default clause $\eta\colon \eta$. Applying the $(\!|\, \eta \,|\!)$ rule has two effects: we replace η with its handler clause, η, and we discard the handler since there is no continuation.

We have started with $\mathtt{get}\, \bar{x}\, (\lambda x.\, \mathtt{get}\, \bar{y}\, (\lambda y.\, \eta\, (x + y)))$ as the denotation of $\textsc{sum}\, (\textsc{var}\, \bar{x})\, (\textsc{var}\, \bar{y})$. By applying the denotation of $\textsc{let}\, \bar{y}\, (\textsc{lit}\, 2)$, we have interpreted away the $\mathtt{get}\, \bar{y}$ request by providing the value 2 as the response, leaving us with $\mathtt{get}\, \bar{x}\, (\lambda x.\, \eta\, (x + 2))$. We will resolve the variable $\bar{x}$ by analogy.

$$\textsc{let}\, \bar{x}\, (\textsc{lit}\, 1) \to_{\ldots} (\!|\, \mathtt{get}\colon (\lambda u k.\, \textbf{if}\ u = \bar{x}\ \textbf{then}\ (k\, 1)\ \textbf{else}\ (\mathtt{get}\, u\, k)) \,|\!)$$

This computation goes down the same way as the one for $\textsc{let}\, \bar{x}\, (\textsc{lit}\, 1)$, only the number and the variable name are different.

$$\begin{aligned}
\textsc{let}\, \bar{x}\, (\textsc{lit}\, 1)\, (\textsc{let}\, \bar{y}\, (\textsc{lit}\, 2)\, (\textsc{sum}\, (\textsc{var}\, \bar{x})\, (\textsc{var}\, \bar{y}))) &= \textsc{let}\, \bar{x}\, (\textsc{lit}\, 1)\, (\mathtt{get}\, \bar{x}\, (\lambda x.\, \eta\, (x + 2))) \\
&\to_{\ldots} \eta\, (1 + 2) \\
&\to_{+} \eta\, 3
\end{aligned}$$

We can perform the reduction from the first line to the second line by repeating the exact same steps as we did from line 8 to line 17 when simplifying $\textsc{let}\, \bar{y}\, (\textsc{lit}\, 2)\, (\mathtt{get}\, \bar{y}\, (\lambda y.\, \eta\, (x + y)))$.

Finally, we can conclude that according to our interpretations, the value of the expression we started off with is 3. In other words, the following holds:

$$\textsc{let}\, \bar{x}\, (\textsc{lit}\, 1)\, (\textsc{let}\, \bar{y}\, (\textsc{lit}\, 2)\, (\textsc{sum}\, (\textsc{var}\, \bar{x})\, (\textsc{var}\, \bar{y}))) \quad \to \quad \eta\, 3$$

2.3.2 Treating Variables without Computations

Before we proceed onto the next chapter, let us compare the calculator semantics that we have developed so far with an alternative that does not rely on the computation abstraction.

As a reminder, this is our calculator semantics from 2.2.1, along with a semantics for $\textsc{try}$.

$$\begin{aligned}
exp &= \mathbb{N} + 1 \\
\textsc{div} &= \lambda XY.\, \textbf{case}\ X\ \textbf{of}\ \{\text{inl}\ x \to \textbf{case}\ Y\ \textbf{of}\ \{\text{inl}\ y \to x/y;\ \text{inr}\ _ \to \text{inr}\ \star\};\ \text{inr}\ _ \to \text{inr}\ \star\} \\
\textsc{try} &= \lambda XY.\, \textbf{case}\ X\ \textbf{of}\ \{\text{inl}\ x \to \textbf{case}\ Y\ \textbf{of}\ \{\text{inl}\ y \to \text{inl}\ y;\ \text{inr}\ _ \to \text{inl}\ x\};\ \text{inr}\ _ \to \text{inr}\ \star\} \\
\textsc{sum} &= \lambda XY.\, \textbf{case}\ X\ \textbf{of}\ \{\text{inl}\ x \to \textbf{case}\ Y\ \textbf{of}\ \{\text{inl}\ y \to \text{inl}\, (x + y);\ \text{inr}\ _ \to \text{inr}\ \star\};\ \text{inr}\ _ \to \text{inr}\ \star\} \\
\textsc{prod} &= \lambda XY.\, \textbf{case}\ X\ \textbf{of}\ \{\text{inl}\ x \to \textbf{case}\ Y\ \textbf{of}\ \{\text{inl}\ y \to \text{inl}\, (x \times y);\ \text{inr}\ _ \to \text{inr}\ \star\};\ \text{inr}\ _ \to \text{inr}\ \star\} \\
\textsc{lit} &= \lambda x.\ \text{inl}\ x
\end{aligned}$$

What denotation should we assign to VAR x? It does not correspond to any particular natural number nor is it an expression that should always fail. We can find an answer to our question by expanding the domain from $\mathbb{N}+1$ to $\gamma \to (\mathbb{N}+1)$ where $\gamma = var \to (\mathbb{N}+1)$ is the type of environments.[41]

$$
\begin{aligned}
exp &= \gamma \to \mathbb{N}+1 \\
\gamma &= var \to \mathbb{N}+1 \\
\textsc{let} &= \lambda vXYe.\, \textbf{case}\ X\, e\ \textbf{of}\ \{\text{inl}\ x \to Y\,(\lambda u.\, \textbf{if}\ u = v\ \textbf{then}\ \text{inl}\ x\ \textbf{else}\ e\, u);\ \text{inr}\ \_ \to \text{inr}\ \star\} \\
\textsc{var} &= \lambda ve.\, e\, v \\
\textsc{div} &= \lambda XYe.\, \textbf{case}\ X\, e\ \textbf{of}\ \{\text{inl}\ x \to \textbf{case}\ Y\, e\ \textbf{of}\ \{\text{inl}\ y \to x/y;\ \text{inr}\ \_ \to \text{inr}\ \star\};\ \text{inr}\ \_ \to \text{inr}\ \star\} \\
\textsc{try} &= \lambda XYe.\, \textbf{case}\ Y\, e\ \textbf{of}\ \{\text{inl}\ y \to \text{inl}\ y;\ \text{inr}\ \_ \to X\, e\} \\
\textsc{sum} &= \lambda XYe.\, \textbf{case}\ X\, e\ \textbf{of}\ \{\text{inl}\ x \to \textbf{case}\ Y\, e\ \textbf{of}\ \{\text{inl}\ y \to \text{inl}\,(x+y);\ \text{inr}\ \_ \to \text{inr}\ \star\};\ \text{inr}\ \_ \to \text{inr}\ \star\} \\
\textsc{prod} &= \lambda XYe.\, \textbf{case}\ X\, e\ \textbf{of}\ \{\text{inl}\ x \to \textbf{case}\ Y\, e\ \textbf{of}\ \{\text{inl}\ y \to \text{inl}\,(x \times y);\ \text{inr}\ \_ \to \text{inr}\ \star\};\ \text{inr}\ \_ \to \text{inr}\ \star\} \\
\textsc{lit} &= \lambda xe.\ \text{inl}\ x
\end{aligned}
$$

Since we needed to expand the type exp into $\gamma \to \mathbb{N}+1$, we need to make appropriate changes to the previous denotations. When producing values of type exp, we now abstract over an argument e of type γ. When consuming values of type exp, we apply to them an environment of type γ to get a value of the old type $\mathbb{N}+1$.

In the rule for VAR, we just lookup the value of the variable in the environment. If the environment does not contain a value for this variable, it returns inr $\star$ of type $\mathbb{N}+1$. In that case, VAR returns the same to indicate that the expression has no reference. In LET, we evaluate X in the current environment e and if it has a value, we also evaluate Y. However, Y itself is evaluated in an environment in which v is bound to x, the value of X.

We can now compare this to the semantics that we have arrived at using effects and handlers.

$$
\begin{aligned}
exp &= \mathcal{F}_{\{\texttt{error}:\chi \rightarrowtail 0, \texttt{get}:var \rightarrowtail \mathbb{N}\}}(\mathbb{N}) \\
\textsc{let} &= \lambda vXY.\, X \gg\!= (\lambda x.\, (\!|\, \texttt{get}\colon (\lambda uk.\, \textbf{if}\ u = v\ \textbf{then}\ (k\, x)\ \textbf{else}\ (\texttt{get}\ u\, k))\, |\!)\, Y) \\
\textsc{var} &= \texttt{get!} \\
\textsc{div} &= \lambda XY.\, X \gg\!= (\lambda x.\, Y \gg\!= (\lambda y.\, \textbf{case}\ (x/y)\ \textbf{of}\ \{\text{inl}\ z \to \eta\, z;\ \text{inr}\ \_ \to \dot{\tfrac{}{0}}\})) \\
\textsc{try} &= \lambda XY.\, (\!|\, \texttt{error}\colon (\lambda ek.\, X)\, |\!)\, Y \\
\textsc{sum} &= \lambda XY.\, X \gg\!= (\lambda x.\, Y \gg\!= (\lambda y.\, \eta\, (x+y))) \\
\textsc{prod} &= \lambda XY.\, X \gg\!= (\lambda x.\, Y \gg\!= (\lambda y.\, \eta\, (x \times y))) \\
\textsc{lit} &= \lambda x.\, \eta\, x
\end{aligned}
$$

We notice that `get` is used in LET and VAR but it is not mentioned in other parts of the fragment. `error` is used in DIV (through $\dot{\tfrac{}{0}}$) and in TRY and again it is absent in the rest of the fragment. Compare this with the approach that models exp as $\gamma \to \mathbb{N}+1$. In all of the entries, we abstract over a variable of type γ for the current environment and we explicitly pass it around. We also do explicit case analysis on all intermediate results and we wrap all our results in inl. Furthermore, changing the type exp to accommodate LET and VAR forced us to change the definitions of DIV, TRY, SUM, PROD and LIT (i.e. all the other constructions in our fragment).

We could solve some of these problems by using the same technique we have seen in 2.2.2. We can find a monad whose type constructor is $T(\alpha) = \gamma \to \alpha + 1$, whose unit is $\eta = (\lambda xe.\ \text{inl}\ x)$ and whose bind simultaneously takes care of passing around the environment and performing case analysis on intermediate results. However, we would still need to rewrite the existing parts of the semantics to use the new monad.

[41] Environments are partial functions from variable names to natural numbers, hence the $+1$ in the result type.

2.4 Summary

In this chapter, we have hoped to convey two things:

- Some sense of familiarity with our calculus.

  We have performed long reduction chains which illustrated a large part of the reduction rules and syntax introduced in Chapter 1. The goal of that was to build up our intuition of how terms written in $(\!|\lambda|\!)$ actually look like and how to use them to write computations.

- A glimpse of the methodology we would like to adopt.

  As a running example in this chapter, we have studied a simple formal language and gave it a compositional semantics. In the process of doing that, we have seen some advantages of using computations as opposed to using simpler terms. Notably the fact that we could extend the fragment with new constructions that demonstrate new semantic phenomena, such as errors or variables, without having to rewrite existing denotations. One of the principal motivations of this thesis is to apply this technique to the problem of natural language semantics and verify whether we can enjoy the same properties.

3

Properties

We have given a formal definition of $(\!| \lambda |\!)$ in Chapter 1 and then demonstrated it on examples in Chapter 2. We now turn back to a study of $(\!| \lambda |\!)$ itself and we derive some of its formal properties.

Contents

3.1 Derived Rules **39**
3.1.1 Function Composition ($\circ$) 40
3.1.2 Monadic Bind ($\gg\!=$) 40
3.1.3 Closed Handlers ($(\!|\,|\!)$) 41
3.2 Type Soundness **42**
3.2.1 Subject Reduction 43
3.2.2 Progress 46
3.3 Algebraic Properties **47**
3.3.1 Denotational Semantics 48
3.3.2 Category 50
3.3.3 The Three Laws 51
3.3.4 Functor 52
3.3.5 Applicative Functor 53
3.3.6 Monad 56
3.3.7 Free Monad 57
3.4 Confluence **58**
3.4.1 Combinatory Reduction Systems 58
3.4.2 $(\!| \lambda |\!)$ as a CRS 60
3.4.3 Orthogonal CRSs 61
3.4.4 Putting η Back in $(\!| \lambda |\!)$ 63
3.5 Termination **64**
3.5.1 Inductive Data Type Systems 65
3.5.2 $(\!| \lambda |\!)$ as an IDTS 66
3.5.3 Termination for IDTSs 70
3.5.4 Higher-Order Semantic Labelling 73
3.5.5 Putting η Back in $(\!| \lambda |\!)$ 81

3.1 Derived Rules

At the end of Chapter 1, in 1.6, we have introduced some new syntax for $(\!| \lambda |\!)$ terms and we have translated that syntax into terms of the core $(\!| \lambda |\!)$ calculus. However, in Chapter 2, we have then seen that expanding

these syntactic extensions during the reduction of a term is a tedious process and we have introduced some shortcuts to allow us to proceed faster and at a higher level of abstraction (e.g. the $\eta.\gg=$ rule). In this section, we will give typing rules and reduction rules to these new constructions and prove their correctness.

3.1.1 Function Composition ($\circ$)

The first piece of syntactic sugar we have introduced was an infix symbol for function composition.

$$f \circ g = \lambda x.\, f\,(g\,x)$$

In order to type terms containing this symbol, it will be useful to have a typing rule.

Proposition 3.1.1. *The following typing rule is derivable in* $(\!|\,\lambda\,|\!)$*:*

$$\frac{\Gamma \vdash M : \beta \to \gamma \qquad \Gamma \vdash N : \alpha \to \beta}{\Gamma \vdash M \circ N : \alpha \to \gamma}\,[\circ]$$

Proof. Since $M \circ N = \lambda x.\, M\,(N\,x)$, we can prove the validity of this rule with the typing rule below:

$$\dfrac{\dfrac{\Gamma, x : \alpha \vdash M : \beta \to \gamma \qquad \dfrac{\Gamma, x : \alpha \vdash N : \alpha \to \beta \qquad \Gamma, x : \alpha \vdash x : \alpha}{\Gamma, x : \alpha \vdash N\,x : \beta}\,[\text{app}]}{\Gamma, x : \alpha \vdash M\,(N\,x) : \gamma}\,[\text{app}]}{\Gamma \vdash \lambda x.\, M\,(N\,x) : \alpha \to \gamma}\,[\text{abs}]$$

x is presumed to be fresh for M and N and so we can equate $\Gamma, x : \alpha \vdash M : \beta \to \gamma$ with $\Gamma \vdash M : \beta \to \gamma$ and the same for N. □

The result of function composition is another function and functions can be applied to arguments. We can derive a reduction rule for this kind of function.

Proposition 3.1.2. *The following reduction is derivable in* $(\!|\,\lambda\,|\!)$*:*

$$(M_1 \circ M_2)\,N \quad \to_\circ \quad M_1\,(M_2\,N)$$

Proof.

$$\begin{aligned}(M_1 \circ M_2)\,N &= (\lambda x.\, M_1\,(M_2\,x))\,N \\ &\to_\beta M_1\,(M_2\,N)\end{aligned}$$

□

3.1.2 Monadic Bind ($\gg=$)

As a reminder, we give the definition of $\gg=$ from 1.6.1.

$$\begin{aligned}M \gg= N &= N^*\,M \\ &= (\!|\,\eta\colon N\,|\!)\,M\end{aligned}$$

First, we will prove the correct typing for $\gg=$.

Proposition 3.1.3. *The following typing rule is derivable in* $(\!|\,\lambda\,|\!)$*:*

$$\frac{\Gamma \vdash M : \mathcal{F}_E(\alpha) \qquad \Gamma \vdash N : \alpha \to \mathcal{F}_E(\beta)}{\Gamma \vdash M \gg= N : \mathcal{F}_E(\beta)}\,[\gg=]$$

Proof. We note that $M \gg\!= N = (\!|\, \eta\colon N \,|\!)\, M$ and construct the following typing derivation in $(\!|\, \lambda \,|\!)$:

$$\dfrac{\dfrac{\Gamma \vdash N : \alpha \to \mathcal{F}_E(\beta)}{\Gamma \vdash (\!|\, \eta\colon N \,|\!) : \mathcal{F}_E(\alpha) \to \mathcal{F}_E(\beta)}\,[(\!|\ |\!)] \qquad \Gamma \vdash M : \mathcal{F}_E(\alpha)}{\Gamma \vdash (\!|\, \eta\colon N \,|\!)\, M : \mathcal{F}_E(\beta)}\,[\mathrm{app}]$$

□

Next, we will prove the validity of two reduction rules for $\gg\!=$.

Proposition 3.1.4. *The following reductions are derivable in* $(\!|\, \lambda \,|\!)$*:*

$$\begin{array}{rcl} \eta\, M \gg\!= N & \to_{\eta.\gg\!=} & N\, M \\ \mathtt{op}\, M_{\mathrm{p}}\, (\lambda x.\, M_{\mathrm{c}}) \gg\!= N & \to_{\mathtt{op}.\gg\!=} & \mathtt{op}\, M_{\mathrm{p}}\, (\lambda x.\, M_{\mathrm{c}} \gg\!= N) \end{array}$$

Proof.

$$\begin{aligned} \eta\, M \gg\!= N &= (\!|\, \eta\colon N \,|\!)\, (\eta\, M) \\ &\to_{(\!|\, \eta \,|\!)} N\, M \end{aligned}$$

$$\begin{aligned} \mathtt{op}\, M_{\mathrm{p}}\, (\lambda x.\, M_{\mathrm{c}}) \gg\!= N &= (\!|\, \eta\colon N \,|\!)\, (\mathtt{op}\, M_{\mathrm{p}}\, (\lambda x.\, M_{\mathrm{c}})) \\ &\to_{(\!|\, \mathtt{op}' \,|\!)} \mathtt{op}\, M_{\mathrm{p}}\, (\lambda x.\, (\!|\, \eta\colon N \,|\!)\, M_{\mathrm{c}}) \\ &= \mathtt{op}\, M_{\mathrm{p}}\, (\lambda x.\, M_{\mathrm{c}} \gg\!= N) \end{aligned}$$

□

3.1.3 Closed Handlers (❪❫)

In 1.6.2, we introduced a notation for closed handlers. Even though we define closed handlers in terms of (open) handlers, their typing and reduction rules are actually simpler, since they do not have to go out of their way to support openness (i.e. passing through uninterpreted operations).

$$❪(\mathtt{op}_i\colon M_i)_{i\in I},\ \eta\colon M_\eta❫\, N = \downarrow_{\circ} (\!|\, (\mathtt{op}_i\colon (\lambda x k.\, \eta\, (M_i\, x\, (\downarrow_{\circ} \circ k))))_{i\in I},\ \eta\colon (\lambda x.\, \eta\, (M_\eta\, x)) \,|\!)\, N)$$

We will first go through the typing rule.

Proposition 3.1.5. *The following typing rule is derivable in* $(\!|\, \lambda \,|\!)$*:*

$$\dfrac{\begin{array}{c} E = \{\mathtt{op}_i : \alpha_i \rightarrowtail \beta_i\}_{i\in I} \\ [\Gamma \vdash M_i : \alpha_i \to (\beta_i \to \delta) \to \delta]_{i\in I} \\ \Gamma \vdash M_\eta : \gamma \to \delta \\ \Gamma \vdash N : \mathcal{F}_E(\gamma) \end{array}}{\Gamma \vdash ❪(\mathtt{op}_i\colon M_i)_{i\in I},\ \eta\colon M_\eta❫\, N : \delta}\,[❪❫]$$

Proof. We have $❪(\mathtt{op}_i\colon M_i)_{i\in I},\ \eta\colon M_\eta❫\, N = \downarrow_{\circ} (\!|\, (\mathtt{op}_i\colon (\lambda x k.\, \eta\, (M_i\, x\, (\downarrow_{\circ} \circ k))))_{i\in I},\ \eta\colon (\lambda x.\, \eta\, (M_\eta\, x)) \,|\!)\, N)$ and we will proceed by building a typing derivation for this term.

$$\dfrac{\dfrac{\begin{array}{c} E = \{\mathtt{op}_i : \alpha_i \rightarrowtail \beta_i\}_{i\in I} \\ [\Gamma \vdash \lambda x k.\, \eta\, (M_i\, x\, (\downarrow_{\circ} \circ k)) : \alpha_i \to (\beta_i \to \mathcal{F}_\emptyset(\delta)) \to \mathcal{F}_\emptyset(\delta)]_{i\in I} \\ \Gamma \vdash \lambda x.\, \eta\, (M_\eta\, x) : \gamma \to \mathcal{F}_\emptyset(\delta) \\ \Gamma \vdash N : \mathcal{F}_E(\gamma) \end{array}}{\Gamma \vdash (\!|\, (\mathtt{op}_i\colon (\lambda x k.\, \eta\, (M_i\, x\, (\downarrow_{\circ} \circ k))))_{i\in I},\ \eta\colon (\lambda x.\, \eta\, (M_\eta\, x)) \,|\!)\, N : \mathcal{F}_\emptyset(\delta)}\,[(\!|\ |\!)]}{\Gamma \vdash \downarrow_{\circ} (\!|\, (\mathtt{op}_i\colon (\lambda x k.\, \eta\, (M_i\, x\, (\downarrow_{\circ} \circ k))))_{i\in I},\ \eta\colon (\lambda x.\, \eta\, (M_\eta\, x)) \,|\!)\, N) : \delta}\,[\downarrow_{\circ}]$$

We still need to prove both $\Gamma \vdash \lambda x k.\, \eta\, (M_i\, x\, (\downarrow_{\circ} \circ k)) : \alpha_i \to (\beta_i \to \mathcal{F}_\emptyset(\delta)) \to \mathcal{F}_\emptyset(\delta)$ for every $i \in I$ and $\Gamma \vdash \lambda x.\, \eta\, (M_\eta\, x) : \gamma \to \mathcal{F}_\emptyset(\delta)$.

$$
\dfrac{\dfrac{\dfrac{\dfrac{\dfrac{\Gamma \vdash M_i : \alpha_i \to (\beta_i \to \delta) \to \delta \qquad \Gamma, x : \alpha_i \vdash x : \alpha_i}{\Gamma, x : \alpha_i \vdash M_i\, x : (\beta_i \to \delta) \to \delta}\,[\mathrm{app}] \qquad \dfrac{\Gamma \vdash \downarrow_{\circ} : \mathcal{F}_{\emptyset}(\delta) \to \delta \qquad \Gamma, k : \beta_i \to \mathcal{F}_{\emptyset}(\delta) \vdash k : \beta_i \to \mathcal{F}_{\emptyset}(\delta)}{\Gamma, k : \beta_i \to \mathcal{F}_{\emptyset}(\delta) \vdash \downarrow_{\circ} \circ k : \beta_i \to \delta}\,[\circ]}{\Gamma, x : \alpha_i, k : \beta_i \to \mathcal{F}_{\emptyset}(\delta) \vdash M_i\, x\, (\downarrow_{\circ} \circ k) : \delta}\,[\mathrm{app}]}{\Gamma, x : \alpha_i, k : \beta_i \to \mathcal{F}_{\emptyset}(\delta) \vdash \eta\, (M_i\, x\, (\downarrow_{\circ} \circ k)) : \mathcal{F}_{\emptyset}(\delta)}\,[\eta]}{\Gamma, x : \alpha_i \vdash \lambda k.\, \eta\, (M_i\, x\, (\downarrow_{\circ} \circ k)) : (\beta_i \to \mathcal{F}_{\emptyset}(\delta)) \to \mathcal{F}_{\emptyset}(\delta)}\,[\mathrm{abs}]}{\Gamma \vdash \lambda x k.\, \eta\, (M_i\, x\, (\downarrow_{\circ} \circ k)) : \alpha_i \to (\beta_i \to \mathcal{F}_{\emptyset}(\delta)) \to \mathcal{F}_{\emptyset}(\delta)}\,[\mathrm{abs}]
$$

x and k are assumed to be fresh for M_i.

$$
\dfrac{\dfrac{\dfrac{\Gamma, x : \gamma \vdash M_\eta : \gamma \to \delta \qquad \Gamma, x : \gamma \vdash x : \gamma}{\Gamma, x : \gamma \vdash M_\eta\, x : \delta}\,[\mathrm{app}]}{\Gamma, x : \gamma \vdash \eta\, (M_\eta\, x) : \mathcal{F}_{\emptyset}(\delta)}\,[\eta]}{\Gamma \vdash \lambda x.\, \eta\, (M_\eta\, x) : \gamma \to \mathcal{F}_{\emptyset}(\delta)}\,[\mathrm{abs}]
$$

x is assumed to be fresh for M_η. □

We can also have reduction rules for closed handlers, which work exactly the same way as the open handler reduction rules (only they do not include cases for uninterpreted operations).

Proposition 3.1.6. *The following reductions are derivable in* $(\!|\, \lambda \,|\!)$*:*

$$
\begin{array}{rcl}
(\!|\, (\mathtt{op}_i \colon M_i)_{i \in I},\ \eta \colon M_\eta \,|\!)\, (\eta\, N) & \to_{(\!|\eta|\!)} & M_\eta\, N \\
(\!|\, (\mathtt{op}_i \colon M_i)_{i \in I},\ \eta \colon M_\eta \,|\!)\, (\mathtt{op}_i\, N_{\mathrm{p}}\, (\lambda x.\, N_{\mathrm{c}})) & \to_{(\!|\mathtt{op}|\!)} & M_i\, N_{\mathrm{p}}\, (\lambda x.\, (\!|\, (\mathtt{op}_i \colon M_i)_{i \in I},\ \eta \colon M_\eta \,|\!)\, N_{\mathrm{c}})
\end{array}
$$

Proof.

$$
\begin{array}{rl}
(\!|\, (\mathtt{op}_i \colon M_i)_{i \in I},\ \eta \colon M_\eta \,|\!)\, (\eta\, N) = & \downarrow_{\circ} (\!|\, (\mathtt{op}_i \colon (\lambda x k.\, \eta\, (M_i\, x\, (\downarrow_{\circ} \circ k))))_{i \in I},\ \eta \colon (\lambda x.\, \eta\, (M_\eta\, x)) \,|\!)\, (\eta\, N)) \\
\to_{(\!|\eta|\!)} & \downarrow_{\circ} ((\lambda x.\, \eta\, (M_\eta\, x))\, N) \\
\to_{\beta} & \downarrow_{\circ} (\eta\, (M_\eta\, N)) \\
\to_{\downarrow_{\circ}} & M_\eta\, N
\end{array}
$$

$$
\begin{array}{rl}
(\!|\, (\mathtt{op}_i \colon M_i)_{i \in I},\ \eta \colon M_\eta \,|\!)\, (\mathtt{op}_i\, N_{\mathrm{p}}\, (\lambda x.\, N_{\mathrm{c}})) = & \downarrow_{\circ} (\!|\, (\mathtt{op}_i \colon (\lambda x k.\, \eta\, (M_i\, x\, (\downarrow_{\circ} \circ k))))_{i \in I},\ \eta \colon (\lambda x.\, \eta\, (M_\eta\, x)) \,|\!)\, (\mathtt{op}_i\, N_{\mathrm{p}}\, (\lambda x.\, N_{\mathrm{c}}))) \\
\to_{(\!|\mathtt{op}|\!)} & \downarrow_{\circ} ((\lambda x k.\, \eta\, (M_i\, x\, (\downarrow_{\circ} \circ k)))\, N_{\mathrm{p}}\, (\lambda x.\, (\!|\, \ldots \,|\!)\, N_{\mathrm{c}})) \\
\to_{\beta} & \downarrow_{\circ} ((\lambda k.\, \eta\, (M_i\, N_{\mathrm{p}}\, (\downarrow_{\circ} \circ k)))\, (\lambda x.\, (\!|\, \ldots \,|\!)\, N_{\mathrm{c}})) \\
\to_{\beta} & \downarrow_{\circ} (\eta\, (M_i\, N_{\mathrm{p}}\, (\downarrow_{\circ} \circ (\lambda x.\, (\!|\, \ldots \,|\!)\, N_{\mathrm{c}})))) \\
\to_{\downarrow_{\circ}} & M_i\, N_{\mathrm{p}}\, (\downarrow_{\circ} \circ (\lambda x.\, (\!|\, \ldots \,|\!)\, N_{\mathrm{c}})) \\
= & M_i\, N_{\mathrm{p}}\, (\lambda x.\, \downarrow_{\circ} ((\lambda x.\, (\!|\, \ldots \,|\!)\, N_{\mathrm{c}})\, x)) \\
\to_{\beta} & M_i\, N_{\mathrm{p}}\, (\lambda x.\, \downarrow_{\circ} (\!|\, \ldots \,|\!)\, N_{\mathrm{c}})) \\
= & M_i\, N_{\mathrm{p}}\, (\lambda x.\, (\!|\, (\mathtt{op}_i \colon M_i)_{i \in I},\ \eta \colon M_\eta \,|\!)\, N_{\mathrm{c}})
\end{array}
$$

In the above, $(\!|\, \ldots \,|\!)$ is taken to be a shortcut for $(\!|\, (\mathtt{op}_i \colon (\lambda x k.\, \eta\, (M_i\, x\, (\downarrow_{\circ} \circ k))))_{i \in I},\ \eta \colon (\lambda x.\, \eta\, (M_\eta\, x)) \,|\!)$. □

3.2 Type Soundness

In Chapter 1, we have introduced both a type system and a reduction semantics for $(\!|\, \lambda \,|\!)$. Now we will give more substance to these two definitions by proving properties which outline the relationship between them.

Types can give us two guarantees: typed terms do not get stuck and typed terms always terminate. The former property is known as *progress* and, in Subsection 3.2.2, we will show that it holds for $(\!|\, \lambda \,|\!)$ as

long as we abstain from using the partial function $\mathcal{C}$. The latter is known as *termination* and its proof is more involved, so we will delay it until 3.5.

For both of these properties to hold, it will be essential to prove that a typed term stays typed after performing a reduction. This will be the object of the next subsection.

3.2.1 Subject Reduction

We now turn our attention to the subject reduction property. We can summarize subject reduction with the slogan "reduction preserves types". The rest of this section will consider a formal proof of this property for $(\!|\lambda|\!)$, but before we begin, we present a small lemma.

Lemma 3.2.1. ***Substitution and types***
Whenever we have $\Gamma, x : \alpha \vdash M : \tau$ and $\Gamma \vdash N : \alpha$, we also have $\Gamma \vdash M[x := N] : \tau$ (i.e. we can substitute in M while preserving the type).

Proof. The proof is carried out by induction on the structure of M (or rather the structure of the type derivation $\Gamma \vdash M : \tau$).

- $M = y$
  - If $y = x$, then $M[x := N] = N$ and $\alpha = \tau$. We immediately have $\Gamma \vdash M[x := N] : \tau$ from the assumption that $\Gamma \vdash N : \alpha$.
  - If $y \neq x$, then $M[x := N] = x$ and we get $\Gamma \vdash M[x := N] : \tau$ from the assumption that $\Gamma, x : \alpha \vdash M : \tau$ and the fact that $x \notin \mathrm{FV}(M)$.
- All the other cases end up being trivial. We follow the definition of substitution (Definition 1.4.11) which just applies substitution to all of the subterms. For every such subterm, we make appeal to the induction hypothesis and construct the new typing derivation.

□

Property 3.2.2. ***Subject reduction***
If $\Gamma \vdash M : \tau$ and $M \to N$, then $\Gamma \vdash N : \tau$.

Proof. We prove this by induction on the reduction rule used in $M \to N$.

- $\boxed{M \to_\beta N}$

  It must be the case that $M = (\lambda x.\, M')\, M''$ and $N = M'[x := M'']$. Since, $\Gamma \vdash M : \tau$, we must have the following typing derivation:

$$\dfrac{\dfrac{\Gamma, x : \alpha \vdash M' : \tau}{\Gamma \vdash \lambda x.\, M' : \alpha \to \tau}\,[\text{abs}] \qquad \Gamma \vdash M'' : \alpha}{\Gamma \vdash (\lambda x.\, M')\, M'' : \tau}\,[\text{app}]$$

  We apply Lemma 3.2.1 to $\Gamma, x : \alpha \vdash M' : \tau$ and $\Gamma \vdash M'' : \alpha$ to get a typing derivation for $\Gamma \vdash M'[x := M''] : \tau$.

- $\boxed{M \to_\eta N}$

  We have $M = \lambda x.\, M'\, x$ with x fresh for M', $N = M'$ and $\tau = \tau_1 \to \tau_2$. Since $\Gamma \vdash M : \tau$, we have the following:

$$\dfrac{\dfrac{\Gamma, x : \tau_1 \vdash M' : \tau_1 \to \tau_2 \qquad \Gamma, x : \tau_1 \vdash x : \tau_1}{\Gamma, x : \tau_1 \vdash M'\, x : \tau_2}\,[\text{app}]}{\Gamma \vdash \lambda x.\, M'\, x : \tau_1 \to \tau_2}\,[\text{abs}]$$

From the above derivation, we can extract $\Gamma, x : \tau_1 \vdash M' : \tau_1 \to \tau_2$. However, since x is fresh for M', we can strengthen this to $\Gamma \vdash M' : \tau_1 \to \tau_2$, which is what we wanted to prove.

- $\boxed{M \to_{(\!|\eta|\!)} N}$

  We have $M = (\!|(\mathtt{op}_i\colon M_i)_{i\in I},\ \eta\colon M_\eta|\!)\,(\eta\, M')$, $N = M_\eta\, M'$, $\tau = \mathcal{F}_{E'}(\delta)$ and the following typing derivation for M:

$$\dfrac{\Gamma \vdash M_\eta : \gamma \to \mathcal{F}_{E'}(\delta) \qquad \dfrac{\Gamma \vdash M' : \gamma}{\Gamma \vdash \eta\, M' : \mathcal{F}_E(\gamma)} \qquad \dots}{\Gamma \vdash (\!|(\mathtt{op}_i\colon M_i)_{i\in I},\ \eta\colon M_\eta|\!)\,(\eta\, M') : \mathcal{F}_{E'}(\delta)}\ [(\!|\ |\!)]$$

  From the inferred typing judgments for M_η and M', we can build the typing derivation for $M_\eta\, M'$.

$$\dfrac{\Gamma \vdash M_\eta : \gamma \to \mathcal{F}_{E'}(\delta) \qquad \Gamma \vdash M' : \gamma}{\Gamma \vdash M_\eta\, M' : \mathcal{F}_{E'}(\delta)}\ [\text{app}]$$

- $\boxed{M \to_{(\!|\mathtt{op}|\!)} N}$

  We have $M = (\!|(\mathtt{op}_i\colon M_i)_{i\in I},\ \eta\colon M_\eta|\!)\,(\mathtt{op}_j\, M_\mathrm{p}\,(\lambda x.\, M_\mathrm{c}))$, $N = M_j\, M_\mathrm{p}\,(\lambda x.\, (\!|(\mathtt{op}_i\colon M_i)_{i\in I},\ \eta\colon M_\eta|\!)\, M_\mathrm{c})$ and $\tau = \mathcal{F}_{E'}(\delta)$.

$$\dfrac{\Gamma \vdash M_j : \alpha_j \to (\beta_j \to \mathcal{F}_{E'}(\delta)) \to \mathcal{F}_{E'}(\delta) \qquad \dfrac{\Gamma \vdash M_\mathrm{p} : \alpha_j \qquad \Gamma, x : \beta_j \vdash M_\mathrm{c} : \mathcal{F}_E(\gamma) \qquad \mathtt{op}_j : \alpha_j \rightarrowtail \beta_j \in E}{\Gamma \vdash \mathtt{op}_j\, M_\mathrm{p}\,(\lambda x.\, M_\mathrm{c}) : \mathcal{F}_E(\gamma)}\ [\mathtt{op}] \qquad \dots}{\Gamma \vdash (\!|(\mathtt{op}_i\colon M_i)_{i\in I},\ \eta\colon M_\eta|\!)\,(\mathtt{op}_j\, M_\mathrm{p}\,(\lambda x.\, M_\mathrm{c})) : \mathcal{F}_{E'}(\delta)}\ [(\!|\ |\!)]$$

  From the types of M_p, M_c and M_j, we can calculate the type of our redex, $M_j\, M_\mathrm{p}\,(\lambda x.\, (\!|(\mathtt{op}_i\colon M_i)_{i\in I},\ \eta\colon M_\eta|\!)\, M_\mathrm{c})$.

$$\dfrac{\dfrac{\Gamma \vdash M_j : \alpha_j \to (\beta_j \to \mathcal{F}_{E'}(\delta)) \to \mathcal{F}_{E'}(\delta) \qquad \Gamma \vdash M_\mathrm{p} : \alpha_j}{\Gamma \vdash M_j\, M_\mathrm{p} : (\beta_j \to \mathcal{F}_{E'}(\delta)) \to \mathcal{F}_{E'}(\delta)}\ [\text{app}] \qquad \dfrac{\dfrac{\Gamma, x : \beta_j \vdash M_\mathrm{c} : \mathcal{F}_E(\gamma) \qquad \dots}{\Gamma, x : \beta_j \vdash (\!|(\mathtt{op}_i\colon M_i)_{i\in I},\ \eta\colon M_\eta|\!)\, M_\mathrm{c} : \mathcal{F}_{E'}(\delta)}\ [(\!|\ |\!)]}{\Gamma \vdash \lambda x.\, (\!|(\mathtt{op}_i\colon M_i)_{i\in I},\ \eta\colon M_\eta|\!)\, M_\mathrm{c} : \beta_j \to \mathcal{F}_{E'}(\delta)}\ [\text{abs}]}{\Gamma \vdash M_j\, M_\mathrm{p}\,(\lambda x.\, (\!|(\mathtt{op}_i\colon M_i)_{i\in I},\ \eta\colon M_\eta|\!)\, M_\mathrm{c}) : \mathcal{F}_{E'}(\delta)}\ [\text{app}]$$

- $\boxed{M \to_{(\!|\mathtt{op}'|\!)} N}$

  We have $M = (\!|(\mathtt{op}_i\colon M_i)_{i\in I},\ \eta\colon M_\eta|\!)\,(\mathtt{op}\, M_\mathrm{p}\,(\lambda x.\, M_\mathrm{c}))$, $N = \mathtt{op}\, M_\mathrm{p}\,(\lambda x.\, (\!|(\mathtt{op}_i\colon M_i)_{i\in I},\ \eta\colon M_\eta|\!)\, M_\mathrm{c})$ and $\tau = \mathcal{F}_{E'}(\delta)$.

$$\dfrac{\dfrac{\Gamma \vdash M_\mathrm{p} : \alpha \qquad \Gamma, x : \beta \vdash M_\mathrm{c} : \mathcal{F}_E(\gamma) \qquad \mathtt{op} : \alpha \rightarrowtail \beta \in E}{\Gamma \vdash \mathtt{op}\, M_\mathrm{p}\,(\lambda x.\, M_\mathrm{c}) : \mathcal{F}_E(\gamma)}\ [\mathtt{op}] \qquad \mathtt{op} : \alpha \rightarrowtail \beta \in E' \qquad \dots}{\Gamma \vdash (\!|(\mathtt{op}_i\colon M_i)_{i\in I},\ \eta\colon M_\eta|\!)\,(\mathtt{op}\, M_\mathrm{p}\,(\lambda x.\, M_\mathrm{c})) : \mathcal{F}_{E'}(\delta)}\ [(\!|\ |\!)]$$

  From the inferred judgments, we can build a typing derivation for the redex.

$$\dfrac{\Gamma \vdash M_\mathrm{p} : \alpha \qquad \dfrac{\Gamma, x : \beta \vdash M_\mathrm{c} : \mathcal{F}_E(\gamma) \qquad \dots}{\Gamma, x : \beta \vdash (\!|(\mathtt{op}_i\colon M_i)_{i\in I},\ \eta\colon M_\eta|\!)\, M_\mathrm{c} : \mathcal{F}_{E'}(\delta)}\ [(\!|\ |\!)] \qquad \mathtt{op} : \alpha \rightarrowtail \beta \in E'}{\Gamma \vdash \mathtt{op}\, M_\mathrm{p}\,(\lambda x.\, (\!|(\mathtt{op}_i\colon M_i)_{i\in I},\ \eta\colon M_\eta|\!)\, M_\mathrm{c}) : \mathcal{F}_{E'}(\delta)}\ [\mathtt{op}]$$

- $\boxed{M \to_{\downarrow} N}$

  In this case, $M = \downarrow (\eta\, M')$ and $N = M'$.

$$\dfrac{\dfrac{\Gamma \vdash M' : \tau}{\Gamma \vdash \eta\, M' : \mathcal{F}_{\emptyset}(\tau)}\,[\eta]}{\Gamma \vdash \downarrow (\eta\, M') : \tau}\,[\downarrow]$$

  We immediately get $\Gamma \vdash M' : \tau$, which is the sought after typing derivation of the redex.

- $\boxed{M \to_{\mathcal{C}_\eta} N}$

  $M = \mathcal{C}\,(\lambda x.\, \eta\, M)$, $N = \eta\,(\lambda x.\, M)$ and $\tau = \mathcal{F}_E(\gamma \to \delta)$.

$$\dfrac{\dfrac{\dfrac{\Gamma, x : \gamma \vdash M : \delta}{\Gamma, x : \gamma \vdash \eta\, M : \mathcal{F}_E(\delta)}\,[\eta]}{\Gamma \vdash \lambda x.\, \eta\, M : \gamma \to \mathcal{F}_E(\delta)}\,[\text{abs}]}{\Gamma \vdash \mathcal{C}\,(\lambda x.\, \eta\, M) : \mathcal{F}_E(\gamma \to \delta)}\,[\mathcal{C}]$$

  From these judgments, we build a type for the redex.

$$\dfrac{\dfrac{\Gamma, x : \gamma \vdash M : \delta}{\Gamma \vdash \lambda x.\, M : \gamma \to \delta}\,[\text{abs}]}{\Gamma \vdash \eta\,(\lambda x.\, M) : \mathcal{F}_E(\gamma \to \delta)}\,[\eta]$$

- $\boxed{M \to_{\mathcal{C}_{\mathsf{op}}} N}$

  $M = \mathcal{C}\,(\lambda x.\, \mathsf{op}\, M_{\mathrm{p}}\,(\lambda y.\, M_{\mathrm{c}}))$, $N = \mathsf{op}\, M_{\mathrm{p}}\,(\lambda y.\, \mathcal{C}\,(\lambda x.\, M_{\mathrm{c}}))$ and $\tau = \mathcal{F}_E(\gamma \to \delta)$.

$$\dfrac{\dfrac{\dfrac{\Gamma, x : \gamma \vdash M_{\mathrm{p}} : \alpha \qquad \Gamma, x : \gamma, y : \beta \vdash M_{\mathrm{c}} : \mathcal{F}_E(\delta) \qquad \mathsf{op} : \alpha \rightarrowtail \beta \in E}{\Gamma, x : \gamma \vdash \mathsf{op}\, M_{\mathrm{p}}\,(\lambda y.\, M_{\mathrm{c}}) : \mathcal{F}_E(\delta)}\,[\mathsf{op}]}{\Gamma \vdash \lambda x.\, \mathsf{op}\, M_{\mathrm{p}}\,(\lambda y.\, M_{\mathrm{c}}) : \gamma \to \mathcal{F}_E(\delta)}\,[\text{abs}]}{\Gamma \vdash \mathcal{C}\,(\lambda x.\, \mathsf{op}\, M_{\mathrm{p}}\,(\lambda y.\, M_{\mathrm{c}})) : \mathcal{F}_E(\gamma \to \delta)}\,[\mathcal{C}]$$

  With the judgments above, we build the derivation below.

$$\dfrac{\Gamma \vdash M_{\mathrm{p}} : \alpha \qquad \dfrac{\dfrac{\Gamma, y : \beta, x : \gamma \vdash M_{\mathrm{c}} : \mathcal{F}_E(\delta)}{\Gamma, y : \beta \vdash \lambda x.\, M_{\mathrm{c}} : \gamma \to \mathcal{F}_E(\delta)}\,[\text{abs}]}{\Gamma, y : \beta \vdash \mathcal{C}\,(\lambda x.\, M_{\mathrm{c}}) : \mathcal{F}_E(\gamma \to \delta)}\,[\mathcal{C}] \qquad \mathsf{op} : \alpha \rightarrowtail \beta \in E}{\Gamma \vdash \mathsf{op}\, M_{\mathrm{p}}\,(\lambda y.\, \mathcal{C}\,(\lambda x.\, M_{\mathrm{c}})) : \mathcal{F}_E(\gamma \to \delta)}\,[\mathsf{op}]$$

  In the above we get $\Gamma \vdash M_{\mathrm{p}} : \alpha$ from $\Gamma, x : \gamma \vdash M_{\mathrm{p}} : \alpha$ and the rule's condition that $x \notin \mathrm{FV}(M_{\mathrm{p}})$.

- $\boxed{C[M'] \to C[N']}$

  The reduction relation of $(\!|\, \lambda \,|\!)$ is defined as the context closure of the individual reduction rules. We have covered the rules themselves, we now address the context closure. By induction hypothesis, we know that the reduction from $M' \to N'$ preserves types, i.e. for any Δ and α such that $\Delta \vdash M' : \alpha$, we have $\Delta \vdash N' : \alpha$.

We observe that the typing rules of $(\!| \lambda |\!)$ (Figure 1.1) are compositional, meaning that the type of a term depends only on the types of its subterms, not on their syntactic form. We can check this easily by looking at the premises of all of the typing rules. For every immediate subterm T, there is a premise $\Delta \vdash T : \alpha$ where T is a metavariable. We can therefore replace T and its typing derivation by some other T' with $\Delta \vdash T' : \alpha$.

Since the typing rules of $(\!| \lambda |\!)$ are compositional, we can replace the $\Delta \vdash M' : \alpha$ in $\Gamma \vdash C[M'] : \tau$ by $\Delta \vdash N' : \alpha$ and get $\Gamma \vdash C[N'] : \tau$.

□

We have proven subject reduction for core $(\!| \lambda |\!)$. The syntax, semantics and types that we have introduced for sums and products are standard. Their proofs of subject reduction carry over into our setting as well.

3.2.2 Progress

Progress means that typed terms are never stuck. Among the terms of $(\!| \lambda |\!)$, we will have to identify terms which are acceptable stopping points for reduction. Progress will mean that if a term is not in one of these acceptable positions, then there must be a way to continue reducing. The term we will use for these acceptable results is *value*.

Definition 3.2.3. *A $(\!| \lambda |\!)$ term is a **value** if it can be generated by the following grammar:*

$$\begin{aligned} V ::= \; & \lambda x.\, M \\ \mid \; & \mathtt{op}\, V\, (\lambda x.\, M) \\ \mid \; & \eta\, V \end{aligned}$$

where M ranges over $(\!| \lambda |\!)$ terms.

The above definition reflects the intuition that $(\!| \lambda |\!)$ consists of functions and computations, where functions are built using λ and computations using $\mathtt{op}$ and η. The other syntactic constructions (application, $(\!|\ |\!)$, $\downarrow_{\circ}$ and $\mathcal{C}$) all have rules which are supposed to eventually replace them with other terms.

As with subject reduction, before we proceed to the main property, we will start with a small lemma.

Lemma 3.2.4. ***Value classification***
Let V be a closed well-typed value (i.e. $\emptyset \vdash V : \tau$). Then the following hold:

- *if $\tau = \alpha \to \beta$, then $V = \lambda x.\, M$*
- *if $\tau = \mathcal{F}_E(\alpha)$, then either $V = \mathtt{op}\, V_{\mathrm{p}}\, (\lambda x.\, M_{\mathrm{c}})$ or $V = \eta\, V'$*

Proof.

- Assume $\tau = \alpha \to \beta$. If $V = \mathtt{op}\, V_{\mathrm{p}}\, (\lambda x.\, M_{\mathrm{c}})$ or $V = \eta\, V'$, then τ must be a computation type $\mathcal{F}_E(\gamma)$, which is a contradiction. The only remaining possibility is therefore $V = \lambda x.\, M$.
- Assume $\tau = \mathcal{F}_E(\alpha)$. If $V = \lambda x.\, M$, then τ must be a function type $\beta \to \gamma$, which is a contradiction. The only remaining possibilities are therefore $V = \mathtt{op}\, V_{\mathrm{p}}\, (\lambda x.\, M_{\mathrm{c}})$ or $V = \eta\, V'$.

□

Property 3.2.5. ***Progress***
Every closed well-typed term M from $(\!| \lambda |\!)$ without $\mathcal{C}$ and constants[42] *is either a value or is reducible to some other term.*

[42] Constants are assumed to be reduced away by some external rule. In our case, this will be the application of an ACG lexicon (5.2).

Proof. We will proceed by induction on M.

- $M = \lambda x.\, M'$

  Then M is already a value.

- $M = x$

  Impossible, since M must be a closed term.

- $M = M_1\, M_2$

  By induction hypothesis, M_1 and M_2 are either values or reducible terms. If either one is reducible, then our term is reducible as well and we are done. If neither is reducible, then they are both values. Since M is a closed well-typed term (i.e. $\emptyset \vdash M : \tau$), then $\vdash M_1 : \alpha \to \tau$ for some α. Thanks to Lemma 3.2.4, we have that $M_1 = \lambda x.\, M_1'$. This means that $M = (\lambda x.\, M_1')\, M_2$ and M is therefore reducible with β.

- $M = \mathtt{op}\, M_{\mathrm{p}}\, (\lambda x.\, M_{\mathrm{c}})$

  By induction hypothesis, M_{p} is either reducible or a value. If M_{p} is reducible, then so is M. If it is a value, then so is M as well.

- $M = \eta\, N$

  The same argument as for $\mathtt{op}$. By induction hypothesis N is reducible or a value and therefore the same holds for M.

- $M = (\!|\, (\mathtt{op}_i \colon M_i)_{i \in I},\ \eta \colon M_\eta \,|\!)\, N$

  By induction hypothesis, N is either a value or it is itself reducible. If it is reducible, then so is M. If it is not, then it must be a (closed) value. The type of N is a computation type $\mathcal{F}_E(\alpha)$ and so by Lemma 3.2.4, it must either be $\mathtt{op}\, V_{\mathrm{p}}\, (\lambda x.\, M_{\mathrm{c}})$ or $\eta\, V$. If $N = \mathtt{op}\, V_{\mathrm{p}}\, (\lambda x.\, M_{\mathrm{c}})$, then $(\!|\, (\mathtt{op}_i \colon M_i)_{i \in I},\ \eta \colon M_\eta \,|\!)\, (\mathtt{op}\, V_{\mathrm{p}}\, (\lambda x.\, M_{\mathrm{c}}))$ is reducible by $(\!|\, \mathtt{op} \,|\!)$ or $(\!|\, \mathtt{op}' \,|\!)$ (depending on whether or not $\mathtt{op} \in \{\mathtt{op}_i\}_{i \in I}$). Otherwise, if $N = \eta\, V$, then $(\!|\, (\mathtt{op}_i \colon M_i)_{i \in I},\ \eta \colon M_\eta \,|\!)\, (\eta\, V)$ is reducible by $(\!|\, \eta \,|\!)$.

- $M = \downarrow_{\circ} N$

  By induction hypothesis, N is either reducible or a value. As before, we only have to focus on the case when N is a value. From Lemma 3.2.4, we know that $N = \mathtt{op}\, V_{\mathrm{p}}\, (\lambda x.\, M_{\mathrm{c}})$ or $N = \eta\, V$. However, we can rule out the former since we know that $\emptyset \vdash N : \mathcal{F}_\emptyset(\alpha)$, meaning that $\mathtt{op}$ is not in the empty effect signature $\emptyset$. We therefore end up with $\downarrow_{\circ} (\eta\, V)$, which is reducible by $\downarrow_{\circ}$.

□

We have shown progress for $(\!|\, \lambda \,|\!)$ without $\mathcal{C}$. It is easy to see that we cannot do better, as the $\mathcal{C}$ operator can violate progress and get us stuck quite easily.

Observation 3.2.6. *There exists a closed well-typed term M from $(\!|\, \lambda \,|\!)$ without constants that is neither a value nor reducible to some other term.*

Proof. The most trivial example is $\mathcal{C}\, (\lambda x.\, x)$. The computation that is performed by the body of the function $\lambda x.\, x$ is entirely determined by the parameter x. It is therefore not possible to pull out this structure outside of the function. Therefore, applying the $\mathcal{C}$ operator to this function is undefined and evaluation gets stuck. □

3.3 Algebraic Properties

In this section, we will clarify what we mean when we say that the $\mathcal{F}_E(\alpha)$ computation types form a functor/applicative functor/monad and we will prove that the constructions in $(\!|\, \lambda \,|\!)$ conform to the laws of these algebraic structures.

The object on which we will build these mathematical structures will be the meanings of $(\!|\, \lambda \,|\!)$ terms. We will therefore start by building an interpretation for $(\!|\, \lambda \,|\!)$, a denotational semantics. Then we will be in measure to define the algebraic structures mentioned above and verify that their laws are satisfied.

3.3.1 Denotational Semantics

We start by identifying the domains of interpretation. For each type, we designate a set such that all terms having that type will be interpreted in that set. Before we do so, we introduce some notation on sets.

Notation 3.3.1. *Let A and B be sets. Then:*

- *A^B is the set of functions from B to A*
- *$A \times B$ is the cartesian product of A and B*
- *$A \sqcup B$ is the disjoint union of A and B*[43]
- *$A_\perp$ is the disjoint union of A and $\{\perp\}$*

Definition 3.3.2. *Given a set A_ν for every atomic type ν, the **interpretation of a type** τ is a set $[\![\tau]\!]$ defined inductively by:*

$$
\begin{aligned}
[\![\nu]\!] &= (A_\nu)_\perp \\
[\![\alpha \to \beta]\!] &= ([\![\alpha]\!] \to [\![\beta]\!])_\perp \\
[\![\mathcal{F}_E(\gamma)]\!] &= ([\![\gamma]\!] \sqcup \bigsqcup_{\mathsf{op}:\alpha \rightarrowtail \beta \in E} [\![\alpha]\!] \times [\![\mathcal{F}_E(\gamma)]\!]^{[\![\beta]\!]})_\perp
\end{aligned}
$$

Note that $[\![\mathcal{F}_E(\gamma)]\!]$ is recursively defined not only by induction on the type itself but also by its use of $[\![\mathcal{F}_E(\gamma)]\!]$ on the right hand side. Formally, we take $[\![\mathcal{F}_E(\gamma)]\!]$ to be the least fixed point of the monotone functional $F(X) = ([\![\gamma]\!] \sqcup \bigsqcup_{\mathsf{op}:\alpha \rightarrowtail \beta \in E} [\![\alpha]\!] \times X^{[\![\beta]\!]})_\perp$, whose existence is guaranteed by the Knaster-Tarski theorem [77, 126].

Notation 3.3.3. *We will use λ notation to write down elements of $[\![\alpha \to \beta]\!]$:*

- *$\lambda x.\, F(x) \in [\![\alpha \to \beta]\!]$ when $F(x) \in [\![\beta]\!]$ for every $x \in [\![\alpha]\!]$*
- *$\perp \in [\![\alpha \to \beta]\!]$*

We will use the following syntax to write down elements of $[\![\mathcal{F}_E(\gamma)]\!]$:

- *$\eta(x) \in [\![\mathcal{F}_E(\gamma)]\!]$ with $x \in [\![\gamma]\!]$*
- *$\mathsf{op}(p, c) \in [\![\mathcal{F}_E(\gamma)]\!]$ with $\mathsf{op} : \alpha \rightarrowtail \beta \in E$, $p \in [\![\alpha]\!]$ and $c \in [\![\mathcal{F}_E(\gamma)]\!]^{[\![\beta]\!]}$*
- *$\perp \in [\![\mathcal{F}_E(\gamma)]\!]$*

The definition of $[\![\tau]\!]$ follows the definition of a value (Definition 3.2.3): function types denote functions and computation types either denote atomic algebraic expressions (η) or applications of algebraic operations (op). In the denotational semantics, we also take care of the fact that terms can get stuck and fail to yield the expected value. We represent this by adding the element $\perp$ to the interpretation of every type.

Definition 3.3.4. *We define the **interpretation of a typing context** Γ as the set $[\![\Gamma]\!]$ of functions that map every $x : \alpha \in \Gamma$ to an element of $[\![\alpha]\!]$.*

*We will call these functions **valuations**. We will use the notation $e[x := f]$ to stand for the **extension** of e with $x \mapsto f$. The domain of the extension is $\mathrm{dom}(e) \cup x$. The extension maps x to f and every other variable in its domain to $e(x)$.*

[43] Note that this disjoint union operator $\sqcup$ is different from the $\uplus$ one from Chapter 1. $A \sqcup B$ is defined as $\{(x, 0) \mid x \in A\} \cup \{(x, 1) \mid x \in B\}$.

Definition 3.3.5. *Assume given $\mathcal{I}(c) \in [\![\alpha]\!]$ for every constant $c : \alpha \in \Sigma$. For a well-typed term M with $\Gamma \vdash M : \tau$, we define the* ***interpretation of term*** *M as a function $[\![M]\!]$ from $[\![\Gamma]\!]$ to $[\![\tau]\!]$. The definition proceeds by induction on M:*[44]

$$
\begin{aligned}
[\![\lambda x.\, M]\!](e) &= \lambda X.\, ([\![M]\!](e[x := X])) \\
[\![x]\!](e) &= e(x) \\
[\![M\, N]\!](e) &= \begin{cases} [\![M]\!](e)([\![N]\!](e)), & \textit{if } [\![M]\!](e) \textit{ is a function} \\ \bot, & \textit{if } [\![M]\!](e) \textit{ is } \bot \end{cases} \\
[\![c]\!](e) &= \mathcal{I}(c) \\
[\![\mathtt{op}\, M_{\mathrm{p}}\, (\lambda x.\, M_{\mathrm{c}})]\!](e) &= \mathtt{op}([\![M_{\mathrm{p}}]\!](e), \lambda X.\, ([\![M_{\mathrm{c}}]\!](e[x := X]))) \\
[\![\eta\, M]\!](e) &= \eta([\![M]\!](e)) \\
[\![(\!|\, (\mathtt{op}_i \colon M_i)_{i \in I},\ \eta \colon M_\eta \,|\!)\, N]\!](e) &= [\![(\!|\, (\mathtt{op}_i \colon M_i)_{i \in I},\ \eta \colon M_\eta \,|\!)]\!](e)([\![N]\!](e)) \\
[\![\downarrow_{\circ} M]\!](e) &= \begin{cases} x, & \textit{if } [\![M]\!](e) = \eta(x) \\ \bot, & \textit{otherwise} \end{cases} \\
[\![\mathcal{C}\, M]\!](e) &= [\![\mathcal{C}]\!]([\![M]\!](e))
\end{aligned}
$$

Definition 3.3.6. *The* ***interpretation of a handler*** *$(\!|\, (\mathtt{op}_i \colon M_i)_{i \in I},\ \eta \colon M_\eta \,|\!)$ within a valuation e (also written as $[\![(\!|\, (\mathtt{op}_i \colon M_i)_{i \in I},\ \eta \colon M_\eta \,|\!)]\!](e)$) is the function h defined inductively by:*

$$
\begin{aligned}
h(\eta(x)) &= \begin{cases} [\![M_\eta]\!](e)(x), & \textit{if } [\![M_\eta]\!](e) \textit{ is a function} \\ \bot, & \textit{otherwise} \end{cases} \\
h(\mathtt{op}_j(p, c)) &= \begin{cases} [\![M_j]\!](e)(p)(\lambda x.\, h(c(x))), & \textit{if } j \in I, \textit{ and } [\![M_j]\!](e) \textit{ and } [\![M_j]\!](e)(p) \textit{ are both functions} \\ \mathtt{op}_j(p, \lambda x.\, h(c(x))), & \textit{if } j \notin I \\ \bot, & \textit{otherwise} \end{cases} \\
h(\bot) &= \bot
\end{aligned}
$$

The equations defining h use h on the right-hand side. Nevertheless, h is well-defined since we can rely on induction. There is a ***well-founded ordering on the elements of*** *$[\![\mathcal{F}_E(\gamma)]\!]$, where $\forall x.\ \mathtt{op}(p, c) > c(x)$.*

The monotonic functional $F(X) = ([\![\gamma]\!] \sqcup \bigsqcup_{\mathtt{op}:\alpha \rightsquigarrow \beta \in E} [\![\alpha]\!] \times X^{[\![\beta]\!]})_\bot$ used in defining $[\![\mathcal{F}_E(\gamma)]\!]$ (Definition 3.3.2) is also Scott-continuous (i.e. it is both monotonic and it preserves suprema). By Kleene fixed-point theorem [74], we have that the least fixed point of F is the supremum of the series $\emptyset \subseteq F(\emptyset) \subseteq F(F(\emptyset)) \subseteq \ldots$.

Let the ***rank*** *of x be the smallest n such that $x \in F^n(\emptyset)$. The ordering $<_{\mathrm{r}}$, defined as $x <_{\mathrm{r}} y$ whenever $\mathrm{rank}(x) < \mathrm{rank}(y)$, is a well-founded ordering. It is also the inductive ordering that we were looking for. Whenever $\mathrm{rank}(\mathtt{op}(p, c)) = n$, then c is a function whose codomain is $F^{n-1}(\emptyset)$ and therefore $\forall x.\ \mathtt{op}(p, c) >_{\mathrm{r}} c(x)$.*

Definition 3.3.7. *The* ***interpretation of the $\mathcal{C}$ operator*** *is a function g defined inductively by:*

$$
g(f) = \begin{cases} \eta(h), & \textit{if } f \textit{ is a function and } \exists h.\ \forall x.\ f(x) = \eta(h(x)) \\ \mathtt{op}(p, \lambda y.\, g(\lambda x.\, c(x)(y))), & \textit{if } f \textit{ is a function and } \exists \mathtt{op}, p, c.\ \forall x.\ f(x) = \mathtt{op}(p, c(x)) \\ \bot, & \textit{otherwise} \end{cases}
$$

As with Definition 3.3.6, we have to show that this is actually a valid definition since we are using g on the right-hand side of an equation defining g. This time around, the arguments to g are functions whose codomain is the interpretation of some computation type $\mathcal{F}_E(\beta)$. We can extend a well-founded ordering on the set $[\![\mathcal{F}_E(\beta)]\!]$ to a ***well-founded ordering on*** *$[\![\alpha \to \mathcal{F}_E(\beta)]\!]$ by stating that $f < g$ whenever f and g are both functions (not $\bot$) and $\forall x \in [\![\alpha]\!].\ f(x) < g(x)$.*

[44] In the definition, we make use of $[\![(\!|\, (\mathtt{op}_i \colon M_i)_{i \in I},\ \eta \colon M_\eta \,|\!)]\!](e)$ and $[\![\mathcal{C}]\!]$. This notation is introduced right after this definition.

We have to show that the recursive call to g in the definition above is performed on an argument which is smaller than the original function. Let $f' = \lambda x.\, c(x)(y)$ be the function to which we recursively apply g. We have that $f(x) = \mathtt{op}(p, c(x))$ and $f'(x) = c(x)(y)$. We know that $\forall y.\ \mathtt{op}(p, c(x)) > c(x)(y)$, since that is the property of the well-founded ordering on the elements of $[\![\mathcal{F}_E(\gamma)]\!]$ established above. Therefore, we have that $\forall x \in [\![\alpha]\!].\ f(x) > f'(x)$ and so $f > f'$.

This was the entire definition of our denotational semantics.[45] We will now compare it to the reduction semantics introduced in 1.4.

Property 3.3.8. ***Soundness of reduction w.r.t. denotations***
Whenever $M \to N$ in $(\!|\lambda|\!)$, then $[\![M]\!] = [\![N]\!]$.

Proof. The property relies on two facts: that our denotational semantics is compositional, which means that the context closure of reduction rules preserves denotations, and that every individual reduction preserves denotations. To prove so for the β rule is a matter of proving a lemma stating that $[\![M]\!](e[x := [\![N]\!](e)]) = [\![M[x := N]]\!](e)$, which follows from the compositionality of the denotational semantics. For all the other rules, it suffices to use the definition of interpretation (Definition 3.3.5) to calculate the denotation of both the left-hand side and the right-hand side and verify that they are the same object. □

We see that equalities from the reduction semantics are carried over to the denotational semantics. The converse, however, is not the case.

Observation 3.3.9. ***Incompleteness of reduction w.r.t. denotations***
There exist terms M and N in $(\!|\lambda|\!)$ such that $[\![M]\!] = [\![N]\!]$ but M and N are not convertible.

Proof. Consider a stuck term such as $M = \mathcal{C}\,(\lambda x.\, x)$ and another term $N = (\!|\,|\!)\, M = (\!|\,|\!)\,(\mathcal{C}\,(\lambda x.\, x))$. Neither one of these two terms is reducible and neither one is a value. They are stuck and the denotational semantics assigns the value $\bot$ to both of them, therefore $[\![M]\!] = [\![N]\!]$. However, as a consequence of confluence (coming up in 3.4), a pair of different normal terms is never convertible, and therefore M and N are not convertible. □

And this concludes the definition of the denotational semantics of $(\!|\lambda|\!)$. Throughout most of the manuscript, we will be using the reduction semantics introduced in 1.4, even though it is incomplete, since it allows us to simplify terms in a mechanical and transparent step-by-step manner. However, the denotational semantics will be useful to us in the rest of this section since it will let us access extra equalities needed to prove some general laws.

3.3.2 Category

We aim to show that the computation types in $(\!|\lambda|\!)$ form a functor, applicative functor and a monad. All these terms are defined w.r.t. some category and so we will start by introducing the category underlying $(\!|\lambda|\!)$.

Definition 3.3.10. *A **category** consists of:*

- *a set of **objects***
- *for every two objects A and B, a set of **arrows** from A to B (an arrow f from A to B is written as $f : A \to B$)*
- *for any two arrows $f : B \to C$ and $g : A \to B$, there exists the composition of the two arrows $f \circ g : A \to C$*
- *for any object A, there exists a special arrow $\mathrm{id}_A : A \to A$*

[45] We could also extend this interpretation to sums and products. The types would be interpreted by $[\![\alpha \times \beta]\!] = ([\![\alpha]\!] \times [\![\beta]\!])_\bot$ and $[\![\alpha + \beta]\!] = ([\![\alpha]\!] \sqcup [\![\beta]\!])_\bot$. The term level definitions would be the standard definitions one would expect for pairs and variants (modulo the treatment of $\bot$).

- *the following equations hold for any $f : C \to D$, $g : B \to C$ and $h : A \to B$:*

$$(f \circ g) \circ h = f \circ (g \circ h) \qquad \textit{(Associativity)} \tag{3.1}$$
$$\mathrm{id}_D \circ f = f \qquad \textit{(Left identity)} \tag{3.2}$$
$$f \circ \mathrm{id}_C = f \qquad \textit{(Right identity)} \tag{3.3}$$

We will be working with a particular category, which we will call $(\!|\lambda|\!)$. The $(\!|\lambda|\!)$ category consists of:

objects: the types of the $(\!|\lambda|\!)$ calculus

arrows: for any two types α and β, the arrows from α to β are the functions from $[\![\alpha]\!]$ to $[\![\beta]\!]$

composition: composition of arrows is defined as composition of functions

identities: for every type α, we define id_α as the identity function with domain $[\![\alpha]\!]$

Since the arrows in our category are functions, the three laws of a category (associativity (3.1), left identity (3.2) and right identity (3.3)) fall out of the same properties for functions.

3.3.3 The Three Laws

Monads form a subset of applicative functors which in turn is a subset of functors. Instead of incrementally building up from a functor all the way to a monad, it will end up being more practical to first prove the monad laws and then illustrate how they let us verify the functor and applicative functor laws. Therefore, we first define our monadic bind operator and prove the three monad laws.

Definition 3.3.11. *Let X be from $[\![\mathcal{F}_E(\alpha)]\!]$ and f be a function from $[\![\alpha]\!]$ to $[\![\mathcal{F}_E(\beta)]\!]$. We define $X \gg\!= f$ inductively on the structure of X:*

$$\begin{aligned} \mathtt{op}(p, c) \gg\!= f &= \mathtt{op}(p, \lambda x.\, c(x) \gg\!= f) \\ \eta(x) \gg\!= f &= f(x) \\ \bot \gg\!= f &= \bot \end{aligned}$$

Note that $X \gg\!= f$ is equivalent to $[\![x \gg\!= y]\!]([x \mapsto X, y \mapsto f])$.

Law 3.3.12. *(Associativity of $\gg\!=$)*
Let X be from $[\![\mathcal{F}_E(\alpha)]\!]$, f be a function from $[\![\alpha]\!]$ to $[\![\mathcal{F}_E(\beta)]\!]$ and g be a function from $[\![\beta]\!]$ to $[\![\mathcal{F}_E(\gamma)]\!]$. Then the following equation holds:

$$(X \gg\!= f) \gg\!= g \;=\; X \gg\!= (\lambda x.\, f(x) \gg\!= g)$$

Proof. Proof by induction on the well-founded structure of X:

- $X = \bot$

$$\begin{aligned} (\bot \gg\!= f) \gg\!= g &= \bot \gg\!= g \\ &= \bot \\ &= \bot \gg\!= (\lambda x.\, f(x) \gg\!= g) \end{aligned}$$

- $X = \eta(x)$

$$\begin{aligned} (\eta(x) \gg\!= f) \gg\!= g &= f(x) \gg\!= g \\ &= (\lambda x.\, f(x) \gg\!= g)(x) \\ &= \eta(x) \gg\!= (\lambda x.\, f(x) \gg\!= g) \end{aligned}$$

- $X = \mathtt{op}(p, c)$

$$\begin{aligned}(\mathtt{op}(p,c) \gg\!= f) \gg\!= g &= \mathtt{op}(p, \lambda y.\, c(y) \gg\!= f) \gg\!= g \\ &= \mathtt{op}(p, \lambda y.\, (c(y) \gg\!= f) \gg\!= g) \\ &= \mathtt{op}(p, \lambda y.\, c(y) \gg\!= (\lambda x.\, (f(x) \gg\!= g))) \\ &= \mathtt{op}(p, c) \gg\!= (\lambda x.\, f(x) \gg\!= g)\end{aligned}$$

□

Law 3.3.13. *(Left identity for $\gg\!=$)*
Let $\eta(x)$ be from $[\![\mathcal{F}_E(\alpha)]\!]$ and f be a function from $[\![\alpha]\!]$ to $[\![\mathcal{F}_E(\beta)]\!]$. Then the following holds:

$$\eta(x) \gg\!= f = f(x)$$

Proof. Follows immediately from the definition of $\gg\!=$ (Definition 3.3.11). □

Law 3.3.14. *(Right identity for $\gg\!=$)*
Let X be from $[\![\mathcal{F}_E(\alpha)]\!]$. Then the following holds:

$$X \gg\!= (\lambda x.\, \eta(x)) = X$$

Proof. By induction on the structure of X:

- $X = \bot$

$$\bot \gg\!= (\lambda x.\, \eta(x)) = \bot$$

- $X = \eta(x)$

$$\begin{aligned}\eta(x) \gg\!= (\lambda x.\, \eta(x)) &= (\lambda x.\, \eta(x))(x) \\ &= \eta(x)\end{aligned}$$

- $X = \mathtt{op}(p, c)$

$$\begin{aligned}\mathtt{op}(p,c) \gg\!= (\lambda x.\, \eta(x)) &= \mathtt{op}(p, \lambda y.\, c(y) \gg\!= (\lambda x.\, \eta(x))) \\ &= \mathtt{op}(p, \lambda y.\, c(y)) \\ &= \mathtt{op}(p, c)\end{aligned}$$

□

3.3.4 Functor

We will start by showing what is a functor and in what way do our computation types form one.

Definition 3.3.15. *A **functor** is a homomorphic mapping from one category to another. A functor F from a category a $\mathcal{C}$ to a category $\mathcal{D}$ consists of:*

- *for every object A in $\mathcal{C}$, an object $F(A)$ in $\mathcal{D}$*
- *for every arrow $f : A \to B$ in $\mathcal{C}$, an arrow $F(f) : F(A) \to F(B)$ in $\mathcal{D}$*
- *the following equations hold:*

$$F(f \circ g) = F(f) \circ F(g) \qquad \textit{(Composition)} \tag{3.4}$$

$$F(\mathrm{id}_A) = \mathrm{id}_{F(A)} \qquad \textit{(Identity)} \tag{3.5}$$

Definition 3.3.16. *An **endofunctor** is a functor from some category $\mathcal{C}$ to the same category.*

For every effect signature E, we will show that $\mathcal{F}_E$ is an endofunctor on the $(\!|\lambda|\!)$ category.

objects: for every type α, we have a type $\mathcal{F}_E(\alpha)$

arrows: for every function $f : [\![\alpha]\!] \to [\![\beta]\!]$,
we have a function $\mathcal{F}_E(f) = \lambda X.\, X \gg\!= (\lambda x.\, \eta(f(x))) : [\![\mathcal{F}_E(\alpha)]\!] \to [\![\mathcal{F}_E(\beta)]\!]$

We also need to prove that $\mathcal{F}_E$ satisfies the necesary laws.
Composition (3.4):

$$
\begin{aligned}
\mathcal{F}_E(f) \circ \mathcal{F}_E(g) &= \lambda X.\, \mathcal{F}_E(f)(\mathcal{F}_E(g)(X)) && (1)\\
&= \lambda X.\, \mathcal{F}_E(f)((\lambda Z.\, Z \gg\!= (\lambda x.\, \eta(g(x))))(X)) && (2)\\
&= \lambda X.\, \mathcal{F}_E(f)(X \gg\!= (\lambda x.\, \eta(g(x)))) && (3)\\
&= \lambda X.\, (\lambda Z.\, Z \gg\!= (\lambda y.\, \eta(f(y))))(X \gg\!= (\lambda x.\, \eta(g(x)))) && (4)\\
&= \lambda X.\, (X \gg\!= (\lambda x.\, \eta(g(x)))) \gg\!= (\lambda y.\, \eta(f(y))) && (5)\\
&= \lambda X.\, X \gg\!= (\lambda z.\, (\lambda x.\, \eta(g(x)))(z) \gg\!= (\lambda y.\, \eta(f(y)))) && (6)\\
&= \lambda X.\, X \gg\!= (\lambda z.\, \eta(g(z)) \gg\!= (\lambda y.\, \eta(f(y)))) && (7)\\
&= \lambda X.\, X \gg\!= (\lambda z.\, (\lambda y.\, \eta(f(y)))(g(z))) && (8)\\
&= \lambda X.\, X \gg\!= (\lambda z.\, \eta(f(g(z)))) && (9)\\
&= \lambda X.\, X \gg\!= (\lambda z.\, \eta((f \circ g)(z))) && (10)\\
&= \mathcal{F}_E(f \circ g) && (11)
\end{aligned}
$$

Throughout most of the proof, we rely only on the definitions of function composition and the $\mathcal{F}_E(f)$ function lifter. On Line 6 though, we make use of the associativity law of $\gg\!=$ (3.3.12).
Identities (3.5):

$$
\begin{aligned}
\mathcal{F}_E(\mathrm{id}_\alpha) &= \lambda X.\, X \gg\!= (\lambda x.\, \eta(\mathrm{id}_\alpha(x))) && (1)\\
&= \lambda X.\, X \gg\!= (\lambda x.\, \eta(x)) && (2)\\
&= \lambda X.\, X && (3)\\
&= \mathrm{id}_{\mathcal{F}_E(\alpha)} && (4)
\end{aligned}
$$

Here, we make use of the right identity law of $\gg\!=$ (3.3.14) on Line 3.
And so we have a functor $\mathcal{F}_E$. Its laws let us think of computations of type $\mathcal{F}_E(\alpha)$ as containing values of type α over which we can map functions the same way one would map a function over a list, satisfying the same basic laws.
We also note that $(\!|\lambda|\!)$ already has syntax for this kind of mapping: $\mathcal{F}_E(f)(X) = [\![x \cdot\!\gg y]\!]([x \mapsto f, y \mapsto X])$ where $\cdot\!\gg$ is the operator introduced in 1.6.1.

3.3.5 Applicative Functor

The two remaining structures, applicative functors and monads, both have two popular and slightly different presentations. One is in terms of natural transformations and is very common in category theory, whereas the other is given in terms of combinators and is preferred in functional programming. Since we will not be using category theory in the rest of this thesis and we will work with lots of combinators, we will focus on the presentation favored in functional programming.

Definition 3.3.17. *An **applicative functor** [91] is a functor F alongside with two combinators,* $\mathsf{pure} : \alpha \to F(\alpha)$ *and* $\circledast : F(\alpha \to \beta) \to F(\alpha) \to F(\beta)$[46] *polymorphic in α and β.*[47] *Furthermore, the following laws must hold:*[48]

$$\mathsf{pure}(\mathrm{id}) \circledast u = u \qquad \textit{(Identity)} \tag{3.6}$$

$$\mathsf{pure}(\circ) \circledast u \circledast v \circledast w = u \circledast (v \circledast w) \qquad \textit{(Composition)} \tag{3.7}$$

$$\mathsf{pure}(f) \circledast \mathsf{pure}(x) = \mathsf{pure}(f(x)) \qquad \textit{(Homomorphism)} \tag{3.8}$$

$$u \circledast \mathsf{pure}(x) = \mathsf{pure}(\lambda f.\, f(x)) \circledast u \qquad \textit{(Interchange)} \tag{3.9}$$

The intuition is that the functor F adds some notion of computational effect. $F(\alpha)$ is the type of computations having that effect and producing a value of type α. The pure combinator injects a value into the domain of computations without doing anything on the effect level. The $\circledast$ operator is then meant as a sequencing of two computations: one yielding a function and the other yielding its argument.

The identity law makes sure that pure is indeed pure and does not add any meaningful effect: the effect of pure must in no way tamper with the effect of u. The composition law tells us that the $\circledast$ operator must be associative: at the level of effects, there is some monoidal structure, where order matters but bracketing does not. The homomorphism law makes sure that $\circledast$ does actually perform function application. Finally, the interchange law guarantees that $\circledast$ is not one-sided: the effect of u is respected, no matter if it occurs to the left or to the right of $\circledast$.

To show that $\mathcal{F}_E$ is an applicative functor, we will need to define pure and $\circledast$ and prove the four applicative functor laws.

$$\begin{aligned} \mathsf{pure}(x) &= \eta(x) \\ F \circledast X &= F \gg\!= (\lambda f.\, X \gg\!= (\lambda x.\, \eta(f(x)))) \end{aligned}$$

Now, we prove the laws.

Identity (3.6):

$$\begin{aligned} \mathsf{pure}(\mathrm{id}) \circledast u &= \eta(\mathrm{id}) \circledast u && (1) \\ &= \eta(\mathrm{id}) \gg\!= (\lambda f.\, u \gg\!= (\lambda x.\, \eta(f(x)))) && (2) \\ &= (\lambda f.\, u \gg\!= (\lambda x.\, \eta(f(x))))(\mathrm{id}) && (3) \\ &= u \gg\!= (\lambda x.\, \eta(\mathrm{id}(x))) && (4) \\ &= u \gg\!= (\lambda x.\, \eta(x)) && (5) \\ &= u && (6) \end{aligned}$$

We make use of the left and right identities of $\gg\!=$ on Lines 3 and 6, respectively.

[46] $\circledast$ is an infix operator that associates to the left, same as application (which it is based on).
[47] This means that the combinators should not have different definitions for different instances of the type variables α and β.
[48] In the Composition Law (3.7), $(\circ)$ is the function composition combinator.

Composition (3.7):

$$
\begin{aligned}
\mathsf{pure}(\circ) \circledast u \circledast v \circledast w &= \mathsf{pure}(\lambda fgx.\, f(g(x))) \circledast u \circledast v \circledast w && (1)\\
&= \eta(\lambda fgx.\, f(g(x))) \circledast u \circledast v \circledast w && (2)\\
&= (\eta(\lambda fgx.\, f(g(x))) \gg\!= (\lambda f.\, u \gg\!= (\lambda x.\, \eta(f(x))))) \circledast v \circledast w && (3)\\
&= ((\lambda f.\, u \gg\!= (\lambda u'.\, \eta(f(u'))))(\lambda fgx.\, f(g(x)))) \circledast v \circledast w && (4)\\
&= (u \gg\!= (\lambda u'.\, \eta((\lambda fgx.\, f(g(x)))(u')))) \circledast v \circledast w && (5)\\
&= (u \gg\!= (\lambda u'.\, \eta(\lambda gx.\, u'(g(x))))) \circledast v \circledast w && (6)\\
&= ((u \gg\!= (\lambda u'.\, \eta(\lambda gx.\, u'(g(x))))) \gg\!= (\lambda f.\, v \gg\!= (\lambda v'.\, \eta(f(v'))))) \circledast w && (7)\\
&= (u \gg\!= (\lambda u'.\, \eta(\lambda gx.\, u'(g(x))) \gg\!= (\lambda f.\, v \gg\!= (\lambda v'.\, \eta(f(v')))))) \circledast w && (8)\\
&= (u \gg\!= (\lambda u'.\, (\lambda f.\, v \gg\!= (\lambda v'.\, \eta(f(v'))))(\lambda gx.\, u'(g(x))))) \circledast w && (9)\\
&= (u \gg\!= (\lambda u'.\, v \gg\!= (\lambda v'.\, \eta((\lambda gx.\, u'(g(x)))(v'))))) \circledast w && (10)\\
&= (u \gg\!= (\lambda u'.\, v \gg\!= (\lambda v'.\, \eta(\lambda x.\, u'(v'(x)))))) \circledast w && (11)\\
&= (u \gg\!= (\lambda u'.\, v \gg\!= (\lambda v'.\, \eta(\lambda x.\, u'(v'(x)))))) \gg\!= (\lambda f.\, w \gg\!= (\lambda w'.\, \eta(f(w')))) && (12)\\
&= u \gg\!= (\lambda u'.\, v \gg\!= (\lambda v'.\, \eta(\lambda x.\, u'(v'(x)))) \gg\!= (\lambda f.\, w \gg\!= (\lambda w'.\, \eta(f(w'))))) && (13)\\
&= u \gg\!= (\lambda u'.\, v \gg\!= (\lambda v'.\, \eta(\lambda x.\, u'(v'(x))) \gg\!= (\lambda f.\, w \gg\!= (\lambda w'.\, \eta(f(w')))))) && (14)\\
&= u \gg\!= (\lambda u'.\, v \gg\!= (\lambda v'.\, (\lambda f.\, w \gg\!= (\lambda w'.\, \eta(f(w'))))(\lambda x.\, u'(v'(x))))) && (15)\\
&= u \gg\!= (\lambda u'.\, v \gg\!= (\lambda v'.\, w \gg\!= (\lambda w'.\, \eta((\lambda x.\, u'(v'(x)))(w'))))) && (16)\\
&= u \gg\!= (\lambda u'.\, v \gg\!= (\lambda v'.\, w \gg\!= (\lambda w'.\, \eta(u'(v'(w'))))))) && (17)\\
&= u \gg\!= (\lambda u'.\, v \gg\!= (\lambda v'.\, w \gg\!= (\lambda w'.\, (\lambda x.\, \eta(u'(x)))(v'(w'))))) && (18)\\
&= u \gg\!= (\lambda u'.\, v \gg\!= (\lambda v'.\, w \gg\!= (\lambda w'.\, \eta(v'(w')) \gg\!= (\lambda x.\, \eta(u'(x)))))) && (19)\\
&= u \gg\!= (\lambda u'.\, v \gg\!= (\lambda v'.\, (w \gg\!= (\lambda w'.\, \eta(v'(w')))) \gg\!= (\lambda x.\, \eta(u'(x))))) && (20)\\
&= u \gg\!= (\lambda u'.\, (v \gg\!= (\lambda v'.\, w \gg\!= (\lambda w'.\, \eta(v'(w'))))) \gg\!= (\lambda x.\, \eta(u'(x)))) && (21)\\
&= u \gg\!= (\lambda u'.\, (v \circledast w) \gg\!= (\lambda x.\, \eta(u'(x)))) && (22)\\
&= u \circledast (v \circledast w) && (23)
\end{aligned}
$$

This law relies heavily on the associativity of $\gg\!=$ (3.3.12). At the beginning, the expression is associated to the left: $((\mathsf{pure}(\circ) \circledast u) \circledast v) \circledast w$. And at the end, it associated to the right: $u \circledast (v \circledast w)$. In the first part of the proof, we use left identity (Lines 4, 9 and 15) and associativity (Lines 8, 13 and 14). We reach a normal form on Line 17 and start expanding the term again. We expand using left identity on Line 19 and then we reassociate using associativity on Lines 20 and 21. The rest of the equalities is due to the definitions of pure and $\circledast$ and β reduction.

Homomorphism (3.8):

$$
\begin{aligned}
\mathsf{pure}(f) \circledast \mathsf{pure}(x) &= \eta(f) \circledast \eta(x) && (1)\\
&= \eta(f) \gg\!= (\lambda f'.\, \eta(x) \gg\!= (\lambda x'.\, \eta(f'(x')))) && (2)\\
&= (\lambda f'.\, \eta(x) \gg\!= (\lambda x'.\, \eta(f'(x'))))(f) && (3)\\
&= \eta(x) \gg\!= (\lambda x'.\, \eta(f(x'))) && (4)\\
&= (\lambda x'.\, \eta(f(x')))(x) && (5)\\
&= \eta(f(x)) && (6)\\
&= \mathsf{pure}(f(x)) && (7)
\end{aligned}
$$

This proof is very direct. We just have to normalize the term using the left identity of $\gg\!=$ (3.3.13).

Interchange (3.9):

$$
\begin{aligned}
\mathsf{pure}(\lambda f.\, f(x)) \circledast u &= \eta(\lambda f.\, f(x)) \circledast u && (1)\\
&= \eta(\lambda f.\, f(x)) \gg\!= (\lambda f'.\, u \gg\!= (\lambda u'.\, \eta(f'(u')))) && (2)\\
&= (\lambda f'.\, u \gg\!= (\lambda u'.\, \eta(f'(u'))))(\lambda f.\, f(x)) && (3)\\
&= u \gg\!= (\lambda u'.\, \eta((\lambda f.\, f(x))(u'))) && (4)\\
&= u \gg\!= (\lambda u'.\, \eta(u'(x))) && (5)\\
&= u \gg\!= (\lambda u'.\, (\lambda x'.\, \eta(u'(x')))(x)) && (6)\\
&= u \gg\!= (\lambda u'.\, \eta(x) \gg\!= (\lambda x'.\, \eta(u'(x')))) && (7)\\
&= u \circledast \eta(x) && (8)\\
&= u \circledast \mathsf{pure}(x) && (9)
\end{aligned}
$$

We start with the law's right-hand side and normalize it on Line 5. Then we do some expansions (left identity on Line 7) to get the desired form.

And there we are, we now have an applicative functor. Applicative functors are very similar to monads, which we will cover in the next part. The trade-off between the two is in terms of expressivity vs composability. When we have two applicative functors, we can compose them and recover pure and $\circledast$ such that they both satisfy the necessary laws. However, the same is as not easy with monads and one has to go a level higher and compose monad transformers. On the other hand, when chaining two computations within a monad, we can use an operator which lets the effects of the second computation depend on the result of the first, whereas when chaining two computations within an applicative functor, the effects of both computations must be independent.

Oleg Kiselyov explores the use of applicative functors for natural language semantics in his recent Applicative Abstract Categorial Grammars [68, 69]. The composability of applicative functors facilitates the combination, which is the objective of our method as well. However, partly in order to compensate for the limited expressivity of applicative functors, several interpretation passes are required until the logical form of a sentence is constructed.

The two combinators that make up an applicative functor are accessible in $(\!|\lambda|\!)$. The pure operator is the function η and $\circledast$ is the combinator $\ll\cdot\gg$ introduced in 1.6.1.

3.3.6 Monad

Definition 3.3.18. *A **monad** is a functor F and two combinators, $\eta : \alpha \to F(\alpha)$ and $\gg\!= : F(\alpha) \to (\alpha \to F(\beta)) \to F(\beta)$, polymorphic in α and β. These objects must also satisfy the following laws:*

$$
(X \gg\!= f) \gg\!= g = X \gg\!= (\lambda x.\, f(x) \gg\!= g) \qquad \textit{(Associativity)} \qquad (3.10)
$$

$$
\eta(x) \gg\!= f = f(x) \qquad \textit{(Left identity)} \qquad (3.11)
$$

$$
X \gg\!= \eta = X \qquad \textit{(Right identity)} \qquad (3.12)
$$

To understand why the laws look the way they do, we will consider functions of type $\alpha \to F(\beta)$. These are the kinds of functions one might use to model call-by-value [95, 94]: we take a value α and then yield some computation $F(\beta)$. Now assume we would use this type of functions to model procedures of input type α and output type β and we would want these procedures to form a category. For every type α, we would like an identity procedure with input type and output type α, therefore a function of type $\alpha \to F(\alpha)$. The polymorphic η combinator will be this identity procedure. We would also like to be able to compose a procedure from α to β with a procedure from β to γ, i.e. compose functions of types $\alpha \to F(\beta)$ and $\beta \to F(\gamma)$. When composing $f : \alpha \to F(\beta)$ with $g : \beta \to F(\gamma)$, we run into the problem of having some $f(x) : F(\beta)$ and $g : \beta \to F(\gamma)$ that we cannot compose. This is where $\gg\!=$ comes in and composes these two values for us. Let $f >\!\!=\!\!> g = \lambda x.\, f(x) \gg\!= g$ be the resulting composition operator. In order for this structure to be a category, it needs to satisfy the following:

$$(f \gtrdot\!\!\Rrightarrow g) \gtrdot\!\!\Rrightarrow h = f \gtrdot\!\!\Rrightarrow (g \gtrdot\!\!\Rrightarrow h)$$
$$\eta \gtrdot\!\!\Rrightarrow f = f$$
$$f \gtrdot\!\!\Rrightarrow \eta = f$$

By taking f to be the constant function that returns X, we end up with the laws of the monad. Conversely, with the principle of extensionality, we can derive these laws from the monad laws. Therefore $\langle F, \eta, \gg\!= \rangle$ forms a monad whenever the derived $\gtrdot\!\!\Rrightarrow$ and η form a category. This kind of category is called a *Kleisli category* and the particular presentation of a monad that we have given here is known as a *Kleisli triple*.

To prove that $\mathcal{F}_E$ is a monad will be trivial: we have already done so! The η combinator is of course our η and $\gg\!=$ is our $\gg\!=$. The three laws that we need to verify are three laws that we have introduced in 3.3.3 and have been using throughout this section.

Monads have been introduced to natural language semantics by Chung-chieh Shan in 2002 [113]. Since then, they have seen occasional use, mostly to handle dynamics without burdening the semantics with context/continuation passing [128, 25], but also other phenomena such as conventional implicature/opacity [51, 48, 49]. The challenge of combining different phenomena which rely on different monads has been tackled from two angles: using distributive laws for monads [50] and using monad transformers [27, 13].

In the $(\!|\lambda|\!)$ calculus, the monadic operations η and $\gg\!=$ are available as the η constructor and the $\gg\!=$ combinator introduced in 1.6.1.

3.3.7 Free Monad

A free monad is a construction of a monad such that the resulting monad satisfies the monad laws but does not satisfy anything more than that. This is similar to the idea of a free monoid generated by some set A. If we have two expressions from a monoid, we know that the operation is associative and so the grouping does not matter. The only thing that matters is which elements are multiplied and in which order. We can therefore think of the elements of this monoid as lists of values from A. The free monad is a similar construction, but instead of building a monoid out of a set, it builds a monad out of a functor.

Let S be some functor, we define the monad $\langle F, \eta, \gg\!= \rangle$ where:

- $F(\gamma) = \gamma + S(F(\gamma))$[49]
- $\eta(x) = \mathrm{inl}(x)$
- $X \gg\!= f = \textbf{case } X \textbf{ of } \{\mathrm{inl}(x) \to f(x);\ \mathrm{inr}(s) \to \mathrm{inr}(S(\lambda X'.\, X' \gg\!= f)(s))\}$

The functor S correspond to our effect signature. For $\mathtt{op} : \alpha \rightarrowtail \beta$, we can define a functor $S_{\mathtt{op}}(\gamma) = \alpha \times \gamma^{\beta}$. This represents providing a value of type α and waiting for an answer of type β before continuing with γ. By taking the free monad of this functor, we end up with a monad where after each operation, we continue with a new computation of the same kind until we yield a γ, $F(\gamma) = \gamma + \alpha \times F(\gamma)^{\beta}$.

If we want to offer more operations with different input and output types, we can take the coproduct of several functors. For an effect signature E, $S_E(\gamma) = \sum_{\mathtt{op} \in E} S_{\mathtt{op}}(\gamma)$. The resulting free monad then looks like $F(\gamma) = \gamma + \sum_{\mathtt{op}:\alpha \rightarrowtail \beta \in E} \alpha \times F(\gamma)^{\beta}$. If we replace sums with disjoint unions and types with their interpretations, you almost get the kind of set in which we interpret our computation types in 3.3.1. Importantly, the η and $\gg\!=$ are the same as the ones of the free monad given above. The only difference between the free monad and the monad used in our interpretation are the occurrences of $\perp$. The monad that we used is actually a combination of the $F(X) = X_{\perp}$ partiality monad and a free monad.

Free monads are another solution to the combination of different effects within a single calculus, hitherto unexplored in its application to natural language semantics. A very early appearance of this technique dates back to 1994 [24]. It has recently gained prominence with the work on extensible effects and effect handlers [71, 16, 63, 22, 104, 110].

[49] The plus is a coproduct, which in the case of our category of sets corresponds to disjoint union. We will work with coproducts using a similar syntax to the one that we have given to sums in 1.5.

3.4 Confluence

The object of our study during this section will be the proof of the *confluence property* of $(\!| \lambda |\!)$. Informally, it means that a single term cannot reduce to two or more different results. Together with the termination from Section 3.5, this will give us the property that every term yields exactly one result and does so in a finite amount of steps (a property known as *strong normalization*). Confluence also gives us a strong tool to prove an inequality on terms. If two terms reduce to different normal forms, confluence guarantees us that they are not convertible.

Definition 3.4.1. *A reduction relution $\to$ on a set A is said to be* ***confluent*** *whenever for each $a, b, c \in A$ such that $a \to b$ and $a \to c$ there is a $d \in A$ such that $b \twoheadrightarrow d$ and $c \twoheadrightarrow d$.*

Proofs of this property are often mechanical and follow the same pattern. Our strategy will be to reuse a general result which applies one such proof for a general class of rewriting systems. Our rewriting system is a system of reductions on terms and the reductions have side conditions concerning the binding of free variables. A good fit for this kind of system are the Combinatory Reduction Systems (CRSs) of Klop [76].

The main result about CRSs that we will make use of is the following (Corollary 13.6 in [76]).

Theorem 3.4.2. ***Confluence of orthogonal CRSs***
Every orthogonal CRS is confluent.

We will model $(\!| \lambda |\!)$ as a CRS. However, η-reduction will deny us orthogonality. We will therefore first prove confluence of $(\!| \lambda |\!)$ without η-reduction and then we will manually show that confluence is preserved on adding η-reduction back.

Notation 3.4.3. *The* ***intensional*** $(\!| \lambda |\!)$ ***calculus*** $(\!| \lambda |\!)_{-\eta}$ *is the* $(\!| \lambda |\!)$ *calculus without the η-reduction rule.*

The rest of this section will go like this:

- CRS: a formalism for higher-order rewriting (3.4.1)
- $(\!| \lambda |\!)$ is a CRS (3.4.2)
- Klop et al [93]: Every orthogonal CRS is confluent (3.4.3)
  - $(\!| \lambda |\!)_{-\eta}$ is an orthogonal CRS $\Rightarrow$ $(\!| \lambda |\!)_{-\eta}$ is confluent (Lemma 3.4.12)
  - η is an orthogonal CRS $\Rightarrow$ η is confluent (Lemma 3.4.13)
- $(\!| \lambda |\!)_{-\eta} + \eta$ is confluent (3.4.4, Theorem 3.4.18)
  - because $(\!| \lambda |\!)_{-\eta}$ and η commute (Lemma 3.4.17)

3.4.1 Combinatory Reduction Systems

A Combinatory Reduction System is defined by an alphabet and a set of rewriting rules. We will first cover the alphabet.

Definition 3.4.4. *A* ***CRS alphabet*** *consists of:*

- *a set* Var *of* variables *(written lower-case as x, y, z,...)*
- *a set* MVar *of* metavariables *(written upper-case as M, N, ...), each with is own arity*
- *a set of* function symbols, *each with its own arity*

Let us sketch the difference between the variables in Var and the metavariables in MVar. The variables in Var are the variables of the object-level terms, in our case it will be the variables of $(\!| \lambda |\!)$. The variables in MVar are the metavariables that will occur in our reduction rules and which we will have to instantiate in order to derive specific application of those rules. In other words, the variables in Var are there to express the binding structure within the terms being reduced and the metavariables in MVar are there to stand in for specific terms when applying a reduction rule.

Definition 3.4.5. *The **metaterms** of a CRS are given inductively:*

- *variables are metaterms*
- *if t is a metaterm and x a variable, then $[x]t$ is a metaterm called **abstraction***
- *if F is an n-ary function symbol and t_1,...,t_n are metaterms, then $F(t_1, \ldots, t_n)$ is a metaterm*
- *if M is an n-ary metavariable and t_1,...,t_n are metaterms, then $M(t_1, \ldots, t_n)$ is a metaterm*

Definition 3.4.6. *The **terms** of a CRS are its metaterms which do not contain any metavariables.*

To finish the formal introduction of CRSs, we give the definition of a CRS reduction rule.

Definition 3.4.7. *A **CRS reduction rule** is a pair of metaterms $s \to t$ such that:*

- *s and t are both closed, i.e. all variables are bound using the $[\_]\_$ abstraction binder*
- *s is of the form $F(t_1, \ldots, t_n)$*
- *all the metavariables that occur in t also occur in s*
- *any metavariable M that occurs in s only occurs in the form $M(x_1, \ldots, x_k)$, where x_i are pairwise distinct variables*

Definition 3.4.8. *A **Combinatory Reduction System (CRS)** is a pair of a CRS alphabet and a set of CRS reduction rules.*

We will only sketch the way that a CRS gives rise to a reduction relation and we will direct curious readers to Sections 11 and 12 of [76].

When we instantiate the metavariables in a CRS rule, we use a *valuation* that assigns to every n-ary metavariable a term with holes labelled from 1 to n. The instantiation of $M(t_1, \ldots, t_n)$ then replaces the metavariable M using the valuation and then fills the holes labelled $1, \ldots, n$ with the terms $t_1, \ldots, t_n$ respectively.

The crucial detail is that in a particular context, a metavariable can only be instantiated with terms M that do not contain any free variables bound in that context. This means that for the instantiation of M to contain a variable bound in its context, M must explicitly take that variable as an argument. All other variables not explicitly declared can therefore be safely assumed to not occur freely within M.

Consider the following examples of β and η-reduction.

$$(\lambda x.\, M(x))\, N \to M(N)$$
$$\lambda x.\, N\, x \to N$$

More formally written as:

$$@(\lambda([x]M(x)), N) \to M(N)$$
$$\lambda([x]@(N, x)) \to N$$

where λ is a unary function symbol and @ is a binary function symbol. In both of the versions, M is a unary metavariable and N is a nullary metavariable. In the rule for β-reduction, we can observe how the idea of instantiating metavariables by terms with holes lets us express the same idea for which we had to introduce the meta-level operation of substitution. In the rule for η-reduction, we see that N appears in a context where x is bound but it does not have x as one of its arguments. Therefore, it will be impossible to instantiate N in such a way that it contains a free occurrence of x. In both of those rules, we were able to get rid of meta-level operations (substitution) and conditions ($x \notin FV(N)$) and have them both implemented by the formalism itself.

3.4.2 $(\!|\,\lambda\,|\!)$ as a CRS

We will now see how to rephrase the reduction rules of $(\!|\,\lambda\,|\!)$ in order to fit in to the CRS framework. We have already seen how to translate the β and η rules in the previous subsection. The next rules to address are the rules defining the semantics of the $(\!|\ |\!)$ handlers.

We will repeat the rules for handlers to make the issue at hand clear.

$$(\!|\,(\mathtt{op}_i\colon M_i)_{i\in I},\ \eta\colon M_\eta\,|\!)\,(\eta\, N) \to M_\eta\, N \qquad \text{rule } (\!|\,\eta\,|\!)$$

$$(\!|\,(\mathtt{op}_i\colon M_i)_{i\in I},\ \eta\colon M_\eta\,|\!)\,(\mathtt{op}_j\, N_\mathrm{p}\,(\lambda x.\, N_\mathrm{c})) \to M_j\, N_\mathrm{p}\,(\lambda x.\,(\!|\,(\mathtt{op}_i\colon M_i)_{i\in I},\ \eta\colon M_\eta\,|\!)\, N_\mathrm{c}) \qquad \text{rule } (\!|\,\mathtt{op}\,|\!) \text{ where } j\in I \text{ and } x\notin \mathrm{FV}((M_i)_{i\in I}, M_\eta)$$

$$(\!|\,(\mathtt{op}_i\colon M_i)_{i\in I},\ \eta\colon M_\eta\,|\!)\,(\mathtt{op}_j\, N_\mathrm{p}\,(\lambda x.\, N_\mathrm{c})) \to \mathtt{op}_j\, N_\mathrm{p}\,(\lambda x.\,(\!|\,(\mathtt{op}_i\colon M_i)_{i\in I},\ \eta\colon M_\eta\,|\!)\, N_\mathrm{c}) \qquad \text{rule } (\!|\,\mathtt{op}'\,|\!) \text{ where } j\notin I \text{ and } x\notin \mathrm{FV}((M_i)_{i\in I}, M_\eta)$$

The syntax of CRSs does not allow us to use the $(\mathtt{op}_i\colon M_i)_{i\in I}$ notation nor capture the $j\in I$ or $j\notin I$ conditions. The symbols $\mathtt{op}_i$ are problematic as well, since technically, they are not concrete $(\!|\,\lambda\,|\!)$ syntax but metavariables standing in for operation symbols.

We do away with all of the above problems by expanding these meta-notations and adding a separate rule for every possible instantiation of the schema. This means that for each sequence of distinct operation symbols $\mathtt{op}_1$,…,$\mathtt{op}_n$, we end up with:

- a special rewriting rule $(\!|\,\mathtt{op}_1\colon M_1,\ldots,\mathtt{op}_n\colon M_n,\ \eta\colon M_\eta\,|\!)\,(\eta\, N) \to M_\eta\, N$
- for every $1\le i\le n$, a special rewriting rule
  $(\!|\,\mathtt{op}_1\colon M_1,\ldots,\mathtt{op}_n\colon M_n,\ \eta\colon M_\eta\,|\!)\,(\mathtt{op}_i\, N_\mathrm{p}\,(\lambda x.\, N_\mathrm{c}(x)))$
  $\to M_i\, N_\mathrm{p}\,(\lambda x.\,(\!|\,\mathtt{op}_1\colon M_1,\ldots,\mathtt{op}_n\colon M_n,\ \eta\colon M_\eta\,|\!)\, N_\mathrm{c}(x))$
- for every $\mathtt{op}'\in\mathcal{E}\setminus\{\mathtt{op}_i | 1\le i\le n\}$, a special rewriting rule
  $(\!|\,\mathtt{op}_1\colon M_1,\ldots,\mathtt{op}_n\colon M_n,\ \eta\colon M_\eta\,|\!)\,(\mathtt{op}'\, N_\mathrm{p}\,(\lambda x.\, N_\mathrm{c}(x)))$
  $\to \mathtt{op}'\, N_\mathrm{p}\,(\lambda x.\,(\!|\,\mathtt{op}_1\colon M_1,\ldots,\mathtt{op}_n\colon M_n,\ \eta\colon M_\eta\,|\!)\, N_\mathrm{c}(x))$

The rule for the cherry $\downarrow$ extraction operator is already in CRS form, so all we have to do is address the rules for the $\mathcal{C}$ operator. We present them side-by-side in their original form and in CRS-style.

Original:

$$\mathcal{C}\,(\lambda x.\,\eta\, M) \to \eta\,(\lambda x.\, M)$$
$$\mathcal{C}\,(\lambda x.\,\mathtt{op}\, M_\mathrm{p}\,(\lambda y.\, M_\mathrm{c})) \to \mathtt{op}\, M_\mathrm{p}\,(\lambda y.\,\mathcal{C}\,(\lambda x.\, M_\mathrm{c})) \quad \text{where } x\notin \mathrm{FV}(M_\mathrm{p})$$

CRS-style:

$$\mathcal{C}\,(\lambda x.\,\eta\,(M(x))) \to \eta\,(\lambda x.\, M(x))$$
$$\mathcal{C}\,(\lambda x.\,\mathtt{op}\, M_\mathrm{p}\,(\lambda y.\, M_\mathrm{c}(x,y))) \to \mathtt{op}\, M_\mathrm{p}\,(\lambda y.\,\mathcal{C}\,(\lambda x.\, M_\mathrm{c}(x,y)))$$

We can see that the only difference is to replace "simple" metavariables M, M_p and M_c with their higher-order versions: the unary M, nullary M_p and binary M_c. We see that every CRS metavariable is applied to the variables in its scope, except for M_p, which thus loses access to the variable x. This way, the condition that x must not appear free in M_p is now encoded directly in the reduction rule itself.

In 3.4.1, we have said that a CRS is formed by a set of reduction rules and by an alphabet. We have already seen all of the rules of our CRS (β and η were given at the end of 3.4.1 and the $\downarrow$ rule is the same as the original one in 1.4). In order to have a complete definition, all that remains is to identify the alphabet.

The set of variables Var is exactly the set of variables $\mathcal{X}$ used in the definition of $(\!|\,\lambda\,|\!)$. The set of metavariables $MVar$ consists of the unary M, nullary N, nullary N_p, unary N_c, nullary M_p, binary M_c, nullary M_i and nullary M_η. The set of function symbols is composed of the following:

- the binary symbol @ for function application
- the unary symbol λ for function abstraction
- a nullary symbol for every constant in the signature Σ
- the unary symbol η for the injection operator
- a binary symbol $\mathtt{op}$ for every $\mathtt{op} \in \mathcal{E}$
- a $(n{+}2)$-ary symbol $(\!|\, \mathtt{op}_1\!:\_,\ \ldots,\ \mathtt{op}_n\!:\_,\ \eta\!:\_ \,|\!)\ \_)$ for every sequence $\mathtt{op}_1, \ldots, \mathtt{op}_n$ of distinct symbols from $\mathcal{E}$ of length n
- the unary symbol $\downarrow_{\circ}$ for the extraction operator
- the unary symbol $\mathcal{C}$ for the $\mathcal{C}$ operator

In giving the CRS-style reduction rules above, we have used the "native" syntax of $(\!|\, \lambda \,|\!)$ instead of writing out everything in terms of function symbols. For clarity, we give the rules governing the relationship of the two. We write:

- $@(t, u)$ as $t\, u$
- $\lambda([x]t)$ as $\lambda x.\, t$
- $\eta(t)$ as $\eta\, t$
- $\mathtt{op}(t_{\mathrm{p}}, [x]t_{\mathrm{c}})$ as $\mathtt{op}\, t_{\mathrm{p}}\, (\lambda x.\, t_{\mathrm{c}})$[50]
- $(\!|\, \mathtt{op}_1\!:\_,\ \ldots,\ \mathtt{op}_n\!:\_,\ \eta\!:\_ \,|\!)\ \_)(t_1, \ldots, t_n, t_\eta, u)$ as $(\!|\, \mathtt{op}_1\!: t_1,\ \ldots,\ \mathtt{op}_n\!: t_n,\ \eta\!: t_\eta \,|\!)\, u$
- $\downarrow_{\circ}(t)$ as $\downarrow_{\circ} t$
- $\mathcal{C}(t)$ as $\mathcal{C}\, t$

We have connected the terms of $(\!|\, \lambda \,|\!)$ with CRS terms and we have also expressed all of our reduction rules in terms of CRS reduction rules. As in $(\!|\, \lambda \,|\!)$, CRS then proceeds to take a context closure of this redex-contractum relation. Our translation from $(\!|\, \lambda \,|\!)$ to a CRS also preserves subterms[51] and so we end up constructing the same reduction relation.

3.4.3 Orthogonal CRSs

In order to use Theorem 3.4.2, we need to show that our CRS is orthogonal, so let us start us by looking at what "orthogonal" means in the context of CRSs.

Definition 3.4.9. *A CRS is **orthogonal** if it is non-overlapping and left-linear.*

We will need to satisfy two criteria: no overlaps and left linearity. We will start with the latter.

Definition 3.4.10. *A CRS is **left-linear** if the left-hand sides of all its reduction rules are linear. A CRS metaterm is **linear** if no metavariable occurs twice within it.*

By going through the rules we have given in 3.4.2, we can see at a glance that no rule uses the same metavariable twice in its left-hand side and so our CRS is indeed left-linear.

[50] Note that with this translation, $\mathtt{op}\, t_{\mathrm{p}}\, (\lambda x.\, t_{\mathrm{c}})$ does not contain $\lambda x.\, t_{\mathrm{c}}$ as a subterm. This is the same as in $(\!|\, \lambda \,|\!)$, where the notion of evaluation context (see 1.4) does not identify $\lambda x.\, t_{\mathrm{c}}$, but rather t_{c}, as a subterm of $\mathtt{op}\, t_{\mathrm{p}}\, (\lambda x.\, t_{\mathrm{c}})$. This becomes important in our discussion of confluence since it makes it impossible to make the λ disappear by something like η-reduction.

[51] More precisely, if a is a subterm of b in $(\!|\, \lambda \,|\!)$ then the CRS version of a is a subterm of the CRS version of b. In the other direction, whenever a is a variable or a function-headed term which is a subterm of b in the CRS version of $(\!|\, \lambda \,|\!)$, then the corresponding a in $(\!|\, \lambda \,|\!)$ is a subterm of the corresponding b.

Definition 3.4.11. *A CRS is* ***non-overlapping*** *if:*

- *Let $r = s \to t$ be some reduction rule of the CRS and let $M_1,\ldots,M_n$ be all the metavariables occurring in the left-hand side s. Whenever we can instantiate the metavariables in s such that the resulting term contains a redex for some other rule r', then said redex must be contained in the instantiation of one of the metavariables M_i.*
- *Similarly, whenever we can instantiate the metavariables in s such that the resulting term* properly contains *a redex* for the same rule *r, then that redex as well must be contained in the instantiation of one of the metavariables M_i.*

In simpler words, no left-hand side of any rule can contain bits which look like the top of the left-hand side of some other rule. Let us try and verify this property in $(\!|\,\lambda\,|\!)$:

- The $(\!|\;|\!)$ rules have no overlaps with any of the other rules. Their left-hand sides are constructed only of the $(\!|\;|\!)$ symbols and the op and η constructors. Since there is no reduction rule headed by op and η, they have no overlap with any of the other rules. Furthermore, the three $(\!|\;|\!)$ rules are mutually exclusive, so there is no overlap between themselves.
- The $\downarrow_{\circ}$ rule does not overlap with any of the other neither, since the left-hand side contains only $\downarrow_{\circ}$ and η, and there is no reduction rule headed by η.
- The $\mathcal{C}$ rules are both mutually exclusives, so there is no overlap between the two. However, their left-hand sides are built not only out of $\mathcal{C}$, op and η, but also of λ, for which there is the η-reduction rule. Fortunately, in this case, the $\mathcal{C}$ rules only apply when the λ-abstraction's body is an η expression or an op expression, whereas the η rule applies only when the body is an application expression.[52] Therefore, there is no overlap.

We have established that all the reduction rules in our system are pairwise non-overlapping *except* for β and η. However, these two have a notorious overlap.

We can instantiate the metavariables in the left-hand side of the β rule to get a term which contains an η-redex which shares the λ-abstraction with the β-redex.

$$(\lambda x.\, y\, x)\, z$$

We can also instantiate the metavariables in the left-hand side of the η rule to create a β-redex which shares the application with the η-redex.

$$\lambda x.\, (\lambda z.\, z)\, x$$

Because of these overlaps, the $(\!|\,\lambda\,|\!)$ CRS is therefore *not* orthogonal. However, we can still make good use of Theorem 3.4.2.

Lemma 3.4.12. ***Confluence of*** $(\!|\,\lambda\,|\!)_{-\eta}$
The $(\!|\,\lambda\,|\!)$ reduction system without the η rule is confluent.

Proof. If we exclude the η rule, we have a CRS which is left-linear and also non-overlapping.[53] Therefore, it is orthogonal and thanks to Theorem 3.4.2, also confluent. □

Lemma 3.4.13. ***Confluence of η-reduction***
The reduction system on $(\!|\,\lambda\,|\!)$ terms containing only the η-reduction rule is confluent.[54]

Proof. We have seen that η is a valid left-linear CRS rule. It also does not overlap itself since its left-hand side does not contain any λ subexpression. The CRS consisting of just the η rule is therefore orthogonal and confluent. □

[52] This is not so much a fortunate conincidence but rather a deliberate choice in the design of the calculus. For example, it is one of the reasons why, in $(\!|\,\lambda\,|\!)$, $\eta\, x$ is not decomposed as an application of the built-in function η to x, but is treated as a special form.

[53] We know that β does not overlap any of the other rules. Neither does it overlap itself since its left-hand side does not have an application subexpression.

[54] This also holds for $(\!|\,\lambda\,|\!)$ with sums and products since their rules are left-linear and do not overlap with the $(\!|\,\lambda\,|\!)$ rules.

3.4.4 Putting η Back in $(\!| \lambda |\!)$

We have shown that both $(\!| \lambda |\!)_{-\eta}$ and η are confluent. The reduction relation of the complete $(\!| \lambda |\!)$ calculus is the union of these two reduction relations. Using the Lemma of Hindley-Rosen (1.0.8.(2) in [75]), we can show that this union is confluent by showing that the two reduction relations commute together.

Definition 3.4.14. *Let $\to_1$ and $\to_2$ be two reduction relations on the same set of terms A. $\to_1$ and $\to_2$* ***commute*** *if for every $a, b, c \in A$ such that $a \twoheadrightarrow_1 b$ and $a \twoheadrightarrow_2 c$, there exists a $d \in A$ such that $b \twoheadrightarrow_2 d$ and $c \twoheadrightarrow_1 d$.*

Lemma 3.4.15. ***Lemma of Hindley-Rosen*** *[75]*
Let $\to_1$ and $\to_2$ be two confluent reduction relations on the same set of terms. If $\to_1$ and $\to_2$ commute, then the reduction relation $\to_1 \cup \to_2$ is confluent.

We will not be proving the commutativity directly from the definition. Instead, we will use a lemma due to Hindley (1.0.8.(3) in [75]).

Lemma 3.4.16. *Let $\to_1$ and $\to_2$ be two reduction relations on the same set of terms A. Suppose that whenever there are $a, b, c \in A$ such that $a \to_1 b$ and $a \to_2 c$, there is also some $d \in A$ such that $b \twoheadrightarrow_2 d$ and $c \to_1^= d$ (meaning $c \to_1 d$ or $c = d$). In that case, $\to_1$ commutes with $\to_2$. [75]*

We can use this to prove that $(\!| \lambda |\!)_{-\eta}$ commutes with the η-reduction rule.

Lemma 3.4.17. ***Commutativity of η and $(\!| \lambda |\!)_{-\eta}$***
The reduction relations induced by η and by the rest of the $(\!| \lambda |\!)$ rules commute.

Proof. We will prove this lemma by an appeal to Lemma 3.4.16. Let $\to_\eta$ be the reduction relation induced by the rule η and $\to_{(\!| \lambda |\!)_{-\eta}}$ the reduction relation induced by all the other reduction rules in $(\!| \lambda |\!)$. We need to prove that for all terms a, b and c where $a \to_{(\!| \lambda |\!)_{-\eta}} b$ and $a \to_\eta c$, we have a term d such that $b \twoheadrightarrow_\eta d$ and $c \to^=_{(\!| \lambda |\!)_{-\eta}} d$.

This will turn out to be a routine proof by induction on the structure of the term a. The base cases are trivial since terms without any proper subterms happen to have no redexes in $(\!| \lambda |\!)$ and therefore trivially satisfy the criterion. In the inductive step, we will proceed by analyzing the relative positions of the redexes which led to the reductions $a \to_{(\!| \lambda |\!)_{-\eta}} b$ and $a \to_\eta c$.

- If both reductions occurred within a common subterm of a, i.e. $a = C[a']$, $b = C[b']$ and $c = C[c']$ while at the same time $a' \to_{(\!| \lambda |\!)_{-\eta}} b'$ and $a' \to_\eta c'$, we can use the induction hypothesis for a'. This gives us a d' such that $b' \twoheadrightarrow_\eta d'$ and $c' \to^=_{(\!| \lambda |\!)_{-\eta}} d'$ and therefore we also have $d = C[d']$ with $b \twoheadrightarrow_\eta d$ and $c \to^=_{(\!| \lambda |\!)_{-\eta}} d$.

- If both reductions occurred within non-overlapping subterms of a, i.e. $a = C[a_1, a_2]$, $b = C[b', a_2]$ and $c = C[a_1, c']$ with $a \to_{(\!| \lambda |\!)_{-\eta}} b$ and $a \to_\eta c$: We can take $d = C[b', c']$ since we have $b \twoheadrightarrow_\eta d$ in one step and $c \to^=_{(\!| \lambda |\!)_{-\eta}} d$ in one step too.

- If the redex in $a \to_{(\!| \lambda |\!)_{-\eta}} b$ is the entire term a, but the redex in $a \to_\eta c$ is a proper subterm of a: We will solve this by case analysis on the form of a:

  - If a is an application: Since a is an application and also a $(\!| \lambda |\!)_{-\eta}$-redex, it must match the left-hand side of the β rule, $(\lambda x.\, M(x))\, N$, and b must be $M(N)$.

    * We will first deal with the case when the η-redex which lead to c originated in $M(x)$. In that case $M(x) \to_\eta M'(x)$ and $c = (\lambda x.\, M'(x))\, N$. Our sought-after d is then $M'(N)$, since $c \to^=_{(\!| \lambda |\!)_{-\eta}} d$ via β in one step and $b = M(N) \twoheadrightarrow_\eta d = M'(N)$.
    * Now we get to one of the two interesting cases which necessitated this whole lemma: the overlap between β and η, with β on the top. If the η-redex did not originate in $M(x)$, then the η-redex must be $\lambda x.\, M(x)$. Therefore, $M = T\, x$ and $a = (\lambda x.\, T\, x)\, N$. Performing the η-reduction yields $c = T\, N$. In this case, both b and c are equal to $T\, N$ and so we can choose $T\, N$ as our d.

- If a is any other kind of term: Let $l \to r$ be the rule used in $a \to_{(\!|\lambda|\!)_{-\eta}} b$. Not counting β, which only acts on applications and which we dealt with just above, the rules of $(\!|\lambda|\!)_{-\eta}$ do not overlap with the η rule. This means the η-redex which led to c must lie entirely inside a part of l which corresponds to a metavariable. Let M be that metavariable, then we will decompose l into $L(M)$ and r into $R(M)$. We have $a = L(a')$ for some a', $b = R(a')$ and $c = L(a'')$[55]. Our d will be $R(a'')$ and we have $b = R(a') \twoheadrightarrow_\eta d = R(a'')$ in several steps[56] and $c = L(a'') \to^{=}_{(\!|\lambda|\!)_{-\eta}} d = R(a'')$ in one step of $l \to r$.

- If the redex in $a \to_\eta c$ is the entire term a, but the redex in $a \to_{(\!|\lambda|\!)_{-\eta}} b$ is a proper subterm of a: In this case, a must be an abstraction that matches the left-hand side of the η rule, i.e. $a = \lambda x.\, N\, x$. Also, we have $c = N$.

  - As before, we will first deal with the case when the $(\!|\lambda|\!)_{-\eta}$-redex is contained completely within N. Then $N \to_{(\!|\lambda|\!)_{-\eta}} N'$ and $b = \lambda x.\, N'\, x$. The common reduct d is N' since $b \twoheadrightarrow_\eta d$ in one step and $c = N \to^{=}_{(\!|\lambda|\!)_{-\eta}} d = N'$ as established before.
  - Now this is where we deal with the second overlap between β and η in our reduction system, the one with η on top. The $(\!|\lambda|\!)_{-\eta}$-redex in a must be $N\, x$ and the reduction rule in question must therefore be β. Therefore, $N = \lambda y.\, T(y)$ and $a = \lambda x.\, (\lambda y.\, T(y))\, x$. Performing the β-reduction gives us $b = \lambda x.\, T(x)$ which is, however, equal to $c = N = \lambda y.\, T(y)$. So we can choose $d = b$ and we are done.

- If a is the redex for both reductions $a \to_{(\!|\lambda|\!)_{-\eta}} b$ and $a \to_\eta c$, then a must match the left-hand side of a $(\!|\lambda|\!)_{-\eta}$ rule and the η rule. However, this is impossible since the left-hand side of the η rule is headed by abstraction, which is the case for none of the rules of $(\!|\lambda|\!)_{-\eta}$.

□

Equipped with this lemma, we can go on to prove our main result, Theorem 3.4.18, the confluence of $(\!|\lambda|\!)$.

Theorem 3.4.18. ***Confluence of*** $(\!|\lambda|\!)$
The reduction relation $\to$ on the set of $(\!|\lambda|\!)$ terms, defined by the reduction rules in 1.4, is confluent.

Proof. From Lemma 3.4.12, we know that the $(\!|\lambda|\!)_{-\eta}$ system is confluent and from Lemma 3.4.13, we know that the η-reduction rule is confluent as well. Lemma 3.4.17 tells us that these two reduction systems commute and therefore, by Lemma 3.4.15, their union, which is the $(\!|\lambda|\!)$ reduction system, commutes as well. □

3.5 Termination

Definition 3.5.1. *A reduction relation is* ***terminating*** *if there is no infinite chain $M_1 \to M_2 \to \ldots$.*

In this section, we will prove termination with a similar strategy as the one we employed for confluence. $(\!|\lambda|\!)$ is an extension of the λ-calculus with computation types and some operations on computations. Our computations can be thought of as algebraic expressions, i.e. they have a tree-like inductive structure. The reason that all computations in $(\!|\lambda|\!)$ terminate is that the operations defined on computations rely on well-founded recursion. However, it is quite tricky to go from this intuition to a formal proof of termination. Fortunately, we can rely on existing results.

Blanqui, Jouannaud and Okada have introduced Inductive Data Type Systems (IDTSs) [20, 19]. Like CRSs, IDTSs are a class of rewriting systems for which we can prove certain interesting general results. In this section, we will start by examining the definition of an IDTS and fitting $(\!|\lambda|\!)$ into that definition. The

[55] Since our rules are left-linear, M is guaranteed to appear in $L(M)$ at most once. Therefore, if $a' \to_\eta a''$ in one step, then also $L(a') \to_\eta L(a'')$ in one step as well.

[56] a' can occur multiple times in $R(a')$ when the rule $l \to r$ is duplicating (which is actually the case for the $(\!|\mathtt{op}|\!)$ rules). However, we are able to go from $R(a')$ to $R(a'')$ in multiple steps. NB: This is why we use Lemma 3.4.16 instead of trying to prove commutativity directly.

theory of IDTSs comes with a sufficient condition for termination known as the General Schema. $(\!| \lambda |\!)$ will not satisfy this condition and so we will first transform it using Hamana's technique of higher-order semantic labelling [55]. As with our proof of confluence, we will first consider the case of $(\!| \lambda |\!)$ without η-reduction and then add η manually while preserving termination.

The plan will look like this:

- IDTS = Typed CRS (3.5.1)
- The $(\!| \lambda |\!)_\tau$ IDTS (3.5.2)
  - if $(\!| \lambda |\!)_\tau$ terminates, then $(\!| \lambda |\!)_{-\eta}$ terminates (Lemma 3.5.11)
- Blanqui [19]: General Schema $\Rightarrow$ termination (3.5.3)
- Hamana [55]: IDTS R terminates iff the labelled IDTS $\overline{R}$ terminates (3.5.4)
  - Theorem 3.5.40: $\overline{(\!| \lambda |\!)_\tau}$ terminates (via Blanqui [19])
  - Corollary 3.5.41: $(\!| \lambda |\!)_\tau$ terminates (via Hamana [55])
  - Corollary 3.5.42: $(\!| \lambda |\!)_{-\eta}$ terminates (via Lemma 3.5.11)
- $(\!| \lambda |\!)_{-\eta} + \eta$ terminates (3.5.5, Theorem 3.5.46)
  - because $(\!| \lambda |\!)_{-\eta}$ and η are exchangeable (Lemma 3.5.44)
  - and therefore $(\!| \lambda |\!)$ is strongly normalizing (Theorem3.5.47)

3.5.1 Inductive Data Type Systems

We will go by the revised definition of Inductive Data Type Systems that figures in [19] and [55]. This formulation extends IDTSs to higher-order rewriting and does so using the CRS formalism that we introduced earlier.

Definition 3.5.2. *An **Inductive Data Type System (IDTS)** is a pair of an IDTS alphabet and a set of IDTS rewrite rules.*

Just like a CRS, an IDTS is an alphabet coupled with some rewrite rules. Let us first look at the alphabet and the rules for building terms out of the elements of the alphabet; the rewrite rules will follow.

Definition 3.5.3. *The set of types $T(\mathcal{B})$ contains:*

- *all the types from $\mathcal{B}$*
- *a type $\alpha \Rightarrow \beta$ for every α and β in $T(\mathcal{B})$*

Definition 3.5.4. *An **IDTS alphabet** consists of:*

- *$\mathcal{B}$, a set of* base types
- *$\mathcal{X}$, a family $(X_\tau)_{\tau \in T(\mathcal{B})}$ of sets of* variables
- *$\mathcal{F}$, a family $(F_{\alpha_1,\ldots,\alpha_n,\beta})_{\alpha_1,\ldots,\alpha_n,\beta \in T(\mathcal{B})}$ of sets of* function symbols
- *$\mathcal{Z}$, a family $(Z_{\alpha_1,\ldots,\alpha_n,\beta})_{\alpha_1,\ldots,\alpha_n,\beta \in T(\mathcal{B})}$ of sets of* metavariables

The distinction between a CRS-alphabet and an IDTS alphabet is that the IDTS alphabet comes equipped with a set of types. Furthermore, all the other symbols in the alphabet are indexed by types, so we end up with typed variables, typed function symbols and typed metavariables.

When we consider IDTS metaterms, we admit only well-typed terms. The definition of IDTS metaterms refines the definition of CRS metaterms by restraining term formation in accordance with the types.

Definition 3.5.5. *The **typed metaterms** of an IDTS are given inductively:*

- *variables from X_τ are metaterms of type τ*
- *if t is a metaterm of type β and x a variable from X_α, then $[x]t$ is a metaterm of type $\alpha \Rightarrow \beta$ called **abstraction***
- *if F is an function symbol from $F_{\alpha_1,\ldots,\alpha_n,\beta}$ and t_1,...,t_n are metaterms of types $\alpha_1, \ldots, \alpha_n$, respectively, then $F(t_1, \ldots, t_n)$ is a metaterm of type β*
- *if M is a metavariable from $Z_{\alpha_1,\ldots,\alpha_n,\beta}$ and t_1,...,t_n are metaterms of types $\alpha_1, \ldots, \alpha_n$, respectively, then $M(t_1, \ldots, t_n)$ is a metaterm of type β*

Definition 3.5.6. *The **terms** of an IDTS are its metaterms which do not contain any metavariables.*

The definition of an IDTS rewrite rule is almost identical to the one for CRS reduction rules. The only difference is the extra condition stating that the redex and contractum must have identical types.

Definition 3.5.7. *An **IDTS rewrite rule** is a pair of metaterms $s \to t$ such that:*

- *s and t are both closed, i.e. all variables are bound using the $[\_]\_$ abstraction binder*
- *s is of the form $F(t_1, \ldots, t_n)$*
- *all the metavariables that occur in t also occur in s*
- *any metavariable M that occurs in s only occurs in the form $M(x_1, \ldots, x_k)$, where x_i are pairwise distinct variables*
- *s and t are both of the same type*

As stated above, an IDTS is just an alphabet along with a set of rewrite rules. An IDTS induces a rewriting relation in exactly the same way as a CRS does, see [19] for more details.

3.5.2 $(\!|\lambda|\!)$ as an IDTS

Now we will link $(\!|\lambda|\!)$ to the IDTS framework in order to benefit from its general termination results. The biggest obstacle will be that IDTS assigns a fixed type to every symbol. In $(\!|\lambda|\!)$, symbols are polymorphic: the η constructor can produce expressions like $\eta \star : \mathcal{F}_E(1)$ or $\eta\,(\lambda x.\, x) : \mathcal{F}_E(\alpha \to \alpha)$ and that for any choice of E. We would therefore like to replace function symbols such as η with specialized symbols $\eta_{\mathcal{F}_E(\alpha)}$. For a given type α and effect signature E, the symbol $\eta_{\mathcal{F}_E(\alpha)}$ would have the type $\alpha \to \mathcal{F}_E(\alpha)$, i.e. it would belong to $F_{\alpha,\mathcal{F}_E(\alpha)}$.

We will call this calculus with specialized symbols $(\!|\lambda|\!)_\tau$. There will not be a bijection between $(\!|\lambda|\!)$ and $(\!|\lambda|\!)_\tau$ since a single term in $(\!|\lambda|\!)$ will generally correspond to a multitude of specialized versions in $(\!|\lambda|\!)_\tau$ (think of $\lambda x.\, x$ in $(\!|\lambda|\!)$ versus $\lambda x_\iota.\, x_\iota, \lambda x_o.\, x_o \ldots$ in $(\!|\lambda|\!)_\tau$). Therefore, the results we prove for $(\!|\lambda|\!)_\tau$ will not automatically transfer to $(\!|\lambda|\!)$. In the rest of this subsection, we will elaborate the definition of $(\!|\lambda|\!)_\tau$ and show why termination carries over from $(\!|\lambda|\!)_\tau$ to $(\!|\lambda|\!)_{-\eta}$.[57]

Defining $(\!|\lambda|\!)_\tau$

$(\!|\lambda|\!)_\tau$ will be defined as an IDTS. This means we need to first identify the alphabet. The base types $\mathcal{B}$ of $(\!|\lambda|\!)_\tau$ will be the set of types of $(\!|\lambda|\!)$. [58] Note that both $(\!|\lambda|\!)$ and IDTS have a notion of function type, but the notation is different. Contrary to common practice, in our exposition of IDTS we use $\alpha \Rightarrow \beta$ for the IDTS function type. This allows us to keep using the $\alpha \to \beta$ notation for $(\!|\lambda|\!)$ types, as we do in the rest of the thesis.

Next, we will introduce function symbols for all the syntactic constructions of $(\!|\lambda|\!)$, except for abstraction, which is handled by the $[\_]\_$ binder construct already found in IDTSs:

[57] In the sequel, we will ignore the η-reduction and use IDTSs to prove the termination of $(\!|\lambda|\!)$ without η-reduction, $(\!|\lambda|\!)_{-\eta}$.

[58] Note that throughout this section, we will make a distinction between two notions of "basic" types: atomic types and base types. Atomic types are the basic types of $(\!|\lambda|\!)$. Base types are the basic types of IDTSs. In our particular IDTS, the base types consist of *all* the types of $(\!|\lambda|\!)$, i.e. the atomic types, the $(\!|\lambda|\!)$ function types $\alpha \to \beta$ and the computation types. This means that, from the point of view of the IDTS, $(\!|\lambda|\!)$ function types and computation types are just another base type.

- $\mathsf{ap}_{\alpha,\beta} \in F_{\alpha\to\beta,\alpha,\beta}$ (i.e. for every pair of types α and β, there will be a function symbol $\mathsf{ap}_{\alpha,\beta}$ of type $(\alpha \to \beta) \Rightarrow \alpha \Rightarrow \beta$ in our alphabet)
- $\lambda_{\alpha,\beta} \in F_{\alpha\Rightarrow\beta,\alpha\to\beta}$
- $c \in F_\alpha$ for any constant $c : \alpha \in \Sigma$
- $\eta_{\alpha,E} \in F_{\alpha,\mathcal{F}_E(\alpha)}$
- $\mathsf{op}_{\gamma,E} \in F_{\alpha,\beta\Rightarrow\mathcal{F}_E(\gamma),\mathcal{F}_E(\gamma)}$ for any operation symbol op from $\mathcal{E}$ and any E such that $\mathsf{op} : \alpha \rightarrowtail \beta \in E$
- $\mathop{\downarrow}_\alpha \in F_{\mathcal{F}_\emptyset(\alpha),\alpha}$
- $(\!|\ |\!)_{\mathsf{op}_1,\ldots,\mathsf{op}_n,\gamma,\delta,E,E'} \in F_{\alpha_1\to(\beta_1\to\mathcal{F}_{E'}(\delta)),\ldots,\alpha_n\to(\beta_n\to\mathcal{F}_{E'}(\delta)),\gamma\to\mathcal{F}_{E'}(\delta),\mathcal{F}_E(\gamma),\mathcal{F}_{E'}(\delta)}$ where:
  - $\mathsf{op}_1 : \alpha_1 \rightarrowtail \beta_1 \in E$, ..., $\mathsf{op}_n : \alpha_n \rightarrowtail \beta_n \in E$
  - $E \setminus \{\mathsf{op}_1, \ldots, \mathsf{op}_n\} \subseteq E'$
- $\mathcal{C}_{\alpha,\beta,E} \in F_{\alpha\to\mathcal{F}_E(\beta),\mathcal{F}_E(\alpha\to\beta)}$

The list above is based on the typing rules of $(\!|\ \lambda\ |\!)$ found on Figure 1.1. We convert the typing rules of $(\!|\ \lambda\ |\!)$ into the typed function symbols of $(\!|\ \lambda\ |\!)_\tau$ with the following process:

- We take a typing rule of $(\!|\ \lambda\ |\!)$, other than [var] (since variables are already present in the language of IDTS terms).
- We identify all the type-level metavariables. That is, metavariables $\alpha, \beta, \gamma \ldots$ ranging over types, metavariables $E, E' \ldots$ ranging over effect signatures and metavariables op ranging over operation symbols.
- We strip these metavariables down to a minimal non-redundant set (e.g. in the $[\mathsf{op}]$ rule, we have that $\mathsf{op} : \alpha \rightarrowtail \beta \in E$, therefore E and op determine α and β and α and β are redundant).
- We introduce a family of symbols: for every possible instantiation of the metavariables mentioned above, we will have a different symbol. The arity of the symbol will correspond to the number of typing judgments that serve as hypotheses to the typing rule. The types of the arguments and of the result will be derived from the types of the judgments of the hypotheses and the conclusion respectively. If a variable of type α is bound in a premise of type β, then that will correspond to the IDTS function type $\alpha \Rightarrow \beta$.
  - Example: In the $[\eta]$ rule, we have two metavariables: α standing in for a type and E standing in for an effect signature. The rule has one typing judgment hypothesis. For every type α and every effect signature E, we will therefore have a unary symbol $\eta_{\alpha,E}$ of type $\alpha \Rightarrow \mathcal{F}_E(\alpha)$ (i.e. belonging to $F_{\alpha,\mathcal{F}_E(\alpha)}$).

A specifically-typed symbol in $(\!|\ \lambda\ |\!)_\tau$ then corresponds to an instantiation of the type metavariables in a $(\!|\ \lambda\ |\!)$ typing rule. We can follow this correspondence further and see that $(\!|\ \lambda\ |\!)_\tau$ IDTS terms, written using the above function symbols, correspond to typing derivations in $(\!|\ \lambda\ |\!)$.

Our alphabet now has types and function symbols. We also need to specify the sets of variables and metavariables and so we will take some arbitrary sets with $x_\tau, y_\tau, \ldots \in X_\tau$ and $M_{\alpha_1,\ldots,\alpha_n,\beta}, N_{\alpha_1,\ldots,\alpha_n,\beta}, \ldots \in Z_{\alpha_1,\ldots,\alpha_n,\beta}$.

To complete our IDTS, we have to give the rewrite rules. The rules for $(\!|\ \lambda\ |\!)_\tau$ are given in Figure 3.1. An important property of an IDTS rewrite rule is that both its left-hand and right-hand side are well-typed and that they have the same type. In order to facilitate the reader's verification that this is indeed the case, we have used a different labelling scheme for function symbols. When we write $f_{\alpha_1,\ldots,\alpha_n,\beta}$, we are referring to the instance of symbol f which has the type $\alpha_1 \Rightarrow \ldots \Rightarrow \alpha_n \Rightarrow \beta$ (i.e. belongs to $F_{\alpha_1,\ldots,\alpha_n,\beta}$). This way, instead of using a symbol name like $\eta_{\alpha,E}$, forcing you to look up its type $\alpha \Rightarrow \mathcal{F}_E(\alpha)$, we will refer to this symbol directly as $\eta_{\alpha,\mathcal{F}_E(\alpha)}$.

$\mathsf{ap}(\lambda([x]M(x)), N) \to$ rules $\beta_{\alpha,\gamma}$
$M(N)$
$\mathsf{ap}_{\alpha\to\gamma,\alpha,\gamma}(\lambda_{\alpha\Rightarrow\gamma,\alpha\to\gamma}([x_\alpha]M_{\alpha,\gamma}(x_\alpha)), N_\alpha) \to$
$M_{\alpha,\gamma}(N_\alpha)$

$\text{let } (\!| \, |\!)^{(\mathsf{op}_i)_{i\in I}}_{\mathcal{F}_E(\gamma),\mathcal{F}_{E'}(\delta)}(N_{\mathcal{F}_E(\gamma)})$
$= (\!| \, |\!)^{(\mathsf{op}_i)_{i\in I}}_{(\alpha_i\to(\beta_i\to\mathcal{F}_{E'}(\delta))\to\mathcal{F}_{E'}(\delta))_{i\in I},\gamma\to\mathcal{F}_{E'}(\delta),\mathcal{F}_E(\gamma),\mathcal{F}_{E'}(\delta)}((M^i_{\alpha_i\to(\beta_i\to\mathcal{F}_{E'}(\delta))\to\mathcal{F}_{E'}(\delta)})_{i\in I}, M^\eta_{\gamma\to\mathcal{F}_{E'}(\delta)}, N_{\mathcal{F}_E(\gamma)})$

$(\!| \, |\!)^{(\mathsf{op}_i)_{i\in I}}(\eta(N)) \to$ rules $(\!| \, \eta \, |\!)^{(\mathsf{op}_i)_{i\in I}}_{\mathcal{F}_E(\gamma),\mathcal{F}_{E'}(\delta)}$
$\mathsf{ap}(M^\eta, N)$
$(\!| \, |\!)^{(\mathsf{op}_i)_{i\in I}}_{\mathcal{F}_E(\gamma),\mathcal{F}_{E'}(\delta)}(\eta_{\gamma,\mathcal{F}_E(\gamma)}(N_\gamma)) \to$
$\mathsf{ap}_{\gamma\to\mathcal{F}_{E'}(\delta),\gamma,\mathcal{F}_{E'}(\delta)}(M^\eta_{\gamma\to\mathcal{F}_{E'}(\delta)}, N_\gamma)$

$(\!| \, |\!)^{(\mathsf{op}_i)_{i\in I}}(\mathsf{op}_j(N^{\mathsf{p}}, [y]N^{\mathsf{c}}(y))) \to$ rules $(\!| \, \mathsf{op} \, |\!)^{(\mathsf{op}_i)_{i\in I}}_{\mathcal{F}_E(\gamma),\mathcal{F}_{E'}(\delta)}$
$\mathsf{ap}(\mathsf{ap}(M^j, N^{\mathsf{p}}), \lambda([y](\!| \, |\!)^{(\mathsf{op}_i)_{i\in I}}(N^{\mathsf{c}}(y))))$ where $j \in I$
$(\!| \, |\!)^{(\mathsf{op}_i)_{i\in I}}_{\mathcal{F}_E(\gamma),\mathcal{F}_{E'}(\delta)}(\mathsf{op}_{j\alpha,\beta\Rightarrow\mathcal{F}_E(\gamma),\mathcal{F}_E(\gamma)}(N^{\mathsf{p}}_\alpha, [y_\beta]N^{\mathsf{c}}_{\beta,\mathcal{F}_E(\gamma)}(y_\beta))) \to$
$\mathsf{ap}_{(\beta\to\mathcal{F}_{E'}(\delta))\to\mathcal{F}_{E'}(\delta),\beta\to\mathcal{F}_{E'}(\delta),\mathcal{F}_{E'}(\delta)}(\mathsf{ap}_{\alpha\to(\beta\to\mathcal{F}_{E'}(\delta))\to\mathcal{F}_{E'}(\delta),\alpha,(\beta\to\mathcal{F}_{E'}(\delta))\to\mathcal{F}_{E'}(\delta)}(M^j_{\alpha\to(\beta\to\mathcal{F}_{E'}(\delta))\to\mathcal{F}_{E'}(\delta)}, N^{\mathsf{p}}_\alpha), \lambda_{\beta\Rightarrow\mathcal{F}_{E'}(\delta),\beta\to\mathcal{F}_{E'}(\delta)}([y_\beta](\!| \, |\!)^{(\mathsf{op}_i)_{i\in I}}_{\mathcal{F}_E(\gamma),\mathcal{F}_{E'}(\delta)}(N^{\mathsf{c}}_{\beta,\mathcal{F}_E(\gamma)}(y_\beta))))$

$(\!| \, |\!)^{(\mathsf{op}_i)_{i\in I}}(\mathsf{op}_j(N^{\mathsf{p}}, [y]N^{\mathsf{c}}(y))) \to$ rules $(\!| \, \mathsf{op}' \, |\!)^{(\mathsf{op}_i)_{i\in I}}_{\mathcal{F}_E(\gamma),\mathcal{F}_{E'}(\delta)}$
$\mathsf{op}_j(N^{\mathsf{p}}, [y](\!| \, |\!)^{(\mathsf{op}_i)_{i\in I}}(N^{\mathsf{c}}(y)))$ where $j \notin I$
$(\!| \, |\!)^{(\mathsf{op}_i)_{i\in I}}_{\mathcal{F}_E(\gamma),\mathcal{F}_{E'}(\delta)}(\mathsf{op}_{j\alpha,\beta\Rightarrow\mathcal{F}_E(\gamma),\mathcal{F}_E(\gamma)}(N^{\mathsf{p}}_\alpha, [y_\beta]N^{\mathsf{c}}_{\beta,\mathcal{F}_E(\gamma)}(y_\beta))) \to$
$\mathsf{op}_{j\alpha,\beta\Rightarrow\mathcal{F}_{E'}(\delta),\mathcal{F}_{E'}(\delta)}(N^{\mathsf{p}}_\alpha, [y_\beta](\!| \, |\!)^{(\mathsf{op}_i)_{i\in I}}_{\mathcal{F}_E(\gamma),\mathcal{F}_{E'}(\delta)}(N^{\mathsf{c}}_{\beta,\mathcal{F}_E(\gamma)}(y_\beta)))$

$\downarrow(\eta(N)) \to$ rules $\downarrow_\alpha$
N
$\downarrow_{\mathcal{F}_\emptyset(\alpha),\alpha}(\eta_{\alpha,\mathcal{F}_\emptyset(\alpha)}(N_\alpha)) \to$
N_α

$\mathcal{C}(\lambda([x]\eta(M(x)))) \to$ rules $\mathcal{C}^\eta_{\alpha,\beta,E}$
$\eta(\lambda([x]M(x)))$
$\mathcal{C}_{\alpha\to\mathcal{F}_E(\beta),\mathcal{F}_E(\alpha\to\beta)}(\lambda_{\alpha\Rightarrow\mathcal{F}_E(\beta),\alpha\to\mathcal{F}_E(\beta)}([x_\alpha]\eta_{\beta,\mathcal{F}_E(\beta)}(M_{\alpha,\beta}(x_\alpha)))) \to$
$\eta_{(\alpha\to\beta),\mathcal{F}_E(\alpha\to\beta)}(\lambda_{\alpha\Rightarrow\beta,\alpha\to\beta}([x_\alpha]M_{\alpha,\beta}(x_\alpha)))$

$\mathcal{C}(\lambda([x]\mathsf{op}(M^p, [y]M^c(x,y)))) \to$ rules $\mathcal{C}^{\mathsf{op}}_{\alpha,\beta,E}$
$\mathsf{op}(M^p, [y]\mathcal{C}(\lambda([x]M^c(x,y))))$
$\mathcal{C}_{\alpha\to\mathcal{F}_E(\beta),\mathcal{F}_E(\alpha\to\beta)}(\lambda_{\alpha\Rightarrow\mathcal{F}_E(\beta),\alpha\to\mathcal{F}_E(\beta)}([x_\alpha]\mathsf{op}_{\gamma,\delta\Rightarrow\mathcal{F}_E(\beta),\mathcal{F}_E(\beta)}(M^p_\gamma, [y_\delta]M^c_{\alpha,\delta,\mathcal{F}_E(\beta)}(x_\alpha, y_\delta)))) \to$
$\mathsf{op}_{\gamma,\delta\Rightarrow\mathcal{F}_E(\alpha\to\beta),\mathcal{F}_E(\alpha\to\beta)}(M^p_\gamma, [y_\delta]\mathcal{C}_{\alpha\to\mathcal{F}_E(\beta),\mathcal{F}_E(\alpha\to\beta)}(\lambda_{\alpha\Rightarrow\mathcal{F}_E(\beta),\alpha\to\mathcal{F}_E(\beta)}([x_\alpha]M^c_{\alpha,\delta,\mathcal{F}_E(\beta)}(x_\alpha, y_\delta))))$

Figure 3.1: The IDTS rewrite rules for $(\!| \, \lambda \, |\!)_\tau$, shown in parallel with the CRS rules for $(\!| \, \lambda \, |\!)$.

In Figure 3.1, you will also find the rewrite rules with all the subscripts removed. This allows you to get a high-level look at the term without any of the type annotation noise. By removing the type indices from the $(\!| \lambda |\!)_\tau$ IDTS rewrite rules, we get the $(\!| \lambda |\!)$ CRS-reduction rules of Section 3.4 (modulo the renaming of @ to ap).

When describing the rewrite rules for handlers, we introduce a shortcut $(\!| \;|\!)^{(\mathtt{op}_i)_{i\in I}}_{\mathcal{F}_E(\gamma),\mathcal{F}_{E'}(\delta)}(N_{\mathcal{F}_E(\gamma)})$, which stands for $(\!| \;|\!)$ partially applied to the clauses M^i and M^η. We then reuse this shortcut in all of the $(\!| \;|\!)$ rules.

Connecting $(\!| \lambda |\!)_\tau$ to $(\!| \lambda |\!)$

We have given a complete formal definition of $(\!| \lambda |\!)_\tau$. This will let us find a proof of termination for $(\!| \lambda |\!)_\tau$ using the theory of IDTSs. However, in order to carry over this result to our original calculus, we will need to formalize the relationship between the two.

Definition 3.5.8. ***Term*** *is a (partial) function from $(\!| \lambda |\!)_\tau$ terms to $(\!| \lambda |\!)$ terms which removes any type annotations (the subscripts on function symbols, variables and metavariables) and translates $(\!| \lambda |\!)_\tau$ syntax to $(\!| \lambda |\!)$ syntax using the following equations:*

$$
\begin{aligned}
&\mathrm{Term}(x) && = x\\
&\mathrm{Term}(\lambda([x]M)) && = \lambda x.\ \mathrm{Term}(M)\\
&\mathrm{Term}(\mathtt{ap}(M,N)) && = (\mathrm{Term}(M))\,(\mathrm{Term}(N))\\
&\mathrm{Term}(c) && = c\\
&\mathrm{Term}(\eta(M)) && = \eta\,(\mathrm{Term}(M))\\
&\mathrm{Term}(\mathtt{op}(M^{\mathrm{p}},[x]M^{\mathrm{c}})) && = \mathtt{op}\,(\mathrm{Term}(M^{\mathrm{p}}))\,(\lambda x.\ \mathrm{Term}(M^{\mathrm{c}}))\\
&\mathrm{Term}(\downarrow_{\circ}(M)) && = \downarrow_{\circ}(\mathrm{Term}(M))\\
&\mathrm{Term}((\!| \;|\!)_{\mathtt{op}_1,\ldots,\mathtt{op}_n}(M_1,\ldots,M_n,M_\eta,N)) && = (\!|\, \mathtt{op}_1\colon \mathrm{Term}(M_1),\ \mathtt{op}_n\colon \mathrm{Term}(M_n),\ \eta\colon \mathrm{Term}(M_\eta)\,|\!)\,(\mathrm{Term}(N))\\
&\mathrm{Term}(\mathcal{C}(M)) && = \mathcal{C}\,(\mathrm{Term}(M))
\end{aligned}
$$

Definition 3.5.9. ***Types*** *is a function from $(\!| \lambda |\!)$ terms to sets of $(\!| \lambda |\!)_\tau$ terms, defined by the equation below.*

$$\mathrm{Types}(M) = \{m \mid \mathrm{Term}(m) = M\}$$

Lemma 3.5.10. *Let M and N be $(\!| \lambda |\!)$ terms. Then,*

$$M \to_{(\!| \lambda |\!)_{-\eta}} N \quad \Rightarrow \quad \forall m \in \mathrm{Types}(M).\ \exists n \in \mathrm{Types}(N).\ m \to n$$

In the above, upper-case letters stand for $(\!| \lambda |\!)$ terms, while lower-case letters stand for $(\!| \lambda |\!)_\tau$ terms.

Proof. This property is essentially a stronger kind of subject reduction for $(\!| \lambda |\!)_{-\eta}$. In proofs of subject reduction, we examine every reduction rule and we show how a typing derivation of the redex can be transformed into a typing derivation of the contractum. We can think of $(\!| \lambda |\!)_\tau$ terms as $(\!| \lambda |\!)$ typing derivations. The reduction rules in Figure 3.1 are the rules which tell us how to take a typing of the redex and transform it into a typing of the contractum.

In order to prove this property, we will need to check the following:

- The redexes and contracta in Figure 3.1 are well-formed (i.e. well-typed). For that reason, we have included the type of every variable, metavariable and function symbol as a subscript.
- Applying Term to the left-hand and right-hand sides of the $(\!| \lambda |\!)_\tau$ rules yields the left-hand and right-hand sides of all the $(\!| \lambda |\!)_{-\eta}$ rules (and therefore the left-hand and right-hand sides of $(\!| \lambda |\!)_\tau$ rules belong to the Types image of the left-hand and right-hand sides of the $(\!| \lambda |\!)_{-\eta}$ rules). Since in Figure 3.1, we have included the terms with their type annotations removed, we can see at a glance that the stripped rules align with the CRS formulation of $(\!| \lambda |\!)$.

- Finally, we have to check whether the rewriting rules in Figure 3.1 actually apply to *all* the $m \in \text{Types}(M)$. In other words, we need to check whether the type annotation scheme used for the left-hand sides is the most general and covers all possible typings of the left-hand side. This is the case because we have followed the typing rule constraints and given the most general type annotations.

Given a reduction in $(\!| \lambda |\!)_{-\eta}$ from M to N, we can find the untyped reduction rule used in Figure 3.1. We know that if $m \in \text{Types}(M)$, then m then matches the left-hand side of the corresponding typed rule. We also know that the right-hand side of the typed rule belongs to $\text{Types}(N)$ and therefore, the property holds. Furthermore, if we were to formalize the correspondence between $(\!| \lambda |\!)$ typing derivations and $(\!| \lambda |\!)_\tau$ terms, we would get another proof of subject reduction for $(\!| \lambda |\!)_{-\eta}$. □

Lemma 3.5.11. *If the reduction relation of $(\!| \lambda |\!)_\tau$ is terminating, then so is the $(\!| \lambda |\!)_{-\eta}$ reduction relation on well-typed terms.*

Proof. Consider the contrapositive: if there exists an infinite $(\!| \lambda |\!)_{-\eta}$ chain of well-typed $(\!| \lambda |\!)$ terms, we can use Lemma 3.5.10 to translate it, link by link, to an infinite $(\!| \lambda |\!)_\tau$ chain. However, infinite $(\!| \lambda |\!)_\tau$ reduction chains do not exist since $(\!| \lambda |\!)_\tau$ is terminating. □

3.5.3 Termination for IDTSs

So far, we have introduced an IDTS and have shown that if this IDTS is terminating, then so is $(\!| \lambda |\!)_{-\eta}$. We will now look at a general result for IDTSs that we will make use of.

Theorem 3.5.12. ***Strong normalization*** *[19]*
Let $\mathcal{I} = (\mathcal{A}, \mathcal{R})$ be a β-IDTS satisfying the assumptions (A). If all the rules of $\mathcal{R}$ satisfy the General Schema, then $\rightarrow_{\mathcal{I}}$ is terminating.

The theorem was lifted verbatim[59] from [19] and parts of it deserve explaining:

- What is a β-IDTS?
- What are the assumptions (A)?
- What is the General Schema?

We will deal with these in order.

A β-IDTS is an IDTS which, for every two types α and β, has a function symbol $@_{\alpha,\beta} \in F_{\alpha \Rightarrow \beta, \alpha, \beta}$ and a rule $@_{\alpha,\beta}([x_\alpha]M_{\alpha,\beta}(x_\alpha), N_\alpha) \rightarrow M_{\alpha,\beta}(N_\alpha)$. Furthermore, there must be no other rules whose left-hand side is headed by @. We can turn our IDTS from 3.5.2 into a β-IDTS by extending it with these function symbols and reduction rules.[60] Termination in a larger system will still imply termination in our system.

Checking Off the Assumptions

Next, we will deal with the assumptions (A).

Definition 3.5.13. *The **Assumptions (A)** are defined as the following four conditions:*

1. *every constructor is positive*
2. *no left-hand side of rule is headed by a constructor*
3. *both $>_{\mathcal{B}}$ and $>_{\mathcal{F}}$ are well-founded*
4. *$stat_f = stat_g$ whenever $f =_{\mathcal{F}} g$*

[59] In fact, the actual Theorem in [19] states that the system is *strongly normalizing*. However, by strongly normalizing they mean that every term is computable, i.e. that there is no infinite reduction chain.

[60] These β rules and application operators are different from the ones already in our IDTS. ap is defined for the $\alpha \rightarrow \beta$ function type from $(\!| \lambda |\!)$ whereas @ serves the $\alpha \Rightarrow \beta$ type of IDTS.

For these to make sense to us, we will need to identify some more structure on top of our IDTS: the notion of a constructor and the $>_{\mathcal{B}}$ and $>_{\mathcal{F}}$ relations.

We will need to designate for every base type γ a set $C_\gamma \subseteq \cup_{p\geq 0,\alpha_1,\ldots,\alpha_p\in T(\mathcal{B})} F_{\alpha_1,\ldots,\alpha_p,\gamma}$ (i.e. a set of function symbols with result type γ). We will call the elements of these sets *constructors* of γ.

The base types of our IDTS consist of atomic types, function types and computation types. We will have no constructors for atomic types. On the other hand, every function type $\alpha \to \beta$ will have a constructor $\lambda_{\alpha,\beta}$ ($\in F_{\alpha\Rightarrow\beta,\alpha\to\beta}$) and every computation type $\mathcal{F}_E(\gamma)$ will have constructors $\eta_{\gamma,E}$ ($\in F_{\gamma,\mathcal{F}_E(\gamma)}$) and $\mathtt{op}_{\gamma,E}$ ($\in F_{\alpha,\beta\Rightarrow\mathcal{F}_E(\gamma),\mathcal{F}_E(\gamma)}$) for every $\mathtt{op} : \alpha \rightarrowtail \beta \in E$.

We can now check assumption (A.2). Since the only constructors in our IDTS are η, $\mathtt{op}$ and λ, we validate this assumption.[61]

Our choice of constructors induces a binary relation on the base types.

Definition 3.5.14. *The base type α **depends on** the base type β if there is a constructor $c \in C_\alpha$ such that β occurs in the type of one of the arguments of c.*

We will use $\geq_{\mathcal{B}}$ to mean the reflexive-transitive closure of this relation. Furthermore, we will use $=_{\mathcal{B}}$ and $>_{\mathcal{B}}$ to mean the associated equivalence and strict ordering, respectively.

Observation 3.5.15. *If $\tau_1 \leq_{\mathcal{B}} \tau_2$, then τ_1 is a subterm of τ_2.*

Proof. We will prove this by induction on the structure of the base type τ_2. If τ_2 is an atomic type, then τ_2 has no constructors, so it does not depend on any other type. If we look at the reflexive-transitive closure of that, $\geq_{\mathcal{B}}$, then the only type α such that $\tau_2 \geq_{\mathcal{B}} \alpha$ is, by reflexivity, τ_2 itself, which is a subterm of τ_2.

If τ_2 is the computation type $\mathcal{F}_E(\gamma)$, then we will have several constructors. We have $\eta_{\gamma,E}$ with a single argument of type γ. We thus know that $\mathcal{F}_E(\gamma)$ depends on γ. For every $\mathtt{op} : \alpha \rightarrowtail \beta \in E$, we have a constructor $\mathtt{op}_{\gamma,E}$ with arguments of types α and $\beta \Rightarrow \mathcal{F}_E(\gamma)$. This tells us that $\mathcal{F}_E(\gamma)$ also depends on α, β and $\mathcal{F}_E(\gamma)$. $\mathcal{F}_E(\gamma)$ does not have any more constructors, so those are all the types it depends on. The $\geq_{\mathcal{B}}$ relation, which is the subject of this observation, is the reflexive-transitive closure of the dependency relation between base types. This means that $\tau_2 \geq_{\mathcal{B}} \tau_1$ if either $\tau_2 = \tau_1$ or τ_2 depends on some $\tau_2' \neq \tau_2$ such that $\tau_2' \geq_{\mathcal{B}} \tau_1$.

- If $\tau_2 = \tau_1$, then trivially τ_1 is a subterm of τ_2 and we are done.
- If τ_2 depends on some $\tau_2' \neq \tau_2$, then τ_2' must be either γ or one of the α or β from E since $\tau_2 = \mathcal{F}_E(\gamma)$. In all these cases, we can apply the induction hypothesis for τ_2'. We know that $\tau_2' \geq_{\mathcal{B}} \tau_1$ and by the induction hypothesis, we now know that τ_1 is a subterm of τ_2'. Since τ_2' is a subterm of τ_2, we have that τ_1 is a subterm of τ_2.

□

Corollary 3.5.16. *If $\tau_1 =_{\mathcal{B}} \tau_2$, then $\tau_1 = \tau_2$.*

Corollary 3.5.17. *If $\tau_1 <_{\mathcal{B}} \tau_2$, then τ_1 is a proper subterm of τ_2.*

We can now check assumption (A.3). Since the proper subterm relation is well-founded (i.e. has no infinite descending chains) and $>_{\mathcal{B}}$ is a subset of the proper subterm relation, then $>_{\mathcal{B}}$ must be well-founded as well.

We can also check assumption (A.1) once we explain what a strictly positive type is.

Definition 3.5.18. *A constructor $c \in C_\beta$ is **positive** if every base type $\alpha =_{\mathcal{B}} \beta$ occurs only at positive positions in the types of the arguments of c.*

Definition 3.5.19. *The base types occurring in **positive positions** (Pos) and the base types occurring in **negative positions** (Neg) within a type are defined by the following mutually recursive equations:*

$$\begin{aligned}
\mathrm{Pos}(\alpha \Rightarrow \beta) &= \mathrm{Neg}(\alpha) \cup \mathrm{Pos}(\beta) \\
\mathrm{Neg}(\alpha \Rightarrow \beta) &= \mathrm{Pos}(\alpha) \cup \mathrm{Neg}(\beta) \\
\mathrm{Pos}(\nu) &= \{\nu\} \quad \textit{with } \nu \textit{ an atomic type} \\
\mathrm{Neg}(\nu) &= \emptyset \quad \textit{with } \nu \textit{ an atomic type}
\end{aligned}$$

[61] This is why we have to prove termination with η reduction separately

In our IDTS, $\alpha =_{\mathcal{B}} \beta$ is true only when $\alpha = \beta$. The only time a base type occurs in the type of one of its constructor's arguments is in the case of the op constructors. Given $\mathtt{op} : \alpha \rightarrowtail \beta \in E$, $\mathtt{op}_{\gamma,E}$ is a constructor of $\mathcal{F}_E(\gamma)$; the type of its second argument is $\beta \Rightarrow \mathcal{F}_E(\gamma)$. This occurrence is positive and so we validate assumption (A.1).

To validate the second half of (A.3), we will need to introduce the $>_{\mathcal{F}}$ relation. As $>_{\mathcal{B}}$ was induced by the structure of constructors, $>_{\mathcal{F}}$ will be induced by the structure of the rewriting rules $\mathcal{R}$ of our IDTS.

Definition 3.5.20. *A function symbol f **depends on** a function symbol g if there is a rule defining f (i.e. whose left-hand side is headed by f) and in the right-hand side of which g occurs.*

We will use $\geq_{\mathcal{F}}$ as the name for the reflexive-transitive closure of this relation. We will also write $=_{\mathcal{F}}$ and $>_{\mathcal{F}}$ for the associated equivalence and strict ordering, respectively.

If we scan the rules of $(\!| \lambda |\!)$, we will see that the $(\!| \; |\!)$ symbols depend on op (for when there is no handler and the op is copied), ap (for applying the handler clauses to their arguments), λ (for the continuation) and on $(\!| \; |\!)$ (for recursion). The $\mathcal{C}$ symbols depend on op (when passing the λ through an op), η (when switching the λ with the η), λ (for the argument) and $\mathcal{C}$ (for recursion). There is no other dependency in our IDTS. This means we can check off the second part of assumption (A.3) since $>_{\mathcal{F}}$ is well-founded (it contains only $(\!| \; |\!) >_{\mathcal{F}} \mathtt{op}$, $(\!| \; |\!) >_{\mathcal{F}} \mathtt{ap}$, $(\!| \; |\!) >_{\mathcal{F}} \lambda$, $\mathcal{C} >_{\mathcal{F}} \mathtt{op}$, $\mathcal{C} >_{\mathcal{F}} \eta$ and $\mathcal{C} >_{\mathcal{F}} \lambda$).

Assumption (A.4) is trivial in our case since, within our IDTS, $f =_{\mathcal{F}} g$ only when $f = g$. This assumption comes into play only in the general theory of IDTSs when one exploits mutual recursion with functions of multiple arguments. The $stat_f$ values mentioned in the assumption (A.4) describe the way in which a function's arguments should be ordered to guarantee that recursive calls are always made to smaller arguments. In the case of mutual recursion, both functions must agree on the order according to which they will decrease their arguments. Since we do not deal with mutual recursion in $(\!| \lambda |\!)$, we will not go into any more detail into this.

General Schema

There is one last obstacle in our way towards proving termination of $(\!| \lambda |\!)_\tau$. We will need to verify that the rewrite rules that we have given in Figure 3.1 follow the General Schema.

Definition 3.5.21. *A rewrite rule $f(l_1, \ldots, l_n) \rightarrow r$ follows the **General Schema** if $r \in \mathcal{CC}_f(l_1, \ldots, l_n)$.*

$\mathcal{CC}_f(l_1, \ldots, l_n)$ refers to the so-called *computable closure* of the left-hand side $f(l_1, \ldots, l_n)$. The idea behind the computable closure is that the left-hand side of a rewrite rule can tell us what are all the possible right-hand sides that still lead to a correct proof of termination.[62] A formal definition of computable closure is given in [19, p. 8].

Informally, $r \in \mathcal{CC}_f(l_1, \ldots, l_n)$ if:

- Every metavariable used in r is accessible in one of l_1, ..., l_n.
- Recursive function calls (i.e. uses of function symbols $g =_{\mathcal{F}} f$) are made to arguments smaller than the arguments l_1, ..., l_n.

A metavariable is **accessible** in a term if it appears at the top of the term or under abstractions or constructors. If a metavariable occurs inside an argument of a function symbol which is not a constructor, then there are some technical constraints on whether it is accessible. However, in every rewrite rule of our IDTS, the arguments of the function symbol being defined contain only constructors as function symbols.

Finally, we will need to show that the arguments being recursively passed to $(\!| \; |\!)$ and $\mathcal{C}$ are smaller than the original arguments and therefore the recursion is well-founded and terminating. However, the General Schema presented in [19] uses a notion of "smaller than" which is not sufficient to capture the decrease of our arguments. On the other hand, when we were defining a denotational semantics for $(\!| \lambda |\!)$,

[62] Theorem 3.5.12 is proven using Tait's method of computability predicates [125]. The term computable closure comes from the fact that the admissible right-hand sides are the metavariables of the left-hand side closed on operations that preserve computability.

we gave a well-founded ordering showing that the successive arguments to these operations are in fact decreasing. We will therefore make use of a technique which will allows us to incorporate this semantic insight into the IDTS so that the General Schema will be able to recognize the decreasing nature of the arguments.

3.5.4 Higher-Order Semantic Labelling

We will make use of the higher-order semantic labelling technique presented by Makoto Hamana in [55]. The idea behind the semantic labelling technique is to label function symbols with the denotations of their arguments. Whereas before, a function symbol was rewritten to the same function symbol on a smaller argument, in the labelled IDTS, a labelled symbol will be rewritten to a different *smaller* symbol (i.e. one with a smaller label).

The theory in [55] is expressed in terms of category theory. This results in a very elegant and concise formulation of the theorems and their proofs. In our thesis, we only care about the applications of the theory and so we will try to introduce the technique without presupposing the reader's familiarity with category theory.

Presheaves and Binding Algebras

We will nevertheless introduce a few terms from category theory.

When dealing with binding and types, it is usually not so useful to consider a mixed set of terms or denotations of different types. It is much more pertinent to speak of families of terms having the same type in the same typing context, i.e. $T_\tau(\Gamma) = \{t \mid \Gamma \vdash t : \tau\}$. In this example, T is a family of sets, indexed first by type and second by context. We can therefore say that $T \in (\mathbf{Set}^{\mathbb{F}\downarrow\mathcal{B}})^{\mathcal{B}}$, where $\mathcal{B}$ is the set of base types of our IDTS (i.e. the set of $(\!|\lambda|\!)$ types) and $\mathbb{F} \downarrow \mathcal{B}$ is the set of $(\!|\lambda|\!)$ typing contexts (functions from finite sets to $\mathcal{B}$).

The category-theoretical presentation of abstract syntax and binding originating in [45] relies on a similar notion known as *presheaf*. Presheaf can be seen as a synonym for functor (see 3.3.4), usually going from some kind of "index category" to some other category. In the above example, $\mathcal{B}$, $\mathbb{F} \downarrow \mathcal{B}$ and **Set** can be seen as categories:

- $\mathcal{B}$ has base types as objects and no arrows besides the mandatory identities
- $\mathbb{F} \downarrow \mathcal{B}$ has typing contexts as objects and renamings of contexts (exchanges, weakenings, contractions) as arrows
- **Set** is the standard category with sets as objects and functions as arrows

$\mathbf{Set}^{\mathbb{F}\downarrow\mathcal{B}}$ is the category of functors from $\mathbb{F} \downarrow \mathcal{B}$ to **Set**. The object component of such a functor maps contexts to sets (usually sets of objects having some type within the given context). The arrow component translates the renamings of contexts into renamings of variables in these objects. The functors in the category $(\mathbf{Set}^{\mathbb{F}\downarrow\mathcal{B}})^{\mathcal{B}}$ map types to the objects of $\mathbf{Set}^{\mathbb{F}\downarrow\mathcal{B}}$; their arrow component is trivial since $\mathcal{B}$ has only trivial arrows.

We will call the objects of $(\mathbf{Set}^{\mathbb{F}\downarrow\mathcal{B}})^{\mathcal{B}}$ presheaves (sometimes, we will also call the objects in $\mathbf{Set}^{\mathbb{F}\downarrow\mathcal{B}}$ presheaves). In our presentation, we will care only about the object level, meaning that we will identify a presheaf with a family of sets. We will now consider some presheaves that will come into play:

- The key presheaf will be the presheaf T of $(\!|\lambda|\!)$ terms, $T_{\tau,\Gamma} = \{M \mid \Gamma \vdash M : \tau\}$. Every element of $T_{\tau,\Gamma}$ is a well-typed $(\!|\lambda|\!)$ term.
- Another useful presheaf is the presheaf V of variables where $V_{\tau,\Gamma} = \{x \mid x : \tau \in \Gamma\}$.
- Z is the presheaf of the IDTS metavariables from $\mathcal{Z}$, $Z_{\tau,(x_1:\alpha_1,\ldots,x_n:\alpha_n)} = \{M \mid M \in Z_{\alpha_1,\ldots,\alpha_n,\tau}\}$.
- $T_\Sigma V$ is the presheaf of IDTS terms with alphabet Σ.
- $M_\Sigma Z$ is the presheaf of IDTS metaterms with alphabet Σ and typed metavariables Z.

Now we will define some endofunctors on the category of presheaves. As before, we will ignore the arrow component and give only a mapping from one family to another.

- First, we introduce an endofunctor on the category of presheaves in $\mathbf{Set}^{\mathbb{F}\downarrow\mathcal{B}}$. For every base type τ, we have a functor $\delta_\tau : \mathbf{Set}^{\mathbb{F}\downarrow\mathcal{B}} \to \mathbf{Set}^{\mathbb{F}\downarrow\mathcal{B}}$. For $A \in \mathbf{Set}^{\mathbb{F}\downarrow\mathcal{B}}$, we define $(\delta_\tau A)(\Gamma) = A(\Gamma + \tau)$ where $\Gamma + \tau$ is the extension of context Γ by a variable of type τ.[63] The idea behind this operation is to model binders, i.e. the arrow type $\Rightarrow$ of IDTS. If the presheaf $A_\beta \in \mathbf{Set}^{\mathbb{F}\downarrow\mathcal{B}}$ models the type β, then the presheaf $\delta_\alpha A_\beta$ models the type $\alpha \Rightarrow \beta$.

- The alphabet of our IDTS, Σ, induces an endofunctor on the category $(\mathbf{Set}^{\mathbb{F}\downarrow\mathcal{B}})^{\mathcal{B}}$ mapping presheaves A to presheaves ΣA.

$$(\Sigma A)_\gamma = \coprod_{f \in F_{\vec{\alpha_1} \Rightarrow \beta_1, \ldots, \vec{\alpha_l} \Rightarrow \beta_l, \gamma}} \prod_{1 \le i \le l} \delta_{\vec{\alpha_i}} A_{\beta_i}$$

  In the above, we use the vector notation $\vec{\alpha} \Rightarrow \beta$ for $\alpha_1 \Rightarrow \ldots \Rightarrow \alpha_n \Rightarrow \beta$ and $\delta_{\vec{\alpha}}$ for $\delta_{\alpha_1} \circ \ldots \circ \delta_{\alpha_n}$. Note that the above definition assumes that $\vec{\alpha_i}$, β_i and γ are all base types. Since in our encoding of $(\!| \lambda |\!)$, we use a function constructor $\to$ on the level of base types, this is the case.

We are now ready to define the notion of a Σ-binding algebra.

Definition 3.5.22. *A Σ-**(binding) algebra** $\mathcal{A}$ is a pair of a presheaf A and a natural transformation[64] $\alpha : \Sigma A \to A$. The presheaf is the **carrier** and the natural transformation interprets the **operations**. Since ΣA is a coproduct over all the $f \in \mathcal{F}$, we can also see α as the copair $[f^{\mathcal{A}}]_{f \in \mathcal{F}}$, where $f^{\mathcal{A}}$ is the interpretation in algebra $\mathcal{A}$ of operation f.*

We will need to construct a $(V + \Sigma)$-algebra in order to proceed, where $(V + \Sigma)(A)$ is defined as $V + \Sigma(A)$. Our algebra will be a term algebra, the carrier will be the presheaf T. We will need to give an interpretation to variables and to every function symbol defined in the alphabet Σ. This interpretation must be given as an $\alpha : V + \Sigma T \to T$, meaning that the interpretation of every function symbol must be compositional. A value from $(\Sigma T)_{\gamma,\Gamma}$ is composed of some function symbol f of result type γ together with the interpretation of all of the symbol's arguments. If the i-th argument of the function symbol f has type β_i and binds variables $\vec{alpha_i}$, then the interpretation of the argument in our term model will be a $(\!| \lambda |\!)$ term whose type in the context $\Gamma + \vec{\alpha_i}$ is β_i.

The translation Term from $(\!| \lambda |\!)_\tau$ terms to $(\!| \lambda |\!)$ terms that we have given in 3.5.2 gives us the operations of the term model algebra. For example, the line defining the translation of the expression $\mathtt{op}(M^{\mathrm{p}}, [x]M^{\mathrm{c}})$ can be transformed into an interpretation for the function symbol $\mathtt{op}$ the following way:

$$\begin{aligned} \mathrm{Term}(\mathtt{op}(M^{\mathrm{p}}, [x]M^{\mathrm{c}})) &= \mathtt{op}\,(\mathrm{Term}(M^{\mathrm{p}}))\,(\lambda x.\ \mathrm{Term}(M^{\mathrm{c}})) \\ \mathtt{op}_\Gamma(M^{\mathrm{p}}, M^{\mathrm{c}}) &= \mathtt{op}\,(\mathrm{Term}(M^{\mathrm{p}}))\,(\lambda x_{n+1}.\ \mathrm{Term}(M^{\mathrm{c}})) \end{aligned}$$

where $n = |\Gamma|$.

Therefore, the Term translation function from 3.5.2 gives us a $(V + \Sigma)$-algebra $\mathcal{T}$ with carrier T.

Building a Quasi-Model

We will now deal with presheaves equipped with partial orders.

[63] When we extend a context, we usually extend it with a pair of a variable name and a type, e.g. $\Gamma, x : \tau$. However, the theory of binding algebras uses Bruijn levels [34], where the names of variables in a context are always integers from 1 to some n. Extending a context $x_1 : \alpha_1, \ldots, x_n : \alpha_n$ with a type τ then yields a context $x_1 : \alpha_1, \ldots, x_n : \alpha_n, x_{n+1} : \tau$.

[64] Natural transformation is the name for an arrow between two functors (presheaves). In our particular setting, naturality boils down to A_γ being a function of $(\Sigma A)_\gamma$ for every γ.

Definition 3.5.23. *A **presheaf equipped with a partial order** is a pair of a presheaf A and a family of partial orders $\geq_A$ such that $\geq_{A_{\tau,\Gamma}}$ is a partial order on the set $A_{\tau,\Gamma}$.*

Definition 3.5.24. *An arrow $f : A^1 \times \cdots \times A^n \to B$ in $\mathbf{Set}^{\mathbb{F}\downarrow\mathcal{B}}$ is **weakly monotonic** if for all Γ and $a_1, b_1 \in A^1_\Gamma$, ..., $a_n, b_n \in A^n_\Gamma$ with $a_k \geq_{A_{k,\Gamma}} b_k$ for some k and $a_j = b_j$ for all $j \neq k$, we have that $f(\Gamma)(a_1, \ldots, a_n) \geq_{B_\Gamma} f(\Gamma)(b_1, \ldots, b_n)$.*

Definition 3.5.25. *A **weakly monotonic** $(V + \Sigma)$**-algebra** is a $(V + \Sigma)$-algebra $\mathcal{A}$ whose carrier A is equipped with a partial order $\geq_A$ such that every operation of $\mathcal{A}$ is weakly monotonic.*

We want to equip our $(V + \Sigma)$-algebra $\mathcal{T}$ with the $\twoheadrightarrow$ order. However, while we know that $\twoheadrightarrow$ is by definition a preorder, reflexive and transitive, we do not know whether it is antisymmetric and therefore whether it forms a partial order. Because of this, we will build a partial order *on top of* the $\twoheadrightarrow$ preorder.

Definition 3.5.26. *We say that terms M and N are **interreducible**, $M \rightleftarrows N$, if $M \twoheadrightarrow N$ and $N \twoheadrightarrow M$.*

The interreducibility relation defined above is an equivalence relation and we can use it to quotient sets of terms. We define the $T^{/\rightleftarrows}$ presheaf as the presheaf with $T^{/\rightleftarrows}_{\tau,\Gamma} = \{\{N \mid M \rightleftarrows N\} \mid \Gamma \vdash M : \tau\}$, i.e. $T^{/\rightleftarrows}$ is the quotient of the T presheaf w.r.t. the interreducibility relation. The elements of $T^{/\rightleftarrows}_{\tau,\Gamma}$ are interreducibility classes of terms having the type τ in the typing context Γ. We will use the metavariables $\mathcal{M}$ and $\mathcal{N}$ for these equivalence classes.

The preorder $\twoheadrightarrow$ on $(\!|\lambda|\!)$ terms can be extended to interreducibility classes of $(\!|\lambda|\!)$ terms. Formally, we have $\mathcal{M} \twoheadrightarrow \mathcal{N}$ if there exists $M \in \mathcal{M}$ and $N \in \mathcal{N}$ such that $M \twoheadrightarrow N$. This preorder is antisymmetric and the $\twoheadrightarrow$ relation on interreducibility classes therefore forms a partial order. This means that $(T^{/\rightleftarrows}, \twoheadrightarrow)$ is a presheaf equipped with a partial order.

We can verify that $\rightleftarrows$ is a congruence on the $(V + \Sigma)$-algebra $\mathcal{T}$. All of the operations in the algebra $\mathcal{T}$ construct new $(\!|\lambda|\!)$ terms with the operands as subterms of the constructed term. Let f be an operation of $\mathcal{T}$ and $M_1, \ldots, M_k$, and $N_1, \ldots, N_k$ be $(\!|\lambda|\!)$ terms such that $\forall i.\, M_i \rightleftarrows N_i$. Then we have $f(M_1, \ldots, M_k) \twoheadrightarrow f(N_1, \ldots, N_k)$ because $\forall i.\, M_i \twoheadrightarrow N_i$ and $f(N_1, \ldots, N_k) \twoheadrightarrow f(M_1, \ldots, M_k)$ because $\forall i.\, N_i \twoheadrightarrow M_i$. Therefore, we have $f(M_1, \ldots, M_k) \rightleftarrows f(N_1, \ldots, N_k)$. Since $\rightleftarrows$ is a congruence on $(V + \Sigma)$-algebra, we can quotient it and get a $(V + \Sigma)$-algebra $\mathcal{T}^{/\rightleftarrows}$ whose carrier is the $T^{/\rightleftarrows}$ presheaf.

We now have a $(V + \Sigma)$-algebra, $\mathcal{T}^{/\rightleftarrows}$, whose carrier is equipped with a partial order, $\twoheadrightarrow$. Because the reduction relation $\twoheadrightarrow$ of $(\!|\lambda|\!)$ is closed on contexts, the operations of $\mathcal{T}^{/\rightleftarrows}$ are weakly monotonic: if we replace one of the arguments $\mathcal{M}_i$ in $f(\mathcal{M}_1, \ldots, \mathcal{M}_n)$ with an $\mathcal{M}'_i$ such that $\mathcal{M}_i \twoheadrightarrow \mathcal{M}'_i$, then we will also have $f(\mathcal{M}_1, \ldots, \mathcal{M}_i, \ldots, \mathcal{M}_n) \twoheadrightarrow f(\mathcal{M}_1, \ldots, \mathcal{M}'_i, \ldots, \mathcal{M}_n)$. Therefore, we have a weakly monotonic $(V + \Sigma)$-algebra $\mathcal{T}^{/\rightleftarrows}$.

Definition 3.5.27. *For a given $(V + \Sigma)$-algebra $\mathcal{A}$, a **term-generated assignment** ϕ is an arrow in $(\mathbf{Set}^{\mathbb{F}\downarrow\mathcal{B}})^{\mathcal{B}}$ from the presheaf Z to the presheaf A such that $\phi = !\circ\theta$, where:*

- *θ is an IDTS valuation,[65] i.e. an arrow from Z to $T_\Sigma V$.*
- *$!$ is the unique homomorphism from the initial $(V + \Sigma)$-algebra $T_\Sigma V$ to A.[66]*

To clarify the nomenclature: valuations replace metavariables with terms, assignments replace metavariables with interpretations in some algebra and term-generated assignments are assignments that can only assign an interpretation x if x can be computed as the interpretation of some term.

Definition 3.5.28. *A weakly monotonic $(V + \Sigma)$-algebra $(\mathcal{A}, \geq_A)$ **satisfies an IDTS rewrite rule** $l \to r$, with l and r of type τ, if for all term-generated assignments ϕ of the free metavariables Z in l and r, we have:*

$$!\,\theta^*_{\tau,\Gamma}(l) \geq_{A_{\tau,\Gamma}} !\,\theta^*_{\tau,\Gamma}(r)$$

where $\phi = !\circ\theta$, θ^* is the extension of the valuation θ to meta-terms and Γ is the context regrouping all the free variables exposed by θ.

[65] Same as the CRS valuations introduced in 3.4.1, but typed.

[66] Homomorphisms between Σ-algebras are defined in the same way as homomorphisms for first-order algebras. The term algebra $T_\Sigma V$ is called an initial algebra because we can find a (unique) homomorphism from $T_\Sigma V$ to any other algebra $\mathcal{A}$ that works by interpreting terms from $T_\Sigma V$ using the operations of $\mathcal{A}$.

Definition 3.5.29. *A weakly monotonic $(V+\Sigma)$-algebra $(\mathcal{A}, \geq_A)$ is a quasi-model for the IDTS $(\Sigma, \mathcal{R})$ if $(\mathcal{A}, \geq_A)$ satisfies every rule in $\mathcal{R}$.*

Our weakly monotonic algebra $(\mathcal{T}^{/\rightleftarrows}, \twoheadrightarrow)$ is a quasi-model for the IDTS $(\!| \lambda |\!)_\tau$. The expressions $L = !\,\theta^*_{\tau,\Gamma}(l)$ and $R = !\,\theta^*_{\tau,\Gamma}(r)$ are instances of the left-hand and right-hand side, respectively, of the $(\!| \lambda |\!)$ reduction rule $l \to r$. Therefore, we always have $L \twoheadrightarrow R$.

Labelling Our System

We will now decide how to label the $(\!| \;|\!)$ and $\mathcal{C}$ symbols. The labels we will choose will be the $(\!| \lambda |\!)$ denotations introduced in 3.3.1. We will build up some orders on the denotations that will become crucial later.

Definition 3.5.30. *For each $(\!| \lambda |\!)$ type τ, we define a well-founded strict partial order $>_{[\![\tau]\!]}$ on the set of denotations $[\![\tau]\!]$ by induction on τ.*

- *τ is an atomic type*

  Then $>_{[\![\tau]\!]}$ is the empty relation.

- *$\tau = \alpha \to \beta$*

  $f >_{[\![\tau]\!]} g$ if and only if f and g are both functions (i.e. not $\bot$) and $\forall x \in [\![\alpha]\!].\ f(x) >_{[\![\beta]\!]} g(x)$. The new order is well-founded: any hypothetical descending chain $f_1 >_{[\![\tau]\!]} f_2 >_{[\![\tau]\!]} \ldots$ could be projected to a descending chain $f_1(x) >_{[\![\beta]\!]} f_2(x) >_{[\![\beta]\!]} \ldots$, which is well-founded by induction hypothesis.

- *$\tau = \mathcal{F}_E(\gamma)$*

  Let $E = \{\mathtt{op}_i : \alpha_i \rightarrowtail \beta_i\}_{i \in I}$. The order $>_{[\![\tau]\!]}$ is the smallest transitive relation satisfying the following:

  - $\forall i \in I,\ \forall p \in [\![\alpha_i]\!],\ \forall c \in [\![\mathcal{F}_E(\gamma)]\!]^{[\![\beta_i]\!]},\ \forall x \in [\![\beta_i]\!].\ \mathtt{op}_i(p, c) >_{[\![\tau]\!]} c(x)$

  The proof of the well-foundedness of this relation was given in the definition of the interpretation of a handler (Definition 3.3.6). It relies on the fact that $[\![\mathcal{F}_E(\gamma)]\!]$ is defined as a union of an increasing sequence of sets where $c(x)$ always belongs to a set preceding the one in which $\mathtt{op}_i(p, c)$ appears for the first time.

As our labels, we will use denotations of (possibly open) $(\!| \lambda |\!)$ terms. These objects are functions from $[\![\Gamma]\!]$ to $[\![\tau]\!]$ for some typing context Γ and type τ. We will need to compare denotations of two objects having the same type but not necessarily occurring in the same typing context. We introduce some notation to deal with context and valuation extensions.

Notation 3.5.31. *Let Γ and Δ be typing contexts. The typing context Γ, Δ, the **extension of** Γ **with** Δ, is defined by:*

$$(\Gamma, \Delta)(x) = \begin{cases} \Delta(x), & \text{if } \Delta(x) \text{ is defined} \\ \Gamma(x), & \text{otherwise} \end{cases}$$

Notation 3.5.32. *Let e and d be valuations[67] for the typing contexts Γ and Δ, respectively. The valuation $e + d$ for the context Γ, Δ, called the **extension of** e **with** d, is defined by:*

$$(e + d)(x) = \begin{cases} d(x), & \text{if } x \in \mathrm{dom}(d) \\ e(x), & \text{otherwise} \end{cases}$$

Notation 3.5.33. *We will use the term $D(\tau)$ for the set $\bigcup_\Gamma [\![\tau]\!]^{[\![\Gamma]\!]}$, the set of **possible denotations** of τ-typed $(\!| \lambda |\!)$ terms.*

Definition 3.5.34. *Let τ be a $(\!| \lambda |\!)$ type. The well-founded strict partial order $>_{D(\tau)}$ on the set $D(\tau)$ is defined by:*

[67] Not IDTS valuations, but the valuations used by the denotational semantics in 3.3.1.

- *$f >_{D(\tau)} g$ if and only if:*
  - $f : [\![\Gamma]\!] \to [\![\tau]\!]$
  - $g : [\![\Gamma, \Delta]\!] \to [\![\tau]\!]$
  - $\forall e \in [\![\Gamma]\!],\ \forall d \in [\![\Delta]\!].\ f(e) >_{[\![\tau]\!]} g(e + d)$

We will use the notation $\geq_{D(\tau)}$ for the reflexive closure of $>_{D(\tau)}$.

For every symbol f to label, we will now choose a non-empty well-founded poset $(S_f, \geq_{S_f})$, called the *semantic label set*. In our application of the technique, we will always choose the set of possible denotations of the argument that is being recursively decreased by the function. For the symbols that we do not care to label, we will assume that their semantic label set is the singleton set 1.

- For $(\!|\ |\!)_{\mathtt{op}_1,\ldots,\mathtt{op}_n,\gamma,\delta,E,E'} \in F_{\alpha_1\to(\beta_1\to\mathcal{F}_{E'}(\delta)),\ldots,\alpha_n\to(\beta_n\to\mathcal{F}_{E'}(\delta)),\gamma\to\mathcal{F}_{E'}(\delta),\mathcal{F}_E(\gamma),\mathcal{F}_{E'}(\delta)}$, we take as the semantic label set the poset $D(\mathcal{F}_E(\gamma))$ ordered by $\geq_{D(\mathcal{F}_E(\gamma))}$.
- For $\mathcal{C}_{\alpha,\beta,E} \in F_{\alpha\to\mathcal{F}_E(\beta),\mathcal{F}_E(\alpha\to\beta)}$, we take as the semantic label set the poset $D(\alpha \to \mathcal{F}_E(\beta))$ ordered by $\geq_{D(\alpha\to\mathcal{F}_E(\beta))}$.

Having fixed the semantic label sets, we will now choose the *semantic label maps*. For each symbol $f \in F_{\vec{\alpha_1}\Rightarrow\beta_1,\ldots,\vec{\alpha_n}\Rightarrow\beta_n,\gamma}$ to be labelled, we define a weakly monotonic arrow $\langle\langle - \rangle\rangle^f$ in $\mathbf{Set}^{\mathbb{F}\downarrow\mathcal{B}}$:

$$\langle\langle - \rangle\rangle^f : \delta_{\vec{\alpha_1}} T^{/\rightleftarrows}_{\beta_1} \times \cdots \times \delta_{\vec{\alpha_n}} T^{/\rightleftarrows}_{\beta_n} \longrightarrow K_{S_f}$$

where K_A is the constant presheaf $K_A(\Gamma) = A$. This semantic label map has access to the interpretations of all of the function symbol's arguments and needs to map them to an element of the semantic label set. In our model, the carrier containing the interpretations is the presheaf $T^{/\rightleftarrows}$ of interreducibility classes of $(\!|\lambda|\!)$ terms. However, the interreducibility relation $\rightleftarrows$ will be a congruence for all of the semantic label maps that we will define and so we will define them directly on terms instead of interreducibility classes.

This means that for every $(\!|\ |\!)_{\mathtt{op}_1,\ldots,\mathtt{op}_n,\gamma,\delta,E,E'} \in F_{\alpha_1\to(\beta_1\to\mathcal{F}_{E'}(\delta)),\ldots,\alpha_n\to(\beta_n\to\mathcal{F}_{E'}(\delta)),\gamma\to\mathcal{F}_{E'}(\delta),\mathcal{F}_E(\gamma),\mathcal{F}_{E'}(\delta)}$, we need to give:

$$\langle\langle - \rangle\rangle^{(\!|\ |\!)} : T_{\alpha_1\to(\beta_1\to\mathcal{F}_{E'}(\delta))} \times \cdots \times T_{\alpha_n\to(\beta_n\to\mathcal{F}_{E'}(\delta))} \times T_{\gamma\to\mathcal{F}_{E'}(\delta)} \times T_{\mathcal{F}_E(\gamma)} \longrightarrow K_{D(\mathcal{F}_E(\gamma))}$$

We do so by projecting the last argument, which is a $(\!|\lambda|\!)$ term of type $\mathcal{F}_E(\gamma)$ in the context Γ, and finding its denotation using $[\![-]\!]$.

$$\langle\langle M_1, \cdots, M_n, M_\eta, N \rangle\rangle^{(\!|\ |\!)}_\Gamma = [\![N]\!]$$

We will do the same for the $\mathcal{C}$ symbols. For every $\mathcal{C}_{\alpha,\beta,E} \in F_{\alpha\to\mathcal{F}_E(\beta),\mathcal{F}_E(\alpha\to\beta)}$, we give a:

$$\langle\langle - \rangle\rangle^{\mathcal{C}} : T_{\alpha\to\mathcal{F}_E(\beta)} \to K_{D(\alpha\to\mathcal{F}_E(\beta))}$$

by:

$$\langle\langle M \rangle\rangle^{\mathcal{C}}_\Gamma = [\![M]\!]$$

We can check that interreducibility is indeed a congruence for these semantic label maps: denotations are preserved under reduction (Property 3.3.8), and therefore all the terms in an interreducibility class have the same denotation. This means that we can extend these semantic label maps to $T^{/\rightleftarrows}$, the carrier of our quasi-model $(\mathcal{T}^{/\rightleftarrows}, \twoheadrightarrow)$.

The semantic label maps must also be weakly monotonic. That is a condition that our maps satisfy: whenever we have $\mathcal{M} \twoheadrightarrow \mathcal{N}$, then by Property 3.3.8, the denotations $[\![M]\!]$ and $[\![N]\!]$ for $M \in \mathcal{M}$ and $N \in \mathcal{N}$ will be equal and therefore so will be the labels $\langle\langle\mathcal{M}\rangle\rangle$ and $\langle\langle\mathcal{N}\rangle\rangle$. Since, $\langle\langle\mathcal{M}\rangle\rangle = \langle\langle\mathcal{N}\rangle\rangle$, we have $\langle\langle\mathcal{M}\rangle\rangle \geq \langle\langle\mathcal{N}\rangle\rangle$.

We have now built up enough structure to correctly label our IDTS with denotations. Let us start with the alphabet.

Definition 3.5.35. *Let $\Sigma = (\mathcal{B}, \mathcal{X}, \mathcal{F}, \mathcal{Z})$ be the alphabet of an IDTS $(\Sigma, \mathcal{R})$ and S_f the chosen semantic label sets. The **alphabet of the labelled IDTS** $(\overline{\Sigma}, \overline{\mathcal{R}})$ is the IDTS alphabet $\overline{\Sigma} = (\mathcal{B}, \mathcal{X}, \overline{\mathcal{F}}, \mathcal{Z})$ where:*

- *For every symbol $f \in \mathcal{F}_{\alpha_1,\ldots,\alpha_n,\beta}$ and for every label p in S_f, we will have $f^p \in \overline{\mathcal{F}}_{\alpha_1,\ldots,\alpha_n,\beta}$.*

To complete our new IDTS, we will also have to transform the rules, so we will need a way to label metaterms.

Definition 3.5.36. *Let $\phi : Z \to T^{/\rightleftarrows}$ be a term-generated assignment with $\phi = \,!\circ\theta$. The **labelling map** $\phi^{\mathrm{L}} : M_\Sigma Z \to M_{\overline{\Sigma}} Z$ is the arrow in $(\mathbf{Set}^{\mathbb{F}\downarrow\mathcal{B}})^{\mathcal{B}}$ defined by:*

$$\begin{aligned}
\phi^{\mathrm{L}}_{\tau,\Gamma}(x) &= x \\
\phi^{\mathrm{L}}_{\tau,\Gamma}(Z_{\alpha_1,\ldots,\alpha_n,\beta}(t_1,\ldots,t_n)) &= Z(\phi^{\mathrm{L}}_{\alpha_1,\Gamma}(t_1),\ldots,\phi^{\mathrm{L}}_{\alpha_n,\Gamma}(t_n)) \\
\phi^{\mathrm{L}}_{\tau,\Gamma}(f([\vec{x_1}]t_1,\ldots,[\vec{x_n}]t_n)) &= f^{\langle\langle !\,\theta^*(t_1),\ldots,!\,\theta^*(t_n)\rangle\rangle^f_\Gamma}([\vec{x_1}]\phi^{\mathrm{L}}_{\beta_i,(\Gamma,\vec{x_1}:\vec{\alpha_1})}(t_1),\ldots,[\vec{x_n}]\phi^{\mathrm{L}}_{\beta_n,(\Gamma,\vec{x_n}:\vec{\alpha_n})}(t_n))
\end{aligned}$$

where $f \in F_{\vec{\alpha_1}\Rightarrow\beta_1,\ldots,\vec{\alpha_n}\Rightarrow\beta_n,\tau}$.

The labelling map traverses an IDTS metaterm and replaces unlabelled function symbols from $\mathcal{F}$ with labelled ones from $\overline{\mathcal{F}}$. Note that the term-generated assignment is not used to rewrite the metavariables: the assignment has values in the carrier presheaf of our $(V + \Sigma)$-algebra and it can therefore be something completely different than an IDTS term. The term-generated assignment $\phi = \,!\circ\theta$ is only used when labelling a function symbol. The IDTS valuation θ is used to replace the metavariables in all of the arguments with some specific terms and the resulting IDTS terms are then interpreted in our algebra $\mathcal{T}^{/\rightleftarrows}$ using ! (which turns them into interreducibility classes of $(\!|\lambda|\!)$ terms). These interpretations are then given as arguments to the semantic label map $\langle\langle -\rangle\rangle^f$, which chooses a label from the label set. Note also that there is no case for bare abstraction ($[x]t$). In the theory of higher-order semantic labelling presented in [55], the IDTS is assumed to not contain any bare abstractions: abstractions should always be arguments to function symbols. This is the case in our IDTS $(\!|\lambda|\!)_\tau$.

Knowing how to label metaterms, we can now label the rules of an IDTS.

Definition 3.5.37. *Given an IDTS $(\Sigma, \mathcal{R})$, a $(V + \Sigma)$-algebra M and a choice of semantic label sets S_f and maps $\langle\langle -\rangle\rangle^f$, we define the **rules of the labelled IDTS** $(\overline{\Sigma}, \overline{\mathcal{R}})$ with:*

- $\overline{\mathcal{R}} = \{\phi^{\mathrm{L}}_{\tau,\emptyset}(l) \to \phi^{\mathrm{L}}_{\tau,\emptyset} \mid l \to r : \tau \in \mathcal{R}$, *term-generated assignment* $\phi : Z \to M\}$

The labelled IDTS will multiply the number of rules. For every possible IDTS valuation of the free metavariables of a rule, there will be a new rule in which the function symbols have been labelled using the interpretations of their arguments. As we have done in 3.5.2, we will have to show that termination of this new labelled system gives us termination of the unlabelled one. This is the object of the principal result in [55] (Theorem 3.7):

Theorem 3.5.38. ***Higher-order semantic labelling***
Let M be a quasi-model for an IDTS $(\Sigma, \mathcal{R})$ and $(\overline{\Sigma}, \overline{\mathcal{R}})$ the labelled IDTS with respect to M. Then $(\Sigma, \mathcal{R})$ is terminating if and only if $(\overline{\Sigma}, \overline{\mathcal{R}} \cup \mathrm{Decr})$ is terminating.

Definition 3.5.39. *Given a labelled IDTS alphabet $\overline{\Sigma}$ with semantic label sets S_f, the rules of the IDTS $(\overline{\Sigma}, \mathrm{Decr})$ (called **decreasing rules**) consist of:*

$$f^p([\vec{x_1}]t_1,\ldots,[\vec{x_n}]t_n) \longrightarrow f^q([\vec{x_1}]t_1,\ldots,[\vec{x_n}]t_n)$$

where $f \in F_{\vec{\alpha_1}\Rightarrow\beta_1,\ldots,\vec{\alpha_n}\Rightarrow\beta_n,\gamma}$ and $p >_{S_f} q$.

The decreasing rules allow us to freely adjust the labels on function symbols to fit rewrite rules as long as we do not increase them.

Verifying the General Schema

Now we will retrace the steps we have carried out in 3.5.3, this time with our semantically labelled system $\overline{(\!| \lambda |\!)_\tau}$.

1. every constructor is positive
2. no left-hand side of rule is headed by a constructor
3. both $>_{\mathcal{B}}$ and $>_{\mathcal{F}}$ are well-founded
4. $stat_f = stat_g$ whenever $f =_{\mathcal{F}} g$

First we have to check off the assumptions (A.1) through (A.4), repeated above. The constructors of $\overline{(\!| \lambda |\!)_\tau}$ are the same as the ones in $(\!| \lambda |\!)_\tau$ and so we validate assumption (A.2). It also means that the induced ordering $>_{\mathcal{B}}$ is the same as before and it is therefore still well-founded, so we have the first half of assumption (A.3). Since $>_{\mathcal{B}}$ is still the same, then so is $=_{\mathcal{B}}$, which is used in the definition of positive constructors (Definition 3.5.18). The constructors are therefore still positive as well and we get assumption (A.1).

To verify the second half of assumption (A.3) and assumption (A.4), we will need to investigate the ordering on function symbols $>_{\mathcal{F}}$ and it is here that we will reap the benefits of our labelling. We need to give a well-founded partial order $\geq_{\mathcal{F}}$ on the function symbols such that whenever we have a rule $f(l_1, \ldots, l_n) \to r$, then $f \geq_{\mathcal{F}} g$ for all function symbols g occurring in r. We propose the following relation:[68]

- $(\!| \, |\!)^p_{\mathtt{op}_1,\ldots,\mathtt{op}_n,\gamma,\delta,E,E'} >_{\mathcal{F}} (\!| \, |\!)^q_{\mathtt{op}_1,\ldots,\mathtt{op}_n,\gamma,\delta,E,E'}$ if $p >_{S_{(\!| \, |\!)}} q$
- $(\!| \, |\!)_{\mathtt{op}_1,\ldots,\mathtt{op}_n} >_{\mathcal{F}} \mathtt{op}_i$
- $(\!| \, |\!) >_{\mathcal{F}} \mathtt{ap}$
- $(\!| \, |\!) >_{\mathcal{F}} \lambda$
- $\mathcal{C}^p_{\alpha,\beta,E} >_{\mathcal{F}} \mathcal{C}^q_{\alpha,\beta,E}$ if $p >_{S_{\mathcal{C}}} q$
- $\mathcal{C} >_{\mathcal{F}} \mathtt{op}$
- $\mathcal{C} >_{\mathcal{F}} \eta$
- $\mathcal{C} >_{\mathcal{F}} \lambda$

Whenever we elide indices in the above (for labels, types or the operations in a handler), we assume that they are universally quantified over. This relation is indeed a well-founded strict partial order: $\mathtt{ap}$, $\mathtt{op}$, η and $\mathtt{id}$ are minimal elements and decreasing chains of $(\!| \, |\!)$ or $\mathcal{C}$ symbols are all finite since the underlying semantic label set orderings $>_{S_f}$ are well-founded. This means that our $>_{\mathcal{F}}$ ordering validates the second half of assumption (A.3). We also let $\geq_{\mathcal{F}}$ be the reflexive closure of $>_{\mathcal{F}}$ and then we validate assumption (A.4) because $f =_{\mathcal{F}} g$ only if $f = g$.

We have checked off all of the assumptions and so now we need to check whether the rewrite rules of our labelled IDTS $\overline{(\!| \lambda |\!)_\tau}$ all follow the General Schema. This boils down to checking whether the $\geq_{\mathcal{F}}$ order correctly describes the recursive behavior of our function definitions. Whenever we use a function symbol g in the right-hand side r of a rule $f(l_1, \ldots, l_n) \to r$, we need to show that $f \geq_{\mathcal{F}} g$. Furthermore, if $f =_{\mathcal{F}} g$, we need to show that the arguments passed to g are smaller than the arguments $l_1, \ldots, l_n$ passed to f. However, thanks to the semantic labelling, we will be able to show that for every rule $f(l_1, \ldots, l_n) \to r$, $f >_{\mathcal{F}} g$ for any function symbol g occurring in r.

We first check the rules in Decr. These work out because $>_{\mathcal{F}}$ contains the label ordering for both labelled function symbols, $(\!| \, |\!)$ and $\mathcal{C}$ (i.e. $(\!| \, |\!)^p >_{\mathcal{F}} (\!| \, |\!)^q$ and $\mathcal{C}^p >_{\mathcal{F}} \mathcal{C}^q$ whenever $p > q$).

[68] In Subsection 3.5.3, we said that the ordering $\geq_{\mathcal{F}}$ is induced by the form of the rewrite rules. Actually, we are free to define $\geq_{\mathcal{F}}$ ourselves as long as it validates the assumptions and the General Schema.

$$
\begin{aligned}
(\!|\,|\!)^p(M_i \ldots, M_\eta, N) &\to (\!|\,|\!)^q(M_i \ldots, M_\eta, N) \\
\mathcal{C}^p(M) &\to \mathcal{C}^q(M) \qquad\qquad \text{whenever } p > q
\end{aligned}
$$

Then we check the rules that correspond to reductions in $(\!|\,\lambda\,|\!)$, looking at either the original formulation on Figure 1.4 or the CRS/IDTS versions on Figure 3.1. For most of the rules, it is just a matter of checking that only certain symbols appear in the right-hand sides of certain rules. However, in rules $(\!|\,\mathtt{op}\,|\!)$, $(\!|\,\mathtt{op}'\,|\!)$ and $C_{\mathtt{op}}$, we have the same (unlabelled) symbol on both the left-hand side and the right-hand side of the rule. In these cases, we will need to prove that the label on the right-hand side occurrence is strictly smaller than the label on the left-hand side occurrence.

We will start with the rules $(\!|\,\mathtt{op}\,|\!)$ and $(\!|\,\mathtt{op}'\,|\!)$.

$$
\begin{aligned}
&(\!|\,(\mathtt{op}_i\colon M_i)_{i\in I},\ \eta\colon M_\eta\,|\!)^p\,(\mathtt{op}_j\,N_{\mathrm{p}}\,(\lambda x.\,N_{\mathrm{c}})) \to M_j\,N_{\mathrm{p}}\,(\lambda x.\,(\!|\,(\mathtt{op}_i\colon M_i)_{i\in I},\ \eta\colon M_\eta\,|\!)^q\,N_{\mathrm{c}}) && \text{where } j \in I \\
&(\!|\,(\mathtt{op}_i\colon M_i)_{i\in I},\ \eta\colon M_\eta\,|\!)^p\,(\mathtt{op}_j\,N_{\mathrm{p}}\,(\lambda x.\,N_{\mathrm{c}})) \to \mathtt{op}_j\,N_{\mathrm{p}}\,(\lambda x.\,(\!|\,(\mathtt{op}_i\colon M_i)_{i\in I},\ \eta\colon M_\eta\,|\!)^q\,N_{\mathrm{c}}) && \text{where } j \notin I
\end{aligned}
$$

In both cases, the $(\!|\,|\!)$ on the left-hand side is applied to $\Gamma \vdash \mathtt{op}_j\,N_{\mathrm{p}}\,(\lambda x.\,N_{\mathrm{c}}) : \mathcal{F}_E(\gamma)$ whereas the $(\!|\,|\!)$ on the right-hand side is applied to $\Gamma, x : \beta_j \vdash N_{\mathrm{c}} : \mathcal{F}_E(\gamma)$ where $\mathtt{op}_j : \alpha_j \rightarrowtail \beta_j \in E$. The label p of the left $(\!|\,|\!)$ will be the denotation $[\![\mathtt{op}_j\,N_{\mathrm{p}}\,(\lambda x.\,N_{\mathrm{c}})]\!]$ whereas the label q of the right $(\!|\,|\!)$ will be the denotation $[\![N_{\mathrm{c}}]\!]$.

The ordering on these labels is the $>_{D(\mathcal{F}_E(\gamma))}$ ordering. For the first to be greater than the second, we will need to prove for all $e \in [\![\Gamma]\!]$ and all $d \in [\![\beta_j]\!]$ that $[\![\mathtt{op}_j\,N_{\mathrm{p}}\,(\lambda x.\,N_{\mathrm{c}})]\!](e) >_{[\![\mathcal{F}_E(\gamma)]\!]} [\![N_{\mathrm{c}}]\!](e[x := d])$.

$$
[\![\mathtt{op}_j\,N_{\mathrm{p}}\,(\lambda x.\,N_{\mathrm{c}})]\!](e) = \mathtt{op}_j([\![N_{\mathrm{p}}]\!](e), \lambda X.\,([\![N_{\mathrm{c}}]\!](e[x := X])))
$$

From the definition of $>_{[\![\mathcal{F}_E(\gamma)]\!]}$ (Definition 3.5.30), we know that for all $d \in [\![\beta_j]\!]$, $\mathtt{op}_j([\![N_{\mathrm{p}}]\!](e), (\lambda X.\,[\![N_{\mathrm{c}}]\!](e[x := X]))) >_{[\![\mathcal{F}_E(\gamma)]\!]} [\![N_{\mathrm{c}}]\!](e[x := d])$ which is exactly what we wanted to show.

Now we look at the $C_{\mathtt{op}}$ rule.

$$
\mathcal{C}^p\,(\lambda x.\,\mathtt{op}\,M_{\mathrm{p}}\,(\lambda y.\,M_{\mathrm{c}})) \to \mathtt{op}\,M_{\mathrm{p}}\,(\lambda y.\,\mathcal{C}^q\,(\lambda x.\,M_{\mathrm{c}}))
$$

On the left-hand side, $\mathcal{C}$ is applied to $\Gamma \vdash \lambda x.\,\mathtt{op}\,M_{\mathrm{p}}\,(\lambda y.\,M_{\mathrm{c}}) : \gamma \to \mathcal{F}_E(\delta)$, and on the right-hand side, it is applied to $\Gamma, y : \beta \vdash \lambda x.\,M_{\mathrm{c}} : \gamma \to \mathcal{F}_E(\delta)$ where $\mathtt{op} : \alpha \rightarrowtail \beta \in E$. The label p of the left $\mathcal{C}$ is the denotation $[\![\lambda x.\,\mathtt{op}\,M_{\mathrm{p}}\,(\lambda y.\,M_{\mathrm{c}})]\!]$ while the label q of the right-hand side $\mathcal{C}$ is $[\![\lambda x.\,M_{\mathrm{c}}]\!]$.

These labels are ordered by the $>_{D(\gamma\to\mathcal{F}_E(\delta))}$ ordering under which $p > q$ if for all $e \in [\![\Gamma]\!]$ and all $d \in [\![\beta]\!]$, we have $p(e) >_{[\![\gamma\to\mathcal{F}_E(\delta)]\!]} q(e[y := d])$. Then to show that $p(e) >_{[\![\gamma\to\mathcal{F}_E(\delta)]\!]} q(e[y := d])$, we will need to show that they are both functions and that for all $c \in [\![\gamma]\!]$, we have $p(e)(c) >_{\mathcal{F}_E(\delta)} q(e[y := d])(c)$.

$$
\begin{aligned}
[\![\lambda x.\,\mathtt{op}\,M_{\mathrm{p}}\,(\lambda y.\,M_{\mathrm{c}})]\!](e)(c) &= (\lambda X.\,([\![\mathtt{op}\,M_{\mathrm{p}}\,(\lambda y.\,M_{\mathrm{c}})]\!](e[x := X])))(c) \\
&= [\![\mathtt{op}\,M_{\mathrm{p}}\,(\lambda y.\,M_{\mathrm{c}})]\!](e[x := c]) \\
&= \mathtt{op}([\![M_{\mathrm{p}}]\!](e[x := c]), \lambda Y.\,([\![M_{\mathrm{c}}]\!](e[x := c, y := Y]))) \\
[\![\lambda x.\,M_{\mathrm{c}}]\!](e[y := d])(c) &= (\lambda X.\,([\![M_{\mathrm{c}}]\!](e[y := d, x := X])))(c) \\
&= [\![M_{\mathrm{c}}]\!](e[y := d, x := c]) \\
&= [\![M_{\mathrm{c}}]\!](e[x := c, y := d])
\end{aligned}
$$

We elaborate both of the expressions. The last step in rewriting $[\![\lambda x.\,M_{\mathrm{c}}]\!](e[y := d])(c)$ is due to $e[x := X, y := Y] = e[y := Y, x := X]$ for distinct variables x and y. From the definition of $>_{\mathcal{F}_E(\delta)}$ (Definition 3.5.30), we get that for all $d \in [\![\beta]\!]$, $\mathtt{op}([\![M_{\mathrm{p}}]\!](e[x := c]), \lambda Y.\,([\![M_{\mathrm{c}}]\!](e[x := c, y := Y]))) > ([\![M_{\mathrm{c}}]\!])(e[x := c, y := d])$, which is exactly what we need.

Having shown that the function symbols that head the left-hand sides of rules are strictly larger (in a well-founded poset) than the function symbols that occur in the right-hand sides gives us termination for the labelled IDTS $\overline{(\!|\,\lambda\,|\!)_\tau}$ via Theorem 3.5.12.

Theorem 3.5.40. *(Termination of $\overline{(\!|\lambda|\!)_\tau}$)*
The reduction relation induced by the labelled IDTS $\overline{(\!|\lambda|\!)_\tau}$ is terminating.[69]

Proof. Proof given above by the application of the General Schema presented in [19]. □

Corollary 3.5.41. *(Termination of $(\!|\lambda|\!)_\tau$)*
The reduction relation induced by the IDTS $(\!|\lambda|\!)_\tau$ is terminating.

Proof. By Theorem 3.5.40 and Theorem 3.5.38. □

Corollary 3.5.42. *(Termination of $(\!|\lambda|\!)_{-\eta}$)*
The reduction relation of $(\!|\lambda|\!)$ without η-reduction is terminating.

Proof. By Corollary 3.5.41 and Lemma 3.5.11. □

3.5.5 Putting η Back in $(\!|\lambda|\!)$

We have shown termination for $(\!|\lambda|\!)_{-\eta}$. We know that the η-reduction on $(\!|\lambda|\!)$ is terminating: it decreases the number of λ-abstractions in the term by one in each step. We would now like to show that the combination of $(\!|\lambda|\!)_{-\eta}$ and η is terminating as well.

Note that we could not have used the General Schema to prove that $(\!|\lambda|\!)$ with η is terminating. The General Schema does not admit η-reduction. The left-hand side of every rule needs to be headed by a function symbol which is not a contructor. If we tried declaring that λ is not a constructor, we would run into problems with the notion of accessibility. When accessing the metavariables of the left-hand side of a rule, we can access all of the arguments of a constructor but we can only access the arguments of a non-constructor symbol that has a basic type. The type of the argument of $\lambda_{\alpha,\beta}$ is $\alpha \Rightarrow \beta$ and so we could not access the arguments of λ in our rules (which would break the β rule, η rule and the $\mathcal{C}$ rules).

Termination is generally not a modular property of higher-order rewriting systems [7]. Our plan will be to show that η-reduction does not interfere with the rewrite rules of $(\!|\lambda|\!)_{-\eta}$. Then we will be able to take any reduction chain in $(\!|\lambda|\!)$ and pull out from it a chain which only uses rules from $(\!|\lambda|\!)_{-\eta}$. Since this chain must be finite due to the termination of $(\!|\lambda|\!)_{-\eta}$, we will have a proof of finiteness for the reduction chain in $(\!|\lambda|\!)$.

Definition 3.5.43. *An **n-ary evaluation context** C^n is a $(\!|\lambda|\!)$ term in which n disjoint subterms have been replaced with the symbol $[]$. We write $C^n[M]$ for the term in which all of the occurrences of the symbol $[]$ have been replaced with M.*

Lemma 3.5.44. *Exchanging η with $(\!|\lambda|\!)_{-\eta}$*
For every well-typed reduction chain $s \to_\eta t \to_{(\!|\lambda|\!)_{-\eta}} u$, there exists a well-typed reduction chain $s \to^+_{(\!|\lambda|\!)_{-\eta}} t' \to^*_\eta u$.

Proof. We will consider all the possible relative positions of the contractum of the first reduction and the redex for the second reduction within t.

- Assume the two are disjoint, i.e. $s = C[M, N]$, $t = C[M', N]$ with $M \to_\eta M'$ and $u = C[M', N']$ with $N \to_{(\!|\lambda|\!)_{-\eta}} N'$. Then we can easily reorder the two reductions, producing the chain $C[M, N] \to_{(\!|\lambda|\!)_{-\eta}} C[M, N'] \to_\eta C[M', N']$.

- Assume that the contractum of the first reduction contains the redex for the second reduction, i.e. $s = C[M]$, $t = C[D[N]]$ with $M \to_\eta D[N]$ and $u = C[D[N']]$ with $N \to_{(\!|\lambda|\!)_{-\eta}} N'$. Since M is an η-redex, $M = \lambda x.\, D[N]\, x$. We can now build the chain $C[\lambda x.\, D[N]\, x] \to_{(\!|\lambda|\!)_{-\eta}} C[\lambda x.\, D[N']\, x] \to_\eta C[D[N']]$.

[69] This result can be extended to $(\!|\lambda|\!)$ with sums and products. The pair construction $\langle -, - \rangle$ and the injections inl and inr will be the constructors for $\alpha \times \beta$ and $\alpha + \beta$, respectively, with $\alpha <_{\mathcal{B}} \alpha \times \beta$, $\beta <_{\mathcal{B}} \alpha \times \beta$, $\alpha <_{\mathcal{B}} \alpha + \beta$ and $\beta <_{\mathcal{B}} \alpha + \beta$. All of the rules defining case analysis and projections π_1 and π_2 satisfy the General Schema.

- Assume that the redex for the second reduction contains the contractum of the first reduction, i.e. $s = C[D[M]]$, $t = C[D[M']]$ with $M \to_\eta M'$ and $u = C[N']$ with $D[M'] \to_{(\!|\lambda|\!)_{-\eta}} N'$. Let R be the rule used in $D[M'] \to_{(\!|\lambda|\!)_{-\eta}} N$. We will now distinguish two scenarios:
  - The occurrence of M' in $D[M']$ is matched by a metavariable in the left-hand side of rule R.
    The R-redex N' of $D[M']$ will be a term $E^n[M', \ldots, M']$ where E^n is an n-ary context for some n which depends on the rule R and the metavariable that was matched.[70] Furthermore, we can replace M' with any other term of the same type and the reduction will still go through, e.g. notably $D[M] \to_R E^n[M, \ldots, M]$. We can now build our chain $C[D[M]] \to_{(\!|\lambda|\!)_{-\eta}} C[E^n[M, \ldots, M]] \to_\eta^* C[E^n[M', \ldots, M']] = C[N']$.
  - The occurrence of M' in $D[M']$ is not matched by a metavariable.
    M' is an η-contractum and must therefore have a function type. If we investigate the left-hand sides of all the rewriting rules in $(\!|\lambda|\!)_{-\eta}$ and search for terms that have a function type, we end up with:[71]
    * $D = [\,]\,N$ and $R = \beta$
    * $D = \mathcal{C}\,[\,]$ and $R = \mathcal{C}_{\mathsf{op}}$ or $R = \mathcal{C}_\eta$

    We note that in all of these rules, the symbol which replaces $[\,]$ must be a λ-abstraction. Therefore, if $D[M'] \to_R N$, then $M' = \lambda x.\, M''$. From $M \to_\eta M'$, we also know that $M = \lambda x.\, M'\, x$. We can replace this step by a β-reduction: $M = \lambda x.\, (\lambda x.\, M'')\, x \to_\beta \lambda x.\, M'' = M'$. The β rule is a part of $(\!|\lambda|\!)_{-\eta}$ and so we can now build the chain $C[D[M]] \to_{(\!|\lambda|\!)_{-\eta}} C[D[M']] \to_{(\!|\lambda|\!)_{-\eta}} C[N']$.

□

Lemma 3.5.45. ***Pulling a $(\!|\lambda|\!)_{-\eta}$ link from a $(\!|\lambda|\!)$ chain***

Let $t_1 \to t_2 \to \ldots$ be an infinite reduction chain in $(\!|\lambda|\!)$. Then there exists another infinite reduction chain $u_1 \to u_2 \to \ldots$ in $(\!|\lambda|\!)$ and $t_1 \to_{(\!|\lambda|\!)_{-\eta}} u_1$.

Proof. The goal of this lemma is to show that we can find an $(\!|\lambda|\!)_{-\eta}$ link in every infinite $(\!|\lambda|\!)$ and move it to the beginning of the chain.

An infinite chain in $(\!|\lambda|\!)$ must use a rule from $(\!|\lambda|\!)_{-\eta}$, otherwise it would be an η chain and those cannot be infinite since η is terminating.

Let $t_k \to t_{k+1}$ be the first link in the chain that uses a rule from $(\!|\lambda|\!)_{-\eta}$. We will prove this lemma by induction on k.

If $k = 1$, then we can use the chain $t_2 \to t_3 \to \ldots$ which also uses rules from $(\!|\lambda|\!)_{-\eta}$ infinitely often and which satisfies $t_1 \to_{(\!|\lambda|\!)_{-\eta}} t_2$.

If $k > 1$, then we replace the segment $t_{k-1} \to_\eta t_k \to_{(\!|\lambda|\!)_{-\eta}} t_{k+1}$ with the segment $t_{k-1} \to^+_{(\!|\lambda|\!)_{-\eta}} t_k \to^*_\eta t_{k+1}$ using Lemma 3.5.44. By induction hypothesis, the chain $t_1 \to \ldots \to t_{k-1} \to^+_{(\!|\lambda|\!)_{-\eta}} t_k \to^*_\eta t_{k+1} \to t_{k+2} \to \ldots$ gives us the necessary chain $u_1 \to u_2 \to \ldots$ with $t_1 \to_{(\!|\lambda|\!)_{-\eta}} u_1$. □

Theorem 3.5.46. ***Termination of $(\!|\lambda|\!)$***

The reduction relation $\to$ on $(\!|\lambda|\!)$ terms given by the rules in Figure 1.4 is terminating.

Proof. We will prove this theorem by contradiction. Let $t_1 \to t_2 \to \ldots$ be an infinite reduction chain in $(\!|\lambda|\!)$. Since we have an infinite chain in $(\!|\lambda|\!)$, we can iterate Lemma 3.5.45 to get an infinite sequence of chains such that the first element of every chain reduces via $(\!|\lambda|\!)_{-\eta}$ to the first element of the next chain in the sequence. The first elements of these chains form an infinite reduction chain $(\!|\lambda|\!)_{-\eta}$, which is in contradiction with the termination of $(\!|\lambda|\!)_{-\eta}$. □

Theorem 3.5.47. ***Strong normalization of $(\!|\lambda|\!)$***

There are no infinite reduction chains in $(\!|\lambda|\!)$ and all maximal reduction chains originating in a $(\!|\lambda|\!)$ term M terminate in the same term, the normal form of M.

[70] Rules like $(\!|\eta|\!)$ can delete metavariables ($n = 0$ for the metavariable M_η), while others, like $(\!|\ |\!)$, can copy them ($n = 2$ for the variable M_j)

[71] The case of $D = [\,]$ is not considered, because it is covered by the case where the contractum of the first reduction containts the redex of the second reduction.

Proof. The lack of infinite reduction chains is due to termination of $(\!| \lambda |\!)$ (Theorem 3.5.46) and the fact that all maximal reduction chains lead to the same term is entailed by confluence of $(\!| \lambda |\!)$ (Theorem 3.4.18). □

4

Continuations

Continuations are a fundamental notion in computer science. They subsume all of the monadic structures that we are interested in [44] and they are central to the meaning of linguistic expressions [14]. In this chapter, we will show the connection between calculi containing operators for manipulating continuations and our $(\!| \lambda |\!)$ calculus. Throughout the chapter, we will introduce different calculi and map their terms and reduction relations onto the terms and reduction relation of $(\!| \lambda |\!)$. For one of the calculi with control operators, we also introduce a type system, which is a simplification of the one found in [33] and we map it onto the type system of $(\!| \lambda |\!)$. See Figure 4.1 for a high-level look at the structure of the chapter.

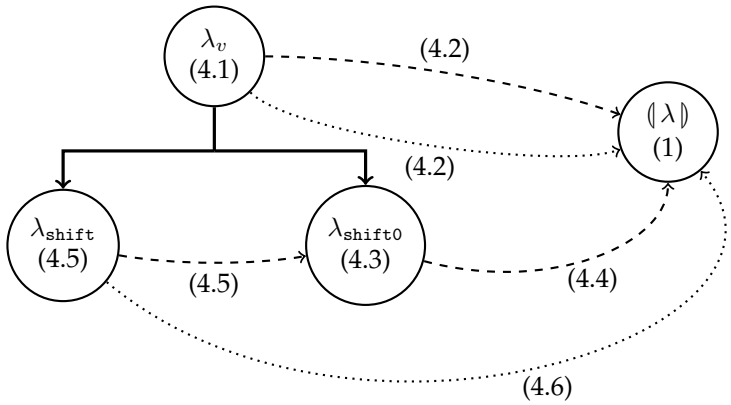


Figure 4.1: The plan of Chapter 4. The nodes are calculi, solid edges correspond to extensions of calculi, dashed edges correspond to translations of rewriting relations and dotted edges correspond to translations of type systems.

Contents

4.1 Introducing Call-by-Value **86**
4.2 Simulating Call-by-Value **86**
4.3 Introducing Control Operators **88**
4.4 Simulating `shift0` and `reset0` **89**
4.5 Turning to `shift` and `reset` **92**
4.6 Considering Types **94**
4.7 Other Control Operators **101**

4.1 Introducing Call-by-Value

The nature of control operators is that their evaluation depends on their context. In order for a language with such control operators to be deterministic, it must have a fixed evaluation order. So in order to set up the stage for our study of delimited control, we will start by simulating the call-by-value λ-calculus in $(\!| \lambda |\!)$.

First, we introduce some key notions of call-by-value and ordered evaluation in the call-by-value λ-calculus.

We will single out some of the terms in λ_v and call them values.

Definition 4.1.1. *The following grammar defines the* ***terms*** *of λ_v (metavariables M and N) and the* ***values*** *(metavariable V).*

$$\begin{aligned} V &::= \lambda x.\, M \\ &\mid x \\ M, N &::= V \\ &\mid (M\, N) \end{aligned}$$

The idea behind this distinction is that values (V) are terms that have already been reduced/evaluated. This distinction will become useful in definining the following notion:

Definition 4.1.2. *We define an* ***evaluation context*** *C as a structure formed by the following grammar:*

$$\begin{aligned} C &::= [] \\ &\mid (C\, M) \\ &\mid (V\, C) \end{aligned}$$

We write $C[M]$ to designate the term that you obtain by replacing the $[]$ in C with M.

We now have all the pieces in play to be able to define the semantics of λ_v.

Definition 4.1.3. *A term M* ***reduces to*** *a term N in one step, written as $M \to N$, when the pair $M \to N$ matches this pattern:*

$$C[(\lambda x.\, M)\, V] \to_\beta C[M[x := V]]$$

Here we see that we only substitute *values* for the variables in a λ-abstraction. Also note that we can only perform reductions inside an evaluation context. Given our definition of C, this enforces a left-to-right evaluation order and also prohibits evaluation under a λ-abstraction.

4.2 Simulating Call-by-Value

We first present the translation from λ_v to $(\!| \lambda |\!)$ and then we elaborate on it.

Definition 4.2.1. *Let M be a term of λ_v. We define its* ***interpretation*** *in $(\!| \lambda |\!)$, written as $[\![M]\!]$:*

$$\begin{aligned} [\![x]\!] &= \eta\, x \\ [\![\lambda x.\, M]\!] &= \eta\, (\lambda x.\, [\![M]\!]) \\ [\![M\, N]\!] &= [\![M]\!] \gg\!= (\lambda m.\, [\![N]\!] \gg\!= (\lambda n.\, m\, n)) \end{aligned}$$

An expression of λ_v is modelled in $(\!| \lambda |\!)$ as a computation. The values form a special case since they are all interpreted as pure computations, terms of the form $(\eta\, M)$ for some M. In interpreting an application $(M\, N)$, we first evaluate M and then N, reflecting the behavior we have defined for λ_v above.

Before we show that this translation is indeed a faithful one, we will discuss the types of the interpretations to get a better understanding of the structures involved.

λ_v can be typed with the type system of the simply-typed λ-calculus. A well-typed λ_v term will then yield a well-typed $(\!| \lambda |\!)$ term since our translation satisifies the following property.

Property 4.2.2. *Let M be λ_v term, α a simple type, Γ a simply-typed environment and E an effect signature. Then the following implication holds.*

$$\Gamma \vdash M : \alpha \quad \Rightarrow \quad [\![\Gamma]\!] \vdash [\![M]\!] : \mathcal{F}_E([\![\alpha]\!])$$

Proof. By structural induction on the structure of M. The Definition 4.2.3 of $[\![.]\!]$ for simple types and simply-typed environments is given just below. □

Definition 4.2.3. *We define the* ***interpretation of types and environments*** *using the following formulas. ν stands for an atomic type and $\emptyset$ for the empty environment.*

$$\begin{aligned}
[\![\alpha \to \beta]\!] &= [\![\alpha]\!] \to \mathcal{F}_E([\![\beta]\!]) \\
[\![\nu]\!] &= \nu \\
[\![\Gamma, x : \alpha]\!] &= [\![\Gamma]\!], x : [\![\alpha]\!] \\
[\![\emptyset]\!] &= \emptyset
\end{aligned}$$

We see that we model λ_v expressions of type α using computations that yield values of type $[\![\alpha]\!]$. $[\![.]\!]$ translates the function type so that it takes values but produces computations (since the body of a λ-abstraction can in general be any expression and the denotation of an expression is a computation).

To show that our translation simulates the behavior of λ_v, we will prove that any reduction chain $M \twoheadrightarrow N$ in λ_v gives rise to a reduction chain $[\![M]\!] \twoheadrightarrow [\![N]\!]$ in $(\!| \lambda |\!)$. We start by proving $[\![(\lambda x.\, M)\, V]\!] \twoheadrightarrow [\![M[x := V]]\!]$.

Property 4.2.4. *Let M be a λ_v term and V a λ_v value. Then the following reduction chain exists in $(\!| \lambda |\!)$:*
$[\![(\lambda x.\, M)\, V]\!] \twoheadrightarrow [\![M[x := V]]\!]$

Proof.

$$\begin{aligned}
[\![(\lambda x.\, M)\, V]\!] &= [\![\lambda x.\, M]\!] \gg\!= (\lambda m.\, [\![V]\!] \gg\!= (\lambda n.\, m\, n)) && (1)\\
&= (\eta\, (\lambda x.\, [\![M]\!])) \gg\!= (\lambda m.\, (\eta\, v) \gg\!= (\lambda n.\, m\, n)) && (2)\\
&\to_{\eta.\gg\!=,\beta} (\eta\, v) \gg\!= (\lambda n.\, (\lambda x.\, [\![M]\!])\, n) && (3)\\
&\to_{\eta.\gg\!=,\beta} (\lambda x.\, [\![M]\!])\, v && (4)\\
&\to_{\beta} [\![M]\!][x := v] && (5)\\
&= [\![M[x := V]]\!] && (6)
\end{aligned}$$

where $[\![V]\!] = \eta\, v$. We first expand the definition of $[\![.]\!]$ for the application, the abstraction and the argument value (we note that $[\![V]\!]$ is always equal to $\eta\, v$ for some v). Since we have two occurrences of a pure computation being piped into a bind ($(\eta\, x) \gg\!= k$), we can simplify using $\eta.\gg\!=$ and β. On line 5, we finally get to the point where we perform the $(\!| \lambda |\!)$ β-reduction that actually corresponds to the λ_v β-reduction that we are modelling.

On line 6, we push the substitution under the $[\![.]\!]$ operator. This is valid, since any free x in $[\![M]\!]$ must have originated as a translation of a free x in M. In the first expression, such an x would get interpreted as $\eta\, x$ and then x would get replaced with v to get $\eta\, v$. In the second expression, the x would first get replaced by V and then V would be interpreted as $\eta\, v$. In both cases, we get the same result.

□

We have proven that our simulation preserves the reduction $(\lambda x.\, M)\, V \to M[x := V]$. However, it might seem that our work is not over since this is just a special case of the rule in λ_v, which licenses the reduction $C[(\lambda x.\, M)\, V] \to C[M[x := V]]$. Nevertheless, our calculus is pure and lets us perform reductions in any syntactic context (see the notion of context closure from 1.4). This means that whenever we have $[\![M]\!] \to [\![N]\!]$, we also always have $[\![C]\!][[\![M]\!]] \to [\![C]\!][[\![N]\!]]$,[72] which is the same as $[\![C[M]]\!] \to [\![C[N]]\!]$.

Corollary 4.2.5. *Let M and N be terms of λ_v such that $M \twoheadrightarrow N$. Then we have that $[\![M]\!] \twoheadrightarrow [\![N]\!]$ in $(\!|\, \lambda \,|\!)$.*

We have shown that our translation *preserves* reduction chains. Can we get anything stronger?

The inverse is not true in this case. Consider the example of $M = \lambda x.\, ((\lambda y.\, y)\, x)$ and $N = \lambda x.\, x$. If we take $[\![M]\!]$ and perform some reductions, we arrive at $\eta\, (\lambda x.\, \eta\, x)$. However, this is equal to $[\![N]\!]$ and so we have $[\![M]\!] \twoheadrightarrow [\![N]\!]$. This is the case even though M does not reduce to N in λ_v (λ_v does not allow reductions inside a λ-abstraction). Our interpretation will end up licencing an equality between M and N even though the language we were modelling (λ_v) does not equate them (in a way, this interpretation is complete but not sound). While this makes the interpretation less appealing, we still elaborate it because it is instructive in showing the similarity between delimited control and effect handlers.

To conclude this section, we have defined a translation from λ_v to $(\!|\, \lambda \,|\!)$ and proven that it preserves types and reductions. This means that we have a way of simulating simply-typed λ_v in our typed $(\!|\, \lambda \,|\!)$ and a way of simulating untyped λ_v in an untyped version of $(\!|\, \lambda \,|\!)$.

4.3 Introducing Control Operators

Now that we have introduced how a λ-calculus with a notion of evaluation order is to be interpreted in $(\!|\, \lambda \,|\!)$, we are ready to introduce control operators. Our claim is that the effects and handlers of $(\!|\, \lambda \,|\!)$ are very close to delimited continuations.[73] The pair of operators that resembles the behavior of handlers the most are the operators `shift0` and `reset0`.

Definition 4.3.1. *We define the **terms** (M, N) and values V of $\lambda_{\texttt{shift0}}$ using the following grammar:*

$$
\begin{aligned}
V &::= \lambda x.\, M \\
&\mid\ x \\
M, N &::= V \\
&\mid\ (M\, N) \\
&\mid\ (\texttt{shift0}\, M) \\
&\mid\ (\texttt{reset0}\, M)
\end{aligned}
$$

The terms of this new calculus, $\lambda_{\texttt{shift0}}$, are just the terms of λ_v extended with the operators `shift0` and `reset0`. Before we give the reduction rules, we will also have to refine our notion of an evaluation context.

[72] We have not defined the $[\![.]\!]$ interpretation operator for contexts. Contexts are simply terms with a $[]$ inside and so we just extend the $[\![.]\!]$ interpretation of terms with the clause $[\![[]]\!] = []$.

[73] This analogy originates with Andrej Bauer who says that effects and handlers are to delimited continuations what while loops or if-then-else statements are to gotos [15].

Definition 4.3.2. *We define* ***evaluation contexts*** *(C) and* ***evaluation frames*** *(F) the following way:*

$$
\begin{aligned}
C ::= {} & [] \\
& \mid (C\,M) \\
& \mid (V\,C) \\
& \mid (\texttt{shift0}\,C) \\
& \mid (\texttt{reset0}\,C) \\
F ::= {} & [] \\
& \mid (F\,M) \\
& \mid (V\,F) \\
& \mid (\texttt{shift0}\,F)
\end{aligned}
$$

For evaluation contexts (C), we add rules saying that when evaluating applications of either `shift0` or `reset0`, we can evaluate their arguments. We also introduce a notion of an evaluation frame (F). Similarly to how values are a subset of terms, evaluation frames are a subset of evaluation contexts. A frame is a context which does not embed [] inside a `reset0`.

Definition 4.3.3. *A term M* ***reduces to*** *a term N in $\lambda_{\texttt{shift0}}$ in one step whenever M and N match one of the patterns below:*

$$
\begin{array}{rcl}
C[(\lambda x.\,M)\,V] & \to_{\beta} & C[M[x := V]] \\
C[\texttt{reset0}\,V] & \to_{\texttt{reset0}} & C[V] \\
C[\texttt{reset0}\,(F[\texttt{shift0}\,V])] & \to_{\texttt{shift0}} & C[V\,(\lambda x.\,\texttt{reset0}\,(F[x]))]
\end{array}
$$

We keep the reduction rule β and we add two new rules for `reset0` and `shift0`. The rule for `reset0` makes `reset0` look redundant but its importance shows up in the rule for `shift0`. In the `shift0` rule, we have an application of `shift0` buried inside the context $C[\texttt{reset0}\,F]$. This kind of context corresponds to a context which embeds [] inside at least one `reset0`. The F then corresponds to the frame which separates the `shift0` from the nearest enclosing `reset0`. The only role of `reset0` is thus to serve as a kind of marker to delimit the context/continuation F of `shift0`. The argument to `shift0` then receives this continuation F, composed with `reset0`, as its argument.[74]

4.4 Simulating `shift0` and `reset0`

To simulate $\lambda_{\texttt{shift0}}$ in $(\!|\,\lambda\,|\!)$, all we have to do is extend the simulation of λ_v with interpretations of the two new syntactic forms.

Definition 4.4.1. *Let M be a term of $\lambda_{\texttt{shift0}}$. We define its* ***interpretation*** *$[\![M]\!]$ as an extension of the interpretation defined for λ_v with the following clauses:*

$$
\begin{aligned}
[\![\texttt{shift0}\,M]\!] &= [\![M]\!] \gg\!= (\lambda m.\,\texttt{shift0}!\,m) \\
[\![\texttt{reset0}\,M]\!] &= (\!|\,\texttt{shift0}\colon (\lambda ck.\,c\,k)\,|\!)\,[\![M]\!]
\end{aligned}
$$

The above translation also gives us a general template for simulating effectful calculi in $(\!|\,\lambda\,|\!)$. Amongst the impure operators of a calculus, we identify those that manipulate a context (raising an exception, modifying a variable, reading a dynamically bound value, accessing the continuation…) and those that establish a context (exception handlers, transactions, binders for dynamic variables, continuation delimiters such as `prompt` or `reset`…). Operators that manipulate the context are translated into operations in $(\!|\,\lambda\,|\!)$, i.e. $[\![\texttt{op}\,M]\!] = [\![M]\!] \gg\!= (\lambda m.\,\texttt{op}!\,m)$. Operators that establish a context are translated into handlers, i.e. $[\![\texttt{op}\,M]\!] = (\!|\ldots|\!)\,[\![M]\!]$.

[74] You may already start to see similarities with the $(\!|\,\texttt{op}\,|\!)$ rule of $(\!|\,\lambda\,|\!)$ and why we chose `shift0` and `reset0` in particular.

To show that this simulation is faithful, we will prove that $M \to N$ in $\lambda_{\texttt{shift0}}$ implies $[\![M]\!] \leftrightarrow\!\!\!\!\twoheadrightarrow [\![N]\!]$ in $(\!|\lambda|\!)$. $[\![M]\!] \leftrightarrow\!\!\!\!\twoheadrightarrow [\![N]\!]$ says that we can go from $[\![M]\!]$ to $[\![N]\!]$ through a series of reductions and expansions, i.e. $[\![M]\!]$ and $[\![N]\!]$ are convertible.[75] In this case, our simulation property will prove that equivalence given by the calculus' equational theory is preserved, i.e. $M = N$ in $\lambda_{\texttt{shift0}}$ implies $[\![M]\!] = [\![N]\!]$ in $(\!|\lambda|\!)$ (where $X = Y$ is to be read as $X \leftrightarrow\!\!\!\!\twoheadrightarrow Y$).

Property 4.4.2. *Let M and N be terms of $\lambda_{\texttt{shift0}}$. If $M \to N$, then $[\![M]\!] \leftrightarrow\!\!\!\!\twoheadrightarrow [\![N]\!]$.*

Proof. We have three reduction rules to tackle: $\to_\beta$, $\to_{\texttt{reset0}}$ and $\to_{\texttt{shift0}}$. We have proven the case of $\to_\beta$ in 4.2 and that proof still holds in this extended interpretation. We also reuse our observation from 4.2 that in order to prove $[\![C[M]]\!] \twoheadrightarrow [\![C[N]]\!]$, it is enough to prove $[\![M]\!] \twoheadrightarrow [\![N]\!]$. The case of $\to_{\texttt{reset0}}$ is a simple one so we will deal with that first:

$$
\begin{aligned}
[\![\texttt{reset0}\, V]\!] &= (\!|\, \texttt{shift0}\colon (\lambda ck.\, c\, k) \,|\!)\, [\![V]\!] && (1)\\
&= (\!|\, \texttt{shift0}\colon (\lambda ck.\, c\, k) \,|\!)\, (\eta\, v) && (2)\\
&\to_{(\!|\eta|\!)} \eta\, v && (3)\\
&= [\![V]\!] && (4)
\end{aligned}
$$

where $[\![V]\!] = \eta\, v$. As we have said in 1.6.2, an (open) handler without an explicit clause for η is presumed to handle η with η. In this case, the $\lambda_{\texttt{shift0}}$ reduction ends up corresponding to exactly one $(\!|\eta|\!)$ reduction in $(\!|\lambda|\!)$.

Now, let's deal with the last reduction rule, $\to_{\texttt{shift0}}$.

$$
\begin{aligned}
[\![\texttt{reset0}\,(F[\texttt{shift0}\, V])]\!] &= (\!|\, \texttt{shift0}\colon (\lambda ck.\, c\, k) \,|\!)\, [\![F[\texttt{shift0}\, V]]\!] && (1)\\
&\leftrightarrow\!\!\!\!\twoheadrightarrow (\!|\, \texttt{shift0}\colon (\lambda ck.\, c\, k) \,|\!)\, (\texttt{shift0}\, v\, (\lambda x.\, [\![F[x]]\!])) && (2)\\
&\to_{(\!|\texttt{op}|\!)} (\lambda ck.\, c\, k)\, v\, (\lambda x.\, (\!|\, \texttt{shift0}\colon (\lambda ck.\, c\, k) \,|\!)\, [\![F[x]]\!]) && (3)\\
&\to_{\beta,\beta} v\, (\lambda x.\, (\!|\, \texttt{shift0}\colon (\lambda ck.\, c\, k) \,|\!)\, [\![F[x]]\!]) && (4)\\
&= v\, (\lambda x.\, [\![\texttt{reset0}\,(F[x])]\!]) && (5)\\
&\leftarrow_{\beta,\eta.\gg=} (\eta\, (\lambda x.\, [\![\texttt{reset0}\,(F[x])]\!])) \gg= (\lambda n.\, v\, n) && (6)\\
&= [\![\lambda x.\, \texttt{reset0}\,(F[x])]\!] \gg= (\lambda n.\, v\, n) && (7)\\
&\leftarrow_{\beta,\eta.\gg=} (\eta\, v) \gg= (\lambda m.\, [\![\lambda x.\, \texttt{reset0}\,(F[x])]\!] \gg= (\lambda n.\, m\, n)) && (8)\\
&= [\![V]\!] \gg= (\lambda m.\, [\![\lambda x.\, \texttt{reset0}\,(F[x])]\!] \gg= (\lambda n.\, m\, n)) && (9)\\
&= [\![V\, (\lambda x.\, \texttt{reset0}\,(F[x]))]\!] && (10)
\end{aligned}
$$

where $[\![V]\!] = \eta\, v$. Lines 2 and 3 are the crucial lines. On line 2, we use the upcoming Lemma 4.4.4 that will show that through a series of reductions and expansions, we can go from $[\![F[\texttt{shift0}\, V]]\!]$ to $(\texttt{shift0}\, v\, (\lambda x.\, [\![F[x]]\!]))$ where $[\![V]\!] = \eta\, v$. Since we have moved `shift0` to the head of the handler's argument, we can apply the $(\!|\texttt{op}|\!)$ rule on line 3. Since the handler clause is basically the identity function, it disappears on line 4 after two β-reductions.

From then on, we perform a series of expansions while trying to push the interpretation operator $[\![.]\!]$ outwards from $F[x]$ to the entire expression. The expansions used above are a reversal of a common idiom we have used before. We used to go from $(\eta\, M) \gg= (\lambda m.\, N)$ to $N[m := M]$ using $\eta.\gg=$ and then a β-reduction. Here, we go the other way from $N[m := M]$ to $(\eta\, M) \gg= (\lambda m.\, N)$ using a β-expansion and the derived expansion $\eta.\gg=$ (lines 6 and 8). □

Corollary 4.4.3. *Let M and N be terms of $\lambda_{\texttt{shift0}}$. If $M \leftrightarrow\!\!\!\!\twoheadrightarrow N$ in $\lambda_{\texttt{shift0}}$, then $[\![M]\!] \leftrightarrow\!\!\!\!\twoheadrightarrow [\![N]\!]$ in $(\!|\lambda|\!)$.*

[75] We will need the expansions when "unevaluating" some of the monadic binds that have been introduced by our translations.

Contexts = Continuations: Proving the Lemma

All that is left to show is a proof of the lemma we mentioned above.

Lemma 4.4.4.

$$[\![F[\texttt{shift0}\,V]]\!] \leftrightarrow (\texttt{shift0}\,v\,(\lambda x.\,[\![F[x]]\!]))$$

where $[\![V]\!] = \eta\,v$.

This lemma not only allows us to prove the simulation property that is the focus of this section, but it also gives us more insight into $(\!|\,\lambda\,|\!)$. If we read it from right to left and slightly generalizing, it tells us how to think of terms of the form $(\texttt{op}\,x\,k)$. They represent computations where the next point of evaluation is a contextually dependent operation op: x is the operation's argument and k captures the context in which op is being used inside the computation.

Proof. Our proof will proceed by induction on the structure of F. We will start with the base case, $F = []$.

$$\begin{aligned}
[\![F[\texttt{shift0}\,V]]\!] &= [\![\texttt{shift0}\,V]\!] && (1)\\
&= [\![V]\!] \gg\!= (\lambda m.\,\texttt{shift0}!\,m) && (2)\\
&\to_{\eta.\gg\!=,\beta} \texttt{shift0}!\,v && (3)\\
&= (\lambda p.\,\texttt{shift0}\,p\,(\lambda x.\,\eta\,x)\,v && (4)\\
&\to_{\beta} \texttt{shift0}\,v\,(\lambda x.\,\eta\,x) && (5)\\
&= \texttt{shift0}\,v\,(\lambda x.\,[\![x]\!]) && (6)\\
&= \texttt{shift0}\,v\,(\lambda x.\,[\![F[x]]\!]) && (7)
\end{aligned}$$

The individual steps are pretty self-explanatory. On line 4, we expand the definition of the exclamation mark from 1.6.2.

Next case, $F = (F'\,M)$:

$$\begin{aligned}
[\![F[\texttt{shift0}\,V]]\!] &= [\![(F'[\texttt{shift0}\,V])\,M]\!] && (1)\\
&= [\![F'[\texttt{shift0}\,V]]\!] \gg\!= (\lambda m.\,[\![M]\!] \gg\!= (\lambda n.\,m\,n)) && (2)\\
&\leftrightarrow (\texttt{shift0}\,v\,(\lambda x.\,[\![F'[x]]\!])) \gg\!= (\lambda m.\,[\![M]\!] \gg\!= (\lambda n.\,m\,n)) && (3)\\
&\to_{\texttt{op}.\gg\!=} \texttt{shift0}\,v\,(\lambda x.\,[\![F'[x]]\!] \gg\!= (\lambda m.\,[\![M]\!] \gg\!= (\lambda n.\,m\,n))) && (4)\\
&= \texttt{shift0}\,v\,(\lambda x.\,[\![(F'[x])\,M]\!]) && (5)\\
&= \texttt{shift0}\,v\,(\lambda x.\,[\![F[x]]\!]) && (6)\\
& && (7)
\end{aligned}$$

Again, the steps are quite mechanical. Line 3 uses the induction hypothesis and on line 4, we see the derived op.$\gg\!=$ rule introduced in 3.1.2 pushing the $\gg\!=$ inside the continuation.

Case $F = (V'\,F')$:

$$\begin{aligned}
[\![F[\texttt{shift0}\,V]]\!] &= [\![V'\,(F'[\texttt{shift0}\,V])]\!] && (1)\\
&= [\![V']\!] \gg\!= (\lambda m.\,[\![F'[\texttt{shift0}\,V]]\!] \gg\!= (\lambda n.\,m\,n)) && (2)\\
&\to_{\eta.\gg\!=,\beta} [\![F'[\texttt{shift0}\,V]]\!] \gg\!= (\lambda n.\,v'\,n) && (3)\\
&\leftrightarrow (\texttt{shift0}\,v\,(\lambda x.\,[\![F'[x]]\!])) \gg\!= (\lambda n.\,v'\,n) && (4)\\
&\to_{\texttt{op}.\gg\!=} \texttt{shift0}\,v\,(\lambda x.\,[\![F'[x]]\!] \gg\!= (\lambda n.\,v'\,n)) && (5)\\
&\leftarrow_{\beta,\eta.\gg\!=} \texttt{shift0}\,v\,(\lambda x.\,[\![V']\!] \gg\!= (\lambda m.\,[\![F'[x]]\!] \gg\!= (\lambda n.\,m\,n))) && (6)\\
&= \texttt{shift0}\,v\,(\lambda x.\,[\![V'\,(F'[x])]\!]) && (7)\\
&= \texttt{shift0}\,v\,(\lambda x.\,[\![F[x]]\!]) && (8)
\end{aligned}$$

where $[\![V']\!] = \eta\, v'$. This proof is very similar to the one for the case before. We have just two extra steps, on lines 3 and 6, where we first push the $[\![V']\!]$ $(= \eta\, v')$ in through the $\gg\!=$ using $\eta.\gg\!=$ and β and then we pull it out in a different context by reversing the process.

Finally, the last case, where $F = (\texttt{shift0}\, F')$:

$$
\begin{aligned}
[\![F[\texttt{shift0}\, V]]\!] &= [\![\texttt{shift0}\,(F'[\texttt{shift0}\, V])]\!] && (1)\\
&= [\![F'[\texttt{shift0}\, V]]\!] \gg\!= (\lambda m.\, \texttt{shift0!}\, m) && (2)\\
&\leftrightarrow\!\!\!\!\rightarrow (\texttt{shift0}\, v\, (\lambda x.\, [\![F'[x]]\!])) \gg\!= (\lambda m.\, \texttt{shift0!}\, m) && (3)\\
&\rightarrow_{\texttt{op.}\gg\!=} \texttt{shift0}\, v\, (\lambda x.\, [\![F'[x]]\!] \gg\!= (\lambda m.\, \texttt{shift0!}\, m)) && (4)\\
&= \texttt{shift0}\, v\, (\lambda x.\, [\![\texttt{shift0}\,(F'[x])]\!]) && (5)\\
&= \texttt{shift0}\, v\, (\lambda x.\, [\![F[x]]\!]) && (6)
\end{aligned}
$$

And this case is just as simple as the $F = (F'\, M)$ one. This concludes our proof of this lemma. Note that we did not include a case for $F = (\texttt{reset0}\, F')$. Such a context is not a frame, since it embeds $[]$ inside a `reset0`. We can also check that our property would no longer hold in this case, since the `shift0` coming from F' would get handled by the `reset0`. □

By proving this lemma, we have also finished our proof of the fact that whenever we have $M \leftrightarrow\!\!\!\!\rightarrow N$ in $\lambda_{\texttt{shift0}}$, we also have $[\![M]\!] \leftrightarrow\!\!\!\!\rightarrow [\![N]\!]$ in $(\!|\, \lambda \,|\!)$.

4.5 Turning to `shift` and `reset`

There are other control operators, similar to `shift0` and `reset0`. One example would be the more common `shift` and `reset`. The difference between these two pairs can be appreciated by comparing the reduction rules for `shift0` and `shift`.

$$
\begin{array}{rcl}
C[\texttt{reset0}\,(F[\texttt{shift0}\, V])] & \rightarrow_{\texttt{shift0}} & C[V\,(\lambda x.\, \texttt{reset0}\,(F[x]))]\\
C[\texttt{reset}\,(F[\texttt{shift}\, V])] & \rightarrow_{\texttt{shift}} & C[\texttt{reset}\,(V\,(\lambda x.\, \texttt{reset}\,(F[x])))]
\end{array}
$$

`shift` preserves the delimiting `reset` and installs a new one into the continuation. `shift0` is different in that it removes the delimiting `reset0`. In all other ways, the definition of $\lambda_{\texttt{shift}}$ (the call-by-value λ-calculus equipped with `shift` and `reset`) is identical to the one of $\lambda_{\texttt{shift0}}$.

We have seen that the semantics of `shift0` and `reset0` aligns closely with the behavior of operations and handlers in $(\!|\, \lambda \,|\!)$. However, we can translate `shift` and `reset` to $(\!|\, \lambda \,|\!)$ too.

We will do so by first translating $\lambda_{\texttt{shift}}$ to $\lambda_{\texttt{shift0}}$.

Definition 4.5.1. *The **interpretation** $[\![M]\!]_0$ of a $\lambda_{\texttt{shift}}$ term M into $\lambda_{\texttt{shift0}}$ is defined as follows:*

$$
\begin{aligned}
[\![\texttt{reset}\, M]\!]_0 &= \texttt{reset0}\, [\![M]\!]_0\\
[\![\texttt{shift}\, M]\!]_0 &= \texttt{shift0}\,((\lambda m.\, \lambda k.\, \texttt{reset0}\,(m\, k))\, [\![M]\!]_0)\\
[\![M\, N]\!]_0 &= [\![M]\!]_0\, [\![N]\!]_0\\
[\![\lambda x.\, M]\!]_0 &= \lambda x.\, [\![M]\!]_0\\
[\![x]\!]_0 &= x
\end{aligned}
$$

NB: We cannot use $(\texttt{shift0}\,(\lambda k.\, \texttt{reset0}\,([\![M]\!]_0\, k)))$ for the interpretation of $[\![\texttt{shift}\, M]\!]$. That would result in $[\![(\texttt{shift}\, [])]\!]_0$ being equal to $(\texttt{shift0}\,(\lambda k.\, \texttt{reset0}\,([]\, k)))$. The problem here is that $(\texttt{shift}\, [])$ is an evaluation frame in $\lambda_{\texttt{shift}}$, but its interpretation is not even an evaluation context in $\lambda_{\texttt{shift0}}$ since the $[]$ is buried under a λ-abstraction.

To see that this interpretation preserves the same kind of property we have been demonstrating in the rest of this section, we prove the following.

Property 4.5.2. *For any $\lambda_{\texttt{shift}}$ terms M and N, $M \to N$ implies $[\![M]\!]_0 \twoheadrightarrow [\![N]\!]_0$.*

Proof. We first note that if C is an evaluation context in $\lambda_{\texttt{shift}}$, $[\![C]\!]_0$ (where $[\![.]\!]_0$ has been extended to contexts with $[\![[]]\!]_0 = []$) is an evaluation context in $\lambda_{\texttt{shift0}}$. The same also holds for evaluation frames and values. With these observations in our hand, we can proceed onto the proof.

We consider the three possible cases of $M \to N$ that correspond to the three reduction rules in $\lambda_{\texttt{shift}}$. Since the rules $\to_\beta$ and $\to_{\texttt{reset}}/\to_{\texttt{reset0}}$ are identical in both calculi and since $[\![.]\!]_0$ preserves evaluation contexts and values, these cases fall out immediately.

We only have to handle the interesting case of $M \to_{\texttt{shift}} N$. In that case, $M = C[\texttt{reset}\,(F[\texttt{shift}\,V])]$ and $N = C[\texttt{reset}\,(V\,(\lambda x.\,\texttt{reset}\,(F[x])))]$ for some context C, frame F and value V.

$$
\begin{aligned}
[\![M]\!]_0 &= [\![C[\texttt{reset}\,(F[\texttt{shift}\,V])]]\!]_0 && (1)\\
&= C'[\texttt{reset0}\,(F'[\texttt{shift0}\,((\lambda m.\,\lambda k.\,\texttt{reset0}\,(m\,k))\,V')])] && (2)\\
&\to_\beta C'[\texttt{reset0}\,(F'[\texttt{shift0}\,(\lambda k.\,\texttt{reset0}\,(V'\,k))])] && (3)\\
&\to_{\texttt{shift0}} C'[(\lambda k.\,\texttt{reset0}\,(V'\,k))\,(\lambda y.\,\texttt{reset0}\,(F'[y]))] && (4)\\
&\to_\beta C'[\texttt{reset0}\,(V'\,(\lambda y.\,\texttt{reset0}\,(F'[y])))] && (5)\\
&= [\![C[\texttt{reset}\,(V\,(\lambda x.\,\texttt{reset}\,(F[x])))]]\!]_0 && (6)\\
&= [\![N]\!]_0 && (7)
\end{aligned}
$$

where $C' = [\![C]\!]_0$, $F' = [\![F]\!]_0$ and $V' = [\![V]\!]_0$. □

Corollary 4.5.3. *For any $\lambda_{\texttt{shift}}$ terms M and N, $M \twoheadrightarrow N$ implies $[\![M]\!]_0 \twoheadrightarrow [\![N]\!]_0$ and $M \leftrightarrow N$ implies $[\![M]\!]_0 \leftrightarrow [\![N]\!]_0$.*

Corollary 4.5.4. *For any $\lambda_{\texttt{shift}}$ terms M and N, $M \leftrightarrow N$ implies $[\![[\![M]\!]_0]\!] \leftrightarrow [\![[\![N]\!]_0]\!]$ in $(\!|\,\lambda\,|\!)$.*

For the latter corollary, we just compose the translations from $\lambda_{\texttt{shift}}$ to $\lambda_{\texttt{shift0}}$ and from $\lambda_{\texttt{shift0}}$ to $(\!|\,\lambda\,|\!)$ and transitively apply their simulation properties (Corollaries 4.4.3 and 4.5.3).

This lets us extend our interpretation $[\![.]\!]$ to $\lambda_{\texttt{shift}}$.

$$
\begin{aligned}
[\![\texttt{shift}\,M]\!] &= [\![M]\!] \gg\!= (\lambda m.\,\texttt{shift0!}\,(\lambda k.\,(\!|\,\texttt{shift0}\colon (\lambda ck.\,c\,k)\,|\!)\,(m\,k)))\\
[\![\texttt{reset}\,M]\!] &= (\!|\,\texttt{shift0}\colon (\lambda ck.\,c\,k)\,|\!)\,[\![M]\!]
\end{aligned}
$$

In the sequel, we will be translating a type system of $\lambda_{\texttt{shift}}$ to the type system of $(\!|\,\lambda\,|\!)$. For the types to work out in this translation, we will need to throw a few bananas into the mix. In what will follow, we will assume that `shift` and `reset` are translated to $(\!|\,\lambda\,|\!)$ using the interpretations given below:

$$
\begin{aligned}
[\![\texttt{shift}\,M]\!] &= [\![M]\!] \gg\!= (\lambda m.\,\texttt{shift0!}\,(\lambda k.\,(\!|\,\texttt{shift0}\colon (\lambda ck.\,c\,k)\,|\!)\,(m\,((\!|\,|\!)\circ k))))\\
[\![\texttt{reset}\,M]\!] &= (\!|\,|\!)\,((\!|\,\texttt{shift0}\colon (\lambda ck.\,c\,k)\,|\!)\,[\![M]\!])
\end{aligned}
$$

$(\!|\,|\!)$ is actually a valid handler. As any other handler without an explicit clause for η, it handles η with η. It also handles all operations with the operations themselves (i.e. rule $(\!|\,\texttt{op}'\,|\!)$). It is therefore an identity function on computations. However, the most general type that we can infer for $(\!|\,|\!)$ is $\mathcal{F}_E(\alpha) \to \mathcal{F}_{E'}(\alpha)$ for any E and E' such that $E \subseteq E'$. It can therefore be used as a kind of explicit weakening operator on computation types.[76]

[76] This need for explicit weakening of computation types could be eliminated by using actual polymorphic types.

4.6 Considering Types

We have shown embeddings of $\lambda_{\texttt{shift0}}$ and $\lambda_{\texttt{shift}}$ into $(\!|\lambda|\!)$. In both of these embeddings, we have seen that the reduction rules in $(\!|\lambda|\!)$ can emulate those of $\lambda_{\texttt{shift}}$ and $\lambda_{\texttt{shift0}}$. However, we have not defined $(\!|\lambda|\!)$ only via terms and their reductions, we have also specified a type system. Can we somehow guarantee that the results of interpreting $\lambda_{\texttt{shift}}$ into $(\!|\lambda|\!)$ are well-typed?

Clearly not without having some kind of type system for $\lambda_{\texttt{shift}}$. Without a type system, we can write a term like $\lambda x.\, x\, x$ whose translation to $(\!|\lambda|\!)$ is impossible to type. Danvy and Filinski [33] give a type system for a calculus with `shift` and `reset`. In their system, typing judgments have the following form:

$$\rho, \alpha \vdash E : \tau, \beta$$

In this schema, ρ stands for a type environment (context Γ in our notation), E is an expression (a term) and τ is the type of E. The types α and β describe the context in which the expression can occur. If we rewrote E in continuation-passing style, we would get a term of type $(\tau \to \alpha) \to \beta$. The expression E can access a context whose answer type is α and supplant it by an answer of type β.

This kind of type system allows us to write a computation that performs a series of `shift`s, each one changing the answer type for the next. If we wanted to guarantee type safety while allowing this amount of flexibility, we would need to use indexed effects [6] to track the answer type as it changes from `shift0` to `shift0`. We will therefore modify Danvy and Filinski's type system to prohibit continuations from changing the answer type so as to fit into the capabilities of our type system.

Our modified type system will have judgments that follow this schema:

$$\Gamma \mid \gamma \vdash M : \tau$$

We switch to our style of notation, Γ is a typing context and M is a term. We give only a single answer type, γ, written to the left of the turnstile. We also separate it from the type context with a vertical bar instead of a comma so as not to be confusing with the notation for context extension $(\Gamma, x : \alpha)$. In continuation-passing style, the type of the above term M would correspond to $(\tau \to \gamma) \to \gamma$.

There is one more subtlety to cover before we look at the typing rules themselves: what are the types?

Definition 4.6.1. *A $\lambda_{\texttt{shift}}$ **type** is either:*

- *an atomic type ν*
- *a function type $\alpha \xrightarrow{\gamma} \beta$ where α, β and γ are other $\lambda_{\texttt{shift}}$ types*

Since the well-typedness of an expression depends on the context in which it is being evaluated, the function type becomes a bit more complicated. By embedding an expression inside a λ-abstraction, we delay its evaluation. In function application, the context of the application becomes the context the function body. In other words, if when type checking the body of a function we assume that the current answer type is γ, then when we apply this function to an argument, we should better do so in a context in which the answer type actually is γ. This means we have to discriminate between functions w.r.t. the context (i.e. answer type) in which they can be applied.

Definition 4.6.2. *We define the **typing relation for** $\lambda_{\texttt{shift}}$ as the set of all judgments derivable from the inference rules given in Figure 4.2.*

To convince ourselves that this type system works, we will need to prove its soundess by way of demonstrating type preservation and progress.

Lemma 4.6.3. ***Substitution and types in** $\lambda_{\texttt{shift}}$*
Whenever we have $\Gamma, x : \alpha \mid \gamma \vdash M : \tau$ and $\Gamma \mid \gamma \vdash V : \alpha$, we also get $\Gamma \mid \gamma \vdash M[x := V] : \tau$ (i.e. we can substitute in M while preserving the type).

Proof. This technical lemma, common to most λ-calculi, is proven by induction on the derivation of $\Gamma, x : \alpha \mid \gamma \vdash M : \tau$ (which is the same as induction on the syntactic structure of M). The only catch here is that when we descend into M through λ-abstractions and `resets`, we might be forced to change the answer

$$\frac{x : \alpha \in \Gamma}{\Gamma \mid \gamma \vdash x : \alpha}\ [\text{var}]$$

$$\frac{\Gamma, x : \alpha \mid \gamma \vdash M : \beta}{\Gamma \mid \delta \vdash \lambda x.\, M : \alpha \xrightarrow{\gamma} \beta}\ [\text{abs}] \qquad \frac{\Gamma \mid \gamma \vdash M : \alpha \xrightarrow{\gamma} \beta \qquad \Gamma \mid \gamma \vdash N : \alpha}{\Gamma \mid \gamma \vdash M\, N : \beta}\ [\text{app}]$$

$$\frac{\Gamma \mid \gamma \vdash M : \gamma}{\Gamma \mid \delta \vdash \texttt{reset}\, M : \gamma}\ [\texttt{reset}] \qquad \frac{\Gamma \mid \gamma \vdash M : (\alpha \xrightarrow{\delta} \gamma) \xrightarrow{\gamma} \gamma}{\Gamma \mid \gamma \vdash \texttt{shift}\, M : \alpha}\ [\texttt{shift}]$$

Figure 4.2: Typing rules for $\lambda_{\texttt{shift}}$.

type from γ to some δ. Now, in order for the induction to work, we will need to change the answer type in $\Gamma \mid \gamma \vdash V : \alpha$ from γ to δ as well. In other words, we need to coerce $\Gamma \mid \gamma \vdash V : \alpha$ to $\Gamma \mid \delta \vdash V : \alpha$.

This is exactly where the condition that the term V that we are substituting must be a value comes into play. If we look at the typing rules for values (variables and λ-abstractions), we see that the answer type is completely free and therefore if we can prove well-typedness w.r.t. one answer type, we also get it for all answer types. □

Property 4.6.4. ***Subject reduction for*** $\lambda_{\texttt{shift}}$
Let us have $\Gamma \mid \gamma \vdash M : \tau$ *and* $M \to N$*. Then also* $\Gamma \mid \gamma \vdash N : \tau$*.*

Proof. We will prove this property case by case for each reduction rule of $\lambda_{\texttt{shift}}$. In the proof, we assume that the context C wrapping the redex and the contractum is just the empty context $[]$. By the compositionality of the type system, it follows that if $M \to N$ preserves types, then so does $C[M] \to C[N]$.

1. $M \to_\beta N$

   We know that $M = (\lambda x.\, M')\, V$, that $N = M'[x := V]$ and that the derivation of the type of M looks like the following:

$$\frac{\dfrac{\Gamma, x : \alpha \mid \gamma \vdash M' : \tau}{\Gamma \mid \gamma \vdash \lambda x.\, M' : \alpha \xrightarrow{\gamma} \tau}\ [\text{abs}] \qquad \Gamma \mid \gamma \vdash V : \alpha}{\Gamma \mid \gamma \vdash (\lambda x.\, M')\, V : \tau}\ [\text{app}]$$

   By applying Lemma 4.6.3 to the typing derivations of M' and V, we directly get the typing judgment we need.

2. $M \to_{\texttt{reset}} N$

   We have $M = \texttt{reset}\, V$, $N = V$ and the following derivation:

$$\frac{\Gamma \mid \tau \vdash V : \tau}{\Gamma \mid \gamma \vdash \texttt{reset}\, V : \tau}\ [\texttt{reset}]$$

   If we recover the typing derivation for V, we run into the same issue as in the proof of Lemma 4.6.3. The context has changed from answer type γ to answer type τ. Again, we rely on the fact that the argument to `reset` must have been a value in order to be able to take the judgment $\Gamma \mid \tau \vdash M : \tau$ and coerce it to a judgment $\Gamma \mid \gamma \vdash M : \tau$.

3. $M \to_{\texttt{shift}} N$

   We have $M = \texttt{reset}\, (F[\texttt{shift}\, V])$, $N = \texttt{reset}\, (V\, (\lambda x.\, \texttt{reset}\, (F[x])))$.

$$\dfrac{\dfrac{\dfrac{\Gamma \mid \tau \vdash V : (\alpha \xrightarrow{\delta} \tau) \xrightarrow{\tau} \tau}{\Gamma \mid \tau \vdash \texttt{shift}\, V : \alpha}\,[\texttt{shift}]}{\vdots\, F[]\, \vdots}}{\dfrac{\Gamma \mid \tau \vdash F[\texttt{shift}\, V] : \tau}{\Gamma \mid \gamma \vdash \texttt{reset}\,(F[\texttt{shift}\, V]) : \tau}\,[\texttt{reset}]}$$

The validity of the above analysis hinges on the fact that the context separating the `reset` and the `shift` is an evaluation frame. If we look at the typing rules [app] and [`shift`], we see that the answer types of the subterms are always the same as the answer type of the compound term. This is what lets us assume that the answer type of both $F[\texttt{shift}\, V]$ and $\texttt{shift}\, V$ is τ.

From this typing derivation, we will extract the typing judgment of V and the evaluation frame F that can take a term M' such that $\Gamma \mid \tau \vdash M' : \alpha$ to a term $F[M']$ such that $\Gamma \mid \tau \vdash F[M'] : \tau$.

$$\dfrac{\dfrac{\Gamma \mid \tau \vdash V : (\alpha \xrightarrow{\delta} \tau) \xrightarrow{\tau} \tau \qquad \dfrac{\dfrac{\dfrac{\dfrac{\Gamma, x : \alpha \mid \tau \vdash x : \alpha}{\vdots\, F[]\, \vdots}}{\Gamma, x : \alpha \mid \tau \vdash F[x] : \tau}}{\Gamma, x : \alpha \mid \delta \vdash \texttt{reset}\,(F[x]) : \tau}\,[\texttt{reset}]}{\Gamma \mid \tau \vdash \lambda x.\, \texttt{reset}\,(F[x]) : \alpha \xrightarrow{\delta} \tau}\,[\text{abs}]}{\Gamma \mid \tau \vdash V\,(\lambda x.\, \texttt{reset}\,(F[x])) : \tau}\,[\text{app}]}{\Gamma \mid \gamma \vdash \texttt{reset}\,(V\,(\lambda x.\, \texttt{reset}\,(F[x]))) : \tau}\,[\texttt{reset}]$$

The proof tree construction is straightforward, plugging in the two parts, V and $F[]$, we got from the typing of M. The only peculiar point is our use of $F[]$ in the environment $\Gamma, x : \alpha$, which presupposes that x is fresh for $F[]$.[77]

□

Property 4.6.5. ***Progress for*** $\lambda_{\texttt{shift}}$

Whenever we have a closed well-typed term M, i.e. one such that $\emptyset \mid \gamma \vdash M : \tau$, then one of the following must hold:

- *$M = V$ for some value V*
- *$M = F[\texttt{shift}\, V]$ for some frame F and value V*
- *there exists an N such that $M \to N$*

Proof. We will prove this property by showing that if M is not a value, then it must either contain a redex inside an evaluation context (and therefore be reducible) or be of the form $F[\texttt{shift}\, V]$. We will proceed by structural induction and case analysis on the well-typed form of M. We will not consider the case of M being a variable or a λ-abstraction since they are both values (on top of that, a variable is an open term and therefore not typable in the empty environment $\emptyset$).

1. $M = M_1\, M_2$

   The typing derivation for M must look like this:

$$\dfrac{\emptyset \mid \gamma \vdash M_1 : \alpha \xrightarrow{\gamma} \tau \qquad \emptyset \mid \gamma \vdash M_2 : \alpha}{\emptyset \mid \gamma \vdash M_1\, M_2 : \tau}\,[\text{app}]$$

[77] As per the Barendregt variable convention [10], we assume bound variables to be different from free variables.

We first call upon the induction hypothesis for M_1 and consider all three possible outcomes:

- $M_1 = F_1[\texttt{shift}\, V]$ — then we have $M = (F_1[\texttt{shift}\, V]\, M_2) = F[\texttt{shift}\, V]$ where $F = (F_1\, M_2)$
- $M_1 \to N_1$ — then we have $M_1\, M_2 \to N_1\, M_2$ since $([]\, M_2)$ is a valid evaluation context
- $M_1 = V_1$ — then we call upon the induction hypothesis for M_2
  - $M_2 = F_2[\texttt{shift}\, V]$ — then we have $M = (V_1\, (F_2[\texttt{shift}\, V])) = F[\texttt{shift}\, V]$ where $F = (V_1\, F_2)$
  - $M_2 \to N_2$ — then we have $V_1\, M_2 \to V_1\, N_2$ since $(V_1\, [])$ is a valid evaluation context
  - $M_2 = V_2$ — Since $\emptyset \mid \gamma \vdash M_1 : \alpha \xrightarrow{\gamma} \gamma$ and M_1 is a value, then $M_1 = \lambda x.\, M_{11}$ (M_1 cannot be a variable because it must be a closed term). We therefore have $M = (\lambda x.\, M_{11})\, V_2$ which we can reduce to $N = M_{11}[x := V_2]$ using $\to_\beta$.

2. $M = \texttt{reset}\, M'$

   From the type of M, we can get a type for M':

$$\frac{\emptyset \mid \tau \vdash M' : \tau}{\emptyset \mid \gamma \vdash \texttt{reset}\, M' : \tau}\ [\texttt{reset}]$$

   M' is another closed well-typed term and so we apply the induction hypothesis to M' and deal with the possible results:

   - $M' = V$ — we can reduce $M = \texttt{reset}\, V$ to $N = V$ using $\to_{\texttt{reset}}$
   - $M' = F[\texttt{shift}\, V]$ — we can reduce $M = \texttt{reset}\, (F[\texttt{shift}\, V])$ to $N = \texttt{reset}\, (V\, (\lambda x.\, \texttt{reset}\, (F[x])))$ using $\to_{\texttt{shift}}$
   - $M' \to N'$ — then we also have $\texttt{reset}\, M' \to \texttt{reset}\, N'$ since $(\texttt{reset}\, [])$ is a valid evaluation context

3. $M = \texttt{shift}\, M'$

   We follow the same process. Analyze the type of M...

$$\frac{\emptyset \mid \gamma \vdash M' : (\tau \xrightarrow{\delta} \gamma) \xrightarrow{\gamma} \gamma}{\emptyset \mid \gamma \vdash \texttt{shift}\, M' : \tau}\ [\texttt{shift}]$$

   ...apply the induction hypothesis and treat all the cases.

   - $M' = V$ — then we have $M = F[\texttt{shift}\, V]$ where $F = []$
   - $M' = F'[\texttt{shift}\, V]$ — we have $M = \texttt{shift}\, (F'[\texttt{shift}\, V]) = F[\texttt{shift}\, V]$ where $F = (\texttt{shift}\, F')$
   - $M' \to N'$ — we have $\texttt{shift}\, M' \to \texttt{shift}\, N'$ since $(\texttt{shift}\, [])$ is a valid evaluation context

□

Definition 4.6.6. *A $\lambda_{\texttt{shift}}$ term is **stuck** when it is not a value and it cannot reduce to any other $\lambda_{\texttt{shift}}$ term.*

Property 4.6.7. ***Type soundness for $\lambda_{\texttt{shift}}$***
Let M be a closed well-typed $\lambda_{\texttt{shift}}$ term whose expression type and answer type agree, i.e. we have $\emptyset \mid \tau \vdash M : \tau$. Then the term $\texttt{reset}\, M$, which is also well-typed, can never reduce to a stuck term.

Proof. Thanks to subject reduction, we know that all the terms N that we can ever reduce $\texttt{reset}\, M$ to are all well-typed (and closed). Thanks to the progress property, we also know that any such N must satisfy one of the following properties:

- N is a value — therefore, N is not stuck

- $N = F[\texttt{shift}\, V]$ — This case is impossible. It would mean that N is not of the shape $\texttt{reset}\, N'$. Somewhere in the reduction chain from $\texttt{reset}\, M$ to N, the $\texttt{reset}$ would have to be removed. The only reduction rule that can do that is $\rightarrow_{\texttt{reset}}$, repeated here:

$$C[\texttt{reset}\, V] \rightarrow_{\texttt{reset}} C[V]$$

  It only applies when the argument of $\texttt{reset}$ is a value. Furthermore, when applied in the empty context $[]$, its result is also a value. Since values are irreducible (they are either simple variables or λ-abstractions, which are not valid evaluation contexts), then this $\rightarrow_{\texttt{reset}}$ would have been the last reduction in the chain $\texttt{reset}\, M \twoheadrightarrow M' = \texttt{reset}\, V \rightarrow_{\texttt{reset}} V = N$. However, this is in contradiction with $N = F[\texttt{shift}\, V]$.

- $N \rightarrow N'$ — N is not stuck because we can reduce to N'.

□

Having proven the type soundness of the above type system for $\lambda_{\texttt{shift}}$, we now show that typed $\lambda_{\texttt{shift}}$ translates to typed $(\!|\, \lambda \,|\!)$. We have already defined a translation from $\lambda_{\texttt{shift}}$ terms to $(\!|\, \lambda \,|\!)$ in 4.5, now we have to define a translation of the types.

Definition 4.6.8. *We define the* ***interpretation*** $[\![\tau]\!]$ ***of a*** $\lambda_{\texttt{shift}}$ ***type*** τ *by:*

- $[\![\nu]\!] = \nu$ *where* ν *is an atomic type*
- $[\![\alpha \xrightarrow{\gamma} \beta]\!] = [\![\alpha]\!] \rightarrow \mathcal{F}_{E_{[\![\gamma]\!]}}([\![\beta]\!])$ *where* $E_\omega = \{\texttt{shift0} : ((\delta \rightarrow \mathcal{F}_E(\omega)) \rightarrow \mathcal{F}_E(\omega)) \rightarrowtail \delta\}_\delta$[78] $\uplus E$

where E *can be any effect signature as long as* $\texttt{shift0} \notin E$.[79]

Property 4.6.9. ***Simulating*** $\lambda_{\texttt{shift}}$ ***types in*** $(\!|\, \lambda \,|\!)$

$$\Gamma \mid \gamma \vdash M : \tau \quad \Rightarrow \quad [\![\Gamma]\!] \vdash [\![M]\!] : \mathcal{F}_{E_{[\![\gamma]\!]}}([\![\tau]\!])$$

where $[\![\Gamma]\!]$ *is defined by interpreting all of the pairs* $x : \tau$ *as* $x : [\![\tau]\!]$.

Proof. We will proceed by induction on the proof of the judgment $\Gamma \mid \gamma \vdash M : \tau$, covering the 5 cases corresponding to the 5 different inference rules in the $\lambda_{\texttt{shift}}$ type system.

1. $M = x$

$$\frac{x : \tau \in \Gamma}{\Gamma \mid \gamma \vdash x : \tau}\ [\text{var}]$$

   We have $x : \tau \in \Gamma$, which means that in $[\![\Gamma]\!]$, we have $x : [\![\tau]\!]$. The interpretation of M, $[\![M]\!]$, is $\eta\, x$ and we can build a proof of the type judgment like this:

$$\frac{\dfrac{x : [\![\tau]\!] \in [\![\Gamma]\!]}{[\![\Gamma]\!] \vdash x : [\![\tau]\!]}\ [\text{var}]}{[\![\Gamma]\!] \vdash \eta\, x : \mathcal{F}_{E_{[\![\gamma]\!]}}([\![\tau]\!])}\ [\eta]$$

2. $M = \lambda x.\, M'$ and $\tau = \tau_1 \xrightarrow{\delta} \tau_2$

[78] The idea behind E_ω is that $\texttt{shift0}$ should be polymorphic in δ. However, we do not have polymorphism in $(\!|\, \lambda \,|\!)$, so in this particular case, we will assume we have sufficiently many distinct instances of $\texttt{shift0}$ covering the different types at which we want to $\texttt{shift}$.

[79] This means that the calculus can be further extended with other effects whose effect signature would be E.

$$\frac{\Gamma, x : \tau_1 \mid \delta \vdash M' : \tau_2}{\Gamma \mid \gamma \vdash \lambda x.\, M' : \tau_1 \xrightarrow{\delta} \tau_2}\ [\text{abs}]$$

By induction hypothesis, we get that $[\![\Gamma]\!], x : [\![\tau_1]\!] \vdash [\![M']\!] : \mathcal{F}_{E_{[\![\delta]\!]}}[\![\tau_2]\!]$ and by definition, we have $[\![\lambda x.\, M']\!] = \eta\,(\lambda x.\, [\![M']\!])$ and $[\![\tau_1 \xrightarrow{\delta} \tau_2]\!] = [\![\tau_1]\!] \to \mathcal{F}_{E_{[\![\delta]\!]}}([\![\tau_2]\!])$. Let us prove its type.

$$\frac{\dfrac{[\![\Gamma]\!], x : [\![\tau_1]\!] \vdash [\![M']\!] : \mathcal{F}_{E_{[\![\delta]\!]}}([\![\tau_2]\!])}{[\![\Gamma]\!] \vdash \lambda x.\, [\![M']\!] : [\![\tau_1]\!] \to \mathcal{F}_{E_{[\![\delta]\!]}}([\![\tau_2]\!])}\ [\text{abs}]}{[\![\Gamma]\!] \vdash \eta\,(\lambda x.\, [\![M']\!]) : \mathcal{F}_{E_{[\![\gamma]\!]}}([\![\tau_1]\!] \to \mathcal{F}_{E_{[\![\delta]\!]}}([\![\tau_2]\!]))}\ [\text{app}]$$

3. $M = M_1\, M_2$

$$\frac{\Gamma \mid \gamma \vdash M_1 : \tau' \xrightarrow{\gamma} \tau \qquad \Gamma \mid \gamma \vdash M_2 : \tau'}{\Gamma \mid \gamma \vdash M_1\, M_2 : \tau}\ [\text{app}]$$

By induction hypothesis, we get $[\![\Gamma]\!] \vdash [\![M_1]\!] : \mathcal{F}_{E_{[\![\gamma]\!]}}([\![\tau']\!] \to \mathcal{F}_{E_{[\![\gamma]\!]}}([\![\tau]\!]))$ and $[\![\Gamma]\!] \vdash [\![M_2]\!] : \mathcal{F}_{E_{[\![\gamma]\!]}}([\![\tau']\!])$. By definition, we also have $[\![M_1\, M_2]\!] = [\![M_1]\!] \gg\!= (\lambda m.\, [\![M_2]\!] \gg\!= (\lambda n.\, m\, n))$.

We can then construct the type derivation in 4.3a.

4. $M = \texttt{reset}\, M'$

$$\frac{\Gamma \mid \tau \vdash M' : \tau}{\Gamma \mid \gamma \vdash \texttt{reset}\, M' : \tau}\ [\texttt{reset}]$$

By induction hypothesis, we get $[\![\Gamma]\!] \vdash [\![M']\!] : \mathcal{F}_{E_{[\![\tau]\!]}}([\![\tau]\!])$, and by definition we have $[\![\texttt{reset}\, M']\!] = (\!|\ |\!)\ ((\!|\, \texttt{shift0}\colon (\lambda ck.\, c\, k)\, |\!)\, [\![M']\!])$.

$$\begin{array}{c}
E_{[\![\tau]\!]} = \{\texttt{shift0} : (\delta \to \mathcal{F}_E([\![\tau]\!])) \to \mathcal{F}_E([\![\tau]\!]) \rightarrowtail \delta\}_\delta \uplus E \\
[\![\Gamma]\!] \vdash \lambda ck.\, c\, k : ((\delta \to \mathcal{F}_E([\![\tau]\!])) \to \mathcal{F}_E([\![\tau]\!])) \to (\delta \to \mathcal{F}_E([\![\tau]\!])) \to \mathcal{F}_E([\![\tau]\!]) \\
\dfrac{\dfrac{[\![\Gamma]\!] \vdash [\![M']\!] : \mathcal{F}_{E_{[\![\tau]\!]}}([\![\tau]\!])}{[\![\Gamma]\!] \vdash (\!|\, \texttt{shift0}\colon (\lambda ck.\, c\, k)\, |\!)\, [\![M']\!] : \mathcal{F}_E[\![\tau]\!]}\ [(\!|\ |\!)]}{[\![\Gamma]\!] \vdash (\!|\ |\!)\ ((\!|\, \texttt{shift0}\colon (\lambda ck.\, c\, k)\, |\!)\, [\![M']\!]) : \mathcal{F}_{E_{[\![\gamma]\!]}}([\![\tau]\!])}\ [(\!|\ |\!)]
\end{array}$$

5. $M = \texttt{shift}\, M'$

$$\frac{\Gamma \mid \gamma \vdash M' : (\tau \xrightarrow{\delta} \gamma) \xrightarrow{\gamma} \gamma}{\Gamma \mid \gamma \vdash \texttt{shift}\, M' : \tau}\ [\texttt{shift}]$$

The induction hypothesis gives us $[\![\Gamma]\!] \vdash [\![M']\!] : \mathcal{F}_{E_{[\![\gamma]\!]}}(([\![\tau]\!] \to \mathcal{F}_{E_{[\![\delta]\!]}}([\![\gamma]\!])) \to \mathcal{F}_{E_{[\![\gamma]\!]}}([\![\gamma]\!]))$. We also have $[\![\texttt{shift}\, M']\!] = [\![M']\!] \gg\!= (\lambda m.\, \texttt{shift0!}\, (\lambda k.\, (\!|\, \texttt{shift0}\colon (\lambda ck.\, c\, k)\, |\!)\, (m\, ((\!|\ |\!) \circ k))))$. In Figure 4.3b, we construct the appropriate typing derivation for this term.

□

$$
\dfrac{\llbracket\Gamma\rrbracket \vdash \llbracket M_1\rrbracket : \mathcal{F}_{E_{\llbracket\gamma\rrbracket}}(\llbracket\tau'\rrbracket \to \mathcal{F}_{E_{\llbracket\gamma\rrbracket}}(\llbracket\tau\rrbracket)) \qquad \dfrac{\dfrac{\llbracket\Gamma\rrbracket, m : \llbracket\tau'\rrbracket \to \mathcal{F}_{E_{\llbracket\gamma\rrbracket}}(\llbracket\tau\rrbracket) \vdash \llbracket M_2\rrbracket : \mathcal{F}_{E_{\llbracket\gamma\rrbracket}}(\llbracket\tau'\rrbracket) \qquad \dfrac{\dfrac{\dfrac{\llbracket\Gamma\rrbracket, m : \llbracket\tau'\rrbracket \to \mathcal{F}_{E_{\llbracket\gamma\rrbracket}}(\llbracket\tau\rrbracket), n : \llbracket\tau'\rrbracket \vdash m : \llbracket\tau'\rrbracket \to \mathcal{F}_{E_{\llbracket\gamma\rrbracket}}(\llbracket\tau\rrbracket) \qquad \llbracket\Gamma\rrbracket, m : \llbracket\tau'\rrbracket \to \mathcal{F}_{E_{\llbracket\gamma\rrbracket}}(\llbracket\tau\rrbracket), n : \llbracket\tau'\rrbracket \vdash n : \llbracket\tau'\rrbracket}{\llbracket\Gamma\rrbracket, m : \llbracket\tau'\rrbracket \to \mathcal{F}_{E_{\llbracket\gamma\rrbracket}}(\llbracket\tau\rrbracket), n : \llbracket\tau'\rrbracket \vdash m\,n : \mathcal{F}_{E_{\llbracket\gamma\rrbracket}}(\llbracket\tau\rrbracket)}\,[\text{app}]}{\llbracket\Gamma\rrbracket, m : \llbracket\tau'\rrbracket \to \mathcal{F}_{E_{\llbracket\gamma\rrbracket}}(\llbracket\tau\rrbracket) \vdash \lambda n.\, m\,n : \llbracket\tau'\rrbracket \to \mathcal{F}_{E_{\llbracket\gamma\rrbracket}}(\llbracket\tau\rrbracket)}\,[\text{abs}]}{\llbracket\Gamma\rrbracket, m : \llbracket\tau'\rrbracket \to \mathcal{F}_{E_{\llbracket\gamma\rrbracket}}(\llbracket\tau\rrbracket) \vdash \llbracket M_2\rrbracket \gg\!= (\lambda n.\, m\,n) : \mathcal{F}_{E_{\llbracket\gamma\rrbracket}}(\llbracket\tau\rrbracket)}\,[\gg\!=]}{\llbracket\Gamma\rrbracket \vdash \lambda m.\, \llbracket M_2\rrbracket \gg\!= (\lambda n.\, m\,n) : (\llbracket\tau'\rrbracket \to \mathcal{F}_{E_{\llbracket\gamma\rrbracket}}(\llbracket\tau\rrbracket)) \to \mathcal{F}_{E_{\llbracket\gamma\rrbracket}}(\llbracket\tau\rrbracket)}\,[\text{abs}]}{\llbracket\Gamma\rrbracket \vdash \llbracket M_1\rrbracket \gg\!= (\lambda m.\, \llbracket M_2\rrbracket \gg\!= (\lambda n.\, m\,n)) : \mathcal{F}_{E_{\llbracket\gamma\rrbracket}}(\llbracket\tau\rrbracket)}\,[\gg\!=]
$$

(a) The case for $M = M_1\ M_2$. NB: m is assumed to be fresh in M_2 (and therefore $\llbracket M_2\rrbracket$), allowing us to get $\llbracket\Gamma\rrbracket, m : \llbracket\tau'\rrbracket \to \mathcal{F}_{E_{\llbracket\gamma\rrbracket}}(\llbracket\tau\rrbracket) \vdash \llbracket M_2\rrbracket : \mathcal{F}_{E_{\llbracket\gamma\rrbracket}}(\llbracket\tau'\rrbracket)$ from $\llbracket\Gamma\rrbracket \vdash \llbracket M_2\rrbracket : \mathcal{F}_{E_{\llbracket\gamma\rrbracket}}(\llbracket\tau'\rrbracket)$.

$$
\dfrac{\llbracket\Gamma\rrbracket \vdash \llbracket M'\rrbracket : \mathcal{F}_{E_{\llbracket\gamma\rrbracket}}((\llbracket\tau\rrbracket \to \mathcal{F}_{E_{\llbracket\delta\rrbracket}}(\llbracket\gamma\rrbracket)) \to \mathcal{F}_{E_{\llbracket\gamma\rrbracket}}(\llbracket\gamma\rrbracket)) \qquad \dfrac{\dfrac{\dfrac{\dfrac{\llbracket\Gamma\rrbracket \vdash (\!|\,\texttt{shift0}\colon (\lambda ck.\, c\,k)\,|\!) : \mathcal{F}_{E_{\llbracket\gamma\rrbracket}}(\llbracket\gamma\rrbracket) \to \mathcal{F}_E(\llbracket\gamma\rrbracket) \qquad \dfrac{\dfrac{\llbracket\Gamma\rrbracket, m : (\llbracket\tau\rrbracket \to \mathcal{F}_{E_{\llbracket\delta\rrbracket}}(\llbracket\gamma\rrbracket)) \to \mathcal{F}_{E_{\llbracket\gamma\rrbracket}}(\llbracket\gamma\rrbracket) \vdash m : (\llbracket\tau\rrbracket \to \mathcal{F}_{E_{\llbracket\delta\rrbracket}}(\llbracket\gamma\rrbracket)) \to \mathcal{F}_{E_{\llbracket\gamma\rrbracket}}(\llbracket\gamma\rrbracket) \qquad \dfrac{\llbracket\Gamma\rrbracket \vdash (\!|\,|\!) : \mathcal{F}_E(\llbracket\gamma\rrbracket) \to \mathcal{F}_{E_{\llbracket\delta\rrbracket}}(\llbracket\gamma\rrbracket) \qquad \llbracket\Gamma\rrbracket, k : \llbracket\tau\rrbracket \to \mathcal{F}_E(\llbracket\gamma\rrbracket) \vdash k : \llbracket\tau\rrbracket \to \mathcal{F}_E(\llbracket\gamma\rrbracket)}{\llbracket\Gamma\rrbracket, k : \llbracket\tau\rrbracket \to \mathcal{F}_E(\llbracket\gamma\rrbracket) \vdash (\!|\,|\!) \circ k : \llbracket\tau\rrbracket \to \mathcal{F}_{E_{\llbracket\delta\rrbracket}}(\llbracket\gamma\rrbracket)}\,[\circ]}{\llbracket\Gamma\rrbracket, m : (\llbracket\tau\rrbracket \to \mathcal{F}_{E_{\llbracket\delta\rrbracket}}(\llbracket\gamma\rrbracket)) \to \mathcal{F}_{E_{\llbracket\gamma\rrbracket}}(\llbracket\gamma\rrbracket), k : \llbracket\tau\rrbracket \to \mathcal{F}_E(\llbracket\gamma\rrbracket) \vdash m\,(\!|\,|\!) \circ k) : \mathcal{F}_{E_{\llbracket\gamma\rrbracket}}(\llbracket\gamma\rrbracket)}\,[\text{app}]}{\llbracket\Gamma\rrbracket, m : (\llbracket\tau\rrbracket \to \mathcal{F}_{E_{\llbracket\delta\rrbracket}}(\llbracket\gamma\rrbracket)) \to \mathcal{F}_{E_{\llbracket\gamma\rrbracket}}(\llbracket\gamma\rrbracket), k : \llbracket\tau\rrbracket \to \mathcal{F}_E(\llbracket\gamma\rrbracket) \vdash (\!|\,\texttt{shift0}\colon (\lambda ck.\, c\,k)\,|\!)\,(m\,(\!|\,|\!) \circ k)) : \mathcal{F}_E(\llbracket\gamma\rrbracket)}\,[\text{app}]}{\llbracket\Gamma\rrbracket, m : (\llbracket\tau\rrbracket \to \mathcal{F}_{E_{\llbracket\delta\rrbracket}}(\llbracket\gamma\rrbracket)) \to \mathcal{F}_{E_{\llbracket\gamma\rrbracket}}(\llbracket\gamma\rrbracket) \vdash \lambda k.\, (\!|\,\texttt{shift0}\colon (\lambda ck.\, c\,k)\,|\!)\,(m\,(\!|\,|\!) \circ k)) : (\llbracket\tau\rrbracket \to \mathcal{F}_E(\llbracket\gamma\rrbracket)) \to \mathcal{F}_E(\llbracket\gamma\rrbracket)}\,[\text{abs}]}{\llbracket\Gamma\rrbracket, m : (\llbracket\tau\rrbracket \to \mathcal{F}_{E_{\llbracket\delta\rrbracket}}(\llbracket\gamma\rrbracket)) \to \mathcal{F}_{E_{\llbracket\gamma\rrbracket}}(\llbracket\gamma\rrbracket) \vdash \texttt{shift0!}\,(\lambda k.\, (\!|\,\texttt{shift0}\colon (\lambda ck.\, c\,k)\,|\!)\,(m\,(\!|\,|\!) \circ k))) : \mathcal{F}_{E_{\llbracket\gamma\rrbracket}}(\llbracket\tau\rrbracket)}\,[\text{op!}]}{\llbracket\Gamma\rrbracket \vdash \lambda m.\, \texttt{shift0!}\,(\lambda k.\, (\!|\,\texttt{shift0}\colon (\lambda ck.\, c\,k)\,|\!)\,(m\,(\!|\,|\!) \circ k))) : ((\llbracket\tau\rrbracket \to \mathcal{F}_{E_{\llbracket\delta\rrbracket}}(\llbracket\gamma\rrbracket)) \to \mathcal{F}_{E_{\llbracket\gamma\rrbracket}}(\llbracket\gamma\rrbracket)) \to \mathcal{F}_{E_{\llbracket\gamma\rrbracket}}(\llbracket\tau\rrbracket)}\,[\text{abs}]}{\llbracket\Gamma\rrbracket \vdash \llbracket M'\rrbracket \gg\!= (\lambda m.\, \texttt{shift0!}\,(\lambda k.\, (\!|\,\texttt{shift0}\colon (\lambda ck.\, c\,k)\,|\!)\,(m\,(\!|\,|\!) \circ k)))) : \mathcal{F}_{E_{\llbracket\gamma\rrbracket}}(\llbracket\tau\rrbracket)}\,[\gg\!=]
$$

(b) The case for $M = \texttt{shift}\ M'$.

Figure 4.3: Proof trees for the proof of Property 4.6.9.

4.7 Other Control Operators

As we have seen in this chapter, there is more than one set of control operators with which we can endow a λ-calculus. One of the earliest variants is Felleisen's `control/prompt` [43]. Its semantics is similar to the one of `shift/reset`.

$$\begin{array}{rcl} C[\texttt{prompt}\, V] & \to_{\texttt{prompt}} & C[V] \\ C[\texttt{prompt}\, (F[\texttt{control}\, V])] & \to_{\texttt{control}} & C[\texttt{prompt}\, (V\, (\lambda x.\, F[x]))] \end{array}$$

We have also explored `shift0/reset0` and `shift/reset`. The former was interesting because of how closely its semantics matched the ones of $(\!|\, \lambda \,|\!)$.[80] We also focused on the latter since it allowed us to show how a type system for continuations translates to the type system of $(\!|\, \lambda \,|\!)$. We did not study `control/prompt`, or their variants `control0/prompt0`. These operators have already been shown to be mutually expressible with `shift/reset` and `shift0/reset0` [118].

When discussing control operators similar to effect handlers, we should mention the `fcontrol/%` operators of Sitaram [121].

$$\begin{array}{rcl} C[(\%\, V\, M)] & \to_{\%} & C[V] \\ C[(\%\, (F[\texttt{fcontrol}\, V])\, M)] & \to_{\texttt{fcontrol}} & C[M\, V\, (\lambda x.\, F[x])] \end{array}$$

This is very close to effect handlers. The delimiting operator `%` is packaged with a function M. This is like having a handler whose clause for treating `fcontrol` operations is M. Then, whenever `fcontrol` is invoked in its scope, M is applied to the parameter of `fcontrol` and to its continuation. The rule is identical to our $(\!|\, \texttt{op} \,|\!)$ rule, [81] the only difference being the `fcontrol` rule does not reinstate the handler in the body of the continuation. This means that the handler will only treat the first occurrence of the `fcontrol` effect: a notion known as *shallow handlers* [63].

For a comprehensive overview of control operators and their operational semantics, we recommend [1].

[80] `shift0/reset0` are also the control operators that one reaches for when implementing effect handlers using delimited continuations [63]. This means that not only can we use effect handlers to implement delimited continuations, we can also do the converse.

[81] Modulo presentation of contexts and evaluation order.

Part II

Effects and Handlers in Natural Language

Having developed the $(\!|\lambda|\!)$ calculus in Part I, we will now turn our attention to its applications in natural language semantics. We interpret sentences involving deixis, conventional implicature, quantification, anaphora and presupposition as $(\!|\lambda|\!)$ computations in which these phenomena are expressed as side effects/operations. This will allow us to construct a grammar in which all of these phenomena are present. Within this grammar, sentences are interpreted as computations that have access to all of the linguistic side effects at the same time.

5

Introduction to Formal Semantics

Semantics is the study of the meaning of language. In this chapter, we will review the very basics of a school of formal semantics which originated with Richard Montague in the early 70's [96, 97, 98] (5.1). We will then present a formalism that embodies the principles of montagovian semantics, the Abstract Categorial Grammars [35] (5.2). In Chapter 7, we will be analyzing anaphora in $(\!|\lambda|\!)$ by emulating theories of dynamic semantics. To that end, we briefly present dynamic semantics by introducing two of its incarnations: Discourse Representation Theory [64] (5.3) and Type-Theoretic Dynamic Logic [38, 80] (5.4).

Contents

5.1 Montague Semantics **105**
5.1.1 Syntax 106
5.1.2 Semantics 108
5.1.3 Summary 111
5.2 Abstract Categorial Grammars **111**
5.2.1 Definition 111
5.2.2 Syntax 113
5.2.3 Semantics 114
5.2.4 Summary 116
5.3 Discourse Representation Theory **116**
5.3.1 Discourse Representation Structures 118
5.3.2 DRSs as Contents 119
5.3.3 DRSs as Contexts 120
5.3.4 Removing Implication and Disjunction 121
5.4 Type-Theoretic Dynamic Logic **121**
5.4.1 Logic 123
5.4.2 Monadic Structure 123
5.4.3 Example Lexical Entries 124

5.1 Montague Semantics

When studying semantics, one of the verifiable predictions that we can make is whether the contents of one utterance entail the contents of another. This issue already preoccupied Aristotle, who addressed the problem in his study of syllogisms. Using his theory, Aristotle could systematically predict that the contents of Example 2 entail the contents of Example 3, i.e. "if (2), then (3)" is a valid argument.

(2) Every man is mortal. Socrates is a man.

(3) Socrates is mortal.

In the 20th century, mathematical logic studied similar properties on formal artificial languages, leading to the developments of new ideas such as model theory and Tarski's definition of truth [58, 127]. Montague then argues that natural languages deserve the same formal treatment as the artificial languages of logic and mathematics:

> There is in my opinion no important theoretical difference between natural languages and the artificial languages of logicians; indeed, I consider it possible to comprehend the syntax and semantics of both kinds of language within a single natural and mathematically precise theory.
>
> Universal Grammar [97]

In formal logic, the formulas of the artificial languages are defined inductively, by a series of construction rules. The definition of truth is then inductive on the structure of the formula: for every rule that lets us form a logical formula, there is a rule which tells us how to compute its truth value in a model. In his approach, Montague applies the very same strategy to natural language [98].

5.1.1 Syntax

Contrary to propositional logic, whose language is made only of propositions, the set of which is given inductively, natural language expressions fall into many syntactic categories. Montague therefore defines the expressions of a language as a family of sets, indexed by categories. The categories are defined inductively as well: e is a category (of entity-denoting expressions), t is a category (of truth-denoting expressions) and for every two categories A and B, A/B and $A/\!\!/B$ are categories.[82] Montague identifies some of the categories which will become useful in formulating the grammar. In our example, we will make use of the following four:

- $IV = t/e$, the category of intransitive verbs
- $T = t/IV$, the category of terms (i.e. noun phrases)
- $TV = IV/T$, the category of transitive verbs
- $CN = t/\!\!/e$, the category of common nouns

Having established the categories, we will now give some of the construction rules in Montague's grammar. The sets P_A of phrases of category A are defined as the smallest sets being closed on the following construction rules (taken almost verbatim from [98]):

S1 $B_A \subseteq P_A$ for every category A.

Here, B_A is the set of *basic expression* of category A. For the categories that we are interested in, these sets look something like this:

$$\begin{aligned} B_{IV} &= \{\textbf{run}, \textbf{walk}, \textbf{talk}, \ldots\} \\ B_T &= \{\textbf{John}, \textbf{Mary}, \textbf{Bill}, \textbf{he}_0, \textbf{he}_1, \textbf{he}_2, \ldots\}^{83} \\ B_{TV} &= \{\textbf{eat}, \textbf{love}, \textbf{find}, \ldots\} \\ B_{CN} &= \{\textbf{man}, \textbf{woman}, \textbf{unicorn}, \ldots\} \end{aligned}$$

[82] The meanings of these connectives will be given by their use in the grammar, but intuitively, A/B is the category of expressions that when combined with an expression of category B yield an expression of category A, and $A/\!\!/B$ is the category of expressions that denote functions from the meanings of category B to the meanings of category A.

[83] B_T contains a countably infinite set of T-typed variables $\textbf{he}_0$, $\textbf{he}_1$, $\textbf{he}_2$, ...

S2 If $\zeta \in P_{CN}$, then $F_0(\zeta), F_1(\zeta), F_2(\zeta) \in P_T$, where

$F_0(\zeta) =$ **every** ζ
$F_1(\zeta) =$ **the** ζ
$F_2(\zeta)$ is **a** ζ or **an** ζ according as to whether the first word in ζ takes **a** or **an**

S4 [84] If $\alpha \in P_{t/IV}$[85] and $\delta \in P_{IV}$, then $F_4(\alpha, \delta) \in P_t$, where $F_4(\alpha, \delta) = \alpha\delta'$ and δ' is the result of replacing the first verb in δ by its third-person singular present.

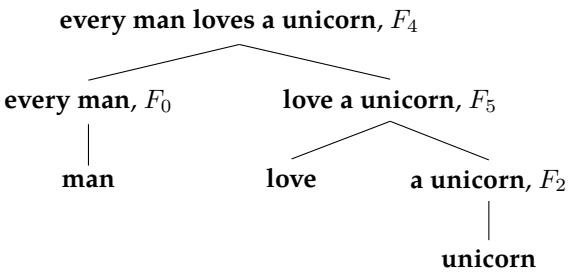
S5 If $\delta \in P_{IV/T}$[86] and $\beta \in P_T$, then $F_5(\delta, \beta) \in P_{IV}$, where $F_5(\delta, \beta) = \delta\beta$ if β does not have the form $\mathbf{he}_n$, and $F_5(\delta, \mathbf{he}_n) = \delta\ \mathbf{him}_n$.

S14 If $\alpha \in P_T$ and $\phi \in P_t$, then $F_{10,n}(\alpha, \phi) \in P_t$, where either:

- α does not have the form $\mathbf{he}_k$, and $F_{10,n}(\alpha, \phi)$ comes from ϕ by replacing the first occurrence of $\mathbf{he}_n$ or $\mathbf{him}_n$ by α and all other occurrences of $\mathbf{he}_n$ or $\mathbf{him}_n$ by **he/she/it** or **him/her/it** respectively, according to whether the first noun in α is masculine/feminine/neuter, or
- $\alpha = \mathbf{he}_k$, and $F_{10,n}(\alpha, \phi)$ comes from ϕ by replacing all occurrences of $\mathbf{he}_n$ or $\mathbf{him}_n$ by $\mathbf{he}_k$ or $\mathbf{him}_k$

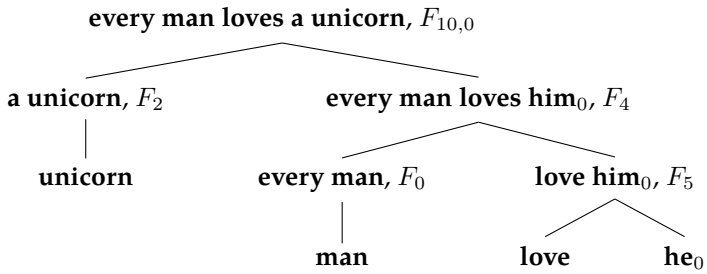
Note that Montague maintains a distinction between deep syntax (tectogrammar) and surface syntax (phenogrammar). For example, the rule S4 tells us that a noun phrase $\alpha \in P_{t/IV}$ can be combined with an intransitive verb $\delta \in P_{IV}$ to yield a sentence $F_4(\alpha, \delta)$ (this is tectogrammar). Then, the definition of F_4 tells us that the noun phrase should precede the verb phrase and that the verb phrase should be in third-person singular present[87] (this is phenogrammar). Similarly for the rule S5, it tells us that a transitive verb $\delta \in P_{IV/T}$ can be combined with a noun phrase $\beta \in P_T$ to form an (intransitive) verb phrase $F_5(\delta, \beta)$. The definition of F_5 then tells us that such a verb phrase is pronounced/written with the transitive verb first and the noun phrase second and that the noun phrase should be in accusative form. In S2, this is the case as well. The rule tells us that for every noun $\zeta \in P_{CN}$, there are the universal, definite and indefinite noun phrases $F_0(\zeta)$, $F_1(\zeta)$ and $F_2(\zeta)$ respectively. The definitions of F_0, F_1 and F_2 then give us the surface realizations of these noun phrases.[88]

Using the grammar given above, we can derive, e.g., the sentence **every man loves a unicorn**. We can record the rules we have used when constructing the sentence in a *derivation tree* (also called an *analysis tree* in [98]).



This is not the only way we can derive this sentence. This is another valid analysis which yields the same string of symbols:

[84] The names of the rules Si and of the functions F_j were kept the same as in [98] for easier reference. Since we are not reproducing the whole grammar, you might notice that some numbers are left out.
[85] i.e. $\alpha \in P_T$
[86] i.e. $\delta \in P_{TV}$
[87] All the noun phrases in this fragment are third-person singular.
[88] If we were to write a grammar of French, we would keep the same tectogrammar and just replace the definitions of F_0, F_1 and F_2 to use **chaque**, **le/la/l'** and **un/une**.



These two derivations will explain the ambiguity of the sentence. As we will see next, the first derivation will produce the subject-wide scope reading: every man loves some unicorn, not necessarily the same one. The second derivation, in which the noun phrase **a unicorn** scopes over the rest of the sentence, will give us the object-wide scope reading: there is a unicorn and every man loves it.

There are infinitely many more derivations of this sentence in this grammar,[89] but their meanings are all equivalent to one of the two analyses we have seen above.

5.1.2 Semantics

Now that we have defined the expressions of our fragment of English, we will study their meaning by establishing their truth conditions: under which conditions (i.e. in which world or model) the sentence is true.

Methodologically, Montague first proposes introducing a formal logic (an artificial language) that is suitable to describe the truth conditions of the natural language under study. Instead of computing directly the truth value of a sentence in a model, Montague translates sentences of natural language into this formal logic. In [98], this formal logic is a higher-order tensed intensional logic with the standard logical connectives, quantifiers over objects of any type, the modal operator of necessity $\Box$ and the past and future modalities H and W. The modalities and the tenses were introduced by Montague to model specific problems in the semantics of a natural language (intensional verbs and tensed verbs). These phenomena will not feature in the small extract from his fragment that we will demonstrate here and therefore we will simplify Montague's approach to use plain higher-order logic (i.e. the λ-calculus).

We establish the meaning of a sentence by translating it to a corresponding formula in higher-order logic (a term of type t) whose meaning will become the meaning of the sentence. However, the expressions of our language are composed not only of sentences (P_t), but also of noun phrases (P_T), common nouns (P_{CN}), transitive verbs (P_{TV}) ... We will interpret these as terms of higher-order logic of different types. The scheme that connects the syntactic categories to the types of interpretations is given below:

$$
\begin{aligned}
&f(t) = t \\
&f(e) = e \\
&f(A/B) = f(A/\!\!/B) = f(B) \rightarrow f(A)
\end{aligned}
$$

Unsurprisingly, the categories t and e of truth-denoting and entity-denoting expressions will be translated to truth values and entities, respectively. The category A/B is the category of expressions that can combine with expressions of category B to form a complex expression of category A. Frege's principle of compositionality states that the meaning of a complex expression ought to be a function of the meanings of its parts. Accordingly, Montague interprets an expression of category A/B as a function from the meanings of the parts (category B) to the meanings of the complex expressions (category A). Finally, the category $A/\!\!/B$ is intended to be the category whose meanings are functions from B to A.

We can apply this definition to the set of categories that feature in our fragment to see what type of interpretation corresponds to each category:

[89] For example, we can repeatedly use rule S14 to replace **him**$_i$ with **him**$_j$ for any i and j.

$$
\begin{aligned}
&f(t) = t \\
&f(T) = f(t/IV) = f(t/(t/e)) = (e \to t) \to t \\
&f(CN) = f(t/\!\!/e) = e \to t \\
&f(IV) = f(t/e) = e \to t \\
&f(TV) = f(IV/T) = f((t/e)/(t/(t/e))) = ((e \to t) \to t) \to e \to t
\end{aligned}
$$

Sentences are expressions of category t and so are interpreted as truth values. Noun phrases (category T) are interpreted as functions of type $(e \to t) \to t$. We call such functions *generalized quantifiers*. They include the quantifiers $\exists$ and $\forall$, as well as functions such as $\lambda P.\, \forall x.\, \mathbf{man}(x) \to P(x)$, which will serve as the meaning of the noun phrase **every man**. The gist of the idea behind generalized quantifiers is that we can represent the meanings of noun phrases such as **some unicorn**, **every pony** or **Bill** as the sets of properties that they satisfy (i.e. which properties hold for some unicorn, which hold for every pony, and which hold for Bill).[90] Next up, common nouns (category CN) are interpreted as sets of entities (e.g. the meaning of **unicorn** is the set of all entities that are unicorns). Likewise, intransitive verbs (category IV) become predicates on entities. Finally, we have transitive verbs (category TV). Syntactically, they combine with an object, which is a noun phrase (category T), to produce a verb phrase (category IV). They will therefore be interpreted as functions from generalized quantifiers (the interpretations of noun phrases) to predicates on entities (the interpretations of verb phrases), type $((e \to t) \to t) \to e \to t$. We can view this type as the type of relations between generalized quantifiers (the objects) and entities (the subjects).

Now that we have decided on the nature of the interpretations that we want to assign to expressions in our fragment, it is time to define the interpretation. For every syntactic construction rule, there will be a corresponding semantic translation rule. If we view the syntactic construction rules as an inductive definition of the expressions in our language, the semantic translation rules are the individual cases of a definition by induction on the structure of these expressions.

T1 We interpret the basic expressions the following way:

- **John**, **Mary** and **Bill** translate to $\lambda P.\, P(\mathbf{j})$, $\lambda P.\, P(\mathbf{m})$ and $\lambda P.\, P(\mathbf{b})$, respectively, where **j**, **m** and **b** are constants of type e[91]
- $\mathbf{he}_n$ translates to $\lambda P.\, P\, x_n$, where x_n is a free variable of type e
- a transitive verb $\delta \in B_{TV}$ translates to the function $\lambda O s.\, O\, (\lambda o.\, \delta'(s, o))$, where δ' is a binary relation on entities (type $e \to e \to t$)[92]
- other basic expressions translate to constants of the corresponding type (e.g. the common noun $\mathbf{man} \in B_{CN}$ translates to a predicate $\mathbf{man}' : e \to t$)

T2 If $\zeta \in P_{CN}$ and ζ translates to ζ', then $F_0(\zeta)$ translates to $G_0(\zeta')$, $F_1(\zeta)$ translates to $G_1(\zeta')$ and $F_2(\zeta)$ translates to $G_2(\zeta')$, where

$$
\begin{aligned}
G_0(\zeta') &= \lambda P.\, \forall x.\, \zeta'(x) \to P(x) \\
G_1(\zeta') &= \lambda P.\, \exists x.\, (\forall y.\, \zeta'(y) \leftrightarrow x = y) \wedge P(y) \\
G_2(\zeta') &= \lambda P.\, \exists x.\, \zeta'(x) \wedge P(x)
\end{aligned}
$$

[90] The type $A \to t$ can both be seen as a property of As and a set of As (given by its characteristic function). The type $(e \to t) \to t$ can thus be seen as a set of properties of entities.

[91] We have to do this because even though we can model proper nouns as constants designating entities, in our fragment, noun phrases are analyzed as generalized quantifiers (in order to account for noun phrases such as **every man**). Therefore, we have to lift the entities **j**, **m** and **b** to generalized quantifiers. The definition $\lambda P.\, P(\mathbf{j})$ states that a property P holds for **John** if and only if it holds for the entity **j**. This construction is known as *type raising* and it is an instance of the generalizing-to-the-worst-case scenario (i.e. even though proper nouns are not quantificational, they have to be raised into generalized quantifiers since other noun phrases might be quantificational).

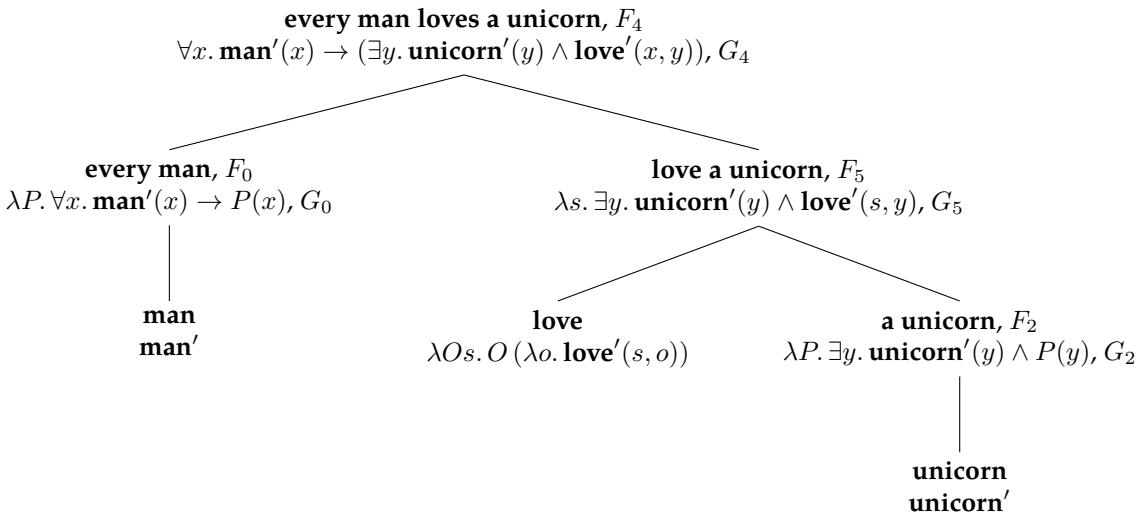
[92] In [98], Montague interprets transitive verbs as constants of type $((e \to t) \to t) \to e \to t$, but then he adds a condition that effectively restricts them to be of the form $\lambda O s.\, O\, (\lambda o.\, \delta'(s, o))$. We also reorder the arguments to the binary relation so that the subject s goes before the object o. This way, the meaning of the sentence **John loves Mary** becomes $\mathbf{love}'(\mathbf{j}, \mathbf{m})$ and not $\mathbf{love}'(\mathbf{m}, \mathbf{j})$.

T4 If $\alpha \in P_{t/IV}$, $\delta \in P_{IV}$, and α, δ translate to α', δ' respectively, then $F_4(\alpha, \delta)$ translates to $G_4(\alpha', \delta')$, where $G_4(\alpha', \delta') = \alpha'(\delta')$.

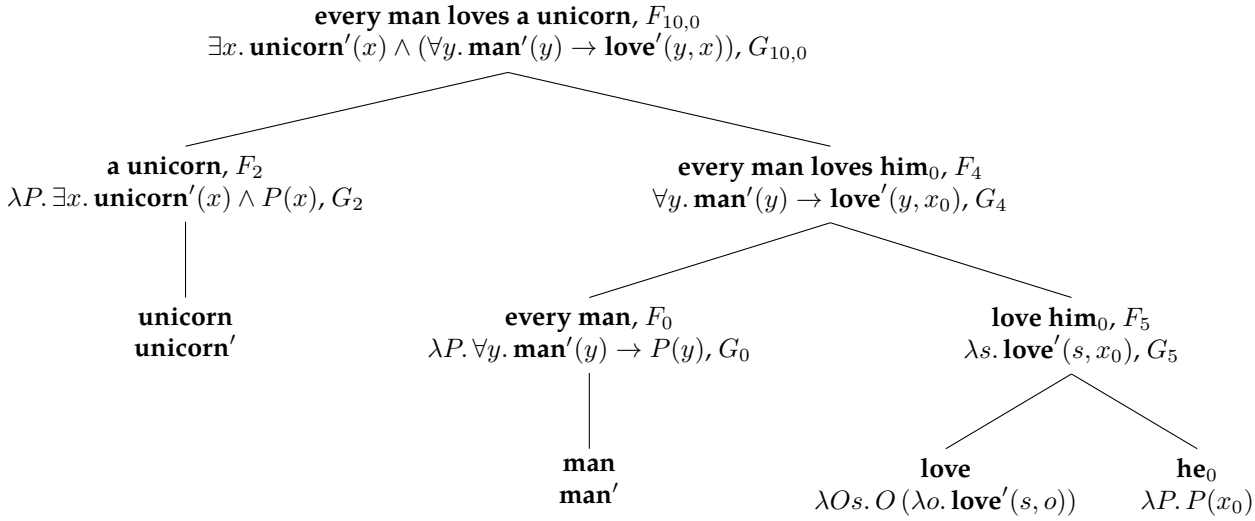
T5 If $\delta \in P_{TV/T}$, $\beta \in P_T$, and δ, β translate to δ', β' respectively, then $F_5(\delta, \beta)$ translates to $G_5(\delta', \beta')$, where $G_5(\delta', \beta') = \delta'(\beta')$.

T14 If $\alpha \in P_T$, $\phi \in P_t$, and α, ϕ translate to α', ϕ' respectively, then $F_{10,n}(\alpha, \phi)$ translates to $G_{10,n}(\alpha', \phi')$, where $G_{10,n}(\alpha', \phi') = \alpha'(\lambda x_n.\, \phi')$.

This completes the definition of the semantics for our fragment. We can now use it to find the meanings of expressions described by our grammar. In the previous subsection, we have seen two derivations of the surface form of the sentence **every man loves a unicorn**. We can now use the semantics to show the translations into higher-order logic side-by-side with the surface form realizations. First, we get the subject-wide scope reading of the sentence:



The second derivation of the sentence gives the object-wide scope reading:



Even though both derivations lead to the same surface forms, the meanings predicted by the two derivations differ, which is how we account for the ambiguity of natural language in a formal setting.

$$\frac{\Gamma \uplus \{x : \alpha\} \vdash_\Sigma M : \beta}{\Gamma \vdash_\Sigma \lambda x.\, M : \alpha \multimap \beta}\ [\text{abs}] \qquad \frac{\Gamma \vdash_\Sigma M : \alpha \multimap \beta \qquad \Delta \vdash_\Sigma N : \alpha}{\Gamma \uplus \Delta \vdash_\Sigma M\, N : \beta}\ [\text{app}]$$

$$x : \alpha \vdash_\Sigma x : \alpha\ [\text{var}] \qquad \vdash_\Sigma c : \tau(c)\ [\text{const}]$$

Figure 5.1: The typing rules for the linear λ-calculus.

5.1.3 Summary

In this section, we have seen the essence of montagovian semantics. We define (usually a small fragment of) natural language using a formal system. For every syntactic construction, there is a description of how to compute the meaning of the construction from the meanings of its constituents. The meanings of sentences are truth values in a mathematical model which represents the state of affairs in the real world. There are other features of Montague grammar that are not essential, but frequently reappear in current formulations of Montague's approach:

- The meaning of a sentence is described in a formal logic. The interpretation of a sentence is then realized by translating sentences of the natural language to formulas of the logic.
- Sentences are built out of constituents which do not all denote truth values (e.g. verb phrases denote predicates). The λ-calculus is used to glue the meanings of all the constituents together. This usually means that the logic in which we interpret natural language is higher-order logic.
- The calculus which governs the possible syntactic derivations is based on categorial grammar. Lexical items can be seen as functions, e.g. a transitive verb is a function from noun phrases (objects) to verb phrases, and their meanings become functions as well.

Montague's work was very influential in the field of natural language semantics. The ideas of Montague stood at the origin of several grammatical frameworks that focus on the syntax-semantics interface [35, 100, 107, 90]. In the next section, we will review one of them.

5.2 Abstract Categorial Grammars

Philippe de Groote's abstract categorial grammars (ACGs) are a grammatical formalism which makes it easy to define Montague-like semantics for fragments of natural languages [35]. It is this formalism that we will be using in the second part of this manuscript and so we will dedicate this section to a brief introduction. We start by presenting the formal definition and then we look at an example of analyzing the syntax and the semantics of a fragment similar to the one we have seen in the Montague example in 5.1.

5.2.1 Definition

The objects at the heart of ACGs are λ-terms. The syntax is the same as in Chapter 1, restricted to abstractions, applications, variables and constants. The important difference comes in the type system. The function type is written as $\alpha \multimap \beta$ and the typing rules differ (see Figure 5.1). This type system enforces the constraint that every λ-binder binds exactly one occurrence of a variable.[93] In the typing rules, we make use of the $\uplus$ operator from Section 1.3 which gives us the union of Γ and Δ, where the domains of Γ and Δ must be disjoint.

As with $(\!| \lambda |\!)$, the linear λ-calculus is parameterized by a set of atomic types and set of typed constants. In ACG terminology, this information is grouped in a *higher-order signature*.

Definition 5.2.1. *A **higher-order signature** is a triple $\langle A, C, \tau \rangle$, where A and C are sets and τ is a function from C to **linear implicative types** $\mathcal{I}(A) ::= A \mid \mathcal{I}(A) \multimap \mathcal{I}(A)$. We call the elements of A the **atomic types** and the elements of C the **constants**.*

[93] This is due to the fact $\Gamma \vdash_\Sigma M : \alpha$ entails that all the variables from the domain of Γ occur free exactly once in M.

With ACGs, we will represent the derivations (i.e. the tectogrammatical structures) of sentences as linear λ-terms. In Section 5.1, we have seen how in Montague semantics, the derivation of a phrase can be seen as a computation that produces its surface form (the rules in Subsection 5.1.1) or as a computation that produces its meaning (the rules in Subsection 5.1.2). We will now define the ACG notion of interpreting derivations.

Definition 5.2.2. *A **lexicon** from signature $\Sigma_1 = \langle A_1, C_1, \tau_1 \rangle$ to signature $\Sigma_2 = \langle A_2, C_2, \tau_2 \rangle$ is a pair $\langle F, G \rangle$ where:*

- *$F : A_1 \to \mathcal{I}(A_2)$ is a function that interprets the atomic types of Σ_1 as types in Σ_2,*
- *$G : C_1 \to \Lambda(\Sigma_2)$ is a function that interprets the constants of Σ_1 as linear λ-terms in Σ_2,*
- *and the interpretations are well-typed, meaning that for every $c \in C_1$, we have $\vdash G(c) : F(\tau(c))$*[94]

With these definitions in place, we can start tracing the parallels between Montague's grammar and ACGs. The level of deep syntactic structure is represented by a higher-order signature Σ_1. Its constants are the basic expressions and the syntactic construction rules from Subsection 5.1.1 and its types are the categories. The types of the construction rules reflect the types of the constituents and of the resulting complex expression. For example, the construction rule F_0, which maps common nouns ζ to the noun phrases **every** ζ, has the type $CN \multimap T$. The target of our semantic translation, higher-order logic, is also represented by a higher-order signature, Σ_2. Its constants are the logical connectives, quantifiers and the relations that we assume to have in the model (**man**$'$, **unicorn**$'$, **love**$'$...). Its types are e, the type of entities, and t, the type of truth values. The translation that we described in Subsection 5.1.2 would correspond to a lexicon from Σ_1 to Σ_2. The interpretation of types would be the homomorphic function f and the interpretation of the constants would be the functions G_i which mirror the syntactic constructors F_i.

We finish with the formal definition of an abstract categorial grammar.

Definition 5.2.3. *An **abstract categorial grammar** is a quadruple $\langle \Sigma_1, \Sigma_2, \mathcal{L}, s \rangle$, where:*

- *$\Sigma_1 = \langle A_1, C_1, \tau_1 \rangle$ is a higher-order signature that we will call the **abstract signature** (sometimes also called the **abstract vocabulary**)*
- *$\Sigma_2 = \langle A_2, C_2, \tau_2 \rangle$ is a higher-order signature that we will call the **object signature** (also called the **object vocabulary**)*
- *$\mathcal{L} : \Sigma_1 \to \Sigma_2$ is a lexicon from Σ_1 to Σ_2*
- *$s \in \mathcal{I}(A_1)$ is a type from the abstract signature that we will call the **distinguished type***

Definition 5.2.4. *The **abstract language** $\mathcal{A}(\mathcal{G})$ of an ACG $\mathcal{G} = \langle \Sigma_1, \Sigma_2, \mathcal{L}, s \rangle$ is the set $\{M \mid \vdash_{\Sigma_1} M : s\}$.*

Definition 5.2.5. *The **object language** $\mathcal{O}(\mathcal{G})$ of an ACG $\mathcal{G} = \langle \Sigma_1, \Sigma_2, \mathcal{L}, s \rangle$ is the set $\{\mathcal{L}(M) \mid M \in \mathcal{A}(\mathcal{G})\}$.*[95]

In our analogy to Montague, by taking the distinguished type to be the category t of truth-denoting expressions, the abstract language is the language of derivations of sentences. The object language is then the language of meanings expressible by the sentences of our language. If we would instead take the ACG where the object signature is the signature of strings and the lexicon is the lexicon mapping syntactic constructions to their surface realizations (i.e. the F_i functions from Subsection 5.1.1), then we would get English strings as the object language.

[94] Here we are applying $F : A_1 \to \mathcal{I}(A_2)$ to a type $\tau(c) \in \mathcal{I}(A_1)$. Whenever $F : A_1 \to \mathcal{I}(A_2)$ is an interpretation of atomic types, we will also use F for its (unique) homomorphic extension, which maps types in $\mathcal{I}(A_1)$ to types in $\mathcal{I}(A_2)$ by replacing all the atomic types with their interpretations. We will also be doing the same when extending an interpretation $G : C_1 \to \Lambda(\Sigma_2)$ to terms in $\Lambda(\Sigma_1)$. The homomorphic extension of G will replace all the constants in the term with their interpretations.

[95] When $\mathcal{L} = \langle F, G \rangle$ is a lexicon, M is a λ-term and τ is a type, we will write $\mathcal{L}(M)$ for $G(M)$ and $\mathcal{L}(\tau)$ for $F(\tau)$.

5.2.2 Syntax

We will now go through a simple example of ACGs, taken from [105], that parallels the extract from Montague that we have seen in Section 5.1. We start by defining a higher-order signature that will describe the derivations of the expressions of our fragment of English:

$$\begin{aligned}
\textsc{John}, \textsc{Mary}, \textsc{Bill} &: NP \\
\textsc{eat}, \textsc{love}, \textsc{find} &: NP \multimap NP \multimap S \\
\textsc{every}, \textsc{the}, \textsc{a} &: N \multimap (NP \multimap S) \multimap S \\
\textsc{man}, \textsc{woman}, \textsc{unicorn} &: N
\end{aligned}$$

In this manuscript, we will establish higher-order signatures by giving type assignments such as those above. The constants can be read off from the domain of the assignments and the atomic types can be read off from the atomic types that occur in the assigned types. In this particular signature, the constants are JOHN, MARY, BILL, EAT, LOVE, FIND, EVERY, THE, A, MAN, WOMAN and UNICORN. The atomic types are S (sentences, t in Montague's grammar), NP (noun phrases, e in Montague's grammar (Montague's T corresponds to $(NP \multimap S) \multimap S$)) and N (common nouns, $CN = t/\!\!/e$ in Montague's grammar). The determiners EVERY, THE and A have the type $N \multimap (NP \multimap S) \multimap S$. In this grammar, all quantificational noun phrases are raised. The type $NP \multimap S$ can be seen as the type of sentences S with a free variable of type NP. The phrase EVERY UNICORN : $(NP \multimap S) \multimap S$ takes such a sentence as an argument and binds the free NP variable within.[96]

This signature describes the deep syntax (tectogrammar) of a fragment of English. For such signatures, we will follow the convention of using SMALL CAPS when typesetting the constants. This signature will be the abstract signature of an ACG. We will now define another signature to serve as the object signature and give a lexicon that interprets one in terms of the other. The signature of strings is defined below:

$$\begin{aligned}
&\text{John}, \text{Mary}, \text{Bill}, \text{eat}, \text{love}, \text{find} : \mathit{string} \\
&\text{every}, \text{the}, \text{a}, \text{man}, \text{woman}, \text{unicorn} : \mathit{string} \\
&(_ + _) : \mathit{string} \multimap \mathit{string} \multimap \mathit{string}
\end{aligned}$$

This signature has a single atomic type, string. For every word that features in our grammar, there is a string-typed constant in the signature. There is also a binary concatenation operator on strings, which we will write in infix form.[97] We can now give a lexicon $[\![\_]\!]_{\text{syntax}}$ that will map deep syntax into strings of words, i.e. surface syntax. On the type level, sentences, noun phrases and common nouns will all be interpreted as strings:

$$\begin{aligned}
[\![S]\!]_{\text{syntax}} &= \mathit{string} \\
[\![NP]\!]_{\text{syntax}} &= \mathit{string} \\
[\![N]\!]_{\text{syntax}} &= \mathit{string}
\end{aligned}$$

We can now give an interpretation to all of the constants of the syntactic signature:

[96] Much the same way that the rules $F_{10,n}$ from S14 and $G_{10,n}$ from T14 do in Montague's grammar.

[97] String concatenation is associative. However, there is nothing guaranteeing that this is the case for our (_ + _) operator. We could make it be the case by introducing another lexicon which would encode the string literals and string concatenation as Church strings and function composition.

$$
\begin{aligned}
[\![\textsc{John}]\!]_{\text{syntax}} &= \text{John} \\
[\![\textsc{Mary}]\!]_{\text{syntax}} &= \text{Mary} \\
[\![\textsc{Bill}]\!]_{\text{syntax}} &= \text{Bill} \\
[\![\textsc{eat}]\!]_{\text{syntax}} &= \lambda os.\, s + \text{eats} + o \\
[\![\textsc{love}]\!]_{\text{syntax}} &= \lambda os.\, s + \text{loves} + o \\
[\![\textsc{find}]\!]_{\text{syntax}} &= \lambda os.\, s + \text{finds} + o \\
[\![\textsc{every}]\!]_{\text{syntax}} &= \lambda nP.\, P\,(\text{every} + n) \\
[\![\textsc{the}]\!]_{\text{syntax}} &= \lambda nP.\, P\,(\text{the} + n) \\
[\![\textsc{a}]\!]_{\text{syntax}} &= \lambda nP.\, P\,(\text{a} + n) \\
[\![\textsc{man}]\!]_{\text{syntax}} &= \text{man} \\
[\![\textsc{woman}]\!]_{\text{syntax}} &= \text{woman} \\
[\![\textsc{unicorn}]\!]_{\text{syntax}} &= \text{unicorn}
\end{aligned}
$$

These two signatures, the lexicon and the distinguished type S together form an ACG. The abstract language of this ACG generates syntactic derivations, such as the terms t_1 and t_2 given below.

$$
\begin{aligned}
t_1 &= \textsc{every man}\,(\lambda x.\, \textsc{a unicorn}\,(\lambda y.\, \textsc{love}\, y\, x)) \\
t_2 &= \textsc{a unicorn}\,(\lambda y.\, \textsc{every man}\,(\lambda x.\, \textsc{love}\, y\, x))
\end{aligned}
$$

The object language contains strings which are the surface realizations of the derivations generated by the abstract language. Therefore, it contains the following English sentence, which can be produced by interpreting either t_1 or t_2:

$$
\begin{aligned}
[\![t_1]\!]_{\text{syntax}} &= \text{every} + \text{man} + \text{loves} + \text{a} + \text{unicorn} \\
[\![t_2]\!]_{\text{syntax}} &= \text{every} + \text{man} + \text{loves} + \text{a} + \text{unicorn}
\end{aligned}
$$

As in the case of Montague's grammar (Subsection 5.1.1), we have two derivations for this sentence. One of the two, t_1, will lead to the subject-wide scope reading of the sentence whereas the other, t_2, will lead to the object-wide scope reading. As in Montague's grammar, ambiguity is explained by the fact that there exist multiple derivations which all yield the same surface realization. Furthermore, Montague noted that in his grammar, there is an infinity of derivations for this sentence, though their differences are unimportant (they are all essentially equivalent to one of the two analyses that we have here). In ACGs, this notion is made precise. The sentence above also has an infinity of possible derivations (e.g. $\textsc{every man}\,(\lambda x.\, \textsc{a unicorn}\,(\textsc{love}\, x))$). However, every such derivation is $\beta\eta$-equivalent to either t_1 or t_2, whereas t_1 and t_2 are not $\beta\eta$-equivalent (they have different $\beta\eta$-normal forms).

5.2.3 Semantics

We will now give a semantics to our fragment. We will introduce a new object signature (that of higher-order logic) and define a lexicon that will interpret the lexical items of our deep syntax abstract signature in higher-order logic. First, we define the signature of logic:

$$
\begin{aligned}
\top, \bot &: o \\
\neg &: o \to o \\
(\_ \wedge \_), (\_ \vee \_), (\_ \to \_), (\_ \leftrightarrow \_) &: o \to o \to o \\
\exists, \forall &: (\iota \to o) \to o \\
\_ = \_ &: \iota \to \iota \to o \\
\mathbf{j}, \mathbf{m}, \mathbf{b} &: \iota \\
\mathbf{man}, \mathbf{woman}, \mathbf{unicorn} &: \iota \to o \\
\mathbf{eat}, \mathbf{love}, \mathbf{find} &: \iota \to \iota \to o
\end{aligned}
$$

The atomic types are o, the type of propositions, and ι, the type of individuals.[98] The signature includes constants for a tautology ($\top$) and a contradiction ($\bot$), the unary logical operator of negation ($\neg$), the binary connectives of conjunction ($\_ \wedge \_$), disjunction ($\_ \vee \_$), implication ($\_ \to \_$) and equivalence ($\_ \leftrightarrow \_$), the existential and universal quantifier over individuals ($\exists$ and $\forall$), and an equality relation on individuals ($\_ = \_$). Furthermore, the signature contains constants from the model: the individuals **j**, **m** and **b**, the unary relations **man**, **woman** and **unicorn**, and the binary relation **eat**, **love** and **find**. We adopt the convention of using **boldface fonts** to typeset the constants of our model. We also use single-letter names for individual constants.

We note that the types make use of the usual function types $\alpha \to \beta$ instead of the linear function type $\alpha \multimap \beta$. In (higher-order) logic, it is often the case that a bound variable occurs more than once in its scope, as we will shortly see in the interpretations. This is a liberty that we will take when dealing with semantic lexicons. The calculus into which we will be translating will not always be the linear λ-calculus. It will suffice for it to be any calculus which includes the simply-typed (linear) λ-calculus. The simply-typed λ-calculus, which we will make use of in this interpretation, is one such calculus. $(\!| \lambda |\!)$, which we will use in the upcoming chapters, is another.

We will now give the lexicon $[\![\_]\!]$ that interprets the syntactic constructions of our fragment in (higher-order) logic. First, we identify the type of interpretations for the atomic abstract types:

$$
\begin{aligned}
[\![S]\!] &= o \\
[\![NP]\!] &= \iota \\
[\![N]\!] &= \iota \to o
\end{aligned}
$$

$$
[\![A \multimap B]\!] = [\![A]\!] \to [\![B]\!]
$$

This follows what we have seen in Montague's semantic interpretation (definition of f in Subsection 5.1.2). The type S (t in Montague) is interpreted as o (t in Montague), the type NP (e in Montague) is interpreted as ι (e in Montague) and the type N ($CN = t/\!\!/e$ in Montague) is interpreted as $\iota \to o$ ($e \to t$ in Montague). In ACGs, it is always the case that $[\![A \multimap B]\!] = [\![A]\!] \multimap [\![B]\!]$.[99] However, since our target calculus is no longer the linear λ-calculus, the linear function connective ($\_ \multimap \_$) is replaced by the usual connective ($\_ \to \_$). This is very similar to Montague's grammar, where the connective ($\_/\_$) on syntactic categories is translated to the function space connective ($\_ \to \_$).

With the interpretations of types in place, we can give the interpretations of all the syntactic constructors:

[98] This is the notation of Church's simple type theory. The types o and ι correspond to Montague's types t and e respectively. In the rest of the manuscript, we will stick to Church's notation.

[99] It is because the action of a lexicon $[\![\_]\!] = \langle F, G \rangle$ on types is defined as the unique homomorphic extension of F.

$$
\begin{aligned}
[\![\textsc{John}]\!] &= \mathbf{j} \\
[\![\textsc{Mary}]\!] &= \mathbf{m} \\
[\![\textsc{Bill}]\!] &= \mathbf{b} \\
[\![\textsc{eat}]\!] &= \lambda os.\, \mathbf{eat}\, s\, o \\
[\![\textsc{love}]\!] &= \lambda os.\, \mathbf{love}\, s\, o \\
[\![\textsc{find}]\!] &= \lambda os.\, \mathbf{find}\, s\, o \\
[\![\textsc{every}]\!] &= \lambda NP.\, \forall x.\, N\, x \rightarrow P\, x \\
[\![\textsc{the}]\!] &= \lambda NP.\, \exists x.\, (\forall y.\, N\, y \leftrightarrow x = y) \wedge P\, x \\
[\![\textsc{a}]\!] &= \lambda NP.\, \exists x.\, N\, x \wedge P\, x \\
[\![\textsc{man}]\!] &= \mathbf{man} \\
[\![\textsc{woman}]\!] &= \mathbf{woman} \\
[\![\textsc{unicorn}]\!] &= \mathbf{unicorn}
\end{aligned}
$$

The interpretations above follow the ones we have seen in Montague's grammar (Subsection 5.1.2). In our manuscript, we will use the term *lexical entry* to refer to the individual interpretations of the constructors in our grammar.

Using the $[\![\_]\!]$ lexicon, we can interpret the terms t_1 and t_2 from the previous subsection to get the two reading of the sentence "*every man loves a unicorn*".

$$
\begin{aligned}
[\![t_1]\!] &= \forall x.\, \mathbf{man}\, x \rightarrow (\exists y.\, \mathbf{unicorn}\, y \wedge \mathbf{love}\, x\, y) \\
[\![t_2]\!] &= \exists y.\, \mathbf{unicorn}\, y \wedge (\forall x.\, \mathbf{man}\, x \rightarrow \mathbf{love}\, x\, y)
\end{aligned}
$$

5.2.4 Summary

We have seen how ACGs can be used to implement Montague's program of defining fragments of natural languages and giving them a compositional model-theoretic semantics. ACGs make explicit the notion of derivation trees, which are generated by a simple type system. These derivation trees (abstract terms) are then interpreted piece-by-piece (a homomorphism that interprets constants), which guarantees compositionality of the interpretation.

In our manuscript, we will be using ACGs as our grammatical formalism. We will be ignoring the lexicons that defines the surface realization of sentences: their definitions would be either trivial or fraught with detail (such as morphological agreement) that we are not interested in our thesis. Throughout the manuscript, we will therefore make use of ACGs in two modes: we will be introducing lexical items into the abstract signature of deep syntax with declarations of the form $\textsc{item} : TYPE$ and we will be giving their interpretations as lexical entries of the form $[\![\textsc{item}]\!] =$ **lambda-banana-term**. The calculus in which we will be giving the meanings of lexical items will be $(\!|\, \lambda \,|\!)$.

You might have noticed that we have already used this style in the first part of the manuscript. In Chapter 2, when we were building up our simple calculator language and giving its semantics, we were using ACGs.[100]

5.3 Discourse Representation Theory

Montague's approach of evaluating natural language in a model has been extended to linguistic units larger than a single sentence. The crucial observation when dealing with linguistic *discourse* is that the meanings (i.e. truth values) of the sentences that make it up cannot be computed independently, without considering the context in which they appear. If we look at the sentence in Example 4, we cannot describe its truth conditions as we do not know to what the pronouns *it* and *him* refer.

[100] The only difference being that we used $(\_ \rightarrow \_)$ in the abstract types instead of $(\_ \multimap \_)$.

(4) It fascinates him.
fascinate ? ?

However, if we place this sentence in a context which provides suitable antecedents for the pronouns, as in Example 5 from [64], we can find the proposition that represents the sentence's truth conditions.

(5) Jones$_1$ owns Ulysses$_2$. It$_2$ fascinates him$_1$.
own j u $\wedge$ **fascinate u j**

Furthermore, the contribution of a sentence can be more complicated than conjoining a new proposition to the logical representation of the preceding discourse. Let us consider an example from [64] (Example (1.28), Section 1.1.3):

(6) Jones$_1$ owns a Porsche$_2$. It$_2$ fascinates him$_1$.
$(\exists x.\, \mathbf{Porsche}\, x \wedge \mathbf{own}\, \mathbf{j}\, x) \wedge \mathbf{fascinate}\, x^{?}\, \mathbf{j}$
$(\exists x.\, \mathbf{Porsche}\, x \wedge \mathbf{own}\, \mathbf{j}\, x \wedge \mathbf{fascinate}\, x\, \mathbf{j})$

In Example 6, the first sentence contains an indefinite and the truth conditions of the sentence are thus expressed by an existentially quantified formula. In the second sentence, the pronoun refers to the indefinite from the first sentence. We would therefore like the referent of the pronoun to co-vary with the referent of the indefinite. However, the variable x which designates the referent of the indefinite is not in scope of the second sentence. Instead of just conjoining the two propositions, we would like to insert the proposition under the scope of the existential quantifier contributed by the indefinite of the first sentence.[101]

These issues with anaphora are not limited to examples that span multiple sentences. We can see the same phenomenon in the single-sentence Example 7 and its paraphrase in Example 8.

(7) Every farmer who owns a donkey$_1$ beats it$_1$.
$\forall x.\, (\mathbf{farmer}\, x \wedge (\exists y.\, \mathbf{donkey}\, y \wedge \mathbf{own}\, x\, y)) \rightarrow \mathbf{beat}\, x\, y^{?}$
$\forall xy.\, (\mathbf{farmer}\, x \wedge \mathbf{donkey}\, y \wedge \mathbf{own}\, x\, y) \rightarrow \mathbf{beat}\, x\, y$

(8) If a farmer$_1$ owns a donkey$_2$, he$_1$ beats it$_2$.
$(\exists xy.\, \mathbf{farmer}\, x \wedge \mathbf{donkey}\, y \wedge \mathbf{own}\, x\, y) \rightarrow \mathbf{beat}\, x^{?}\, y^{?}$
$\forall xy.\, (\mathbf{farmer}\, x \wedge \mathbf{donkey}\, y \wedge \mathbf{own}\, x\, y) \rightarrow \mathbf{beat}\, x\, y$

If we try to compute the meaning of the noun *farmer who owns a donkey* in Montague semantics,[102] we get the predicate $\lambda x.\, \mathbf{farmer}\, x \wedge (\exists y.\, \mathbf{donkey}\, y \wedge \mathbf{own}\, x\, y)$. However, using this interpretation for the relative clause, we get the first logical form under Example 7, in which the pronoun is not in scope of the variable y that refers to the donkey. Again, we can get the correct reading by lifting and extending the scope of the quantifier contributed by the indefinite.

A similar thing happens in Example 8, which paraphrases Example 7. The (montagovian) meaning of the antecedent *a farmer owns a donkey* is the proposition $\exists xy.\, \mathbf{farmer}\, x \wedge \mathbf{donkey}\, y \wedge \mathbf{own}\, x\, y$. However, if we use this meaning in the conditional of Example 8, the consequent *he beats it* will be outside the scope of the variables x and y that refer to the farmer and the donkey respectively. This leads to the first logical form under Example 8, which is unsatisfactory since the x and y variables are not correctly bound. Yet again, the desired logical representation can be recovered by lifting and extending the scope of the quantifiers contributed by the indefinites.

When Montague wanted to treat intensional and tensed verbs in his fragment, he decided to translate natural language to a formal logic which was intensional and tensed. A similar strategy was employed by semanticists who studied dynamic phenomena (such as anaphora) and developed their dynamic logics [53, 64, 38, 80]. In dynamic logic, $(\exists x.\, A) \wedge B$ becomes equivalent to $(\exists x.\, A \wedge B)$ and $(\exists x.\, A) \rightarrow B$ becomes equivalent to $(\forall x.\, A \rightarrow B)$. With these equivalences, we can lift and extend the scope of indefinites so as to obtain the binding that we observe in natural language.

[101] We can also think of this as extending the existential quantification from the first sentence to cover the second sentence.
[102] In [98], Montague's grammar includes the similar construction *farmer such that he owns a donkey* (rules $F_{3,n}$ in S3).

Our analysis of dynamics using $(\!| \lambda |\!)$ draws on two theories of dynamic semantics: Discourse Representation Theory (DRT) [64] and Type-Theoretic Dynamic Logic (TTDL) [38, 80]. In this section, we briefly introduce DRT and in the next, we talk about TTDL.

5.3.1 Discourse Representation Structures

In DRT, sentences contribute to the construction of a *Discourse Representation Structure* (DRS). This structures serve at the same time as the representation of the *content* of the discourse and also as the *context* in which any subsequent discourse is evaluated.

We begin the formal definition by assuming an infinite set $\mathcal{D}$ of *discourse referents*. These are just like the variables of first-order logic; they are symbolic references to individuals. Our formalization will also rely on a vocabulary of relation symbols.

Definition 5.3.1. *A **Discourse Representation Structure (DRS)** K is a pair $\langle U, C\rangle$, where U, the **universe**, is a set of discourse referents and C is a set of DRS conditions.*

***DRS conditions** consist of:*

- *atomic conditions $\mathbf{P}\, x_1 \ldots x_n$, where $\mathbf{P}$ is an n-ary relation symbol and $x_1, \ldots, x_n$ are discourse referents*
- *$\neg K$, where K is a DRS*
- *$K_1 \vee K_2$, where K_1 and K_2 are DRSs*
- *$K_1 \Rightarrow K_2$, where K_1 and K_2 are DRSs*

Notation 5.3.2. *A DRS $\langle\{x_1, \ldots, x_n\}, \{c_1, \ldots, c_m\}\rangle$ will be usually presented using the following notation:*

$$\begin{array}{|c|}\hline x_1 \ldots x_n \\ \hline c_1 \\ \vdots \\ c_m \\ \hline\end{array}$$

We will formalize the meaning of DRSs in the next two subsections. For now, we can think of the DRS $\langle\{x_1, \ldots, x_n\}, \{c_1, \ldots, c_m\}\rangle$ as standing for the formula $\exists x_1 \ldots x_n.\, c_1 \wedge \ldots \wedge c_m$.

DRSs are computed incrementally, starting from the empty DRS:

$$\begin{array}{|c|}\hline \\ \hline \\ \hline\end{array}$$

If we were to interpret the discourse in Example 6, we would start with the first sentence. Both indefinites and proper nouns are handled by introducing a new discourse referent and a condition that describes them. After interpreting the subject *Jones* and the object *a Porsche*, we would get the following DRS:

$$\begin{array}{|c|}\hline x\ y \\ \hline \textbf{Jones}\, x^{103} \\ \textbf{Porsche}\, y \\ \hline\end{array}$$

The transitive verb *owns* will then add another condition to the DRS. Its arguments will be the discourse referents that correspond to the subject and the object.

$$\begin{array}{|c|}\hline x\ y \\ \hline \textbf{Jones}\, x \\ \textbf{Porsche}\, y \\ \textbf{own}\, x\, y \\ \hline\end{array}$$

[103] In our formalization of DRT, we do not admit constants. Instead of having a constant **j** for Jones, we have a unary relation **Jones** which is satisfied only by Jones (or only by people whose name is "Jones"). The consequence of this is that in DRT, we can only refer to individuals via discourse referents, which must be introduced in the universe of some DRS.

This DRS is the result of interpreting the first sentence of Example 6. We can now use it as the context for interpreting the second sentence. Pronouns choose a discourse referent accessible from the DRS in which they are to be interpreted.[104] The neutral pronoun *it* will choose y, since from the presence of the **Porsche** y condition, we can infer it designates an inanimate object. The masculine pronoun *him* will select x because it designates a masculine entity. Then as in the case of the previous sentence, the transitive verb contributes a condition that will relate the referent of the subject to the referent of the object.

$$\begin{array}{|c|}\hline x\ y \\ \hline \textbf{Jones}\ x \\ \textbf{Porsche}\ y \\ \textbf{own}\ x\ y \\ \textbf{fascinate}\ y\ x \\ \hline \end{array}$$

The precise mechanism of DRS construction will be given in Section 7.1, where we will see the derivation of the DRS for Example 6 in more detail. In the rest of this section, we will talk about the truth-conditional meaning of DRSs and of the accessibility relation.

5.3.2 DRSs as Contents

The result of interpreting a discourse in DRT is a DRS. Our interest in interpreting natural language is to be able to recover the truth conditions of a linguistic utterance. Therefore, we need a way to read off the truth conditions of a discourse from a DRS. We find that the most instructive way to give a formal semantics to DRT is to give a translation to an already established logic. In the case of our formalization of DRT, we will be translating DRSs and DRS conditions to first-order logic.[105]

Definition 5.3.3. *We define a* ***mapping*** $(\_)^{fo}$ ***from DRSs and DRS conditions to first-order logic formulas*** *by the following rules:*

- $$\begin{array}{|c|}\hline x_1 \ldots x_n \\ \hline c_1 \\ \vdots \\ c_m \\ \hline \end{array}^{fo} = \exists x_1 \ldots x_n.\, c_1^{fo} \wedge \ldots \wedge c_m^{fo}$$
- $$\left(\begin{array}{|c|}\hline x_1 \ldots x_n \\ \hline c_1 \\ \vdots \\ c_m \\ \hline \end{array} \Rightarrow K\right)^{fo} = \forall x_1 \ldots x_n.\, (c_1^{fo} \wedge \ldots \wedge c_m^{fo}) \rightarrow K^{fo}$$
- $(\boldsymbol{P}\, x_1\ \ldots\ x_n)^{fo} = \boldsymbol{P}\, x_1\ \ldots\ x_n$
- $(\neg K)^{fo} = \neg(K^{fo})$
- $(K_1 \vee K_2)^{fo} = K_1^{fo} \vee K_2^{fo}$

In the translation defined above, the first two rules expose the nature of DRSs. As more and more conditions are added to a DRS, they all fall under the scope of the existential quantifiers introduced by the universe of the DRS. This intuitively corresponds to the $(\exists x.\, A) \wedge B = (\exists x.\, A \wedge B)$ law of dynamic logic. Furthermore, any (existentially quantified) variables contributed by a DRS which acts as an *antecedent* in

[104] Accessibility will be formally defined in a coming subsection.

[105] There exist varieties of DRT that are more complex and might include, e.g., modal operators. There, it might be necessary to translate to some other logic.

an implication take scope over the *consequent* of the implication using universal quantification. This is a realization of the $(\exists x.\, A) \rightarrow B = (\forall x.\, A \rightarrow B)$ law of dynamic logic.

We can now retrieve the truth conditions of the DRS that we have computed in the previous subsection.

$$\begin{array}{|c|}\hline x\ y \\ \hline \textbf{Jones}\, x \\ \textbf{Porsche}\, y \\ \textbf{own}\, x\, y \\ \textbf{fascinate}\, y\, x \\ \hline \end{array}^{fo} = \exists xy.\, \textbf{Jones}\, x \wedge \textbf{Porsche}\, y \wedge \textbf{own}\, x\, y \wedge \textbf{fascinate}\, y\, x$$

Assuming that **j** is the only individual that satisfies the **Jones** predicate, this proposition is equivalent to the reading that we have given before.

As another example, we can take the DRS that is the result of interpreting the sentence in Example 7.

$$\begin{array}{|c|}\hline \\ \hline \begin{array}{|c|}\hline x\ y \\ \hline \textbf{farmer}\, x \\ \textbf{donkey}\, y \\ \textbf{own}\, x\, y \\ \hline \end{array} \Rightarrow \begin{array}{|c|}\hline \\ \hline \textbf{beat}\, x\, y \\ \hline \end{array} \\ \hline \end{array}^{fo} = \forall xy.\, (\textbf{farmer}\, x \wedge \textbf{donkey}\, y \wedge \textbf{own}\, x\, y) \rightarrow \textbf{beat}\, x\, y$$

Note that in the above DRS, the discourse referents x and y in the consequent of the implication are bound by the universe of the DRS that is the antecedent of the implication. We will explain the notions of scope and accessibility in DRT in the next subsection.

5.3.3 DRSs as Contexts

After having interpreted a part of discourse as a DRS, that DRS then serves as the *context* in which further discourse is to be evaluated. An important feature of the context is the set of discourse referents it makes available for anaphoric binding in subsequent discourse. The availability of these referents is governed by the notion of *accessibility*, which we define formally along with the notion of DRS *subordination*.

Definition 5.3.4. *An occurrence of a DRS $K_i = \langle U_i, C_i \rangle$ within a DRS K is **immediately subordinate** to an occurrence of a DRS $K_j = \langle U_j, C_j \rangle$ within the same DRS if one of the following is true:*

- $\neg K_i \in C_j$
- $K_i \Rightarrow K' \in C_j$ *for some DRS* K'
- $K_i \vee K' \in C_j$ *for some DRS* K'
- $K' \vee K_i \in C_j$ *for some DRS* K'
- $K_j \Rightarrow K_i$ *occurs within* K

Definition 5.3.5. *The relation "K_i is **(weakly) subordinate** to K_j" is defined as the (reflexive)-transitive closure of the relation "K_i is immediately subordinate to K_j".*

Definition 5.3.6. *An occurrence of a DRS $K_i = \langle U_i, C_i \rangle$ and the discourse referents from its universe U_i are **accessible** to an occurrence of a DRS K_j if and only if K_j is weakly subordinate to K_i.*

The definition of subordination can be summed up by saying that a DRS subordinates every DRS it contains, and within implications, the antecedent subordinates the consequent. This definition of accessibility is compatible with our translation from DRSs to first-order logic formulas. A discourse referent x is accessible to a DRS K if and only if the first-order translation of K falls within the scope of the quantifier of x.

$$K_0: \begin{array}{|c|}\hline \\ \hline K_1: \begin{array}{|c|}\hline x\ y \\ \hline \textbf{farmer}\ x \\ \textbf{donkey}\ y \\ \textbf{own}\ x\ y \\ \hline \end{array} \Rightarrow K_2: \begin{array}{|c|}\hline \\ \hline \textbf{beat}\ x\ y \\ \hline \end{array} \\ \hline \end{array}$$

In the above, x and y, the discourse referents of K_1, are available to K_2. Therefore, the occurrences of x and y in K_2 are bound. However, x and y are not available to K_0. This means that if we were to continue the discourse in Example 7 with the sentence *he is cruel*, we could not interpret the pronoun *he* as referring to the farmer x since new contributions are evaluated within the main DRS K_0. This correctly predicts the infelicity of the discourse in Example 9.

(9) * Every farmer$_1$ who owns a donkey$_2$ beats it$_2$. He$_1$ is cruel.

5.3.4 Removing Implication and Disjunction

In dynamic logics such as DPL [53] or TTDL [38, 80], implication, disjunction and universal quantification are expressed in terms of conjunction, negation and existential quantification, which are primitive. In DRT, we can do something similar and reduce the $\Rightarrow$ connective (which plays the role of implication and universal quantification) and the $\vee$ connective to the other primitives.

$$\begin{array}{|c|}\hline x_1 \ldots x_n \\ \hline c_1 \\ \vdots \\ c_m \\ \hline \end{array} \Rightarrow K \quad = \quad \neg \begin{array}{|c|}\hline x_1 \ldots x_n \\ \hline c_1 \\ \vdots \\ c_m \\ \neg K \\ \hline \end{array}$$

$$K_1 \vee K_2 \quad = \quad \neg \begin{array}{|c|}\hline \\ \hline \neg K_1 \\ \neg K_2 \\ \hline \end{array}$$

In both cases, it is easy to verify that the truth conditions are not affected: applying the $(\_)^{fo}$ mapping to both sides of an equation yields equivalent propositions. Similarly, we can verify that the reduction preserves accessibility (i.e. that the set of available discourse referents in an argument to either $\Rightarrow$ or $\vee$ does not change). After the reduction, the consequent of the implication is still immediately subordinate to the antecedent. As for disjunction, neither of the disjuncts is subordinate to the other.

5.4 Type-Theoretic Dynamic Logic

We end this chapter by reviewing another theory of dynamics which served as a basis for our analysis of anaphora and presupposition in Chapter 7. The theory originates from de Groote's paper on a montagovian treatment of dynamics [38] and it was extended in the dissertations of Lebedeva [80] and Qian [111], where it is referred to as Type-Theoretic Dynamic Logic (TTDL). TTDL aims to reproduce the predictions of DRT and dynamic semantics while using only standard logical tools. Unlike DPL and DRT, there is no need to introduce a new logic with non-standard notions of binding and scope. Meanings are represented as functions and propositions in the simply-typed λ-calculus (i.e. higher-order logic) and their composition does not necessitate the use of any special primitive operator (such as the merge operator on DRSs [99]).

We will start our introduction of TTDL by showing the meaning that is assigned to the two sentences of Example 6 and then we will explain what is going on.

$$[\![\text{Jones owns a Porsche}]\!] = \lambda e\phi.\, \exists x.\, \textbf{Porsche}\, x \wedge \textbf{own}\, \textbf{j}\, x \wedge \phi\, (x :: \textbf{j} :: e)$$
$$[\![\text{It fascinates him}]\!] = \lambda e\phi.\, \textbf{fascinate}\, (\mathsf{sel}_{\mathsf{it}}\, e)\, (\mathsf{sel}_{\mathsf{he}}\, e) \wedge \phi\, e$$

As we have seen in 5.3, sentences with anaphoric pronouns cannot be given a truth value unless we know the context in which they have been produced. The meanings are therefore functions of their environments, also called (left) *contexts*. By convention, we use the variable e for these environments. In the denotation for the second sentence, we retrieve the referents of the two anaphoric pronouns from the context e using the functions $\mathsf{sel}_{\mathsf{it}}$ and $\mathsf{sel}_{\mathsf{he}}$. The denotations also need a way to update the context, e.g. when introducing new discourse referents. To enable this, the meanings have access to their continuation (also called a right context), for which we will always use the variable ϕ. In the denotation for the first sentence, we apply this continuation to a context which was updated with the discourse referents **j** (for Jones) and x (for the Porsche). The nature of the context in TTDL is left open. In [38], the context is taken to be a list of available discourse referents, hence the ML-style syntax for consing the elements x and **j** to the context e. Finally, in 5.3, we have also seen that when interpreting indefinites with existential quantifiers, we need the scope of the quantifier to extend and cover the meanings of any sentences that follow. The use of continuations allows us to accomplish that. After the denotation of a sentence applies the continuation to the updated context, it can place the resulting proposition under the scope of any existential quantifiers it introduces, as is the case in the denotation of the first sentence.

We will now look at the types involved. As in Church's simple theory of types, there is the o type for propositions and the ι type for individuals. The contexts that sentences have access to are of type γ. The continuations take (updated) contexts as arguments and return propositions which represent the meaning of the following discourse, hence their type is $\gamma \to o$. Therefore, the type of the denotations that is assigned to sentences in TTDL is $\gamma \to (\gamma \to o) \to o$. This type is abbreviated as Ω and terms of this type are called *dynamic propositions*. Below, we give the types of the auxiliary operations on contexts.

$$\mathsf{nil} : \gamma$$
$$(\_ :: \_) : \iota \to \gamma \to \gamma$$
$$\mathsf{sel}_{\mathsf{he}}, \mathsf{sel}_{\mathsf{she}}, \mathsf{sel}_{\mathsf{it}} : \gamma \to \iota$$

Existing work on TTDL focuses on discourse representation and composition rather than anaphora resolution. Anaphora resolution is delegated to the sel oracle functions, which are assumed to retrieve the correct antecedent from the environment.

The meanings of the two sentences we have given above are both dynamic propositions. TTDL gives us a way to compose dynamic propositions using the discourse concatenation operator $(\_\,.\,\_)$:

$$(\_\,.\,\_) : \Omega \to \Omega \to \Omega$$
$$A\,.\,B = \lambda e\phi.\, A\, e\, (\lambda e'.\, B\, e'\, \phi)$$

The left context e of the resulting proposition is passed directly to A. The right context of A contains both the dynamic proposition B and the right context ϕ. The left context passed to B is the updated context e' that A passes to its continuation. The right context of B is the right context ϕ of the resulting proposition $A\,.\,B$.

Using the discourse concatenation operator, we can compute the meaning of the discourse from Example 6.

$$\begin{aligned}
&[\![\text{Jones owns a Porsche. It fascinates him.}]\!] \\
&= [\![\text{Jones owns a Porsche}]\!]\,.\,[\![\text{It fascinates him}]\!] \\
&= \lambda e\phi.\, \exists x.\, \textbf{Porsche}\, x \wedge \textbf{own}\, \textbf{j}\, x \wedge \textbf{fascinate}\, (\mathsf{sel}_{\mathsf{it}}\, (x :: \textbf{j} :: e))\, (\mathsf{sel}_{\mathsf{he}}\, (x :: \textbf{j} :: e)) \wedge \phi\, (x :: \textbf{j} :: e) \\
&= \lambda e\phi.\, \exists x.\, \textbf{Porsche}\, x \wedge \textbf{own}\, \textbf{j}\, x \wedge \textbf{fascinate}\, x\, \textbf{j} \wedge \phi\, (x :: \textbf{j} :: e)
\end{aligned}$$

In the above, we assume that anaphora resolution will pick up the Porsche x as the antecedent for the pronoun *it* (i.e. $\mathsf{sel}_{\mathsf{it}}\,(x{::}\mathbf{j}{::}e) = x$) and Jones, **j**, as the antecedent for the pronoun *him* (i.e. $\mathsf{sel}_{\mathsf{he}}\,(x{::}\mathbf{j}{::}e) = \mathbf{j}$).

We can retrieve the truth conditions of the discourse by applying it to some default context (e.g. the nil context which does not make available any discourse referent) and the empty right context (i.e. the continuation $\lambda e.\,\top$, where $\top$ is a tautology).

$$\begin{aligned}&\mathsf{static} : \Omega \to o\\ &\mathsf{static} = \lambda A.\, A\,\mathsf{nil}\,(\lambda e.\,\top)\end{aligned}$$

We can define a combinator static which will map dynamic propositions to static ones by interpreting them in the empty context. Applying this combinator to the dynamic proposition that is the meaning of Example 6 gives us the desired truth conditions.

$$\mathsf{static}\,[\![\text{Jones owns a Porsche. It fascinates him.}]\!] = \exists x.\,\mathbf{Porsche}\,x \wedge \mathbf{own}\,\mathbf{j}\,x \wedge \mathbf{fascinate}\,x\,\mathbf{j}$$

5.4.1 Logic

We have seen the notion of a dynamic proposition. In formal logics, propositions are defined as expressions of a formal language which includes logical connectives such as conjunction, disjunction and quantifiers. In her dissertation [80], Lebedeva defines such connectives for TTDL.[106]

$$\begin{aligned}A \mathbin{\overline{\wedge}} B &= \lambda e\phi.\, A\,e\,(\lambda e'.\, B\,e'\,\phi)\\ \overline{\neg} A &= \lambda e\phi.\, \neg\,(A\,e\,(\lambda e'.\,\top)) \wedge \phi\,e\\ \overline{\exists}\,P &= \lambda e\phi.\, \exists x.\, P\,x\,(x :: e)\,\phi\\ A \mathbin{\overline{\Rightarrow}} B &= \overline{\neg}\,(A \mathbin{\overline{\wedge}} \overline{\neg} B)\\ A \mathbin{\overline{\vee}} B &= \overline{\neg}\,(\overline{\neg} A \mathbin{\overline{\wedge}} \overline{\neg} B)\\ \overline{\forall}\,P &= \overline{\neg}\,(\overline{\exists}\,x.\, \overline{\neg}\,(P\,x))\end{aligned}$$

Implication, disjunction and universal quantification are expressed in terms of conjunction, negation and universal quantification, as in DPL (3.4 in [53]) or DRT (Subsection 5.3.4). Dynamic conjunction is the same thing as the discourse concatenation combinator, i.e. we acknowledge the discourse update of both of the conjuncts. Dynamic negation denies the discourse update of the negated proposition by applying it to the empty right context $\lambda e'.\,\top$. Applying the current left context e and the empty right context to the proposition A and then negating it is like interpreting A within its own DRS K and then producing the condition $\neg K$. The context that is passed to the continuation of the negation is the same context e that the negated proposition received. The dynamic existential quantifier wraps an (ordinary) existential quantifier over the predicate *as well as* its continuation ϕ. This has the effect of achieving the desired right-extension of existential quantifiers in dynamic semantics (i.e. $(\exists x.\, A) \wedge B = (\exists x.\, A \wedge B)$). At the same time, it also introduces the variable x into the context as an available discourse referent.

5.4.2 Monadic Structure

Within TTDL, we find the structure of a monad (definition in 3.3.6). The dynamics monad is given by the following functor D and combinators η and $\gg\!=$:

$$\begin{aligned}D(\alpha) &= \gamma \to (\alpha \to \gamma \to o) \to o\\ \eta\,x &= \lambda e\phi.\, \phi\,x\,e\\ X \gg\!= f &= \lambda e\phi.\, X\,e\,(\lambda x e'.\, f\,x\,e'\,\phi)\end{aligned}$$

[106] The definitions below differ slightly from those of Lebedeva [80]. Instead of updating the context with available individuals, Lebedeva updates the context with facts added to the common ground. Instead of using individuals (type ι), Lebedeva uses dynamic individuals (type $\gamma \to \iota$).

These operations satisfy all three of the monad laws. The D monad characterizes computations that have read/write access to some state of type γ and can manipulate their continuation of type o. We can actually derive D as the combination of the state monad and the continuation monad using monad transformers. Let S be the state monad which maps the type α to the type $\gamma \to (\alpha \times \gamma)$. This is the monad that underlies state, such as the storing and retrieving of discourse referents. There exists also a monad transformer S^T which maps monads F into monads $S(F)$ that enrich the structure of F with state. The monad $S(F)$ maps the type α to the type $\gamma \to F(\alpha \times \gamma)$. Let C be the continuation monad which maps the type α to the type $(\alpha \to o) \to o$. By applying the S monad transformer to the C monad, we get a monad $S(C)$ that maps the type α to the type $\gamma \to ((\alpha \times \gamma) \to o) \to o$. This type is isomorphic to the type $D(\alpha) = \gamma \to (\alpha \to \gamma \to o) \to o$ by currying.

We have presented the D monad. Now, we will show how it features in TTDL. We will focus on the type $D(1)$, i.e. the type of computations that produce no output and where we are only interested in the side effects.

$$D(1) \;=\; \gamma \to (1 \to \gamma \to o) \to o \;\equiv\; \gamma \to (\gamma \to o) \to o \;=\; \Omega$$

The types $D(1)$ and Ω are isomorphic due to the isomorphism between the types $1 \to \gamma$ and γ. We can also take the η and $\gg\!=$ combinators, specialize them to the $D(1)$ type and then apply the $1 \to \gamma \equiv \gamma$ isomorphism to simplify them. On the type level, η will end up being a dynamic proposition and $\gg\!=$ a binary operator on dynamic propositions.

$$\begin{aligned}
\eta &: \alpha \to D(\alpha) \\
\eta_1 &: 1 \to D(1) \\
&: D(1) \\
&: \Omega \\
\gg\!= &: D(\alpha) \to (\alpha \to D(\beta)) \to D(\beta) \\
\gg\!=_1 &: D(1) \to (1 \to D(1)) \to D(1) \\
&: D(1) \to D(1) \to D(1) \\
&: \Omega \to \Omega \to \Omega
\end{aligned}$$

On the term level, we find out that η_1 is the dynamic tautology and $\gg\!=_1$ is dynamic conjunction.

$$\begin{aligned}
\eta\, x &= \lambda e \phi.\, \phi\, x\, e \\
\eta_1 &= \lambda e \phi.\, \phi\, e \\
X \gg\!= f &= \lambda e \phi.\, X\, e\, (\lambda x e'.\, f\, x\, e'\, \phi) \\
X \gg\!=_1 Y &= \lambda e \phi.\, X\, e\, (\lambda e'.\, Y\, e'\, \phi)
\end{aligned}$$

Furthermore, by verifying the monad laws for D, we have verified the monoid laws for dynamic conjunction with dynamic tautology as the neutral element.

In TTDL, discourse is composed with the discourse concatenation operator, which is just the dynamic conjunction operation. Now we have seen that dynamic conjunction corresponds to monadic composition in the D monad. Therefore, composition of discourse corresponds to composition within the D monad.

5.4.3 Example Lexical Entries

We end our exposition of TTDL by a quick glance at the semantics of a tiny fragment, a subset of the one in [38]. We start by giving the abstract syntactic types to the lexical items in our fragment.

$$\begin{aligned}
\textsc{Mary}, \textsc{she} &: NP \\
\textsc{a}, \textsc{every} &: N \multimap NP \\
\textsc{woman} &: N \\
\textsc{love} &: NP \multimap NP \multimap S
\end{aligned}$$

Next, we identify the types of interpretations that we will use.

$$\begin{aligned}
[\![S]\!] &= \Omega \\
[\![N]\!] &= \iota \to \Omega \\
[\![NP]\!] &= (\iota \to \Omega) \to \Omega
\end{aligned}$$

Sentences will denote dynamic propositions, nouns will denote dynamic properties (functions from individuals to dynamic propositions) and noun phrases will denote dynamic generalized quantifiers (same type as generalized quantifiers but with o replaced with Ω). We can now give a semantics to the lexical items.

$$\begin{aligned}
[\![\textsc{Mary}]\!] &= \lambda P e \phi.\, P\, \mathbf{m}\, (\mathbf{m} :: e)\, \phi \\
[\![\textsc{she}]\!] &= \lambda P e \phi.\, P\, (\mathsf{sel}_{\mathsf{she}}\, e)\, e\, \phi \\
[\![\textsc{a}]\!] &= \lambda N P.\, \overline{\exists}\, x.\, N\, x \,\overline{\wedge}\, P\, x \\
[\![\textsc{every}]\!] &= \lambda N P.\, \overline{\forall}\, x.\, N\, x \,\overline{\Rightarrow}\, P\, x \\
[\![\textsc{woman}]\!] &= \lambda x e \phi.\, \mathbf{woman}\, x \wedge \phi\, e \\
[\![\textsc{love}]\!] &= \lambda O S.\, S\, (\lambda s.\, O\, (\lambda o e \phi.\, \mathbf{love}\, s\, o \wedge \phi\, e))
\end{aligned}$$

The first four lexical entries give semantics to noun phrases. These are written as generalized quantifiers: they all abstract over a variable P which defines the scope of the quantifier. Neither the proper noun Mary nor the pronoun she are quantificational and so they dispose of P by applying it immediately to the individuals that they refer to. However, the proper noun Mary takes advantage of the continuation P to access the context e and modify it to include the discourse referent for Mary in the following context. Similarly, the pronoun she uses the continuation P to access the context e and find within it a suitable referent. The denotations for the determiners a and every are the same as the ones we have seen in 5.2.3, only the classical logical operators have been replaced with the dynamic ones from 5.4.1. The lexical entry for the common noun woman differs in that the proposition **woman** x had to be raised from a (static) proposition to a dynamic proposition. Finally, we have the lexical entry for the transitive verb love. The solution here differs to the one in 5.2.3. The abstract type NP is interpreted as the type of (dynamic) generalized quantifiers, not just individuals. The transitive verb, abstract type $NP \multimap NP \multimap S$, thus has access to the generalized quantifiers of the subject and the object, not just their referents. The entry arranges those two quantifiers so as to always generate a subject-wide scope reading. This restriction is necessary, because arranging the quantifiers in an object-wide scope configuration would make it so that the discourse referents from the object could bind anaphoric elements of the subject, which is not the linguistic reality we want to model. We return to the problem of generating inverse scope while avoiding licensing right-to-left anaphora (cataphora) in 8.5.1.

In 5.4.2, we have seen that the monadic structure underlying TTDL is composed of state manipulation (the state monad — introducing and recovering discourse referents) and continuation manipulation (the continuation monad — scoping existential quantifiers over future discourse). Now we have seen that in the fragment from [38], there is another layer of continuation structure at the level of the noun phrases/generalized quantifiers. In Section 6.4, we will express the continuation-passing style of generalized quantifiers using $(\!|\lambda|\!)$ operations and handlers, and in Chapter 7, we will do the same for the state and the continuations at the level of discourse dynamics. Then, in Chapter 8, we will build a grammar that will, among other effects and phenomena, include all of these layers.

6

Introducing the Effects

In this chapter, we will take a miniature fragment and extend it in different directions, studying various linguistic phenomena. The common point in all of these analyses is that they will all rely on the notion of computations introduced in $(\!| \lambda |\!)$.

We will start by first taking the initial fragment and lifting its usual semantics into the domain of computations (6.1), so that it will be compatible with the development in the following sections. We will then consider the semantics of deictic pronouns (6.2), appositives (6.3) and quantified noun phrases (6.4). After a tour of these analyses, we will take a step back and reflect on the methodological process that underlies their development (6.5).

Contents

6.1 Lifting Semantics into Computations **127**
6.2 Deixis **131**
6.2.1 Quotations 132
6.2.2 Algebraic Considerations 133
6.3 Conventional Implicature **134**
6.3.1 Algebraic Considerations 136
6.4 Quantification **137**
6.4.1 Quantifier Ambiguity 139
6.4.2 Algebraic Considerations 142
6.5 Methodology **142**
6.5.1 Using Computation Types 143
6.5.2 Digression: Call-by-Name and Call-by-Value 144
6.5.3 Choosing an Effect Signature 146
6.5.4 When (Not) to Use Effects 150

6.1 Lifting Semantics into Computations

Let us start with a very tiny fragment with proper names to name individuals and verbs to act as predicates over these individuals:

$$\begin{aligned} \textsc{John}, \textsc{Mary} &: NP \\ \textsc{loves} &: NP \multimap NP \multimap S \end{aligned}$$

In this tiny fragment of proper names and predicates, we could interpret noun phrases as individuals ($[\![NP]\!] = \iota$) and sentences as propositions ($[\![S]\!] = o$). Provided we have some constants $\mathbf{j} : \iota$ and $\mathbf{m} : \iota$ and a predicate $\mathbf{love} : \iota \to \iota \to o$, we can give the following semantics to these items:

$$
\begin{aligned}
[\![\textsc{John}]\!] &= \mathbf{j} \\
[\![\textsc{Mary}]\!] &= \mathbf{m} \\
[\![\textsc{loves}]\!] &= \lambda os.\, \mathbf{love}\, s\, o
\end{aligned}
$$

This interpretation works fine for simple sentences such as *John loves Mary*.

However, the denotations that we will assign to noun phrases that are deictic or quantified will not fit into the type ι. Instead, we will interpret these constituents as *computations* producing individuals, type $\mathcal{F}_E(\iota)$. In order to satisfy the homomorphism property of ACGs and to have a sound syntax-semantics interface, we will need to lift the denotations of these basic constants and predicates into computations. This is very much like the case when one introduces quantified noun phrases and switches to using generalized quantifiers instead of simple individuals as denotations of noun phrases.

We will want linguistic expressions to denote computations. One systematic way to achieve that is to say that the atomic types of our abstract syntactic signature should be interpreted as computations. We will write $[\![-]\!]_{\mathrm{v}}$ for the semantic interpretation using simple values and $[\![-]\!]_{\mathrm{c}}$ for the semantic interpretation using computations. On the type level, we will define $[\![a]\!]_{\mathrm{c}} = \mathcal{F}_E([\![a]\!]_{\mathrm{v}})$ for atomic abstract types a. By applying this to the common Montagovian interpretation, we get:

$$
\begin{aligned}
[\![S]\!]_{\mathrm{v}} &= o & \qquad [\![S]\!]_{\mathrm{c}} &= \mathcal{F}_E(o) \\
[\![NP]\!]_{\mathrm{v}} &= \iota & \qquad [\![NP]\!]_{\mathrm{c}} &= \mathcal{F}_E(\iota) \\
[\![N]\!]_{\mathrm{v}} &= \iota \to o & \qquad [\![N]\!]_{\mathrm{c}} &= \mathcal{F}_E(\iota \to o)
\end{aligned}
$$

To lift the denotations of noun phrases from $[\![NP]\!]_{\mathrm{v}} = \iota$ to $[\![NP]\!]_{\mathrm{c}} = \mathcal{F}_E(\iota)$, it suffices to use η to inject ι inside of $\mathcal{F}_E(\iota)$. This goes the same for any other lexical item whose abstract type is an atomic type. For syntactic constructors that take arguments, such as verbs or adjectives, we will chain the computations of their arguments and apply the meaning of the constructor to the meaning of the results of these computations. We will limit ourselves to syntactic constructors of *second-order type*, i.e. abstract constants whose type is $a_1 \multimap \cdots a_n \multimap b$ where all a_i and b are atomic types. If we use higher-order syntactic constructors, we will give them a bespoke semantics.

$$
\begin{aligned}
\mathsf{lift}^{\mathsf{L}}_{\alpha} &: [\![\alpha]\!]_{\mathrm{v}} \to [\![\alpha]\!]_{\mathrm{c}} \\
\mathsf{lift}^{\mathsf{L}}_{a}(x) &= \eta\, x \\
\mathsf{lift}^{\mathsf{L}}_{a \multimap \beta}(f) &= \lambda X.\, X \gg\!= (\lambda x.\, \mathsf{lift}^{\mathsf{L}}_{\beta}(f\, x))
\end{aligned}
$$

This particular schema chains the evaluation of its arguments from left-to-right (as can be seen by looking at the expanded non-recursive definition below). While indexicality is order-independent, some of the effects that we will introduce later (such as anaphora) are order-dependent. The order that we would like to reflect in the evaluation is the linear lexical order in which the arguments appear in the spoken/written form of the sentence. Since in categorial grammars of English, it is often the case that operators first take their complements from the right and then apply to their argument on the left (e.g. transitive verbs or relative pronouns of type $(NP \multimap S) \multimap N \multimap N$), we will often like to chain the evaluation of the arguments in the order opposite to the one in which we receive them. This will give rise to the $\mathsf{lift}^{\mathsf{R}}$ operation.

$$
\begin{aligned}
\mathsf{lift}^{\mathsf{L}}_{a_1 \multimap \cdots \multimap a_n \multimap b}(f) &= \lambda X_1 \ldots X_n.\, X_1 \gg\!= (\lambda x_1.\, \ldots\, X_n \gg\!= (\lambda x_n.\, \eta\, (f\, x_1\, \ldots\, x_n))) \\
\mathsf{lift}^{\mathsf{R}}_{a_1 \multimap \cdots \multimap a_n \multimap b}(f) &= \lambda X_1 \ldots X_n.\, X_n \gg\!= (\lambda x_n.\, \ldots\, X_1 \gg\!= (\lambda x_1.\, \eta\, (f\, x_1\, \ldots\, x_n)))
\end{aligned}
$$

With these in hand, we can now lift the interpretations of our simple fragment into computations:

$$
\begin{aligned}
[\![\textsc{John}]\!]_{\mathrm{c}} &= \mathsf{lift}^{\mathsf{L}}_{NP}(\mathbf{j}) \\
&= \eta\,\mathbf{j} \\
[\![\textsc{Mary}]\!]_{\mathrm{c}} &= \mathsf{lift}^{\mathsf{L}}_{NP}(\mathbf{m}) \\
&= \eta\,\mathbf{m} \\
[\![\textsc{loves}]\!]_{\mathrm{c}} &= \mathsf{lift}^{\mathsf{R}}_{NP \multimap NP \multimap S}(\lambda os.\, \eta\,(\mathbf{love}\; s\; o)) \\
&= \lambda OS.\, S \gg\!= (\lambda s.\, O \gg\!= (\lambda o.\, \mathbf{love}\; s\; o)) \\
&= \lambda OS.\, \mathbf{love} \cdot\!\gg S \ll\!\cdot\!\gg O \\
&= \lambda OS.\, \eta\,\mathbf{love} \ll\!\cdot\!\gg S \ll\!\cdot\!\gg O
\end{aligned}
$$

In the lexical entry for LOVES, we can express the series of binds using the application operators from 1.6.1. The idea behind the notation is that you are supposed to put double brackets on a side of the operator whenever the argument on that side is a computation. In our example, S and O are both computations and **love** is a pure function term. We can expand this term so that it is a bit more regular by making **love** a computation as well. We can then a notice a connection between the denotations in $[\![-]\!]_{\mathrm{v}}$ and the denotations in $[\![-]\!]_{\mathrm{c}}$. The $[\![-]\!]_{\mathrm{c}}$ denotations are just the $[\![-]\!]_{\mathrm{v}}$ denotations where every constant **c** was replaced with $\eta\,\mathbf{c}$ and every application $f\,x$ was replaced with $f \ll\!\cdot\!\gg x$. Note that η and $\ll\!\cdot\!\gg$ make up the applicative functor $\mathcal{F}_E$ (see 3.3.5).

We close this section by proving that adding this computation layer does not affect the predictions that this semantics makes. First, we start with a lemma that will allow us to exploit the fact that we restrict ourselves to second-order types.

Lemma 6.1.1. ***Second-order terms hide no abstractions***

Every β-normal term $\Gamma \vdash M : \tau$ of second-order type τ in the simply-typed λ-calculus is of the form $\lambda x_1 \ldots x_n.\, N$ where N contains no abstractions, provided that any constants used have only a second-order type and any variables present in Γ have a first-order (i.e. atomic) type.

Proof. Let $M = \lambda x_1 \ldots x_n.\, N$ where N is not an abstraction (n might be 0). We will need to prove that N contains no abstractions.

We will use proof by contradiction. Let us assume that N contains some abstractions. We order the subterms of N left-to-right, depth-first. Let $N' = \lambda x.\, N''$ be the first abstraction we find in this traversal. N' must be a proper subterm of N since N is known not to be an abstraction.

We now consider the contexts in which N' can occur:

- N' occurs as the function in an application $N'\,A$

  This is in contradiction with M being a β-normal form since we have an abstraction in function position, i.e. a β-redex.

- N' occurs as the argument in an application $F\,N'$

  We know that F does not contain any abstractions since N' is the first abstraction we encountered in the left-to-right ordering of subterms. Furthermore, N' is the first abstraction in a depth-first ordering and so the only (free) variables that occur in F are the variables $x_1 \ldots x_n$ and the variables in Γ, all of which are of first-order type. F is therefore not a variable (since it must have a functional type). F is either a constant or an application. Furthermore, if F is an application, we can apply the same reasoning and by induction conclude that F must be a constant c applied to some n arguments, $F = c\,A_1\;\ldots\;A_n$, with n possibly being 0. However, since c is of second-order type, all of its arguments are of first-order type. This is in contradiction with N' being an abstraction.

- N' occurs as the body of an abstraction $\lambda y.\, N'$

  This is in contradiction with N' being the first abstraction we encounter in the depth-first order.

□

Corollary 6.1.2. ***Second-order terms are trees***

Assuming all constants are of second-order type, every closed term $\vdash M : \nu$ of atomic type ν has a β-normal form $c\, M_1 \ldots M_n$ where M_i are also closed terms of atomic types.

Proof. From Lemma 6.1.1, we know that the normal form of M is of the shape $\lambda x_1 \ldots x_n.\, N$ where N contains no abstraction. Furthermore, we know that M is of an atomic type, therefore $n = 0$ and the normal form is just N. Since N is closed, it contains no free variables, and since it is abstraction-free, it contains no bound variables either. Note that this is also the case for every subterm of N.

N is composed entirely of constants and applications. If N is an application, then the function is either a constant or some smaller application composed entirely of constants and applications. By induction, we thus show that M's normal form, N, is of the shape $c\, M_1\ \ldots\ M_n$. Since c has a second-order type, all of its arguments must have a first-order, atomic, type. □

Observation 6.1.3. ***Conservativity of lifting***

Let $\langle \Sigma_{\mathrm{a}}, \Sigma_{\mathrm{o}}, [\![-]\!]_{\mathrm{v}}, S\rangle$ be an ACG where every constant $c : \tau \in \Sigma_{\mathrm{a}}$ is of at most second-order type ($\tau = a_1 \multimap \ldots \multimap a_n \multimap b$ where all a_i and b are atomic types) and let $\langle \Sigma_{\mathrm{a}}, \Sigma_{\mathrm{o}}, [\![-]\!]_{\mathrm{c}}, S\rangle$ be the ACG whose lexicon $[\![-]\!]_{\mathrm{c}}$ satisfies the following conditions:

- *there exists some effect signature E such that for every abstract atomic type $\tau \in \Sigma_{\mathrm{a}}$, $[\![\tau]\!]_{\mathrm{c}} = \mathcal{F}_E([\![\tau]\!]_{\mathrm{v}})$*
- *for every abstract constant $c : \tau \in \Sigma_{\mathrm{a}}$, $[\![c]\!]_{\mathrm{c}} = \mathsf{lift}_\tau([\![c]\!]_{\mathrm{v}})$ where lift is either $\mathsf{lift}^{\mathsf{L}}$ or $\mathsf{lift}^{\mathsf{R}}$*[107]

Then for every closed well-typed abstract term $\vdash_{\Sigma_{\mathrm{a}}} M : \nu$ where ν is an atomic abstract type from Σ_{a}, we have:

$$[\![M]\!]_{\mathrm{c}} = \eta\,([\![M]\!]_{\mathrm{v}})$$

Proof. From Corollary 6.1.2, we know that M can be β-converted to a term of the form $c\, M_1\ \ldots\ M_n$ where all M_i are also closed abstract terms of atomic type. Let $a_1 \multimap \ldots a_n \multimap \nu$ be the type of c in Σ_{a}.

We will proceed by induction on the structure of this normal form:

- $M = c$ with $c : \nu \in \Sigma_{\mathrm{a}}$

  In that case, we have the desired property by definition of $[\![-]\!]_{\mathrm{c}}$.

- $M = c\, M_1\ \ldots\ M_n$ with $n > 0$

  First, we apply $[\![-]\!]_{\mathrm{c}}$ to M and make use of the induction hypothesis for M_i:

$$\begin{aligned}[\![c\, M_1\ \ldots\ M_n]\!]_{\mathrm{c}} &= [\![c]\!]_{\mathrm{c}}\, [\![M_1]\!]_{\mathrm{c}}\ \ldots\ [\![M_n]\!]_{\mathrm{c}} \\ &= \mathsf{lift}_{a_1 \multimap \ldots a_n \multimap \nu}([\![c]\!]_{\mathrm{v}})\,(\eta\,[\![M_1]\!]_{\mathrm{v}})\ \ldots\ (\eta\,[\![M_n]\!]_{\mathrm{v}})\end{aligned}$$

  If lift is $\mathsf{lift}^{\mathsf{L}}$, then:

$$\begin{aligned}&\mathsf{lift}^{\mathsf{L}}_{a_1 \multimap \ldots a_n \multimap \nu}([\![c]\!]_{\mathrm{v}})\,(\eta\,[\![M_1]\!]_{\mathrm{v}})\ \ldots\ (\eta\,[\![M_n]\!]_{\mathrm{v}}) \\ &= (\lambda X_1 \ldots X_n.\, X_1 \gg\!= (\lambda x_1.\ \ldots\ X_n \gg\!= (\lambda x_n.\, \eta\,([\![c]\!]_{\mathrm{v}}\, x_1\ \ldots\ x_n))))\,(\eta\,[\![M_1]\!]_{\mathrm{v}})\ \ldots\ (\eta\,[\![M_n]\!]_{\mathrm{v}}) \\ &= (\eta\,[\![M_1]\!]_{\mathrm{v}}) \gg\!= (\lambda x_1.\ \ldots\ (\eta\,[\![M_n]\!]_{\mathrm{v}}) \gg\!= (\lambda x_n.\, \eta\,([\![c]\!]_{\mathrm{v}}\, x_1\ \ldots\ x_n))) \\ &= \eta\,([\![c]\!]_{\mathrm{v}}\,[\![M_1]\!]_{\mathrm{v}}\ \ldots\ [\![M_n]\!]_{\mathrm{v}}) \\ &= \eta\,[\![c\, M_1\ \ldots\ M_n]\!]_{\mathrm{v}}\end{aligned}$$

  Similarly, if lift is $\mathsf{lift}^{\mathsf{R}}$, then:

[107] This result is analogous to Barker's Simulation Theorem for the Continuation Schema in [12]. As in Barker's theorem, this result holds not only for $\mathsf{lift}^{\mathsf{L}}$ and $\mathsf{lift}^{\mathsf{R}}$ but to liftings that arbitrarily permute the evaluation order of their arguments.

$$
\begin{aligned}
&\mathsf{lift}^{\mathsf{R}}_{a_1 \multimap \ldots a_n \multimap \nu}(\llbracket c \rrbracket_{\mathrm{v}})\,(\eta\,\llbracket M_1 \rrbracket_{\mathrm{v}})\ \ldots\ (\eta\,\llbracket M_n \rrbracket_{\mathrm{v}}) \\
&= (\lambda X_1 \ldots X_n.\, X_n \gg\!= (\lambda x_n.\ \ldots\ X_1 \gg\!= (\lambda x_1.\, \eta\,(\llbracket c \rrbracket_{\mathrm{v}}\, x_1\ \ldots\ x_n))))\,(\eta\,\llbracket M_1 \rrbracket_{\mathrm{v}})\ \ldots\ (\eta\,\llbracket M_n \rrbracket_{\mathrm{v}}) \\
&= (\eta\,\llbracket M_n \rrbracket_{\mathrm{v}}) \gg\!= (\lambda x_n.\ \ldots\ (\eta\,\llbracket M_1 \rrbracket_{\mathrm{v}}) \gg\!= (\lambda x_1.\, \eta\,(\llbracket c \rrbracket_{\mathrm{v}}\, x_1\ \ldots\ x_n))) \\
&= \eta\,(\llbracket c \rrbracket_{\mathrm{v}}\, \llbracket M_1 \rrbracket_{\mathrm{v}}\ \ldots\ \llbracket M_n \rrbracket_{\mathrm{v}}) \\
&= \eta\,\llbracket c\, M_1\ \ldots\ M_n \rrbracket_{\mathrm{v}}
\end{aligned}
$$

□

Corollary 6.1.4. *For every term M in the abstract language of a second-order ACG with an atomic distinguished type S, we have:*

$$\llbracket M \rrbracket_{\mathrm{c}} = \eta\,(\llbracket M \rrbracket_{\mathrm{v}})$$

6.2 Deixis

The first phenomenon that we will speak about is *deixis* [81]. Deictic expressions is the class of expressions that depend on the time and place of the utterance, the speaker and the addressee and any kind of pointing/presenting the speaker might be doing to draw the attention of the addressee. These expressions include personal pronouns, temporal expressions, tenses, demonstratives and others. All of these are characterized by their dependence on the extra-linguistic context. In this section, we will restrict our attention to a very limited subset of these expressions: singular first-person pronouns (*I*, *me*).

$$\textsc{me} : NP$$

The meanings that we assign to expressions in natural languages must reflect this context-sensitivity: the truth conditions of *Mary loves me* change when it is pronounced by John and when by Peter. Montague [98] achieved this by having the meaning of every expression depend on a point of reference: a pair of a possible world and a moment in time (i.e. the modal *where* and the *when* of the utterance). To model first-person pronouns, we will need to have our meanings depend on the identity of the speaker.

In the case of a deictic expression, we have an expression whose referent cannot be determined solely from its form and the meaning of its parts. We will need to reach out into the context and it is for this that we will be using the *operations* in $(\!|\,\lambda\,|\!)$. The first-person pronoun has an interaction with its context, which consists of asking the context for the identity of the speaker. For this kind of interaction with the context, we will introduce an operation symbol, `speaker`. We will also fix the symbol's input and output types. The input type represents the information and/or the parameters that the denotation of the first-person pronoun or any other expression necessitating the identity of the speaker will need to provide to the context. Since we have no information or parameter to give to the context, we will use the trivial input type 1, whose only value is $\star$. The output type represents the information that the context will provide us in return. We are interested in the identity of the speaker and so the type of this information will be the type of individuals, ι.

We can now model the meaning of a first-person pronoun as a computation of type $\mathcal{F}_E(\iota)$ that interacts with the context and produces a referent of type ι. The effect signature E can be any signature provided that $\texttt{speaker} : 1 \rightarrowtail \iota \in E$.

$$
\begin{aligned}
\llbracket \textsc{me} \rrbracket &= \texttt{speaker}\ \star\ (\lambda x.\, \eta\, x) \\
&= \texttt{speaker!}\,\star
\end{aligned}
$$

The denotation of $\textsc{me}$ demands the context for the identity of the speaker x using the operation `speaker` and then declares that x to be its. Taking the output of an operation and then immediately

returning it as the result of the computation will be a common pattern and so we use the `speaker`! shorthand introduced in 1.6.2.

Now the question is how to use the denotation given above to build meanings of sentences containing first-person pronouns, e.g. *Mary loves me*. In Montague's use of points of reference, Montague introduces an intermediate language of intensional logic [98]. When Montague then gives an interpretation to this language, the point of reference at which an expression is to be evaluated is passed through to its subexpressions. We will be making use of the lifted semantics introduced in the previous section. The chaining of the computations will serve our purpose of propagating the indexicality of the noun phrase to the sentence containing it (or vice versa, the propagation of the sentence's speaker to the noun phrase contained within).

With the interpretations given before, we can now analyse the following sentences:

(10) John loves Mary.

(11) John loves me.

whose meanings we can calculate as:

$$[\![\textsc{loves Mary John}]\!] \twoheadrightarrow \eta\,(\mathbf{love\ j\ m}) \tag{10}$$

$$[\![\textsc{loves me John}]\!] \twoheadrightarrow \texttt{speaker} \star (\lambda x.\, \eta\,(\mathbf{love\ j}\ x)) \tag{11}$$

For (10), we get a pure computation that produces the proposition **love j m**, which is the same proposition that the sentence denoted before we modified the fragment to use computations. However, in the case of (11), we do not have any single proposition as the denotation. Instead, we have a request to identify the speaker of the utterance and then we have a different proposition **love j** x for each possible speaker x. The truth conditions of this sentence can only be found by considering some hypothetical speaker s. Given such a speaker, we could resolve all of the requests for the speaker's identity and since our effect signature E does not contain any other operations, arrive at the desired truth conditions. This function that will interpret the `speaker` operation symbols will be a handler.

$$\mathsf{withSpeaker} : \iota \to \mathcal{F}_{\{\texttt{speaker}:1 \rightarrowtail \iota\} \uplus E}(\alpha) \to \mathcal{F}_E(\alpha)$$
$$\mathsf{withSpeaker} = \lambda s.\, (\!|\, \texttt{speaker}\colon (\lambda\_.\, k\, s) \,|\!)$$ [108]

The type tells us that $\mathsf{withSpeaker}\ s$ is a handler for the `speaker` operation. It takes any computation of type $\mathcal{F}_{\{\texttt{speaker}:1 \rightarrowtail \iota\} \uplus E}(\alpha)$ and gives back a computation of type $\mathcal{F}_E(\alpha)$, in which `speaker` will not be used. Since `speaker` is the only operation in our effect signature, by applying this handler to the denotation of a sentence in our fragment, we get a denotation of type $\mathcal{F}_\emptyset(o)$, which is isomorphic to o.[109]

$$\mathsf{withSpeaker}\ s\ [\![\textsc{loves me John}]\!] \twoheadrightarrow \eta\,(\mathbf{love\ j}\ s)$$

6.2.1 Quotations

Up to now, we could have assumed that at the object level, we have a constant **speaker** standing in for the speaker. After adding a constant to our logical signature, our new models would have interpretations for symbols in the original signature and for the **speaker** constant, i.e. our models would become descriptions of the world paired with some deictic index. However, removing the notion of a context and making the speaker be a part of the model would make it difficult to analyze the difference between the following two sentences:

(12) John said Mary loves me.

[108] We will be using _ as the name of a variable whose value we are not interested in. Such a variable will always have the trivial type 1.

[109] The two directions of the isomorphism are given by $\downarrow_\circ : \mathcal{F}_\emptyset(\alpha) \to \alpha$ and $\eta : \alpha \to \mathcal{F}_\emptyset(\alpha)$.

(13) John said "Mary loves me".

In our setting, we can model this kind of behavior since the withSpeaker handler is not a meta-level operation, but it is a term in our calculus like any other. We can therefore have lexical entries for direct and indirect speech that will interact differently with deictic expressions.

$$\textsc{said}_{\textsc{is}} : S \multimap NP \multimap S$$
$$\textsc{said}_{\textsc{ds}} : S \multimap NP \multimap S$$

We have two lexical items that correspond to the use of SAID in both direct speech and indirect speech. They differ in surface realization (both in prosody and punctuation) *and* semantic interpretation.[110]

$$\begin{aligned}
[\![\textsc{said}_{\textsc{is}}]\!] &= \lambda CS.\, \mathbf{say} \cdot\!\gg S \ll\!\cdot\!\gg C \\
&= \lambda CS.\, S \gg\!= (\lambda s.\, \mathbf{say}\, s \cdot\!\gg C) \\
[\![\textsc{said}_{\textsc{ds}}]\!] &= \lambda CS.\, S \gg\!= (\lambda s.\, \mathbf{say}\, s \cdot\!\gg (\mathsf{withSpeaker}\, s\, C))
\end{aligned}$$

The indirect speech use of *said* has the same kind of lexical entry as the transitive verb *loves*. In the direct speech entry, we would like to bind the speaker within the complement clause to the referent of the subject. We will therefore want to wrap the complement clause in a handler for `speaker`. However, the handler $(\mathsf{withSpeaker}\, s)$ needs to know the referent s of the subject S. We will need to first evaluate S and bind its result to s. To highlight the fact that the two entries differ only in the use of the $(\mathsf{withSpeaker}\, s)$ handler, we have expanded the entry for indirect speech into the same form. Also note that in this solution, we have had to use $\gg\!=$ and we cannot get by with only η and $\ll\!\cdot\!\gg$ (i.e. we need not only an applicative functor, but also a monad).

We can now plug this new entry in and compute the meanings of Examples 12 and 13.

$$[\![\textsc{said}_{\textsc{is}}\, (\textsc{loves me}\, \textsc{Mary})\, \textsc{John}]\!] \twoheadrightarrow \texttt{speaker} \star (\lambda x.\, \eta\, (\mathbf{say\ j}\, (\mathbf{love\ m}\, x))) \tag{12}$$
$$[\![\textsc{said}_{\textsc{ds}}\, (\textsc{loves me}\, \textsc{Mary})\, \textsc{John}]\!] \twoheadrightarrow \eta\, (\mathbf{say\ j}\, (\mathbf{love\ m\ j})) \tag{13}$$

In (12), the `speaker` operation projects outside the complement clause and we end up with another speaker-dependent proposition. On the other hand, in (13), the dependence on the identity of the speaker has been discharged by the handler contained in the denotation of $\textsc{said}_{\textsc{ds}}$.

6.2.2 Algebraic Considerations

One of the traditions from which the technique of effects and handlers originates is the study of algebraic effects by Plotkin, Power, Pretnar and Hyland [60, 103, 110, 104]. The semantics of a system of operations is not given by handlers but by a system of equations. Instead of writing a handler which would interpret two computations as the same object, we would give equations that let us prove the two computations equivalent. This perspective can give us some insights.

Assuming that the `speaker` operation is only ever handled by the withSpeaker handler, the following equations become admissible:

$$\begin{aligned}
\texttt{speaker} \star (\lambda x.\, \texttt{speaker} \star (\lambda y.\, M(x,y))) &= \texttt{speaker} \star (\lambda x.\, M(x,x)) \\
M &= \texttt{speaker} \star (\lambda x.\, M)
\end{aligned}$$

M is a metavariable ranging over computations of type $\mathcal{F}_{\{\texttt{speaker}:1 \rightarrowtail \iota\}}(\alpha)$.[111] We can check by reduction that whenever $M_1 = M_2$ via the above equations, then $(\mathsf{withSpeaker}\, s\, M_1)$ and $(\mathsf{withSpeaker}\, s\, M_2)$ are

[110] The semantics of quotations hides many more complexities [119] and whether quotations shift the interpretations of indexical expressions is under debate. We limit ourselves to using *said* as an example of how a lexical entry that shifts indexicals would look like in $(\!|\, \lambda \,|\!)$.

[111] For a formalization of the use of metavariables, see 3.4.1 or [76].

convertible in $(\!| \lambda |\!)$. The insight we get from these equations is that asking for the speaker is an idempotent operation: if we ask twice within the same computation, we are guaranteed to get the same answer. It also tells us that asking for the speaker has no other effect than to make available the identity of the speaker: we can add a request for the current speaker and if we do not use the answer, this addition will not change anything.

The motivation behind the choice of exactly these two equations is normalization. Denotations in $\mathcal{F}_{\{\texttt{speaker}:1 \rightarrowtail \iota\}}(\alpha)$ are formed by a series of `speaker` operations followed by a value of type α. We can use the first equation to collapse all of the `speaker` operations into one, or if there were no `speaker` operations we can use the second equation to include one. This means that we can see every value of type $\mathcal{F}_{\{\texttt{speaker}:1 \rightarrowtail \iota\}}(\alpha)$ as being equal to one which is of the shape $\texttt{speaker} \star (\lambda x.\, \eta\, (M(x)))$, i.e. a family which to every $x : \iota$ assigns an $M(x) : \alpha$. This connects us back to the treatment of deixis in Montague's approach [98] where meanings are functions which assign to every point of reference x some referent $M(x)$. This normalization is implemented by the withSpeaker handler. If we flip its arguments, getting $(\lambda M s.\, \mathsf{withSpeaker}\, s\, M)$, we get a function of type $\mathcal{F}_{\{\texttt{speaker}:1 \rightarrowtail \iota\}}(\alpha) \to (\iota \to \alpha)$.

6.3 Conventional Implicature

We have seen an example of an expression asking the context for some missing information. We will now look at a phenomenon which incurs communication in the opposite direction. *Conventional implicatures* [108] are parts of the entailed meaning which are not at-issue, i.e. are not being asserted, simply mentioned. One of the distinguishing signs is that they project out of logical contexts such as negation, disjunction or implication. Typical examples include supplements such as nominal appositives and supplementary relative clauses, and expressives such as epithets.

Here, we will deal with supplements, namely nominal appositives and supplementary relative clauses. We will assume abstract constants for the (supplementary) relative pronoun, the appositive construction and a relational noun[112] with the following types:

$$\begin{aligned} \textsc{who}_{\textsc{s}} &: (NP \multimap S) \multimap NP \multimap NP \\ \textsc{appos} &: NP \multimap NP \multimap NP \\ \textsc{best-friend} &: NP \multimap NP \end{aligned}$$

The point of our modeling is to show that the conventional implicatures engendered by the supplements project out of all sorts of logical contexts. We will therefore also consider a fragment that contains syntactic constructions for negation ("it is not the case that X"), implication ("if X, then Y") and disjunction ("either X, or Y").

$$\begin{aligned} \textsc{not-the-case} &: S \multimap S \\ \textsc{if-then} &: S \multimap S \multimap S \\ \textsc{either-or} &: S \multimap S \multimap S \end{aligned}$$

In our fragment, a noun phrase contributes both its referent and also possibly some conventional implicature. If we would model this fact by interpreting NPs as pairs of referents and conventional implicatures, we would be forced to revisit all the other lexical entries to take this change into account (e.g. transitive verbs such as LOVES would need to explicitly aggregate the conventional implicatures of both their subjects and objects). Rather than do that, we will make use of $(\!| \lambda |\!)$ and model this interaction with the context as an operation. When a linguistic expression wants to conventionally implicate something, it will use the `implicate` operation. The expression will need to communicate what exactly it wants to conventionally implicate. This will be a proposition and so the input type of `implicate` will be the type o of propositions. We do not need to collect any information from the context and so the output type

[112] We are working in a minimal fragment without determiners. Relational nouns will let us use some meaningful noun phrases as nominal appositives in the examples to come.

will be 1. Using this operation, we can now give denotations to expressions that generate conventional implicatures:

$$
\begin{aligned}
[\![\textsc{who}_{\mathrm{s}}]\!] &= \lambda CX.\ X \gg\!= (\lambda x. \\
&\qquad C\,(\eta\, x) \gg\!= (\lambda i. \\
&\qquad \texttt{implicate}\, i\, (\lambda\_. \\
&\qquad \eta\, x))) \\
[\![\textsc{appos}]\!] &= \lambda YX.\ X \gg\!= (\lambda x. \\
&\qquad Y \gg\!= (\lambda y. \\
&\qquad \texttt{implicate}\, (x = y)\, (\lambda\_. \\
&\qquad \eta\, x))) \\
[\![\textsc{best-friend}]\!] &= \lambda X.\, \textbf{best-friend} \cdot\!\gg X
\end{aligned}
$$

In both of the supplement constructions, we first evaluate the head NP X to get its referent x. We use x twice: once to construct the conventional implicature and once to produce the referent of the entire complex noun phrase. The conventional implicature constructed by the relative clause is the clause with its gap filled in by an expression that refers to the same referent as the head noun X. The conventional implicature of the appositive is a statement of equality between the referents of the two noun phrases. Finally, we also give the semantics to the relational noun "X's best friend" by assuming that we have a function **best-friend** $: \iota \to \iota$ in the model.

We will want to show that these denotations project through the operators that make up the logical structure of a sentence/discourse. Since the conventional implicature mechanism is implemented using operations, we can get this behavior for free by just using the standard operators and lifting them to computations.

$$
\begin{aligned}
[\![\textsc{not-the-case}]\!] &= \mathsf{lift}^{\mathsf{L}}_{S \multimap S}(\neg) \\
&= \lambda X.\, \neg \cdot\!\gg X \\
[\![\textsc{if-then}]\!] &= \mathsf{lift}^{\mathsf{L}}_{S \multimap S \multimap S}(\to) \\
&= \lambda XY.\, X \ll\!\!\to\!\!\gg Y \\
[\![\textsc{either-or}]\!] &= \mathsf{lift}^{\mathsf{L}}_{S \multimap S \multimap S}(\vee) \\
&= \lambda XY.\, X \ll\!\!\vee\!\!\gg Y
\end{aligned}
$$

We can now look at several examples of conventional implicatures buried inside logical operators:

(14) Either John loves Sarah, or Mary, John's best friend, loves John.

(15) If it is not the case that John, whom Sarah loves, loves Sarah then Mary loves John.

We expect Example 14 to conventionally implicate that Mary is John's best friend and Example 15 to conventionally implicate that Sarah loves John. If we compute their denotations, we find out that these actually are the propositions that the two sentences try to `implicate`.

$$
\begin{aligned}
&[\![\textsc{either-or}\ (\textsc{loves}\ \textsc{Sarah}\ \textsc{John})\ (\textsc{loves}\ \textsc{John}\ (\textsc{appos}\ (\textsc{best-friend}\ \textsc{John})\ \textsc{Mary}))]\!] \\
&\twoheadrightarrow \texttt{implicate}\, (\mathbf{m} = \textbf{best-friend}\, \mathbf{j})\, (\lambda\_.\, \eta\, (\textbf{love}\, \mathbf{j}\, \mathbf{s} \vee \textbf{love}\, \mathbf{m}\, \mathbf{j})) \qquad (14) \\
&[\![\textsc{if-then}\ (\textsc{not-the-case}\ (\textsc{loves}\ \textsc{Sarah}\ (\textsc{who}_{\mathrm{s}}\ (\lambda x.\ \textsc{loves}\ x\ \textsc{Sarah})\ \textsc{John})))\ (\textsc{loves}\ \textsc{John}\ \textsc{Mary})]\!] \\
&\twoheadrightarrow \texttt{implicate}\, (\textbf{love}\, \mathbf{s}\, \mathbf{j})\, (\lambda\_.\, \eta\, (\neg(\textbf{love}\, \mathbf{j}\, \mathbf{s}) \to \textbf{love}\, \mathbf{m}\, \mathbf{j})) \qquad (15)
\end{aligned}
$$

To go full circle and fulfill the empirical criterion that the meaning of Example 14 should entail that Mary is John's best friend, we will need to translate the term containing `implicate` operations into a proposition. This will be another question of interpreting operation symbols and so we will use a handler.

$$\begin{aligned}&\mathsf{withImplicatures} : \mathcal{F}_{\{\texttt{implicate}:o\rightarrow 1\}}(o) \rightarrow o\\&\mathsf{withImplicatures} = \llparenthesis\, \texttt{implicate}\colon (\lambda ik.\, i \wedge k\,\star)\,\rrparenthesis\end{aligned}$$

This handler applies only to computations that produce propositions. It collects all of the conventional implicatures and conjoins them with the (at-issue) proposition. We used a closed handler to make the presentation slightly simpler (no need to mention computation types inside the handler). In Chapter 8, we will introduce an open handler which does the same but works inside arbitrary effect signatures.

If we apply the withImplicatures handler to the denotations of (14) and (15), we can then verify that we get propositions that do entail the intended conventional implicatures.

$$\begin{aligned}&\mathsf{withImplicatures}\,[\![\textsc{either-or}\,(\textsc{loves}\,\textsc{Sarah}\,\textsc{John})\,(\textsc{loves}\,\textsc{John}\,(\textsc{appos}\,(\textsc{best-friend}\,\textsc{John})\,\textsc{Mary}))]\!]\\&\twoheadrightarrow (\mathbf{m} = \textbf{best-friend}\,\mathbf{j}) \wedge (\textbf{love}\,\mathbf{j}\,\mathbf{s} \vee \textbf{love}\,\mathbf{m}\,\mathbf{j})\\&\mathsf{withImplicatures}\,[\![\textsc{if-then}\,(\textsc{not-the-case}\,(\textsc{loves}\,\textsc{Sarah}\,\textsc{John}))\,(\textsc{loves}\,\textsc{John}\,\textsc{Mary})]\!]\\&\twoheadrightarrow (\textbf{love}\,\mathbf{s}\,\mathbf{j}) \wedge (\neg(\textbf{love}\,\mathbf{j}\,\mathbf{s}) \rightarrow \textbf{love}\,\mathbf{m}\,\mathbf{j})\end{aligned}$$

6.3.1 Algebraic Considerations

As in 6.2.2, we will look for equations that are admissible w.r.t. the withImplicatures handler (equations such that equivalent computations will be interpreted with the same value) with an eye towards deriving some canonical form.

$$\begin{aligned}\texttt{implicate}\,A\,(\lambda x.\,\texttt{implicate}\,B\,(\lambda y.\,M(x,y))) &= \texttt{implicate}\,(A \wedge B)\,(\lambda x.\,M(x,x))^{113}\\ M &= \texttt{implicate}\,\top\,(\lambda x.\,M)\end{aligned}$$

The equations are quite similar to those we have seen with deixis. Here, M ranges over computations of type $\mathcal{F}_{\{\texttt{implicate}:o\rightarrow 1\}}(\alpha)$. Since we know that our only handler, withImplicatures, ends up combining all the conventional implicatures using conjunction, we can collapse two conventional implicatures into a single conventional implicature. Furthermore, implicating the tautology $\top$ has no observable effect. By making an appeal to the conjunction operator, we can make use of the equations of propositions and derive equations for, e.g., the commutativity or idempotence of `implicate`:

$$\begin{aligned}\texttt{implicate}\,A\,(\lambda x.\,\texttt{implicate}\,B\,(\lambda y.\,M(x,y))) &= \texttt{implicate}\,(A \wedge B)\,(\lambda x.\,M(x,x))\\ &= \texttt{implicate}\,(B \wedge A)\,(\lambda x.\,M(x,x))\\ &= \texttt{implicate}\,B\,(\lambda x.\,\texttt{implicate}\,A\,(\lambda y.\,M(y,x)))\end{aligned}$$

A denotation in $\mathcal{F}_{\{\texttt{implicate}:o\rightarrow 1\}}(\alpha)$ is a sequence of conventionally-implicated propositions terminated by a value of type α. This is very much like the result of the *parsetree interpretation* in [108] (Definition 2.50), which is an $(n+1)$-tuple of the interpretation of some term of type α together with the propositions which are the interpretations of the n conventional implicatures embedded within the term.

Instead of constructing an n-tuple of the conventionally-implicated propositions, our handler conjoins all the propositions into a single proposition. This lets us admit the above equations, which tell us that a computation in $\mathcal{F}_{\{\texttt{implicate}:o\rightarrow 1\}}(\alpha)$ is always equivalent to one of the form $\texttt{implicate}\,p\,(\lambda x.\,\eta\,M)$, where p is a proposition and M is a value of type α. For the case of $\mathcal{F}_{\{\texttt{implicate}:o\rightarrow 1\}}(o)$ in particular, the domain in which we interpret sentences, we have that the denotation is a pair of propositions: the conventionally-implicated (CI) content and the at-issue content.

As in 6.2.2, we can construct a handler which maps a computation to this canonical representation:

[113] The output type of `implicate` is the unit type 1 whose only value is $\star$. We can therefore replace y with x since they are guaranteed to be equal. For simplicity, we could also assume that M is fresh for x and y.

$$\begin{aligned}&\text{accommodate}' : \mathcal{F}_{\{\texttt{implicate}:o \rightarrowtail 1\}}(\alpha) \to (o \times \alpha)\\ &\text{accommodate}' = (\!|\texttt{implicate}\colon (\lambda ik.\ \langle i \wedge \pi_1\,(k\,\star), \pi_2\,(k\,\star)\rangle),\ \eta\colon (\lambda x.\ \langle \top, x\rangle)|\!)\end{aligned}$$

6.4 Quantification

Next, we turn our attention to in-situ quantification. This is the phenomenon of quantified noun phrases such as *every man* or *a woman* acting as quantifiers over the sentence in which they appear. One of the mechanisms which is used to model the inversion from being an argument of a verb inside a sentence to taking the verb and the whole sentence as an argument are continuations [36, 12].

We will be working with the following abstract syntax:

$$\begin{aligned}\textsc{every}, \textsc{a} &: N \multimap NP\\ \textsc{man}, \textsc{woman} &: N\end{aligned}$$

The challenge will be how to fit in the denotations of *every man* or *some woman* into the type $[\![NP]\!]_c = \mathcal{F}_E(\iota)$. We cannot find a unique referent for either of these noun phrases and we need to take into account their quantificational effect on the meaning of the whole sentence. We will make use of the operations and handlers present in $(\!|\lambda|\!)$, as we have done in all the previous section of this chapter. We will use an operation symbol `scope` of type $((\iota \to o) \to o) \rightarrowtail \iota$, for which we can give two different motivations:

- We know that continuations are a useful technique for dealing with quantification and in Chapter 4, we have seen how to encode continuations in $(\!|\lambda|\!)$. The type that we have given to the `shift0` operator there was $((\delta \to \mathcal{F}_E(\omega)) \to \mathcal{F}_E(\omega)) \rightarrowtail \delta$. We can specialize this to our case. The type ω of observations will be the type o of propositions, as it will be over propositions that our quantifiers will scope. The type δ of expressions at which we will shift will be the type ι of individuals as that is the type of referents for noun phrases. Finally, in this section we will be dealing only with quantification and no other effects and therefore the signature E will be empty and so we can also identify $\mathcal{F}_\emptyset(\omega)$ with ω. This leaves us with the type $((\iota \to o) \to o) \rightarrowtail \iota$.
- Like in the previous sections, we can also figure out the input and output types of `scope` by considering what information the quantified noun phrase can offer to its context and what it expects in return. As Montague showed, we can model the meanings of quantified noun phrases as generalized quantifiers [98]: e.g. *every man* becomes $\lambda P.\, \forall x.\, \mathbf{man}\, x \to P\, x$ of type $(\iota \to o) \to o$. We also want quantified noun phrases to be like other noun phrases in that they should behave as if their referents were individuals of type ι. Inside the denotation, we therefore have a generalized quantifier of type $(\iota \to o) \to o$ and we would like to trade it for an individual of type ι, leading us to the type for `scope` : $((\iota \to o) \to o) \rightarrowtail \iota$. We give a generalized quantifier to the context and the context will have that quantifier scope over the sentence. In return, we get the variable of type ι that is being quantified over and that stands for the referent of the noun phrase.

With this `scope` operation, we can now give a semantics to our determiners:

$$\begin{aligned}[\![\textsc{every}]\!] &= \lambda N.\, \texttt{scope!}\,(\lambda k.\, \forall x.\, (\mathsf{SI}\,(N \ll\cdot\, x)) \to k\, x)\\ [\![\textsc{a}]\!] &= \lambda N.\, \texttt{scope!}\,(\lambda k.\, \exists x.\, (\mathsf{SI}\,(N \ll\cdot\, x)) \wedge k\, x)\\ [\![\textsc{man}]\!] &= \eta\, \mathbf{man}\\ [\![\textsc{woman}]\!] &= \eta\, \mathbf{woman}\\ \mathsf{SI} &= (\!|\texttt{scope}\colon (\lambda ck.\, c\, k)|\!)\end{aligned}$$

The denotations of EVERY and A are both making use of the new `scope` operation. If we look at the meaning of $[\![\textsc{every man}]\!] = \texttt{scope}\,(\lambda k.\, \forall x.\, \mathbf{man}\, x \to k\, x)\,(\lambda x.\, \eta\, x)$, we see that it is composed of two parts:

some logical material $\forall x.\, \mathbf{man}\, x \to k\, x$ that is to scope over the enclosing context k and a placeholder NP denotation $\eta\, x$ for some variable x. This is very much like Cooper storage [32]. We store the request to scope certain material over the sentence and we keep a variable as a placeholder denotation for the NP. Before we proceed to explain this fragment any further, we will consider the following example:

(16) Every man loves a woman.

and its denotation:

$$\begin{aligned}
&[\![\textsc{loves}\,(\textsc{a woman})\,(\textsc{every man})]\!] \\
\twoheadrightarrow\ &\texttt{scope}\,(\lambda k.\, \forall x.\, \mathbf{man}\, x \to k\, x)\,(\lambda x. \\
&\texttt{scope}\,(\lambda k.\, \exists y.\, \mathbf{woman}\, y \wedge k\, y)\,(\lambda y. \\
&\eta\,(\mathbf{love}\, x\, y)))
\end{aligned} \tag{16}$$

We have $\eta\,(\mathbf{love}\, x\, y)$ as the core meaning, with both $(\lambda k.\, \forall x.\, \mathbf{man}\, x \to k\, x)$ and $(\lambda k.\, \exists y.\, \mathbf{woman}\, y \wedge k\, y)$ scheduled to scope over it. We see that as in Cooper's approach, the scope material that we stored using the NPs gets propagated to the meaning of the entire sentence. Now we need an analogue to Cooper's retrieval procedure that can take this scope material and apply it to the meaning of the nucleus. This role is carried out by the handler SI, short for Scope Island.

$$\begin{aligned}
&\mathsf{SI}\,[\![\textsc{loves}\,(\textsc{a woman})\,(\textsc{every man})]\!] \\
&\twoheadrightarrow \forall x.\, \mathbf{man}\, x \to (\exists y.\, \mathbf{woman}\, y \wedge \mathbf{love}\, x\, y)
\end{aligned}$$

The handler SI will also end up being part of our denotations. Quantifiers can only take scope up to the limit of the nearest enclosing scope island and that is a constraint that we can encode using this handler. We could, for example, implement the constraint that tensed clauses should form scope islands by including the SI handler in the denotations of the constructors of tensed clauses.

$$[\![\textsc{loves}]\!] = \lambda OS.\, (\eta \circ \mathsf{SI})\,(\mathbf{love} \cdot\!\gg S \ll\!\cdot\!\gg O)$$

Here, we apply not only SI, but $\eta \circ \mathsf{SI}$, to the sentence denotation. This is because our type for interpreting sentences, $[\![S]\!]$, is $\mathcal{F}_E(o)$ and since in this section, we are using a closed handler (type $\mathcal{F}_E(o) \to o$), we have to follow with the injection η (type $o \to \mathcal{F}_E(o)$).[114]

Using this new lexical entry for LOVES, we can directly compute a reading for Example 16:

$$\begin{aligned}
&[\![\textsc{loves}\,(\textsc{a woman})\,(\textsc{every man})]\!] \\
&\twoheadrightarrow \eta\,(\forall x.\, \mathbf{man}\, x \to (\exists y.\, \mathbf{woman}\, y \wedge \mathbf{love}\, x\, y))
\end{aligned} \tag{16}$$

Finally, we will address the use of SI in the denotations of EVERY and A. We will note that we interpret constituents of type N with the type $\mathcal{F}_E(\iota \to o)$. However, in the fragment so far, we have only seen pure nouns, nouns whose denotation is of the form $\eta\, M$. Nevertheless, we can imagine complex constituents of type N which do have a quantificational effect (e.g. *owner of a cat*).

$$\begin{aligned}
\textsc{owner-of} &: NP \multimap N \\
[\![\textsc{owner-of}]\!] &= \mathsf{lift}^{\mathsf{L}}_{NP \multimap N}(\lambda yx.\, \mathbf{own}\, x\, y) \\
&= \lambda Y.\, Y \gg\!= (\lambda y.\, \eta\,(\lambda x.\, \mathbf{own}\, x\, y))
\end{aligned}$$

[114] We could also just as well choose $[\![S]\!] = o$ but we want to be consistent with the other treatments we have shown so far and the ones we will see later.

The relational noun *owner* does not contribute any quantificational effect, but the complex noun *owner of Y* inherits any effects of Y. We will see this in the denotation of *owner of a cat*:

$$
\begin{aligned}
&[\![\textsc{owner-of}\ (\textsc{a cat})]\!] \\
&\twoheadrightarrow [\![\textsc{owner-of}]\!]\,(\mathtt{scope}\,(\lambda k.\,\exists y.\,\textbf{cat}\,y \wedge k\,y)\,(\lambda y.\,\eta\,y)) \\
&\twoheadrightarrow (\lambda yx.\,\textbf{own}\,x\,y) \cdot\!\gg (\mathtt{scope}\,(\lambda k.\,\exists y.\,\textbf{cat}\,y \wedge k\,y)\,(\lambda y.\,\eta\,y)) \\
&\twoheadrightarrow \mathtt{scope}\,(\lambda k.\,\exists y.\,\textbf{cat}\,y \wedge k\,y)\,(\lambda y. \\
&\quad \eta\,((\lambda yx.\,\textbf{own}\,x\,y)\,y)) \\
&\to_\beta \mathtt{scope}\,(\lambda k.\,\exists y.\,\textbf{cat}\,y \wedge k\,y)\,(\lambda y. \\
&\quad\ \eta\,(\lambda x.\,\textbf{own}\,x\,y))
\end{aligned}
$$

We can look at the result through the analogy to Cooper storage. We started with the meaning of *a cat*, which stored the generalized quantifier $\lambda k.\,\exists y.\,\textbf{cat}\,y \wedge k\,y$ in the storage and whose semantic placeholder was the variable y. We then applied the function $\lambda yx.\,\textbf{own}\,x\,y$ to this semantic placeholder and we arrived at $\lambda x.\,\textbf{own}\,x\,y$ with the storage still holding the quantifier $\lambda k.\,\exists y.\,\textbf{cat}\,y \wedge k\,y$.

And so we have seen that constituents of type N can have a quantificational effect in much the same way as other constituents, such as those of type NP. In the generalized quantifier for indefinites, $(\lambda k.\,\exists y.\,n\,y \wedge k\,y)$, we make use of the meaning n of the restrictor noun. By looking at the type of $\mathtt{scope} : ((\iota \to o) \to o) \rightarrowtail \iota$, we see that we cannot pass a computation with quantificational effects as an argument to `scope`.[115] So instead, we discharge the quantificational potential of the noun in the restrictor of the generalized quantifier using the SI handler.[116] We will look at the kind of readings this leads to by considering the classic example from [23]:

(17) Every owner of a siamese cat loves a therapist.

Assuming the inclusion of the common nouns *siamese cat* and *therapist* in our fragment (with semantics analogous to the ones for *man* and *woman*), we can compute the following interpretation:

$$
\begin{aligned}
&[\![\textsc{loves}\ (\textsc{a therapist})\ (\textsc{every}\ (\textsc{owner-of}\ (\textsc{a siamese-cat})))]\!] \\
&\twoheadrightarrow \eta\,(\forall x.\,(\exists y.\,\textbf{siamese-cat}\,y \wedge \textbf{own}\,x\,y) \to (\exists z.\,\textbf{therapist}\,z \wedge \textbf{love}\,x\,z))
\end{aligned} \tag{17}
$$

6.4.1 Quantifier Ambiguity

The lexical entry that we have considered for the determiner *every*[117] so far was:

$$[\![\textsc{every}]\!] = \lambda N.\,\mathtt{scope!}\,(\lambda k.\,\forall x.\,(\mathsf{SI}\,(N \ll\!\cdot\, x)) \to k\,x)$$

However, had we followed the analogy to `shift` and `reset` (i.e. wrapping the entire body of the function that is the argument to `scope/shift0` in `SI/reset`, see Section 4.5), we might have tried the following:

$$[\![\textsc{every}']\!] = \lambda N.\,\mathtt{scope!}\,(\lambda k.\,\mathsf{SI}\,(N \gg\!= (\lambda n.\,\eta\,(\forall x.\,n\,x \to k\,x))))$$

Here, we put the SI handler above the quantifier introduced by the semantics of the determiner. For reasons of readability, we have also pulled the only impure part, N, in front of the expression using the $\gg\!=$ operator. This lexical entry puts the scope of any quantifiers found in the restrictor noun over the

[115] In that case, we would need to have $\mathtt{scope} : ((\iota \to \mathcal{F}_E(o)) \to o) \rightarrowtail \iota \in E$. This is problematic since E, which contains `scope`, is then recursive.

[116] This is very much like the situation we have seen towards the end of Chapter 4. The SI handler is our `reset`. We can only pass the type check if we use `shift/reset` (Section 4.6). We encode `shift` as `shift0` by using a `reset` (in our case SI) inside the argument to `shift0` (in our case `scope`), as in Section 4.5.

[117] Though the same applies to the lexical entry for the indefinite article *a* as well.

scope of the quantifier contribued by the determiner. If we try using this lexical entry when computing the meaning of Example 17, we find another possible reading:

$$\begin{aligned}&[\![\textsc{loves}\ (\textsc{a therapist})\ (\textsc{every}'\ (\textsc{owner-of}\ (\textsc{a siamese-cat})))]\!]\\ &\twoheadrightarrow \eta\,(\exists y.\ \textbf{siamese-cat}\,y \wedge (\forall x.\ \textbf{own}\,x\,y \rightarrow (\exists z.\ \textbf{therapist}\,z \wedge \textbf{love}\,x\,z)))\end{aligned} \tag{17}$$

Both of the readings we have seen are actually valid interpretations of this sentence; quantifiers are a notorious source of ambiguities. We could include both EVERY and EVERY' in our grammar, one giving narrow scope to resrictor quantifiers and the other giving them wide scope (the same would be the case for other determiners such as A). This way, both readings would be available.

Note that Example 16 is ambiguous as well. The existential quantifier in *a woman* can take either wide or narrow scope (we call it wide scope when the object quantifier scopes over the subject quantifier and we call it narrow when it scopes under). The relative scope of the quantifiers is given by the order in which they appear in the computation: to get the object in wide scope, we would need to chain the computations of the subject and object by putting object first, subject last.

$$\begin{aligned}[\![\textsc{loves}]\!] &= \lambda OS.\,(\eta \circ \mathsf{SI})\,(\textbf{love} \cdot\!\gg S \ll\!\cdot\!\gg O)\\ &= \lambda OS.\,(\eta \circ \mathsf{SI})\,(S \gg\!= (\lambda s.\,O \gg\!= (\lambda o.\,\eta\,(\textbf{love}\,s\,o))))\\ [\![\textsc{loves}']\!] &= \lambda OS.\,(\eta \circ \mathsf{SI})\,(O \gg\!= (\lambda o.\,S \gg\!= (\lambda s.\,\eta\,(\textbf{love}\,s\,o))))\end{aligned}$$

Using the lexical entry, we can now derive the other reading for Example 16:

$$\begin{aligned}&[\![\textsc{loves}'\ (\textsc{a woman})\ (\textsc{every man})]\!]\\ &\twoheadrightarrow \eta\,(\exists y.\ \textbf{woman}\,y \wedge (\forall x.\ \textbf{man}\,x \rightarrow \textbf{love}\,x\,y))\end{aligned} \tag{16}$$

as well as two more readings for Example 17:

$$\begin{aligned}&[\![\textsc{loves}'\ (\textsc{a therapist})\ (\textsc{every}\ (\textsc{owner-of}\ (\textsc{a siamese-cat})))]\!]\\ &\twoheadrightarrow \eta\,(\exists z.\ \textbf{therapist}\,z \wedge (\forall x.\ (\exists y.\ \textbf{siamese-cat}\,y \wedge \textbf{own}\,x\,y) \rightarrow \textbf{love}\,x\,z))\end{aligned} \tag{17}$$

$$\begin{aligned}&[\![\textsc{loves}'\ (\textsc{a therapist})\ (\textsc{every}'\ (\textsc{owner-of}\ (\textsc{a siamese-cat})))]\!]\\ &\twoheadrightarrow \eta\,(\exists z.\ \textbf{therapist}\,z \wedge (\exists y.\ \textbf{siamese-cat}\,y \wedge (\forall x.\ \textbf{own}\,x\,y \rightarrow \textbf{love}\,x\,z)))\end{aligned} \tag{17}$$

However, Example 17 has one more reading that we will not be able to get at using this technique.

$$\exists y.\ \textbf{siamese-cat}\,y \wedge (\exists z.\ \textbf{therapist}\,z \wedge (\forall x.\ \textbf{own}\,x\,y \rightarrow \textbf{love}\,x\,z))$$

This reading is equivalent to the last reading, $[\![\textsc{loves}'\ (\textsc{a therapist})\ (\textsc{every}'\ (\textsc{owner-of}\ (\textsc{a siamese-cat})))]\!]$. The only difference is that this time, *a siamese cat* has scope over *a therapist*. This distinction would be important in sentences involving a different quantifier then the existential coming from the indefinite, such as in the sentence "*Every researcher of a company saw most samples*" [23]. In our fragment, we cannot reproduce this reading because the order in which the quantifiers take place goes like this: *a siamese cat* from the subject, *a therapist* from the object and *every owner* from the subject. However, we have no simple way to crack open the chain of quantifications from the subject and insert the quantification from the object in the middle. We will therefore turn to a more robust and elegant solution which will get us the readings we want and will not profilerate duplicate entries such as EVERY/EVERY' and LOVES/LOVES'.

Quantifier Raising

We will make use of the fact that ACGs are a categorial grammar formalism and their syntactic structures are not just constituency trees, but derivations which can use hypothetical reasoning. That is to say, the

terms in the abstract signature which represent the tectogrammatic structure can contain λ-binders and variables. We will use this to implement Montague's treatment of quantifiers [98], called "quantifying in" or "quantifier raising".[118]

Montague's technique can be explained by saying that sentences such as "*every man loves a woman*" can be paraphrased as "*a woman, every man loves her*". The quantifier that is to be raised is replaced by a pronoun/variable. Later, we use a construction that lets us form a sentence by combining a noun phrase with a sentence in which a variable is to be bound. We can licence this kind of construction in our ACG by adding the following (unlexicalised) abstract constant:

$$\begin{aligned} \mathrm{QR} &: NP \multimap (NP \multimap S) \multimap S \\ [\![\mathrm{QR}]\!] &= \lambda X K.\, X \gg\!= (\lambda x.\, K\,(\eta\, x)) \end{aligned}$$

The type of QR can also be read as a kind of type-lifting operator on NPs: it takes an NP and gives back an $(NP \multimap S) \multimap S$ (the syntactic type for quantified noun phrases used in [105]). As for the semantics, we can regard it as a variation on the term $\lambda X K.\, K\, X$: instead of passing X directly to K though, we first evaluate it to get its referent x and then we pass to K a trivial computation that always this referent x.

We will demonstrate QR by deriving the object wide scope reading of Example 16:

$$\begin{aligned} &\mathsf{SI}\,[\![(\mathrm{QR}\,(\textsc{a woman})\,(\lambda y.\,\textsc{loves}\,y\,(\textsc{every man})))]\!] \\ &\twoheadrightarrow \mathsf{SI}\,(\mathtt{scope}\,(\lambda k.\,\exists y.\,\mathbf{woman}\,y \wedge k\,y)\,(\lambda y.\,\forall x.\,\eta\,(\mathbf{man}\,x \rightarrow \mathbf{love}\,x\,y))) \\ &\twoheadrightarrow (\lambda k.\,\exists y.\,\mathbf{woman}\,y \wedge k\,y)\,(\lambda y.\,\forall x.\,\mathbf{man}\,x \rightarrow \mathbf{love}\,x\,y) \\ &\twoheadrightarrow \exists y.\,\mathbf{woman}\,y \wedge (\forall x.\,\mathbf{man}\,x \rightarrow \mathbf{love}\,x\,y) \end{aligned} \tag{16}$$

We can also generate all five readings of Example 17, without incurring the overgeneration due to Cooper's storage [23] thanks to the ACG type system.[119]

$$\begin{aligned} &[\![\textsc{loves}\,(\textsc{a therapist})\,(\textsc{every}\,(\textsc{owner-of}\,(\textsc{a siamese-cat})))]\!] \\ &\twoheadrightarrow \eta\,(\forall x.\,(\exists y.\,\mathbf{siamese\text{-}cat}\,y \wedge \mathbf{own}\,x\,y) \rightarrow (\exists z.\,\mathbf{therapist}\,z \wedge \mathbf{love}\,x\,z)) && (17) \\ &\mathsf{SI}\,[\![Q R\,(\textsc{a siamese-cat})\,(\lambda y.\,(\textsc{loves}\,(\textsc{a therapist})\,(\textsc{every}\,(\textsc{owner-of}\,y))))]\!] \\ &\twoheadrightarrow \exists y.\,\mathbf{siamese\text{-}cat}\,y \wedge (\forall x.\,\mathbf{own}\,x\,y \rightarrow (\exists z.\,\mathbf{therapist}\,z \wedge \mathbf{love}\,x\,z)) && (17) \\ &\mathsf{SI}\,[\![Q R\,(\textsc{a therapist})\,(\lambda z.\,(\textsc{loves}\,z\,(\textsc{every}\,(\textsc{owner-of}\,(\textsc{a siamese-cat})))))]\!] \\ &\twoheadrightarrow \exists z.\,\mathbf{therapist}\,z \wedge (\forall x.\,(\exists y.\,\mathbf{siamese\text{-}cat}\,y \wedge \mathbf{own}\,x\,y) \rightarrow \mathbf{love}\,x\,z) && (17) \\ &\mathsf{SI}\,[\![Q R\,(\textsc{a siamese-cat})\,(\lambda y.\,(Q R\,(\textsc{a therapist})\,(\lambda z.\,(\textsc{loves}\,z\,(\textsc{every}\,(\textsc{owner-of}\,y))))))]\!] \\ &\twoheadrightarrow \exists y.\,\mathbf{siamese\text{-}cat}\,y \wedge (\exists z.\,\mathbf{therapist}\,z \wedge (\forall x.\,\mathbf{own}\,x\,y \rightarrow \mathbf{love}\,x\,z)) && (17) \\ &\mathsf{SI}\,[\![Q R\,(\textsc{a therapist})\,(\lambda z.\,(Q R\,(\textsc{a siamese-cat})\,(\lambda y.\,(\textsc{loves}\,z\,(\textsc{every}\,(\textsc{owner-of}\,y))))))]\!] \\ &\twoheadrightarrow \exists z.\,\mathbf{therapist}\,z \wedge (\exists y.\,\mathbf{siamese\text{-}cat}\,y \wedge (\forall x.\,\mathbf{own}\,x\,y \rightarrow \mathbf{love}\,x\,z)) && (17) \end{aligned}$$

The QR operator allows us to displace the scope of any quantifier. [120] We might now risk overgenerating by letting quantifiers leak outside of scope islands. However, this can be remedied at the level of the abstract type signature. Pogodalla and Pompigne [106] show how to use dependent types in the abstract signature of an ACG to enforce a scope island constraint — namely that quantified noun phrases should not take scope outside of the nearest enclosing tensed clause.

[118] We saw quantifier raising in Section 5.1. It is the rule $F_{10,n}$ from clause S14, or rather its semantic counter part $G_{10,n}$ in T14.

[119] The overgenerating case is due to a variable escaping its scope. However, such a term with a free occurrence of a variable would not be a well-typed closed term and so such a derivation does not exist in our grammar.

[120] Actually, it can displace any kind of NP effect, including dynamics, and so it can be used, e.g., to implement a kind of cataphora. This will turn out to be a bug, not a feature, because of crossover constraints [120], to be treated in Subsection 8.5.1.

6.4.2 Algebraic Considerations

Unlike deixis and conventional implicature, we will not derive many useful admissible equations for scope. If we try to collapse two uses of scope into a single one, we can succeed only partially:

$$\texttt{scope}\, F_1\, (\lambda x.\, \texttt{scope}\, F_2\, (\lambda y.\, M(x,y))) = \texttt{scope}\, (\lambda k.\, F_1\, (\lambda x.\, F_2\, (\lambda y.\, k\ \langle x,y\rangle)))\, (\lambda\, \langle x,y\rangle\, .\, M(x,y))$$

We can compose the two quantifiers F_1 and F_2, but then we are quantifying over pairs of individuals $\langle x,y\rangle$, which is not something that we have planned to do with scope, whose output type is ι, the type of (single) individuals. Therefore, the above would not even type-check correctly. However, we can derive an equation which shows us how pure computations that yield individuals correspond to computations using scope.

$$\eta\, M = \texttt{scope!}\, (\lambda k.\, k\, M)$$

M ranges over values of type ι. This equation shows us why, in $(\!|\, \lambda \,|\!)$, we are not obliged to raise the denotations of our non-quantified noun phrases (such as the proper names John and Mary) into generalized quantifiers: the pure individuals $\eta\, \mathbf{j}$ and $\eta\, \mathbf{m}$ behave exactly the same as the generalized quantifiers $\lambda k.\, k\, \mathbf{j}$ and $\lambda k.\, k\, \mathbf{m}$, respectively. This means that a sentence can mix the lexical entries for proper nouns from 6.1 with the new lexical entries for quantified noun phrases from this section in a sound way, without violating the homomorphism property of the ACG.

(18) John loves a man.

$$\begin{aligned} &[\![\textsc{loves}\ (\textsc{a man})\ \textsc{John}]\!] \\ &\twoheadrightarrow \eta\, (\exists x.\, \mathbf{man}\, x \wedge \mathbf{love}\, \mathbf{j}\, x) \end{aligned} \tag{18}$$

Since we do not have any useful laws to simplify the denotations in $\mathcal{F}_{\{\texttt{scope}:((\iota\to o)\to o)\rightarrowtail\iota\}}(\alpha)$, the canonical representations will be a hierarchy of generalized quantifiers, one scoping over the other, with a value of type α at the bottom. Previously, we have drawn an analogy to Cooper storage. However, in Cooper storage, the quantifiers are not stored hierarchically, but side-by-side, so that any quantifier can be retrieved. However, this can lead to generating undesired meanings in which variables escape from the scopes of their intended binders. Our approach is closer to *Keller storage* [66], also known as *nested Cooper storage*. In Keller storage, quantifiers can be stored both side-by-side and embedded: any of the quantifiers stored side-by-side can be retrieved, but whenever a quantifier is retrieved, all of its embedded quantifiers are retrieved at the same time. Our representation lacks the side-by-side mode of composition. The chain of scope operations in an $\mathcal{F}_{\{\texttt{scope}:((\iota\to o)\to o)\rightarrowtail\iota\}}(\alpha)$ denotation corresponds to a series of embedded quantifiers in Keller storage: we cannot retrieve a quantifier without first retrieving the quantifiers which precede it in the chain. This is exemplified by this chain from Example 17 where the y in the second quantifier is bound by the first quantifier:

$$\begin{aligned} &\texttt{scope}\, (\lambda k.\, \exists y.\, \mathbf{cat}\, y \wedge k\, y)\, (\lambda y. \\ &\texttt{scope}\, (\lambda k.\, \forall x.\, \mathbf{own}\, x\, y \to k\, x)\, (\lambda x. \\ &\eta\, x)) \end{aligned}$$

This means that our representation behaves like Keller storage in that it prevents scope extrusion (variables escaping out of the scope of their intended binders). However, it lacks the basic feature of Cooper (and Keller) storage that is the side-by-side storing of quantifiers to enable ambiguity. In 6.4.1, we have dealt with the ambiguity issue by use of Montague's "quantifying in".

6.5 Methodology

We will now take a step back and identify the methodology we have used to analyse these three phenomena: deixis, conventional implicature and quantification.

6.5.1 Using Computation Types

At the basis of any linguistic modelling that uses $(\!| \lambda |\!)$ is the notion of a computation: all of the extensions to the simply-typed λ-calculus in $(\!| \lambda |\!)$ deal with computation types, types of the form $\mathcal{F}_E(\alpha)$. We will want to use computations in the semantic interpretations of our lexical items.

If we start from some existing Montagovian semantics (interpreting sentences as propositions, noun phrases as individuals, nouns as sets), there is a question of where to introduce the computation types. In our approach, we choose to make the interpretation of every atomic abstract type a computation. This has roughly the effect of making every constituent a computation. Kiselyov' Applicative Abstract Categorial Grammars [68, 69] use exactly the same strategy, and Barker's continuization approach [12], which replaces the NP interpretation type ι with a "computation type" $(\iota \to o) \to o$, uses a similar approach. Other possible strategies include:

- Turning every atomic semantic type (ι, o, ...) into a computation

  This idea is well-established in formal semantics. Examples include de Groote's montagovian treatment of anaphora [38] (o is replaced with $\gamma \to (\gamma \to o) \to o$), Lebedeva's extension of this formalism [80] (furthermore replaces ι with $\gamma \to \iota$), Ben-Avi and Winter's [18] intensionalization procedure (replaces o with $\sigma \to o$) and de Groote and Kanazawa's variation [40] (also replaces ι with $\sigma \to \iota$).

  We can contrast this approach to ours. Let $[\![-]\!]$ be some interpretation of the atomic abstract types. Our strategy leads us to the interpretation $[\![-]\!]_{\mathrm{c}}$ with:

$$
\begin{aligned}
[\![\alpha \multimap \beta]\!]_{\mathrm{c}} &= [\![\alpha]\!]_{\mathrm{c}} \to [\![\beta]\!]_{\mathrm{c}} \\
[\![\nu]\!]_{\mathrm{c}} &= \mathcal{F}_E([\![\nu]\!])
\end{aligned}
$$

  $[\![\alpha \multimap \beta]\!]_{\mathrm{c}}$ is the homomorphic interpretation of an ACG type. The interpretations of the atomic abstract types are wrapped in the functor $\mathcal{F}_E$.

  With the intensionalization procedure of de Groote and Kanazawa [40], we have:

$$
\begin{aligned}
[\![\alpha \multimap \beta]\!]_{\mathrm{i}} &= [\![\alpha]\!]_{\mathrm{i}} \to [\![\beta]\!]_{\mathrm{i}} \\
[\![\nu]\!]_{\mathrm{i}} &= \overline{[\![\nu]\!]} \\
\overline{\alpha \to \beta} &= \overline{\alpha} \to \overline{\beta} \\
\overline{\nu} &= \sigma \to \nu
\end{aligned}
$$

  Again, $[\![-]\!]_{\mathrm{i}}$ is an ACG type homomorphism. The interpretations of the atomic types are passed through another type homomorphism, which traverses the structure of the object (semantic) type. The atomic semantic types are then wrapped in the functor $F(\alpha) = \sigma \to \alpha$, where σ is the type of possible worlds.

- Turning every semantic function type $\alpha \to \beta$ into the type $\alpha \to \mathcal{F}_E(\beta)$

  This corresponds to interpreting a call-by-value language using the monad $\mathcal{F}_E$ [95, 133]. Techniques that use a call-by-value impure language for their semantic entries fall in this category as well. Examples include Shan's use of `shift` and `reset` [114, 115] and our previous attempts [89] using the Eff language [16].

- Turning every semantic function type $\alpha \to \beta$ into the type $\mathcal{F}_E(\alpha) \to \mathcal{F}_E(\beta)$

  This is very similar to the idea above but instead of call-by-value, it lets us get a call-by-name interpretation. Call-by-name side effects in natural language semantics have been proposed by Kiselyov [67].

This palette of strategies ranges the possibility space, offering a tradeoff between flexibility and simplicity. Inserting computations into more and more places gives us more expressivity but this comes at the price of the system's simplicity:

- A noun denotation of type $\mathcal{F}_E(\iota \to o)$ has some effect and then yields a pure predicate of type $\iota \to o$.
- A noun denotation of type $\iota \to \mathcal{F}_E(o)$ might have different effects, depending on the semantic argument that it will be applied to. As a consequence, this effect becomes available only when we apply the denotation to some argument of type ι.[121]
- A noun denotation of type $\mathcal{F}_E(\iota) \to \mathcal{F}_E(o)$ might evaluate its argument first, or it might perform some other effects, or it might handle some of the operations used in the computation of its argument.

Furthermore, in a calculus such as $(\!| \lambda |\!)$, where the order of evaluation is controlled manually using monadic combinators, having an overabundance of computation types clouds the terms with uninteresting plumbing and enlarges the possibility space to a point that facilitates ad hoc solutions. In the analyses presented in this manuscript, we have found that the simplest strategy which is sufficiently expressive for our purposes is the one which introduces a computation type into the interpretation of every atomic abstract type[122] and so we stick with this strategy throughout the whole manuscript. However, the techniques developed here can be also used in the other settings.

Choosing the Return Types and Seeding a Grammar

We can now take some base grammar that will serve as our starting point in investigating some phenomenon. In our case, we consider one of the smallest fragments imaginable: transitive verbs and names. We presuppose a semantics for this fragment that uses computations. We either build it from scratch or we lift an existing semantics as we did in 6.1. Our base semantics is very simple, it does not use generalized quantifiers or dynamic logics, since we can treat these phenomena using effects. Again, we use the simplest possible types for the return types of the computations. For example for noun phrases, we do not use generalized quantifiers, since we can do quantification as an effect, we use computations that yield individuals ($\mathcal{F}_E(\iota)$ instead of $\mathcal{F}_E((\iota \to o) \to o)$). We cannot go simpler than that since verbs still need to know what the referents of their subjects and objects are, to what their predicates should be applied. Similarly for sentences, we will use simple propositions instead of dynamic propositions since we will treat dynamicity as an effect ($\mathcal{F}_E(o)$ instead of $\mathcal{F}_E(\gamma \to (\gamma \to o) \to o)$). Again, it might seem that a proposition is the bare minimum that a sentence must denote because that is the product that we are interested in. However, in analyses of dynamic semantics, it might make sense to model sentences as having no referent, only side effects that contribute to some knowledge base (see 7.2.4).

6.5.2 Digression: Call-by-Name and Call-by-Value

We have mentioned the possible use of call-by-name and call-by-value. Our approach is more conservative. We do not assume that a single strategy will always suffice and we wire up the evaluation order manually using operators like $\gg\!=$. Nevertheless, we can still find some regularity in how we treat evaluation order. Syntactic functions (e.g. movement) are call-by-name and semantic functions (e.g. predicates) are call-by-value. By syntactic function, we mean a function of type $\alpha \multimap \beta$ at the abstract level which was mapped by the homomorphism $[\![-]\!]$ to the object-level function $\alpha \to \beta$. A semantic function is any other function present at the object (semantic) level, e.g. the predicate **love** : $\iota \to \iota \to o$.

We will demonstrate this on Example 16: *every man loves a woman*.

[121] We could use the $\mathcal{C}$ operator of $(\!| \lambda |\!)$ to get a value of the above type $\mathcal{F}_E(\iota \to o)$. However, this partial operation succeeds only if the effects do not actually depend on the argument. If that is the case, we might as well use $\mathcal{F}_E(\iota \to o)$ directly.

[122] With some caveats when dealing with restrictive relative clauses and presuppositions (see 8.6)

$$
\begin{aligned}
&[\![\textsc{loves}\,(\textsc{a woman})\,(\textsc{every man})]\!] \\
&= (\lambda OS.\,(\eta \circ \mathsf{SI})\,(\mathbf{love} \cdot\!\gg S \ll\!\cdot\!\gg O))\,[\![\textsc{a woman}]\!]\,[\![\textsc{every man}]\!] \\
&\twoheadrightarrow \eta\,(\mathsf{SI}\,(\mathbf{love} \cdot\!\gg [\![\textsc{every man}]\!] \ll\!\cdot\!\gg [\![\textsc{a woman}]\!]))
\end{aligned}
$$

We pass the computations $[\![\textsc{a woman}]\!]$ and $[\![\textsc{every man}]\!]$ to the lexical entry for the transitive verb $\textsc{loves} : NP \multimap NP \multimap S$ intact, without forcing their evaluation using $\gg\!=$ or other operators. Once inside the transitive verb, we apply to them the predicate **love**. This time, we use the $\cdot\!\gg$ and $\ll\!\cdot\!\gg$ which first evaluate both arguments left-to-right and pass the results to **love**. All of this happens in the scope of SI, so any quantification that takes place in the verb's arguments is resolved. If we were to simulate call-by-value in the outer application (evaluating the meanings of $\textsc{a woman}$ and $\textsc{every man}$ and then passing the results to the meaning of $\textsc{loves} : NP \multimap NP \multimap S$), we would have the two arguments evaluated in the opposite order and both of them escaping the scope island we set up in the verb's lexical entry.

$$
\begin{aligned}
&(\lambda OS.\,(\eta \circ \mathsf{SI})\,(\eta\,(\mathbf{love}\,S\,O))) \cdot\!\gg [\![\textsc{a woman}]\!] \ll\!\cdot\!\gg [\![\textsc{every man}]\!] \\
&\twoheadrightarrow [\![\textsc{a woman}]\!] \gg\!= (\lambda o.\,[\![\textsc{every man}]\!] \gg\!= (\lambda s.\,\eta\,(\mathsf{SI}\,(\eta\,(\mathbf{love}\,s\,o))))) \\
&\twoheadrightarrow [\![\textsc{a woman}]\!] \gg\!= (\lambda o.\,[\![\textsc{every man}]\!] \gg\!= (\lambda s.\,\eta\,(\mathbf{love}\,s\,o)))
\end{aligned}
$$

Since quantifiers are known to have ambiguous scope and the strength to leave scope islands under certain conditions, this reading might still be acceptable, even though it was not what our lexical entry intended to do (the SI handler had no effect at all). We can look at another example, Example 13: *John said "Mary loves me"*.

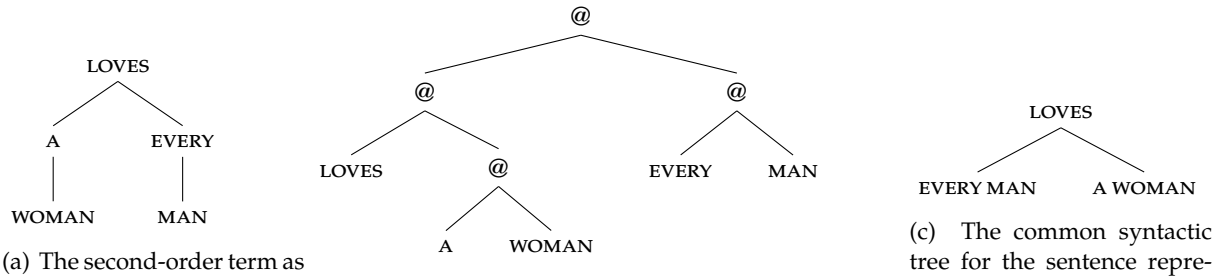
$$
\begin{aligned}
&[\![\textsc{said}_{\textsc{ds}}\,(\textsc{loves me}\,\textsc{Mary})\,\textsc{John}]\!] \\
&= (\lambda CS.\,S \gg\!= (\lambda s.\,\mathbf{say}\,s \cdot\!\gg (\mathsf{withSpeaker}\,s\,C)))\,[\![\textsc{loves me}\,\textsc{Mary}]\!]\,[\![\textsc{John}]\!] \\
&\twoheadrightarrow [\![\textsc{John}]\!] \gg\!= (\lambda s.\,\mathbf{say}\,s \cdot\!\gg (\mathsf{withSpeaker}\,s\,[\![\textsc{loves me}\,\textsc{Mary}]\!])) \\
&\twoheadrightarrow \mathbf{say}\,\mathbf{j} \cdot\!\gg (\mathsf{withSpeaker}\,\mathbf{j}\,[\![\textsc{loves me}\,\textsc{Mary}]\!]) \\
&\twoheadrightarrow \eta\,(\mathbf{say}\,\mathbf{j}\,(\mathbf{love}\,\mathbf{m}\,\mathbf{j}))
\end{aligned}
$$

Here we perform the application of the lexical entry of $\textsc{said}_{\textsc{ds}} : S \multimap NP \multimap S$ in a call-by-name manner. The computation $[\![\textsc{loves me}\,\textsc{Mary}]\!]$ gets evaluated only in the scope of the withSpeaker handler. On the other hand, the predicate $\mathbf{say} : \iota \to o \to o$ is applied to these computations in a call-by-value manner, forcing their evaluation down to their referents, **j** and **love m j**.

If we were to perform a call-by-value application already when applying the lexical entry for $\textsc{said}_{\textsc{ds}} : S \multimap NP \multimap S$ to the meanings of its arguments, we would end up binding the first-person pronoun not to John, but to the speaker of the utterance.

$$
\begin{aligned}
&(\lambda CS.\,\eta\,(\mathbf{say}\,S \cdot\!\gg (\mathsf{withSpeaker}\,S\,(\eta\,C)))) \cdot\!\gg [\![\textsc{loves me}\,\textsc{Mary}]\!] \ll\!\cdot\!\gg [\![\textsc{John}]\!] \\
&\twoheadrightarrow [\![\textsc{loves me}\,\textsc{Mary}]\!] \gg\!= (\lambda c.\,[\![\textsc{John}]\!] \gg\!= (\lambda s.\,\eta\,(\mathbf{say}\,s \cdot\!\gg (\mathsf{withSpeaker}\,s\,(\eta\,c))))) \\
&\twoheadrightarrow [\![\textsc{loves me}\,\textsc{Mary}]\!] \gg\!= (\lambda c.\,\eta\,(\mathbf{say}\,\mathbf{j} \cdot\!\gg (\mathsf{withSpeaker}\,\mathbf{j}\,(\eta\,c)))) \\
&\twoheadrightarrow \texttt{speaker} \star (\lambda x.\,\eta\,(\mathbf{say}\,\mathbf{j} \cdot\!\gg (\mathsf{withSpeaker}\,\mathbf{j}\,(\eta\,(\mathbf{love}\,\mathbf{m}\,x))))) \\
&\twoheadrightarrow \texttt{speaker} \star (\lambda x.\,\eta\,(\mathbf{say}\,\mathbf{j} \cdot\!\gg (\eta\,(\mathbf{love}\,\mathbf{m}\,x)))) \\
&\twoheadrightarrow \texttt{speaker} \star (\lambda x.\,\eta\,(\mathbf{say}\,\mathbf{j}\,(\mathbf{love}\,\mathbf{m}\,x)))
\end{aligned}
$$

When looking at a term like $\textsc{loves}\,(\textsc{a woman})\,(\textsc{every man})$, we might intuitively picture it as the tree in Figure 6.1a. If we wrap the denotation of the sentence produced by a transitive verb in a handler such as SI, we would expect the handler to apply to any operations that would be triggered by the subject or the object. However, if we look at the actual syntactic tree of that term (Figure 6.1b), we see that the subject



(a) The second-order term as a tree.

(b) The syntax of the term as a tree.

(c) The common syntactic tree for the sentence represented by the term.

Figure 6.1: Various trees representing the structure of the term LOVES (A WOMAN) (EVERY MAN).

and the object are not dominated by the transitive verb. If we would use call-by-value evaluation, their effects would not be captured by any handler inside the verb's lexical entry. That is why we perform these applications with call-by-name, moving the computations which are the denotations of the subject and the object inside the lexical entry for the transitive verb. The idea is that the structure of the sentence corresponds to the third tree in Figure 6.1c and we want to treat LOVES as a handler and EVERY MAN and A WOMAN as expressions having effects. Call-by-name takes care of plugging in the holes and moving the computations into the correct place (under the scope of the correct handlers, in the correct order) and that is why we use call-by-name for syntactic functions $\alpha \multimap \beta$.

If we find out that we always adhere to this rule, we can go further. We could introduce an impure language with algebraic effects (or use an existing one like λ_{eff} [63] or *Eff* [16]) and translate it to $(\!| \lambda |\!)$ in the same way that we translated $\lambda_{\texttt{shift}}$ and $\lambda_{\texttt{shift0}}$ in Chapter 4. The translation would take care of managing the computation types and ordering evaluation, freeing our hands from having to distinguish the types α and $\mathcal{F}_E(\alpha)$ and having to use $\gg\!=$ to dictate evaluation order. As we have seen above, the language would need to have both call-by-name and call-by-value abstractions (much like the language proposed by Kiselyov in [67]). In our previous attempt, we used a language without call-by-name abstractions and had to resort to wrapping expressions in thunks. In this manuscript, we found the order in which computations are evaluated too important to be handled by a translation hidden behind the calculus and we prefer to use explicit constructions such as $\gg\!=$ and computation types to make the evaluation explicit.

6.5.3 Choosing an Effect Signature

We have a simple grammar using computations in some principled way. We are ready to start introducing computations which actually do something and the reason we want to do that is to analyse phenomena which would otherwise defy compositionality. The workflow that we follow goes like this:

1. Choose a suitable effect signature

   This means identifying the operations that we will be using to implement the phenomenon, what their meaning should be, what input and output types they should have.

2. Introduce the operations in the lexical entries

   If we designed the operations correctly, we should be able to give meanings to lexical items that exhibit the phenomenon that we are studying.

3. Write handlers for the new operations

   Finally, we formalize the meaning of the new operations by writing handlers that interpret them in more basic terms. These handlers can then also be parts of lexical entries.

The first step is the most important. Having found the right set of operations, their use is usually straightforward. An effect signature is validated by writing handlers that give the desired semantics to computations that use the new operations. There are two processes that we follow in order to find the correct effect signature. We will give guidelines on how to use both.

Identifying the Interactions with the Context

Imagine that we are studying some non-compositional phenomenon. Let us remind ourselves of the definition of compositionality: the meaning of a complex expression is determined by its structure and the meanings of its constituents [124] (and *nothing else*). For meaning to be non-compositional is for meaning to depend on something that is not one of the above. If the meaning depends on something which is not part of the expression itself at all, we call that something the *context*.

If the meaning depends on some part of the context which can be represented with objects of type β, then we will introduce an operation with output type β. If there are multiple such parts of the context and the meaning needs to identify which one it depends on, the operation will have input type α, where α is the type of the object used to identify the part of the context we are interested in. If there is no need to identify a part of the context, we will use the trivial input type 1. Operations such as this can be though of as *getters* or *consumers*, their purpose is to retrieve something from the context.

Examples:

- Deixis, in particular first-person pronouns.

  A first-person pronoun is an atomic expression: it has no constituents (not counting case markers). Therefore its meaning cannot really depend on anything and would have to be fixed. However, the referent of the first-person pronoun varies from speaker to speaker. We want the first-person pronoun to designate the speaker, which is part of the context. The speaker is an individual (an entity) and so we will use an operation of output type ι. We assume that there is always at most a single entity which can be identified as the speaker of the utterance and so choose the input type 1. This gives us an operation of type $1 \rightarrowtail \iota$, same as the operation `speaker` that we have used in 6.2.

  $$\texttt{speaker} : 1 \rightarrowtail \iota$$

- Anaphora, such as in third-person pronouns.

  A third-person pronoun is also an atomic expression whose referent varies from situation to situation. Its referent depends on the individuals which are currently under discussion. Out of all such individuals, the pronoun needs the identity of the most salient one. We can therefore choose the output type of our operation to be ι. At any time, we could have mutliple salient individuals as candidate answers. The right answer will depend on which third-person pronoun we have used (gender and number). Thus we choose $\mu \times \nu$ as the input type, where μ is the type of grammatical genders and ν is the type of grammatical numbers.

  $$\texttt{select} : \mu \times \nu \rightarrowtail \iota$$

- Definite descriptions.

  A definite description serves to designate an individual. It is composed of the article *the* and a noun, the meaning of which is a set of individuals. For example, *the lazy cat* should designate some contextually salient member of the set of lazy cats. Without recourse to context, we do not know which individual to choose and we will therefore use an operation. What we ask for from the context is an individual and so the output type will be ι once more. Contrary to pronouns, we are not looking for just any salient individual but a salient individual that belongs to some specific set (the set of lazy cats in our example). Therefore, we will choose $\iota \rightarrow o$ as the input type of the operation.

  $$\texttt{find} : (\iota \rightarrow o) \rightarrowtail \iota$$

- Possible worlds.

  This one is similar to deixis. If we are working with modal operators such as *might* or *must* without using a modal logic, i.e. by quantifying over possible worlds and parameterizing predicates by possible worlds, then we will often need to access the current salient world. In order to know the

meaning of the noun *lazy cat*, i.e. the set of lazy cats, we will need to know w.r.t. which world we are speaking since different cats are lazy in different worlds. Similarly when we deal with verbs, we will need to know the current world to know whether or not entities are in some relation together. The information we want from the context is the current salient world. If σ is the type of possible worlds, we will introduce an operation of output type σ. There is no need to specify which current salient world we mean and so we leave the input type at 1.

$$\texttt{world} : 1 \rightarrowtail \sigma$$

We have examined the case of the expression's meaning being dependent on something which is not part of the expression itself. Note that this is not the only source of non-compositionality. The meaning could also depend on some part A of the expression which is not manifest in the meanings of its constituents. In these cases, we will use an operation in A to smuggle out the necessary information so that the expression can access it using a handler. The input type of this operation will be the piece of information that we will need to communicate "upwards". The output type will usually be less important, i.e. the trivial type 1. We can think of operations with the output type 1 as *setters* or *producers*. This pattern will be particularly useful for smuggling out non-at-issue content (conventional implicatures and presuppositions).

Examples:

- Conventional implicature.

  In Potts' logic of conventional implicatures [108], a *parsetree interpretation* step traverses the syntactic tree of a sentence and scoops up all the conventional implicatures generated by nominal appositives, non-restrictive relative clauses or other supplements. The truth conditions of the sentence are then a combination of the at-issue proposition and all the conventionally-implicated propositions. We will need an operation to use inside supplements and other conventional implicature triggers for smuggling out the conventional implicatures. The conventional implicatures will be propositions and so the input type will be o. We do not need to communicate anything through the output type and so we will leave it at 1.

$$\texttt{implicate} : o \rightarrowtail 1$$

- Presuppositions.

  Presuppositions form another truth-conditional component of the meaning of a sentence next to at-issue content and conventional implicatures. Them being not at-issue makes them a good candidate for being analyzed using effects. Let us be more specific and look at the presuppositions triggered by definite descriptions. The noun phrase *the X* presupposes that the set of Xs is not empty (e.g. the noun phrase *the king of France* presupposes that there is a king of France). Previously, we have introduced an operation $\texttt{find} : (\iota \to o) \rightarrowtail \iota$ for dealing with definite descriptions. We now see that this operation doubles as a mechanism with which the definite description can report a presupposition: $\texttt{find}!\,X$ triggers a presupposition that X is not empty and asks for the contextually salient element of X. If we ignore the output type, we can think of it as a producer of presuppositions of the form $\exists x.\, x \in X$.

Finally, sometimes we will run into cases for which the getter/setter or consumer/producer intuitions will not apply, as in the case of the `scope` operator we used in 6.4. In order to derive the type for `scope`, we have used the second process we use for designing effect signatures, which we will discuss shortly. However, even in the case of `scope`, we can reason about its type by asking what it gives to the context (the input type) and what it expects from the context (the output type). When we are dealing with a quantified noun phrases, what we have in hand, semantically, is a generalized quantifier. However, we are trying to find an individual as the noun phrase's referent. By juxtaposing what we have, $(\iota \to o) \to o$, and what we want, ι, we get the type of $\texttt{scope} : ((\iota \to o) \to o) \rightarrowtail \iota$. We can see the quantified noun phrase as contributing a quantifier to the context (the nearest sentence will need this quantifier in order

to correctly compute its own meaning) and asking in return for the bound variable which is to act as the noun phrase's "referent". The idea that quantified noun phrases can be interpreted by a process that sends the generalized quantifier somewhere else and puts a variable in its place was already established by Cooper's storage [32].

Looking for Inspiration Elsewhere

We now look at the other process we use to find effect signatures: looking for existing theories of effects. This process of finding effect signatures was itself inspired by Shan's paper on monads in natural language semantics [113]. In it, he revisits several semantic analyses and points out that the structures of types and combinators used within correspond to monads. The point behind the paper is that none of these analyses set out to use a monad. Since our computation types form a free monad (see 3.3.7), we can profit from identifying a monad in an existing linguistic analysis and then embedding that monad in our free monad. Other times, we may find a linguistic analysis which uses non-compositional/impure operators to construct the semantics and then base our operations on those operators.

Our most ambitious use of this technique combines both approaches. The types of de Groote's type-theoretic dynamic logic correspond to a monad of state and continuations. At the same time, the construction rules of DRT are based on a small set of effectful operations. In 7.2, we rewrite DRT construction rules into $(\!| \lambda |\!)$ computations and then write a handler which interprets such computations as dynamic propositions in de Groote's dynamic logic.

It is through this process that we chose the type for `scope`. We have seen delimited continuations, namely `shift` and `reset`, used to treat quantification [114, 115] and in Chapter 4, we have seen how to implement `shift` and `reset` in $(\!| \lambda |\!)$. Hence for the type of `scope`, we use the type of `shift0` developed in 4.6. The variant of `scope` that we use in Section 8.5 and that can be used in combination with other effects has the same type as the `shift0` in 4.6, $((\delta \to \mathcal{F}_E(\omega)) \to \mathcal{F}_E(\omega)) \rightarrowtail \delta$, with $\delta = \iota$ and $\omega = o$. In the simplified variant used in 6.4, we discard the $\mathcal{F}_E$'s and use the simpler type $((\iota \to o) \to o) \rightarrowtail \iota$.

One of the simplest monads, known as the *reader* monad, occurs very frequently in formal semantics. The reader monad is the monad that maps types α to types $\gamma \to \alpha$. It is the monad behind intensionalization, deixis and other pieces of context that meanings tend to be parameterized by. The dual to the reader monad is the *writer* monad which maps types α to types $\omega \times \alpha$ where ω is some monoid. Writer monads have been used in theories of expressives, which are a kind of conventional implicature, by Giorgolo and Asudeh [51, 48], Kiselyov and Shan [72] and Barker and Bumford [13].

Spotting monads like this is usually not that difficult because there is just a few common monads that tend to be reused and combined a lot. Most of the interesting ones are introduced already in Moggi's original paper [95]. Other sources of inspiration are libraries in Haskell that implement monads. After we have identified the monad, we still have to figure out the operations we will need to introduce in order to represent the monad in our free monad $\mathcal{F}_E$. A great source for this is the technique of characterizing monad transformers by the operations that they enable inside a monad [61, 84]. In these papers and in the documentation to the Haskell monad transformer library **mtl** [3], one can find for every common monad (transformer) the set of operations that it enables on the monad. For example, the reader monad $F(\alpha) = \gamma \to \alpha$ allows computations to *ask* for a value of type γ with a getter while the writer monad $F(\alpha) = \omega \times \alpha$ lets computations *write* values of type ω with a setter. We can then have the same functionality by making sure that we include these operations as either operation symbols in the effect signature or as handlers. This is exactly what we did with deixis in 6.2 and conventional implicature in 6.3: `speaker` is the operation that lets computations *ask* for the identity of the speaker, `implicate` is the operation that lets computations *write* conventional implicatures (the monoid in this writer monad is the conjunctive monoid on propositions).

Writing Handlers

After having decided on what operations to include in the effect signature, writing the denotations that use these operations is easy. The other challenging part is being able to give a meaningful interpretation to these operations, i.e. to write handlers. A handler is just a catamorphism on computations. To become

familiar with what a catamorphism on a structure like this can do and see more examples, we recommend the existing literature on calculi and languages with algebraic effects and handlers, namely [16] and [63].

6.5.4 When (Not) to Use Effects

The motivation for using a calculus of effects such as $(\!| \lambda |\!)$ when doing semantics is that we can eliminate a lot of the boring boilerplate which manages:

- passing the current possible world, current speaker and other deictic parameters from expression to subexpression
- threading the discourse state from one denotation to another
- chaining continuations of meanings which might project quantification
- collecting conventional implicatures from supplements

We can use the general notion of chaining computations using $\gg\!=$ and a lot of these issues work themselves out. This is because the effects automatically project themselves out (unless they are handled) and we can ignore them and focus on the at-issue content. These kinds of projecting behaviours are common in language (as testified by the above list of things we normally have to pass around) and so this technique is quite useful. However, there are cases when existing semantic analyses lift their types to accommodate some new phenomenon, maybe even using a monad, but adapting the analysis to use effects turns out not to be a good idea.

Blom, de Groote, Winter and Zwarts [21] use option types in the abstract language of an ACG to represent optional arguments. The option type NP^{o} is inhabited by NPs and by the value $\star$ (NP^{o} can be written using sums and products as $NP + 1$). Interpretations of constructions that accept optional arguments are then obliged to specify what interpretation should be used in case the argument is not present. In the syntactic interpretation, a missing argument is often interpreted as an empty string whereas in the semantic interpretation, a missing argument is commonly interpreted by some existential quantification. The system can then assign the meaning $\exists x.\ \mathbf{love\ j}\ x$ to the sentence "John loves" (abstract syntax LOVES $\star$ JOHN).

There is an option monad which maps types α to types α^{o}. The side effect that this monad implements is partiality. Computations with this effect can stop and decline to provide a result. This can be expressed with an operation $\texttt{abort} : 1 \rightarrowtail 0$. The impossible output type 0 signifies that this operation cannot be resumed or answered to. In the lexical entries for constructions that accept optional arguments, we use a handler for `abort` that handles missing arguments by replacing them with the default argument (e.g. an existential quantification).

This solution can be problematic in light of the projecting behavior of effects. If we ever forget to include a handler for `abort` in e.g. the lexical entry for the verb *reads*, then we would assign the meaning $\exists x.\ \mathbf{love\ m}\ x$ the sentence "*Mary loves that John reads*". The missing argument to *reads* would not be handled and project out which would result in *loves* seeing an argument which uses `abort` and therefore treating it as a missing argument. This can be easily solved by having distinct abstract types for optional and non-optional NPs and only permit the appearance of `abort` in the interpretations of optional constituents (i.e. $[\![NP]\!] = \mathcal{F}_E(\iota)$ and $[\![NP^{\mathrm{o}}]\!] = \mathcal{F}_{E \uplus \{\texttt{abort}:1 \rightarrowtail 0\}}(\iota)$). Then, a lexical entry which accepts an optional argument but does not explicitly handle the case when it is missing would not type-check. However, since we end up having to use distinct types for optional and non-optional items and are always forced to explicitly handle the case when an argument is missing, it makes little sense to treat optionality as an effect instead of directly using option types (i.e. having $[\![NP^{\mathrm{o}}]\!] = [\![NP]\!]^{\mathrm{o}} = \mathcal{F}_E(\iota)^{\mathrm{o}}$). This is a thing to remember: when using $(\!| \lambda |\!)$, we are not obliged to use effects every time it is possible. If it does not lead to a simpler solution, we can use existing techniques side-by-side with effects.

We will see another example of a semantic analysis that is based on lifting the types of denotations and that is not suitable for a solution using effects. Sai Qian's implementation [111] of Double Negation DRT [78] extends de Groote's type-theoretic dynamic logic. However, the lifting of the types and the associated terms do not form a functor, and therefore not a monad. Since our computation types form a

free monad, embedding a structure which does not fit the laws within them is difficult. We will return to this problem after having covered dynamic semantics, in 7.4.

7

Dynamic Semantics in $(\!| \lambda |\!)$

We will now examine dynamic semantics. We will consider two theories of dynamics in natural language: Kamp's DRT [64] and de Groote's TTDL [38, 80]. We will build a $(\!| \lambda |\!)$ analysis of dynamics and link it to both theories, as a side effect showing how DRT links to TTDL. The analysis we will present can be motivated on the grounds of either DRT or TTDL. We can look at the monad at the core of TTDL and devise operations that let us perform interesting things within the monad (quantifying over the discourse and modifying the discourse state). While this is the process that we have followed to discover this analysis (presented in [89]), we will follow a novel strategy in this exposition. We will start with DRT, more specifically its presentation in Kamp and Reyle's canonical textbook [64], and show how to translate it into $(\!| \lambda |\!)$ computations (Section 7.2). We will then interpret those computations as TTDL dynamic propositions.

Afterwards, we will have our first taste of combining different effects. We will enhance our theory of dynamics with a treatment of presuppositions covering projection, cancellation and accommodation. Our analysis will be based on Lebedeva's extension of TTDL [80], which uses exceptions and exception handlers to handle presupposition projection and accommodation. We will see a variation of this approach in terms of effects and handlers (Section 7.3).

Finally, we consider one more extension of TTDL, Qian's double-negation TTDL. This will be an example of a negative result. We will see why and how Qian's DN-TTDL evades an analysis in $(\!| \lambda |\!)$ (Section 7.4).

Contents

7.1 DRT as a Programming Language . . . 154
7.2 $(\!| \lambda |\!)$ Analysis . . . 159
7.2.1 Example . . . 162
7.2.2 Handler for Dynamics . . . 163
7.2.3 Negation . . . 164
7.2.4 Truth Conditions as Side Effects . . . 166
7.2.5 Algebraic Considerations . . . 167
7.3 Presuppositions . . . 169
7.3.1 Revising the Dynamic Handler . . . 171
7.3.2 Presupposition in Action . . . 174
7.3.3 Cancelling Presuppositions . . . 176
7.3.4 Ambiguous Accommodation . . . 181
7.3.5 Comparison with TTDL . . . 193
7.4 Double Negation . . . 194
7.4.1 Double Negation as an Effect . . . 195
7.4.2 DN-TTDL is Not Monadic . . . 196
7.4.3 DN-TTDL is Not Functorial . . . 197
7.5 Summary . . . 198

7.1 DRT as a Programming Language

We will argue that the construction rules for DRSs as presented in [64] can be seen as an operational semantics for a programming language. Once we will have established that DRT is a programming language, we will use techniques similar to those in Chapter 4 to embed this language in ⦇ λ ⦈.

DRS-Construction Algorithm

Input: discourse $D = S_1, \ldots, S_i, S_{i+1}, \ldots, S_n$
the empty DRS K_0

Keep repeating for $i = 1, \ldots, n$**:**

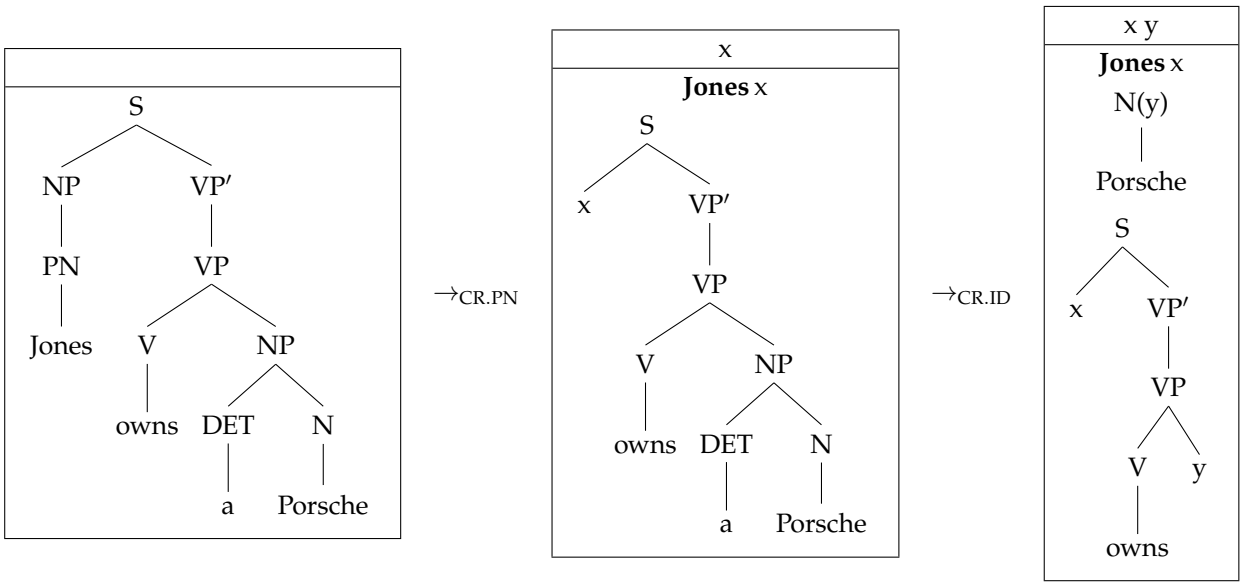
(i) add the syntactic analysis $[S_i]$ of (the next) sentence S_i to the conditions of K_{i-1}; call this DRS K_i^*. Go to (ii).

(ii) Input: a set of reducible conditions of K_i^*
Keep on applying construction principles to each reducible condition of K_i^* until a DRS K_i is obtained that only contains irreducible conditions. Go to (i).

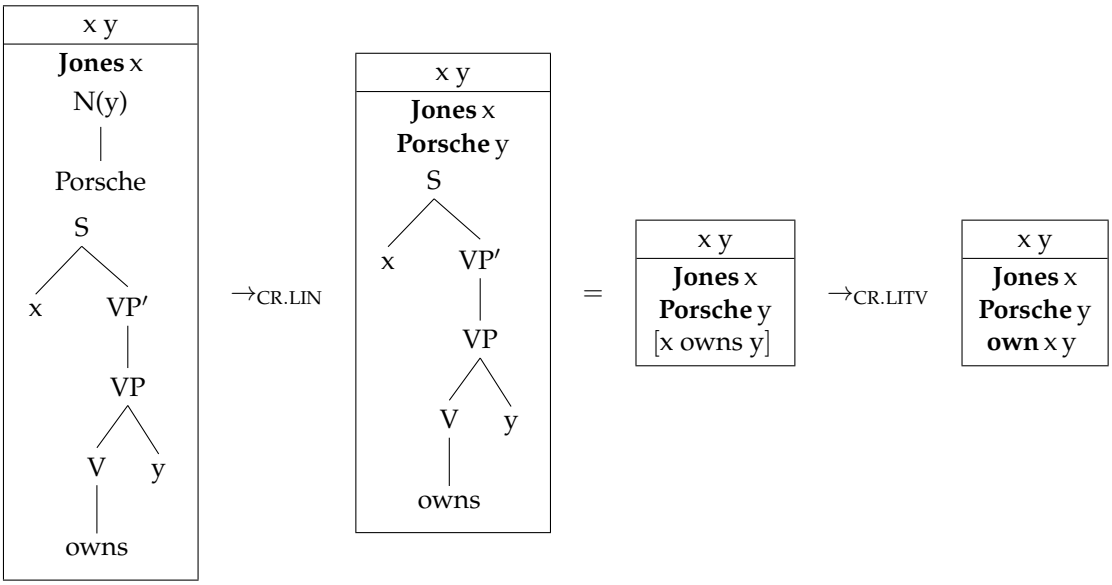
In 5.3.1, we have seen that the conditions of a DRS resemble formulas of predicate logic: predicates applied to variables and logical operators such as negation, implication or disjunction. However, during DRS construction, this notion is expanded. The formulas described above are called *irreducible conditions*. Along with them, we will also have syntactic trees as conditions. Furthermore, these syntactic trees might contain discourse referents.

We will look at DRS construction for an example from [64] (Example (1.28), Section 1.1.3):

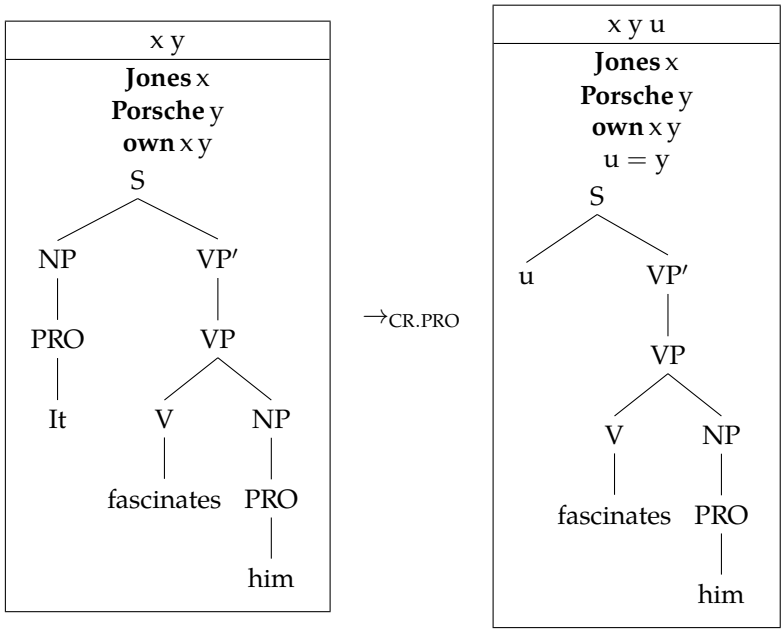
(6) Jones owns a Porsche. It fascinates him.



We insert the syntactic analysis of the first sentence into the empty DRS. Then there is a reduction rule which replaces the NP *Jones* with a discourse referent x while at the same time introducing the discourse referent x and the condition **Jones** x into the DRS. In the next DRS, we evaluate the object by replacing it with y and adding the discourse referent y and a condition in which (the meaning of) the noun *Porsche* is applied to y.



Having reduced the noun phrase *a Porsche*, we are now led to evaluate the noun *Porsche* itself. It yields the predicate **Porsche**. At this point, the algorithm presented in [64] stops. However, in order to be a little bit more uniform, we will reduce the piece of syntax [x owns y] and replace it with an atomic formula **own** x y. We can now add the syntactic analysis of the second sentence and proceed with computation.



Again, we start by evaluating the subject. This time, it is a pronoun, but the process remains largely the same. We replace the pronoun with a new discourse referent u, which we introduce into the DRS along with the condition u = y.

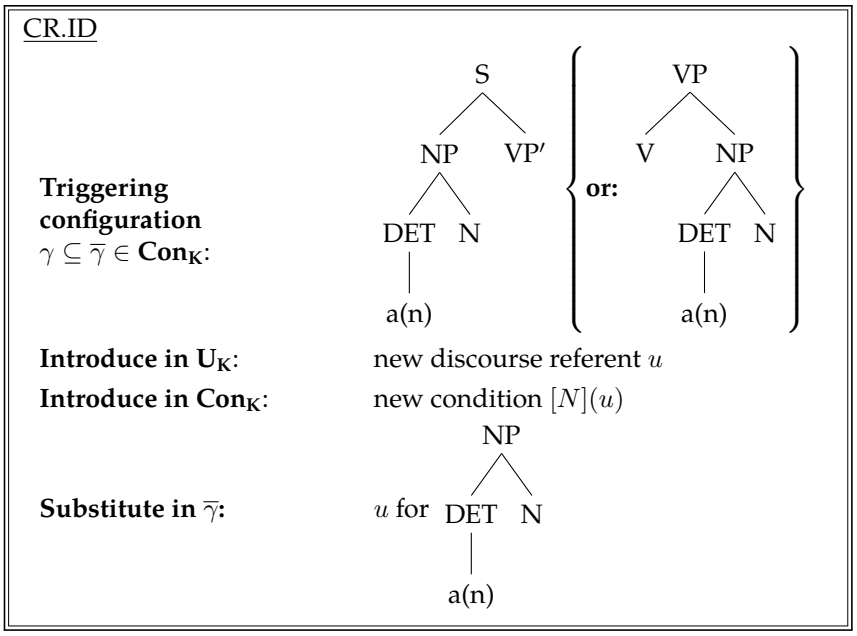
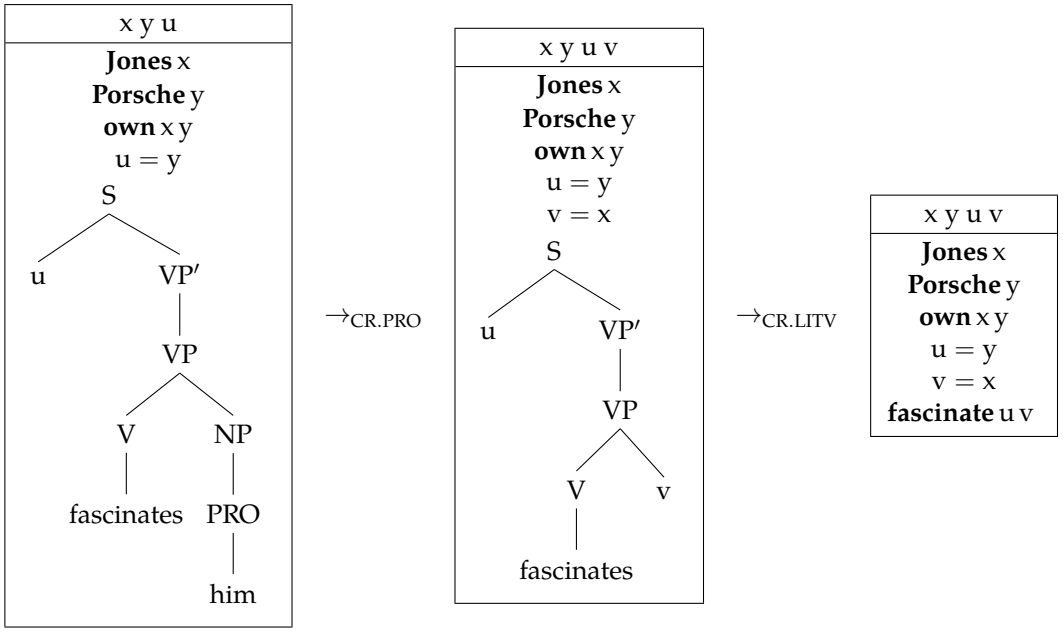


Figure 7.1: CR.ID: The construction rule for indefinite descriptions.



For the object pronoun we do the same, introducing the discourse referent v and the condition v = x. Finally, all that is left is to translate the transitive verb into a binary relation and we get the final DRS representation.

Let us now look at the formulation of the construction rules, starting with CR.ID in Figure 7.1.[123]

The *triggering configuration* describes the rule's redex (and part of its context). In the case of CR.ID, the rule for indefinite descriptions, the redex is a noun phrase of the form *a(n)* N. The actual triggering configuration also includes the node dominating the NP as well. This is for reasons of evaluation order:

"A reducible condition γ must be reduced by applying the appropriate rule to its *highest* trig-

[123]The construction rules presented in [64] also include gender features, which are necessary for correct anaphora resolution. We will omit them from the rules as we will not be studying anaphora resolution in this work.

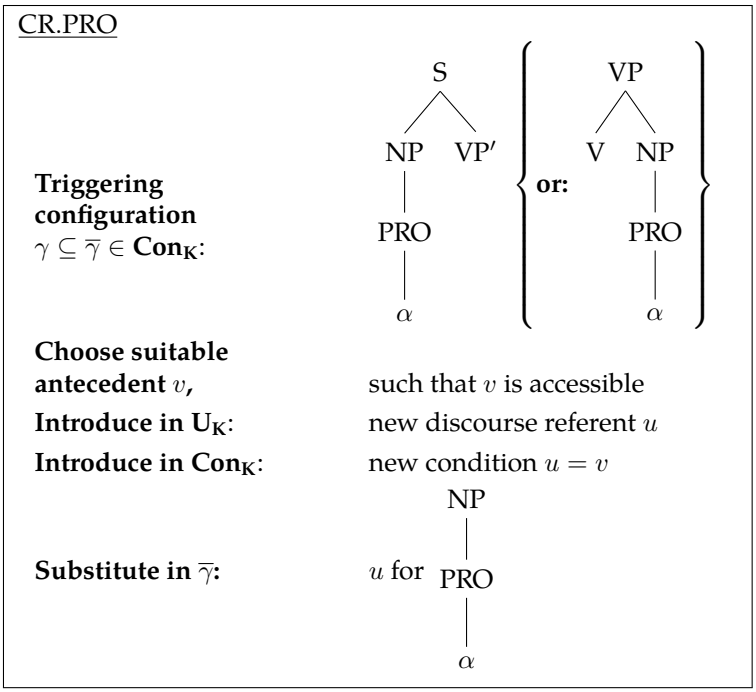


Figure 7.2: CR.PRO: The construction rule for (anaphoric third-person singular) pronouns.

gering configuration, i.e. that triggering configuration τ such that the highest node of τ dominates the highest node of any other triggering configuration that γ contains."

From Discourse to Logic [64] (Section 1.1.4, page 87 in the Student Edition)

In this rule, the extra piece of context in the triggering configuration makes it so that the application of the rule to the subject dominates the application of the same rule to the object and therefore fixes evaluation order: first subject, then object.

The rule then has its contractum, which is given on the last line (**Substitute in** $\overline{\gamma}$). In this case, the contractum is the discourse referent u. The rule also has two important side effects. First, the discourse referent u is introduced into the DRS that contains this condition. Second, the condition $[N](u)$ is added to the conditions of that same DRS. The notation $[N](u)$ means that we copy the whole syntactic structure of N (i.e. the whole program for computing the meaning of the noun) and once its meaning (a predicate) is computed, we apply it to the discourse referent u.

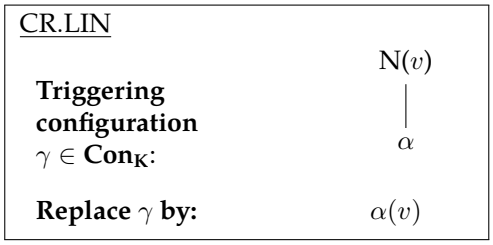
The rule for pronouns, CR.PRO in Figure 7.2, is very similar. It introduces a new kind of operation, in which the NP being evaluated is asking its context for a suitable anaphoric referent.

We will leave the rule for proper nouns until Section 7.3 and give the two lexical insertion rules for nouns and transitive verbs [124] — Figures 7.3a and 7.3b, respectively.

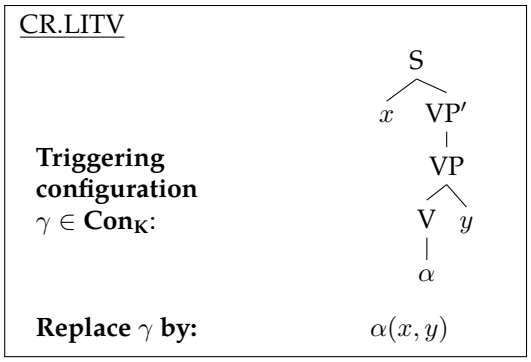
The system from [64] described above can be presented in a manner reminiscent of operational semantics of programming languages.

The terms of this language are DRSs which contain as conditions syntactic trees with discourse referents. The values in this language are discourse referents and plain DRSs (DRSs whose conditions are formulas). We will define evaluation contexts C. These should reflect the fact the construction algorithm of DRT permits reduction in any of the conditions within a DRS.

[124] The latter is not present in [64]. Instead, [x loves y] is treated as an irreducible condition.



(a) CR.LIN: The lexical insertion rule for (common) nouns.



(b) CR.LITV: The lexical insertion rule for (transitive) verbs.

Figure 7.3: The lexical insertion construction rules.

$$C ::= [] \mid \begin{array}{|c|}\hline \mathrm{x}_1 \ldots \mathrm{x}_n \\ \hline \gamma_1 \\ \ldots \\ \neg C \\ \ldots \\ \gamma_m \\ \hline \end{array} \mid \begin{array}{|c|}\hline \mathrm{x}_1 \ldots \mathrm{x}_n \\ \hline \gamma_1 \\ \ldots \\ C \Rightarrow K \\ \ldots \\ \gamma_m \\ \hline \end{array} \mid \begin{array}{|c|}\hline \mathrm{x}_1 \ldots \mathrm{x}_n \\ \hline \gamma_1 \\ \ldots \\ K \Rightarrow C \\ \ldots \\ \gamma_m \\ \hline \end{array} \mid \begin{array}{|c|}\hline \mathrm{x}_1 \ldots \mathrm{x}_n \\ \hline \gamma_1 \\ \ldots \\ C \vee K \\ \ldots \\ \gamma_m \\ \hline \end{array} \mid \begin{array}{|c|}\hline \mathrm{x}_1 \ldots \mathrm{x}_n \\ \hline \gamma_1 \\ \ldots \\ K \vee C \\ \ldots \\ \gamma_m \\ \hline \end{array}$$

The reducible conditions are syntactic trees. The triggering configuration can be found in any part of a syntactic tree and so we define C_γ to be a context that places [] inside a syntactic tree. For every rule $A \to B_1 \ldots B_n$, there will be a production rule for the context $C_\gamma ::= A(B_1, \ldots, B_{i-1}, C_\gamma, B_{i+1}, \ldots, B_n)$.

$$C_\gamma ::= [] \mid \begin{array}{c}\mathrm{S} \\ C_\gamma \quad \mathrm{VP}\end{array} \mid \begin{array}{c}\mathrm{S} \\ \mathrm{NP} \quad C_\gamma\end{array} \mid \begin{array}{c}\mathrm{VP} \\ C_\gamma \quad \mathrm{NP}\end{array} \mid \begin{array}{c}\mathrm{VP} \\ \mathrm{V} \quad C_\gamma\end{array} \mid \ldots$$

The construction rule CR.ID is then a reduction rule on these forms:

$$C \left[\begin{array}{|c|}\hline \mathrm{x}_1 \ldots \mathrm{x}_n \\ \hline \gamma_1 \\ \ldots \\ C_\gamma \left[\begin{array}{c}\mathrm{S} \\ \mathrm{NP} \quad \mathrm{VP}' \\ \mathrm{DET} \quad \mathrm{N} \\ \mathrm{a(n)} \end{array} \right] \\ \ldots \\ \gamma_m \\ \hline \end{array} \right] \to_{\mathrm{CR.ID}} C \left[\begin{array}{|c|}\hline \mathrm{x}_1 \ldots \mathrm{x}_n \ \mathrm{u} \\ \hline \gamma_1 \\ \ldots \\ [\mathrm{N}](\mathrm{u}) \\ C_\gamma \left[\begin{array}{c}\mathrm{S} \\ \mathrm{u} \quad \mathrm{VP}' \end{array} \right] \\ \ldots \\ \gamma_m \\ \hline \end{array} \right]$$

Likewise, there is an analogue for evaluating indefinite descriptions in object positions which differs only in the triggering configuration. The reduction rules for CR.PRO, CR.LIN and CR.LITV can be derived in the same way. Then, the DRS that corresponds to a sentence can be seen as a normal form[125] in this reduction system.

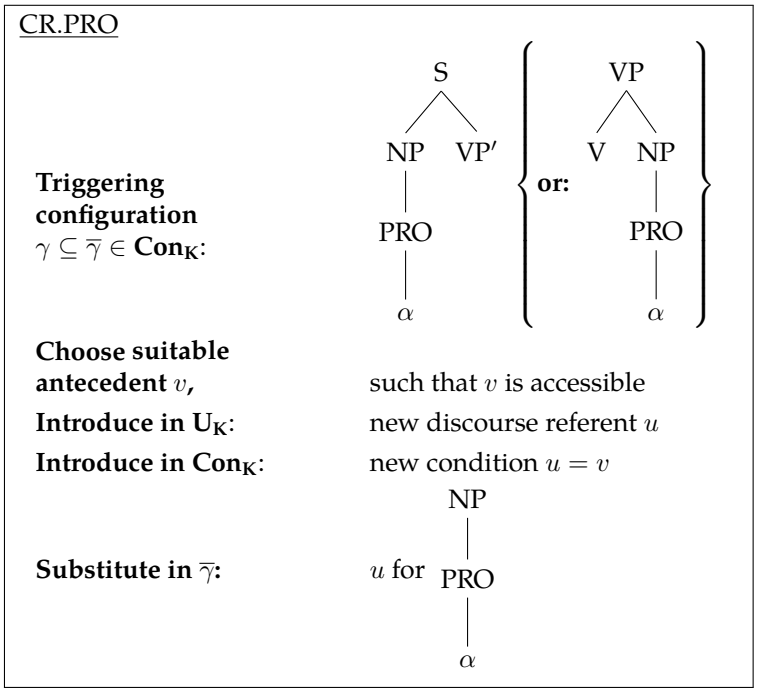
If we look at the rule, then we can see a remarkable similarity to the rules seen in Chapter 4 for $\lambda_{\texttt{shift}}$ and $\lambda_{\texttt{shift0}}$. Inside the context C, we have some kind of delimiting construction, a DRS in one case and

[125]The system permits reductions in different conditions at the same time and is therefore not confluent.

a `reset` in the other. As one of the arguments of this construction, we have an expression buried inside a more limited context, C_γ, which cannot contain any more nested DRSs, and F, which cannot contain any more `resets`. In the case of DRT, this buried expression is an indefinite or a pronoun which wants to access the context's DRS to add (and possibly look for) discourse referents and conditions. In the case of $\lambda_{\texttt{shift}}$, the buried expression is a `shift` that wants complete control over the context inside the `reset`. In our analysis of $\lambda_{\texttt{shift}}$, `reset` corresponded directly to a handler. Therefore, we will be treating DRSs as handlers in the coming ⦇λ⦈ analysis. The denotations of indefinites and pronouns will use operations to introduce new discourse referents and conditions and to query the state of discourse to resolve anaphora.

7.2 ⦇λ⦈ Analysis

The first step in building a ⦇λ⦈ analysis of dynamics is to design the effect signature: how many operations we will need, what their types should be and what they should do. However, our task is largely facilitated by the fact that in their exposition of DRT [64], Kamp and Reyle have structured the construction rules by using a limited set of operations to manipulate the DRSs. It is these operations that we will include in our effect signature. Consider the construction rule for pronouns and the corresponding representation as a ⦇λ⦈ computation.



$$\alpha : NP$$

$$[\![NP]\!] = \mathcal{F}_E(\iota)$$

$$\begin{array}{lll} [\![\alpha]\!] = \texttt{choose} & \star & (\lambda v. \\ \quad\;\; \texttt{introduce} & \star & (\lambda u. \\ \quad\;\; \texttt{assert} & (u = v) & (\lambda\_. \\ & & \\ & & \eta\, u))) \end{array}$$

Let α be a third-person singular pronoun.[126] The construction rule reduces the NP node formed by α into a discourse referent. In the corresponding analysis in our formalism (ACG + ⦇λ⦈), we have α as an abstract constant of type NP whose semantic interpretation is a computation of type $\mathcal{F}_E(\iota)$. The pronoun asks the context for a suitable antecedent, which is then referred to as v. We mimic the verb **choose** with an operation `choose`. It does not take any input, as no input is given to **choose** in the construction rule,[127] and expects a discourse referent as output. Since we will be using the type ι for terms that designate individuals, we will identify the type of discourse referents with ι.

$$\texttt{choose} : 1 \rightarrowtail \iota$$

[126] Since we ignore gender, we can think of α as *the* third-person singular pronoun.

[127] If we were to care about gender markers, the input of this operation would be the gender marker/predicate, much like in the example of the `select` operator proposed in 6.5.3.

Next up, the construction rule demands the introduction of a new discourse referent into the DRS **K** that contains the condition being evaluated. This instruction and its use of the verb **introduce** gives rise to the `introduce` operation. `introduce` asks for a fresh discourse referent and so its type ends being the same as the one for `choose`, only the semantics differ.

$$\texttt{introduce} : 1 \rightarrowtail \iota$$

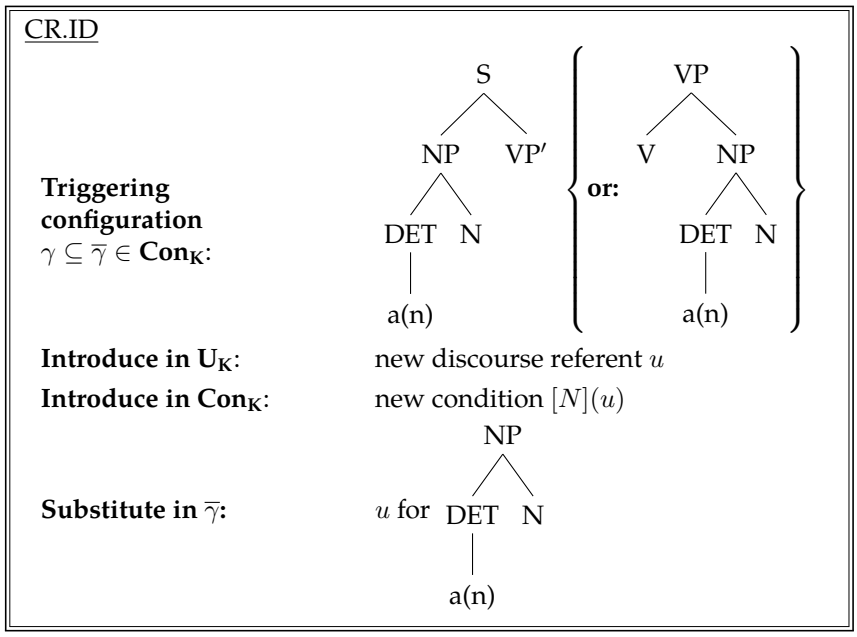
The next step in the construction rule asks the DRS **K** to introduce a new condition. For this kind of interaction, **introducing** a condition, we will use a new operation, `assert`. The NP indicates the condition it wants to add to the DRS, the truth condition that it wants to assert. Conditions are atomic formulas of predicate logic and so we will use o, the type of propositions, to represent them. The output will be of the trivial type 1.

$$\texttt{assert} : o \rightarrowtail 1$$

Finally, the construction rule tells us to replace the NP node with the discourse referent u. This means that when this NP occurs as an argument to a predicate, the predicate should be applied to the discourse referent u. In ⦇ λ ⦈, this role is played by the return values of computations (see equation below). Therefore, we return u with the computation $\eta\, u$.

$$\textbf{predicate} \cdot\!\gg (\texttt{op}\, M_{\mathrm{p}}\, (\lambda x.\ \ldots\ \eta\, u)) = \texttt{op}\, M_{\mathrm{p}}\, (\lambda x.\ \ldots\ \eta\, (\textbf{predicate}\, u))$$

We can do the same kind of analysis/translation for the CR.ID rule for indefinite descriptions.



$$\begin{aligned} \textsc{a} &: N \multimap NP \\ [\![N]\!] &= \mathcal{F}_E(\iota \to o) \\ [\![NP]\!] &= \mathcal{F}_E(\iota) \end{aligned}$$

$$\begin{array}{llll} [\![\textsc{a}]\!] = \lambda N. & \texttt{introduce} & \star & (\lambda u. \\ & N & \gg\!= & (\lambda n. \\ & \texttt{assert} & (n\, u) & (\lambda\_. \\ & & & \\ & \eta\, u))) & & \end{array}$$

The CR.ID rule evaluates a noun phrase which is composed of the indefinite article followed by some noun N. This construction is represented in our ACG as an abstract constant $\textsc{a} : N \multimap NP$. Its denotation will be a function from $[\![N]\!]$ to $[\![NP]\!]$. As in the rest of the analyses seen in this chapter, we will take $[\![N]\!] = \mathcal{F}_E(\iota \to o)$, which meshes with the fact that DRT expects nouns to reduce to predicates.

The evaluation of the noun phrase "*a N*" starts by introducing a fresh discourse referent u and so we use the same operation as in CR.PRO. Then we will proceed by adding the condition $[N](u)$. Note that in the DRT rule, we are dealing with a reducible condition ($[N]$ is the syntax of N). In adding this reducible condition, DRT essentially schedules the evaluation of the syntactic expression $[N]$ via some other construction rule (CR.LIN if N is only a common noun, other rules if it is, e.g., restricted by a relative clause or an adjective). In ⦇ λ ⦈, we achieve a similar effect by asking for the evaluation of N using

the $\gg\!=$ operator. We then state that once the noun has been evaluated down to a predicate, we want this predicate, applied to the referent u, to be a condition inside the DRS. Finally, as in CR.PRO, we present the discourse referent u as the referent of the noun phrase.

For completeness, we will also give translations for the CR.LIN and CR.LITV rules, even though they do not have any dynamic effects of their own.

<u>CR.LIN</u>

Triggering configuration $\gamma \in$ **Con$_\mathbf{K}$**:

Replace γ by: $\alpha(v)$

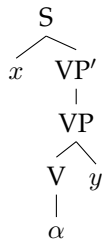
$$\begin{aligned}
\textsc{common} &: CN \multimap N \\
[\![CN]\!] &= \iota \to o \\
[\![N]\!] &= \mathcal{F}_E(\iota \to o)
\end{aligned}$$

$$\begin{aligned}
[\![\textsc{common}]\!] &= \lambda\alpha.\, \eta\,(\lambda v.\, \alpha\, v) \\
&= \lambda\alpha.\, \eta\, \alpha \\
&= \eta
\end{aligned}$$

We assume that behind every common noun α lies a set of individuals, a predicate α. The lexical insertion rule for nouns replaces the noun by that predicate. We can capture the same line of reasoning in ACGs. We contrast the category CN of common nouns (such as *snowman*, *snake*, *ladder*) to the larger category N of nouns (*animal in your garden*, *man who owns a donkey*). The common nouns will correspond to plain sets of individuals, $[\![CN]\!] = \iota \to o$. However, more complex nouns might also have effects[128] and so we have $[\![N]\!] = \mathcal{F}_E(\iota \to o)$. In ACGs, to say that every common noun CN is a noun N is to provide an injection of type $CN \multimap N$. Homomorphically, its denotation will be an injection from $\iota \to o$ to $\mathcal{F}_E(\iota \to o)$, the constructor η.

<u>CR.LITV</u>

Triggering configuration $\gamma \in$ **Con$_\mathbf{K}$**:



Replace γ by: $\alpha(x, y)$

$$\begin{aligned}
\textsc{trans} &: V \multimap NP \multimap NP \multimap S \\
[\![V]\!] &= \iota \to \iota \to o \\
[\![NP]\!] &= \mathcal{F}_E(\iota) \\
[\![S]\!] &= \mathcal{F}_E(o)
\end{aligned}$$

$$\begin{aligned}
[\![\textsc{trans}]\!] &= \lambda\alpha Y X.\, X \gg\!= (\lambda x. \\
&\qquad Y \gg\!= (\lambda y. \\
&\qquad \eta\,(\alpha\, x\, y))) \\
&= \lambda\alpha Y X.\, \alpha \cdot\!\gg X \ll\!\cdot\!\gg Y
\end{aligned}$$

With CR.LITV, the idea behind the DRT/⦇λ⦈ analogy is the same as with CR.LIN. Behind every (extensional transitive) verb α lies a binary relation, also called α. When combined with a subject and an object, verbs form sentences. This is embodied by the ACG abstract constant TRANS which maps verbs from V into functions in $NP \multimap NP \multimap S$. From the triggering configuration of the rule CR.LITV, we see that (the parent of) the subject dominates (the parent of) the object. This leads DRT to always evaluate the dynamic effects of the subject before the object. In ⦇λ⦈, this feature is expressed in the lexical entry for the construction that combines the subject, verb and object into a sentence, TRANS, where X is evaluated before Y using $\gg\!=$.

[128] In the case of dynamics, it might be just anaphora, but more generally it might also be indexicality, quantification, conventional implicature…

7.2.1 Example

We have seen how to map the syntax-semantics construction rules of DRT into the ACG formalism and how the extra steps performed when reducing indefinites or pronouns in DRT correspond to operations, with which we have extended ACG's λ-calculus in ⦇ λ ⦈. We can now look at an example in action.

(19) A man owns a Porsche. It fascinates him.

This is a small variation of Example 6 in which *Jones* was replaced by *a man*.[129]

$$
\begin{aligned}
&[\![\textsc{trans owns}\,(\textsc{a}\,(\textsc{common}\,\textsc{Porsche}))\,(\textsc{a}\,(\textsc{common man}))]\!]\\
&\quad \twoheadrightarrow \texttt{introduce}\ \star\ (\lambda x.\\
&\qquad \texttt{assert}\,(\mathbf{man}\,x)\,(\lambda\_.\\
&\qquad \texttt{introduce}\ \star\ (\lambda y.\\
&\qquad \texttt{assert}\,(\mathbf{Porsche}\,y)\,(\lambda\_.\\
&\qquad \eta\,(\mathbf{own}\,x\,y)))))
\end{aligned}
$$

The only effects are due to the indefinites that introduce new discourse referents and assert truth conditions. The operations are ordered subject-first, object-last and the computation returns the predicate that is the application of the verb's predicate to the referents of the subject and the object. The same goes for the second sentence in Example 19:

$$
\begin{aligned}
&[\![\textsc{trans fascinates him it}]\!]\\
&\quad \twoheadrightarrow \texttt{choose}\ \star\ (\lambda y'.\\
&\qquad \texttt{introduce}\ \star\ (\lambda u.\\
&\qquad \texttt{assert}\,(u = y')\,(\lambda\_.\\
&\qquad \texttt{choose}\ \star\ (\lambda x'.\\
&\qquad \texttt{introduce}\ \star\ (\lambda v.\\
&\qquad \texttt{assert}\,(v = x')\,(\lambda\_.\\
&\qquad \eta\,(\mathbf{fascinate}\,u\,v)))))))
\end{aligned}
$$

We can compose the two using $\ll\wedge\gg$, the conjunction of propositions raised to computations.

$$
\begin{aligned}
&[\![\textsc{trans owns}\,(\textsc{a}\,(\textsc{common}\,\textsc{Porsche}))\,(\textsc{a}\,(\textsc{common man}))]\!] \ll\wedge\gg [\![\textsc{trans fascinates him it}]\!]\\
&\quad \twoheadrightarrow \texttt{introduce}\ \star\ (\lambda x.\\
&\qquad \texttt{assert}\,(\mathbf{man}\,x)\,(\lambda\_.\\
&\qquad \texttt{introduce}\ \star\ (\lambda y.\\
&\qquad \texttt{assert}\,(\mathbf{Porsche}\,y)\,(\lambda\_.\\
&\qquad \texttt{choose}\ \star\ (\lambda y'.\\
&\qquad \texttt{introduce}\ \star\ (\lambda u.\\
&\qquad \texttt{assert}\,(u = y')\,(\lambda\_.\\
&\qquad \texttt{choose}\ \star\ (\lambda x'.\\
&\qquad \texttt{introduce}\ \star\ (\lambda v.\\
&\qquad \texttt{assert}\,(v = x')\,(\lambda\_.\\
&\qquad \eta\,(\mathbf{own}\,x\,y \wedge \mathbf{fascinate}\,u\,v)))))))))))
\end{aligned}
$$

[129] We relegate the discussion of proper nouns to Section 7.3.

The resulting computation introduces 4 discourse referents: x, y, u and v. Assuming that `choose` will give us x for x' and y for y', then we end up introducing and returning the conditions **man** x, **Porsche** y, $u = x$, $v = y$, **own** $x\,y$ and **fascinate** $u\,v$, i.e. the same conditions as in the DRS for Example 6 (replacing **Jones** x with **man** x). However, while that might be our desired interpretation of this computation, for now it is just a string of (operation) symbols, to which we will now need to give some meaning.

7.2.2 Handler for Dynamics

In order to give a formal semantics to the operations that we have introduced for DRT, we will write a handler. The type of dynamic propositions of de Groote's Type-Theoretic Dynamic Logic (TTDL), $\gamma \to (\gamma \to o) \to o$, will serve as a suitable interpretation domain. On top of that, apart from implementing a semantics for our operations, we will have demonstrated the link between DRT and TTDL. The type of the handler's input will be $\mathcal{F}_E(o)$, where E is the effect signature containing only `choose`, `introduce` and `assert`.

$$
\begin{aligned}
\mathsf{TTDL} &: \mathcal{F}_E(o) \to \gamma \to (\gamma \to o) \to o \\
\mathsf{TTDL} &= \llparenthesis\ \texttt{choose}\colon (\lambda\_ke\phi.\, k\,(\mathsf{sel}\,e)\,e\,\phi), \\
&\quad\ \texttt{introduce}\colon (\lambda\_ke\phi.\, \exists x.\, k\,x\,(x :: e)\,\phi), \\
&\quad\ \texttt{assert}\colon (\lambda pke\phi.\, p \wedge k \star (p :: e)^{130}\,\phi), \\
&\quad\ \eta\colon (\lambda pe\phi.\, p \wedge \phi\,e)\ \rrparenthesis
\end{aligned}
$$

We write the handler by following the types, keeping in mind the intended semantics. There is a notable similarity between the clauses of our handler and the operators and semantic entries used in TTDL. The η clause is the operation that lifts plain propositions into dynamic propositions [80]. The clause for `choose` is (ignoring the first dummy argument) exactly the denotation assigned to pronouns in TTDL [38] (Subsection 5.4.3). The clause for `introduce` is the denotation of the indefinite *someone*, also known as the dynamic existential quantifier in [80] (Subsection 5.4.1). This is not a coincidence. The clauses for `choose` and `introduce` are both values of type ι written in continuation-passing style with observation type $\Omega = \gamma \to (\gamma \to o) \to o$, i.e. values of type $(\iota \to \Omega) \to \Omega$. TTDL uses continuation-passing style (generalized quantifiers) in the denotation of its noun phrases and so their denotations end up having the same type. The two noun phrases, the pronoun and the indefinite, stand for the two major ways that dynamic noun phrases interact with their contexts, retrieving their referents from context or introducing new ones into the context. Together with `assert`, which lets us conjoin another proposition to the observed proposition, these form the three operations with which we characterize DRT-style dynamics. Our original discovery of these three operations with which we can characterize dynamic effects came from the realization that we can express all the denotations in TTDL with: reading from the context (`choose`, or later `get`), wrapping an existential quantifier over the continuation while adding the bound variable into the context (`introduce`) and conjoining a proposition to the continuation (`assert`).

Once we have applied the TTDL handler to a computation of type $\mathcal{F}_E(o)$, we can extract the corresponding (static) proposition by applying the result to some left context and the trivial right context ($\lambda e.\, \top$). Since this what we will be doing most of the time, we can simplify the TTDL handler by assuming that ϕ is always equal to $\lambda e.\, \top$.

[130] Unlike the TTDL we presented in Section 5.4, the contexts will be composed not only of accessible discourse referents (which are added to the context in the `introduce` clause) but also of propositions added to the common ground (which are added by `assert`). As in [80], this information will be necessary to correctly handle presuppositions.

$$
\begin{aligned}
\mathsf{box} &: \mathcal{F}_E(o) \to \gamma \to o \\
\mathsf{box} &= \llparenthesis\ \texttt{choose}\colon (\lambda\_ke.\, k\,(\mathsf{sel}\, e)\, e), \\
&\qquad \texttt{introduce}\colon (\lambda\_ke.\, \exists x.\, k\, x\, (x :: e)), \\
&\qquad \texttt{assert}\colon (\lambda pke.\, p \wedge k \star (p :: e)), \\
&\qquad \eta\colon (\lambda pe.\, p)\ \rrparenthesis
\end{aligned}
$$

The handler is called box to stress the analogy to DRSs, which are customarily drawn as boxes. The crucial common point is that putting something in a box means that any truth conditions or discourse referents that it introduces are contained within the box. Throughout this chapter, we will refer to applications of this handler as boxes, DRSs or contexts.[131]

Getting back to Example 19, we can now apply one of these two handlers to get the dynamic proposition.

$$
\begin{aligned}
&\mathsf{TTDL}\,(\llbracket \textsc{trans owns}\,(\textsc{a}\,(\textsc{common Porsche}))\,(\textsc{a}\,(\textsc{common man}))\rrbracket \ll\wedge\gg \llbracket \textsc{trans fascinates him it}\rrbracket) \\
&\twoheadrightarrow \lambda e\phi.\, \exists x.\, \mathbf{man}\, x \wedge (\exists y.\, \mathbf{Porsche}\, y \wedge (\exists u.\, u = \mathsf{sel}\, e_1 \wedge (\exists v.\, v = \mathsf{sel}\, e_2 \wedge \mathbf{own}\, x\, y \wedge \mathbf{fascinate}\, u\, v \wedge \phi\, e_3))) \\
&= \lambda e\phi.\, \exists xyuv.\, \mathbf{man}\, x \wedge \mathbf{Porsche}\, y \wedge u = \mathsf{sel}\, e_1 \wedge v = \mathsf{sel}\, e_2 \wedge \mathbf{own}\, x\, y \wedge \mathbf{fascinate}\, u\, v \wedge \phi\, e_3 \\
&= \lambda e\phi.\, \exists xyuv.\, \mathbf{man}\, x \wedge \mathbf{Porsche}\, y \wedge u = y \wedge v = x \wedge \mathbf{own}\, x\, y \wedge \mathbf{fascinate}\, u\, v \wedge \phi\, e_3 \\
&= \lambda e\phi.\, \exists xy.\, \mathbf{man}\, x \wedge \mathbf{Porsche}\, y \wedge \mathbf{own}\, x\, y \wedge \mathbf{fascinate}\, y\, x \wedge \phi\, e_1
\end{aligned}
$$

where:

$$
\begin{aligned}
e_1 &= (\mathbf{Porsche}\, y) :: y :: (\mathbf{man}\, x) :: x :: e \\
e_2 &= (u = \mathsf{sel}\, e_1) :: u :: e_1 \\
e_3 &= (v = \mathsf{sel}\, e_2) :: v :: e_2
\end{aligned}
$$

We have used Kamp and Reyle's construction rules to compute a dynamic proposition in de Groote's TTDL. The first line shows the result as we get it from the handler. On the second line, we pull out all of the existential quantifiers (prenex form). On the third line, we resolve the anaphora and we get a dynamic proposition which corresponds, referent by referent, condition by condition, to the DRS derived by Kamp and Reyle. The last line removes spurious variables and equalities.

7.2.3 Negation

In order to match the empirical coverage of the original TTDL from [38], we have two more features to cover: negation and (universal) quantification. We will cover negation here. Quantification reuses the idea from negation and the `scope` operation of quantification: we will see how the two connect in Chapter 8.

The construction rule for negation, CR.NEG, is displayed in Figure 7.4. We are reducing a condition which is in the form of a negated sentence. The DRT mechanism will place the sentence in its own DRS, whose negation will then become the new condition that replaces the sentence. The dynamic effects that we have seen so far reach only into the nearest enclosing DRS. This means that wrapping something inside a new DRS is significant: the reach of any dynamic effects within will be limited to that DRS. We have also seen that this is akin to the way effects like `shift` are delimited/handled by the nearest `reset`. It will come as no surprise then that our implementation of negation will use a handler for dynamic effects.

The definition of dynamic negation in TTDL from [80] will be a great guide in how to proceed:

[131] Since the handler acts as something that provides and regulates access to a (left) TTDL context.

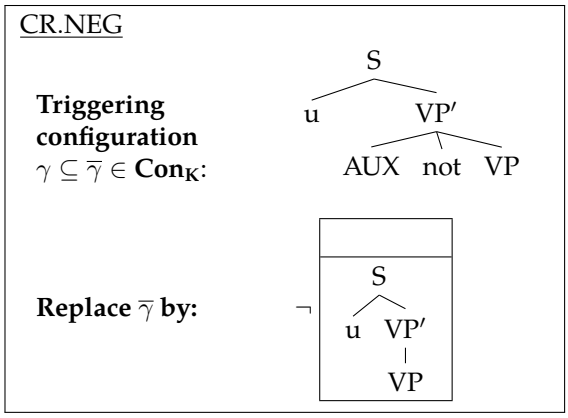


Figure 7.4: CR.NEG: The construction rule for negation.

$$\underset{\mathsf{TTDL}}{\neg} : \Omega \to \Omega$$
$$\underset{\mathsf{TTDL}}{\neg} A = \lambda e\phi.\, \neg(A\, e\, (\lambda e.\, \top)) \wedge \phi\, e$$

We know that $\lambda e\phi.\, M \wedge \phi\, e$ corresponds to $\eta\, M$ and that $A\, e\, (\lambda e.\, \top)$ is $\mathsf{box}\, A\, e$. Therefore, we can define our dynamic negation by:

$$\neg\_ : \mathcal{F}_E(o) \to \mathcal{F}_E(o)$$
$$\neg A = \eta\, (\neg(\mathsf{box}\, A\, e))$$

However, there is a small catch. We have a free variable e of type γ. This is supposed to be the context in which the new negated condition is to appear. This is necessary so that the anaphoric elements within the negated condition can refer to not only the referents proper to the negated DRS, but also to those originating in superordinate DRSs. We can introduce a new operation for accessing the context.

$$\mathtt{get} : 1 \rightarrowtail \gamma$$

Now, we can have dynamic negation as:

$$\neg A = \mathtt{get} \star (\lambda e.\, \eta\, (\neg(\mathsf{box}\, A\, e)))$$

We can see the analogy with CR.NEG: the negation of A puts A inside a box, the box is then negated and returned as the new condition. The name box for this kind of handler was motivated exactly by this kind of analogy, wherein a handler is used to contain dynamic effects inside some scope.

We have introduced a new operation into our dynamic effect signature. This means that we will have to extend our handlers to cover the new operation. However, before we do so, we note that `choose` can be expressed in terms of `get` and sel:

$$\mathtt{choose} = \lambda\_k.\, \mathtt{get} \star (\lambda e.\, k\, (\mathsf{sel}\, e))$$

Therefore, we will drop `choose` and keep only `get`. We have now arrived at our final effect signature for DRT dynamics. This signature allows us to treat dynamic effects the same way as the other effects that we have analyzed in Chapter 6.

$$\begin{aligned} E_{\mathrm{DRT}} = \{\ & \mathtt{get} : 1 \rightarrowtail \gamma, \\ & \mathtt{introduce} : 1 \rightarrowtail \iota, \\ & \mathtt{assert} : o \rightarrowtail 1\ \} \end{aligned}$$

The closed handlers TTDL and box are updated to use `get` instead of `choose`:

$$
\begin{aligned}
\mathsf{TTDL} &: \mathcal{F}_{E_{\mathrm{DRT}}}(o) \to \gamma \to (\gamma \to o) \to o \\
\mathsf{TTDL} &= \llparenthesis\ \texttt{get}\colon (\lambda\_ke\phi.\, k\, e\, e\, \phi), \\
&\quad\ \texttt{introduce}\colon (\lambda\_ke\phi.\, \exists x.\, k\, x\, (x :: e)\, \phi), \\
&\quad\ \texttt{assert}\colon (\lambda pke\phi.\, p \wedge k \star (p :: e)\, \phi), \\
&\quad\ \eta\colon (\lambda pe\phi.\, p \wedge \phi\, e)\ \rrparenthesis
\end{aligned}
$$

$$
\begin{aligned}
\mathsf{box} &: \mathcal{F}_{E_{\mathrm{DRT}}}(o) \to \gamma \to o \\
\mathsf{box} &= \llparenthesis\ \texttt{get}\colon (\lambda\_ke.\, k\, e\, e), \\
&\quad\ \texttt{introduce}\colon (\lambda\_ke.\, \exists x.\, k\, x\, (x :: e)), \\
&\quad\ \texttt{assert}\colon (\lambda pke.\, p \wedge k \star (p :: e)), \\
&\quad\ \eta\colon (\lambda pe.\, p)\ \rrparenthesis
\end{aligned}
$$

7.2.4 Truth Conditions as Side Effects

Let us look back the computation that we assigned to the first sentence in Example 19: *a man owns a Porsche*.

$$
\begin{aligned}
&\texttt{introduce} \star (\lambda x. \\
&\texttt{assert}\,(\mathbf{man}\, x)\,(\lambda\_. \\
&\texttt{introduce} \star (\lambda y. \\
&\texttt{assert}\,(\mathbf{Porsche}\, y)\,(\lambda\_. \\
&\eta\,(\mathbf{own}\, x\, y)))))
\end{aligned}
$$

The truth condition that corresponds to the verb is being returned by the computation, whereas the truth conditions that arise from the nouns are expressed using `assert`. The at-issue content of this sentence consists of all of these conditions and therefore there is no reason to separate out the verb's predicate. We could write this instead:

$$
\begin{aligned}
&\texttt{introduce} \star (\lambda x. \\
&\texttt{assert}\,(\mathbf{man}\, x)\,(\lambda\_. \\
&\texttt{introduce} \star (\lambda y. \\
&\texttt{assert}\,(\mathbf{Porsche}\, y)\,(\lambda\_. \\
&\texttt{assert}\,(\mathbf{own}\, x\, y)\,(\lambda\_. \\
&\eta \star)))))
\end{aligned}
$$

This is a computation of type $\mathcal{F}_{E_{\mathrm{DRT}}}(1)$. As before, we use a handler to extract the proposition that is represented by this computation.

$$
\begin{aligned}
\mathsf{TTDL} &: \mathcal{F}_{E_{\mathrm{DRT}}}(1) \to \gamma \to (\gamma \to o) \to o \\
\mathsf{TTDL} &= \llparenthesis\ \texttt{choose}\colon (\lambda\_ke\phi.\, k\, (\mathsf{sel}\, e)\, e\, \phi), \\
&\quad\ \texttt{introduce}\colon (\lambda\_ke\phi.\, \exists x.\, k\, x\, (x :: e)\, \phi), \\
&\quad\ \texttt{assert}\colon (\lambda pke\phi.\, p \wedge k \star (p :: e)\, \phi), \\
&\quad\ \eta\colon (\lambda\_e\phi.\, \phi\, e)\ \rrparenthesis
\end{aligned}
$$

The handler differs from the original TTDL only in the η clause, which substitutes the dynamic tautology $\top$ for the returned proposition p in $\lambda e\phi.\, p \wedge \phi\, e$.

We can use computations of type $\mathcal{F}_{E_{\text{DRT}}}(1)$ as our type of dynamic (i.e. effectful) propositions. We can embed simple propositions using `assert!`, which has type $o \to \mathcal{F}_{E_{\text{DRT}}}(1)$. We can perform dynamic conjunction $M \overline{\wedge} N$ by chaining two computations, $M \gg\!= (\lambda\_.\, N) : \mathcal{F}_{E_{\text{DRT}}}(1)$ for $M, N : \mathcal{F}_{E_{\text{DRT}}}(1)$. We can modify box the same way we modified TTDL. Dynamic negation will be the same as before, with η being replaced with `assert!`.

$$
\begin{aligned}
\mathsf{box} &: \mathcal{F}_{E_{\text{DRT}}}(1) \to \gamma \to o \\
\mathsf{box} &= \llparenthesis\ \mathtt{get}\colon (\lambda\_ke.\, k\, e\, e), \\
&\qquad \mathtt{introduce}\colon (\lambda\_ke.\, \exists x.\, k\, x\, (x :: e)), \\
&\qquad \mathtt{assert}\colon (\lambda pke.\, p \wedge k \star (p :: e)), \\
&\qquad \eta\colon (\lambda\_e.\, \top)\ \rrparenthesis
\end{aligned}
$$

$$
\overline{\neg} A = \mathtt{get} \star (\lambda e.\, \mathtt{assert!}\, (\neg(\mathsf{box}\, A\, e)))
$$

In the lexical entries derived from the construction rules, not much will have to change. We will only see a difference in the lexical entry TRANS that we have introduced for the CR.LITV rule for transitive verbs (which was extrapolated from the DRT treatment in [64]). Instead of returning the proposition as the result of evaluating the sentence, we add it as a condition using `assert!`, which is actually what the DRS construction algorithm ends up doing.

$$
\begin{aligned}
\textsc{trans} &: V \multimap NP \multimap NP \multimap S \\
[\![\textsc{trans}]\!] &= \lambda\alpha Y X.\, (\alpha \cdot\!\gg X \ll\!\cdot\!\gg Y) \gg\!= \mathtt{assert!}
\end{aligned}
$$

This means that we can build up our dynamic semantics using computations that do not return anything, only modify the context using side effects. This is very much in the spirit of dynamic semantics: meaning is context change potential [101].[132]

The switch from $\mathcal{F}_E(o)$ to $\mathcal{F}_{E \uplus DRT}(1)$ is very much like the one from $\mathcal{F}_E((\iota \to o) \to o)$ to $\mathcal{F}_{E \uplus \mathsf{scope}}(\iota)$: we found a way to encode quantifiers/truth conditions as side effects and so we move the extra structure from the return type to the effect signature. The choice between the two is rather arbitrary. Keeping o as the return type for computations that interpret sentences is more consistent with what we have been doing in Chapter 6. On the other hand, using 1 as the return type has several advantages:

- it is more uniform by not admitting both $\eta\, A$ and $\mathtt{assert}\, A\, (\lambda\_.\, \eta\, \top)$
- this in turn leads to nicer canonical representations and equational theory in 7.2.5
- and in order to correctly treat the cancellation of presuppositions in 7.3, we will need to add the truth conditions generated by verbs into the context using something like `assert` anyway

For the remainder of this chapter, we will use the encoding that models sentences as computations of return type 1.

7.2.5 Algebraic Considerations

Now that we have handlers for our computations, we can study which computations share the same interpretations and try to derive some equational theory over them.

We will first start by looking at how `get` behaves w.r.t. itself and the other operations.

[132] When studying the monadic structure of TTDL in 5.4.2, we have found that dynamic propositions correspond to monadic computations with the trivial result type 1.

$$\begin{aligned}\texttt{get} \star (\lambda e.\, \texttt{get} \star (\lambda e'.\, M(e, e'))) &= \texttt{get} \star (\lambda e.\, M(e, e)) \\ M &= \texttt{get} \star (\lambda e.\, M)\end{aligned}$$

As with the speaker getter from 6.2, we get two laws telling us that asking for the context is idempotent (first equation) and that it has no other bearing on the result of the computation (second equation).

Since we know how assert and introduce modify the context, we can reorder get w.r.t. these two operations:

$$\begin{aligned}\texttt{assert}\, A\, (\lambda u.\, \texttt{get} \star (\lambda e.\, M(u, e))) &= \texttt{get} \star (\lambda e.\, \texttt{assert}\, A\, (\lambda u.\, M(u, A :: e))) \\ \texttt{introduce} \star (\lambda x.\, \texttt{get} \star (\lambda e.\, M(x, e))) &= \texttt{get} \star (\lambda e.\, \texttt{introduce} \star (\lambda x.\, M(x, x :: e)))\end{aligned}$$

This means we can assume that every computation of type $\mathcal{F}_{E_{\text{DRT}}}(1)$ uses get exactly once and does so at the very beginning, i.e. it is of the form $\texttt{get} \star (\lambda e.\, M(e))$ where $M(e) : \mathcal{F}_{\{\texttt{implicate},\texttt{assert}\}}(1)$.[133]

If we assume that the relative order of discourse referents and conditions is not important (e.g. they are both separate parts, as in a DRS), or in other words we have that $(x :: p :: e) = (p :: x :: e)$ for every $x : \iota$, $p : o$ and $e : \gamma$, then we get the following equation:

$$\texttt{assert}\, A\, (\lambda u.\, \texttt{introduce} \star (\lambda x.\, M(u, x))) = \texttt{introduce} \star (\lambda x.\, \texttt{assert}\, A\, (\lambda u.\, M(u, x)))$$

This will allows us to move introduce operations above assert operations so that we can have the following canonical representation for computations of type $\mathcal{F}_{E_{\text{DRT}}}(1)$:

$$\begin{aligned}&\texttt{get} \star (\lambda e. \\ &\texttt{introduce} \star (\lambda x_1. \\ &\quad\vdots \\ &\texttt{introduce} \star (\lambda x_n. \\ &\texttt{assert}\, c_1\, (\lambda\_. \\ &\quad\vdots \\ &\texttt{assert}\, c_m\, (\lambda\_. \\ &\eta \star)))))\end{aligned}$$

In other words, the computation examines its context e and then produces the DRS:

$$\begin{array}{|c|}\hline x_1 \ldots x_n \\ \hline c_1 \\ \vdots \\ c_m \\ \hline\end{array}$$

Note that as in TTDL, discourse referents correspond to λ-binders in ⦇ λ ⦈ and their standard notion of α-equivalence gives rise to α-equivalence for our representations.

If we were to further assume that the order in contexts does not matter at all (e.g. the discourse referents and conditions form sets, as in a DRS), meaning that we have $(x :: y :: e) = (y :: x :: e)$ and $(p :: q :: e) = (q :: p :: e)$ for every $x, y : \iota$, $p, q : o$ and $e : \gamma$, then we get the following:

[133] When writing effect signatures in subscripts, we will often omit the types of operations, which are presumed to be constant.

$$\texttt{assert}\, A\, (\lambda u_1.\, \texttt{assert}\, B\, (\lambda u_2.\, M(u_1, u_2))) = \texttt{assert}\, B\, (\lambda u_2.\, \texttt{assert}\, A\, (\lambda u_1.\, M(u_1, u_2)))$$
$$\texttt{introduce} \star (\lambda x.\, \texttt{introduce} \star (\lambda y.\, M(x, y))) = \texttt{introduce} \star (\lambda y.\, \texttt{introduce} \star (\lambda x.\, M(x, y)))$$

Since the discourse referents and conditions are (unordered) sets in the presentation in [64], this would make our representation closer to DRSs.

Furthermore, if we assume that the conditions in a context can be seen as a big conjunction, i.e. $(p :: q :: e) = ((p \wedge q) :: e)$ for every $p, q : o$ and $e : \gamma$, then we admit the following equations:

$$\texttt{assert}\, A\, (\lambda u.\, \texttt{assert}\, B\, (\lambda u'.\, M(u, u'))) = \texttt{assert}\, (A \wedge B)\, (\lambda u.\, M(u, u))$$

which gives us a simpler canonical representation:

$$\begin{array}{l}
\texttt{get} \star (\lambda e. \\
\texttt{introduce} \star (\lambda x_1. \\
\quad \vdots \\
\texttt{introduce} \star (\lambda x_n. \\
\texttt{assert}\, p\, (\lambda _. \\
\eta \star))))
\end{array}$$

This boils down a computation of type $\mathcal{F}_{E_{\text{DRT}}}(1)$ into a proposition p that depends on some context e and introduces the new discourse referents x_1, ..., x_n.

Finally, we will look at the particular case of the computation that serves as the denotation of an anaphoric pronoun:

$$\begin{array}{l}
\texttt{get} \star (\lambda e. \\
\texttt{introduce} \star (\lambda y. \\
\texttt{assert}\, (\mathsf{sel}\, e = y)\, (\lambda _. \\
\eta\, y)))
\end{array}
=
\begin{array}{l}
\texttt{get} \star (\lambda e. \\
\texttt{introduce} \star (\lambda y. \\
\texttt{assert}\, (\mathsf{sel}\, e = y)\, (\lambda _. \\
\eta\, (\mathsf{sel}\, e))))
\end{array}
=
\begin{array}{l}
\texttt{get} \star (\lambda e. \\
\eta\, (\mathsf{sel}\, e))
\end{array}$$

The first equation, which exchanges y with $\mathsf{sel}\, e$, is licensed by the fact that the box handler will interpret the first computation as the proposition $\exists y.\, \mathsf{sel}\, e = y \wedge k\, y$ and the second computation as the proposition $\exists y.\, \mathsf{sel}\, e = y \wedge k\, (\mathsf{sel}\, e)$, both of which are equivalent.

The simplification in the second equation is based on an assumption that $((\mathsf{sel}\, e = y) :: y :: e) = e$. In other words, if an individual is already present in the context (the individual $\mathsf{sel}\, e$), adding them again under a different name (y) does not change the context. While certain strategies of anaphora resolution might rely on individuals being repeatedly added to the context on every use,[134] we allow ourselves this modification to simplify the derivations in the coming examples and make the resulting logical formulas more readable. Therefore, we will be using this simplified denotation for anaphoric pronouns:

$$\begin{array}{l}
[\![\text{HE}]\!] = \texttt{get} \star (\lambda e. \\
\qquad \eta\, (\mathsf{sel}\, e))
\end{array}$$

7.3 Presuppositions

In our $(\!|\, \lambda \,|\!)$ analysis of anaphora, we have completely omitted proper nouns, even though they are treated by DRT in [64] and feature in Example 6. This is due to the fact that proper nouns trigger presuppositions:

[134] For example when contexts are lists of individuals ordered by saliency and adding the same individual multiple times increases their saliency.

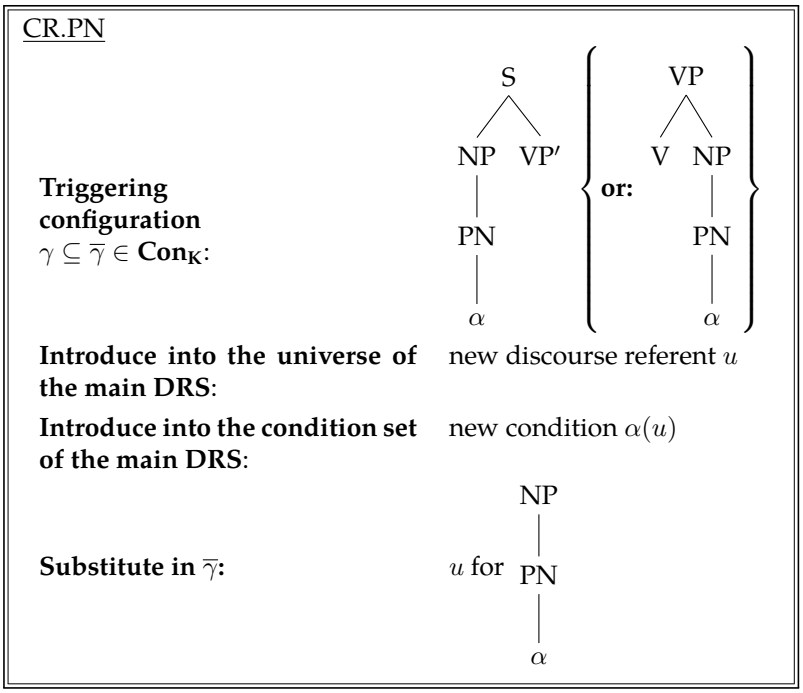


Figure 7.5: CR.PN: The construction rule for proper nouns.

Jones sleeps presupposes that there is some (contextually salient) entity named Jones. These presuppositions project outside of entailment-cancelling operators such as negation, outside of the DRSs in which they are contained.

(20) It is not the case that Jones$_1$ owns a Porsche. He$_1$ owns a Mercedes.

In Example 20, the proper noun *Jones* contributes a discourse referent which is then picked up in the second sentence. This would be impossible to achieve using `introduce`, since that would contribute the referent only to the DRS (box) that is being negated and the discourse referent would not reach the second sentence.

If we look at the construction rule for proper nouns in DRT, CR.PN, displayed on Figure 7.5, we will see that the discourse referent and the condition describing it are being inserted into the main DRS and not into the DRS K in which the condition being reduced appears.

In TTDL, this issue was resolved by adding exceptions into the lambda calculus used for the semantic terms [80]. The lexical entry for a presupposition trigger such as a proper noun or a definite description throws the exception `AbsentIndividualExc` P. This exception carries the predicate P that describes the entity that is presupposed to exist. At the top level, a handler catches these exceptions and accommodates these presuppositions. We have seen that in ⦇ λ ⦈, dynamic propositions are modelled as computations and these computations already have a notion of throwing exceptions (effects) and handling them (handlers). We will introduce an operation named `presuppose`. Its input type will be $\iota \to o$, a predicate that describes the entity whose existence is presupposed. The output type will be ι, the presupposed entity satisfying the predicate.

$$\texttt{presuppose} : (\iota \to o) \rightarrowtail \iota$$

The type of this operation is not exactly equivalent to the exception `AbsentIndividualExc` from [80]. As we have seen in Chapter 2.2.3, exceptions are effects that have the impossible output type 0, which means that no handler will be able to resume the computation by using the continuation. In Lebedeva's approach, when the toplevel handler intercepts the exception, it accommodates the presuppo-

sition, rewinds the dynamic state and evaluates the sentence again, this time with a context which now contains the presupposed entity. This is not suitable to our approach because of two reasons:

- We will deal with many other effects besides dynamic state and rewinding all of their effects in order to evaluate the sentence again in a new context would be needlessly complex.

- This approach over-generates by licencing the reading "He$_1$ loves John's$_1$ car". When evaluating *John*, the existence of John is presupposed and the sentence is evaluated again, this time in a context in which John is accessible. However, this will allow the pronoun in subject position to resolve to John and end up being bound by the object.

The effects in $(\!| \lambda |\!)$ differ from traditional exceptions in that they are resumable. Besides having the input type (which is the message type of a traditional exception), they also have an output type. Handlers can then resume computation at the point of the effect by calling the continuation with a value of the output type. We make full use of this in our treatment of presuppositions by using ι as the output type of `presuppose`. This way, we do not have to abort the evaluation of the sentence, rewind all of the effects and evaluate it again; we return the presupposed entity immediately and the evaluation of the sentence continues. This strategy also avoids the over-generation mentioned above, in which a presupposition trigger binds a pronoun preceding it.

We have shown how the `presuppose` operation can be motivated on the grounds of the analysis done by Lebedeva in [80]. We can also look at how it ties to the DRT construction rule for presupposition triggers such as CR.PN. When translating construction rules into computations, we have implemented the **Introduce in U**$_K$ command as the `introduce` operation and the **Introduce in Con**$_K$ command as the `assert` operation. We might therefore proceed the same way and translate the **Introduce into the universe of the main DRS** command as some operation `introduce_main` $: \star \rightarrowtail \iota$ and **Introduce into the condition set of the main DRS** command as `assert_main` $: o \rightarrowtail 1$. However, by looking at the construction rules of presupposition triggers, we see that they tend to use the two operations in tandem: first introduce a new discourse referent and then some condition(s) describing it. Therefore, we can fuse the two into a single operation `presuppose` $: (\iota \rightarrow o) \rightarrowtail \iota$, which introduces and outputs a new discourse referent x while at the same it introduces the condition $P\,x$, where P is its argument.

Besides lowering the number of basic operations that we have to introduce, this also has an advantage when we start to consider ambiguous ways of accommodating presuppositions (7.3.4). Suppose we have two operations, `introduce_somewhere` and `assert_somewhere`, that let us introduce discourse referents and conditions into arbitrary DRSs on the projection line.[135] We might then end up accommodating the discourse referent and the condition in different DRS, yielding an undesired meaning. If we package the two operations into a single `presuppose_somewhere`, then we do not have this problem.

7.3.1 Revising the Dynamic Handler

We will want to introduce the `presuppose` operation into our dynamic semantics in such a way that `presuppose` projects outside of boxes (DRSs, applications of the `box` handler). This means that applying the `box` handler to a computation should yield a computation free of `get`, `introduce` and `assert` but still possibly using `presuppose`: `box` will be an open handler for `get`, `introduce` and `assert`. The open handler will be more complex, since where before we had continuations which returned simple propositions, or rather functions from contexts to propositions, we will now have continuations which return computations.

[135] A *projection line* is a path in the immediate-subordination tree from a sub-DRS to the root DRS [130]. See the definition of immediate subordination in 5.3.3.

$$
\begin{aligned}
&\mathsf{box} : \mathcal{F}_{E_{\mathrm{DRT}}}(1) \to \gamma \to o \\
&\mathsf{box} = \llparenthesis\, \texttt{get}\colon (\lambda\_ke.\, k\, e\, e), \\
&\qquad\quad \texttt{introduce}\colon (\lambda\_ke.\, \exists x.\, k\, x\, (x :: e)), \\
&\qquad\quad \texttt{assert}\colon (\lambda pke.\, p \wedge k \star (p :: e)), \\
&\qquad\quad \eta\colon (\lambda\_e.\, \top)\, \rrparenthesis
\end{aligned}
$$

$$
\begin{aligned}
&\mathsf{box} :\ \mathcal{F}_{E \uplus E_{\mathrm{DRT}}}(1) \to \gamma \to \mathcal{F}_E(o) \\
&\mathsf{box} = \lambda Ae.\, (\llparenthesis\, \texttt{get}\colon (\lambda\_k.\, \eta\, (\lambda e.\, k\, e \lll\cdot e)), \\
&\qquad\qquad\qquad \texttt{introduce}\colon (\lambda\_k.\, \eta\, (\lambda e.\, \exists_{\gg} x.\, k\, x \lll\cdot (x :: e))), \\
&\qquad\qquad\qquad \texttt{assert}\colon (\lambda pk.\, \eta\, (\lambda e.\, p \wedge\!\gg (k \star \lll\cdot (p :: e)))), \\
&\qquad\qquad\qquad \eta\colon (\lambda\_.\, \eta\, (\lambda e.\, \top))\, \rrparenthesis\, A) \lll\cdot e
\end{aligned}
$$

$$
\begin{aligned}
\_ \lll\cdot \_ &: \mathcal{F}_E(\alpha \to \mathcal{F}_E(\beta)) \to \alpha \to \mathcal{F}_E(\beta) \\
F \lll\cdot x &= F \gg\!= (\lambda f.\, f\, x) \\
\exists_{\gg} &: (\iota \to \mathcal{F}_E(o)) \to \mathcal{F}_E(o) \\
\exists_{\gg} P &= \exists \cdot\!\gg (\mathcal{C}\, P)
\end{aligned}
$$

We include the closed box handler for comparison. At the heart of the open box handler, we have a ⦇ ⦈ of type $\mathcal{F}_{E \uplus E_{\mathrm{DRT}}}(1) \to \mathcal{F}_E(\gamma \to \mathcal{F}_E(o))$. This ⦇ ⦈ differs from the closed ⦃⦄ in the following ways:

- Since the interpretations are no longer functions (type $\gamma \to o$), but computations that produce functions (type $\mathcal{F}_E(\gamma \to \mathcal{F}_E(o))$), we start all the clauses with $\eta\, (\lambda e.\, \ldots)$ instead of $\lambda e.\, \ldots$
- When we apply the continuation k to the output (the context e in `get`, the referent x in `introduce` and the trivial $\star$ in `assert`), we now get a computation (type $\mathcal{F}_E(\gamma \to \mathcal{F}_E(o))$) instead of a function (type $\gamma \to o$). We need to first evaluate the computation to get a function and then apply that function to the new context. For that purpose, we define the $\lll\cdot$ combinator below.
- The proposition that is the result of applying the continuation to the output and to the new context is now a computation (type $\mathcal{F}_E(o)$) instead of a simple proposition (type o). In the `assert` clause, if we want to conjoin a proposition p to this proposition, we need to use the $\wedge\!\gg$ operator, which takes a computation as its right argument. In the `introduce` clause, we have an impure predicate $\lambda x.\, k\, x \lll\cdot (x :: e)$ of type $\iota \to \mathcal{F}_E(o)$. We use $\mathcal{C}$ to push the effects out and get a computation that returns a pure predicate, type $\mathcal{F}_E(\iota \to o)$. We can then apply the existential quantifier under the $\mathcal{F}_E$ type wrapper using $\cdot\!\gg$. We write this way of applying the existential quantifier to an effectful predicate P as $\exists_{\gg} P$.

After having interpreted $A : \mathcal{F}_{E \uplus E_{\mathrm{DRT}}}(1)$ using the ⦇ ⦈, we get a computation of type $\mathcal{F}_E(\gamma \to \mathcal{F}_E(o))$. In order to apply it to an $e : \gamma$, we use the $\lll\cdot$ combinator one last time.

The handler above is much more complicated than any we have seen so far in this manuscript. However, note that:

- The translation from the closed box handler was mechanical. The open box handler does not include any ad-hoc features built into it to make it compatible with presuppositions. The only changes we made to the handler was adding η and $\mathcal{C}$ and replacing operators with versions that can act on computations ($\cdot\!\gg$ and $\lll\cdot$ for application, $\wedge\!\gg$ for conjunction). In an impure calculus with call-by-value semantics in which a distinction between the types α and $\mathcal{F}_E(\alpha)$ does not exist, neither η nor the $\ll\_\gg$ operators would be needed; the only operator we would need to add would be $\mathcal{C}$, which pushes effects out of a function body.

- The cost of replacing the simpler closed handler with the open handler is a price we would have to pay anyway if we wanted to use this handler in a setting that involved other effects (such as quantification, deixis, conventional implicature); it is not limited to adding support for presuppositions. E.g., in Chapter 8, we will deal almost exclusively with open handlers.

- The box handlers are by far the largest handlers in our analyses. Furthermore, open handlers that thread some mutable state throughout the computation are slightly awkward to write (as testified by all the uses of $\lll\cdot$) in pure languages without using some syntactic sugar, such as parameterized handlers that can automatically thread some parameter from operation to operation as in [63, 70].

The open box handler outputs computations of propositions instead of simple propositions, which is something we have to take into account when using it in, e.g., negation:

$$\neg A = \texttt{get} \star (\lambda e.\, \mathsf{box}\, A\, e \gg\!= (\lambda a.\, \texttt{assert!}\, (\neg a)))$$

Now that we have an open version of the box handler, there is one change that we will make in how it actually functions. We will motivate it by the following example.

(21) It is not the case that John$_1$ likes his$_1$ car.

The negation will use get to retrieve the current context e and then proceed by evaluating the expression $(\mathsf{box}\, [\![\text{John likes his car}]\!]\, e)$. The proper noun *John* will then use presuppose, which will project out of the box and introduce John into the global context. But now we have to evaluate the pronoun *his* in the context e which was untouched by *John* and we will thus fail to retrieve John and get the expected reading. The problem is in our definition of $\neg A$. We want anaphoric expressions within A to have access to referents introduced outside of A. We do this by using get to recover the context e in which we are about to evaluate $\neg A$ and then use that context as the initial context when interpreting anaphora in A with $\mathsf{box}\, A\, e$. While this works well for TTDL, in which dynamic propositions have no way to modify contexts other than the local one, it breaks when we let presuppose modify the top context.

The solution is to dismiss the assumption that surrounding contexts are immutable. Whenever we handle a get, we use another get to gather the current state of the surrounding (global) context and then combine that with the local context. This solution will actually simplify the definition of $\neg$ and the type of box, since box does not need to be seeded with its surrounding context but retrieves it itself when necessary. Furthermore, note that this solution was not available to us before we made the handler open, since it interprets computations of type $\mathcal{F}_{E_{\mathrm{DRT}}}(1)$ as computations of type $\mathcal{F}_{\{\texttt{get}\}}(o)$.

$$\begin{aligned}
&\mathsf{box} : \mathcal{F}_{E \uplus E_{\mathrm{DRT}}}(1) \to \mathcal{F}_{E \uplus \{\texttt{get}\}}(o)\\
&\mathsf{box} = \lambda A.\, (\!|\, \texttt{get}\colon (\lambda_k.\, \eta\, (\lambda e.\, \texttt{get} \star (\lambda e'.\, k\, (e +\!\!+ e') \lll\cdot e))),\\
&\qquad\qquad \texttt{introduce}\colon (\lambda_k.\, \eta\, (\lambda e.\, \exists_{\gg}\, x.\, k\, x \lll\cdot (x :: e))),\\
&\qquad\qquad \texttt{assert}\colon (\lambda pk.\, \eta\, (\lambda e.\, p \wedge\!\gg (k \star \lll\cdot (p :: e)))),\\
&\qquad\qquad \eta\colon (\lambda_.\, \eta\, (\lambda e.\, \top))\, |\!)\, A) \lll\cdot \mathsf{nil}
\end{aligned}$$

The get now uses another get to retrieve the current surrounding context e', which is then combined with the local context e as the answer to the original get. In order to combine the two contexts, we assume that there is some operation $(+\!\!+) : \gamma \to \gamma \to \gamma$. If contexts are (pairs of) lists, then this operation would be (pointwise) list concatenation. Since we always ask again for the surrounding context, we do not need the surrounding context as an argument to box. Instead, we pass in nil as the initial local context, where $\mathsf{nil} : \gamma$ is a constant representing the empty context, e.g. an empty list.

As we no longer need to supply the initial surrounding context, dynamic negation becomes simpler:

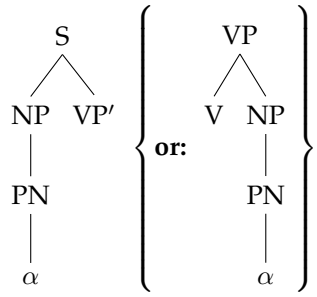
$$\neg A = \mathsf{box}\, A \gg\!= (\lambda a.\, \texttt{assert!}\, (\neg a))$$

7.3.2 Presupposition in Action

With `presuppose` in place, we can now translate the DRT construction rule for proper nouns, CR.PN, into ⦇ λ ⦈:

<u>CR.PN</u>

Triggering configuration $\gamma \subseteq \overline{\gamma} \in \mathbf{Con_K}$:

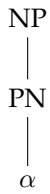


$$\textsc{proper} : PN \multimap NP$$
$$[\![PN]\!] = \iota \to o$$
$$[\![NP]\!] = \mathcal{F}_E(\iota)$$

Introduce into the universe of the main DRS: new discourse referent u

Introduce into the condition set of the main DRS: new condition $\alpha(u)$

Substitute in $\overline{\gamma}$: u for



$$[\![\textsc{proper}]\!] = \lambda\alpha.\, \texttt{presuppose}\ (\lambda u.\, \alpha\, u)\ (\lambda u.\, \eta\, u)$$

We model the introduction of a discourse referent and a condition into the main DRS with the `presuppose` operation. As with the lexical insertion rules for common nouns and transitive verbs, this rule is parameterized by a lexical item, this time a proper noun. In DRT, proper nouns are described using predicates and so we introduce an abstract atomic type PN for proper nouns whose interpretation $[\![PN]\!]$ is the type of predicates. The definition of PROPER above is expanded in order to parallel the construction. We can actually make it a lot shorter:

$$\begin{aligned}
[\![\textsc{proper}]\!] &= \lambda\alpha.\, \texttt{presuppose}\ (\lambda u.\, \alpha\, u)\ (\lambda u.\, \eta\, u) \\
&\to_\eta \lambda\alpha.\, \texttt{presuppose}\ \alpha\ (\lambda u.\, \eta\, u) \\
&= \lambda\alpha.\, \texttt{presuppose!}\ \alpha \\
&\to_\eta \texttt{presuppose!}
\end{aligned}$$

We will also need a handler to give a meaning to the `presuppose` operation. The meaning behind `presuppose` is clear: it will introduce a discourse referent and a condition.

$$\begin{aligned}
&\mathsf{accommodate} : \mathcal{F}_{E \uplus \{\texttt{presuppose}\}}(\alpha) \to \mathcal{F}_E(\alpha) \\
&\mathsf{accommodate} = ⦇\texttt{presuppose}\colon (\lambda Pk.\, \texttt{introduce} \star (\lambda x.\, \texttt{assert}\ (P\, x)\ (\lambda_.\, k\, x)))⦈
\end{aligned}$$

The accommodate handler turns presupposed content into asserted content. The place we want to do that is usually on the level of the topmost DRS. We can define a combinator which corresponds to the idea of a top DRS.

$$\begin{aligned}
&\mathsf{top} : \mathcal{F}_{E \uplus E_{\mathrm{DRT}} \uplus \{\texttt{presuppose}\}}(1) \to \mathcal{F}_{E \uplus \{\texttt{get}\}}(o) \\
&\mathsf{top} = \mathsf{box} \circ \mathsf{accommodate}
\end{aligned}$$

We can now deal with Example 20. We start by computing the denotation of the first sentence.

$$
\begin{aligned}
[\![S_1]\!] &= [\![\textsc{trans owns}\ (\textsc{a}\ (\textsc{common}\ \textsc{Porsche}))\ (\textsc{proper}\ \textsc{Jones})]\!] \\
&\twoheadrightarrow \texttt{presuppose}\ \textbf{Jones}\ (\lambda x. \\
&\quad \texttt{introduce}\ \star\ (\lambda y. \\
&\quad \texttt{assert}\ (\textbf{Porsche}\ y)\ (\lambda\_. \\
&\quad \texttt{assert}\ (\textbf{own}\ x\ y)\ (\lambda\_. \\
&\quad \eta\ \star))))
\end{aligned}
$$

Wrapping a dynamic negation over this will resolve all the `introduce` and `assert` operations, but the `presuppose` will prevail.

$$
\begin{aligned}
&\neg\ [\![S_1]\!] \\
&\twoheadrightarrow \mathsf{box}\ [\![S_1]\!] \gg\!= (\lambda a.\ \texttt{assert!}\ (\neg a)) \\
&\twoheadrightarrow \texttt{presuppose}\ \textbf{Jones}\ (\lambda x. \\
&\quad \eta\ (\exists y.\ \textbf{Porsche}\ y \wedge \textbf{own}\ x\ y)) \gg\!= (\lambda a.\ \texttt{assert!}\ (\neg a)) \\
&\twoheadrightarrow \texttt{presuppose}\ \textbf{Jones}\ (\lambda x. \\
&\quad \texttt{assert}\ (\neg(\exists y.\ \textbf{Porsche}\ y \wedge \textbf{own}\ x\ y))\ (\lambda\_. \\
&\quad \eta\ \star))
\end{aligned}
$$

We now turn to the second sentence in Example 20, whose meaning is derived below.

$$
\begin{aligned}
[\![S_2]\!] &= [\![\textsc{trans owns}\ (\textsc{a}\ (\textsc{common}\ \textsc{Mercedes}))\ \textsc{he}]\!] \\
&\twoheadrightarrow \texttt{get}\ (\lambda e. \\
&\quad \texttt{introduce}\ \star\ (\lambda y. \\
&\quad \texttt{assert}\ (\textbf{Mercedes}\ y)\ (\lambda\_. \\
&\quad \texttt{assert}\ (\textbf{own}\ (\mathsf{sel}\ e)\ y)\ (\lambda\_. \\
&\quad \eta\ \star))))
\end{aligned}
$$

Dynamic propositions of type $\mathcal{F}_{E_{\text{DRT}}}(1)$ are conjoined by chaining their computations. By chaining the two computations, we get a meaning for the whole discourse in Example 20.

$$
\begin{aligned}
[\![\text{not}\ S_1.\ S_2]\!] &= (\neg\ [\![S_1]\!]) \gg\!= (\lambda\_.\ [\![S_2]\!]) \\
&\twoheadrightarrow \texttt{presuppose}\ \textbf{Jones}\ (\lambda x. \\
&\quad \texttt{assert}\ (\neg(\exists y.\ \textbf{Porsche}\ y \wedge \textbf{own}\ x\ y))\ (\lambda\_. \\
&\quad \texttt{get}\ (\lambda e. \\
&\quad \texttt{introduce}\ \star\ (\lambda y. \\
&\quad \texttt{assert}\ (\textbf{Mercedes}\ y)\ (\lambda\_. \\
&\quad \texttt{assert}\ (\textbf{own}\ (\mathsf{sel}\ e)\ y)\ (\lambda\_. \\
&\quad \eta\ \star))))))
\end{aligned}
$$

We can now embed this in the top-level box.

$$
\begin{aligned}
\mathsf{top}\,[\![\text{not } S_1.\ S_2]\!] &= \mathsf{box}\,(\mathsf{accommodate}\,[\![\text{not } S_1.\ S_2]\!]) \\
&\twoheadrightarrow \mathsf{box}\,(\ \mathtt{introduce} \star (\lambda x. \\
&\qquad \mathtt{assert}\,(\mathbf{Jones}\,x)\,(\lambda\_. \\
&\qquad \mathtt{assert}\,(\neg(\exists y.\,\mathbf{Porsche}\,y \wedge \mathbf{own}\,x\,y))\,(\lambda\_. \\
&\qquad \mathtt{get}\ (\lambda e. \\
&\qquad \mathtt{introduce} \star (\lambda y. \\
&\qquad \mathtt{assert}\,(\mathbf{Mercedes}\,y)\,(\lambda\_. \\
&\qquad \mathtt{assert}\,(\mathbf{own}\,(\mathsf{sel}\,e)\,y)\,(\lambda\_. \\
&\qquad \eta \star)))))))\ ) \\
&\quad \twoheadrightarrow \mathtt{get} \star (\lambda e. \\
&\qquad \eta\,(\exists x.\,\mathbf{Jones}\,x \wedge \neg(\exists y.\,\mathbf{Porsche}\,y \wedge \mathbf{own}\,x\,y) \wedge (\exists y.\,\mathbf{Mercedes}\,y \wedge \mathbf{own}(\mathsf{sel}(e'), y))))
\end{aligned}
$$

where

$$
e' = \neg(\exists y.\,\mathbf{Porsche}\,y \wedge \mathbf{own}\,x\,y) :: \mathbf{Jones}\,x :: x :: e
$$

By assuming that x is *the* salient antecedent for the pronoun or by evaluating in the empty context nil, we get the intended reading.

$$
\begin{aligned}
&\mathsf{empty} : \mathcal{F}_{E \uplus \{\mathtt{get}\}}(\alpha) \to \mathcal{F}_E(\alpha) \\
&\mathsf{empty} = ⦇\,\mathtt{get}\colon (\lambda\_k.\,k\,\mathsf{nil})\,⦈ \\
&\qquad \mathsf{empty}\,(\mathsf{top}\,[\![\text{not } S_1.\ S_2]\!]) \\
&\qquad\ \twoheadrightarrow \eta\,(\exists x.\,\mathbf{Jones}\,x \wedge \neg(\exists y.\,\mathbf{Porsche}\,y \wedge \mathbf{own}\,x\,y) \wedge (\exists y.\,\mathbf{Mercedes}\,y \wedge \mathbf{own}\,x\,y))
\end{aligned}
$$

7.3.3 Cancelling Presuppositions

It has been observed that not every presupposition projects. A presupposition can be blocked or cancelled in certain contexts. The sentence "John's car is cheap" presupposes that there is someone named John who owns a car. However, if we put this sentence into a conditional context, as in Example 22, the presupposition that John owns a car disappears.

(22) If John owns a car, then his car is cheap.

Lebedeva [80] solves this in TTDL by making the presupposition triggers first search the context for their referent. If the necessary referent is present in an accessible context, then it is retrieved and no presupposition is triggered. We can do the same in our setting. Let $\mathsf{sel_P} : (\iota \to o) \to \gamma \to (\iota + 1)$ be a function that retrieves from a context of type γ the salient individual that satisfies the predicate of type $\iota \to o$, if such an individual exists in the context. Otherwise, it returns inr $\star$.

$$
\begin{aligned}
&\mathsf{find} : (\iota \to o) \to \mathcal{F}_{E \uplus \{\mathtt{get},\mathtt{presuppose}\}}(\iota) \\
&\mathsf{find} = \lambda P.\,\mathtt{get} \star (\lambda e.\,\mathbf{case}\ \mathsf{sel_P}\,P\,e\ \mathbf{of}\ \{\mathsf{inl}\ x \to \eta\,x;\ \mathsf{inr}\ \_ \to \mathtt{presuppose!}\,P\})
\end{aligned}
$$

find examines its context and searches it for an individual satisfying the predicate. If no such individual is found, it triggers a presupposition. The type of find is (almost[136]) the same as that of presuppose! and so we can replace all uses of presuppose! with find to get the correct behavior w.r.t. cancelling of presuppositions.

[136] The type of presuppose! is $(\iota \to o) \to \mathcal{F}_{E \uplus \{\mathtt{presuppose}\}}(\iota)$. However, find relies not only on presuppose, but also on get. In most contexts, both are available and so the two are interchangeable.

$$
\begin{aligned}
\textsc{proper} &: PN \multimap NP \\
[\![\textsc{proper}]\!] &= \mathsf{find} \\
\textsc{poss} &: NP \multimap N \multimap NP \\
[\![\textsc{poss}]\!] &= \lambda XN.\, X \gg\!= (\lambda x.\, N \gg\!= (\lambda n.\, \mathsf{find}\,(\lambda y.\, n\, y \wedge \mathbf{own}\, x\, y)))
\end{aligned}
$$

The possessive construction *X's N* uses find to behave either as a bound pronoun, using $\mathsf{sel_P}$ to retrieve its referent from the context, or if there is no suitable referent, as a presupposition trigger.

The final part of the puzzle is how to make the context of the antecedent (*John owns a car*) available in the consequent (*his car is cheap*). DRT has special accessibility rules for implications, but in Subsection 5.3.4, we have seen how to translate DRT implications into negations and conjunctions while preserving accessibility.[137] We will use the same strategy to correctly deal with accessibility in conditional utterances.

$$
\begin{aligned}
\textsc{if-then} &: S \multimap S \multimap S \\
[\![\textsc{if-then}]\!] &= \lambda AB.\, A \Rightarrow B \\
&= \lambda AB.\, \neg(A \barwedge \neg B) \\
&= \lambda AB.\, \neg(A \gg\!= (\lambda\_.\, \neg B))
\end{aligned}
$$

The dynamic implication used in $[\![\textsc{if-then}]\!]$ explains how conditionals can filter presuppositions. A new box is opened using the outer negation. Into that box, A will introduce new discourse referents and truth conditions. These will then be able available to B, which might resolve some of B's presuppositions. Since the whole conditional is inside a box due to the dynamic negation, the filtering effect will subside after the consequent, i.e. dynamic implication is externally static [53].

In Example 22, the sentence in the antecedent introduces into the context John (x, $\mathbf{John}\, x$), a car (y, $\mathbf{car}\, y$) and the fact that John owns the car ($\mathbf{own}\, x\, y$). The sentence in the consequent then has access to this context and can resolve *his* to x and then retrieve y using $\mathsf{sel_P}$, thus preventing the triggering of a presupposition.

Instead of using find in the lexical entry of every presupposition trigger, there is also an alternative implementation that is more in the spirit of $(\!|\, \lambda \,|\!)$. We can change the semantics of `presuppose` from "introduce a new top-level discourse referent satisfying the predicate" to "find a salient individual satisfying the predicate". If the context can satisfy the `presuppose` operation by binding the presupposition trigger to an existing discourse referent, it might choose to do so. This way, all presupposition triggers consistently use `presuppose`. The cancelling then happens at the level of a box (a DRS) which might decide to satisfy the presupposition by supplying an existing discourse referent.

$$
\begin{aligned}
\mathsf{useFind} &: \mathcal{F}_{E \uplus \{\texttt{presuppose}\}}(\alpha) \to \mathcal{F}_{E \uplus \{\texttt{get},\texttt{presuppose}\}}(\alpha) \\
\mathsf{useFind} &= (\!|\, \texttt{presuppose}\colon (\lambda Pk.\, \mathsf{find}\, P \gg\!= k) \,|\!) \\
\overline{\mathsf{box}} &: \mathcal{F}_{E \uplus E_{\mathrm{DRT}} \uplus \{\texttt{presuppose}\}}(1) \to \mathcal{F}_{E \uplus \{\texttt{get},\texttt{presuppose}\}}(o) \\
\overline{\mathsf{box}} &= \mathsf{box} \circ \mathsf{useFind}
\end{aligned}
$$

We define a handler for `presuppose` called useFind that tries to cancel the presupposition by looking into the current context for a possible referent. In case there is no such referent, the presupposition is projected onwards. Since useFind can be defined on its own, we can define the new box that cancels presuppositions, $\overline{\mathsf{box}}$, by composing the old box with useFind.[138] The useFind handler, which is included in $\overline{\mathsf{box}}$, handles some uses of `presuppose` and projects others. This captures the reality that some contexts cancel some presuppositions. If we use the $\overline{\mathsf{box}}$ handler, we can now use `presuppose` in the lexical entries

[137] The same technique is applied in the TTDL of [80].

[138] We also replace all uses of box with $\overline{\mathsf{box}}$, i.e. the one in dynamic negation.

of presupposition triggers without having to remember to use find in order to consistently get the correct prediction w.r.t. the cancellation of presuppositions.

$$
\begin{aligned}
\textsc{proper} &: PN \multimap NP \\
[\![\textsc{proper}]\!] &= \texttt{presuppose!} \\
\textsc{poss} &: NP \multimap N \multimap NP \\
[\![\textsc{poss}]\!] &= \lambda XN.\, X \gg\!= (\lambda x.\, N \gg\!= (\lambda n.\, \texttt{presuppose!}\, (\lambda y.\, n\, y \wedge \mathbf{own}\, x\, y)))
\end{aligned}
$$

We can now compute the denotation of Example 22. The meaning of Example 22 is equal to $[\![\textsc{if-then}\ S_1\ S_2]\!] = [\![S_1]\!] \Rightarrow [\![S_2]\!] = \neg([\![S_1]\!] \barwedge \neg[\![S_2]\!])$, where S_1 is the antecedent and S_2 is the consequent. We start with the denotation of the consequent, S_2: *his car is cheap*.

$$
\begin{aligned}
[\![S_2]\!] = [\![&\textsc{is-cheap}\ (\textsc{poss}\ \textsc{he}\ (\textsc{common}\ \textsc{car}))]\!] \\
&\twoheadrightarrow \texttt{get} \star (\lambda e. \\
&\quad \texttt{presuppose}\, (\lambda z.\, \mathbf{car}\, z \wedge \mathbf{own}\, (\mathsf{sel}\, e)\, z)\, (\lambda z. \\
&\quad \texttt{assert}\, (\mathbf{cheap}\, z)\, \lambda\_. \\
&\quad \eta \star))
\end{aligned}
$$

Now we will work towards the dynamic negation of $[\![S_2]\!]$, starting with $\overline{\mathsf{box}}\, [\![S_2]\!]$.

$$
\begin{aligned}
\overline{\mathsf{box}}\, [\![S_2]\!] &= \mathsf{box}\, (\mathsf{useFind}\, [\![S_2]\!]) \\
&\twoheadrightarrow \mathsf{box}\, (\texttt{get} \star (\lambda e. \\
&\qquad \texttt{get} \star (\lambda e'. \\
&\qquad (\mathbf{case}\ \mathsf{sel_P}\, (\lambda z.\, \mathbf{car}\, z \wedge \mathbf{own}\, (\mathsf{sel}\, e)\, z)\, e'\ \mathbf{of} \\
&\qquad\quad \{\mathrm{inl}\ z \to \eta\, z; \\
&\qquad\quad\ \ \mathrm{inr}\ \_ \to \texttt{presuppose!}\, (\lambda z.\, \mathbf{car}\, z \wedge \mathbf{own}\, (\mathsf{sel}\, e)\, z)\}) \gg\!= (\lambda z. \\
&\qquad \texttt{assert}\, (\mathbf{cheap}\, z)\, (\lambda\_. \\
&\qquad \eta \star))))) \\
&\approx \texttt{get} \star (\lambda e. \\
&\quad (\mathbf{case}\ \mathsf{sel_P}\, (\lambda z.\, \mathbf{car}\, z \wedge \mathbf{own}\, (\mathsf{sel}\, e)\, z)\, e\ \mathbf{of} \\
&\qquad \{\mathrm{inl}\ z \to \eta\, z; \\
&\qquad\ \ \mathrm{inr}\ \_ \to \texttt{presuppose!}\, (\lambda z.\, \mathbf{car}\, z \wedge \mathbf{own}\, (\mathsf{sel}\, e)\, z)\}) \gg\!= (\lambda z. \\
&\quad \eta\, (\mathbf{cheap}\, z)))
\end{aligned}
$$

We see that the meaning $\overline{\mathsf{box}}\, [\![S_2]\!]$ is anaphoric, it uses `get` to find an antecedent for *his* and *his car*. Furthermore, depending on the retrieved context e, it might trigger a presupposition about the existence of *his car*. In reducing the expression, we have also allowed ourselves to collapse the two successive uses of `get`, since we have derived an equation for dynamic computations that tells us that `get` is idempotent under the handlers we use (see 7.2.5), hence the use of the $\approx$. To finish computing $\neg[\![S_2]\!]$, we take $\overline{\mathsf{box}}\, S_2$, replace the final η with an `assert`! and negate its argument.

$$
\begin{aligned}
\neg[\![S_2]\!] = \overline{\mathsf{box}}\,[\![S_2]\!] &\gg\!= (\lambda a.\, \mathtt{assert!}\,(\neg a)) \\
&\twoheadrightarrow \mathtt{get}\ \star\ (\lambda e. \\
&\quad (\mathbf{case}\ \mathsf{sel_P}\ (\lambda z.\, \mathbf{car}\, z \wedge \mathbf{own}\,(\mathsf{sel}\, e)\, z)\, e\ \mathbf{of} \\
&\qquad \{\mathrm{inl}\ z \to \eta\, z; \\
&\qquad\ \ \mathrm{inr}\ \_ \to \mathtt{presuppose!}\,(\lambda z.\, \mathbf{car}\, z \wedge \mathbf{own}\,(\mathsf{sel}\, e)\, z)\}) \gg\!= (\lambda z. \\
&\quad \mathtt{assert}\,(\neg(\mathbf{cheap}\, z))\ \lambda\_. \\
&\quad \eta\, \star))
\end{aligned}
$$

We now move up to the antecedent S_1: *John owns a car*.

$$
\begin{aligned}
[\![S_1]\!] = [\![\textsc{trans owns}\,(\textsc{a}\,(\textsc{common car}))\,(\textsc{proper John})]\!] \\
&\twoheadrightarrow \mathtt{presuppose}\ \mathbf{John}\ (\lambda x. \\
&\quad \mathtt{introduce}\ \star\ (\lambda y. \\
&\quad \mathtt{assert}\,(\mathbf{car}\, y)\,(\lambda\_. \\
&\quad \mathtt{assert}\,(\mathbf{own}\, x\, y)\,(\lambda\_. \\
&\quad \eta\, \star))))
\end{aligned}
$$

We can now compute the conjunction of $[\![S_1]\!]$ and $\neg[\![S_2]\!]$ and observe how the context of $[\![S_1]\!]$ cancels the presupposition in $\neg[\![S_2]\!]$.

$$
\begin{aligned}
[\![S_1]\!] \barwedge \neg[\![S_2]\!] = [\![S_1]\!] &\gg\!= (\lambda\_.\ \neg[\![S_2]\!]) \\
&\twoheadrightarrow \mathtt{presuppose}\ \mathbf{John}\ (\lambda x. \\
&\quad \mathtt{introduce}\ \star\ (\lambda y. \\
&\quad \mathtt{assert}\,(\mathbf{car}\, y)\,(\lambda\_. \\
&\quad \mathtt{assert}\,(\mathbf{own}\, x\, y)\,(\lambda\_. \\
&\quad \mathtt{get}\ \star\ (\lambda e. \\
&\quad (\mathbf{case}\ \mathsf{sel_P}\ (\lambda z.\, \mathbf{car}\, z \wedge \mathbf{own}\,(\mathsf{sel}\, e)\, z)\, e\ \mathbf{of} \\
&\qquad \{\mathrm{inl}\ z \to \eta\, z; \\
&\qquad\ \ \mathrm{inr}\ \_ \to \mathtt{presuppose!}\,(\lambda z.\, \mathbf{car}\, z \wedge \mathbf{own}\,(\mathsf{sel}\, e)\, z)\}) \gg\!= (\lambda z. \\
&\quad \mathtt{assert}\,(\neg(\mathbf{cheap}\, z))\ \lambda\_. \\
&\quad \eta\, \star)))))) \\
&\approx \mathtt{presuppose}\ \mathbf{John}\ (\lambda x. \\
&\quad \mathtt{get}\ \star\ (\lambda e. \\
&\quad \mathtt{introduce}\ \star\ (\lambda y. \\
&\quad \mathtt{assert}\,(\mathbf{car}\, y)\,(\lambda\_. \\
&\quad \mathtt{assert}\,(\mathbf{own}\, x\, y)\,(\lambda\_. \\
&\quad (\mathbf{case}\ \mathsf{sel_P}\ (\lambda z.\, \mathbf{car}\, z \wedge \mathbf{own}\,(\mathsf{sel}\, e')\, z)\, e'\ \mathbf{of} \\
&\qquad \{\mathrm{inl}\ z \to \eta\, z; \\
&\qquad\ \ \mathrm{inr}\ \_ \to \mathtt{presuppose!}\,(\lambda z.\, \mathbf{car}\, z \wedge \mathbf{own}(\mathsf{sel}(e'), z))\}) \gg\!= (\lambda z. \\
&\quad \mathtt{assert}\,(\neg(\mathbf{cheap}\, z))\ \lambda\_. \\
&\quad \eta\, \star)))))
\end{aligned}
$$

where

$$e' = \mathbf{own}\, x\, y :: \mathbf{car}\, y :: y :: e$$

We compose the two dynamic propositions by concatenating their effects. We then make use of our equations from 7.2.5 to move `get` above the `introduce` and `assert` to make it clear that the context e' that is given as argument to sel and sel_P contains the necessary material. We do not move `get` above `presuppose` since we cannot predict how `presuppose` will modify the context: if the presupposition gets cancelled and bound by an existing discourse referent, it does not modify the context, but if it is novel and gets accommodated, then it introduces a new discourse referent into the context. However, in all cases we can assume that after performing `presuppose` P, the context for the subsequent dynamic operations will contain some referent x satisfying the predicate P. Assuming that the context e, and therefore by extension e', contains the discourse referent x and the condition $\mathbf{John}\, x$, $\mathsf{sel}\, e'$ can choose the x as the antecedent for *his*. We can now compute the denotation of the noun phrase *his car* in this context and see that the presupposition is cancelled.

$$\begin{aligned}
&\mathbf{case}\ \mathsf{sel}_\mathsf{P}\ (\lambda z.\, \mathbf{car}\, z \wedge \mathbf{own}\, (\mathsf{sel}\, e')\, z)\, e'\ \mathbf{of} \\
&\qquad \{\mathrm{inl}\ z \to \eta\, z; \\
&\qquad\ \ \mathrm{inr}\ _ \to \mathtt{presuppose!}\, (\lambda z.\, \mathbf{car}\, z \wedge \mathbf{own}\, (\mathsf{sel}\, e')\, z)\} \\
&= \mathbf{case}\ \mathsf{sel}_\mathsf{P}\ (\lambda z.\, \mathbf{car}\, z \wedge \mathbf{own}\, x\, z)\, e'\ \mathbf{of} \\
&\qquad \{\mathrm{inl}\ z \to \eta\, z; \\
&\qquad\ \ \mathrm{inr}\ _ \to \mathtt{presuppose!}\, (\lambda z.\, \mathbf{car}\, z \wedge \mathbf{own}\, x\, z)\} \\
&= \mathbf{case}\ \mathrm{inl}\ y\ \mathbf{of} \\
&\qquad \{\mathrm{inl}\ z \to \eta\, z; \\
&\qquad\ \ \mathrm{inr}\ _ \to \mathtt{presuppose!}\, (\lambda z.\, \mathbf{car}\, z \wedge \mathbf{own}\, x\, z)\} \\
&\to_{\beta.+1} \eta\, y
\end{aligned}$$

We can plug this result back into the computation for $[\![S_1]\!] \barwedge \bar{\neg} [\![S_2]\!]$:

$$\begin{aligned}
&[\![S_1]\!] \barwedge \bar{\neg} [\![S_2]\!] \\
&\twoheadrightarrow \mathtt{presuppose}\, \mathbf{John}\, (\lambda x. \\
&\quad \mathtt{get} \star (\lambda e. \\
&\quad \mathtt{introduce} \star (\lambda y. \\
&\quad \mathtt{assert}\, (\mathbf{car}\, y)\, (\lambda_. \\
&\quad \mathtt{assert}\, (\mathbf{own}\, x\, y)\, (\lambda_. \\
&\quad \eta\, y \gg\!= (\lambda z. \\
&\quad \mathtt{assert}\, (\neg(\mathbf{cheap}\, z))\, \lambda_. \\
&\quad \eta \star)))))) \\
&\to_{\eta.\gg=} \mathtt{presuppose}\, \mathbf{John}\, (\lambda x. \\
&\qquad \mathtt{introduce} \star (\lambda y. \\
&\qquad \mathtt{assert}\, (\mathbf{car}\, y)\, (\lambda_. \\
&\qquad \mathtt{assert}\, (\mathbf{own}\, x\, y)\, (\lambda_. \\
&\qquad \mathtt{assert}\, (\neg(\mathbf{cheap}(y)))\, \lambda_. \\
&\qquad \eta \star))))
\end{aligned}$$

We use the $\to_{\eta.\gg=}$ that we derived in Proposition 3.1.4 and we also drop the `get` since we no longer use the context e (an equation from 7.2.5 admissible under our handlers). The denotation of Example 22 is then the dynamic negation of this computation.

$$
\begin{aligned}
[\![\textsc{if-then}\ S_1\ S_2]\!] &= [\![S_1]\!] \mathrel{\overline{\Rightarrow}} [\![S_2]\!] \\
&= \overline{\neg}([\![S_1]\!] \mathrel{\overline{\wedge}} \overline{\neg}[\![S_2]\!]) \\
&\twoheadrightarrow \texttt{get} \star\ (\lambda e. \\
&\quad (\textbf{case}\ \mathsf{sel_P}\ \textbf{John}\ e\ \textbf{of} \\
&\qquad \{\mathrm{inl}\ x \to \eta\, x; \\
&\qquad\ \ \mathrm{inr}\ \_ \to \texttt{presuppose!}\,\textbf{John}\}) \gg\!= (\lambda x. \\
&\quad \texttt{assert}\ (\neg(\exists y.\ \textbf{car}\, y \wedge \textbf{own}\, x\, y \wedge \neg(\textbf{cheap}(y))))\ (\lambda\_. \\
&\quad \eta \star)))
\end{aligned}
$$

Note that the resulting denotation is still accessing the context to determine whether it will presuppose the existence of John or whether it will retrieve John from some (potentially hypothetical) context. This means our solution is compositional and if we were to place this denotation inside the following context, the presupposition of there being some salient John would be cancelled:

(23) If there is a poor man called John, then if John owns a car, his car is cheap.

If we consider the sentence of Example 22 in an empty, "out of the blue" context, into which John will need to be accommodated, we get the intended reading.

$$
\begin{aligned}
&\mathsf{empty}\ (\mathsf{top}\ [\![\textsc{if-then}\ S_1\ S_2]\!]) \\
&\twoheadrightarrow \eta\,(\exists x.\ \textbf{John}\, x \wedge \neg(\exists y.\ \textbf{car}\, y \wedge \textbf{own}\, x\, y \wedge \neg(\textbf{cheap}(y)))) \\
&= \eta\,(\exists x.\ \textbf{John}\, x \wedge (\forall y.\ (\textbf{car}\, y \wedge \textbf{own}\, x\, y) \to \textbf{cheap}(y)))
\end{aligned}
$$

7.3.4 Ambiguous Accommodation

Presuppositions do not always have to accommodate at the top level. Consider the following example from [17].

(24) (c_0) Maybe (c_1) Wilma thinks that (c_2) her husband is having an affair.

The NP *her husband* presupposes that Wilma is married. If the sentence is uttered in a context in which it was not yet established that Wilma is married, the fact will need to be accomodated somewhere. We can do so in the global context c_0, which would mean interpreting the sentence as "*Wilma is married and maybe she thinks that her husband is having an affair*". However, we can also accommodate the presupposition in the intermediate context c_1 to get the interpretation "*Maybe Wilma is married and she thinks that her husband is having an affair*". Finally, we can even accommodate the presupposition in the local context c_2 to get "*Maybe Wilma thinks that she is married and her husband is having an affair*".

If we look at the different possible accommodation sites in Example 24, we remark that they are all lexically generated. There is the possible context/accommodation site c_1 in the argument of the modal operator *maybe* and there is c_2 in the argument of the attitude verb *think*. At these points, we would like to be able to interrupt presuppositions and accommodate them. In $(\!|\lambda|\!)$, this would mean applying a `presuppose` handler to these arguments. However, at the same time, we want to also have the reading in which the presupposition projects out.

In 6.4.1, we dealt with ambiguity by changing the syntax, moving quantifiers into their scope using QR. This would not be suitable for presuppositions since they can be accommodated in positions which do not correspond to sentences (e.g. the restrictor of a quantified noun phrase). This time, instead of having a different abstract term for every possible reading, we will make it so that a single object term (i.e. denotation) will represent multiple readings, as when using underspecified representations. Our denotations are computations and we want the denotations to evaluate to multiple different values. This is a feature of nondeterministic computations and we can implement nondeterminism using effects.

$$\texttt{amb} : 1 \rightarrowtail 2$$

The amb operation allows us to branch into two different computations. We can imagine this as asking an oracle or flipping a coin. We give no input and we receive a Boolean: either **T** (inl $\star$) or **F** (inr $\star$). A handler for amb might then consult some oracle, flip (pseudo-)random coins, collect all the possibilities in a list or a set or just return one of the results by always answering **T**. If we look back to the algebraic formulation of ⦇ λ ⦈ computations given in Chapter 1, we find that amb corresponds to a single binary operation on computations. This algebraic notation will actually be more useful then the computational one we usually use and so we introduce the following:

$$\begin{aligned} \_ + \_ &: \mathcal{F}_{E \uplus \{\texttt{amb}\}}(\alpha) \to \mathcal{F}_{E \uplus \{\texttt{amb}\}}(\alpha) \to \mathcal{F}_{E \uplus \{\texttt{amb}\}}(\alpha) \\ M + N &= \texttt{amb} \star (\lambda b.\, \textbf{if}\ b\ \textbf{then}\ M\ \textbf{else}\ N) \end{aligned}$$

We can now write a handler that tries both projecting a presupposition and accommodating it.

$$\begin{aligned} &\mathsf{maybeAccommodate} : \mathcal{F}_{E \uplus \{\texttt{presuppose}\}}(\alpha) \to \mathcal{F}_{E \uplus \{\texttt{presuppose,amb}\}}(\alpha) \\ &\mathsf{maybeAccommodate} = ⦇ \texttt{presuppose}\colon (\lambda P k.\, \texttt{presuppose}\, P\, k + \texttt{introduce} \star (\lambda x.\, \texttt{assert}\, (P\, x)\, (\lambda\_.\, k\, x))) ⦈ \end{aligned}$$

Now we need to include this handler into the lexical entries of our grammar, wherever presuppositions can be accommodated. Here, we can follow existing analyses. Projective DRT [131] lets presupposed content accommodate within DRSs on the projection line. Lebedeva's use of exceptions for presuppositions in TTDL [80] employs an exception handler (iacc) that allows for accommodation at every point where a discourse referent is being bound (i.e. at every dynamic existential quantification). We can achieve the same result in our ⦇ λ ⦈ analysis by including the maybeAccommodate in the $\overline{\mathsf{box}}$ handler:

$$\begin{aligned} &\overline{\mathsf{box}} : \mathcal{F}_{E \uplus E_{\mathrm{DRT}} \uplus \{\texttt{presuppose}\}}(1) \to \mathcal{F}_{E \uplus \{\texttt{get,presuppose,amb}\}}(o) \\ &\overline{\mathsf{box}} = \mathsf{box} \circ \mathsf{maybeAccommodate} \circ \mathsf{useFind} \end{aligned}$$

Now, if our semantics for the modal operator *maybe* and the attitude verb *think* use boxes, then we get as denotation of Example 24 a nondeterministic computation that yields the three readings mentioned above. Looking back at those three readings, [17] states that while they are all possible, the first reading (global accommodation at c_0) is strongly preferred to the other readings. This is generalized to the following empirically motivated principle:

> PGA (Preference for Global Accommodation): Global accommodation is preferred to non-global accommodation.
>
> [17], 5.1

If it were only for this principle, we might wonder why we bothered with allowing accommodation in any DRS on the projection line in the first place if we are going to always prefer accommodating globally. However, there are other forces which push in the opposite direction and impose constraints on presupposition projection. The one that we will tend to in this section is the effect of variable binding. A proposition being presupposed cannot escape the scope of a binder that binds one of its variables. In [17], this is illustrated on the following example.

(25) Most Germans wash their car on Saturday.

The expression *their car* triggers the presupposition of the existence of a car belonging to the referent of *their*. We could accommodate this presupposition locally, in the nucleus of the quantifier *most Germans*, giving the reading "*Most Germans have a car and they wash it on Saturday.*". We could also accommodate

it in the restrictor of that quantifier (which is accessible, i.e. on the projection line, from the nucleus), giving us "*Most Germans who have a car wash it on Saturday*".

However, we cannot get the global accommodation reading. The problem is that the referent of *their* is a variable being quantified over by the quantifier *most Germans*. If this variable is x, then the presupposition being triggered is the existence of an individual satisfying $\lambda y.\, \mathbf{car}\, y \wedge \mathbf{own}\, x\, y$. This predicate contains a free variable x, which means that for every different value x, we actually have a different predicate that identifies a different car. The solution to this *binding problem* in theories that implement accommodation, such as van der Sandt's DRT approach [130] or Lebedeva's extension of TTDL [80], is to reflect this logical impossibility into a linguistic constraint. Presupposed material cannot project out of a binder that binds some part of that material.

We will look at how a similar example turns out in our system.

(26) (c_0) If (c_1) a man gets angry, (c_2) his children get frightened.

Example 26, taken from [130], is similar in nature to Example 25. We have a presupposition trigger which can be accommodated in a local context (c_2) and in an intermediate context (c_1), but not in the global context (c_0), because of a bound variable. Furthermore, Example 26 is built out of constructions we have already covered and so we are ready to compute the denotation. We do not analyze the construction *X's children* as a possessive (POSS X CHILDREN), which would refer to some children owned by X. Since *children* is a relational noun, we assume a binary relation **children** $y\, x$ expressing that y are children of x. We then analyze *X's children* as CHILDREN X where the semantics of CHILDREN makes use of the relation **children**. For the construction X *gets* A, where A is an adjective, we introduce the abstract constant GET.

$$
\begin{aligned}
\textsc{children} &: NP \multimap NP \\
[\![\textsc{children}]\!] &= \lambda X.\, X \gg\!= (\lambda x.\, \mathtt{presuppose!}\, (\lambda y.\, \mathbf{children}\, y\, x)) \\
\textsc{get} &: ADJ \multimap NP \multimap S \\
[\![ADJ]\!] &= \iota \to o \\
[\![\textsc{get}]\!] &= \lambda a X.\, X \gg\!= (\lambda x.\, \mathtt{assert!}\, (a\, x))
\end{aligned}
$$

We can now compute the denotation of Example 26. The structure of Example 26 is IF-THEN $S_1\, S_2$ where S_1 and S_2 are the antecedent and consequent, respectively. Its semantics, $[\![\textsc{if-then}\, S_1\, S_2]\!]$, is $S_1 \Rightarrow S_2 = \neg([\![S_1]\!] \barwedge \neg[\![S_2]\!])$. We start with the denotation of the consequent S_2: *his children get frightened*.

$$
\begin{aligned}
[\![S_2]\!] &= [\![\textsc{get frightened}\, (\textsc{children he})]\!] \\
&\twoheadrightarrow (\mathtt{get} \star (\lambda e. \\
&\quad \mathtt{presuppose}\, (\lambda y.\, \mathbf{children}\, y\, (\mathsf{sel}\, e))\, (\lambda y. \\
&\quad \mathtt{assert}\, (\mathbf{frightened}\, y)\, (\lambda\_. \\
&\quad \eta \star))))
\end{aligned}
$$

Our next step will be computing $\neg[\![S_2]\!]$. Dynamic negation is defined by boxing and then asserting the negation of the boxed proposition. We start by boxing $[\![S_2]\!]$, i.e. evaluating $\overline{\mathsf{box}}\, [\![S_2]\!]$.

$$
\begin{aligned}
\overline{\mathsf{box}}\,[\![S_2]\!] &\twoheadrightarrow \overline{\mathsf{box}}\,(\mathtt{get}\ \star\ (\lambda e.\\
&\qquad \mathtt{presuppose}\,(\lambda y.\,\mathbf{children}\,y\,(\mathsf{sel}\,e))\,(\lambda y.\\
&\qquad \mathtt{assert}\,(\mathbf{frightened}\,y)\,(\lambda\_.\\
&\qquad \eta\,\star))))\\
&= \mathsf{box}\,(\mathsf{maybeAccommodate}\,(\mathsf{useFind}\ (\mathtt{get}\ \star\ (\lambda e.\\
&\qquad\qquad\qquad\qquad \mathtt{presuppose}\,(\lambda y.\,\mathbf{children}\,y\,(\mathsf{sel}\,e))\,(\lambda y.\\
&\qquad\qquad\qquad\qquad \mathtt{assert}\,(\mathbf{frightened}\,y)\,(\lambda\_.\\
&\qquad\qquad\qquad\qquad \eta\,\star))))))\\
&\twoheadrightarrow \mathsf{box}\,(\mathsf{maybeAccommodate}\ (\mathtt{get}\ \star\ (\lambda e.\\
&\qquad\qquad\qquad\qquad \mathtt{get}\ \star\ (\lambda e.\\
&\qquad\qquad\qquad\qquad \mathbf{case}\ \mathsf{sel_P}\,(\lambda y.\,\mathbf{children}\,y\,(\mathsf{sel}\,e))\,e\ \mathbf{of}\\
&\qquad\qquad\qquad\qquad\quad \{\mathrm{inl}\ y \to \eta\,y;\\
&\qquad\qquad\qquad\qquad\quad\ \mathrm{inr}\ \_ \to \mathtt{presuppose!}\,(\lambda y.\,\mathbf{children}\,y\,(\mathsf{sel}\,e))\} \gg\!= (\lambda y.\\
&\qquad\qquad\qquad\qquad \mathtt{assert}\,(\mathbf{frightened}\,y)\,(\lambda\_.\\
&\qquad\qquad\qquad\qquad \eta\,\star))))))\\
&\approx \mathsf{box}\,(\mathsf{maybeAccommodate}\ (\mathtt{get}\ \star\ (\lambda e.\\
&\qquad\qquad\qquad\qquad \mathtt{presuppose}\,(\lambda y.\,\mathbf{children}\,y\,(\mathsf{sel}\,e))\,(\lambda y.\\
&\qquad\qquad\qquad\qquad \mathtt{assert}\,(\mathbf{frightened}\,y)\,(\lambda\_.\\
&\qquad\qquad\qquad\qquad \eta\,\star)))))\\
&\twoheadrightarrow \mathsf{box}\,(\mathtt{get}\ \star\ (\lambda e.\\
&\qquad\qquad (\mathtt{presuppose}\,(\lambda y.\,\mathbf{children}\,y\,(\mathsf{sel}\,e))\,(\lambda y.\\
&\qquad\qquad\ \mathtt{assert}\,(\mathbf{frightened}\,y)\,(\lambda\_.\\
&\qquad\qquad\ \eta\,\star)))\\
&\qquad\qquad + (\mathtt{introduce}\ \star\ (\lambda y.\\
&\qquad\qquad\ \mathtt{assert}\,(\mathbf{children}\,y\,(\mathsf{sel}\,e))\,(\lambda\_.\\
&\qquad\qquad\ \mathtt{assert}\,(\mathbf{frightened}\,y)\,(\lambda\_.\\
&\qquad\qquad\ \eta\,\star))))))\\
&\twoheadrightarrow \mathtt{get}\ \star\ (\lambda e.\\
&\qquad (\mathtt{presuppose}\,(\lambda y.\,\mathbf{children}\,y\,(\mathsf{sel}\,e))\,(\lambda y.\\
&\qquad\ \eta\,(\mathbf{frightened}\,y)))\\
&\qquad + (\eta\,(\exists y.\,\mathbf{children}\,y\,(\mathsf{sel}\,e) \wedge \mathbf{frightened}\,y)))
\end{aligned}
$$

$\overline{\mathsf{box}}$ is composed of box, maybeAccommodate and useFind. We apply all of these handlers in turn. We start with useFind, where we assume that the sentence is uttered in a context in which the existence of *his children* was not established. This means that $\mathsf{sel_P}$ will not find a referent within e and the presupposition will not be cancelled. We also rely on the idempotence of `get` to collapse the two `gets` into one and so useFind ends up having no effect. Next up is maybeAccommodate that tries both projecting the presupposition and accommodating it using `introduce` and `assert`. Finally, box translates the dynamic propositions that use `introduce` and `assert` into simple propositions. We have one proposition with a presupposition projecting out of it and another one in which the presupposition was accommodated using an existential quantifier. The dynamic negation will negate these propositions and wrap them in `assert`:

$$
\begin{aligned}
\overline{\neg}[\![S_2]\!] = \overline{\mathsf{box}}\,[\![S_2]\!] &\gg\!= (\lambda a.\, \mathtt{assert!}\,(\neg a)) \\
&\twoheadrightarrow \mathtt{get} \star (\lambda e. \\
&\qquad (\mathtt{presuppose}\,(\lambda y.\, \textbf{children}\, y\, (\mathsf{sel}\, e))\,(\lambda y. \\
&\qquad\ \ \eta\,(\textbf{frightened}\, y))) \\
&\quad + (\eta\,(\exists y.\, \textbf{children}\, y\, (\mathsf{sel}\, e) \wedge \textbf{frightened}\, y)) \\
&\quad \gg\!= (\lambda a.\, \mathtt{assert!}\,(\neg a))) \\
&\twoheadrightarrow \mathtt{get} \star (\lambda e. \\
&\qquad (\mathtt{presuppose}\,(\lambda y.\, \textbf{children}\, y\, (\mathsf{sel}\, e))\,(\lambda y. \\
&\qquad\ \ \mathtt{assert}\,(\neg(\textbf{frightened}\, y))\,(\lambda\_. \\
&\qquad\ \ \eta \star))) \\
&\quad + (\mathtt{assert}\,(\neg(\exists y.\, \textbf{children}\, y\, (\mathsf{sel}\, e) \wedge \textbf{frightened}\, y))\,(\lambda\_. \\
&\qquad\ \ \eta \star)))
\end{aligned}
$$

We now turn to the antecedent S_1: *a man gets angry*.

$$
\begin{aligned}
[\![S_1]\!] = [\![\textsc{get angry}\,(\textsc{a}\,(\textsc{common man}))]\!] \\
\twoheadrightarrow \mathtt{introduce} \star (\lambda x. \\
\mathtt{assert}\,(\textbf{man}\, x)\,(\lambda\_. \\
\mathtt{assert}\,(\textbf{angry}\, x)\,(\lambda\_. \\
\eta \star)))
\end{aligned}
$$

The denotation of Example 26 is $\overline{\neg}([\![S_1]\!] \mathbin{\overline{\wedge}} \overline{\neg}[\![S_2]\!])$ and so we have to compute $[\![S_1]\!] \mathbin{\overline{\wedge}} \overline{\neg}[\![S_2]\!]$ and then the dynamic negation of the result.

$$
\begin{aligned}
[\![S_1]\!] \mathbin{\overline{\wedge}} \overline{\neg}[\![S_2]\!] = [\![S_1]\!] &\gg\!= (\lambda\_.\, \overline{\neg}[\![S_2]\!]) \\
&\twoheadrightarrow \mathtt{introduce} \star (\lambda x. \\
&\quad \mathtt{assert}\,(\textbf{man}\, x)\,(\lambda\_. \\
&\quad \mathtt{assert}\,(\textbf{angry}\, x)\,(\lambda\_. \\
&\quad \mathtt{get} \star (\lambda e. \\
&\qquad (\mathtt{presuppose}\,(\lambda y.\, \textbf{children}\, y\, (\mathsf{sel}\, e))\,(\lambda y. \\
&\qquad\ \ \mathtt{assert}\,(\neg(\textbf{frightened}\, y))\,(\lambda\_. \\
&\qquad\ \ \eta \star))) \\
&\quad + (\mathtt{assert}\,(\neg(\exists y.\, \textbf{children}\, y\, (\mathsf{sel}\, e) \wedge \textbf{frightened}\, y))\,(\lambda\_. \\
&\qquad\ \ \eta \star))))))
\end{aligned}
$$

We start computing $\overline{\neg}([\![S_1]\!] \mathbin{\overline{\wedge}} \overline{\neg}[\![S_2]\!])$ by boxing $[\![S_1]\!] \mathbin{\overline{\wedge}} \overline{\neg}[\![S_2]\!]$:

$$
\begin{array}{l}
\overline{\mathsf{box}}\,(\llbracket S_1 \rrbracket \,\overline{\wedge}\, \overline{\neg} \llbracket S_2 \rrbracket) \\
\twoheadrightarrow \overline{\mathsf{box}}\,(\,\mathtt{introduce} \star (\lambda x. \\
\qquad \mathtt{assert}\,(\mathbf{man}\,x)\,(\lambda\_. \\
\qquad \mathtt{assert}\,(\mathbf{angry}\,x)\,(\lambda\_. \\
\qquad \mathtt{get} \star (\lambda e. \\
\qquad\quad (\mathtt{presuppose}\,(\lambda y.\,\mathbf{children}\,y\,(\mathsf{sel}\,e))\,(\lambda y. \\
\qquad\quad\ \mathtt{assert}\,(\neg(\mathbf{frightened}\,y))\,(\lambda\_. \\
\qquad\quad\ \eta\star))) \\
\qquad + (\mathtt{assert}\,(\neg(\exists y.\,\mathbf{children}\,y\,(\mathsf{sel}\,e) \wedge \mathbf{frightened}\,y))\,(\lambda\_. \\
\qquad\quad\ \eta\star))))))) \\
\twoheadrightarrow \mathsf{box}\,(\mathsf{maybeAccommodate}\,(\,\mathtt{introduce} \star (\lambda x. \\
\qquad\qquad\qquad\qquad \mathtt{assert}\,(\mathbf{man}\,x)\,(\lambda\_. \\
\qquad\qquad\qquad\qquad \mathtt{assert}\,(\mathbf{angry}\,x)\,(\lambda\_. \\
\qquad\qquad\qquad\qquad \mathtt{get} \star (\lambda e. \\
\qquad\qquad\qquad\qquad\quad (\mathtt{presuppose}\,(\lambda y.\,\mathbf{children}\,y\,(\mathsf{sel}\,e))\,(\lambda y. \\
\qquad\qquad\qquad\qquad\quad\ \mathtt{assert}\,(\neg(\mathbf{frightened}\,y))\,(\lambda\_. \\
\qquad\qquad\qquad\qquad\quad\ \eta\star))) \\
\qquad\qquad\qquad\qquad + (\mathtt{assert}\,(\neg(\exists y.\,\mathbf{children}\,y\,(\mathsf{sel}\,e) \wedge \mathbf{frightened}\,y))\,(\lambda\_. \\
\qquad\qquad\qquad\qquad\quad\ \eta\star)))))))) \\
\twoheadrightarrow \mathsf{box}\,(\,\mathtt{introduce} \star (\lambda x. \\
\qquad \mathtt{assert}\,(\mathbf{man}\,x)\,(\lambda\_. \\
\qquad \mathtt{assert}\,(\mathbf{angry}\,x)\,(\lambda\_. \\
\qquad \mathtt{get} \star (\lambda e. \\
\qquad\quad (\mathtt{presuppose}\,(\lambda y.\,\mathbf{children}\,y\,(\mathsf{sel}\,e))\,(\lambda y. \\
\qquad\quad\ \mathtt{assert}\,(\neg(\mathbf{frightened}\,y))\,(\lambda\_. \\
\qquad\quad\ \eta\star))) \\
\qquad + (\mathtt{introduce} \star (\lambda y. \\
\qquad\quad\ \mathtt{assert}\,(\mathbf{children}\,y\,(\mathsf{sel}\,e))\,(\lambda\_. \\
\qquad\quad\ \mathtt{assert}\,(\neg(\mathbf{frightened}\,y))\,(\lambda\_. \\
\qquad\quad\ \eta\star)))) \\
\qquad + (\mathtt{assert}\,(\neg(\exists y.\,\mathbf{children}\,y\,(\mathsf{sel}\,e) \wedge \mathbf{frightened}\,y))\,(\lambda\_. \\
\qquad\quad\ \eta\star))))))) \\
\approx \mathsf{box}\,(\,\mathtt{get} \star (\lambda e. \\
\qquad \mathtt{introduce} \star (\lambda x. \\
\qquad \mathtt{assert}\,(\mathbf{man}\,x)\,(\lambda\_. \\
\qquad \mathtt{assert}\,(\mathbf{angry}\,x)\,(\lambda\_. \\
\qquad\quad (\mathtt{presuppose}\,(\lambda y.\,\mathbf{children}(y, \mathsf{sel}\,(\mathbf{angry}\,x :: \mathbf{man}\,x :: x :: e)))\,(\lambda y. \\
\qquad\quad\ \mathtt{assert}\,(\neg(\mathbf{frightened}\,y))\,(\lambda\_. \\
\qquad\quad\ \eta\star))) \\
\qquad + (\mathtt{introduce} \star (\lambda y. \\
\qquad\quad\ \mathtt{assert}\,(\mathbf{children}(y, \mathsf{sel}\,(\mathbf{angry}\,x :: \mathbf{man}\,x :: x :: e)))\,(\lambda\_. \\
\qquad\quad\ \mathtt{assert}\,(\neg(\mathbf{frightened}\,y))\,(\lambda\_. \\
\qquad\quad\ \eta\star)))) \\
\qquad + (\mathtt{assert}\,(\neg(\exists y.\,\mathbf{children}(y, \mathsf{sel}\,(\mathbf{angry}\,x :: \mathbf{man}\,x :: x :: e)) \wedge \mathbf{frightened}\,y))\,(\lambda\_. \\
\qquad\quad\ \eta\star)))))))
\end{array}
$$

We skip useFind, since, as before, it does not cancel any presupposition. Then maybeAccommodate will take the presupposition and consider the two alternatives: projecting this presupposition further down the projection line or accommodating it here, in the antecedent. In the last step, we commute `get` with `introduce` and `assert` using the equations from 7.2.5. This makes it more obvious that x is one of the available discourse referents for the pronoun *his* in the consequent. Assuming that this pronoun will actually resolve to x and not to some other referent (i.e. sel $(\mathbf{angry}\, x :: \mathbf{man}\, x :: x :: e) = x$), then the argument to `presuppose` will be $\lambda y.\, \mathbf{children}\, y\, x$, in which x occurs free.

The computation is nondeterministic and uses `amb` $(+)$ to split into three possible interpretations:

- projecting the presupposition out of the conditional (the way of global accommodation)
- accommodating the presupposition at the level of the antecedent of the conditional (intermediate accommodation)
- accommodating the presupposition already at the level of the consequent of the conditional (local accommodation)

We continue by applying the box handler.

$$
\begin{aligned}
&\mathsf{box}\,(\,\mathtt{get} \star (\lambda e.\\
&\quad \mathtt{introduce} \star (\lambda x.\\
&\quad \mathtt{assert}\,(\mathbf{man}\, x)\,(\lambda\_.\\
&\quad \mathtt{assert}\,(\mathbf{angry}\, x)\,(\lambda\_.\\
&\qquad (\mathtt{presuppose}\,(\lambda y.\, \mathbf{children}\, y\, x)\,(\lambda y.\\
&\qquad\ \mathtt{assert}\,(\neg(\mathbf{frightened}\, y))\,(\lambda\_.\\
&\qquad\ \eta \star)))\\
&\quad + (\mathtt{introduce} \star (\lambda y.\\
&\qquad\ \mathtt{assert}\,(\mathbf{children}\, y\, x)\,(\lambda\_.\\
&\qquad\ \mathtt{assert}\,(\neg(\mathbf{frightened}\, y))\,(\lambda\_.\\
&\qquad\ \eta \star))))\\
&\quad + (\mathtt{assert}\,(\neg(\exists y.\, \mathbf{children}\, y\, x \wedge \mathbf{frightened}\, y))\,(\lambda\_.\\
&\qquad\ \eta \star)))))))\\
\leftrightarrow\ &\mathsf{box}\,(\mathtt{introduce} \star (\lambda x.\\
&\qquad\ \mathtt{assert}\,(\mathbf{man}\, x)\,(\lambda\_.\\
&\qquad\ \mathtt{assert}\,(\mathbf{angry}\, x)\,(\lambda\_.\\
&\qquad\ \mathtt{presuppose}\,(\lambda y.\, \mathbf{children}\, y\, x)\,(\lambda y.\\
&\qquad\ \mathtt{assert}\,(\neg(\mathbf{frightened}\, y))\,(\lambda\_.\\
&\qquad\ \eta \star))))))\\
&\quad + \eta\,(\exists x.\, \mathbf{man}\, x \wedge \mathbf{angry}\, x \wedge \exists y.\, \mathbf{children}\, y\, x \wedge \neg(\mathbf{frightened}\, y))\\
&\quad + \eta\,(\exists x.\, \mathbf{man}\, x \wedge \mathbf{angry}\, x \wedge \neg(\exists y.\, \mathbf{children}\, y\, x \wedge \mathbf{frightened}\, y))\\
\rightarrow\ &\exists_{\gg}\, x.\, \mathtt{presuppose}\,(\lambda y.\, \mathbf{children}\, y\, x)\,(\lambda y.\\
&\qquad\ \eta\,(\mathbf{man}\, x \wedge \mathbf{angry}\, x \wedge \neg(\mathbf{frightened}\, y)))\\
&\quad + \eta\,(\exists x.\, \mathbf{man}\, x \wedge \mathbf{angry}\, x \wedge \exists y.\, \mathbf{children}\, y\, x \wedge \neg(\mathbf{frightened}\, y))\\
&\quad + \eta\,(\exists x.\, \mathbf{man}\, x \wedge \mathbf{angry}\, x \wedge \neg(\exists y.\, \mathbf{children}\, y\, x \wedge \mathbf{frightened}\, y))
\end{aligned}
$$

As a result, we get two propositions and one stuck computation. The two propositions will (after negation) gives us the intermediate accommodation reading and the local accommodation reading, whereas the stuck computation represents the impossibility of the global accommodation reading.

The term $\exists_{\gg}\, x.\, \texttt{presuppose}\,(\lambda y.\, \textbf{children}\, y\, x)\, M_{\mathrm{c}}$ is stuck, because when we expand $\exists_{\gg}$, it becomes $\exists \cdot \gg (\mathcal{C}\,(\lambda x.\, \texttt{presuppose}\,(\lambda y.\, \textbf{children}\, y\, x)\, M_{\mathrm{c}}))$. There is no rule in ⦇ λ ⦈ to reduce $\mathcal{C}\,(\lambda x.\, \texttt{presuppose}\, M_{\mathrm{p}}\, M_{\mathrm{c}})$ when x occurs free in M_{p} and the ⦇ λ ⦈ denotational semantics would assign $\bot$ to this expression.

We finish computing the denotation of Example 26 by wrapping up the dynamic negation: negating and asserting the propositions.

$$
\begin{aligned}
[\![\textsc{if-then}\, S_1\, S_2]\!] &= \overline{\neg}([\![S_1]\!]\, \overline{\wedge}\, \overline{\neg}[\![S_2]\!]) \\
&= \overline{\mathsf{box}}\,([\![S_1]\!]\, \overline{\wedge}\, \overline{\neg}[\![S_2]\!]) \gg\!= (\lambda a.\, \texttt{assert!}\,(\neg a)) \\
&\twoheadrightarrow (\exists_{\gg}\, x.\, \texttt{presuppose}\,(\lambda y.\, \textbf{children}\, y\, x)\,(\lambda y. \\
&\qquad\qquad \eta\,(\textbf{man}\, x \wedge \textbf{angry}\, x \wedge \neg(\textbf{frightened}\, y))) \\
&\qquad + \eta\,(\exists x.\, \textbf{man}\, x \wedge \textbf{angry}\, x \wedge \exists y.\, \textbf{children}\, y\, x \wedge \neg(\textbf{frightened}\, y)) \\
&\qquad + \eta\,(\exists x.\, \textbf{man}\, x \wedge \textbf{angry}\, x \wedge \neg(\exists y.\, \textbf{children}\, y\, x \wedge \textbf{frightened}\, y))) \\
&\qquad \gg\!= (\lambda a.\, \texttt{assert!}\,(\neg a)) \\
&\twoheadrightarrow^{139} ((\exists_{\gg}\, x.\, \texttt{presuppose}\,(\lambda y.\, \textbf{children}\, y\, x)\,(\lambda y. \\
&\qquad\qquad\qquad \eta\,(\textbf{man}\, x \wedge \textbf{angry}\, x \wedge \neg(\textbf{frightened}\, y)))) \\
&\qquad\qquad\qquad \gg\!= (\lambda a.\, \texttt{assert!}\,(\neg a))) \\
&\qquad\quad + (\,\eta\,(\exists x.\, \textbf{man}\, x \wedge \textbf{angry}\, x \wedge \exists y.\, \textbf{children}\, y\, x \wedge \neg(\textbf{frightened}\, y)) \\
&\qquad\qquad \gg\!= (\lambda a.\, \texttt{assert!}\,(\neg a))) \\
&\qquad\quad + (\,\eta\,(\exists x.\, \textbf{man}\, x \wedge \textbf{angry}\, x \wedge \neg(\exists y.\, \textbf{children}\, y\, x \wedge \textbf{frightened}\, y)) \\
&\qquad\qquad \gg\!= (\lambda a.\, \texttt{assert!}\,(\neg a))) \\
&\twoheadrightarrow ((\exists_{\gg}\, x.\, \texttt{presuppose}\,(\lambda y.\, \textbf{children}\, y\, x)\,(\lambda y. \\
&\qquad\qquad\qquad \eta\,(\textbf{man}\, x \wedge \textbf{angry}\, x \wedge \neg(\textbf{frightened}\, y)))) \\
&\qquad\qquad\qquad \gg\!= (\lambda a.\, \texttt{assert!}\,(\neg a))) \\
&\qquad + (\texttt{assert!}\,(\neg(\exists x.\, \textbf{man}\, x \wedge \textbf{angry}\, x \wedge \exists y.\, \textbf{children}\, y\, x \wedge \neg(\textbf{frightened}\, y)))) \\
&\qquad + (\texttt{assert!}\,(\neg(\exists x.\, \textbf{man}\, x \wedge \textbf{angry}\, x \wedge \neg(\exists y.\, \textbf{children}\, y\, x \wedge \textbf{frightened}\, y)))) \\
&= ((\exists_{\gg}\, x.\, \texttt{presuppose}\,(\lambda y.\, \textbf{children}\, y\, x)\,(\lambda y. \\
&\qquad\qquad \eta\,(\textbf{man}\, x \wedge \textbf{angry}\, x \wedge \neg(\textbf{frightened}\, y)))) \\
&\qquad\qquad \gg\!= (\lambda a.\, \texttt{assert!}\,(\neg a))) \\
&\qquad + (\texttt{assert!}\,(\forall xy.\, (\textbf{man}\, x \wedge \textbf{angry}\, x \wedge \textbf{children}\, y\, x) \rightarrow \textbf{frightened}\, y)) \\
&\qquad + (\texttt{assert!}\,(\forall x.\, (\textbf{man}\, x \wedge \textbf{angry}\, x) \rightarrow (\exists y.\, \textbf{children}\, y\, x \wedge \textbf{frightened}\, y)))
\end{aligned}
$$

In the steps above, we pass the $\gg\!=$ operator through the `amb` operation symbol, which is represented by $+$. We then perform the negation and assertion on the two successful computations. We also change the resulting propositions into equivalent but more readable ones. The resulting denotation is a nondeterministic computation that can either produce the intermediate accommodation reading (*"If a man who has children gets angry, then his children get frightened"*), the local accommodation reading (*"If a man gets angry, then he also has children who get frightened"*) or get stuck. The PGA principle holds here since the intermediate accommodation reading is preferred to the local accommodation one.

We can write a handler that will recover the reading where the presupposition projects as far as possible without getting blocked by a binding. Our operation symbol for nondeterminism was called `amb` because of its similarity to McCarthy's *amb* operator for writing ambiguous (i.e. nondeterministic) functions [92, 4]. The expression $C[amb(M, N)]$ reduces to either $C[M]$ or $C[N]$, ensuring that the

[139] Here we are making use of the following rule: $(A + B) \gg\!= F \;\rightarrow\; (A \gg\!= F) + (B \gg\!= F)$. However, this rule is not derivable in ⦇ λ ⦈. The $A + B$ is de-sugared into two parts: the `amb` operation and an if expression (case analysis). The `op`.$\gg\!=$ rule lets us move the $\gg\!=$ under the operation. However, we have no rule which lets us commute a $\gg\!=$ (which is a handler) and case analysis (we will speak more about this in 9.3.1). Nevertheless, in this example, we permit ourselves to apply this rule as it does not affect the denotation of the term nor the final value and makes the notation a bit more bearable. If we were stricter, we would not be able to proceed with the $\gg\!=$ until after we have selected the preferred reading using the `search` handler that we will present next.

resulting expression successfully produces a value. This kind of choice operator is sometimes called "angelic" because it watches out for us by making choices that will lead to a successful evaluation. This is the kind of oversight we need in our choice operator as well. Consider the `presuppose` clause of the maybeAccommodate handler:

$$\mathsf{maybeAccommodate} = (\!|\, \mathtt{presuppose}\colon (\lambda P k.\, \mathtt{presuppose}\, P\, k + \mathtt{introduce} \star (\lambda x.\, \mathtt{assert}\, (P\, x)\, k))\, |\!)$$

Neither $\mathtt{presuppose}\, P\, k$ nor $\mathtt{introduce} \star (\lambda x.\, \mathtt{assert}\, (P\, x)\, k)$ are stuck computations, but if in some larger context we decide to choose, e.g., the former, we might end up being stuck. This was the case in the denotation of Example 26: we decided twice to project the presupposition and later, when we applied the box handler, we were stuck.

The amb operator can be implemented in Scheme using continuations [122] and so we should be able to do the same using effects and handlers in $(\!|\, \lambda \,|\!)$. We will give amb the following semantics: $C[amb(M, N)]$ reduces to $C[M]$ if $C[M]$ reduces to a value; otherwise, $C[amb(M, N)]$ reduces to N. Our choice operator will be left-leaning, preferring to take the first choice but accepting the second if the first one leads to a failure. This way, we can encode a notion of preference into the choices. As when implementing `shift` in Chapter 4, we will use an operation for the amb operator, the `amb` operation symbol, written using $+$, and a handler to delimit the context C whose result we do not want to get stuck.

$$\begin{aligned}
&\mathsf{search} : \mathcal{F}_{E \uplus \{\mathtt{amb}\}}(\alpha) \to \mathcal{F}_E(\alpha) \\
&\mathsf{search} = (\!|\, \mathtt{amb}\colon (\lambda_k.\, k\, \mathbf{T};\ k\, \mathbf{F})\, |\!)
\end{aligned}$$

In the above, we use a new construction $M; N$. The amb operator considers the result of a decision inside some context and then checks whether the result is a success or not in order to decide. The provisioning of the context is handled by search, whereas checking whether a computation is stuck is done by the $M; N$ construction. The idea behind $M; N$ is that $M; N$ should reduce to N if M is stuck; otherwise, it should reduce to M. We will give a formal definition shortly.

If we apply this handler to the denotation of Example 26, we will get the intended reading: the one in which the presupposition projects as widely as possible without violating binding.[140]

$$\begin{aligned}
\mathsf{search}\, [\![\textsc{if-then}\, S_1\, S_2]\!] \twoheadrightarrow\ & \mathsf{search}\, (\, ((\exists_{\gg}\, x.\ \mathtt{presuppose}\, (\lambda y.\, \mathbf{children}\, y\, x)\, (\lambda y. \\
& \qquad \eta\, (\mathbf{man}\, x \wedge \mathbf{angry}\, x \wedge \neg(\mathbf{frightened}\, y)))) \\
& \qquad \gg\!= (\lambda a.\, \mathtt{assert!}\, (\neg a))) \\
& \quad + (\mathtt{assert!}\, (\forall xy.\, (\mathbf{man}\, x \wedge \mathbf{angry}\, x \wedge \mathbf{children}\, y\, x) \to \mathbf{frightened}\, y)) \\
& \quad + (\mathtt{assert!}\, (\forall x.\, (\mathbf{man}\, x \wedge \mathbf{angry}\, x) \to (\exists y.\, \mathbf{children}\, y\, x \wedge \mathbf{frightened}\, y)))) \\
\twoheadrightarrow\ & ((\exists_{\gg}\, x.\ \mathtt{presuppose}\, (\lambda y.\, \mathbf{children}\, y\, x)\, (\lambda y. \\
& \qquad \eta\, (\mathbf{man}\, x \wedge \mathbf{angry}\, x \wedge \neg(\mathbf{frightened}\, y)))) \\
& \qquad \gg\!= (\lambda a.\, \mathtt{assert!}\, (\neg a))); \\
& \quad (\mathtt{assert!}\, (\forall xy.\, (\mathbf{man}\, x \wedge \mathbf{angry}\, x \wedge \mathbf{children}\, y\, x) \to \mathbf{frightened}\, y)); \\
& \quad (\mathtt{assert!}\, (\forall x.\, (\mathbf{man}\, x \wedge \mathbf{angry}\, x) \to (\exists y.\, \mathbf{children}\, y\, x \wedge \mathbf{frightened}\, y))) \\
\twoheadrightarrow\ & \mathtt{assert!}\, (\forall xy.\, (\mathbf{man}\, x \wedge \mathbf{angry}\, x \wedge \mathbf{children}\, y\, x) \to \mathbf{frightened}\, y)
\end{aligned}$$

Identifying Stuck Computations

We will now give a formal semantics to the $M; N$ construction. We will not specify the reduction relation as a reduction rule using pattern matching because stuck computations are not easy to distinguish syntactically. The stuck redex can be buried deep within the term M and even if M contains a stuck redex,

[140] If we wanted to retrieve all admissible readings, we could write a handler that would use the $M; N$ construction to build a list containing all the non-stuck solutions.

it is possible that this redex might disappear through some other reduction and therefore not cause the resulting computation to get stuck. Instead of identifying stuck computations syntactically, we will use our denotational semantics from 3.3.1, which already identifies stuck computations with $\bot$. However, the constructors for computation types are not strict. If their arguments are $\bot$, the results do not have to be (e.g. if $[\![M]\!](e) = \bot$, then $[\![\eta\, M]\!](e) = \eta(\bot) \neq \bot$). We will consider a computation successful if it does not contain $\bot$, a notion we define formally as being $\bot$-less. But first, we give the typing rule for $M; N$ expressions.

Definition 7.3.1. *The **types of** $M; N$ are given by the following inference rule:*

$$\frac{\Gamma \vdash M : \alpha \qquad \Gamma \vdash N : \alpha}{\Gamma \vdash M; N : \alpha}\ [;]$$

Definition 7.3.2. *For every $(\!|\lambda|\!)$ type τ, we define a predicate "**is $\bot$-less**" on the domain of τ, $[\![\tau]\!]$.*
By induction on the structure of type τ:

- *$x \in [\![\nu]\!]$ is $\bot$-less if $x \neq \bot$*
- *$f \in [\![\alpha \to \beta]\!]$ is $\bot$-less if $f \neq \bot$ and for every $\bot$-less $x \in [\![\alpha]\!]$, $f(x)$ is $\bot$-less*
- *$e \in [\![\mathcal{F}_E(\gamma)]\!]$, by induction on the structure of e*
  - *$\bot$ is not $\bot$-less*
  - *$\eta(x)$ is $\bot$-less if x is $\bot$-less*
  - *$\mathtt{op}(p, c)$ is $\bot$-less if p is $\bot$-less and for every $x \in [\![\beta]\!]$, $c(x)$ is $\bot$-less ($\mathtt{op} : \alpha \rightarrowtail \beta \in E$)*

Definition 7.3.3. *For $\Gamma \vdash M; N : \tau$, the **interpretation** $[\![M; N]\!]$ is defined as the following function from $[\![\Gamma]\!]$ to $[\![\tau]\!]$:*

$$[\![M; N]\!](e) = \begin{cases} [\![M]\!](e), & \textit{if } [\![M]\!](e) \textit{ is } \bot\textit{-less} \\ [\![N]\!](e), & \textit{otherwise} \end{cases}$$

We now have a denotational semantics for $M; N$. We can use this denotational semantics to produce a reduction semantics for the construction.

Definition 7.3.4. *We define a binary **reduction relation** $R_;$ on the terms of $(\!|\lambda|\!)$. We write $\to_;$ for its context closure.*[141]

- *$(M; N)\ R_;\ M$ if for every Γ and τ such that $\Gamma \vdash M; N : \tau$ and for every $e \in [\![\Gamma]\!]$, $[\![M]\!](e)$ is $\bot$-less*
- *$(M; N)\ R_;\ N$ if for every Γ and τ such that $\Gamma \vdash M; N : \tau$ and for every $e \in [\![\Gamma]\!]$, $[\![M]\!](e)$ is not $\bot$-less*

Adding the $\to_;$ reduction relation into the reduction relation $\to$ of $(\!|\lambda|\!)$ preserves subject reduction (Property 3.2.2) since $M; N : \alpha$ always reduces to either $M : \alpha$ or $N : \alpha$. It also preserves the soundness of our denotational semantics (Property 3.3.8) because soundness relies on two properties: the compositionality of the denotational semantics, which we preserve, and the fact that the individual reduction rules preserve denotations, which the $\to_;$ rule does by definition. The new construction and its reduction rule also preserve confluence and termination, as we will show below, and therefore they also preserve strong normalization.

Notation 7.3.5. *The $(\!|\lambda|\!)^;$ **calculus** is the $(\!|\lambda|\!)$ calculus with the $M; N$ construction and its reduction rule, $\to_;$.*

Lemma 7.3.6. ***Termination of** $(\!|\lambda|\!)^;$*
The reduction relation $\to$ of $(\!|\lambda|\!)^;$ is terminating.

[141] The notion of evaluation context being expanded to include $C ::= C; M \mid M; C$.

Proof. We consider $(\!| \lambda |\!)$ with the $M; N$ construction and the following two reduction rules.

$$M; N \to M$$
$$M; N \to N$$

This calculus, which we will call $(\!| \lambda |\!)^{;\to}$, is not confluent but, as we will show below, it is terminating. Since the reduction relation of $(\!| \lambda |\!)^{;}$ is a subset of the reduction relation of $(\!| \lambda |\!)^{;\to}$, then the reduction relation of $(\!| \lambda |\!)^{;}$ will turn out to be terminating as well.

For every type α, we add a binary function symbol $(;_\alpha)$ of type $\alpha \Rightarrow \alpha \Rightarrow \alpha$ and typed versions of the two reduction rules above to the IDTS $\overline{(\!| \lambda |\!)_\tau}$ from 3.5, forming a new IDTS $\overline{(\!| \lambda |\!)_\tau^{;\to}}$. Since we have added no constructor symbols and the right-hand sides of the new reduction rules use no function symbols (and therefore are not recursive), we still validate the General Schema and Theorem 3.5.40 holds for $\overline{(\!| \lambda |\!)_\tau^{;\to}}$. Following the proof of Corollary 3.5.42, we know that $(\!| \lambda |\!)^{;\to}$ without η-reduction, $(\!| \lambda |\!)^{;\to}_{-\eta}$, terminates.

We now need to show that adding η-reduction still preserves termination. The original proof stood on Lemma 3.5.44, which states that we can delay η-reduction until the very end of evaluation and which relies on the idea that η-reduction never opens up a new redex which would neccessitate the use of another rule. This is still the case since η-reduction cannot make a new ;-redex appear without actually inserting a new semicolon and because the two reduction rules for $M; N$ given above are not sensitive to the structure of M and N. □

To prove that $(\!| \lambda |\!)^{;}$ is confluent, we prove that $\to_;$ is confluent and that $\to_;$ and the reduction relation of $(\!| \lambda |\!)$ commute. We prove the latter using Lemma 3.4.16, as we did for $\to_\eta$ and $(\!| \lambda |\!)_{-\eta}$ in Lemma 3.4.17, and for the former we use the following lemma, which is a special case of Proposition 1.0.2 in [75].

Lemma 7.3.7. *A relation $\to$ is confluent iff its reflexive-transitive closure $\twoheadrightarrow$ is subcommutative, i.e. if for all a, b, c such that $a \to b$ and $a \to c$, there exists a d such that $b \to^= d$ and $c \to^= d$.*

Proof. Follows from Proposition 1.0.2 in [75]. □

Lemma 7.3.8. *The $\to_;$ reduction relation is confluent.*

Proof. We will proceed by proving that $\to_;$ is subcommutative (i.e. that $\forall a, b, c.\ (a \to_;\ b \wedge a \to_;\ c) \Rightarrow \exists d.\ (b \to_;^=\ d \wedge c \to_;^=\ d)$) using structural induction on the term a, as in 3.4.17. We will consider the relative positions of the redex in the reductions $a \to_;\ b$ and $a \to_;\ c$.

1. If both reductions occurred within a proper subterm of a, then we use the induction hypothesis and the context closure of $\to_;$ (see the analogous proof of 3.4.17 for technical details).

2. If the reductions occurred in non-overlapping subterms, then we can take the common reduct d as the term in which both subterms have been reduced.

3. If the redex in $a \to_;\ b$ is the entire term a and the redex in $a \to_;\ c$ is a proper subterm of a:

   Then $a = M; N$ for some M and N. We split the proof based on whether the redex R of $a \to_;\ c$ is a subterm of M or N and whether b is equal to M or N.

   - $b = M$ and $M = C[R]$ with $R \to_;\ R'$ ($c = C[R']; N$)
     We will choose $d = C[R']$. Since $R \to_;\ R'$ and $\to_;$ is closed on contexts, we have that $b = M = C[R] \to_;\ C[R'] = d$. Since $\to_;$ preserves denotations, then $[\![R]\!] = [\![R']\!]$ and by compositionality of denotations, $[\![M]\!] = [\![C[R]]\!] = [\![C[R']]\!]$. Because of $M; N \to_;\ M$, we know that for every e, $[\![M]\!](e)$ is $\perp$-less and therefore so is $[\![C[R']]\!](e)$. Since $[\![C[R']]\!](e)$ is $\perp$-less for all e, $c = C[R']; N \to_;\ C[R'] = d$.
   - $b = N$ and $M = C[R]$ with $R \to_;\ R'$ ($c = C[R']; N$)
     We will choose $d = N$. We immediately have $b = N \to_;^=\ N = d$. By the same argument is in the previous case, we have that $[\![C[R]]\!] = [\![C[R']]\!]$. For $M; N$ to have reduced to N, $[\![M]\!](e)$ must have *not* been $\perp$-less for any e, the same being the case for $[\![C[R']]\!](e)$. Because $[\![C[R'](e)]\!]$ is not $\perp$-less for any e, we have $c = C[R']; N \to_;\ N = d$.

- $b = M$ and $N = C[R]$ with $R \to_{;} R'$ ($c = M; C[R']$)
  We will choose $d = M$. This case is symmetric to the previous one where we have $M; N$ reducing to one branch and the inner reduction happening in the abandoned branch.
- $b = N$ and $N = C[R]$ with $R \to_{;} R'$ ($c = M; C[R']$)
  We will choose $d = C[R']$. This case is symmetric to the first one in which the inner reduction ($R \to_{;} R'$) happens in the chosen branch (M in the first case, N in this one).

4. If the redex in $a \to_{;} c$ is the entire term a and the redex in $a \to_{;} b$ is a proper subterm of a:

   Symmetric to the previous case.

5. If the redex in both $a \to_{;} b$ and $a \to_{;} c$ is the entire term a, then $b = c$ and their common reduct is $d = b = c$. If b and c were different, i.e. $M; N \to_{;} M$ and at the same time $M; N \to_{;} N$, then $[\![M]\!](e)$ would have to be simultaneously both $\perp$-less and not $\perp$-less for every e, which is a contradiction.

□

Lemma 7.3.9. *The reduction relation $\to_{(\!|\lambda|\!)}$ of $(\!|\lambda|\!)$ commutes with $\to_{;}$.*

Proof. As in the proof of Lemma 3.4.17, we will make us of Hindley's Lemma 3.4.16. We will be proving that for every $(\!|\lambda|\!)^{;}$ terms a, b, c such that $a \to_{(\!|\lambda|\!)} b$ and $a \to_{;} c$, there exists a d such that $b \twoheadrightarrow_{;} d$ and $c \to^{=}_{(\!|\lambda|\!)} d$. The proof will be analogous to the one of Lemma 3.4.17, proceeding by induction on the structure of a and considering the relative positions of the redexes in the reductions $a \to_{(\!|\lambda|\!)} b$ and $a \to_{;} c$.

- If both reductions occur in a proper subterm of a, then we use the induction hypothesis and the context closure of the two relations.

- If the reductions occur in non-overlapping subterms of a, then the common reduct d is a term in which both subterms have been reduced.

- If the redex in $a \to_{(\!|\lambda|\!)} b$ is the entire term a and the redex in $a \to_{;} c$ is some proper subterm of a:

  Let $l \to r$ be the $(\!|\lambda|\!)$ rule used in $a \to_{(\!|\lambda|\!)} b$. None of the left-hand sides of the reduction rules in $(\!|\lambda|\!)$ contain the semicolon operator and so the ;-redex in a must be matched by some metavariable in l. Let M be that metavariable. We decompose the left-hand side l into $L(M)$ and the right-hand side r into $R(M)$. We have $a = L(a')$, $b = R(a')$ and $a' \to_{;} a''$ with $c = L(a'')$ for some a' and a''. Our common reduct d will be $R(a'')$. For b, we have $b = R(a') \twoheadrightarrow_{;} R(a'') = d$ by $a \to_{;} a'$ and the context closure of $\to_{;}$. Note that $R(a')$ might contain multiple copies of a' or maybe even no copies of a' (as is the case when rules copy or delete terms). No matter the number of copies, we always have $R(a') \twoheadrightarrow_{;} R(a'')$, because $\twoheadrightarrow_{;}$ is reflexive and transitive. For c, we have $c = L(a'') \to_{(\!|\lambda|\!)} R(a'') = d$ because $L(a'') \to R(a'')$ is an instance of the $(\!|\lambda|\!)$ rule $l \to r$.

- If the redex in $a \to_{;} c$ is the entire term a and the redex in $a \to_{(\!|\lambda|\!)} b$ is some proper subterm of a:

  Let $a = M; N$. We proceed the same way as in case 3 in the proof of Lemma 7.3.8. We have an outer ;-reduction and an inner denotation-preserving reduction. Performing the inner reduction $a \to_{(\!|\lambda|\!)} b$ first does not change the denotation and so we can still apply the ;-reduction $b \to_{;} d$. Performing the outer reduction $a \to_{;} c$ first either throws away the redex for the $(\!|\lambda|\!)$ reduction and so we have $c = d$, or it preserves it whole and then we can still perform the reduction $c \to_{(\!|\lambda|\!)} d$.

- a is at the same time a $(\!|\lambda|\!)$-redex and a ;-redex:

  This is impossible because there is no rule in $(\!|\lambda|\!)$ whose left-hand side is headed by the $M; N$ construction, which is, on the other hand, the case in every ;-redex.

□

Corollary 7.3.10. ***Confluence of*** $(\!|\lambda|\!)^{;}$
The reduction relation of $(\!|\lambda|\!)^{;}$ is confluent.

Proof. Corollary of Lemma 7.3.8, Lemma 7.3.9 and the Lemma of Hindley-Rosen (Lemma 3.4.15). □

Theorem 7.3.11. ***Strong normalization of*** $(\!|\lambda|\!)^{;}$

$(\!|\lambda|\!)^{;}$ is strongly normalizing, i.e. there are no infinite reduction chains in $(\!|\lambda|\!)^{;}$ and all maximal reduction chains originating in a $(\!|\lambda|\!)^{;}$ term M terminate in the same term, the normal form of M.

Proof. Corollary of Lemma 7.3.6 and Corollary 7.3.10. □

7.3.5 Comparison with TTDL

The $\mathcal{C}$ construction introduced partiality to $(\!|\lambda|\!)$. With the $M; N$ construction in our calculus, we now have a way to react to partiality, to avoid stuck terms. We have seen how to use that feature together with effects and handlers to implement McCarthy's ambiguous operator amb and then use that to implement the presupposition accommodation strategy used in Lebedeva's extension of TTDL [80], which projects presupposition as widely as possible without breaking variable binding. Our solution improves in some aspects on Lebedeva's work:

- The underlying calculus is strongly normalizing.

  Lebedeva extends the simply-typed λ-calculus with (unrestricted) exceptions and the resulting calculus is non-terminating, as shown by Lillibridge's encoding of recursive types [85]. Our first attempt at formalizing $(\!|\lambda|\!)$ actually had the same deficiency and we were able to encode recursive types using Lillibridge's method.[142] Furthermore, when Lebedeva sketched out a possible solution to the binding problem, she defined a recursive function which was then part of the lexical semantic entries.[143] This is also problematic since general recursion precludes termination. Our calculus gets around this by relying on inductive types to provide a limited form of recursion which can be proven terminating.

  The $(\!|\lambda|\!)$ calculus, as it is presented in this manuscript, is terminating and strongly normalizing, while still allowing us to implement the accommodation strategy used by van der Sandt [130] and Lebedeva [80].

- Our analysis fixes a bug which allowed presupposition triggers to bind pronouns preceding them.

  The exceptions used in Lebedeva's extension of the simply-typed λ-calculus are not resumable. If a referential noun phrase triggers a presupposition about the existence of its referent, the evaluation of the sentence is restarted with the presupposed referent now in the context. However, this overgenerates by allowing any anaphoric expressions in the same sentence to bind to that referent, even those expressions which precede the presupposition trigger.

  (27) * He$_1$ loves John's$_1$ car.

  The operations in $(\!|\lambda|\!)$ are resumable exceptions. Since we can resume the interpretation of a sentence after accommodating the presupposition, we do not have to restart the evaluation of the sentence and therefore we avoid the above situation.

- We treat presuppositions using the same formal apparatus as dynamics and anaphora.

  Lebedeva uses terms to encode dynamic propositions, following de Groote's schema [38]. The terms and their types correspond to a monad of state and continuations. When this theory is then extended to cover presuppositions, instead of augmenting the monad to include another effect, the

[142] The non-terminating $(\!|\lambda|\!)$ assumed a global effect signature E and used computation types $\mathcal{F}(\alpha)$ which all shared the effect signature E (i.e. every $\mathcal{F}(\alpha)$ was implicitly $\mathcal{F}_E(\alpha)$). This was problematic because one could use the type $\mathcal{F}(\alpha)$ somewhere in the type of one of the operations in E. This created a loop where E explicitly referenced $\mathcal{F}(\alpha)$ which implicitly referenced E and it is this loop which is exploited by Lillibridge's encoding of recursive types. This no longer works in the version of $(\!|\lambda|\!)$ presented in this manuscript since the type $\mathcal{F}_E(\alpha)$ now has to explicitly mention E and if E contains $\mathcal{F}_E(\alpha)$, then it leads to an infinite type. This was sufficient to make Lillibridge's encoding impossible since we have proven $(\!|\lambda|\!)$ terminating in 3.5.

[143] The function in question is the iacc handler for intermediate accommodation from Definition 6.29. This handler is part of the definition of the dynamic existential quantifier, which is the only operator using which new variables are introduced.

underlying λ-calculus is replaced by an impure calculus with order of evaluation, exceptions and handlers.

Our approach is based on and heavily inspired by Lebedeva's use of exception mechanisms to treat presupposition but instead of using a term encoding for dynamicity and side effects, such as exceptions, for presupposition, we use a free monad for everything.[144] Our free monad is a term encoding of a computation, much like de Groote's original approach was a term encoding of a dynamic proposition, which means that our calculus has no fixed order of evaluation (i.e. is pure). The free monad lets us combine both the dynamic effects of manipulating a state and a continuation together with the effect of throwing exceptions for presuppositions in a single encoding with relative ease.

7.4 Double Negation

We now turn our attention to another extension of TTDL. In his thesis [111], Sai Qian considers examples such as the following one from Barbara Partee [112]:

(28) Either there's no bathroom$_1$ in the house, or it's$_1$ in a funny place.

The bathroom mentioned in the first clause is embedded under a negation but nevertheless, we can still access it in the second clause. If we look at the disjunction $A \overline{\vee} B$ in DRT and TTDL, it has the same truth conditions and accessibility characteristics as $\overline{\neg}(\overline{\neg} A \overline{\wedge} \overline{\neg} B)$. This is how TTDL defines dynamic disjunction. If we take the meaning of *there's no bathroom* to be $\overline{\neg} A'$ where A' is the meaning of *there is a bathroom*, then the meaning of Example 28 comes out as:

$$\overline{\neg} A' \overline{\vee} B = \overline{\neg}(\overline{\neg}(\overline{\neg} A') \overline{\wedge} \overline{\neg} B)$$

In classical logic, we may use the law of double negation to go from $\overline{\neg}(\overline{\neg} A')$ to A'.

$$\overline{\neg}(\overline{\neg}(\overline{\neg} A') \overline{\wedge} \overline{\neg} B) = \overline{\neg}(A' \overline{\wedge} \overline{\neg} B) = A' \overline{\Rightarrow} B$$

If we do so, the dynamic contributions of A' will no longer be blocked by a negation and will become accessible to any anaphoric pronouns in B. As we see above, the sentence ends up being paraphrased as "*if there's a bathroom, then it's in a funny place*".

In order for this line of reasoning to hold within the framework, we would need the law of double negation to hold. However, that is not the case in neither DRT nor TTDL. For DRT, this problem is addressed by Krahmer and Muskens in Double Negation DRT (DN-DRT) [78] and for TTDL with a very similar strategy by Qian in Double Negation TTDL (DN-TTDL) [111].

The DN-TTDL approach is an instance of the following general strategy. Let us imagine we have some set A (e.g. the set of DRSs or TTDL dynamic propositions) and a function $f : A \to A$ (e.g. negation) that we would like to adapt into some involution g.[145] Applying g to a value once should have the same (or somehow similar) effect as applying f once (i.e. g should simulate/extend f). Furthermore, applying g twice must act as the identity function. We can consider an extended domain $A \times A$ and for each value $x \in A$ pairs $i(x)$ of the form $\langle x, f(x) \rangle$. The swapping operation $\lambda \langle a, b \rangle \,.\, \langle b, a \rangle$ is then just the function g we were looking for. We have the following:

$$\begin{aligned} \pi_1(i(x)) &= x \\ \pi_1(g(i(x))) &= f(x) \\ \pi_1(g(g(i(x)))) &= \pi_1(i(x)) = x \end{aligned}$$

The first equation shows us that the i injection is a right inverse to the π_1 projection. The second equation shows us that using g in the extended domain $A \times A$ is exactly like using f in the original domain A. Finally, the third equation shows us that g is an involution.

[144] Not only for dynamicity and presuppositions, but also for quantification, conventional implicature and deixis.

[145] An *involution* is a function which is its own inverse, i.e. $g(g(x)) = x$.

This is exactly the approach adopted by Qian in [111]. The original domain is the type of dynamic propositions $\Omega = \gamma \to (\gamma \to o) \to o$ and the function f is TTDL's dynamic negation. The domain of DN-TTDL is defined as the type $\Omega \times \Omega$, propositions are injected from TTDL to DN-TTDL using $i(x) = \langle x, f(x) \rangle$ and DN-TTDL dynamic negation works by swapping the two elements of the pair. We can conceive of this pair as the positive and negative representation of a proposition, the second item of the pair always being a negated form of the first one.

DN-DRT [78] adopts a similar strategy. Two interpretation functions are given: one for positive readings and another for negative readings. Negation becomes a new constructor for DRSs, its positive interpretation being the negative interpretation of its argument and its negative interpretation being the positive interpretation of its argument. Therefore, every DRS has two interpretations and negation is implemented as exchanging these two interpretations.

7.4.1 Double Negation as an Effect

Our methodology is built on the assumption that the phenomena that we treat using effects have a projective nature. If a certain construction does not play a role in the phenomenon, it should transfer any operations related to this phenomenon from its arguments up to its context. For example:

- An indefinite noun phrase like *a man* introduces a new discourse referent x along with the condition **man** x. If this NP becomes an argument of a verb, then the verb applies itself to the NP's referent and transfers these dynamic effects up to the nearest enclosing box. The meaning of *a man sleeps* still introduces a new discourse referent x along with the condition **man** x while also introducing the condition **sleep** x. Since the verb had no interaction with dynamicity, all of the NP's effects were preserved.

- An anaphoric pronoun such as *he* accesses the context to find its antecedent x. Applying the predicate *sleeps* preserves this context dependence and yields a computation that also starts by accessing the context to find an antecedent x and then producing the proposition **sleep** x. Deictic and intensional meanings access some external information to find their referents as well and they behave the same in these scenarios.

- A definite description such as *the king of France* presupposes the existence of a king of France x and again, we can compute the meaning of *the king of France is bald* by applying the predicate **bald** to the referent x while still presupposing the existence of the king of France x.

Lexical items that want to interact with a certain phenomenon, a certain level of meaning, do so by either signalling an operation or handling one. The entries of all the other lexical items which are not involved in this phenomenon do not have to be modified in any way and they project these requests automatically. As we have seen in the last two chapters, for many phenomena this is the case. This way, semantics of deictic NPs are not affected by the existence of anaphoric NPs or presuppositional NPs, and neither of these affect the semantics of simple predicates (e.g. extensional transitive verbs denoting relations on individuals such as *loves*).

Now when we look at the treatment of double negation as an effect, it does not follow the same pattern. The change in types in TTDL is from Ω to $\Omega \times \Omega$. If we want to treat this as an effect, we can look at monads which employ similar types. Two come to mind: the writer monad that maps the type α to $\alpha \times \Omega$ and the reader monad that maps the type α to $2 \to \alpha$.

Polarity Sensitivity — Reader Monad

In the reader monad approach, we look at the type $\Omega \times \Omega$ as a family of dynamic propositions of type Ω indexed by the type 2 (i.e. $\Omega \times \Omega \simeq 2 \to \Omega$). These computations live in some context in which they have access to a polarity of type 2 and based on that polarity, they should either return a positive version of themselves or a negative one. The corresponding operation would be something like `get_polarity` : $1 \rightarrowtail 2$. We will see that such an approach is problematic.

Let us look at the VP "*trusts nobody*" whose meaning is a function from individuals to polarity-sensitive propositions, that when given an individual x will ask for the polarity and if it is positive, return "x *trusts nobody*", and if it is negative, return "x *trusts somebody*". If we then embed this VP inside the sentence "*I met a man who trusts nobody*", the resulting meaning would ask for the polarity and if it is positive, return the meaning of "*I met a man who trusts nobody*", and if it is negative, return "*I met a man who trusts somebody*", which is not the desired negation of the sentence. We do not get the correct meaning by ignoring the `get_polarity` operation and applying the meaning of the context "*I met a man who* []" to (either of) the results.

If a proposition occurs in some context which is not a negation, then it appears with positive polarity and we should apply a handler which signals that. However, we would have to include this handler in *all* lexical entries that embed other propositions as arguments, which would defeat the point of our method. Futhermore, if we would end up implementing negation as a handler that switches polarity, then we would be obliged to also change *all* lexical entries that produce propositions so that they do not forget to ask for polarity. The η injection of the reader monad maps a dynamic proposition A to a function $\lambda p.\, A$, which ignores the polarity p and assigns the dynamic proposition A to both. This is clearly not what we want as it ends up making every dynamic proposition equal to its negation.

Providing Negations — Writer Monad

The writer monad is about computations outputting something: in our case they would be outputting their suggested negations, as in the technique for building up involutions using pairs shown above. This would correspond to some operation `negative` $: \Omega \rightarrowtail 1$ with which a proposition would suggest to its context a preferred negation that ought to be used if someone would try to negate it.

In this case, the VP "*trusts nobody*" would be a function from individuals x to dynamic propositions "x *trusts nobody*" that would suggest "x *trusts somebody*" as their negation. If we then look at the complex sentence "*I met a man who trusts nobody*", its meaning would be a computation that produces the meaning of "*I met a man* x *who trusts nobody*" but also suggests that "x *trusts somebody*" is its negation. Again, it becomes incorrect to ignore the `negative` effect in constructions that embed propositions. Furthermore, what would be the semantics if a proposition used the `negative` effect multiple times? The writer monad expects the type of the material being to form a monoid, which would be difficult to arrange in our case.

7.4.2 DN-TTDL is Not Monadic

So far, we have tried taking existing monads which superficially look like DN-TTDL (i.e. they use the same types) and we have seen that neither models the semantics of DN-TTDL faithfully, forcing us to explicitly introduce handlers in all lexical entries that work with propositions, forfeiting the benefit of using ⦇ λ ⦈ computations in the first place. In this and the next subsection, we will try to find out why monads and effects are not a good fit for DN-TTDL.

We will start by showing that the constructions in DN-TTDL are not monadic. We have seen the definition of a monad in 3.3.6: we need to provide a functor F and the combinators η and $\gg\!=$ which satisfy the monad laws.

$$
\begin{aligned}
F(\Omega) &= \Omega \times \Omega \\
\eta\, A &= \langle A, \neg A \rangle \\
A \gg\!= f &= f\,(\pi_1\, A)
\end{aligned}
$$

This captures the idea behind the DN-TTDL approach:

- the type of propositions is generalized to become the type of pairs of positive and negative propositions,
- older values (simple dynamic propositions) are lifted by creating pairs in which the second item corresponds to the dynamic negation of the original value

- if we want to apply a function f on some DN-TTDL proposition, we extract its positive variant and use that as the argument of f

We can check that our translation is faithful by trying to use the two combinators above to automatically raise operators such as dynamic conjunction into the new double-negation theory. We can reuse the general monadic lifting functions from 6.1.

$$\begin{aligned}
\mathsf{lift}^{\mathsf{L}}_{\alpha \multimap \beta \multimap \gamma} &: ([\![\alpha]\!] \to [\![\beta]\!] \to [\![\gamma]\!]) \to (F([\![\alpha]\!]) \to F([\![\beta]\!]) \to F([\![\gamma]\!])) \\
\mathsf{lift}^{\mathsf{L}}_{\alpha \multimap \beta \multimap \gamma}\, f &= \lambda XY.\, X \gg\!= (\lambda x.\, Y \gg\!= (\lambda y.\, \eta\, (f\, x\, y))) \\
&= \lambda XY.\, \langle f\, (\pi_1\, X)\, (\pi_1\, Y), \neg(f\, (\pi_1\, X)\, (\pi_1\, Y))\rangle \\
\mathsf{lift}^{\mathsf{L}}_{S \multimap S \multimap S}\, (\overline{\wedge}) &= \lambda XY.\, \langle (\pi_1\, X)\, \overline{\wedge}\, (\pi_1\, Y), \neg((\pi_1\, X)\, \overline{\wedge}\, (\pi_1\, Y))\rangle
\end{aligned}$$

By applying $\mathsf{lift}^{\mathsf{L}}_{S \multimap S \multimap S}$ to dynamic conjunction, we arrive at the extended definition of conjunction used by Qian in [111]. Thus we manage to derive from the general monadic lifting function and the two lines that define η and $\gg\!=$ for this monad the same operator that Qian introduced in his thesis. This seems to suggest that this translation is indeed a faithful one.

However, there are two problems with the "monad" we just introduced. First of all, η is not general and is only applicable to values of type Ω (dynamic propositions). It begs the question what would be the interpretation of types such as $F(\iota)$, i.e. what is the negation of an individual. Secondly, even if we ignore this and just stay in the domain of dynamic propositions, permitting only types $F(\Omega)$, we run into a more severe problem. The "monad" that we proposed does not actually satisfy the right identity law (Law (3.12)).

$$X \gg\!= \eta = X$$

The reason is simple. The $\gg\!=$ operator forgets about the negative form of X and then η replaces that negative form with the default one produced by dynamic negation. The two terms are thus not equal, since the one on the left replaces the proposed negative form X with the default dynamic negation.

In our method, we model denotations as $(\!|\, \lambda \,|\!)$ computations and $(\!|\, \lambda \,|\!)$ computations form a monad, i.e. for any effect signature E, $\langle \mathcal{F}_E, \eta, \gg\!= \rangle$ is a monad. Likewise, all of the phenomena we treat have a monadic structure as well: Barker's continuization uses the continuation monad [12], de Groote and Lebedeva's TTDL uses a combination of the state monad and the continuation monad [38, 80] (see Subsection 5.4.2), de Groote and Kanazawa's intensionalization uses the reader monad [40], Giorgolo and Asudeh's implementation of Potts' conventional implicature analysis uses the writer monad [51]. The fact that the structure used by Qian [111] is not a monad could be a reason why its implementation using $(\!|\, \lambda \,|\!)$ effects is difficult.

We have shown that DN-TTDL is not monadic, but there are also other abstractions which have weaker laws than monads and which are still useful. The applicative functors (see Subsection 3.3.5) that are used in Oleg Kiselyov's Applicative Abstract Categorial Grammars [68, 69] are one such example. However, as we will show next, DN-TTDL also violates the laws of functors, which are even weaker than those of applicative functors.

7.4.3 DN-TTDL is Not Functorial

A functor F maps types α to types $F(\alpha)$ and functions $f : \alpha \to \beta$ to functions $F(f) : F(\alpha) \to F(\beta)$. The functor underlying the almost-monad from the previous subsection is the following:

$$\begin{aligned}
F(\Omega) &= \Omega \times \Omega \\
F(f) &= \lambda X.\, \langle f\, (\pi_1\, X), \neg(f\, (\pi_1\, X))\rangle
\end{aligned}$$

Same as for the monad, this is still applicable only to functions whose return type is Ω. However, even if we ignore this, we find out that this structure does not satisfy the laws of a functor. Functors

have to adhere to two laws: homomorphism w.r.t. to function composition and homomorphism w.r.t. to identities. It is the latter which this structure fails. Consider the identity law of functors (Law (3.5)):

$$F(\mathrm{id}_A) = \mathrm{id}_{F(A)}$$

If we try to elaborate $F(\mathrm{id}_\Omega)$ with this structure, we find that it does not correspond to an identity:

$$\begin{aligned} F(\mathrm{id}_\Omega) &= \lambda X.\ \langle \pi_1\, X, \neg(\pi_1\, X)\rangle \\ \mathrm{id}_{F(\Omega)} &= \lambda X.\ \langle \pi_1\, X, \pi_2\, X\rangle \\ F(\mathrm{id}_\Omega) &\neq \mathrm{id}_{F(\Omega)} \end{aligned}$$

Therefore, the structure in DN-TTDL is not a functor and by extension neither an applicative functor.

The failure in the functor identity law highlights why this kind of strategy is incompatible with our approach. It is exactly because of the fact that the negative variants of propositions are not things that project — if A_{n} is the negative variant of A, that does not mean that A_{n} is the negative variant of $A \wedge B$ — that this structure breaks the functor laws. In order to stop the projection of the proposed negative variant out of a context, the functor forgets the old negative variant and replaces it with a new one. However, this kind of behavior that suppresses this extra information is non-functorial, since if we apply the functor to the identity function, we get a function that forgets the extra annotation (the negative variant) and therefore we do not have an identity function.

When we say that this solution to the double negation problem is incompatible with our approach, we mean that we cannot find effects that implement this kind of functionality. However, this does not mean that we cannot combine our technique of using ⦇ λ ⦈ computations with the technique proposed by Qian [111]. We have seen how to implement TTDL in ⦇ λ ⦈, on which DN-TTDL is built. We can therefore use the DN-TTDL schema while replacing the TTDL implementation with our ⦇ λ ⦈ implementation. The downside is that we have to then always work with pairs of computations of propositions everywhere, which is a heavy price to pay for the offered empirical coverage increase.

7.5 Summary

In this chapter, we have seen how to implement dynamic semantics in the ⦇ λ ⦈ framework. Our goal was to motivate a core set of operations which characterize dynamic meanings. In order to do that, we have taken DRT, a well-established theory of dynamic semantics, and shown how its canonical formulation in Kamp and Reyle's textbook [64] can be seen as a collection of effectful programs (7.1). This validated our intuition that dynamic semantics can be suitably modelled as effectful computation and also shown us what effect signature to choose and how to use those effects to give denotations to lexical items in our grammar (7.2). In translating the DRT fragment into an ACG+⦇ λ ⦈ fragment, we have made it compositional by showing that what DRT actually composes are instructions on how to build DRSs. We have then connected our analysis to de Groote's Type-Theoretic Dynamic Logic [38, 80] by interpreting the computations as dynamic propositions. Finally, inspired by the algebraic effects literature [60, 103, 110, 104], we have given a system of equations which derives DRSs as the canonical forms of dynamic computations (a collection of variables accessible to subsequent formulas coupled with a collection of simple propositions about those variables) (7.2.5).

After having introduced anaphora, we moved onto another effect that is closely related: presuppositions (7.3). Our treatment of presuppositions, which was in the style of Lebedeva [80] and van der Sandt [130], was built on top of our dynamic semantics and was the first time we got to see how to combine side effects from different phenomena. We first started with changing the dynamic effects handler from a closed handler into an open one (7.3.1). While this involved some complicated λ-terms, this is a price we will have to pay only once. Now that we have an open handler for dynamics, we can combine dynamics with other effects with much less effort.

Then we took the following steps to integrate presuppositions into our dynamic semantics:

1. We have fixed a bug in the dynamics handler (box) which relied on the assumption that the contents of DRSs higher up on the projection line than the closest immediate DRS will not change; an assumption that would be invalidated by the introduction of presuppositions (7.3.1). While without fixing this, our analysis was undergenerating, Lebedeva's analysis [80] suffers from a similar bug and is overgenerating (licensing the reading "He$_1$ loves John's$_1$ car").

2. We introduced a new operation presuppose : $(\iota \to o) \rightarrowtail \iota$ and used it in lexical entries such as PROPER for proper names and POSS for possessive constructions.

3. We introduced a handler for presuppose called accommodate and we added it to a handler representing the top-level DRS. At this point, we could analyze discourses such as Example 20, repeated below, which were not covered by the original TTDL [38] and which motivated the use of exceptions in Lebedeva's extension [80].

   (20) It is not the case that Jones$_1$ owns a Porsche. He$_1$ owns a Mercedes.

4. We introduced a new handler, useFind, that tried resolving presuppositions by looking up the referent in the context instead of always presupposing its existence, as in [80]. This let us analyze cases such as Example 22 without projecting a presupposition.

   (22) If John owns a car, then his car is cheap.

5. We enriched the box handler with a clause for nondeterministically accommodating presuppositions, maybeAccommodate. This allowed us to account for the ambiguity inherent in the accommodation of presuppositions, such as in Example 24:

   (24) (c_0) Maybe (c_1) Wilma thinks that (c_2) her husband is having an affair.

6. We have encoded the Preference for Global Accommodation principle and its binding problem constraint by introducing an operator to $(\!| \lambda |\!)$ for identifying stuck computations and by writing a handler for nondeterministic choice that gave it the semantics of McCarthy's amb operator [92]. With this, we analyzed Example 26:

   (26) (c_0) If (c_1) a man gets angry, (c_2) his children get frightened.

Notably, we did not need to touch the dynamic denotations that we have developed in 7.2 in order to make them presupposition-compatible. We only modified dynamic negation, $\overline{\neg}$, by adding new clauses to the box handler to implement the interaction between DRSs and presuppositions. Our objective was to have empirical parity with Lebedeva's analysis [80] while using the same mechanism to implement both dynamics and presuppositions ($(\!| \lambda |\!)$ computations) and without having to use a calculus with a non-standard evaluation order. In this, we have succeeded[146] and we have also shown an example in which we can overcome an empirical limitation of Lebedeva's approach.

Finally, we addressed another extension of TTDL, Qian's DN-TTDL [111]. We have analyzed his proposal and shown that it is out of the scope of our approach as it involves a non-functorial structure at its core. While we cannot translate Qian's technique into effects and handlers, we can use his technique directly to model propositions as pairs of computations. As this would be a heavy change to our fragment, we avoid this technique in the rest of the manuscript.

[146] We have not covered the interplay of quantification and dynamics, which TTDL studies as well. This will be the subject of Chapter 8.

8

Composing the Effects

In Chapters 6 and 7, we have seen $(\!| \lambda |\!)$ analyses of deixis (6.2), conventional implicature (6.3), quantification (6.4), anaphora (7.2) and presupposition (7.3). All of those analyses shared the same structure: constituents of atomic abstract type were interpreted as $(\!| \lambda |\!)$ computations.[147] The fact that there is a common underlying structure present in all of these phenomena has already been discovered by the pioneers of monads in natural language semantics [113, 27, 50, 13]. However, the point of expressing all of these phenomena in a single framework is not just to have a uniform set of fragments, but to also be able to combine those fragments into a wider picture.

Most recently, proposals for combining monads in natural language semantics have started appearing [27, 50, 13]. We can divide them into two strategies:

- Constructing a "supermonad" that combines the structural elements of all the monads that we want to use.

  The denotations of constituents are then all computations within this monad (or their denotations are injected into this supermonad). The monad's $\gg\!=$ operator serves as the universal glue, allowing us to combine any two meanings, regardless of the effects they use. This approach was adopted by Charlow in his dissertation [27].

- Making all the necessary components to build semantic glue lexical items in the grammar.

  Instead of deriving a general monad that encompasses all of the necessary monadic structure, we include the η and $\gg\!=$ of every monad we want to use into our grammar. Different lexical entries will use different (combinations of) monads and it is one of the duties of parsing to find the necessary semantic glue which combines these meanings in a sound way. This approach was proposed by Giorgolo and Asudeh in their ESSLLI 2015 course [50] and by Charlow in Barker's ESSLLI 2015 course [13, 31].

Since ACGs separate the object-level types (where we use monads and semantic types) from the abstract-level types (where we use linear implications and syntactic types and where we control the set of valid syntactic structures), the latter approach becomes impractical. If we wanted to add the monadic combinators η and $\gg\!=$ and give them their correct types, we would need to change the logic of the abstract-level type system, e.g. by including a modality for every monad. In doing this, we would no longer be working with ACGs, but with some other type-logical grammar formalism, and we would lose the benefit of existing results for ACGs (e.g. on the complexity of parsing [65]). Our approach is an instance of the former method. Our "supermonad" is the free monad $\mathcal{F}_E$ from 3.3.6 and 3.3.7. In contrast to the approach used in Charlow's dissertation [27], we explore the use of effects and handlers instead of monad transformers and we perform all computation in a formally defined object language $(\!| \lambda |\!)$, which is an extension of the simply-typed λ-calculus.

[147] Note though that in Chapter 6, we had $[\![S]\!] = \mathcal{F}_E(o)$ and after the shift to dynamic semantics in Chapter 7, we switched to $[\![S]\!] = \mathcal{F}_E(1)$.

In 7.3, we have already seen how to combine our treatment of anaphora with a treatment of presuppositions. We will extend the fragment from Chapter 7 with the effects from Chapter 6 using the same process that we have used when adding presuppositions:

1. We will translate any closed handlers used in lexical entries into open handlers. While closed handlers can often be simpler to compute with, they are no longer applicable in the presence of other effects. For an example of this process, look in 7.3.1, where we translate the box handler, which is used in entries that make use of dynamic negation, into an open handler.

2. We will update the existing entries in the grammar(s) to reflect any interactions between the lexical entries and phenomena present in both treatments. In the case of presuppositions, we have made it so that contexts can cancel presuppositions (7.3.3) and accommodate presuppositions (7.3.4) by modifying the box (and later $\overline{\text{box}}$) handler used in dynamic negation.

Contents

8.1 Dynamic Kernel **202**
8.2 Adding Presuppositions **206**
8.3 Adding Conventional Implicature **208**
8.3.1 Connection to the Standalone Theory 210
8.3.2 Connection to Layered DRT and Projective DRT 211
8.4 Adding Deixis **212**
8.5 Adding Quantification **214**
8.5.1 Quantifier Raising — Inverse Scope and Crossover 215
8.6 Considering Restrictive Relative Clauses **219**
8.6.1 Different Interpretation for Nouns 221
8.6.2 Relative Clauses and Presuppositions 222
8.7 Summary **227**
8.7.1 Note on Conservativity 228

8.1 Dynamic Kernel

As per the methodology described in 6.5, we will start with a basic "seed" grammar into which we will incorporate treatments of diverse phenomena. The grammar that we will start with will be a small dynamic fragment, very much like the one in 7.2.[148] While in 7.2, we tried to keep our dynamic grammar as close as possible to the presentation of DRT in Kamp and Reyle's textbook [64], the grammar that we will use here will be more in line with the categorial tradition. Instead of using non-lexicalized rules such as TRANS : $VP \multimap NP \multimap NP \multimap S$ and COMMON : $CN \multimap N$ (which were standing in for the DRT construction rules CR.LITV and CR.LIN), we will be using lexicalized entries such as LOVES : $NP \multimap NP \multimap S$ and MAN : N.

We first give the lexical items in our grammar and their syntactic types (i.e. the abstract signature).

[148] We could have also started with a more basic fragment without any effects and add dynamics separately. However, adding dynamics would force us to change almost all of the fragment: every individual meaning would need to add its referent to the context and every propositional meaning would need to add the proposition to the context. Furthermore, as we have seen in 7.2.4, we would also change the type of sentence interpretations from $\mathcal{F}_E(o)$ to $\mathcal{F}_E(1)$. A very similar translation from a static grammar to a dynamic one is described in [80] (Definition 4.27).

$$
\begin{aligned}
\textsc{she, he, it} &: NP \\
\textsc{a} &: N \multimap NP \\
\textsc{man, woman, Porsche, Mercedes} &: N \\
\textsc{loves, owns, fascinates} &: NP \multimap NP \multimap S \\
\textsc{not-the-case} &: S \multimap S \\
\textsc{and, if-then, either-or} &: S \multimap S \multimap S \\
\_\cdot\_ &: D \multimap S \multimap D \\
\textsc{nil} &: D
\end{aligned}
$$

This grammar is about dynamic semantics and anaphora. Therefore, it contains indefinites (indefinite article + common nouns) and pronouns to showcase the introduction and retrieval of discourse referents. It also contains entries for verbs which let use these noun phrases inside of sentences. One of the key issues of dynamic semantics is how anaphora interacts with logical operators. For these purposes, we include in our grammar lexical items corresponding to constructions that mimic the logical operators of negation, conjunction, implication and disjunction. Finally, dynamic semantics studies how anaphora works across sentences, through a discourse. We treat discourse as a list of sentences. We introduce an atomic abstract type D of discourses, an empty discourse $\textsc{nil} : D$ and a discourse extension operator $\_\cdot\_ : D \multimap S \multimap D$.

We will now give a semantic interpretation to the abstract language generated by these items. First, we define the object signature that will be the target of our interpretation.

$$
\begin{aligned}
\top, \bot &: o \\
\neg &: o \to o \\
(\_ \wedge \_), (\_ \to \_), (\_ \vee \_) &: o \to o \to o \\
\exists, \forall &: (\iota \to o) \to o \\
\_ = \_ &: \iota \to \iota \to o \\
\textbf{man}, \textbf{woman}, \textbf{Porsche}, \textbf{Mercedes} &: \iota \to o \\
\textbf{love}, \textbf{own}, \textbf{fascinate} &: \iota \to \iota \to o \\
\mathsf{nil} &: \gamma \\
\_ :: \_ &: \iota \to \gamma \to \gamma \\
\_ :: \_ &: o \to \gamma \to \gamma \\
\_ +\!\!+ \_ &: \gamma \to \gamma \to \gamma \\
\mathsf{sel_{he}}, \mathsf{sel_{she}}, \mathsf{sel_{it}} &: \gamma \to \iota
\end{aligned}
$$

We will have a type o of propositions, which will be built out of first-order logic (FOL) formulas. We include constants for all FOL constructors: tautology ($\top$), contradiction ($\bot$), negation ($\neg$), conjunction ($\wedge$), implication ($\to$), disjunction ($\vee$), existential quantification ($\exists$), universal quantification ($\forall$), equality on terms ($=$), unary predicates (corresponding to common nouns) and binary predicates (corresponding to transitive verbs). We aim to treat anaphoric binding but not anaphora resolution, and so we introduce constants for operations that will perform anaphora resolution: the oracles $\mathsf{sel_{he}}$, $\mathsf{sel_{she}}$ and $\mathsf{sel_{it}}$.[149] The anaphora resolution operators work on contexts, which contain all the knowledge in the common ground and all individuals available for discussion. These contexts, of type γ, are built up using nil, $::$ and $+\!\!+$, where $::$ is overloaded to work both for individuals and propositions.

Now that we have defined the object signature into which we want to interpret our abstract language, we are ready to lay down the lexicon. We interpret the atomic abstract types as computations:

[149] In Chapter 7, we presented a simplified account of DRT that omitted gender features. We add them back in a limited form to make the examples more comprehensible.

$$
\begin{aligned}
[\![NP]\!] &= \mathcal{F}_E(\iota) \\
[\![N]\!] &= \mathcal{F}_E(\iota \to o) \\
[\![S]\!] &= \mathcal{F}_E(1) \\
[\![D]\!] &= \mathcal{F}_E(1)
\end{aligned}
$$

The effect signature E is the DRT effect signature from 7.2.3

$$
\begin{aligned}
E = \{\ & \texttt{get} : 1 \rightarrowtail \gamma, \\
& \texttt{introduce} : 1 \rightarrowtail \iota, \\
& \texttt{assert} : o \rightarrowtail 1\ \}
\end{aligned}
$$

The interpretation of the constants in our abstract signature is given next:

$$
\begin{aligned}
[\![\textsc{she}]\!] &= \texttt{get} \star (\lambda e. \\
&\quad \eta\,(\mathsf{sel}_{\mathsf{she}}(e))) \\
[\![\textsc{he}]\!] &= \texttt{get} \star (\lambda e. \\
&\quad \eta\,(\mathsf{sel}_{\mathsf{he}}(e))) \\
&\vdots \\
[\![\textsc{a}]\!] &= \lambda N.\, \texttt{introduce} \star (\lambda x. \\
&\qquad N \gg\!= (\lambda n. \\
&\qquad \texttt{assert}\,(n\,x)\,(\lambda\_. \\
&\qquad \eta\,x))) \\
[\![\textsc{man}]\!] &= \eta\,\textbf{man} \\
[\![\textsc{woman}]\!] &= \eta\,\textbf{woman} \\
&\vdots \\
[\![\textsc{loves}]\!] &= \lambda OS.\,(\textbf{love} \cdot\!\gg S \ll\!\cdot\!\gg O) \gg\!= \texttt{assert}! \\
[\![\textsc{owns}]\!] &= \lambda OS.\,(\textbf{own} \cdot\!\gg S \ll\!\cdot\!\gg O) \gg\!= \texttt{assert}! \\
&\vdots \\
[\![\textsc{not-the-case}]\!] &= \lambda A.\, \neg A \\
[\![\textsc{and}]\!] &= \lambda AB.\, A \mathbin{\overline{\wedge}} B \\
[\![\textsc{if-then}]\!] &= \lambda AB.\, A \mathbin{\overline{\Rightarrow}} B \\
[\![\textsc{either-or}]\!] &= \lambda AB.\, A \mathbin{\overline{\vee}} B \\
[\![\_.\_]\!] &= \lambda DS.\, D \gg\!= (\lambda\_.\, S) \\
[\![\textsc{nil}]\!] &= \eta \star
\end{aligned}
$$

The interpretations are almost the same as the ones given in 7.2, with the following changes:

- We use the simplified entry for pronouns, derived in 7.2.5, to which we add gender markings.
- We also give interpretations to the two new constants, _._ and NIL. We interpret the empty discourse NIL as a discourse which contributes nothing, a trivial computation that immediately returns the dummy value $\star$.[150] The discourse extension operator _._ is interpreted the same as the dynamic conjunction $\overline{\wedge}$, by chaining the evaluation of its constituents.

[150] $\eta \star$ is also the neutral element for dynamic conjunction.

In the interpretations, we make use of *dynamic* logical operators that work with propositions of type $\mathcal{F}_E(1)$, most of which we have seen in Chapter 7. Below, we give the definition for the complete set of first-order dynamic logical operators:

$$
\begin{aligned}
A \barwedge B &= A \gg\!= (\lambda\_.\, B) \\
\neg A &= \mathsf{box}\, A \gg\!= (\lambda a.\, \mathtt{assert!}\, (\neg a)) \\
\bar{\exists}\, x.\, A &= \mathtt{introduce} \star (\lambda x.\, A) \\
A \Rightarrow B &= \neg(A \barwedge \neg B) \\
A \barvee B &= \neg(\neg A \barwedge \neg B) \\
\bar{\forall}\, x.\, A &= \neg(\bar{\exists}\, x.\, \neg A)
\end{aligned}
$$

From their definitions, we can glean some of the dynamic characteristics of these operators:

- In $A \barwedge B$, the effects of A combine with and scope over the effects of B. The discourse referents introduced by A are therefore accessible in B, i.e. $\barwedge$ is an *internally dynamic* operator.
- The existential quantifier $\bar{\exists} : (\iota \to \mathcal{F}_E(1)) \to \mathcal{F}_E(1)$ uses the `introduce` operation to scope over its continuation. This, in combination with the previous fact, allows us to derive the key law of dynamic logic $(\bar{\exists}\, x.\, A) \barwedge B = (\bar{\exists}\, x.\, A \barwedge B)$.[151]
- The $\neg A$ dynamic negation uses the box handler to interpret the dynamic operations in A and therefore stop their projection. This makes $\neg$ an *externally static operator* since the dynamic effects of its argument do not project (i.e. are not accessible) out of the resulting proposition.
- The last three operators are all headed by $\neg$ and are therefore all externally static.
- The left conjunct in the definition of $A \barvee B$ is negated and $\barvee$ is therefore internally static (discourse contributions of A are not accessible in B). On the other hand, the left conjunct in the definition of $A \Rightarrow B$ is not negated and $\Rightarrow$ is therefore internally dynamic (as in the example "*If John owns a car*$_1$, *then it*$_1$ *is cheap*").

The final piece of the puzzle is the box handler, which we have defined in 7.3.1 and which we repeat here.

$$
\begin{aligned}
&\mathsf{box} : \mathcal{F}_{E \uplus E_{\mathrm{DRT}}}(1) \to \mathcal{F}_{E \uplus \{\mathtt{get}\}}(o) \\
&\mathsf{box} = \lambda A.\, (\!|\, \mathtt{get}\colon (\lambda\_k.\, \eta\, (\lambda e.\, \mathtt{get} \star (\lambda e'.\, k\, (e +\!\!+ e') \lll\!\cdot\, e))), \\
&\qquad\qquad \mathtt{introduce}\colon (\lambda\_k.\, \eta\, (\lambda e.\, \exists_{\gg}\, x.\, k\, x \lll\!\cdot\, (x :: e))), \\
&\qquad\qquad \mathtt{assert}\colon (\lambda pk.\, \eta\, (\lambda e.\, p \wedge\!\!\gg (k \star \lll\!\cdot (p :: e)))), \\
&\qquad\qquad \eta\colon (\lambda\_.\, \eta\, (\lambda e.\, \top))\, |\!|\, A) \lll\!\cdot\, \mathsf{nil}
\end{aligned}
$$

$$
\begin{aligned}
&\_ \lll\!\cdot\, \_ : \mathcal{F}_E(\alpha \to \mathcal{F}_E(\beta)) \to \alpha \to \mathcal{F}_E(\beta) \\
&F \lll\!\cdot\, x = F \gg\!= (\lambda f.\, f\, x) \\
&\exists_{\gg} : (\iota \to \mathcal{F}_E(o)) \to \mathcal{F}_E(o) \\
&\exists_{\gg}\, P = \exists \cdot\!\gg (\mathcal{C}\, P)
\end{aligned}
$$

We have described the evolution of this handler in Chapter 7, so we will not go through the details again. We will highlight just one thing, in connection to the definition of $\bar{\exists}$ above. The $\bar{\exists}$ uses the `introduce` operation and the reduction rule $\mathtt{op}.\gg\!=$ tells us that operations project out of computations ($\mathtt{op}\, M_{\mathrm{p}}\, (\lambda x.\, M_{\mathrm{c}}) \gg\!= N \to_{\mathtt{op}.\gg\!=} \mathtt{op}\, M_{\mathrm{p}}\, (\lambda x.\, M_{\mathrm{c}} \gg\!= N)$). The box handler replaces the `introduce` operation

[151] The equation follows from a single reduction using the $\mathtt{op}.\gg\!=$ rule (Property 3.1.4).

with an existential quantifier. Using `introduce` thus has the effect of installing an existential quantifier at the scope of the nearest enclosing box, much like the DRT construction rule for indefinites, CR.ID (see Figure 7.1), introduces a discourse referent to the nearest enclosing DRS.

The objective of our semantics is to assign truth conditions to sentences. These truth conditions are expressed as propositions, terms of type o. As we progress, we will be adding more and more effects to implement a compositional semantics for "non-compositional" phenomena such as anaphora, presuppositions and conventional implicatures. We will define a handler that will strip away this extra structure and give us the truth conditions of a sentence in some default context.

$$\begin{aligned}
\mathsf{empty} &: \mathcal{F}_{E \uplus \{\mathtt{get}\}}(\alpha) \to \mathcal{F}_E(\alpha) \\
\mathsf{empty} &= (\!|\, \mathtt{get}\colon (\lambda_k.\, k\, \mathsf{nil})\, |\!) \\
\mathsf{top} &: \mathcal{F}_{E \uplus E_{\mathrm{DRT}}}(1) \to \mathcal{F}_E(o) \\
\mathsf{top} &= \mathsf{empty} \circ \mathsf{box}
\end{aligned}$$

$$\text{\rotatebox{180}{t}} \circ \mathsf{top} : \mathcal{F}_{E_{\mathrm{DRT}}}(1) \to o$$

The empty handler evaluates a meaning in the empty context nil, interpreting away the `get` operation. On the other hand, the box handler interprets away the `introduce` and `assert` operations, and so by composing them, we can interpret away all the effects in E_{DRT}. The resulting handler, top, plays the role of a top-most (top-level) DRS: it is a box in an empty context (i.e. there is no other DRS that is accessible from this one).

Furthermore, if we look at the special case of the type of top when $E = \emptyset$, we get a pure computation (type $\mathcal{F}_\emptyset(o)$) as the result. This means we can use the $\text{\rotatebox{180}{$\mathsf{t}$}}$ operator to get at the resulting proposition directly. The $\text{\rotatebox{180}{$\mathsf{t}$}} \circ \mathsf{top}$ combinator (pronounced "cherry on top") gives us a formal way to associate a proposition to the denotation of a sentence or discourse (remember that $[\![S]\!] = [\![D]\!] = \mathcal{F}_E(1)$ with E currently being E_{DRT}).

We will now integrate the effects that we have seen in Chapters 6 and 7 into our dynamic grammar. We will start with the effect that we have already seen interact with anaphora in Section 7.3.

8.2 Adding Presuppositions

We will be enriching our fragment with the following referring expressions: proper names such as *John* and *Mary*, possessive constructions expressing ownership or other relations (*X's car*, *X's children*, *X's best friend*…) and definite descriptions (*the car*). Here are the new entries into our abstract signature:

$$\begin{aligned}
\textsc{John}, \textsc{Mary} &: NP \\
\textsc{poss} &: NP \multimap N \multimap NP \\
\textsc{children-of}, \textsc{best-friend} &: NP \multimap NP \\
\textsc{the} &: N \multimap NP
\end{aligned}$$

In order to give a meaning to these constructions, we will need some extra structure in our model. We will therefore add the following into our object signature:

$$\begin{aligned}
\textbf{John}, \textbf{Mary} &: \iota \to o \\
\textbf{children}, \textbf{best-friend} &: \iota \to \iota \to o \\
\mathsf{sel_P} &: (\iota \to o) \to \gamma \to \iota
\end{aligned}$$

We represent proper names in our models as predicates. The idea behind a predicate such as **John** is that **John** x should be true for any x which is called John.

Now we can describe how we extend our lexicon so that it covers the new constructions. The interpretation of the abstract types will stay the same, we will only change the effect signature to include the following two operations:

$$\begin{aligned} \texttt{presuppose} &: (\iota \to o) \rightarrowtail \iota \\ \texttt{amb} &: 1 \rightarrowtail 2 \end{aligned}$$

We extend the lexicon to the new constructions, using the new `presuppose` operation.

$$\begin{aligned} [\![\textsc{John}]\!] &= \texttt{presuppose}!\,\textbf{John} \\ &\vdots \\ [\![\textsc{poss}]\!] &= \lambda X N.\, X \gg\!= (\lambda x.\, N \gg\!= (\lambda n.\, \texttt{presuppose}!\,(\lambda y.\, n\, y \wedge \textbf{own}\, x\, y))) \\ [\![\textsc{children-of}]\!] &= \lambda X.\, X \gg\!= (\lambda x.\, \texttt{presuppose}!\,(\lambda y.\, \textbf{children}\, y\, x)) \\ &\vdots \\ [\![\textsc{the}]\!] &= \lambda N.\, N \gg\!= (\lambda n.\, \texttt{presuppose}!\, n) \end{aligned}$$

These entries are the same as the ones we have seen in Section 7.3. Besides introducing interpretations for the new lexical items, we will also modify some of the existing interpretations or combinators to reflect the interactions between the existing effects and the effect being added. In this chapter, whenever we will revise the interpretations of existing lexical items or the definitions of existing combinators, we will use the $:=$ symbol and if the right-hand side of the definition will make use of any of the symbols being redefined, those symbols will be meant to refer to the existing (old) definition.

$$\begin{aligned} &\mathsf{top} : \mathcal{F}_{E \uplus E_{\mathrm{DRT}} \uplus \{\texttt{presuppose},\texttt{amb}\}}(1) \to \mathcal{F}_E(o) \\ &\mathsf{top} := \mathsf{search} \circ \mathsf{top} \circ \mathsf{accommodate} \circ \mathsf{useFind} \\ &\quad\;\; = \mathsf{search} \circ \mathsf{empty} \circ \mathsf{box} \circ \mathsf{accommodate} \circ \mathsf{useFind} \\ &\mathsf{box} : \mathcal{F}_{E \uplus E_{\mathrm{DRT}} \uplus \{\texttt{presuppose}\}}(1) \to \mathcal{F}_{E \uplus \{\texttt{get},\texttt{presuppose},\texttt{amb}\}}(o) \\ &\mathsf{box} := \mathsf{box} \circ \mathsf{maybeAccommodate} \circ \mathsf{useFind} \end{aligned}$$

The changes proposed above account for the following features of presuppositions:

- we add useFind to all boxes (box), including also the topmost one (top), so that presuppositional expressions referring to entities already available in the context do not trigger presuppositions
- we add maybeAccommodate to the box handler because a presupposition can be accommodated in any DRS on the projection line from the point where the presupposition was triggered
- we add accommodate to the top handler because we want any presuppositions that have been neither cancelled nor (locally) accommodated to be accommodated at the top level
- we add search to the top handler so that the proposition that we recover is the most preferred available reading of the sentence (w.r.t. the presupposition accommodation ambiguity in 7.3.4)

Below, we give the definition of the handlers for the new `presuppose` operation which account for the ways a presupposition can be eliminated: global accommodation (accommodate), local accommodation (maybeAccommodate) or cancellation (useFind).

$$
\begin{aligned}
\mathsf{accommodate} &: \mathcal{F}_{E \uplus \{\mathtt{presuppose}\}}(\alpha) \to \mathcal{F}_E(\alpha) \\
\mathsf{accommodate} &= (\!| \mathtt{presuppose}\colon (\lambda P k.\, \mathtt{introduce} \star (\lambda x.\, \mathtt{assert}\,(P\,x)\,(\lambda\_.\, k\,x))) |\!) \\
\mathsf{maybeAccommodate} &: \mathcal{F}_{E \uplus \{\mathtt{presuppose}\}}(\alpha) \to \mathcal{F}_{E \uplus \{\mathtt{presuppose},\mathtt{amb}\}}(\alpha) \\
\mathsf{maybeAccommodate} &= (\!| \mathtt{presuppose}\colon (\lambda P k.\, \mathtt{presuppose}\,P\,k + \mathtt{introduce} \star (\lambda x.\, \mathtt{assert}\,(P\,x)\,(\lambda\_.\, k\,x))) |\!) \\
\mathsf{useFind} &: \mathcal{F}_{E \uplus \{\mathtt{presuppose}\}}(\alpha) \to \mathcal{F}_{E \uplus \{\mathtt{get},\mathtt{presuppose}\}}(\alpha) \\
\mathsf{useFind} &= (\!| \mathtt{presuppose}\colon (\lambda P k.\, \mathsf{find}\,P \gg\!= k) |\!)
\end{aligned}
$$

These all come from Section 7.3, as well as the definitions of find and $+$ given below.

$$
\begin{aligned}
\mathsf{find} &: (\iota \to o) \to \mathcal{F}_{E \uplus \{\mathtt{get},\mathtt{presuppose}\}}(\iota) \\
\mathsf{find} &= \lambda P.\, \mathtt{get} \star (\lambda e.\, \mathbf{case}\ (\mathsf{sel_P}\,P\,e)\ \mathbf{of}\ \{\mathsf{inl}\ x \to \eta\,x;\ \mathsf{inr}\ \_ \to \mathtt{presuppose!}\,P\}) \\
\_ + \_ &: \mathcal{F}_{E \uplus \{\mathtt{amb}\}}(\alpha) \to \mathcal{F}_{E \uplus \{\mathtt{amb}\}}(\alpha) \to \mathcal{F}_{E \uplus \{\mathtt{amb}\}}(\alpha) \\
M + N &= \mathtt{amb} \star (\lambda b.\, \mathbf{if}\ b\ \mathbf{then}\ M\ \mathbf{else}\ N)
\end{aligned}
$$

Finally, we give the handler for the amb effect.

$$
\begin{aligned}
\mathsf{search} &: \mathcal{F}_{E \uplus \{\mathtt{amb}\}}(\alpha) \to \mathcal{F}_E(\alpha) \\
\mathsf{search} &= (\!| \mathtt{amb}\colon (\lambda\_k.\, k\,\mathbf{T};\ k\,\mathbf{F}) |\!)
\end{aligned}
$$

This uses the $M; N$ notation, whose typing and reduction rules were given in Definition 7.3.1 and Definition 7.3.4, respectively.

8.3 Adding Conventional Implicature

We move to conventional implicature, which we have treated (in isolation) in Section 6.3. We consider conventional implicatures triggered by supplements: nominal appositives and supplementary relative clauses. We will use the same abstract constants as in Section 6.3, $\textsc{who}_{\mathrm{s}}$ and $\textsc{appos}$. The constant $\textsc{who}_{\mathrm{s}}$ stands for the supplementary (appositive) use of the relative pronoun *who*.

$$
\begin{aligned}
\textsc{who}_{\mathrm{s}} &: (NP \multimap S) \multimap NP \multimap NP \\
\textsc{appos} &: NP \multimap NP \multimap NP
\end{aligned}
$$

We will not be adding any new predicates or operators to the object level: the new lexical items represent new syntactic structures and function words, not new concepts.

$$
\begin{aligned}
[\![\textsc{who}_s]\!] &= \lambda CX.\ X \gg\!= (\lambda x. \\
&\qquad \mathsf{asImplicature}\,(C\,(\eta\,x)) \gg\!= (\lambda\_. \\
&\qquad \eta\,x)) \\
[\![\textsc{appos}]\!] &= \lambda YX.\ X \gg\!= (\lambda x. \\
&\qquad \mathsf{asImplicature}\,(\mathsf{eq}\,(\eta\,x)\,Y) \gg\!= (\lambda\_. \\
&\qquad \eta\,x))
\end{aligned}
$$

$$
\begin{aligned}
&\mathsf{eq} : \mathcal{F}_E(\iota) \to \mathcal{F}_E(\iota) \to \mathcal{F}_E(1) \\
&\mathsf{eq} = \lambda XY.\, X \gg\!= (\lambda x.\, Y \gg\!= (\lambda y.\, \mathtt{assert!}\,(x = y))) \\
&\mathsf{eq} = \lambda XY.\, (X \lll\!\ggg Y) \gg\!= \mathtt{assert!}
\end{aligned}
$$

$$
\begin{aligned}
&\mathsf{asImplicature} : \mathcal{F}_{E \uplus \{\mathtt{assert}, \mathtt{introduce}\}}(\alpha) \to \mathcal{F}_{E \uplus \{\mathtt{implicate}, \mathtt{introduce^i}\}}(\alpha) \\
&\mathsf{asImplicature} = (\!|\, \mathtt{assert}\colon \mathtt{implicate},\ \mathtt{introduce}\colon \mathtt{introduce^i} \,|\!)
\end{aligned}
$$

Let us compare the entries for $[\![\textsc{who}_s]\!]$ and $[\![\textsc{appos}]\!]$ with those in Section 6.3. The first reason for the difference is that we now encode truth conditions as side effects: if we evaluate the relative clause, its truth conditions are contributed to the current context. We could separate the implicated truth conditions of the embedded clause from the (asserted) truth conditions of the surrounding material by wrapping the embedded clause in a box.

$$
\begin{aligned}
[\![\textsc{who}_s]\!] \stackrel{?}{=} \lambda CX.\ X \gg\!= (\lambda x. \\
\qquad \mathsf{box}\,(C\,(\eta\,x)) \gg\!= (\lambda i. \\
\qquad \mathtt{implicate}\,i\,(\lambda\_. \\
\qquad \eta\,x)))
\end{aligned}
$$

However, if we do this, none of the discourse referents introduced within the supplement will be available in subsequent discourse. This was the behavior predicted for parentheticals by Nunberg [102]. However, in his theory of conventional implicature [108], Potts opposes this view and shows examples which seem to contradict Nunberg's position. This was further supported by corpus studies in [5]. Here is an example of anaphoric binding out of an appositive:[152]

(29) John, who nearly killed a woman$_1$ with his car, visited her$_1$ in the hospital.

Dynamic propositions contribute truth conditions ($\mathtt{assert} : o \rightarrowtail 1$) and discourse referents ($\mathtt{introduce} : 1 \rightarrowtail \iota$) to the local context. In Section 6.3, we have introduced the operation $\mathtt{implicate} : o \rightarrowtail 1$ which contributes the truth conditions of conventional implicatures to the global context. We will complement $\mathtt{implicate}$ with an operation $\mathtt{introduce^i} : 1 \rightarrowtail \iota$ for introducing discourse referents of conventional implicatures to the global context. We can now move at-issue content into the conventional implicature layer by treating $\mathtt{assert}$ as $\mathtt{implicate}$ and $\mathtt{introduce}$ as $\mathtt{introduce^i}$, which is exactly what the $\mathsf{asImplicature}$ handler does. If we review the lexical entry for $\textsc{who}_s$, we see that the at-issue content of the embedded relative clause C gets treated *as an implicature* in the embedding expression "X, *who* C".

$$
\begin{aligned}
[\![\textsc{who}_s]\!] = \lambda CX.\ X \gg\!= (\lambda x. \\
\qquad \mathsf{asImplicature}\,(C\,(\eta\,x)) \gg\!= (\lambda\_. \\
\qquad \eta\,x))
\end{aligned}
$$

In particular in Example 29, the indefinite *a woman* in the sentence "x *nearly killed a woman*" uses the $\mathtt{introduce}$ operation to establish a new discourse referent. Upon being used as an appositive clause, this

[152] Shown in [5], but, as far as we can tell, not from a corpus.

`introduce` turns into an `introduce`$^{\mathtt{i}}$, which will project to the global context, from which it will be able to bind upcoming pronouns. Therefore we get the desired binding in Example 29.

Finally, the implicatures signalled by the `implicate` and `introduce`$^{\mathtt{i}}$ operations have to be resolved somewhere. We adopt the approach of Projective DRT [131] by interpreting conventional implicatures as belonging to the topmost box.

$$\mathsf{top} := \mathsf{top} \circ \mathsf{withImplicatures}$$

$$\begin{aligned}
&\mathsf{withImplicatures} : \mathcal{F}_{E \uplus E_{\mathrm{DRT}} \uplus \{\mathtt{implicate}, \mathtt{introduce}^{\mathtt{i}}\}}(\alpha) \to \mathcal{F}_{E \uplus E_{\mathrm{DRT}} \uplus \{\mathtt{assert}, \mathtt{introduce}\}}(\alpha) \\
&\mathsf{withImplicatures} = (\!|\, \mathtt{implicate} \colon \mathtt{assert},\ \mathtt{introduce}^{\mathtt{i}} \colon \mathtt{introduce} \,|\!)
\end{aligned}$$

8.3.1 Connection to the Standalone Theory

The lexical entries and handlers introduced in this section have been quite different from the ones introduced in Section 6.3. This raises the issue of whether the analysis that was done in Section 6.3 is upheld in our extension. We will draw out the parallels between the original definitions and the new ones to show that:

- the most visible changes are due to us representing truth conditions as side effects
- the important change is the treatment of `introduce` which accounts for the binding potentinal of appositives (an interaction between anaphora and conventional implicature)

We start with the appositive relative clause constructor.

$$\begin{aligned}
[\![\textsc{who}_{\mathrm{s}}]\!] = \lambda C X.\, & X \gg\!= (\lambda x. \\
& \quad \mathsf{asImplicature}\,(C\,(\eta\, x)) \gg\!= (\lambda_. \\
& \quad \eta\, x))
\end{aligned}
\qquad
\begin{aligned}
[\![\textsc{who}_{\mathrm{s}}]\!]' = \lambda C X.\, & X \gg\!= (\lambda x. \\
& \quad C\,(\eta\, x) \gg\!= (\lambda i. \\
& \quad \mathtt{implicate}\, i\, (\lambda_. \\
& \quad \eta\, x)))
\end{aligned}$$

In the standalone treatment (seen on the right), we used a static grammar, where sentences denoted propositions. To turn the proposition into an implicature, all we had to do was to pass that proposition to the `implicate` operation. In our dynamic grammar, sentences use `assert` to convey their truth conditions and so we need to use a handler to pass these truth conditions to `implicate`, which is exactly what (the `assert` clause of) the asImplicature handler does.

Now on to nominal appositives.

$$\begin{aligned}
[\![\textsc{appos}]\!] = \lambda Y X.\, & X \gg\!= (\lambda x. \\
& \quad \mathsf{asImplicature}\,(\mathsf{eq}\,(\eta\, x)\, Y) \gg\!= (\lambda_. \\
& \quad \eta\, x))
\end{aligned}
\qquad
\begin{aligned}
[\![\textsc{appos}]\!]' = \lambda Y X.\, & X \gg\!= (\lambda x. \\
& \quad \mathsf{eq}'\,(\eta\, x)\, Y \gg\!= (\lambda i. \\
& \quad \mathtt{implicate}\, i\, (\lambda_. \\
& \quad \eta\, x)))
\end{aligned}$$

$$\begin{aligned}
&\mathsf{eq} : \mathcal{F}_E(\iota) \to \mathcal{F}_E(\iota) \to \mathcal{F}_E(1) \\
&\mathsf{eq} = \lambda X Y.\, (X \Longleftrightarrow Y) \gg\!= \mathtt{assert}!
\end{aligned}
\qquad
\begin{aligned}
&\mathsf{eq}' : \mathcal{F}_E(\iota) \to \mathcal{F}_E(\iota) \to \mathcal{F}_E(o) \\
&\mathsf{eq}' = \lambda X Y.\, X \Longleftrightarrow Y
\end{aligned}$$

We have refactored the original interpretation $[\![\textsc{appos}]\!]'$ so that the actual differences are easier to spot. Again, we see the same difference as in $\textsc{who}_{\mathrm{s}}$, where we use the asImplicature handler. The only other difference is that the condition about the equality of the referents of X and Y is not expressed as a computation that produces a proposition but as a computation that uses `assert`. This is again due to us having switched to a dynamic grammar that treats truth conditions as side effects.

Finally, we look at the handler, withImplicatures. The original handler from Section 7.3 is given below:

$$\begin{aligned}&\mathsf{withImplicatures}' : \mathcal{F}_{\{\texttt{implicate}:o\to 1\}}(o) \to o\\ &\mathsf{withImplicatures}' = \llparenthesis\, \texttt{implicate}\colon (\lambda ik.\, i \wedge k\,\star)\,\rrparenthesis\end{aligned}$$

The first step is to translate this handler into an open handler. The continuation k will now return a computation of a proposition (type $\mathcal{F}_E(o)$) and so we will use $\wedge\!\!\gg$ to conjoin the implicature i to it.

$$\begin{aligned}&\mathsf{withImplicatures}'' : \mathcal{F}_{E\uplus\{\texttt{implicate}:o\to 1\}}(o) \to \mathcal{F}_E(o)\\ &\mathsf{withImplicatures}'' = (\!|\, \texttt{implicate}\colon (\lambda ik.\, i \wedge\!\!\gg k\,\star)\,|\!)\end{aligned}$$

Our dynamic grammar expresses truth conditions using side effects and so we will replace $i : o$ with $\texttt{assert}!\, i : \mathcal{F}_{E_{\text{DRT}}}(1)$ and $\wedge\!\!\gg$ with $\overline{\wedge}$.

$$\begin{aligned}\mathsf{withImplicatures}''' &: \mathcal{F}_{E\uplus E_{\text{DRT}}\uplus\{\texttt{implicate}:o\to 1\}}(\alpha) \to \mathcal{F}_{E\uplus E_{\text{DRT}}}(\alpha)\\ \mathsf{withImplicatures}''' &= (\!|\, \texttt{implicate}\colon (\lambda ik.\, (\texttt{assert}!\, i)\, \overline{\wedge}\, k\,\star)\,|\!)\\ &= (\!|\, \texttt{implicate}\colon (\lambda ik.\, (\texttt{assert}\, i\, (\lambda x.\, \eta\, x)) \gg\!= (\lambda\_.\, k\,\star))\,|\!)\\ &= (\!|\, \texttt{implicate}\colon (\lambda ik.\, \texttt{assert}\, i\, (\lambda x.\, k\,\star))\,|\!)\\ &\approx (\!|\, \texttt{implicate}\colon (\lambda ik.\, \texttt{assert}\, i\, (\lambda x.\, k\, x))\,|\!)^{153}\\ &= (\!|\, \texttt{implicate}\colon (\lambda ik.\, \texttt{assert}\, i\, k)\,|\!)\\ &= (\!|\, \texttt{implicate}\colon \texttt{assert}\,|\!)\end{aligned}$$

For comparison, this is the withImplicatures handler that we use when adding conventional implicatures to our dynamic grammar:

$$\begin{aligned}&\mathsf{withImplicatures} : \mathcal{F}_{E\uplus E_{\text{DRT}}\uplus\{\texttt{implicate},\texttt{introduce}^{\mathtt{i}}\}}(\alpha) \to \mathcal{F}_{E\uplus E_{\text{DRT}}}(\alpha)\\ &\mathsf{withImplicatures} = (\!|\, \texttt{implicate}\colon \texttt{assert},\ \texttt{introduce}^{\mathtt{i}}\colon \texttt{introduce}\,|\!)\end{aligned}$$

The difference between the two is the `introduce` operation and how we decide to deal with it. If we contain it locally, within the implicature, we get a theory à la Nunberg [102], where referents introduced in appositives are not accessible outside of the at-issue layer. On the other hand, if we project the `introduce` operations of the appositives as $\texttt{introduce}^{\mathtt{i}}$, we get a theory which licenses the binding in Example 29.

8.3.2 Connection to Layered DRT and Projective DRT

Geurts and Maier [46] introduced *Layered DRT*, wherein every discourse referent and condition is annotated with a *label* that tells us to which layer of meaning a referent/condition belongs. The layers treated in [46] include the layers of assertions, presuppositions and implicatures. The idea is that thanks to these labels, it is possible to separate out the individual layers while still allowing binding to work from layer to layer.

There is a close correspondence between our use of effects and layered DRT:

- `assert` adds conditions to the assertion layer
- `introduce` adds discourse referents to the assertion layer
- `implicate` adds conditions to the implicature layer

[153] The variable x is of type 1, which has only one value, $\star$. However, the set of closed normal forms of type 1 also includes stuck computations. Nevertheless, our handlers for `assert` only ever return $\star$ as the output.

- `introduce`$^{\mathtt{i}}$ adds discourse referents to the implicature layer
- `presuppose` adds discourse referents and conditions to the presupposition layer

We can compare the layered DRS with the corresponding computation. Geurts and Maier [46] give the following layered DRS as the meaning of Example 30.

(30) The porridge is warm.

$$\begin{array}{|c|}\hline x_{\mathrm{p}} \\ \hline \mathbf{porridge}_{\mathrm{p}}(x) \\ \mathbf{warm}_{\mathrm{a}}(x) \\ \neg_{\mathrm{i}} \boxed{\mathbf{hot}_{\mathrm{i}}(x)} \\ \hline \end{array} \qquad \begin{array}{l} \mathtt{presuppose}\ \mathbf{porridge}\ (\lambda p. \\ \mathtt{assert}\ (\mathbf{warm}(x))\ (\lambda\_. \\ \mathtt{implicate}\ (\neg(\mathbf{hot}(x)))\ (\lambda\_. \\ \eta\,\star))) \end{array}$$

However, our use of distinct operation symbols for assertions, implicatures and presuppositions was not motivated by the separation of these layers of meaning, but by their different projectional behavior. In that, our approach is closer to Projective DRT. In Projective DRT, the DRSs are labelled and every discourse referent and condition is annotated with a pointer which either points to a label or is free. Assertions point to the containing DRS while conventional implicatures point to the topmost DRS. Presuppositions either point to a local/intermediate DRS when the presupposition is bound ("cancelled"), or they are free, in which case they project and accommodate in the topmost DRS.

$$\begin{array}{|l|}\hline \overset{1}{f \leftarrow x} \\ \hline f \leftarrow \mathbf{porridge}(x) \\ 1 \leftarrow \mathbf{warm}(x) \\ \qquad\qquad \overset{2}{} \\ 0 \leftarrow \neg \boxed{2 \leftarrow \mathbf{hot}(x)} \\ \hline \end{array} \qquad \begin{array}{l} \mathtt{presuppose}\ \mathbf{porridge}\ (\lambda p. \\ \mathtt{assert}\ (\mathbf{warm}(x))\ (\lambda\_. \\ \mathtt{implicate}\ (\neg(\mathbf{hot}(x)))\ (\lambda\_. \\ \eta\,\star))) \end{array}$$

The correspondence between Projective DRT and our approach is similar to the one with Layered DRT. Material addressed to the global DRS 0 is analysed as implicature (`implicate` and `introduce`$^{\mathtt{i}}$), material addressed to the DRS in which it appears is treated as assertion (`assert` and `introduce`) and material that is addressed to some free label f is treated as presupposition (`presuppose`). Projective DRSs might also contain pointers to labels which are neither local nor global. These presumably correspond to bound/cancelled presuppositions, which our approach treats by retrieving the presupposed referent from the context. The pointers that appear next to discourse referents and conditions faithfully describe the way that our approach will project and accommodate them:

- implicatures (pointers to 0) use `implicate` and `introduce`$^{\mathtt{i}}$ and therefore project all the way to the top handler
- assertions (pointers to the enclosing DRS) use `assert` and `introduce` and are therefore handled by the nearest enclosing box handler
- presuppositions (free pointers) use `presuppose` and can therefore be bound at any box thanks to the useFind handler (see 7.3.3) or accommodated at the top thanks to the accommodate handler (or even accommodated lower due to binding constraints, see 7.3.4)

8.4 Adding Deixis

We now move to our treatment of indexical expressions, namely of the first-person pronoun. We will be enriching the abstract signature with a new constant representing the pronoun as well as verbs for reported speech.

$$\begin{aligned}\textsc{me} &: NP\\ \textsc{said}_{\textsc{is}} &: S \multimap NP \multimap S\\ \textsc{said}_{\textsc{ds}} &: S \multimap NP \multimap S\end{aligned}$$

To give a meaning to reported speech, we will introduce into the object signature the predicate **say**:

$$\mathbf{say} : \iota \to o \to o$$

Finally, we extend the lexicon, giving definitions to the new lexical items.

$$\begin{aligned}[\![\textsc{me}]\!] &= \texttt{speaker!}\star\\ [\![\textsc{said}_{\textsc{is}}]\!] &= \lambda CS.\,(\mathbf{say} \cdot\!\gg S \ll\!\cdot\!\gg (\mathsf{box}\,C)) \gg\!= \texttt{assert!}\\ &= \lambda CS.\,S \gg\!= (\lambda s.\,(\mathsf{box}\,C) \gg\!= (\lambda c.\,\texttt{assert!}\,(\mathbf{say}\,s\,c)))\\ [\![\textsc{said}_{\textsc{ds}}]\!] &= \lambda CS.\,S \gg\!= (\lambda s.\,(\mathsf{top}\,s\,C) \gg\!= (\lambda c.\,\texttt{assert!}\,(\mathbf{say}\,s\,c)))\end{aligned}$$

$$\begin{aligned}\mathsf{top} &: \iota \to \mathcal{F}_{E \uplus E_{\text{DRT}} \uplus \{\texttt{presuppose},\texttt{amb},\texttt{implicate},\texttt{introduce}^{\texttt{i}},\texttt{speaker}\}}(1) \to \mathcal{F}_E(o)\\ \mathsf{top} &:= \lambda s.\,\mathsf{top} \circ \mathsf{withSpeaker}\,s\\ \mathsf{withSpeaker} &: \iota \to \mathcal{F}_{\{\texttt{speaker}:1 \rightarrowtail \iota\} \uplus E}(\alpha) \to \mathcal{F}_E(\alpha)\\ \mathsf{withSpeaker} &= \lambda s.\,(\!|\,\texttt{speaker}\colon (\lambda\_k.\,k\,s)\,|\!)\end{aligned}$$

The semantics of ME are exactly the same as the original ones in Section 6.2 and so is the withSpeaker handler. We also extend the top handler so that it still covers all the effects in our grammar. However, unlike with the dynamics, where we could use the handler to supply an "out-of-the-blue" context, it is more difficult to identify a default speaker. Therefore, we add an argument to top so that if someone is to recover the meaning of a sentence in our growing fragment, they will have to identify the speaker.

Finally, we will examine the entries for reported speech. Again, we have differences due to our use of side effects to convey truth conditions:

- the propositions generated by the **say** predicate need to be asserted
- the quoted sentence is evaluated down to a proposition using box

Then we have an important difference in the $\textsc{said}_{\textsc{ds}}$ entry for direct speech:

- we want to bind the speaker to the referent of the subject, s, using withSpeaker s
- we want to use box to evaluate the clause down to a proposition, same as for the $\textsc{said}_{\textsc{is}}$ entry
- we do not want pronouns or other anaphoric elements within the quoted sentence to be bound by referents from the quoting context and so we use the empty handler (see 8.1)[154]
- we do not want the presuppositions triggered by the quoted sentence to be considered as presuppositions of the report and so we accommodate them within the scope of **say**
- we also do not want the same to happen for implicatures and so we use the withImplicatures handler to ascribe them to the person being quoted

In the end, what we want is to apply a handler for all of the effects that we have introduced so far. Since we have defined the composition of all these handlers as the combinator top, we can use that in the lexical entry for $\textsc{said}_{\textsc{ds}}$. The intuition behind it is that a directly quoted sentence does not get to have any of its usual linguistic effects.

[154] A more involved treatment of quotation would use anaphora to look into the context for the situation in which the original sentence was produced and evaluate the sentence in that context.

8.5 Adding Quantification

We finish extending our fragment by adding the last of the phenomena that we have studied in Chapter 6, (in-situ) quantification. We add into our fragment the determiners *every* and *a*,[155] as well as a genitive construction with a relational noun[156] (*owner*) so that we can experiment with quantifiers embedded within quantifiers.

$$\textsc{every}, \textsc{a}' : N \multimap NP$$
$$\textsc{owner-of} : NP \multimap N$$

The object signature already contains the necessary logical material to interpret these new lexical items (logical operators and quantifiers for the determiners and the **own** relation for the relational noun). To account for how quantified noun phrases take scope over their context, we will be using the `scope` operator, as in Section 6.4.

$$\texttt{scope} : ((\iota \to \mathcal{F}_{E'}(1)) \to \mathcal{F}_{E'}(1)) \rightarrowtail \iota$$

Its type has changed from $((\iota \to o) \to o) \rightarrowtail \iota$ to $((\iota \to \mathcal{F}_{E'}(1)) \to \mathcal{F}_{E'}(1)) \rightarrowtail \iota$ to reflect the change due to using dynamic propositions $\mathcal{F}_{E'}(1)$ instead of static propositions o. Also note that the effect signature that we refer to in the type of `scope` is E'. Throughout this chapter, we use E to mean the effect signature containing all the effects introduced so far. Now, we will be adding `scope` into the effect signature E. However, if we gave it the type $((\iota \to \mathcal{F}_E(1)) \to \mathcal{F}_E(1)) \rightarrowtail \iota$, then we would be giving a circular definition of E: E is an effect signature which, among others, contains an effect `scope` : $((\iota \to \mathcal{F}_E(1)) \to \mathcal{F}_E(1)) \rightarrowtail \iota$, where E is an effect signature which, among others, contains an effect `scope`.... Therefore, we use a different effect signature in the type of `scope`, one that does not contain `scope`: $E' = E \setminus \{\texttt{scope}\}$.[157]

We can now give the lexical entries for the determiners and the relational noun. These will follow the ones from Section 6.4, modulo the use of dynamic propositions (dynamic logical operators and quantifiers, `assert`).

$$\begin{aligned}
[\![\textsc{every}]\!] &= \lambda N.\, \texttt{scope!}\,(\lambda k.\, \overline{\forall}\, x.\, \mathsf{SI}\,((N \ll\cdot\, x) \gg\!= \texttt{assert!}) \rightrightarrows k\, x) \\
[\![\textsc{a}']\!] &= \lambda N.\, \texttt{scope!}\,(\lambda k.\, \overline{\exists}\, x.\, \mathsf{SI}\,((N \ll\cdot\, x) \gg\!= \texttt{assert!}) \overline{\wedge}\, k\, x) \\
[\![\textsc{owner-of}]\!] &= \lambda Y.\, Y \gg\!= (\lambda y.\, \eta\,(\lambda x.\, \mathbf{own}\, x\, y))
\end{aligned}$$

$$\begin{aligned}
\mathsf{SI} &: \mathcal{F}_{E \uplus \{\texttt{scope}\}}(1) \to \mathcal{F}_E(1) \\
\mathsf{SI} &= (\!|\, \texttt{scope}\colon (\lambda ck.\, c\, k)\, |\!)
\end{aligned}$$

$$\mathsf{top} := \lambda s.\, \mathsf{top}\, s \circ \mathsf{SI}$$

We can check the types of $[\![\textsc{every}]\!]$ and $[\![\textsc{a}']\!]$, namely their uses of `scope`. We know that the use of `scope` is allowed only when its argument is guaranteed *not* to use `scope`. The continuation k that we get from SI is guaranteed to be free of `scope` since SI will have handled any occurrences of `scope` within before passing it to our scope-taker. The denotation of the noun N can trigger its own effects, including `scope`,

[155] We already have a lexical entry for the indefinite article from our theory of dynamics. We include another one, which will turn out to be equivalent in most situations, to parallel the one we had in Section 6.4.

[156] When studying presuppositions, we introduced entries for genitive constructions with relational nouns that had the syntactic type $NP \multimap NP$. Those correspond to referring expressions such as *X's children* or *X's best friend*. The lexical entry $\textsc{owner-of} : NP \multimap N$ instead corresponds to the complex noun *owner of X*, which can appear in expressions like *the owner of X*, *every owner of X*, *an owner of X*...

[157] Note that this is the same as the `shift0` operation used in 4.6, where we extend an existing effect signature E with $\texttt{shift0} : ((\delta \to \mathcal{F}_E(\omega)) \to \mathcal{F}_E(\omega)) \rightarrowtail \delta$.

as in *every owner of a car*. We have two ways to dispense with this: having the quantifier embedded within the noun take scope either below or above the quantifier of the determiner.

$$[\![\textsc{every}_1]\!] = \lambda N.\, \mathtt{scope!}\,(\lambda k.\ \overline{\forall}\, x.\, \mathsf{SI}\,((N \ll\cdot\, x) \gg\!= \mathtt{assert!}) \Rightarrow k\, x)$$
$$[\![\textsc{every}_2]\!] = \lambda N.\, \mathtt{scope!}\,(\lambda k.\, \mathsf{SI}\,(\overline{\forall}\, x.\, ((N \ll\cdot\, x) \gg\!= \mathtt{assert!}) \Rightarrow k\, x))$$

We choose the entry which assigns scope that corresponds to the linear order of the determiners within the sentence. To account for the alternative readings, we will adopt the solution of quantifier raising from 6.4.1.

We can also check that the new entry for the indefinite article A′ amounts to almost the same thing as the existing one for A:[158]

$$\begin{aligned}
\mathsf{SI}\,([\![\textsc{a}']\!]\, N) &= \mathsf{SI}\,(\mathtt{scope!}\,(\lambda k.\ \overline{\exists}\, x.\, \mathsf{SI}\,((N \ll\cdot\, x) \gg\!= \mathtt{assert!})\, \overline{\wedge}\, k\, x)) \\
&= \overline{\exists}\, x.\, \mathsf{SI}\,((N \ll\cdot\, x) \gg\!= \mathtt{assert!})\, \overline{\wedge}\, \eta\, x \\
&\approx \overline{\exists}\, x.\, ((N \ll\cdot\, x) \gg\!= \mathtt{assert!})\, \overline{\wedge}\, \eta\, x \\
&= \mathtt{introduce} \star (\lambda x.\, ((N \ll\cdot\, x) \gg\!= \mathtt{assert!})\, \overline{\wedge}\, \eta\, x) \\
&= \mathtt{introduce} \star (\lambda x.\, (N \gg\!= (\lambda n.\, \mathtt{assert!}\,(n\, x)))\, \overline{\wedge}\, \eta\, x) \\
&= \mathtt{introduce} \star (\lambda x.\, (N \gg\!= (\lambda n.\, \mathtt{assert!}\,(n\, x))) \gg\!= (\lambda_.\, \eta\, x)) \\
&= \mathtt{introduce} \star (\lambda x.\, N \gg\!= (\lambda n.\, \mathtt{assert}\,(n\, x)\,(\lambda_.\, \eta\, x))) \\
&= [\![\textsc{a}]\!]\, N
\end{aligned}$$

Quantified noun phrases using the `scope` effect take scope over all the material up to the nearest enclosing SI handler. We should therefore modify some of the existing constructions in our grammar to use SI to designate scope islands. Notably, we will make it so that every tensed clause acts as a scope island by including SI in the lexical entries of tensed verbs.

$$\begin{aligned}
[\![\textsc{loves}]\!] &:= \lambda OS.\, \mathsf{SI}\,([\![\textsc{loves}]\!]\, O\, S) \\
[\![\textsc{owns}]\!] &:= \lambda OS.\, \mathsf{SI}\,([\![\textsc{owns}]\!]\, O\, S) \\
[\![\textsc{said}_{\textsc{is}}]\!] &:= \lambda CS.\, \mathsf{SI}\,([\![\textsc{said}_{\textsc{is}}]\!]\, C\, S) \\
[\![\textsc{said}_{\textsc{ds}}]\!] &:= \lambda CS.\, \mathsf{SI}\,([\![\textsc{said}_{\textsc{ds}}]\!]\, C\, S)
\end{aligned}$$

8.5.1 Quantifier Raising — Inverse Scope and Crossover

The strategy that we have used in Subsection 6.4.1 to deal with quantifier scope ambiguity was to change the order of evaluation of the quantified noun phrases. This was the case both in the first approach, which considered adding different lexical items which evaluate their arguments in different orders, and in the second, final approach, which used a general operator QR to displace the evaluation of an NP.

However, freely changing the order of evaluation is problematic. Consider the following sentence (from [120]):

(31) * His$_1$ mother likes every man$_1$.

Our strategy of generating an inverse scope reading by evaluating the object first and then the subject would lead us to bind the pronoun *his* to the variable introduced by *every man*. However, this is considered unacceptable and we would therefore like to avoid doing that in our model.

This kind of problem represents a challenge to the fundamental assumptions behind our methodology. Anaphora forces us to evaluate the constituents in linear order, subject first and object last, so that

[158] The difference being that A′ projects to the box which contains the nearest scope island marker SI, whereas A project to the nearest box.

we do not license cataphora from object to subject. Inverse scope forces us to evaluate the constituents in inverse order, object first and subject last, so that the object can take scope over the sentence before the subject does.

In our current setting, this conflict seems impossible to resolve. However, we can find a solution by decomposing the action of a quantified noun phrases into two steps:

1. The quantified noun phrase takes scope over its matrix clause *without* making the variable anaphorically accessible. The quantified noun phrase is "replaced" by a *trace* computation.

2. The trace is evaluated, making the variable anaphorically accessible.

This strategy can be summarized by saying that even though a quantifier can move out and in front of a sentence, it is its original position, represented by a trace, which controls its anaphoric behavior.

Our denotations $[\![\text{A}']\!]$ and $[\![\text{EVERY}]\!]$ use the $\overline{\exists}$ and $\overline{\forall}$ dynamic quantifiers, where $\overline{\forall}$ is expressed in terms of $\overline{\neg}$ and $\overline{\exists}$. The $\overline{\exists}$ quantifier itself is defined by the `introduce` operation.

$$\begin{aligned}\overline{\exists}\, x.\, A &= \texttt{introduce} \star (\lambda x.\, A)\\ \overline{\forall}\, x.\, A &= \overline{\neg}(\overline{\exists}\, x.\, \overline{\neg} A)\end{aligned}$$

The `introduce` operation is interpreted by the box handler as installing an existential handler *and* introducing the variable into the context.

$$\mathsf{box} = \ldots (\!|\, \ldots,\ \texttt{introduce}\colon (\lambda\_k.\, \eta\, (\lambda e.\ \exists_{\gg}\, x.\, k\, x \lll\cdot\, (x :: e))),\ \ldots |\!)\, \ldots$$

We can decompose `introduce` into two operations, `fresh` and `push`, one to wrap an existential quantifier with a fresh variable over the discourse (`fresh`) and another to add an individual into the context (`push`).

$$\begin{aligned}\texttt{fresh} &: 1 \rightarrowtail \iota\\ \texttt{push} &: \iota \rightarrowtail 1\end{aligned}$$

These two operations will replace `introduce`, which is expressible as their composition, and their interpretations replace the interpretation of `introduce` in the box handler:

$$\begin{aligned}\mathsf{box} = \ldots (\!|\ & \ldots,\\ & \texttt{fresh}\colon (\lambda\_k.\, \eta\, (\lambda e.\ \exists_{\gg}\, x.\, k\, x \lll\cdot\, e)),\\ & \texttt{push}\colon (\lambda x k.\, \eta\, (\lambda e.\, k \star \lll\cdot (x :: e))),\\ & \ldots |\!)\, \ldots\end{aligned}$$

$$\begin{aligned}&\texttt{introduce} : 1 \to (\iota \to \mathcal{F}_{E \uplus \{\texttt{fresh},\texttt{push}\}}(\alpha)) \to \mathcal{F}_{E \uplus \{\texttt{fresh},\texttt{push}\}}(\alpha)\\ &\texttt{introduce} = \lambda\_k.\ \texttt{fresh} \star (\lambda x.\\ &\qquad \texttt{push}\, x\, (\lambda\_.\\ &\qquad k\, x))\end{aligned}$$

We can now adjust the dynamic quantifiers so that they only introduce a quantifier over the discourse without modifying the context.

$$\begin{aligned}\overline{\exists}\, x.\, A &= \texttt{fresh} \star (\lambda x.\, A)\\ \overline{\forall}\, x.\, A &= \overline{\neg}(\overline{\exists}\, x.\, \overline{\neg} A)\end{aligned}$$

We can now give a new meaning to the determiners *every* and *a*.

$$
\begin{aligned}
&\textsc{every}, \textsc{a}' : N \multimap QNP \\
&[\![QNP]\!] = \mathcal{F}_{\{\texttt{scope}\}}(\mathcal{F}_{E'}(\iota))
\end{aligned}
$$

$$
\begin{aligned}
[\![\textsc{every}]\!] &= \lambda N.\ \texttt{scope}\,(\lambda k.\ \overline{\forall}\,x.\ \mathsf{SI}\,((N \ll\cdot\, x) \gg\!= \texttt{assert!}) \rightrightarrows k\,x)\,(\lambda x. \\
&\qquad \eta\,(\mathsf{trace}\,x)) \\
[\![\textsc{a}']\!] &= \lambda N.\ \texttt{scope}\,(\lambda k.\ \overline{\exists}\,x.\ \mathsf{SI}\,((N \ll\cdot\, x) \gg\!= \texttt{assert!}) \overline{\wedge}\, k\,x)\,(\lambda x. \\
&\qquad \eta\,(\mathsf{trace}\,x))
\end{aligned}
$$

$$
\begin{aligned}
&\mathsf{trace} : \iota \to \mathcal{F}_{E'}(\iota) \\
&\mathsf{trace} = \lambda x.\ \texttt{push}\,x\,(\lambda\_.\ \eta\,x)
\end{aligned}
$$

We split the computations into two layers: an outer layer using the `scope` effect, where the order will determine the relative scope of quantifiers (using the `fresh` effect in the $\overline{\exists}$ and $\overline{\forall}$ quantifiers), and an inner layer using all the other effects in E', where the order will determine the behavior of other phenomena such as anaphora (e.g., the `push` of the QNP's trace and the `get` of an anaphoric pronoun). This way, we can have a different order of evaluation in the two layers and get inverse scope readings without risking the violation of crossover constraints.

We could have introduced two layers of computation into all of our NP meanings but this would needlessly create complexity. Instead, only the meanings of quantified noun phrases (abstract type QNP) will have two layers. We can then plug these meanings into our grammars using one of these two lexical items:

$$
\begin{aligned}
\textsc{in-situ} &: QNP \multimap NP \\
[\![\textsc{in-situ}]\!] &= \lambda Q.\ Q \gg\!= (\lambda X.\ X) \\
\mathrm{QR} &: QNP \multimap (NP \multimap S) \multimap S \\
[\![\mathrm{QR}]\!] &= \lambda QK.\ Q \gg\!= K
\end{aligned}
$$

IN-SITU collapses the two layers of effects so that quantified noun phrases become single-layer computations, just like other NPs. With this lexical item, we recover the original readings that were possible with the single-layer denotations of every EVERY and A′ that we had at the beginning of this section.

Furthermore, we can use QR to generate inverse scope (and all the other permutations of quantifiers from 6.4.1). QR evaluates the outer layer of effects before the matrix clause, allowing it to take scope. The computation in the inner layer, which is responsible for adding the quantified variable into the context and making it anaphorically accessible, takes the place of the QNP's trace.

Using QR, we can explain the inverse scope reading of Example 32.

(32) A woman loves every man.

$$
\begin{aligned}
&(\downarrow \circ \mathsf{top}^{159})\ [\![\mathrm{QR}\,(\textsc{every man})\,(\lambda O.\ \textsc{loves}\,O\,(\textsc{a}'\ \textsc{woman}))]\!] \\
&\twoheadrightarrow \forall y.\ \mathbf{man}\,y \to (\exists x.\ \mathbf{woman}\,x \wedge \mathbf{love}\,x\,y)
\end{aligned}
$$

However, we cannot use the same mechanism to derive the incorrect reading of Example 31.

[159] After adding deixis, the type of top became $\iota \to \mathcal{F}_E(1) \to \mathcal{F}_\emptyset(o)$. However, we will often look at sentences which are not indexical. In such cases, we will overload top to also mean "the top handler without the withSpeaker handler", type $\mathcal{F}_{E\setminus\{\texttt{speaker}\}}(1) \to \mathcal{F}_\emptyset(o)$. This means we will not be obliged to provide a referent for the speaker.

$$(\downarrow_{\circ} \circ \mathsf{top} \circ (\!| \mathtt{get}\colon (\lambda\_k.\, k\, e) |\!))\, [\![\mathrm{QR}\, (\textsc{every man})\, (\lambda O.\, \textsc{loves}\, O\, (\textsc{mother he}))]\!]$$
$$\twoheadrightarrow \exists x.\, \mathbf{mother}\, x\, (\mathsf{sel}_{\mathsf{he}}\, e) \wedge (\forall y.\, \mathbf{man}\, y \rightarrow \mathbf{love}\, x\, y)^{160}$$

We now have a grammar that is both capable of licensing inverse scope and respects (certain) crossover constraints.[161] The rest of the grammar needs to be modified only insofar as handlers for `introduce` need to be replaced with handlers for `fresh` and `push`. The only other handler for `introduce` besides box is the asImplicature handler.

$$\mathsf{asImplicature} : \mathcal{F}_{E \uplus \{\mathtt{assert}, \mathtt{fresh}, \mathtt{push}\}}(\alpha) \rightarrow \mathcal{F}_{E \uplus \{\mathtt{implicate}, \mathtt{fresh}^{\mathtt{i}}, \mathtt{push}^{\mathtt{i}}\}}(\alpha)$$
$$\mathsf{asImplicature} = (\!| \mathtt{assert}\colon \mathtt{implicate},\ \mathtt{fresh}\colon \mathtt{fresh}^{\mathtt{i}},\ \mathtt{push}\colon \mathtt{push}^{\mathtt{i}} |\!)$$

Since we no longer use `introduce` as a primitive operation, asImplicature will act on `fresh` and `push`, turning them into new operations, $\mathtt{fresh}^{\mathtt{i}} : 1 \rightarrowtail \iota$ and $\mathtt{push}^{\mathtt{i}} : \iota \rightarrowtail 1$. The converse handler, withImplicatures, will therefore also need to be changed to treat $\mathtt{fresh}^{\mathtt{i}}$ and $\mathtt{push}^{\mathtt{i}}$.

$$\mathsf{withImplicatures} : \mathcal{F}_{E \uplus E_{\mathrm{DRT}} \uplus \{\mathtt{implicate}, \mathtt{fresh}^{\mathtt{i}}, \mathtt{push}^{\mathtt{i}}\}}(\alpha) \rightarrow \mathcal{F}_{E \uplus E_{\mathrm{DRT}} \uplus \{\mathtt{assert}, \mathtt{fresh}, \mathtt{push}\}}(\alpha)$$
$$\mathsf{withImplicatures} = (\!| \mathtt{implicate}\colon \mathtt{assert},\ \mathtt{fresh}^{\mathtt{i}}\colon \mathtt{fresh},\ \mathtt{push}^{\mathtt{i}}\colon \mathtt{push} |\!)$$

Connections to Other Solutions

We will draw parallels to two other approaches.

In the interpretations of EVERY and A′, we introduce an extra layer of computation. Then, instead of generalizing all other noun phrases to employ this extra layer, we added IN-SITU and QR as lexical items into our grammar, allowing us to use quantified noun phrases, which use this extra layer of computation, in contexts where an ordinary noun phrase was expected. The interpretation of QR is the $\gg\!=$ operator. This is reminiscent of the way of composing different monads proposed by Simon Charlow during Barker's and Bumford's ESSLLI 2015 course [13]. The idea is that we should not try to build some large monad that encompasses all aspects of meaning but rather build meanings in smaller, different monads, each meaning using only as much structure as it needs. The gluing together of the meanings is then performed by the parser which can insert $\gg\!=$ for the relevant monads and other plumbing as necessary. This is also the kind of approach adopted by Giorgolo and Asudeh [50].

We also notice a similarity between our approach to inverse scope and that of Shan in [117]. Shan adopts multistage programming to suspend the evaluation of the context of the object until after the object itself has been evaluated, effectively displacing the evaluation of the object before the evaluation of the rest of the sentence. Our strategy is based on similar techniques: QR has the same effect of displacing evaluation and the denotations of EVERY and A′ suspend the evaluation of their anaphoric effects. A distinguishing sign of meta-programming or multistage programming are types like $\mathcal{F}_{E_1}(\mathcal{F}_{E_2}(\alpha))$, i.e. programs producing programs, which is exactly the kind of type we use for $[\![QNP]\!]$.

Limitations

Our treatment of inverse scope eliminates both primary crossover (the raised quantifier is the object, e.g. Example 33) and secondary crossover (the raised quantifier is embedded in the object, e.g. Example 34). Examples from [120]:

[160] Note that the quantifier ranging over mothers has wider scope than the one ranging over men, even though we raised the QNP *every man* over the whole sentence. This is because the referring expression *his mother* is presuppositional and is accommodated globally, above the meaning of this particular sentence. The context in which the operator $\mathsf{sel}_{\mathsf{he}}$ looks for an antecedent to the pronoun *his* is the context $(\mathbf{man}\, y) :: e$, where e is the context in which the sentence is being evaluated. Since adding a proposition to a context is not supposed to change the set of possible antecedents, we write simply $\mathsf{sel}_{\mathsf{he}}\, e$.

[161] We will see shortly that there are situations for which our treatment is not sufficient.

(33) * He$_1$ likes every man$_1$.

(34) * He$_1$ likes every man$_1$'s mother.

However, there is a configuration that we can construct in our grammar in which a kind of crossover is permitted.

(35) * It$_1$ loves every owner of a dog$_1$.

If we aim for the reading where there is a dog and it loves every one of its owners (i.e. we raise the QNP *a dog*), then everything works correctly: we predict that the binding is impossible. However, if we raise the whole object QNP *every owner of a dog* to get the reading in which there are possibly multiple different dogs, then we will license the cataphoric binding from *a dog* to *it*. Even though the raising of the object will not make the dog-owner available as an antecedent to the subject, we will still have evaluated the object QNP's restrictor *owner of a dog*, which has the effect of introducing a new discourse referent, a dog. If we were to outlaw cases like these, we would need a more robust way of delaying the introduction of discourse referents.

8.6 Considering Restrictive Relative Clauses

There is one construction that we have not treated in our grammar and those are restrictive relative clauses. We will sketch out possible ways of dealing with those along with some of the challenges.

$$\textsc{who}_{\textsc{r}} : (NP \multimap S) \multimap N \multimap N$$

Contrary to supplementary (appositive) relative clauses, which attach to noun phrases and add extra information about their referent, restrictive relative clauses attach to nouns and narrow down the set of individuals under consideration, just like subsective adjectives (e.g. *woman*, and *woman who loves books* as *book-loving woman*).

Restrictive relative clauses let us use the extension of any verb phrase as the extension of a noun: for a given verb phrase P, we can form the noun *one who Ps*. We therefore have an injection from verb phrases to nouns, telling us verb phrases are nouns too. The question now is whether the type of interpretations we use for nouns is large enough to fit the denotations of verb phrases.

$$\begin{aligned}[\![NP \multimap S]\!] = [\![NP]\!] \multimap [\![S]\!] &= \mathcal{F}_E(\iota) \to \mathcal{F}_E(1) \\ [\![N]\!] &= \mathcal{F}_E(\iota \to o)\end{aligned}$$

There are two important issues here. Firstly, the denotations of our sentences are computations that express their truth conditions using side effects. On the other hand, the denotations of our nouns are taken to be computations that produce predicates, pure functions from individuals to propositions. Secondly, the effects in the denotations $[\![NP \multimap S]\!]$ can depend on the individual in $[\![NP]\!] = \mathcal{F}_E(\iota)$ whereas the effects in the denotations $\mathcal{F}_E(\iota \to o)$ must be independent of the argument individual.

The latter will lead us to trouble if we try fixing the former too naively:

$$\begin{aligned}[\![\textsc{who}_{\textsc{r}}]\!] = \lambda KN.\, \mathcal{C}\, (\lambda x.\, N \gg\!= (\lambda n.\, & \\ & K\,(\eta\, x) \gg\!= (\lambda\_. \\ & \eta\,(n\, x))))\end{aligned}$$

Here we add the truth conditions of the relative clause to the noun by including the side effects of the clause in the effects of the noun. However, this only works when the effects of the relative clause are independent of the entity x under consideration,[162] which is rarely the case. For example, assuming that $[\![\lambda x.\, \textsc{loves}\, \textsc{Mary}\, x]\!] = \lambda x.\, \texttt{assert!}\, (\mathbf{love}\, x\, \mathbf{m})$, the meaning of *man who loves Mary* becomes:

[162] Because of our use of $\mathcal{C}$.

$$\begin{aligned}&[\![\textsc{who}_{\textsc{r}}\,(\lambda x.\,\textsc{loves}\,\textsc{Mary}\,x)\,\textsc{man}]\!]\\&\twoheadrightarrow \mathcal{C}\,(\lambda x.\,\texttt{assert}\,(\mathbf{love}\,x\,\mathbf{m})\,(\lambda\_.\,\eta\,(\mathbf{man}\,x)))\end{aligned}$$

The evaluation of $\mathcal{C}$ is blocked because the operation $\texttt{assert}\,(\mathbf{love}\,x\,\mathbf{m})$ depends on x. The truth conditions of $x\ Ps$, where P is a verb phrase, will always depend on x and so this approach will not work.

However, our system already has a well-defined way for turning dynamic propositions into simple propositions, and that is the box handler.

$$\begin{aligned}[\![\textsc{who}_{\textsc{r}}]\!] &= \lambda KN.\,\mathcal{C}\,(\lambda x.\,N \gg\!= (\lambda n.\\&\qquad\qquad \mathsf{box}\,(K\,(\eta\,x)) \gg\!= (\lambda p.\\&\qquad\qquad \eta\,(n\,x \wedge p))))\\[\![\textsc{who}_{\textsc{r}}]\!] &= \lambda KN.\,\mathcal{C}\,(\lambda x.\,(N \ll\!\cdot\, x) \ll\!\wedge\!\gg \mathsf{box}\,(K\,(\eta\,x)))\end{aligned}$$

By wrapping the meaning of the relative clause in a box, its truth conditions (`assert` and `introduce`) are contained within the noun's predicate and they do not block the $\mathcal{C}$ operator. However, adding the box in the entry for $\textsc{who}_{\textsc{r}}$ does have its repercussions.[163] It will lead to the blocking of anaphoric binding from the noun, as in the Example 7.

(7) Every farmer who owns a donkey$_1$ beats it$_1$.

Our entry for $[\![\textsc{who}_{\textsc{r}}]\!]$ gives the following meaning to the noun *farmer who owns a donkey*:

$$\begin{aligned}&[\![\textsc{who}_{\textsc{r}}\,(\lambda x.\,\textsc{owns}\,(\textsc{a donkey})\,x)\,\textsc{farmer}]\!]\\&\twoheadrightarrow \eta\,(\lambda x.\,\mathbf{farmer}\,x \wedge (\exists y.\,\mathbf{donkey}\,y \wedge \mathbf{own}\,x\,y))\end{aligned}$$

The result is a pure computation that is not going to introduce any referents into the discourse (because all of the `introduce` operations were captured by box) and therefore it is not going to be able to license the anaphoric binding to the pronoun *it* in Example 7.

We have seen that by not using box evaluation gets stuck because the `assert` operations that carry the truth conditions of the relative clause are not independent of the argument to which the noun's extension is applied. On the other hand, by using box, the `introduce` operations that introduce new discourse referents are blocked and cannot license the kind of anaphora that we see in Example 7. We can find the middle ground between the two by only handling `assert` to get the truth conditions and letting `introduce` project outside of the noun.

$$\begin{aligned}\mathsf{withAssertions} &: \mathcal{F}_{E\uplus\{\texttt{assert}\}}(1) \to \mathcal{F}_E(o)\\\mathsf{withAssertions} &= (\!|\ \texttt{assert}\colon (\lambda pk.\,p \wedge\!\gg k\,\star),\\&\qquad\qquad \eta\colon (\lambda\_.\,\top)\ |\!)\\[\![\textsc{who}_{\textsc{r}}]\!] &= \lambda KN.\,\mathcal{C}\,(\lambda x.\,(N \ll\!\cdot\, x) \ll\!\wedge\!\gg \mathsf{withAssertions}\,(K\,(\eta\,x)))\end{aligned}$$

Now, if we reexamine the meaning of the noun *farmer who owns a donkey* under the new interpretation of $\textsc{who}_{\textsc{r}}$, we find that the new discourse referent projects outside of the noun and will therefore be able to bind the pronoun in Example 7:

$$\begin{aligned}&[\![\textsc{who}_{\textsc{r}}\,(\lambda x.\,\textsc{owns}\,(\textsc{a donkey})\,x)\,\textsc{farmer}]\!]\\&\twoheadrightarrow \texttt{introduce}\,\star\,(\lambda y.\\&\qquad \eta\,(\lambda x.\,\mathbf{man}\,x \wedge \mathbf{donkey}\,y \wedge \mathbf{own}\,x\,y))\end{aligned}$$

[163] We have seen that the box handler corresponds to the boundary of a DRS. We note that DRT does not instruct us to wrap the relative clause in a DRS and we can therefore expect our theory to diverge w.r.t. accessibility.

In the rest of this section, we will discuss two topics related to restrictive relative clauses. First, we will outline an analysis which chooses a richer type of interpretations for nouns so that the lexical entry for the relative pronoun will be less ad-hoc. Then, we will discuss how relative clauses and other kinds of complex nouns interact with presuppositions and definite descriptions.

8.6.1 Different Interpretation for Nouns

At the start of this section, we have shown the challenge that is posed by folding in the meanings of sentences into the meanings of nouns when the two are interpreted completely differently:

$$\begin{aligned}[\![NP \multimap S]\!] &= \mathcal{F}_E(\iota) \to \mathcal{F}_E(1)\\ [\![N]\!] &= \mathcal{F}_E(\iota \to o)\end{aligned}$$

If we change the type of interpretations of nouns so that truth conditions are expressed as side effects and the effects can depend on the individual argument, then there is no conflict any more and the lexical entry for $\textsc{who}_{\textsc{r}}$ becomes trivial:

$$\begin{aligned}[\![NP \multimap S]\!] &= \mathcal{F}_E(\iota) \to \mathcal{F}_E(1)\\ [\![N]\!] &= \iota \to \mathcal{F}_E(1)\end{aligned}$$

$$[\![\textsc{who}_{\textsc{r}}]\!] = \lambda knx.\, n\, x \,\overline{\wedge}\, k\, (\eta\, x)$$

The schema underlying the use of computation types in such a grammar could be expressed the following way. Let the type of extensions $[\![A]\!]_{\mathrm{v}}$ of an atomic abstract type A be of the form $a_1 \to \cdots \to a_n \to b$, where a_i are atomic object types. Then we will use the type $a_1 \to \cdots \to a_n \to \mathcal{F}_E(b)$ as the interpretation $[\![A]\!]_{\mathrm{c}}$ of the atomic abstract type A. This means that the interpretations are call-by-value functions, taking n values as arguments and then producing a computation.

$$\begin{aligned}[\![S]\!]_{\mathrm{c}} &= \mathcal{F}_E(o)\\ [\![NP]\!]_{\mathrm{c}} &= \mathcal{F}_E(\iota)\\ [\![N]\!]_{\mathrm{c}} &= \iota \to \mathcal{F}_E(o)\end{aligned}$$

Then, if we adopt the use of side effects to encode truth conditions, as in our dynamic grammar, we get the following:

$$\begin{aligned}[\![S]\!] &= \mathcal{F}_E(1)\\ [\![NP]\!] &= \mathcal{F}_E(\iota)\\ [\![N]\!] &= \iota \to \mathcal{F}_E(1)\end{aligned}$$

If we were to adopt this schema universally, we could also adapt the lifting operators and prove their conservativity by slightly modifying the proof of Observation 6.1.3.

$$\begin{aligned}\mathsf{lift}^{\mathsf{L}}_{\alpha} &: [\![\alpha]\!]_{\mathrm{v}} \to [\![\alpha]\!]_{\mathrm{c}}\\ \mathsf{lift}^{\mathsf{L}}_{S}(p) &= \eta\, p\\ \mathsf{lift}^{\mathsf{L}}_{NP}(x) &= \eta\, x\\ \mathsf{lift}^{\mathsf{L}}_{N}(n) &= \lambda x.\, \eta\, (n\, x)\\ \mathsf{lift}^{\mathsf{L}}_{S \multimap \beta}(f) &= \lambda P.\, P \gg\!= (\lambda p.\, \mathsf{lift}^{\mathsf{L}}_{\beta}(f\, p))\\ \mathsf{lift}^{\mathsf{L}}_{NP \multimap \beta}(f) &= \lambda X.\, X \gg\!= (\lambda x.\, \mathsf{lift}^{\mathsf{L}}_{\beta}(f\, x))\\ \mathsf{lift}^{\mathsf{L}}_{N \multimap \beta}(f) &= \lambda N.\, (\mathcal{C}\, N) \gg\!= (\lambda n.\, \mathsf{lift}^{\mathsf{L}}_{\beta}(f\, n))\end{aligned}$$

Finally, we could integrate this type of noun interpretations and the simpler lexical entry for the relative pronoun by adjusting the grammar that we have been building during this chapter.

$$
\begin{aligned}
[\![\textsc{man}]\!] &= \lambda x.\, \texttt{assert!}\,(\mathbf{man}\, x) \\
[\![\textsc{woman}]\!] &= \lambda x.\, \texttt{assert!}\,(\mathbf{woman}\, x) \\
&\vdots \\
[\![\textsc{a}]\!] &= \lambda N.\, \texttt{introduce} \star N \\
[\![\textsc{every}]\!] &= \lambda N.\, \texttt{scope}\ (\lambda k.\ \overline{\forall}\, x.\, \mathsf{SI}\,(N\, x) \Rightarrow k\, x) \\
&\qquad (\lambda x.\, \eta\,(\mathsf{trace}\, x)) \\
[\![\textsc{a}']\!] &= \lambda N.\, \texttt{scope}\ (\lambda k.\ \overline{\exists}\, x.\, \mathsf{SI}\,(N\, x) \overline{\wedge}\, k\, x) \\
&\qquad (\lambda x.\, \eta\,(\mathsf{trace}\, x)) \\
[\![\textsc{owner-of}]\!] &= \lambda Y x.\, (\mathbf{own}\, x \cdot\!\!\gg Y) \gg\!= \texttt{assert!}
\end{aligned}
$$

$$
\begin{aligned}
[\![\textsc{man}]\!] &= \eta\, \mathbf{man} \\
[\![\textsc{woman}]\!] &= \eta\, \mathbf{woman} \\
&\vdots \\
[\![\textsc{a}]\!] &= \lambda N.\, \texttt{introduce} \star (\lambda x. \\
&\qquad N \gg\!= (\lambda n. \\
&\qquad \texttt{assert}\,(n\, x)\,(\lambda\_. \\
&\qquad \eta\, x))) \\
[\![\textsc{every}]\!] &= \lambda N.\, \texttt{scope}\ (\lambda k.\ \overline{\forall}\, x.\, \mathsf{SI}\,((N \ll\!\cdot\, x) \gg\!= \texttt{assert!}) \Rightarrow k\, x) \\
&\qquad (\lambda x.\, \eta\,(\mathsf{trace}\, x)) \\
[\![\textsc{a}']\!] &= \lambda N.\, \texttt{scope}\ (\lambda k.\ \overline{\exists}\, x.\, \mathsf{SI}\,((N \ll\!\cdot\, x) \gg\!= \texttt{assert!}) \overline{\wedge}\, k\, x) \\
&\qquad (\lambda x.\, \eta\,(\mathsf{trace}\, x)) \\
[\![\textsc{owner-of}]\!] &= \lambda Y.\, Y \gg\!= (\lambda y.\, \eta\,(\lambda x.\, \mathbf{own}\, x\, y))
\end{aligned}
$$

These are all the changes that need to be made to the grammar, *except* for the entries having to do with presuppositions (POSS and THE), which we will deal with next.

8.6.2 Relative Clauses and Presuppositions

While changing the semantics of nouns and introducing restrictive relative clauses, we have so far ignored their interaction with presupposition triggers that depend on them. We will now give the semantics to these referring expressions, observe some of their deficiencies and propose solutions.[164]

$$
\begin{aligned}
[\![\textsc{poss}]\!] &= \lambda X N.\ X \gg\!= (\lambda x. \\
&\qquad \mathcal{C}\,(\mathsf{box} \circ N) \gg\!= (\lambda n. \\
&\qquad \texttt{presuppose!}\,(\lambda y.\, n\, y \wedge \mathbf{own}\, x\, y))) \\
[\![\textsc{the}]\!] &= \lambda N.\ \mathcal{C}\,(\mathsf{box} \circ N) \gg\!= (\lambda n. \\
&\qquad \texttt{presuppose!}\, n)
\end{aligned}
$$

The `presuppose` operation accepts *static properties*, functions of type $\iota \to o$, as arguments. Our noun is a *dynamic property*[165], a function of type $\iota \to \mathcal{F}_E(1)$. To go from dynamic to static, we can use the box handler to cast the `introduce` and `assert` operations down to propositions. Note that the same strategy is employed by Lebedeva in [80] (Equations (5.14) and (5.22)):

$$
\begin{aligned}
\widetilde{\mathsf{sel}} &= \lambda \mathbf{P} e.\, \mathsf{sel}\,(\lambda x.\, \mathbf{P}\,(\lambda e.\, x)\, e\,(\lambda e.\, \top))\, e \\
\widetilde{[\![the]\!]} &= \lambda \mathbf{NP}.\, \mathbf{P}\,(\widetilde{\mathsf{sel}}\, \mathbf{N})
\end{aligned}
$$

The entry for the definite article uses the operator sel, on which our find operator is based. The sel operator is made compatible with dynamic properties in $\widetilde{\mathsf{sel}}$. This is done by applying a context e and the trivial continuation $\lambda e.\, \top$ to the dynamic proposition $\mathbf{P}\,(\lambda e.\, x)$.[166]

[164] Below, we use the interpretation of nouns introduced in 8.6.1. However, we could draw similar conclusions for the interpretation of nouns used in the rest of this chapter.

[165] Terminology due to [80].

[166] This is equivalent to the use of the box handler in our approach.

However, this solution can be lacking when the dynamic property which is the denotation of the noun introduces new discourse referents itself (e.g. *the man who owns a dog*).

$$\begin{aligned}&[\![\textsc{the}\,(\textsc{who}_{\textsc{r}}\,(\lambda x.\,\textsc{owns}\,(\textsc{a dog})\,x)\,\textsc{man})]\!]\\ &\twoheadrightarrow \texttt{presuppose!}\,(\lambda x.\,\exists y.\,\mathbf{man}\,x \wedge \mathbf{dog}\,y \wedge \mathbf{own}\,x\,y)\end{aligned}$$

If we were to accommodate such a presupposition, we would introduce at the global level a new discourse referent satisfying the above condition. However, the binding potential of the NP *a dog* would have already been wasted by the use of box (or in the case of TTDL, the use of $\lambda e.\,\top$). This means that we would not be able to account for the binding in Example 36.

(36) The man who owns a dog_1 loves it_1.

Before we address this issue, we turn to another similar problem. We will consider a very similar situation, but instead of introducing a discourse referent, the relative clause will trigger a presupposition where the description of the presupposed individual will contain a variable bound by the relative clause. For example, in *the man*$_1$ *who loves his*$_1$ *dog*, the genitive construction *his*$_1$ *dog* is a presupposition trigger but the presupposition cannot project outside of the relative clause because the variable 1 has scope only over the relative clause. If we were to calculate the meaning of *the man*$_1$ *who loves his*$_1$ *dog* in the TTDL of [80], we would get the following:

$$\begin{aligned}&[\![\textit{the man}_1\ \textit{who loves his}_1\ \textit{dog}]\!]\\ &= \lambda \mathbf{P} e \phi.\,\mathbf{P}\,(\mathsf{sel}\,(\lambda x.\,\mathsf{raise}\,(\texttt{AbsentIndividualExc}\,(\lambda y.\,\mathbf{dog}\,y \wedge \mathbf{poss}\,x\,y)))\,e)\,e\,\phi\end{aligned}$$

At this point, we would get stuck since the exception cannot propagate outside of the λ-abstraction λx, because x is bound within the exception's message, $\lambda y.\,\mathbf{dog}\,y \wedge \mathbf{poss}\,x\,y$.[167] However, in TTDL we can fix this by using the same intermediate accommodation handler as is used in the dynamic existential quantifier to solve the binding problem.[168] In our approach, the handler responsible for intermediate accommodation, maybeAccommodate, is already part of the box handler and so evaluation does not get stuck and the presupposition is accommodated. Nevertheless, to get the intended reading of the noun phrase above, we will have to make the argument to the property anaphorically accessible within the noun:[169]

$$\begin{aligned}[\![\textsc{the}]\!] = \lambda N.\,\mathcal{C}\,(\lambda x.\,\mathsf{box}\,(\texttt{push}\,x\,(\lambda\_.\,N\,x))) \gg\!= (\lambda n.\\ \texttt{presuppose!}\,n)\end{aligned}$$

We can now use this entry to compute the meaning of the noun phrase *the man*$_1$ *who loves his*$_1$ *dog*. When presenting the result, we use the empty and search handlers to filter out the irrelevant branches (looking for the antecedent of *his dog* in the context and trying to project outside of the relative clause).

$$\begin{aligned}&\mathsf{search}\,(\mathsf{empty}\,[\![\textsc{the}\,(\textsc{who}_{\textsc{r}}\,(\lambda x.\,\textsc{loves}\,(\textsc{poss he dog})\,x)\,\textsc{man})]\!])\\ &\twoheadrightarrow \texttt{presuppose!}\,(\lambda x.\,\exists y.\,\mathbf{dog}\,y \wedge \mathbf{own}\,x\,y \wedge \mathbf{man}\,x \wedge \mathbf{love}\,x\,y)\end{aligned}$$

We have solved the problem of the definite description *the* N *who* Ps where P triggers a presupposition dependent on N. However, there is still the same issue that we encountered at the beginning of this subsection. When we accommodate the presupposition triggered by the definite description *the man who loves his dog*, we introduce the man as a discourse referent. However, the scope of the quantifier ranging over dogs will be limited to the proposition $\exists y.\,\mathbf{dog}\,y \wedge \mathbf{own}\,x\,y \wedge \mathbf{man}\,x \wedge \mathbf{love}\,x\,y$. This means that we will not be able to explain the anaphoric binding in Example 37.

[167] This kind of stuck term is very similar to the $\mathcal{C}\,(\lambda x.\,\texttt{presuppose}\,(\lambda y.\,\mathbf{dog}\,y \wedge \mathbf{poss}\,x\,y)\,(\lambda y.\,M_{\mathrm{c}}(y)))$ in $(\!|\,\lambda\,|\!)$.

[168] The iacc handler that we mentioned in 7.3.5, defined in [80], Definition 6.29.

[169] In the presentation of TTDL in Lebedeva's thesis [80], adding the condition $\mathbf{man}\,x$ to the context suffices to make x accessible. In our setting, we make a distinction between adding a fact to the common ground using `assert` versus introducing a new discourse referent for discussion using `push` or `introduce` (this lets us give a (partial) account of crossover constraints).

(37) The man$_1$ who loves his$_1$ dog$_2$ treats it$_2$ well.

Sketching a Solution — Dynamic Presuppositions

If we want the dynamic effects of definite descriptions to have discourse-wide scope, then we should move their whole evaluation, not just their results, to the global context. We can change the type of `presuppose` so that we are not signalling the presupposition of static properties but of dynamic ones.

$$\texttt{presuppose} : (\iota \to \mathcal{F}_{E^*}(1)) \rightarrowtail \iota$$

However, we have to be careful about the effect signature E^* used above. From the discussion in Section 8.5, we know that E^* cannot include $\texttt{presuppose} : (\iota \to \mathcal{F}_{E^*}(1)) \rightarrowtail \iota$ because then E^* is not well-defined. For similar reasons, E^* cannot contain $\texttt{scope} : ((\iota \to \mathcal{F}_{E'}(1)) \to \mathcal{F}_{E'}(1)) \rightarrowtail \iota$, because $\texttt{presuppose} \in E'$.

We can dispose of the `presuppose` effects by evaluating them and accommodating them using `introduce` and `assert` should they depend on the argument of the property (e.g. *the man who is talking to Peter* should presuppose the existence of a salient individual called Peter and *the man$_1$ who is talking to his$_1$ friend* should, when accommodated, introduce two individuals, a man and his friend).

The `scope` effect can be eliminated either by evaluating it (i.e. projecting it) or by handling it with SI. Handling it would give us narrow scope w.r.t. the presupposition, whereas projecting it would give us wider scope. We find that both are possible: narrow scope in Example 38 and wide scope in Example 39. We will therefore choose narrow scope by default, since in that case we can always derive the wider scope by quantifier raising with QR.

(38) The owner of every book ever published by Elsevier must be very rich.

(39) Mary sent an email to the representative of every country.

Coming at it from a different angle, we can start looking at the effects that we *need* to have in E^*. Our dynamic properties need to convey truth conditions (`assert`) and introduce discourse referents (`introduce`, or rather its parts: `fresh` and `push`).

Next we look at what to do with the anaphoric operation `get`. If we were not to evaluate the `get` operation at the point where the presupposition is triggered but rather project it as part of the presupposed material and evaluate in the global context (or some other accommodation site), then we would not be able to derive the reading of Example 40. If we were to evaluate *x is the sound of their voice* in the global context, we could no longer bind the pronoun *their* to its intended antecedent. On the other hand, we cannot evaluate the `get` operations without evaluating the `introduce` operations (which we do not want to do since we want to evaluate them at accommodation-time) because otherwise we would break anaphoric binding *within* the dynamic proposition being presupposed. For example, in Example 41, we would need to evaluate the `introduce` operation of the indefinite *a car* in order for the `get` operation used by the pronoun to be able to bind to the car.

(40) Everyone$_1$ hates the sound of their$_1$ voice.

(41) John met the woman who stole a car$_1$ by hacking into its$_1$ computer.

We can deal with this by preserving the `get` operations in the material being presupposed (so that internal anaphoric binding will work correctly) but making sure that they have access to all the referents available at the point of triggering the presupposition. For this kind of manipulation, we introduce the inTheContext handler, which for any context e and computation X will yield the computation inTheContext $e\,X$ that, when evaluated, will have access to the material in the context e.

$$\begin{aligned}
&\mathsf{inTheContext} : \gamma \to \mathcal{F}_{E \uplus \{\texttt{get}\}}(\alpha) \to \mathcal{F}_{E \uplus \{\texttt{get}\}}(\alpha) \\
&\mathsf{inTheContext} = \lambda e.\, (\!|\, \texttt{get}\colon (\lambda_k.\, \texttt{get} \star (\lambda e'.\, k\,(e \mathbin{+\!\!+} e'))) \,|\!)
\end{aligned}$$

Finally, we have the rest of the effects, such as `speaker` and `implicate`. We could either evaluate them at the point where the presupposition is triggered or include them into the set of effects E^* that is

projected with the presupposition. Note that if we include them in E^*, then they might get evaluated twice: once when the argument to `presuppose` is evaluated down to a static property to be used with sel_P to search for an existing referent satisfying the definite description, and then, if the referent does not exist, a second time to accommodate the presupposition and evaluate all of its dynamic effects in the global context. If we include these effects into the dynamic property that is given to `presuppose` as an argument, then these should be idempotent so that repeated evaluation does not interfere with the semantics, or we should take special care of them when doing multiple evaluations. Both `speaker` and `implicate` are idempotent, so we are not limited by this factor. However, we still want to keep E^* as simple as possible and so we will evaluate these effects at the point where the presupposition is triggered.

This means that the effect signature E^* in $\texttt{presuppose} : (\iota \to \mathcal{F}_{E^*}(1)) \rightarrowtail \iota$ will be the E_{DRT} effect signature from Chapter 7, though with `introduce` replaced with `fresh` and `push` due to the changes we have made in 8.5.1.

$$
\begin{aligned}
E^* = \{\ & \texttt{get} : 1 \rightarrowtail \gamma, \\
& \texttt{fresh} : 1 \rightarrowtail \iota, \\
& \texttt{push} : \iota \rightarrowtail 1, \\
& \texttt{assert} : o \rightarrowtail 1\ \}
\end{aligned}
$$

Our intention is to handle or evaluate the effects of the noun in the definite description with the exception of the effects in E^*. This means that we want to split the computation of the noun's meaning into two layers: one using the effects in $E \setminus E^*$ and another using the effects in E^*. We can write a handler for this.[170]

$$
\begin{aligned}
&\mathsf{separateDynamics} : \mathcal{F}_{E \uplus E^*}(\alpha) \to \mathcal{F}_E(\mathcal{F}_{E^*}(\alpha)) \\
&\mathsf{separateDynamics} = (\!|\ \texttt{get}\colon (\lambda x k.\, \texttt{get}\, x \cdot\!\!\gg (\mathcal{C}\, k)), \\
&\qquad\qquad \texttt{fresh}\colon (\lambda x k.\, \texttt{fresh}\, x \cdot\!\!\gg (\mathcal{C}\, k)), \\
&\qquad\qquad \texttt{push}\colon (\lambda x k.\, \texttt{push}\, x \cdot\!\!\gg (\mathcal{C}\, k)), \\
&\qquad\qquad \texttt{asssert}\colon (\lambda x k.\, \texttt{assert}\, x \cdot\!\!\gg (\mathcal{C}\, k)), \\
&\qquad\qquad \eta\colon (\lambda x.\, \eta\, (\eta\, x))\ |\!)
\end{aligned}
$$

We can now finally give an interpretation to the definite article THE as well as the other presupposition triggers.

$$
\begin{aligned}
&\mathsf{packageProperty} : (\iota \to \mathcal{F}_{E \uplus E^*}(1)) \to \mathcal{F}_{E \uplus \{\texttt{get}\}}(\iota \to \mathcal{F}_{E^*}(1)) \\
&\mathsf{packageProperty} = \lambda P.\ \texttt{get} \star (\lambda e. \\
&\qquad\qquad \mathcal{C}\, (\lambda x.\, \mathsf{separateDynamics}\, (\mathsf{inTheContext}\, e\, (\mathsf{maybeAccommodate}\, (\mathsf{SI}\, (P\, x))))))
\end{aligned}
$$

[170] This amounts to reordering all of the operations in a computation so that the ones in $E \setminus E^*$ go before those in E^*. This is not always possible since some of the operations from $E \setminus E^*$ might depend on the results of some of the operations in E^*. We can therefore expect the handler to make use of $\mathcal{C}$, our partial operator for messing around with evaluation and functional dependencies.

$$
\begin{aligned}
[\![\textsc{the}]\!] &= \lambda N.\ \mathsf{packageProperty}\,(\lambda x.\,\mathtt{push}\,x\,(\lambda\_.\,N\,x)) \gg\!= (\lambda N'.\\
&\qquad \mathtt{presuppose!}\,N')\\
[\![\textsc{poss}]\!] &= \lambda X N.\ X \gg\!= (\lambda x.\\
&\qquad \mathsf{packageProperty}\,(\lambda y.\,\mathtt{push}\,y\,(\lambda\_.\,\mathtt{assert}\,(\mathbf{own}\,x\,y)\,(\lambda\_.\,N\,y))) \gg\!= (\lambda N'.\\
&\qquad \mathtt{presuppose!}\,N'))\\
[\![\textsc{children-of}]\!] &= \lambda X.\ X \gg\!= (\lambda x.\\
&\qquad \mathsf{packageProperty}\,(\lambda y.\,\mathtt{assert!}\,(\mathbf{children}\,y\,x)) \gg\!= (\lambda N'.\\
&\qquad \mathtt{presuppose!}\,N'))\\
[\![\textsc{children-of}]\!] &\approx \lambda X.\ X \gg\!= (\lambda x.\\
&\qquad \mathtt{presuppose!}\,(\lambda y.\,\mathtt{assert!}\,(\mathbf{children}\,y\,x)))
\end{aligned}
$$

Finally, since we have changed the type (and therefore also the intended semantics) of the `presuppose` operation, we will also have to modify its handlers (accommodate, maybeAccommodate and useFind) accordingly.

$$
\begin{aligned}
\mathsf{accommodate} &: \mathcal{F}_{E \uplus \{\mathtt{presuppose}\}}(\alpha) \to \mathcal{F}_{E \uplus E^*}(\alpha)\\
\mathsf{accommodate} &= (\!|\,\mathtt{presuppose}\colon (\lambda P k.\,\mathtt{introduce} \star (\lambda x.\,(P\,x) \gg\!= (\lambda\_.\,k\,x)))\,|\!)\\
\mathsf{maybeAccommodate} &: \mathcal{F}_{E \uplus \{\mathtt{presuppose}\}}(\alpha) \to \mathcal{F}_{E \uplus E^* \uplus \{\mathtt{presuppose,amb}\}}(\alpha)\\
\mathsf{maybeAccommodate} &= (\!|\,\mathtt{presuppose}\colon (\lambda P k.\,\mathtt{presuppose}\,P\,k + \mathtt{introduce} \star (\lambda x.\,(P\,x) \gg\!= (\lambda\_.\,k\,x)))\,|\!)\\
\mathsf{useFind} &: \mathcal{F}_{E \uplus \{\mathtt{presuppose}\}}(\alpha) \to \mathcal{F}_{E \uplus \{\mathtt{get,presuppose}\}}(\alpha)\\
\mathsf{useFind} &= (\!|\,\mathtt{presuppose}\colon (\lambda P k.\,\mathsf{find}\,P \gg\!= k)\,|\!)
\end{aligned}
$$

$$
\begin{aligned}
\mathsf{find} &: (\iota \to \mathcal{F}_{E^*}(1)) \to \mathcal{F}_{E \uplus \{\mathtt{get,presuppose}\}}(\iota)\\
\mathsf{find} &= \lambda P.\,\mathtt{get} \star (\lambda e.\,\mathbf{case}\ \mathsf{sel_P}\,(\downarrow_{\circ} \circ \mathsf{empty} \circ P)\,e\ \mathbf{of}\ \{\mathsf{inl}\ x \to \eta\,x;\ \mathsf{inr}\ \_ \to \mathtt{presuppose!}\,P\})
\end{aligned}
$$

In accommodate and maybeAccommodate, instead of asserting $P\,x$, we evaluate it, letting it have its dynamic effects at wherever the presupposition is being accomodated. The useFind handler will work the same way as before but with a new definition for find which works on dynamic predicates of type $\iota \to \mathcal{F}_{E^*}(1)$. find will map dynamic predicates to static predicates using $\downarrow_{\circ} \circ \mathsf{empty} \circ \mathsf{box}$. The use of $\mathsf{empty} \circ \mathsf{box}$ blocks any dynamic effects in the predicate P.[171] This can be intuitively understood as the listener considering whether there exists some x such that $P\,x$ without committing the (dynamic) effects of P to the common ground.

With all of this heavy machinery in place, we now have a system that can deal with sentences like Example 36 and Example 37. The noun phrases *the man who owns a dog* and *the man who loves his dog* both trigger presuppositions with dynamic content. This content is then accommodated globally and all the new discourse referents are available in subsequent discourse.

(36) The man who owns a dog_1 loves it_1.

(37) The man_1 who loves his_1 dog_2 treats it_2 well.

[171] And thanks to separateDynamics, dynamic effects (those from E^*) are the only effects in P.

$$
\begin{array}{ll}
\llbracket \textsc{the}\,(\textsc{who}_{\mathrm{R}}\,(\lambda x.\,\textsc{owns}\,(\textsc{a dog})\,x)\,\textsc{man})\rrbracket & \\
\rightsquigarrow \texttt{presuppose!}\,(\lambda x.\,\texttt{push}\,x\,(\lambda\_. & \approx \texttt{presuppose!}\,(\lambda x.\,\texttt{assert}\,(\mathbf{man}\,x)\,(\lambda\_. \\
\qquad \texttt{assert}\,(\mathbf{man}\,x)\,(\lambda\_. & \qquad \texttt{introduce}\,\star\,(\lambda y. \\
\qquad \texttt{introduce}\,\star\,(\lambda y. & \qquad \texttt{assert}\,(\mathbf{dog}\,y)\,(\lambda\_. \\
\qquad \texttt{assert}\,(\mathbf{dog}\,y)\,(\lambda\_. & \qquad \texttt{assert}\,(\mathbf{own}\,x\,y)\,(\lambda\_. \\
\qquad \texttt{assert}\,(\mathbf{own}\,x\,y)\,(\lambda\_. & \qquad \eta\star)))))) \\
\qquad \eta\star)))))) &
\end{array}
$$

$$
\begin{array}{ll}
\llbracket \textsc{the}\,(\textsc{who}_{\mathrm{R}}\,(\lambda x.\,\textsc{loves}\,(\textsc{poss he dog})\,x)\,\textsc{man})\rrbracket & \\
\rightsquigarrow \texttt{presuppose!}\,(\lambda x.\,\texttt{push}\,x\,(\lambda\_. & \approx \texttt{presuppose!}\,(\lambda x.\,\texttt{assert}\,(\mathbf{man}\,x)\,(\lambda\_. \\
\qquad \texttt{assert}\,(\mathbf{man}\,x)\,(\lambda\_. & \qquad \texttt{introduce}\,\star\,(\lambda y. \\
\qquad \texttt{introduce}\,\star\,(\lambda y. & \qquad \texttt{assert}\,(\mathbf{dog}\,y)\,(\lambda\_. \\
\qquad \texttt{assert}\,(\mathbf{dog}\,y)\,(\lambda\_. & \qquad \texttt{assert}\,(\mathbf{own}\,x\,y)\,(\lambda\_. \\
\qquad \texttt{get}\,\star\,(\lambda e. & \qquad \texttt{assert}\,(\mathbf{love}\,x\,y)\,(\lambda\_. \\
\qquad \texttt{assert}\,(\mathbf{own}\,(\mathsf{sel}_{\mathsf{he}}\,e)\,y)\,(\lambda\_. & \qquad \eta\star)))))) \\
\qquad \texttt{assert}\,(\mathbf{love}\,x\,y)\,(\lambda\_. & \\
\qquad \eta\star))))))))) &
\end{array}
$$

We also derive the intended readings behind Examples 38, 39, 40 and 41, but that was already the case before delving into dynamic presuppositions.

8.7 Summary

In this chapter, we have built up a single grammar that covered anaphora, presuppositions (of referring expressions), deixis (of first person pronouns), quantifiers and conventional implicatures (of appositive clauses). The grammar can derive the meanings of all the sentences we analyzed in Chapters 6 and 7 using the same abstract syntactic structure.[172] Furthermore, the grammar covers sentences which combine these phenomena, such as Example 42.[173]

(42) My best friend, who owns a dog$_1$, said it$_1$ loves everyone.

$$
\begin{array}{l}
\llbracket \textsc{said}_{\mathrm{IS}}\,(\textsc{loves}\,(\textsc{in-situ everyone})\,\textsc{it})\,(\textsc{who}_{\mathrm{S}}\,(\textsc{owns}\,(\textsc{a dog}))\,(\textsc{best-friend me}))\rrbracket \\
\twoheadrightarrow \texttt{speaker}\,\star\,(\lambda s. \\
\quad \texttt{presuppose}\,(\lambda x.\,\texttt{assert!}\,(\mathbf{best\text{-}friend}\,x\,s))\,(\lambda x. \\
\quad \texttt{introduce}^{\mathtt{i}}\,\star\,(\lambda y. \\
\quad \texttt{implicate}\,(\mathbf{dog}\,y)\,(\lambda\_. \\
\quad \texttt{implicate}\,(\mathbf{own}\,x\,y)\,(\lambda\_. \\
\quad \texttt{get}\,\star\,(\lambda e. \\
\quad \texttt{assert}\,(\mathbf{say}\,x\,(\forall z.\,\mathbf{love}\,(\mathsf{sel}_{\mathsf{it}}\,e)\,z))\,(\lambda\_. \\
\quad \eta\star))))))) \\
\\
\downarrow_{\circ}\,(\mathsf{top}\,s\,\llbracket \textsc{said}_{\mathrm{IS}}\,(\textsc{loves}\,(\textsc{in-situ everyone})\,\textsc{it})\,(\textsc{who}_{\mathrm{S}}\,(\textsc{owns}\,(\textsc{a dog}))\,(\textsc{best-friend me}))\rrbracket) \\
\twoheadrightarrow \exists x.\,\mathbf{best\text{-}friend}\,x\,s \wedge (\exists y.\,\mathbf{dog}\,y \wedge \mathbf{own}\,x\,y \wedge \mathbf{say}\,x\,(\forall z.\,\mathbf{love}\,y\,z))
\end{array}
$$

[172] With the exception of in-situ quantifiers which now use the IN-SITU operator.

[173] For the detailed process of computing the meaning of Example 42, take a look in Appendix B.

8.7.1 Note on Conservativity

An interesting feature of the grammar is that as we added more and more phenomena throughout Sections 8.2, 8.3, 8.4 and 8.5, we rarely modified the interpretations of existing entries. For example, we did not need to turn our interpretations of proper nouns or pronouns into generalized quantifiers when adding quantification, we did not need to change the interpretation of transitive verbs to account for the fact that both subject and object might carry conventional implicatures, etc.

The changes that we did to existing interpretations consisted almost exclusively of inserting handlers for new effects which did not exist in the grammar before. For example, in Section 8.2, we added the useFind and maybeAccommodate handlers for the new `presuppose` operation into the box combinator and in Section 8.5, we added the SI handler for the new `scope` operation to the interpretations of tensed verbs. In all those four sections, we also modified the top combinator by composing it with a handler for the newly defined effect. However, we know that if we apply a handler to a computation in which none of the operations being handled occur, then the result will be the same computation (modulo stuck computations). Therefore, we have a strong conservativity result for our incremental grammar. In the grammars developed from Sections 8.1 to 8.5 (Subsection 8.5.1 not included), if an abstract term has an interpretation in one grammar, then it has the same interpretation in every other grammar in which it is derivable (i.e. in every one of its extensions). This is because any extension would only change the interpretation by inserting handlers for effects which are not used in the interpretation and therefore the handlers would make no difference.

9

Conclusion

We start the conclusion of our thesis by summarizing the results we have obtained (Section 9.1). We proceed chapter by chapter and review the material, highlighting the important contributions. We then examine related work out there and compare it to our approach (Section 9.2). We first speak about the calculus $(\!|\lambda|\!)$ itself, considering the alternatives that exist already and explaining in which ways $(\!|\lambda|\!)$ is different (9.2.1). Then, we turn to the linguistic analyses that we presented in our dissertation and relate them to the closest analogues in semantics literature (9.2.2). Finally, we review recent work which focuses on combining multiple monads or linguistic effects in a single grammar and see how $(\!|\lambda|\!)$ compares to those (9.2.3). We end our thesis bringing up directions in which the work presented here can be extended (Section 9.3).

Contents

9.1 Summary of Results **229**
9.2 Comparison with Existing Work **231**
9.2.1 Calculus 231
9.2.2 Linguistic Modelling 234
9.2.3 Combining Linguistic Effects 236
9.3 Future Work **240**
9.3.1 Future Work on the Calculus 240
9.3.2 Future Work on the Linguistic Applications 242

9.1 Summary of Results

In Part I, we have introduced $(\!|\lambda|\!)$, a formal calculus that extends the simply-typed λ-calculus (STLC) with effects and handlers.

The definition of $(\!|\lambda|\!)$ is given in Chapter 1. $(\!|\lambda|\!)$ introduces a new family of types into STLC, the computation types, and new terms, which are built out of computation constructors and destructors. We gave a type system to the calculus which extends that of STLC and a reduction semantics which combines the STLC β and η reductions with definitions of the new function symbols. During the course of the chapter, we maintain two perspectives on the intended meaning of the terms: computations can be seen as programs that interact with a system through a selected set of "system calls" (operations) or they can be seen as algebraic expressions built upon an infinitary algebraic signature.

In Chapter 2, we gave an example of using the $(\!|\lambda|\!)$ calculus. Besides familiarizing the reader with the notation and reductions of the calculus, the example served as a preview of the kind of language engineering we would be doing. During the chapter, we developed a compositional semantics for a simple computing language with errors and variables. This let us demonstrate the modularity of using our computation monad, as we could add variables to the language without needing to modify the semantics of the other constructions.

The main contribution of Part I lies in Chapter 3, in which we developed the metatheory of $(\!| \lambda |\!)$. In Section 3.1, concepts which are primitive in some other languages (closed handlers and the $\gg\!=$ operator) were defined within $(\!| \lambda |\!)$ and their typing rules and reduction rules were derived from those of $(\!| \lambda |\!)$. In Section 3.3, we then connected the calculus to the theory of monads by identifying a monad in the category in which we interpret $(\!| \lambda |\!)$ with our *denotational semantics*. In Section 3.2, we proved *subject reduction* of $(\!| \lambda |\!)$. This result gives a basic coherence between the type system of $(\!| \lambda |\!)$ and the reduction semantics, guaranteeing that types are preserved under reduction. This is complemented by a proof of *progress*, which states that terms which do not use any of the partial operators and which can no longer be reduced must have a very specific shape.

We followed this with another fundamental property: *strong normalization*. Its proof was split into two parts: *confluence* (proved in Section 3.4) and *termination* (proved in Section 3.5). The proofs of both confluence and termination proceed by similar strategies: prove the property for the calculus without η-reduction by applying a general result and then extend the property to the complete calculus. In the case of confluence, the general result is the confluence of orthogonal Combinatory Reduction Systems [76]. In the case of termination, we rely on two techniques: the termination of the reduction relation of Inductive Data Type Systems that validate the General Schema [19] and Higher-Order Semantic Labelling [55], which lets us use our denotational semantics to label the terms of our calculus so that it validates the General Schema.

Andrej Bauer made the analogy that effects and handlers are to delimited continuations what while loops or if-then-else statements are to gotos [15]. Continuations themselves have proven to be a useful tool in natural language semantics [36, 12, 115, 38, 11, 14]. In Chapter 4, we have shown how $(\!| \lambda |\!)$ can simulate delimited continuations, namely the `shift/reset` delimited control operators. We presented a typed call-by-value λ-calculus with `shift` and `reset` and have simulated its types and reductions in $(\!| \lambda |\!)$.

In Part II, we have demonstrated the applications of $(\!| \lambda |\!)$ to the problem of modelling the meaning of natural language utterances.

After having reviewed the basics of formal semantics in Chapter 5, we have shown how computations in $(\!| \lambda |\!)$ can be used to give a compositional account of several "non-compositional" linguistic phenomena in Chapter 6. We have described how to introduce computations into a compositional semantics while preserving the meanings assigned by the semantics in Section 6.1. We have then presented analyses of several linguistic phenomena in the framework of $(\!| \lambda |\!)$ computations: deixis (Section 6.2), conventional implicature à la Potts (Section 6.3) and quantification à la Montague (Section 6.4). We have then explicitly described the methodology used to find the kinds of analyses that we have presented in Chapter 6 as to encourage researchers to develop other analyses in the framework.

We dedicated Chapter 7 to a particularly complex phenomenon: dynamics. In Sections 7.1 and 7.2, we have shown how a $(\!| \lambda |\!)$ analysis of dynamics can be extracted from Discourse Representation Theory. This gave us a way to handle dynamics in $(\!| \lambda |\!)$ as well as strengthen the claim that effects and handlers are suitable mechanisms for dealing with natural language. We have also shown how to interpret the $(\!| \lambda |\!)$ computations as the dynamic propositions of de Groote's Type-Theoretic Dynamic Logic (TTDL) [38]. In her dissertation [80], Lebedeva extended TTDL with exceptions to treat presuppositions and in Section 7.3, we integrated Lebedeva's analysis of presupposition into our $(\!| \lambda |\!)$ analysis of dynamics (we compare our adaptation with the original in the next section, 9.2.2). In Section 7.4, we considered another extension of TTDL, Double Negation TTDL [111] and we have shown why the kind of generalization of denotations done in Double Negation TTDL is not amenable to being analyzed as a side effect in $(\!| \lambda |\!)$.

In Chapter 8, we have supported our claim that using effects and handlers lets us combine distinct phenomena in a single grammar. We have started with the dynamic grammar developed in Chapter 7, repeated in Sections 8.1 and 8.2. We have then extended this grammar with conventional implicatures (8.3), deixis (8.4) and quantification (8.5) with little to no modification of the original semantics. We finished the chapter by sketching out an analysis of restrictive relative clauses and their interactions with presuppositions in Section 8.6.

9.2 Comparison with Existing Work

9.2.1 Calculus

$(\!| \lambda |\!)$ can be compared to several existing calculi and implementations of effects and handlers:

- System F (i.e. the polymorphic λ-calculus or the second-order λ-calculus)

  $(\!| \lambda |\!)$ extends the simply-typed λ-calculus with computation types $\mathcal{F}_E(\alpha)$. Computations are algebraic expressions and as such can be expressed as inductive data types.[174] Inductive data types, along with the sums and products that we add to the calculus in Section 1.5, can be expressed in System F [132].

  In $(\!| \lambda |\!)$, a computation of type $\mathcal{F}_E(\alpha)$ can also be given the type $\mathcal{F}_{E \uplus E'}(\alpha)$, where $E \uplus E'$ is an extension of E. However, in the direct encoding of $(\!| \lambda |\!)$ into System F, for every effect signature $E \uplus E'$ that we would like to ascribe to a computation, we would end up with a different term. On the other hand, in $(\!| \lambda |\!)$ we can keep using the same term. This lets us give a semantics to lexical items that does not have to change when new effects are introduced into the theory.

- *Eff*

  The *Eff* language [16] is an ML-like programming language with effects and handlers. Like in ML, effects can be freely used within any expression, without any term encoding (we say that the calculus is *direct-style*). For this to work correctly, the calculus has a fixed evaluation order, which, following ML, is call-by-value.

  We have used *Eff* in our first explorations of effects and handlers in natural language semantics [89], benefiting from the existing implementation. However, we have found that besides call-by-value, call-by-name evaluation is also common, notably on the boundaries of lexical items (see 6.5.2). Call-by-name can be simulated in call-by-value by passing around thunks (functions of type $1 \to \alpha$ for some α). However, in the presence of both call-by-name and call-by-value, we have opted for an indirect presentation of effects using monads which favors neither call-by-value nor call-by-name and that lets us manipulate the order of execution using $\gg\!=$.

  Finally, we note that *Eff* is a general-purpose programming language which includes general recursion (`let rec`) and therefore it is not terminating, contrary to $(\!| \lambda |\!)$.

- λ_{eff}

  The λ_{eff} calculus [63] is a call-by-push-value λ-calculus [82] with operations and handlers. Call-by-push-value is special in introducing two kinds of terms: computations and values. The intuition behind the two is that computations *do*, whereas values *are*. Two of the crucial things that computations do are to pop values from a stack (that is what abstractions do) and to push values to the stack (that is what applications do). Therefore, applications and abstractions are considered as computations. Furthermore, the function in an application term must be a computation term (which is expected to, among other things, pop a value from the stack), whereas the argument, which is the value to be pushed to the stack, must be a value term.

  This might make it look like that call-by-push-value is like call-by-value since all the arguments passed to functions are values. However, in true call-by-value, we can use complex expressions as arguments and we expect that the reduction system will evaluate the arguments down to values before passing them to the function. To do this in call-by-push-value, we have to implement this manually by evaluating the argument computation down to a value x and then passing the value x to the function in question (i.e. in λ_{eff} syntax **let** $x \leftarrow M$ **in** $F\,x$). In $(\!| \lambda |\!)$, this amounts to the term $M \gg\!= F$, where $M : \mathcal{F}_E(\alpha)$ and $F : \alpha \to \mathcal{F}_E(\beta)$. To implement call-by-name, computations can be mapped to values by wrapping them in thunks, which are primitive constructs in call-by-push-value (in λ_{eff} syntax $F\,\{M\}$, where M is a computation and the thunk $\{M\}$ is a value). In $(\!| \lambda |\!)$, the corresponding term is $F\,M$, where $M : \mathcal{F}_E(\alpha)$ and $F : \mathcal{F}_E(\alpha) \to \mathcal{F}_E(\beta)$.

[174] An inductive type is a recursive type with positive constructors. In 3.5.3, we have seen that a computation type $\mathcal{F}_E(\alpha)$ has positive constructors η and $\mathtt{op}$ for every $\mathtt{op} \in E$.

λ_{eff} presents an intriguing alternative to $(\!| \lambda |\!)$. The call-by-push-value calculus is flexible enough to accommodate both call-by-name and call-by-value. λ-abstractions and operations are both treated as effects, which might make the definition of the $\mathcal{C}$ operator, which permutes λ with operations, more intuitive.[175] λ_{eff} also has a well-developed metatheory, developed in [63]: it is both confluent (due to its reduction relation being deterministic) and terminating (thanks to its effect type system).

λ_{eff} served as an inspiration to the design of $(\!| \lambda |\!)$; notably, $(\!| \lambda |\!)$'s effect system is based on that of λ_{eff}. However, $(\!| \lambda |\!)$ diverges from λ_{eff} in that it is a proper extension of the simply-typed λ-calculus (STLC): every term, type, typing judgment or reduction in STLC is also a term, type, typing judgment or reduction in $(\!| \lambda |\!)$. For example, the STLC term $\lambda x.\, x$ is not a λ_{eff} term. Its closest counterparts in λ_{eff} would be either $\vdash_{\emptyset} \lambda x.\, \mathbf{return}\, x : A \to F(A)$, where A is a value type, or $\vdash_E \lambda x.\, x! : U_E(C) \to C$, where C is a computation type (a function or an effectful computation). On the other hand, in $(\!| \lambda |\!)$, $\vdash \lambda x.\, x : \alpha \to \alpha$ is a valid term for any α, be it an atomic type such as o, a function type such as $\iota \to o$ or a computation type such as $\mathcal{F}_E(o)$.

The fact that $(\!| \lambda |\!)$ is an extension of STLC motivates its use for two reasons. First, STLC is the lingua franca of formal semantics. $(\!| \lambda |\!)$ already introduces a lot of new notation and the use of effects in natural language semantics is not yet ubiquitous. By basing $(\!| \lambda |\!)$ on STLC, we narrow the gap between the common practice of formal semantics and our use of effects and monads, hopefully making the technique more approachable to researchers in the field. Second, the purpose of the calculus is to write down computations that produce logical representations. By having STLC as a subpart of $(\!| \lambda |\!)$, terms of Church's simple theory of types (i.e. formulas of higher-order logic) are already included in our calculus and we can reuse the same notions of λ-abstraction and variables. In λ_{eff}, we would either need to add constructors for logic formulas (i.e. having some logic as an object language over the terms of which the meta language λ_{eff} would calculate) or use call-by-push-value computations in our logical representations.

- Extensible Effects of Kiselyov et al [71] and other implementations of effect systems in pure functional programming languages (Haskell, Idris …)

  Our adoption of a free monad and effect handlers was motivated by the paper of Kiselyov, Sabry and Swords on *extensible effects* [71]. The paper presented a Haskell library for encoding effectful computations, combining computations with diverse effects and interpreting them by composing a series of modular interpreters. The library used a free monad (in the style of [123]): a computation is either a pure value (η in $(\!| \lambda |\!)$) or a request to perform some kind of effect (an operation in $(\!| \lambda |\!)$). These requests are then handled by interpreters which behave similarly to effect handlers (the authors of [71] also relate handlers to the technique of "extensible denotational language specifications" published in 1994 by Cartwright and Felleisen [24]). The paper demonstrated that the approach is more flexible when it comes to combining interacting effects than the existing state-of-the-art technique of using monad transformers. A more refined version of the approach was published in [70] and similar implementations of effects and handlers exist also in other pure functional programming languages such as Idris [22].

  The extensible effects discipline provides the tools that we would like to use to build a modular semantics of natural language. However, we do not want our formal semantics to depend on the semantics of a large programming language such as Haskell[176] or Idris. We created $(\!| \lambda |\!)$ to reap the benefits of extensible effects without incurring the complexity of using a language like Haskell as our meta language. $(\!| \lambda |\!)$ extends STLC only with computation types, two constructors (η and operations), two destructors (handlers and $\downarrow_{\circ}$) and the $\mathcal{C}$ operator. Unlike Haskell, our extension of STLC preserves strong normalization.

[175] The extra typing rule for the $\mathcal{C}$ construction in λ_{eff} would look like this:

$$\frac{\Gamma \vdash_E M : A \to C}{\Gamma \vdash_E \mathcal{C}\, M : F(U_{\emptyset}(A \to C))}$$

[176] The implementations of extensible effects in Haskell make use of a wealth of language extensions which are not even part of the Haskell standard.

The Case of the $\mathcal{C}$ Operator

A notable feature which distinguishes $(\!|\lambda|\!)$ from all of the above-mentioned calculi is the $\mathcal{C}$ operator. The $\mathcal{C}$ operator was added relatively late to the $(\!|\lambda|\!)$ calculus as a solution to the following problem.

(43) A man$_1$ walks in the park. He$_1$ whistles.

In Example 43, the noun phrase *a man* introduces a quantifier ranging over men that takes scope over its continuation $\lambda x.\, [\![x \text{ walks in the park. He whistles.}]\!] : \iota \to \mathcal{F}_E(o)$.[177] The problem is how to combine an effectful predicate of type $\iota \to \mathcal{F}_E(o)$ with a quantifier such as $\exists : (\iota \to o) \to o$. The key insight is that the effects of the continuation usually do not depend on the individual being talked about. No matter whether the sentence speaks about Albert, Bill or Charles, the continuation will proceed the same. The continuation will look into the context to retrieve the antecedent x, which is known to satisfy the predicate **man** and therefore be eligible for use with a masculine pronoun. It will then produce the logical formula **walk-in-the-park** $x \wedge$ **whistle** x. The resolution of the anaphoric pronoun does not depend on the particular individual being discussed.

DRT is capable of deriving a meaning for Example 43 without considering which individual the noun phrase *a man* refers to. DRT does this by calculating with symbolic representations. The noun phrase *a man* will introduce a *discourse referent*, a symbolic object distinct from any individual found in the model. Then the evaluation of the discourse continues and the anaphoric pronoun can resolve to this symbolic object. We could do the same in $(\!|\lambda|\!)$. We could state that the ι type is not the type of individuals in the model, but the type of some symbolic objects (discourse referents or variables), and that the type o is not the type of propositions, but the type of logical formulas. Assuming that we had an operation $\texttt{gensym} : 1 \rightarrowtail \iota$ which could give us fresh variables and that the type of the constructor for formulas of existential quantification would be $\exists : \iota \to o \to o$, we could wrap the existential quantifier over an effectful computation $P : \iota \to \mathcal{F}_E(o)$ the following way:

$$\begin{aligned}&\texttt{gensym} \star (\lambda x.\\ &P\, x \gg\!= (\lambda p.\\ &\eta\, (\exists\, x\, p)))\end{aligned}$$

By changing the meaning of the type ι to be the type of symbolic references, we solve our problem. When we need to evaluate the effects of a continuation of type $\iota \to \mathcal{F}_E(o)$, we do not need to know the precise identity of the individual ι. Instead, we apply the continuation to a symbolic object which will stand in for any such individual and then proceed to evaluate the effects of the continuation. This is the approach that we were using at the beginning of the project [89].

However, this approach has several downsides. First, our formal semantics is contaminated by extra complexity due to the management of variables and operations like `gensym`. Second, there is an extra level of indirection in place. Instead of computing the truth value of a sentence in some model, we compute a formula and then that formula itself can be evaluated in a model. Third, binding in logical formulas is no longer managed by the (meta) calculus. This means that it is easy to `gensym` a variable that is supposed to occur in the scope of a quantifier and then have that variable accidentally project outside of its scope, leading to the generation of malformed logical formulas.

The $\mathcal{C}$ operator presents another solution to this problem. If we know that in a continuation of type $\iota \to \mathcal{F}_E(o)$, the effects of the $\mathcal{F}_E(o)$ do not depend on the ι, we could move them out of the body of the function. This is exactly what the $\mathcal{C}$ operator does.[178] With it, we can wrap the existential quantifier over the continuation using the following expression, which we called $\exists_{\gg} P$ in 7.3.1:[179]

$$\exists \cdot\!\gg (\mathcal{C}\, P)$$

[177] In our presentation of dynamic semantics, where $[\![S]\!] = \mathcal{F}_E(1)$, the type of the continuation would be $\iota \to \mathcal{F}_E(1)$.

[178] If we are mistaken and the effects do depend on the bound variable, then computation gets stuck (instead of producing a malformed logical formula).

[179] The type of $\exists$ is $(\iota \to o) \to o$ and the type of P is $\iota \to \mathcal{F}_E(o)$.

The inspiration for the $\mathcal{C}$ operator came from Philippe de Groote's work on logical relations and *conservativity* [39]. Within his work, de Groote makes use of the following mathematical structure: a type transformer T equipped with three operation U, $\bullet$ and C with the following types:

$$\begin{aligned}
&\mathsf{U} : \alpha \to \mathsf{T}\alpha \\
&\bullet : \mathsf{T}(\alpha \to \beta) \to \mathsf{T}\alpha \to \mathsf{T}\beta \\
&\mathsf{C} : (\alpha \to \mathsf{T}\beta) \to \mathsf{T}(\alpha \to \beta)
\end{aligned}$$

and obeying the following laws:

$$\begin{aligned}
(\mathsf{U}\, f) \bullet (\mathsf{U}\, a) &= \mathsf{U}\,(f\, a) \\
\mathsf{C}\,(\lambda x.\, \mathsf{U}\,(f\, x)) &= \mathsf{U}\, f
\end{aligned}$$

If we ignore the C operator for the moment and focus only on the T type transformer, the U and $\bullet$ operations and the first of the two laws, we see that we have a weaker version of an applicative functor. An applicative functor is a functor, which, in our category, is a type transformer such as T,[180] equipped with two operations, pure and $\circledast$, which have the same types as U and $\bullet$ respectively, and which satisfy the four laws given in 3.3.5. The first of the two laws given above is the homomorphism law 3.8 of applicative functors. We note that all three instances of this structure that are used in the examples of [39] also satisfy the other three applicative functor laws.

We now look at the C operator, whose type is the same as the $\mathcal{C}$ operator that we added to $(\!|\, \lambda \,|\!)$ (with $\mathcal{F}_E$ corresponding to T). Its behavior is specified only by the second law, which defines the value of C when applied to a pure function (a function of the form $\lambda x.\, \mathsf{U}\,(f\, x)$). Our $\mathcal{C}$ also obeys this law by including it as the $\mathcal{C}_\eta$ reduction rule, which is part of the definition of $\mathcal{C}$:[181]

$$\mathcal{C}\,(\lambda x.\, \eta\,(f\, x)) \;\to_{\mathcal{C}_\eta}\; \eta\,(\lambda x.\, f\, x) \;\to_\eta\; \eta\, f$$

Our definition of $\mathcal{C}$ also includes the $\mathcal{C}_{\mathsf{op}}$ reduction rule. This extends the definition of $\mathcal{C}$ to computations which use operations. However, our definition of $\mathcal{C}$ is not total, as there are well-typed arguments for $\mathcal{C}$ which will cause computation to get stuck. This correlates with the C operator in de Groote's work, where certain values of the $\mathcal{C}$ operator are left undefined.

It is interesting that the $\mathcal{C}$ operator which turned out to be useful in our work originated in another work whose objective was also to capture the way meanings are extended in formal semantics. In our approach, we address conservativity,[182] the subject of de Groote's paper, in two ways. First, we show how to lift a simply-typed semantics into a monadic semantics while preserving the meanings it assigns to sentences in Section 6.1. Then, in Chapter 8, we extend a dynamic grammar with presuppositions (8.2), conventional implicature (8.3), deixis (8.4) and quantification (8.5), and at every step, the only modifications we make to existing entries is to add handlers for new effects, which does not affect the meaning of any of the sentences analyzed before. Furthermore, since for any effect signature E, $\mathcal{F}_E$ is a type transformer which together with η as U, $\ll\cdot\gg$ as $\bullet$ and $\mathcal{C}$ as C has the structure needed in [39], de Groote's technique of logical relations should apply to the kinds of meanings produced by our approach as well.[183]

9.2.2 Linguistic Modelling

Most of the linguistic analyses presented in our thesis are translations of existing analyses to the $(\!|\, \lambda \,|\!)$ framework:

[180] Ignoring the arrow-component of the functor.

[181] The fact that we can take this law and use it as a definition of the $\mathcal{C}$ operator is due to η being a constructor in $(\!|\, \lambda \,|\!)$, η-headed expressions are not head-reducible and therefore can be reliably pattern-matched on.

[182] Conservativity is understood as the notion that extending the grammar to cover a new phenomenon should not change the meanings which were already correct in the simpler grammar.

[183] Modulo the partiality of $\mathcal{C}$.

- Quantification

  Our technique of using the `scope` operation follows approaches that propose the use of continuations in natural language [36, 12]. The type and the semantics of our `scope` operation match those of `shift`, a control operator for delimited continuations used to treat quantification in [115]. Instead of context levels, we use quantifier raising to account for scope ambiguity. As in Montague's work [98], we have a syntactic construction rule that takes a quantificational noun phrase and uses it to bind an NP variable within a sentence (QR : $QNP \multimap (NP \multimap S) \multimap S$). The syntactic type of QR is based on the type given to the determiners *every* and *some* in [105] ($C_{\text{every}} : N \multimap (NP \multimap S) \multimap S$). By composing $\textsc{every} : N \multimap QNP$ and QR : $QNP \multimap (NP \multimap S) \multimap S$, we get $\lambda N.\, \text{QR}\,(\textsc{every}\, N) : N \multimap (NP \multimap S) \multimap S$. In 6.4.2, we have also pointed out that a $(\!|\lambda|\!)$ computation invoking a series of `scope` operations is like a nested sequence of quantifiers in Keller storage [66]. However, unlike Keller storage (and Cooper storage [32]), the quantifiers cannot be retrieved in any order since one could bind a variable in another.[184] Therefore, we solve ambiguity by other means (QR).

- Conventional Implicature

  Our analysis stems from Potts' dissertation on the logic of conventional implicatures [108]. This multidimensional approach to the semantics of conventional implicatures was adapted into a monadic treatment by Giorgolo, Asudeh et al in 2011 [51, 48]. Monadic treatments of conventional implicatures have also appeared in recent ESSLLI courses: the treatment of expressives by Kiselyov and Shan at ESSLLI 2013 [72] and by Barker and Bumford at ESSLLI 2015 [13]. The underlying monad in all the monadic treatments is the writer monad. The writer monad is parameterized by a monoid. Every computation in the monad is accompanied by an element of the monoid and composing computations has the effect of combining their monoidal components. In the case of the writer monad for conventional implicatures, the monoid is some monoid of implicatures, usually the conjunctive monoid on propositions. The structure needs to be a monoid because we need a neutral element for phrases without any conventional implicatures and we need associativity of the monoid to get associativity of the monad.

  In the free monad approach, we can characterize the writer monad by a single operation which computations can use to communicate elements from the monoid. This is what we do in Section 6.3, with `implicate` : $o \rightarrowtail 1$ as the operation and accommodate′ from 6.3.1 as the handler which defines the monoid ($\wedge$ as the binary operation in the `implicate` clause and $\top$ as the neutral element in the η clause). This treatment of conventional implicatures is equivalent to the existing monadic analyses.

  However, in Section 8.3, we study the interaction of dynamics with conventional implicatures. In our approach, the at-issue truth conditions are communicated by the two operations `assert` : $o \rightarrowtail 1$ and `introduce` : $1 \rightarrowtail \iota$. We take our cue from Layered DRT [46] and Projective DRT [131] to do the same for conventional implicatures: we complement `implicate` : $o \rightarrowtail 1$ with $\texttt{introduce}^{\texttt{i}} : 1 \rightarrowtail \iota$. This way, the monadic structure behind conventional implicatures becomes similar to the one used for dynamics (continuations with state). Conventional implicatures become updates to the context which are not blocked by logical operators such as negation. In our system, we get a compositional semantics which can account for binding out of appositives, as in Example 29.

  (29) John, who nearly killed a woman$_1$ with his car, visited her$_1$ in the hospital.

- Deixis

  In his treatment of indexicals, Kaplan models meanings as *characters*, functions from contexts to intensions. Contexts have several components, among them the *agent* which is making the utterance. In our approach, we add getters for the components of the context. In dynamics, we use `get` : $1 \rightarrowtail \gamma$ to get the anaphoric part of the context, and in treating the first-person pronoun, we use `speaker` : $1 \rightarrowtail \iota$ to recover the speaker of the utterance.

[184] We could imagine adding a rule allowing us to permute the order of certain operations, as long as the result of one is not bound in the input to the other. However, we would lose confluence of our calculus at that point.

We use deixis as the first example in Chapter 6. Wanting to give an example of a handler for `speaker` that appears within a lexical entry, we turned to direct quotations. However, the meanings that we give to sentences such as Example 44 only express part of the meaning. Namely, we capture the fact that Peter claims that Mary kissed him. Nevertheless, the sentence also entails that Peter uttered the sentence "Mary kissed me". Therefore, in our analysis, the sentence in Example 44 would be judged equivalent to the sentence in Example 45. The meaning of direct quotation does not depend only on the meaning of the quoted clause, but on its exact form. Since our system is strictly compositional,[185] we would need to make it so that the meaning of every sentence is the sentence itself. More elaborate treatments of quotation can be found in [47, 109, 119].

(44) Peter said "Mary kissed me".

(45) Peter said "I was kissed by Mary".

- Dynamics

  Our treatment of dynamics was presented as a reengineering of DRT and hence it covers DRT as it is presented in Chapter 1 of [64]. It also originated as a reconstruction of de Groote's Type Theoretic Dynamic Logic (TTDL) [38, 80] using monads of effects and handlers [89]. Therefore, it also covers the fragment of TTDL presented in [38] and [80].

  Giorgolo and Unger have used the state monad to model DRT-style dynamics [52] and the technique was later refined by Unger [129]. The most challenging aspect in both solutions ends up being the management of accessibility with respect to the scope of quantifiers or logical operators such as negation. In the former [52], the state is a tree data structure with a pointer and the denotations manipulate it by appending children to the tree and moving the pointer up the tree. In the latter [129], the state is a stack of contexts and denotations have to allocate new contexts on the stack and clear them from the stack when the contexts should no longer be accessible. This pattern of encapsulating the dynamic effects of some part of a sentence lends itself very well to the handler abstraction. In our treatment of dynamics, we use an effect handler (box) and avoid the overt manipulation of any tree or stack of contexts; our contexts are just sets discourse referents and propositions.

- Presuppositions

  In her thesis [80], Lebedeva gave a compositional account of presupposition projection and accommodation by introducing exceptions and handlers into TTDL. Our use of effects and handlers was strongly influenced by Lebedeva's use of exceptions and so our analysis of presuppositions follows the same strategy. However, we argue that the resumable nature of effects makes them more appropriate for the treatment of presuppositions. We have given a detailed comparison of our approach and Lebedeva's original in 7.3.5, which we summarize here:

  - We preserve strong normalization, even though we have exceptions and recursion (in handlers).
  - We do not licence cataphoric binding from presupposition triggers because we can resume on presupposition failure without reevaluating the previous parts of discourse.
  - We reuse the same mechanism we have used to treat dynamicity, effects and handlers (i.e. the same formal apparatus that projects context updates outside of sentence boundaries is used to project presuppositions outside of downward entailing contexts).

9.2.3 Combining Linguistic Effects

Our work is not the only work aimed at combining different linguistic effects in a single grammar. During our research, several proposals have appeared. Below, we relate them to our approach.

[185] Because we are using abstract categorial grammars for the syntax-semantics interface.

- Monad Transformers [27, 13]

  Monad transformers map simpler monads into more complex monads with some additional structure. If we want to have a monad with enough structure for, e.g., state, nondeterminism and exceptions, we can take the corresponding monad transformers and apply them, one after another, to some base monad. Monad transformers were evoked as a possible solution to the problem of composing monads by Shan in his first paper on monads for natural language semantics [113]. Since then, they have been used by Charlow in his dissertation [27] and featured in Barker and Bumford's ESSLLI 2015 course [13].

  To motivate one of the reasons one might prefer effects and handlers instead of monad transformers, we will look at an example lexical entry from [27]:

$$\textbf{pro} := \lambda s.\, \{\langle s_\top, s\rangle\}$$

  The monads in use here are the state monad to treat discourse updates and the set monad to treat indefinites. Meanings in the composite monad can depend on some discourse state, they can modify the discourse state and they can propose a set of possible readings. The meaning of pronouns, **pro**, relies on one of these three capabilities. It makes use of the discourse state s, in which it looks for the salient ("topical") referent, $s_\top$. Then, to have the correct type and fit in with the other definitions, it must also supply the output state and the set of possible readings. The output state will be just the input state s and the set of possible readings will be the singleton set $\{\langle s_\top, s\rangle\}$. This is the case even though the pronoun has no nondeterministic effect, i.e. its structure in the set monad is trivial. We contrast this with the entry for pronouns that we use in $(\!|\, \lambda \,|\!)$:

$$[\![\textsc{she}]\!] = \texttt{get} \star (\lambda e.\, \eta\, (\mathsf{sel}_{\mathsf{she}}\, e))$$

  We use the `get` operation to access the discourse state e, which corresponds to the λs abstraction in the monad transformer example. Then we use the $\mathsf{sel}_{\mathsf{she}}$ function to retrieve the salient referent from the state of discourse e, which corresponds to the use of the $\__\top$ operator. Finally, we return this referent as the result using η, which corresponds to packing the referent in a pair with the output state s and then wrapping it in a singleton set. Note that no other effect (such as `introduce`, which would be the analogue to the set monad structure) is mentioned within the lexical entry. If we were to extend the grammar with new effects, the interpretation of $[\![\textsc{she}]\!]$ would stay the same. On the other hand, if we were to add a new monad to the stack of monad transformers in the former example, then the semantics of **pro** would need to be lifted.

  There is a way to avoid lifting when working with monad transformers. We characterize every monad transformer by some capabilities/operations it gives us and then we write abstract polymorphic terms which can be interpreted in different monads provided that they have enough structure to interpret the capability/operation. This is the method presented in [84, 61] and used in (Haskell) libraries implementing monad transformers [3]. However, formalizing this method already leads us half of the way towards effect and handlers (we write computations using abstract operations and the type of the computation indicates the operations that must be interpreted).

- Setting the Monads Loose [31, 50, 30, 28, 29]

  During his invited lecture at Barker and Bumford's ESSLLI 2015 course [13, 31], Simon Charlow presented another strategy for combining the different monads. It consists of adding the η and the $\gg\!=$ of every monad to the lexicon as type shifters. When given a sentence, the grammar then assigns a reading to every possible way of gluing the meanings of the parts of the sentence in a type-sound way. There is therefore no need to define any supermonad to serve as the universal glue since the task of gluing together the pieces from different monads is up to the parser. The same strategy was also adopted by Giorgolo and Asudeh in their ESSLLI 2015 course [50]. They presented fragments of type-logical grammars with different modalities $\diamond_i$ for different monads. The logic itself contained inference rules which correspond to the monadic primitives (e.g. η is the rule which lets us deduce $\diamond A$ from A). Giorgolo and Asudeh further strengthen the glue available

to the logic by adding distributivity laws: rules that can commute certain monadic structures (e.g. $\diamond_i \diamond_j A \to \diamond_j \diamond_i A$).

In our thesis, instead of going this road, we build a composite (free) monad. We do so because of two reasons. First, our investigations are motivated by the search for a wide-coverage abstract categorial grammar (ACG) with formal semantics. In ACGs, derivations are expressed in an abstract grammar, which intuitively corresponds to a level of deep syntax. Using a similar approach would force us to introduce semantic types into the level of abstract syntax. Furthermore, the logic which licences derivations would need to be extended to include monadic types (i.e. we would need to add modalities to the implicative fragment of linear logic used in ACGs), which means we would risk losing the existing ACG meta-theory, such as parsing results. Second, the point of using $(\!| \lambda |\!)$ is to deal with "non-compositional" phenomena in a compositional setting. If we rearrange the deep syntax so as to facilitate composition, then we could be considered cheating.[186]

- Conservativity via Logical Relations [39]

  In [39], de Groote proves a general conservativity result that guarantees the preservation of predicted meanings under a class of extensions/embeddings, including those that a semanticist might use when introducing a new phenomenon to a grammar. While this gives us a way of porting an existing grammar into a new framework with different types of interpretation, it does not tell us how to add new entries to the ported grammar. As an example, we will take the dynamic grammar from [27], which we featured in the monad transformer example with the following example:

$$\textbf{pro} := \lambda s.\, \{\langle s_\top, s\rangle\}$$

  Suppose we want to introduce conventional implicatures (CI) to this grammar and we do so by embedding the interpretations into a domain in which at-issue meanings are paired with CI contributions. If our embedding function satisfies the preconditions in [39], we will have preserved all of the existing meanings. Now, we want to add non-restrictive relative clauses (NRRCs), which contribute CIs. However, we have to give their meaning in a domain with states, sets and conventional implicatures. Below is the de-sugared semantics that Charlow gives to the NRRC construction [28]:

$$\textbf{comma}\,\mathsf{k} = \lambda x.\, \lambda s.\, \{\langle x \bullet p, s'\rangle \mid \langle p, s'\rangle \in \mathsf{k}\, x\, s\}$$

  The salient part of the entry is that the referent of the appositive is x and the contributed CI is p, the meaning of the relative clause k applied to x. However, we also need to handle the state and so we abstract over s, pass it to $\mathsf{k}\, x$ to get an output state s' and then bundle the output state s' with our result $x \bullet p$. Furthermore, there is the set structure, and so we need to take into account the fact that $\mathsf{k}\, x\, s$ gives us a set with multiple pairs $\langle p, s'\rangle$ and we need to give our meaning for all of them. The structure of this entry on the set monad level and state monad level is trivial since (under this analysis) NRRCs have no interesting interaction with discourse state or indefiniteness. As the number of phenomena in the grammar grows, we would like a way to abstract over the non-relevant parts of the meaning structure. This is something that can be seen in the sugared version of the semantics that Charlow gives to this operator [39]:

$$\begin{aligned}\textbf{comma}\,\mathsf{k} = \lambda x.\, \mathsf{do}\ \ & p \leftarrow \mathsf{k}\, x \\ & \mathsf{return}\colon\ x \bullet p\end{aligned}$$

  as well as in our lexical entry for the same construction:[187]

[186] Though when it comes to "non-compositional" phenomena, cheating compositionality might actually be the methodologically sound approach.

[187] The entry we see here is simplified from the one in 6.3 to match that of Charlow. This is because in ACGs, we must specify the meaning of the phrase X, *who* K, in terms of the meanings of X and K, where X is a noun phrase with potentially complex monadic structure and K is a sentence missing such a noun phrase (i.e. a relative clause). Therefore, in the entry in 6.3, we have to evaluate $X : \mathcal{F}_E(\iota)$ down to $x : \iota$ and when applying K to x, we have to wrap x in η. In Charlow's approach, it is not the duty of the lexical entries to manage evaluation. Rather, it is the grammar which includes the necessary combinators and type shifters which will evaluate X for us.

$$\begin{aligned}[\![\textsc{who}_{\mathsf{s}}\,\mathsf{k}]\!] = \lambda x.\ \mathsf{k}\,x \gg\!= (\lambda p.\\ &\texttt{implicate}\,p\,(\lambda\_.\\ &\eta\,x))\end{aligned}$$

While the conservativity result of de Groote lets us be confident when porting a grammar to a different type of interpretation, it does not help us when extending it with new entries. It is this problem that is addressed by the techniques presented here and in the works of Simon Charlow. Specifically, in our setting, we do not view the different phenomena in a language as forming a hierarchy in which one extends the other.[188] Instead, it is the case that any lexical entry can use any subset of the phenomena it needs. For example, *she* is anaphoric but not quantificational, *every man* is quantificational but not anaphoric, *you* is neither anaphoric nor quantificational but deictic, *your husband* is both deictic and presuppositional but not quantificational…

- Applicative Abstract Categorial Grammars [68, 69]

  In Subsection 3.3.5, we spoke about applicative functors. Applicative functors generalize monads since every monad is also an applicative functor but not vice versa. The structure of an applicative gives us two combinators, $\mathsf{pure} : \alpha \to F\alpha$ and $\circledast : F(\alpha \to \beta) \to F\alpha \to F\beta$. The intuition behind applicative functors vs monads is that applicatives embody computations with static control flow whereas monads embody computations with dynamic control flow [86].

$$\begin{aligned}\circledast &: F(\alpha \to \beta) \to F\alpha \to F\beta\\ =\!\!\ll^{189} &: (\alpha \to F\beta) \to F\alpha \to F\beta\end{aligned}$$

  When combining computations using $\gg\!=$, as in the case of a monad, we can take a computation $F\alpha$ and chain it with a continuation of type $\alpha \to F\beta$, where the computational structure (i.e. the effects) can vary depending on the value of type α. However, when combining computations using $\circledast$, as in the case of an applicative, we can only take two computations whose computational structure is already fixed.

  Kiselyov argues that since sentence structure is itself also static, applicative functors are a good fit [68]. In Section 6.1, we have seen that we can lift the semantics of a second-order abstract categorial grammar (ACG) into computations using only η and $\ll\cdot\gg$, which constitute the applicative structure of $(\!|\,\lambda\,|\!)$. Second-order ACGs are relevant not only because they can be efficiently parsed [65], but also because they are sufficient to encode mildly context-sensitive grammar formalisms such as tree-adjoining grammars [37]. If second-order ACGs are sufficient to model language, then applicative functors could be sufficient to model their meanings. The switch to actually using applicative functors can then be motivated on the basis of the fact that applicatives are composable, unlike monads.[190]

  Kiselyov's applicative abstract categorial grammars (AACGs) exploit this. In a second-order ACG, terms of the abstract language are formed only by constants and applications. In AACGs, the abstract and object languages are defined to be so-called *T-languages*, which admit only constants and a binary application operator. Terms of an abstract T-language are then homomorphically mapped to λ-terms that produce terms in an object T-language. As in all ACGs, the function type of the abstract language corresponds to functions. Furthermore, as in our approach, atomic abstract types are mapped to types of the form $\mathcal{F}(\alpha)$, where $\mathcal{F}$ is some applicative functor and α some object-level type.

  The biggest difference between AACGs and our approach is whether we use a composition of applicative functors or a free monad of effects and handlers. The use of applicative functors relies on

[188] There are exceptions, such as our treatment of presuppositions, which extends our treatment of dynamics.

[189] $=\!\!\ll$ is $\gg\!=$ with its arguments reversed.

[190] However, the applicative structure that one gets by composing two monad transformers is not necessarily the same as the one that is obtained by composing the corresponding applicative functors.

the assumption that the computational structure (i.e. the control flow) in the semantics is always static. However, even though the structure of sentences is static, we find that the control flow in the semantics can be dynamic. For example, if we consider the noun phrase *her car*, then the pronoun is a computation of an individual, type $\mathcal{F}(\iota)$, and the genitive construction is a function over individuals that triggers presuppositions, type $\iota \to \mathcal{F}_E(\iota)$. To find the referent of this noun phrase, we first have to evaluate the pronoun down to an individual x and then apply the meaning of the genitive construction to this individual, triggering the presupposition that x owns a car. We cannot evaluate the effects of the genitive construction independently of the pronoun since we would not know whose car we are presupposing the existence of and therefore we cannot combine the two meanings using only the $\circledast$ combinator we have in an applicative functor.

Note that all of the work on combining linguistic effects that was cited in this subsection follows the publication of our initial report [89] from when we were at about a third of the way into working on this dissertation. That is to say that the motivation behind this dissertation was not to challenge the work cited in this subsection, with which it mostly agrees in saying that effects, monads or some other monad-like structure can be used to combine semantic analyses of linguistic phenomena. Nevertheless, we believe that the use of effects and handlers might be interesting to those considering using monad transformers to build a formal semantics. This is especially the case if one is considering to do so in the context of an ACG and wants to therefore use a formal λ-calculus for the semantic calculations. Out of the alternatives mentioned in this subsection, we found the approach of setting monads loose and letting the grammar figure out the semantic glue particularly intriguing as it tries to elegantly sidestep the issue of composing the monads.

9.3 Future Work

Future work on $(\!| \lambda |\!)$ could be either focused on the calculus itself or its linguistic applications. We finish our thesis with a discussion of both.

9.3.1 Future Work on the Calculus

- Adequacy of Denotational Semantics, Observational Equivalence

  In Subsection 3.3.1, we have given a denotational semantics to $(\!| \lambda |\!)$. Even though we first gave the reduction semantics, we perceive the denotational semantics as the primary semantics as it assigns to the terms of $(\!| \lambda |\!)$ the mathematical objects that we want them to stand for. The reduction semantics can then be seen as a mechanization of computing with these objects. In Chapter 3, we have proven Property 3.3.8, which tells us that our reduction semantics is sound w.r.t. the denotations. However, the converse is not the case (i.e. there are terms which the denotational semantics considers equal but which are not convertible using our reduction rules).

  The denotational semantics is useful for reasoning about $(\!| \lambda |\!)$ programs. Strengthening the formal link between the denotational semantics and the reduction semantics would therefore be most welcome. The kind of property we are looking for is known as *adequacy* and it states that if a program and a value have the same denotation, then the program must be able to reduce to that value. For this property to hold, it can be necessary to restrict it only to programs and values of *ground* types (types which do not "hide" any more computation, such as functions or continuations). An example of such a ground type in $(\!| \lambda |\!)$ would be the Boolean type 2. Proving adequacy for terms of type 2 would then yield a result about *observational equivalence*: if two terms M and N have the same denotation, then they are observationally equivalent, i.e. there is no way to write a Boolean expression in $(\!| \lambda |\!)$ whose value differs when switching M for N.

- Commuting Conversions and Extensionality

  During the reduction of one of the examples in 7.3.4, to simplify the presentation, we have made use of a reduction rule for passing $\gg\!=$ (a handler) under case analysis. The formal rule is given below:

$$\begin{aligned}&(\!|\, (\mathtt{op}_i\colon M_i)_{i\in I},\ \eta\colon M_\eta \,|\!)\ (\textbf{case}\ N_\mathrm{c}\ \textbf{of}\ \{\mathrm{inl}\ x \to N_\mathrm{l}(x);\ \mathrm{inr}\ y \to N_\mathrm{r}(y)\})\\ \to\ &\textbf{case}\ N_\mathrm{c}\ \textbf{of}\ \{\mathrm{inl}\ x \to (\!|\, (\mathtt{op}_i\colon M_i)_{i\in I},\ \eta\colon M_\eta \,|\!)\ N_\mathrm{l}(x);\ \mathrm{inr}\ y \to (\!|\, (\mathtt{op}_i\colon M_i)_{i\in I},\ \eta\colon M_\eta \,|\!)\ N_\mathrm{r}(y)\}\end{aligned}$$

We note that both the redex and the contractum have the same denotation. On the other hand, if all the M and N terms are normal, then so are both the redex and contractum. Since they are distinct normal terms, this means that the redex and the contractum are not convertible in $(\!|\, \lambda \,|\!)$. In other words, this rule is not derivable in our system (even though it is licensed by the denotational semantics).

For completeness' sake, we might like to have rules such as these in $(\!|\, \lambda \,|\!)$. They further close the gap between the denotational semantics and the reduction semantics and they can be useful when working with the calculus, as we have seen in the example in 7.3.4. Instead of adding a multitude of *commuting conversions* such as the one for handlers and sums given above, we can also add *extensionality* principles, such as the following η-reduction for sums:

$$\begin{aligned}&\textbf{case}\ M\ \textbf{of}\ \{\mathrm{inl}\ x_1 \to N[x \coloneqq \mathrm{inl}\ x_1];\ \mathrm{inr}\ x_2 \to N[x \coloneqq \mathrm{inr}\ x_2]\}\\ \to\ &N[x \coloneqq M]\end{aligned}$$

The commuting conversions that commute over sums can then be derived from this principle [9]. It might also be interesting to seek what would such an extensionality principle look like for computation types. However, the biggest obstacle is that enriching the calculus with new reduction rules puts in jeopardy the established meta-theory (e.g. confluence or termination, whose proofs are usually brittle).

- Conservativity

  In the summary of the last chapter, Subsection 8.7.1, we have claimed that modifying a grammar by inserting handlers for effects which are not used anywhere in the grammar has no effect on the semantics. Since this claim plays an important part in motivating the use of $(\!|\, \lambda \,|\!)$, it would be desirable to formalize and prove it.[191]

- Algebraic Theories of Effects

  Within the subsections titled "Algebraic Considerations" in Chapters 6 and 7, we studied the equational theories on computations induced by handlers (two computations are considered equal if the handler assigns them the same interpretation). Within these chapters, we looked for general equalities that might become useful and we used those equalities to derive a kind of normal form for the computations. However, the normalization assumes that computations are only sequences of operations terminated by a return value, i.e. they are generated by the grammar below:[192]

  $$S ::= \mathtt{op}\, M\, (\lambda x.\, S) \mid \eta\, M$$

  Is it possible to derive similar normal forms for computations built using a larger grammar (e.g. the full $(\!|\, \lambda \,|\!)$ language)?

  In the examples of Chapter 7, we have been making use of these equalities to simplify terms before passing them to a handler. If this practice becomes common, it might be interesting to formalize it.

  A good start for both of these problems would be a more thorough investigation of Matija Pretnar's dissertation on the logic and handling of algebraic effects [110], which introduces a calculus of effects, effect theories and handlers.

[191] This might be one of the useful applications of adequacy of our denotational semantics. We can prove that $(\!|\, \mathtt{new\_op}\colon M \,|\!)\ N$ has the same denotation as N, provided that `new_op` does not occur in the denotation of N. By using adequacy, we could translate the equality of denotations into a reduction in the reduction semantics.

[192] This grammar only allows for computations which always execute the same sequence of operations, irrespective of any of the their outputs. In Lindley's λ_{flow} calculus [86], such computations would be called *arrow* computations [59].

9.3.2 Future Work on the Linguistic Applications

The plan for future work on the linguistic applications is simple: get more phenomena, more detail, more interactions.

- Low-Hanging Fruit

  Intensional meanings are parameterized by some possible world. The kind of structure introduced in [18] and [40] is yet another instance of the reader monad pattern. As with deixis, meanings will use a `world` : $1 \rightarrowtail \sigma$ operation to access the current world. Modal operators, which will quantify over possible worlds, will then interpret a piece of language while binding the current world to some variable (very much like the way we treated direct quotation, which was binding the speaker to the quotee).

  There are also phenomena for which there are already analyses in terms of monads or (delimited) continuations. An example of this is focus, which can be modelled using the pointed powerset monad [113] or using delimited control operators [11]. While such phenomena ought not to be difficult to implement in $(\!| \lambda |\!)$, it might still take some care and empirical work to ensure that the predictions in cases that involve the new phenomenon and the old phenomena turn out the way they should.

- Better Crossover

  Approaches to semantics that exploit the analogy to computations and treat anaphora and scope-taking as side effects run into trouble when it comes to inverse scope readings and the crossover constraints. In Subsection 8.5.1, we proposed a solution to the problem which decomposes the effect of a quantificational NP into the scope-taking, which can happen out of linear order, and the anaphora, which must happen in linear order. However, there are still cases in which our grammar can violate the crossover constraints, such as when the contributor of a discourse referent (e.g. the indefinite *a dog*) hitches a ride in the restrictor of a scope taker (e.g. *every owner of a dog*), as in Example 35.

  (35) * It$_1$ loves every owner of a dog$_1$.

  Crossover issues with inverse scope readings are an old problem [120] and therefore a solution which improves coverage without incurring extra complexity to the rest of the grammar would be of great interest.

- More Dynamics — Double Negation, Modal Subordination and Rhetorical Structure

  The treatment of dynamics is easily the most complex application of $(\!| \lambda |\!)$. However, anaphora is much more complicated than what we have seen in this dissertation. Extending it might thus prove to be a good stress test on the level of complexity that is still manageable in $(\!| \lambda |\!)$. Directions for extending dynamics include:

  - Double negation, which was studied in Qian's dissertation [111] and which we reviewed in Section 7.4. This is very difficult to apply in our current implementation of dynamics, since negation uses box to encapsulate all of the dynamic effects that take place in its argument. However, to have the law of double negation, we would need to find a way to break this encapsulation, which is tricky since the box handler does encapsulation by throwing away the local structure (i.e. permanently blocking the dynamic contributions).
  - Modal subordination, as in Example 46, is another accessibility constraint that was modelled within an extension of TTDL in Qian's thesis [111]. Furthermore, it is a phenomenon that lies at the intersection of two linguistic effects, modality and anaphora, which makes it all the more interesting for a $(\!| \lambda |\!)$ analysis.

    (46) A wolf$_1$ might walk in. It$_1$ would growl.

- Rhetorical structure informs anaphoric accessibility via the Right Frontier Constraint, as in the famous Example 47 [8].

  (47) * John had a great evening last night. He had a great meal. He ate salmon$_1$. He devoured lots of cheese. He then won a dancing competition. It$_1$ was pink.

It remains to be seen whether the handler abstraction, which turned out to be useful to encapsulate the dynamic effects in negation and modelling DRT, would be also useful in analyses of accessibility constraints due to modal subordination or rhetorical structure.

- Champollion — Sentence Meanings as Generalized Quantifiers over Events

  In [26], Lucas Champollion presents a semantic fragment in which he combines event semantics with classical phenomena of compositional semantics such as quantification or negation. His work therefore seems to be an ideal starting point for anyone wanting to add events into our $(\!| \lambda |\!)$ grammar. However, the types of meanings used within the fragment are quite atypical and therefore it would be interesting to see how many of the theories of the other phenomena developed in this thesis would be compatible with such a semantics.

Bibliography

[1] Continuations — racket reference. `https://docs.racket-lang.org/reference/cont.html`. Accessed: 2016-06-22.

[2] extensible-effects: An alternative to monad transformers. `https://hackage.haskell.org/package/extensible-effects`. Accessed: 2016-09-10.

[3] mtl: Monad classes, using functional dependencies. `http://hackage.haskell.org/package/mtl`. Accessed: 2016-07-22.

[4] Harold Abelson, Gerald Jay Sussman, and Julie Sussman. *Structure and interpretation of computer programs*. Justin Kelly, 1996.

[5] Scott AnderBois, Adrian Brasoveanu, and Robert Henderson. Crossing the appositive/at-issue meaning boundary. In *Semantics and Linguistic Theory*, volume 20, pages 328–346, 2010.

[6] Stevan Andjelkovic. Towards indexed algebraic effects and handlers. The 3rd ACM SIGPLAN Workshop on Higher-Order Programming with Effects, 2014.

[7] Claus Appel, V van Oostrom, and Jakob Grue Simonsen. Higher-order (non-) modularity. *Logic Group Preprint Series*, 284:1–26, 2010.

[8] Nicholas Asher and Alex Lascarides. *Logics of conversation*. Cambridge University Press, 2003.

[9] Vincent Balat, Roberto Di Cosmo, and Marcelo Fiore. Extensional normalisation and type-directed partial evaluation for typed lambda calculus with sums. *ACM SIGPLAN Notices*, 39(1):64–76, 2004.

[10] H.P. Barendregt. *The lambda calculus: its syntax and semantics*. Studies in logic and the foundations of mathematics. North-Holland, 1984.

[11] C. Barker. Continuations in natural language. 2006.

[12] Chris Barker. Continuations and the nature of quantification. *Natural language semantics*, 10(3):211–242, 2002.

[13] Chris Barker and Dylan Bumford. Monads for natural language, 2015.

[14] Chris Barker and Chung-chieh Shan. *Continuations and natural language*, volume 53. Oxford University Press, USA, 2014.

[15] Andrej Bauer. Lambda the ultimate — programming with algebraic effects and handlers. Personal Communication, 2012.

[16] Andrej Bauer and Matija Pretnar. Programming with algebraic effects and handlers. *J. Log. Algebr. Meth. Program.*, 84(1):108–123, 2015.

[17] David I. Beaver and Bart Geurts. Presupposition. In Edward N. Zalta, editor, *The Stanford Encyclopedia of Philosophy*. Winter 2014 edition, 2014.

[18] Gilad Ben-Avi and Yoad Winter. The semantics of intensionalization. *ESSLLI 2007*, page 98, 2007.

[19] Frédéric Blanqui. Termination and confluence of higher-order rewrite systems. In *Rewriting Techniques and Applications*, pages 47–61. Springer, 2000.

[20] Frédéric Blanqui, Jean-Pierre Jouannaud, and Mitsuhiro Okada. Inductive-data-type systems. *Theoretical Computer Science*, 272(1):41–68, 2002.

[21] Chris Blom, Philippe De Groote, Yoad Winter, and Joost Zwarts. Implicit arguments: event modification or option type categories? In *Logic, Language and Meaning*, pages 240–250. Springer, 2012.

[22] Edwin Brady. Programming and reasoning with algebraic effects and dependent types. In *Proceedings of the 18th ACM SIGPLAN international conference on Functional programming*, pages 133–144. ACM, 2013.

[23] Aljoscha Burchardt, Alexander Koller, and Stephan Walter. Computational semantics, 2004.

[24] Robert Cartwright and Matthias Felleisen. Extensible denotational language specifications. In *Theoretical Aspects of Computer Software*, pages 244–272. Springer, 1994.

[25] Lucas Champollion. Back to events: More on the logic of verbal modification. *University of Pennsylvania Working Papers in Linguistics*, 21(1):7, 2015.

[26] Lucas Champollion. The interaction of compositional semantics and event semantics. *Linguistics and Philosophy*, 38(1):31–66, 2015.

[27] Simon Charlow. *On the semantics of exceptional scope*. PhD thesis, New York University, 2014.

[28] Simon Charlow. Conventional implicature as a scope phenomenon. Workshop on continuations and scope, New York University, May 22, 2015.

[29] Simon Charlow. The grammar of exceptional scope. Colloquium, University of Maryland, November 20, 2015.

[30] Simon Charlow. Monadic dynamic semantics for anaphora. Invited talk, Dynamic Semantics: Modern type theoretic and category theoretic approaches, The Ohio State University, October 24, 2015.

[31] Simon Charlow. Monads, applicative functors, and scope. ESSLLI 2015 — Monads for Natural Language, 2015.

[32] Robin Hayes Cooper. Montague's semantic theory and transformational syntax a dissertation. 1979.

[33] Olivier Danvy and Andrzej Filinski. *A functional abstraction of typed contexts*. Univ., 1989.

[34] Nicolaas Govert De Bruijn. Lambda calculus notation with nameless dummies, a tool for automatic formula manipulation, with application to the church-rosser theorem. In *Indagationes Mathematicae (Proceedings)*, volume 75, pages 381–392. Elsevier, 1972.

[35] Philippe de Groote. Towards abstract categorial grammars. In *Proceedings of the 39th Annual Meeting on Association for Computational Linguistics*, pages 252–259. Association for Computational Linguistics, 2001.

[36] Philippe de Groote. Type raising, continuations, and classical logic. In *Proceedings of the thirteenth Amsterdam Colloquium*, 2001.

[37] Philippe de Groote. Tree-adjoining grammars as abstract categorial grammars. In *TAG+6, Proceedings of the sixth International Workshop on Tree Adjoining Grammars and Related Frameworks*, pages 145–150. Università di Venezia, 2002.

[38] Philippe de Groote. Towards a montagovian account of dynamics. In *Proceedings of SALT*, volume 16, 2006.

[39] Philippe de Groote. On logical relations and conservativity. 2015.

[40] Philippe de Groote and Makoto Kanazawa. A note on intensionalization. *Journal of Logic, Language and Information*, 22(2):173–194, 2013.

[41] Stephen Dolan, Leo White, K Sivaramakrishnan, Jeremy Yallop, and Anil Madhavapeddy. Effective concurrency through algebraic effects. In *OCaml Users and Developers Workshop*, 2015.

[42] Matthias Felleisen, Robert Bruce Findler, and Matthew Flatt. *Semantics engineering with PLT Redex*. Mit Press, 2009.

[43] Matthias Felleisen, Mitch Wand, Daniel Friedman, and Bruce Duba. Abstract continuations: a mathematical semantics for handling full jumps. In *Proceedings of the 1988 ACM conference on LISP and functional programming*, pages 52–62. ACM, 1988.

[44] Andrzej Filinski. Representing monads. In *Proceedings of the 21st ACM SIGPLAN-SIGACT symposium on Principles of programming languages*, pages 446–457. ACM, 1994.

[45] Marcelo P Fiore, Gordon Plotkin, and Daniele Turi. Abstract syntax and variable binding. 2003.

[46] Bart Geurts and Emar Maier. Layered drt. *Ms., University of Nijmegen*, 2003.

[47] Bart Geurts and Emar Maier. Quotation in context. *Belgian Journal of Linguistics*, 17(1):109–128, 2003.

[48] Gianluca Giorgolo and Ash Asudeh. Monads for conventional implicatures. In *Proceedings of sinn und bedeutung*, volume 16, pages 265–278, 2012.

[49] Gianluca Giorgolo and Ash Asudeh. Monads as a solution for generalized opacity. *EACL 2014*, page 19, 2014.

[50] Gianluca Giorgolo and Ash Asudeh. Natural language semantics with enriched meanings, 2015.

[51] Gianluca Giorgolo, Ash Asudeh, Miriam Butt, and Tracy Holloway King. Multidimensional semantics with unidimensional glue logic. *Proceedings of LFG11*, pages 236–256, 2011.

[52] Gianluca Giorgolo and Christina Unger. Coreference without discourse referents: a non-representational drt-like discourse semantics. *LOT Occasional Series*, 14:69–81, 2009.

[53] Jeroen Groenendijk and Martin Stokhof. Dynamic predicate logic. *Linguistics and philosophy*, 14(1):39–100, 1991.

[54] Daniel Gutzmann. *Use-conditional meaning: Studies in multidimensional semantics*. OUP Oxford, 2015.

[55] Makoto Hamana. Higher-order semantic labelling for inductive datatype systems. In *Proceedings of the 9th ACM SIGPLAN international conference on Principles and practice of declarative programming*, pages 97–108. ACM, 2007.

[56] Daniel Hillerström, Sam Lindley, and K Sivaramakrishnan. Compiling links effect handlers to the ocaml backend. In *ML Workshop*, 2016.

[57] J.R. Hobbs and S.J. Rosenschein. Making computational sense of montague's intensional logic. *Artificial Intelligence*, 9(3):287–306, 1977.

[58] Wilfrid Hodges. Tarski's truth definitions. In Edward N. Zalta, editor, *The Stanford Encyclopedia of Philosophy*. Fall 2014 edition, 2014.

[59] John Hughes. Generalising monads to arrows. *Science of computer programming*, 37(1):67–111, 2000.

[60] Martin Hyland, Gordon Plotkin, and John Power. Combining effects: Sum and tensor. *Theoretical Computer Science*, 357(1):70–99, 2006.

[61] Mark P Jones. Functional programming with overloading and higher-order polymorphism. In *International School on Advanced Functional Programming*, pages 97–136. Springer, 1995.

[62] Simon L Peyton Jones. *Haskell 98 language and libraries: the revised report*. Cambridge University Press, 2003.

[63] Ohad Kammar, Sam Lindley, and Nicolas Oury. Handlers in action. In *Proceedings of the 18th ACM SIGPLAN international conference on Functional programming*, pages 145–158. ACM, 2013.

[64] Hans Kamp and Uwe Reyle. *From discourse to logic: Introduction to modeltheoretic semantics of natural language, formal logic and discourse representation theory*. Number 42. Kluwer Academic Pub, 1993.

[65] Makoto Kanazawa. Parsing and generation as datalog queries. In *Annual Meeting-Association for Computational Linguistics*, volume 45, page 176, 2007.

[66] William R Keller. Nested cooper storage: The proper treatment of quantification in ordinary noun phrases. In *Natural language parsing and linguistic theories*, pages 432–447. Springer, 1988.

[67] Oleg Kiselyov. Call-by-name linguistic side effects. In *ESSLLI 2008 Workshop on Symmetric calculi and Ludics for the semantic interpretation*, 2008.

[68] Oleg Kiselyov. Applicative abstract categorial grammars. In *Proceedings of the Third Workshop on Natural Language and Computer Science*, 2015.

[69] Oleg Kiselyov. Applicative abstract categorial grammars in full swing. In *Proceedings of the Twelth Workshop on Logic and Engineering of Natural Language Semantics*, 2015.

[70] Oleg Kiselyov and Hiromi Ishii. Freer monads, more extensible effects. In *ACM SIGPLAN Notices*, volume 50, pages 94–105. ACM, 2015.

[71] Oleg Kiselyov, Amr Sabry, and Cameron Swords. Extensible effects: an alternative to monad transformers. In *Proceedings of the 2013 ACM SIGPLAN symposium on Haskell*, pages 59–70. ACM, 2013.

[72] Oleg Kiselyov and Chung-chieh Shan. Lambda: the ultimate syntax-semantics interface, 2010.

[73] Oleg Kiselyov and KC Sivaramakrishnan. Eff directly in ocaml. In *ML Workshop*, 2016.

[74] Stephen Cole Kleene. Introduction to metamathematics. 1952.

[75] Jan Willem Klop et al. Term rewriting systems. *Handbook of logic in computer science*, 2:1–116, 1992.

[76] Jan Willem Klop, Vincent Van Oostrom, and Femke Van Raamsdonk. Combinatory reduction systems: introduction and survey. *Theoretical computer science*, 121(1):279–308, 1993.

[77] Bronisław Knaster and A Tarski. Un théoreme sur les fonctions d'ensembles. *Ann. Soc. Polon. Math*, 6(133):2013134, 1928.

[78] Emiel Krahmer and Reinhard Muskens. Negation and disjunction in discourse representation theory. *Journal of Semantics*, 12(4):357–376, 1995.

[79] S Kuschert, M Kohlhase, and M Pinkal. A type-theoretic semantics for lambda-drt. In *Proceedings of the Tenth Amsterdam Colloquium*, 1995.

[80] Ekaterina Lebedeva. *Expression de la dynamique du discours à l'aide de continuations*. PhD thesis, Université de Lorraine, 2012.

[81] Stephen C Levinson. Deixis. In *The handbook of pragmatics*, pages 97–121. Blackwell, 2004.

[82] Paul Blain Levy. Call-by-push-value: A subsuming paradigm. In *Typed Lambda Calculi and Applications*, pages 228–243. Springer, 1999.

[83] David Lewis. General semantics. *Synthese*, 22(1):18–67, 1970.

[84] Sheng Liang, Paul Hudak, and Mark Jones. Monad transformers and modular interpreters. In *Proceedings of the 22nd ACM SIGPLAN-SIGACT symposium on Principles of programming languages*, pages 333–343. ACM, 1995.

[85] Mark Lillibridge. Exceptions are strictly more powerful than call/cc. Technical report, DTIC Document, 1995.

[86] Sam Lindley. Algebraic effects and effect handlers for idioms and arrows. In *Proceedings of the 10th ACM SIGPLAN workshop on Generic programming*, pages 47–58. ACM, 2014.

[87] Sam Lindley, Conor McBride, and Craig McLaughlin. Do be do be do. draft available at `http://homepages.inf.ed.ac.uk/slindley/papers/frankly-draft-july2016.pdf`, 2016.

[88] Saunders Mac Lane. *Categories for the working mathematician*, volume 5. Springer Science & Business Media, 1978.

[89] Jiří Maršík and Maxime Amblard. Algebraic effects and handlers in natural language interpretation. In *Natural Language and Computer Science*. Center for Informatics and Systems of the University of Coimbra, 2014.

[90] Scott Martin and Carl Pollard. A dynamic categorial grammar. In *International Conference on Formal Grammar*, pages 138–154. Springer, 2014.

[91] Conor McBride and Ross Paterson. Applicative programming with effects. *Journal of functional programming*, 18(01):1–13, 2008.

[92] John McCarthy. A basis for a mathematical theory of computation, preliminary report. In *Papers presented at the May 9-11, 1961, western joint IRE-AIEE-ACM computer conference*, pages 225–238. ACM, 1961.

[93] Erik Meijer, Maarten Fokkinga, and Ross Paterson. Functional programming with bananas, lenses, envelopes and barbed wire. In *Functional Programming Languages and Computer Architecture*, pages 124–144. Springer, 1991.

[94] Eugenio Moggi. *An abstract view of programming languages*. University of Edinburgh, Department of Computer Science, Laboratory for Foundations of Computer Science, 1990.

[95] Eugenio Moggi. Notions of computation and monads. *Information and computation*, 93(1):55–92, 1991.

[96] R. Montague. English as a formal language. *Linguaggi nella societae nella tecnica*, pages 189–224, 1970.

[97] R. Montague. Universal grammar. *Theoria*, 36(3):373–398, 1970.

[98] R. Montague. The proper treatment of quantification in ordinary english. 1973.

[99] Reinhard Muskens. Combining montague semantics and discourse representation. *Linguistics and philosophy*, 19(2):143–186, 1996.

[100] Reinhard Muskens. Lambda grammars and the syntax-semantics interface. In *Proceedings of the Thirteenth Amsterdam Colloquium*, pages 150–155, 2001.

[101] Rick Nouwen, Adrian Brasoveanu, Jan van Eijck, and Albert Visser. Dynamic semantics. In Edward N. Zalta, editor, *The Stanford Encyclopedia of Philosophy*. Fall 2016 edition, 2016.

[102] Geoffrey Nunberg. *The linguistics of punctuation*. Number 18. Center for the Study of Language (CSLI), 1990.

[103] Gordon Plotkin and Matija Pretnar. Handlers of algebraic effects. In *Programming Languages and Systems*, pages 80–94. Springer, 2009.

[104] Gordon D Plotkin and Matija Pretnar. Handling algebraic effects. *arXiv preprint arXiv:1312.1399*, 2013.

[105] Sylvain Pogodalla. Generalizing a proof-theoretic account of scope ambiguity. In *7th International Workshop on Computational Semantics*, 2007.

[106] Sylvain Pogodalla and Florent Pompigne. Controlling extraction in abstract categorial grammars. In *Formal Grammar*, 2012.

[107] Carl Pollard. Convergent grammar, 2008.

[108] Christopher Potts. *The logic of conventional implicatures*. Oxford University Press Oxford, 2005.

[109] Christopher Potts. The dimensions of quotation. *Direct compositionality*, pages 405–431, 2007.

[110] Matija Pretnar. Logic and handling of algebraic effects. 2010.

[111] Sai Qian. *Accessibility of Referents in Discourse Semantics*. PhD thesis, Université de Lorraine, 2014.

[112] Craige Roberts. Modal subordination and pronominal anaphora in discourse. *Linguistics and philosophy*, 12(6):683–721, 1989.

[113] Chung-chieh Shan. Monads for natural language semantics. *arXiv preprint cs/0205026*, 2002.

[114] Chung-chieh Shan. Delimited continuations in natural language: Quantification and polarity sensitivity. *arXiv preprint cs/0404006*, 2004.

[115] Chung-chieh Shan. Linguistic side effects. In *In Proceedings of the Eighteenth Annual IEEE Symposium on Logic and Computer Science (LICS 2003) Workshop on Logic and Computational*, pages 132–163. University Press, 2005.

[116] Chung-chieh Shan. *Linguistic side effects*. PhD thesis, Harvard University Cambridge, Massachusetts, 2005.

[117] Chung-chieh Shan. Inverse scope as metalinguistic quotation in operational semantics. In *Annual Conference of the Japanese Society for Artificial Intelligence*, pages 123–134. Springer, 2007.

[118] Chung-chieh Shan. A static simulation of dynamic delimited control. *Higher-Order and Symbolic Computation*, 20(4):371–401, 2007.

[119] Chung-chieh Shan. The character of quotation. *Linguistics and Philosophy*, 33(5):417–443, 2010.

[120] Chung-chieh Shan and Chris Barker. Explaining crossover and superiority as left-to-right evaluation. *Linguistics and Philosophy*, 29(1):91–134, 2006.

[121] Dorai Sitaram. Handling control. In *ACM SIGPLAN Notices*, volume 28, pages 147–155. ACM, 1993.

[122] Dorai Sitaram. Teach yourself scheme in fixnum days. *Voir http://www. ccs. neu. edu/home/dorai/ty-scheme/ty-scheme. html*, 1998.

[123] Wouter Swierstra. Data types à la carte. *Journal of functional programming*, 18(04):423–436, 2008.

[124] Zoltán Gendler Szabó. Compositionality. In Edward N. Zalta, editor, *The Stanford Encyclopedia of Philosophy*. Fall 2013 edition, 2013.

[125] William W Tait. Intensional interpretations of functionals of finite type i. *The journal of symbolic logic*, 32(02):198–212, 1967.

[126] Alfred Tarski et al. A lattice-theoretical fixpoint theorem and its applications. *Pacific journal of Mathematics*, 5(2):285–309, 1955.

[127] Alfred Tarski and R Vaught. Arithmetical extensions of relational systems', compositio mathematicae 13, 81-102. *Reprinted in A Tarski*, pages 31945–1957, 1986.

[128] Christina Unger. Dynamic semantics as monadic computation. In *JSAI International Symposium on Artificial Intelligence*, pages 68–81. Springer, 2011.

[129] Christina Unger. Dynamic semantics as monadic computation. In *Proceedings of the 8th International Workshop on Logic and Engineering of Natural Language Semantics (LENLS 8)*, 2012.

[130] Rob A Van der Sandt. Presupposition projection as anaphora resolution. *Journal of semantics*, 9(4):333–377, 1992.

[131] Noortje J Venhuizen, Johan Bos, and Harm Brouwer. Parsimonious semantic representations with projection pointers. In *Proceedings of the 10th International Conference on Computational Semantics (IWCS 2013)–Long Papers*, pages 252–263, 2013.

[132] Philip Wadler. Recursive types for free, 1990.

[133] Philip Wadler. The essence of functional programming. In *Proceedings of the 19th ACM SIGPLAN-SIGACT symposium on Principles of programming languages*, pages 1–14. ACM, 1992.

[134] Nicolas Wu. Transformers, handlers in disguise. Haskell eXchange 2015, recording at `https://skillsmatter.com/skillscasts/6733-transformers-handlers-in-disguise`, 2015.

A

List of Examples

(1) She might still be dating that idiot. 1

(2) Every man is mortal. Socrates is a man. 105

(3) Socrates is mortal. 105

(4) It fascinates him. 117

(5) Jones$_1$ owns Ulysses$_2$. It$_2$ fascinates him$_1$. 117

(6) Jones$_1$ owns a Porsche$_2$. It$_2$ fascinates him$_1$. 117

(7) Every farmer who owns a donkey$_1$ beats it$_1$. 117

(8) If a farmer$_1$ owns a donkey$_2$, he$_1$ beats it$_2$. 117

(9) * Every farmer$_1$ who owns a donkey$_2$ beats it$_2$. He$_1$ is cruel. 121

(10) John loves Mary. 132

(11) John loves me. 132

(12) John said Mary loves me. 132

(13) John said "Mary loves me". 133

(14) Either John loves Sarah, or Mary, John's best friend, loves John. 135

(15) If it is not the case that John, whom Sarah loves, loves Sarah then Mary loves John. 135

(16) Every man loves a woman. 138

(17) Every owner of a siamese cat loves a therapist. 139

(18) John loves a man. 142

(19) A man owns a Porsche. It fascinates him. 162

(20) It is not the case that Jones$_1$ owns a Porsche. He$_1$ owns a Mercedes. 170

(21) It is not the case that John$_1$ likes his$_1$ car. 173

(22) If John owns a car, then his car is cheap. 176

(23) If there is a poor man called John, then if John owns a car, his car is cheap. 181

(24) (c_0) Maybe (c_1) Wilma thinks that (c_2) her husband is having an affair. 181

(25) Most Germans wash their car on Saturday. 182

(26) (c_0) If (c_1) a man gets angry, (c_2) his children get frightened. 183

(27) * He$_1$ loves John's$_1$ car. 193

(28) Either there's no bathroom$_1$ in the house, or it's$_1$ in a funny place. 194

(29) John, who nearly killed a woman$_1$ with his car, visited her$_1$ in the hospital. 209

(30) The porridge is warm. 212

(31) * His$_1$ mother likes every man$_1$. 215

(32) A woman loves every man. 217

(33) * He$_1$ likes every man$_1$. 219

(34) * He$_1$ likes every man$_1$'s mother. 219

(35) * It$_1$ loves every owner of a dog$_1$. 219

(36) The man who owns a dog$_1$ loves it$_1$. 223

(37) The man$_1$ who loves his$_1$ dog$_2$ treats it$_2$ well. 224

(38) The owner of every book ever published by Elsevier must be very rich. 224

(39) Mary sent an email to the representative of every country. 224

(40) Everyone$_1$ hates the sound of their$_1$ voice. 224

(41) John met the woman who stole a car$_1$ by hacking into its$_1$ computer. 224

(42) My best friend, who owns a dog$_1$, said it$_1$ loves everyone. 227

(43) A man$_1$ walks in the park. He$_1$ whistles. 233

(44) Peter said "Mary kissed me". 236

(45) Peter said "I was kissed by Mary". 236

(46) A wolf$_1$ might walk in. It$_1$ would growl. 242

(47) * John had a great evening last night. He had a great meal. He ate salmon$_1$. He devoured lots of cheese. He then won a dancing competition. It$_1$ was pink. 243

B

Example from the Final Fragment

We dedicate this chapter to the following sentence:

(42) My best friend, who owns a dog$_1$, said it$_1$ loves everyone.

This sentence features all of the aspects of language that we covered in Part II:

- we have the deictic first-person pronoun *my*
- we have the presupposition triggering noun phrase *my best friend*
- we have the appositive relative clause *who owns a dog*
- we have the anaphoric pronoun *it*, which is bound by an indefinite inside an appositive
- we have the quantificational noun phrase *everyone*

As with the examples in Chapter 7, we will proceed by incrementally building up the meanings of the parts of the sentence. We start with the clause *it loves everyone*. First, we introduce the noun phrase $\textsc{everyone} : NP$, which can be seen as a special case of the $\textsc{every} : N \multimap NP$ determiner that already exists in our grammar.

$$\textsc{everyone} : NP$$
$$[\![\textsc{everyone}]\!] = \texttt{scope}\ \overline{\forall}\,(\lambda x.\, \eta\,(\mathsf{trace}\, x))$$

Now, we move to the clause. The entry for the transitive verb *loves* tells us to evaluate the subject and the object, to assert that the relation **love** holds for the two, and to handle any `scope` operations using SI to enforce the "tensed clauses are scope islands" constraint.

$$\begin{aligned}
&[\![\textsc{loves}\ (\textsc{in-situ}\ \textsc{everyone})\ \textsc{it}]\!]\\
&\twoheadrightarrow \mathsf{SI}\,(\,\texttt{get} \star (\lambda e.\\
&\qquad \texttt{scope}\ \overline{\forall}\,(\lambda z.\\
&\qquad \texttt{push}\, z\,(\lambda_.\\
&\qquad \texttt{assert}\,(\mathbf{love}\,(\mathsf{sel}_{\mathsf{it}}\, e)\, z)\,(\lambda_.\\
&\qquad \eta \star)))))\\
&\twoheadrightarrow \texttt{get} \star (\lambda e.\\
&\quad \overline{\forall}\,(\lambda z.\, \texttt{push}\, z\,(\lambda_.\, \texttt{assert!}\,(\mathbf{love}\,(\mathsf{sel}_{\mathsf{it}}\, e)\, z))))\\
&\twoheadrightarrow \texttt{get} \star (\lambda e.\\
&\quad \texttt{assert}\,(\forall z.\, \mathbf{love}\,(\mathsf{sel}_{\mathsf{it}}\, e)\, z)\,(\lambda_.\\
&\quad \eta \star))
\end{aligned}$$

We will now give a meaning to the verb phrase *said it loves everyone*. The verb *said* evaluates its complementary clause down to a proposition using the box handler. We will therefore first calculate the value of applying box to the meaning of *it loves everyone*.

$$
\begin{aligned}
&\mathsf{box}\,[\![\textsc{loves}\,(\textsc{in-situ everyone})\,\textsc{it}]\!]\\
&\twoheadrightarrow \mathsf{box}\,(\,\mathtt{get}\star(\lambda e.\\
&\qquad\qquad \mathtt{assert}\,(\forall z.\,\mathbf{love}\,(\mathsf{sel}_{\mathsf{it}}\,e)\,z)\,(\lambda\_.\\
&\qquad\qquad \eta\star)))\,)\\
&\twoheadrightarrow \mathtt{get}\star(\lambda e.\\
&\qquad \eta\,(\forall z.\,\mathbf{love}\,(\mathsf{sel}_{\mathsf{it}}\,e)\,z))
\end{aligned}
$$

We can now plug this result into the lexical entry $\textsc{said}_{\textsc{is}}$.

$$
\begin{aligned}
&[\![\textsc{said}_{\textsc{is}}\,(\textsc{loves}\,(\textsc{in-situ everyone})\,\textsc{it})]\!]\\
&\twoheadrightarrow \lambda X.\,\mathsf{SI}\,(\,X \gg\!= (\lambda x.\\
&\qquad\qquad \mathtt{get}\star(\lambda e.\\
&\qquad\qquad \mathtt{assert}\,(\mathbf{say}\,x\,(\forall z.\,\mathbf{love}\,(\mathsf{sel}_{\mathsf{it}}\,e)\,z))\,(\lambda\_.\\
&\qquad\qquad \eta\star)))\,)
\end{aligned}
$$

Now we move to the subject of the sentence. The meaning of the first-person pronoun *me* is obtained by asking the context for the identity of the speaker and the meaning of the possessive construction *X's best friend* is obtained by asking the context for an individual that is X's best friend. To obtain the meaning of the noun phrase *my best friend*, it suffices to chain/concatenate the two computations.

$$
\begin{aligned}
&[\![\textsc{best-friend me}]\!]\\
&\twoheadrightarrow \mathtt{speaker}\star(\lambda s.\\
&\quad \mathtt{presuppose}\,(\lambda x.\,\mathtt{assert!}\,(\mathbf{best\text{-}friend}\,x\,s))\,(\lambda x.\\
&\quad \eta\,x))
\end{aligned}
$$

Next, we compute the meaning of the verb phrase *owns a dog* that occurs in the relative clause. The indefinite *a dog* introduces a discourse referent which it claims to be a dog.

$$
\begin{aligned}
&[\![\textsc{owns}\,(\textsc{a dog})]\!]\\
&\twoheadrightarrow \lambda X.\,\mathsf{SI}\,(\,X \gg\!= (\lambda x.\\
&\qquad\qquad \mathtt{introduce}\star(\lambda y.\\
&\qquad\qquad \mathtt{assert}\,(\mathbf{dog}\,y)\,(\lambda\_.\\
&\qquad\qquad \mathtt{assert}\,(\mathbf{own}\,x\,y)\,(\lambda\_.\\
&\qquad\qquad \eta\star))))\,)
\end{aligned}
$$

According to the lexical entry for $\textsc{who}_{\textsc{s}}$, the verb phrase inside an appositive relative clause is applied to a variable x and passed through the asImplicature, which moves the contents of the relative clause from the at-issue layer to the layer of conventional implicatures.

$$
\begin{aligned}
&\mathsf{asImplicature}\,[\![\textsc{owns}\,(\textsc{a dog})\,(\eta\,x)]\!]\\
&\twoheadrightarrow \mathsf{asImplicature}\,(\,\mathtt{introduce}\,\star\,(\lambda y.\\
&\qquad\qquad\qquad \mathtt{assert}\,(\mathbf{dog}\,y)\,(\lambda\_.\\
&\qquad\qquad\qquad \mathtt{assert}\,(\mathbf{own}\,x\,y)\,(\lambda\_.\\
&\qquad\qquad\qquad \eta\,\star)))\,)\\
&\twoheadrightarrow \mathtt{introduce}^{\mathtt{i}}\,\star\,(\lambda y.\\
&\quad \mathtt{implicate}\,(\mathbf{dog}\,y)\,(\lambda\_.\\
&\quad \mathtt{implicate}\,(\mathbf{own}\,x\,y)\,(\lambda\_.\\
&\quad \eta\,\star)))
\end{aligned}
$$

To get the meaning of the noun phrase *my best friend* as modified by the appositive relative clause *who owns a dog*, we chain their computations.

$$
\begin{aligned}
&[\![\textsc{who}_{\mathrm{s}}\,(\textsc{owns}\,(\textsc{a dog}))\,(\textsc{best-friend me})]\!]\\
&\twoheadrightarrow \mathtt{speaker}\,\star\,(\lambda s.\\
&\quad \mathtt{presuppose}\,(\lambda x.\,\mathtt{assert!}\,(\mathbf{best\text{-}friend}\,x\,s))\,(\lambda x.\\
&\quad \mathtt{introduce}^{\mathtt{i}}\,\star\,(\lambda y.\\
&\quad \mathtt{implicate}\,(\mathbf{dog}\,y)\,(\lambda\_.\\
&\quad \mathtt{implicate}\,(\mathbf{own}\,x\,y)\,(\lambda\_.\\
&\quad \eta\,x)))))
\end{aligned}
$$

Finally, to compute the meaning of the whole sentence of Example 42, we chain the computation of the subject with the computation of the verb phrase.

$$
\begin{aligned}
&[\![\textsc{said}_{\mathrm{IS}}\,(\textsc{loves}\,(\textsc{in-situ everyone})\,\textsc{it})\,(\textsc{who}_{\mathrm{s}}\,(\textsc{owns}\,(\textsc{a dog}))\,(\textsc{best-friend me}))]\!]\\
&\twoheadrightarrow \mathtt{speaker}\,\star\,(\lambda s.\\
&\quad \mathtt{presuppose}\,(\lambda x.\,\mathtt{assert!}\,(\mathbf{best\text{-}friend}\,x\,s))\,(\lambda x.\\
&\quad \mathtt{introduce}^{\mathtt{i}}\,\star\,(\lambda y.\\
&\quad \mathtt{implicate}\,(\mathbf{dog}\,y)\,(\lambda\_.\\
&\quad \mathtt{implicate}\,(\mathbf{own}\,x\,y)\,(\lambda\_.\\
&\quad \mathtt{get}\,\star\,(\lambda e.\\
&\quad \mathtt{assert}\,(\mathbf{say}\,x\,(\forall z.\,\mathbf{love}\,(\mathsf{sel}_{\mathsf{it}}\,e)\,z))\,(\lambda\_.\\
&\quad \eta\,\star)))))))
\end{aligned}
$$

We now have all the instructions that tell us how to find the truth conditions of the sentence within any context. We can use the handler $\mathsf{top}\,s$ to get the meaning of the sentence within an "empty" context with the speaker s. The top handler is defined as a composition of all the partial handlers for the different aspects of our fragment. We will apply the handlers to the meaning of Example 42 one by one.

$$
\begin{aligned}
&\mathsf{top} : \iota \to \mathcal{F}_E(1) \to \mathcal{F}_\emptyset(o)\\
&\mathsf{top} = \lambda s.\,\mathsf{search} \circ \mathsf{empty} \circ \mathsf{box} \circ \mathsf{accommodate} \circ \mathsf{useFind} \circ \mathsf{withImplicatures} \circ \mathsf{withSpeaker}\,s \circ \mathsf{SI}
\end{aligned}
$$

We label the meaning of Example 42 as t_0 and the successive applications of the handlers as t_1, t_2...

$$
t_0 = [\![\textsc{said}_{\mathrm{IS}}\,(\textsc{loves}\,(\textsc{in-situ everyone})\,\textsc{it})\,(\textsc{who}_{\mathrm{s}}\,(\textsc{owns}\,(\textsc{a dog}))\,(\textsc{best-friend me}))]\!]
$$

We start with the SI handler. Since the computation t_0 uses no `scope` operation, the SI handler will have no effect.

$$t_1 = \mathsf{SI}\, t_0 \twoheadrightarrow t_0$$

Next up is the withSpeaker s handler. This will resolve the `speaker` operation by replacing the speaker variable s by the hypothetical speaker s, i.e. the rest of the computation stays the same.

$$\begin{aligned} t_2 = \mathsf{withSpeaker}\, s\, t_1 \twoheadrightarrow\ & \texttt{presuppose}\,(\lambda x.\, \texttt{assert!}\,(\textbf{best-friend}\, x\, s))\,(\lambda x. \\ & \texttt{introduce}^{\texttt{i}}\ \star\ (\lambda y. \\ & \texttt{implicate}\,(\textbf{dog}\, y)\,(\lambda\_. \\ & \texttt{implicate}\,(\textbf{own}\, x\, y)\,(\lambda\_. \\ & \texttt{get}\ \star\ (\lambda e. \\ & \texttt{assert}\,(\textbf{say}\, x\, (\forall z.\, \textbf{love}\,(\mathsf{sel}_{\mathsf{it}}\, e)\, z))\,(\lambda\_. \\ & \eta\, \star)))))) \end{aligned}$$

Then we have the withImplicatures handler that will accommodate the conventional implicatures of the sentence as part of its truth conditions.

$$\begin{aligned} t_3 = \mathsf{withImplicatures}\, t_2 \twoheadrightarrow\ & \texttt{presuppose}\,(\lambda x.\, \texttt{assert!}\,(\textbf{best-friend}\, x\, s))\,(\lambda x. \\ & \texttt{introduce}\ \star\ (\lambda y. \\ & \texttt{assert}\,(\textbf{dog}\, y)\,(\lambda\_. \\ & \texttt{assert}\,(\textbf{own}\, x\, y)\,(\lambda\_. \\ & \texttt{get}\ \star\ (\lambda e. \\ & \texttt{assert}\,(\textbf{say}\, x\, (\forall z.\, \textbf{love}\,(\mathsf{sel}_{\mathsf{it}}\, e)\, z))\,(\lambda\_. \\ & \eta\, \star)))))) \end{aligned}$$

Next is the useFind handler that will try to look up a referent for the speaker's best friend within the context.

$$\begin{aligned} t_4 = \mathsf{useFind}\, t_3 \twoheadrightarrow\ & \texttt{get}\ \star\ (\lambda e. \\ & (\textbf{case}\ \mathsf{sel}_{\mathsf{P}}\,(\lambda x.\, \textbf{best-friend}\, x\, s)\, e\ \textbf{of} \\ & \quad \{\mathrm{inl}\ x \to \eta\, x; \\ & \quad\ \ \mathrm{inr}\ \_ \to \texttt{presuppose!}\,(\lambda x.\, \texttt{assert!}\,(\textbf{best-friend}\, x\, s))\}) \gg\!= (\lambda x. \\ & \texttt{introduce}\ \star\ (\lambda y. \\ & \texttt{assert}\,(\textbf{dog}\, y)\,(\lambda\_. \\ & \texttt{asssert}\,(\textbf{own}\, x\, y)\,(\lambda\_. \\ & \texttt{get}\ \star\ (\lambda e. \\ & \texttt{assert}\,(\textbf{say}\, x\, (\forall z.\, \textbf{love}\,(\mathsf{sel}_{\mathsf{it}}\, e)\, z))\,(\lambda\_. \\ & \eta\, \star))))))) \end{aligned}$$

We know ahead of time that the context e in which the sentence will be evaluated will not contain the speaker's best friend and so we allow ourselves to reduce $\mathsf{sel}_{\mathsf{P}}\,(\lambda x.\, \textbf{best-friend}\, x\, s)\, e$ to inr $\star$.

$$
\begin{aligned}
t_4 \twoheadrightarrow\ & \texttt{get} \star (\lambda e. \\
& \texttt{presuppose}\,(\lambda x.\,\texttt{assert!}\,(\textbf{best-friend}\,x\,s))\,(\lambda x. \\
& \texttt{introduce} \star (\lambda y. \\
& \texttt{assert}\,(\textbf{dog}\,y)\,(\lambda\_. \\
& \texttt{assert}\,(\textbf{own}\,x\,y)\,(\lambda\_. \\
& \texttt{get} \star (\lambda e. \\
& \texttt{assert}\,(\textbf{say}\,x\,(\forall z.\,\textbf{love}\,(\mathsf{sel}_{\mathsf{it}}\,e)\,z))\,(\lambda\_. \\
& \eta\,\star)))))))
\end{aligned}
$$

The presupposition is then accommodated by the accommodate handler.

$$
\begin{aligned}
t_5 = \mathsf{accommodate}\,t_4 \twoheadrightarrow\ & \texttt{get} \star (\lambda e. \\
& \texttt{introduce} \star (\lambda x. \\
& \texttt{assert}\,(\textbf{best-friend}\,x\,s)\,(\lambda\_. \\
& \texttt{introduce} \star (\lambda y. \\
& \texttt{assert}\,(\textbf{dog}\,y)\,(\lambda\_. \\
& \texttt{assert}\,(\textbf{own}\,x\,y)\,(\lambda\_. \\
& \texttt{get} \star (\lambda e. \\
& \texttt{assert}\,(\textbf{say}\,x\,(\forall z.\,\textbf{love}\,(\mathsf{sel}_{\mathsf{it}}\,e)\,z))\,(\lambda\_. \\
& \eta\,\star)))))))))
\end{aligned}
$$

Next up is the box handler which takes care of the sentence dynamics.

$$
\begin{aligned}
t_6 = \mathsf{box}\,t_5 \twoheadrightarrow\ & \texttt{get} \star (\lambda e. \\
& \texttt{get} \star (\lambda e. \\
& \eta\,(\exists x.\,\textbf{best-friend}\,x\,s \wedge (\exists y.\,\textbf{dog}\,y \wedge \textbf{own}\,x\,y \wedge \textbf{say}\,x\,(\forall z.\,\textbf{love}\,(\mathsf{sel}_{\mathsf{it}}\,e')\,z)))))
\end{aligned}
$$

where

$$
e' = ((\textbf{own}\,x\,y) :: (\textbf{dog}\,y) :: y :: (\textbf{best-friend}\,x\,s) :: x :: \mathsf{nil}) +\!\!+ e
$$

The empty handler, which comes next, will identify the (anaphoric) context e in which the sentence is being evaluated as the empty context nil.

$$
t_7 = \mathsf{empty}\,t_6 \twoheadrightarrow \eta\,(\exists x.\,\textbf{best-friend}\,x\,s \wedge (\exists y.\,\textbf{dog}\,y \wedge \textbf{own}\,x\,y \wedge \textbf{say}\,x\,(\forall z.\,\textbf{love}\,(\mathsf{sel}_{\mathsf{it}}\,e'')\,z)))
$$

where

$$
\begin{aligned}
e'' &= ((\textbf{own}\,x\,y) :: (\textbf{dog}\,y) :: y :: (\textbf{best-friend}\,x\,s) :: x :: \mathsf{nil}) +\!\!+ \mathsf{nil} \\
&= (\textbf{own}\,x\,y) :: (\textbf{dog}\,y) :: y :: (\textbf{best-friend}\,x\,s) :: x :: \mathsf{nil}
\end{aligned}
$$

The context e'' contains only two entities: the speaker's best friend x and his dog y. Assuming that anaphora resolution will choose the dog y as the referent of the pronoun *it*, we can reduce $\mathsf{sel}_{\mathsf{it}}\,e''$ to y.

$$
\begin{aligned}
t_7 = \mathsf{top}\,s\,[\![\textsc{said}_{\textsc{is}}\,(\textsc{loves}\,(\textsc{in-situ everyone})\,\textsc{it})\,(\textsc{who}_{\textsc{s}}\,(\textsc{owns}\,(\textsc{a dog}))\,(\textsc{best-friend me}))]\!] \\
\twoheadrightarrow \eta\,(\exists x.\,\textbf{best-friend}\,x\,s \wedge (\exists y.\,\textbf{dog}\,y \wedge \textbf{own}\,x\,y \wedge \textbf{say}\,x\,(\forall z.\,\textbf{love}\,y\,z)))
\end{aligned}
$$

By applying the $\mathsf{top}\ s$ composition of handlers, we have interpreted away all the operations. The result is of the form $\eta\, A$ where A are the truth conditions of the sentence. To retrieve A from $\eta\, A$, we can use $\downarrow_{\circ}$.

$$\downarrow_{\circ} t_7 = \downarrow_{\circ} (\mathsf{top}\ s\ [\![\textsc{said}_{\textsc{is}}\ (\textsc{loves}\ (\textsc{in-situ}\ \textsc{everyone})\ \textsc{it})\ (\textsc{who}_{\textsc{s}}\ (\textsc{owns}\ (\textsc{a}\ \textsc{dog}))\ (\textsc{best-friend}\ \textsc{me}))]\!])$$
$$\twoheadrightarrow \exists x.\ \textbf{best-friend}\ x\ s \wedge (\exists y.\ \textbf{dog}\ y \wedge \textbf{own}\ x\ y \wedge \textbf{say}\ x\ (\forall z.\ \textbf{love}\ y\ z))$$

C

Computer Mechanization of the Calculus

During the development of our approach, we have experimented with several implementations of effects and handlers. In our initial explorations, we used Kiselyov's extensible effects Haskell library [71, 2]. However, this made it difficult to separate our contribution from the details of the Haskell encoding of effects and handlers. In the next step, we turned to a programming language which included effects and handlers as first-class primitives, Bauer and Pretnar's *Eff* [16]. We have used *Eff* to prototype the grammars which led to us reporting our first results with effects and handlers in natural language [89]. Nevertheless, we found it difficult to develop larger grammars in a direct-style calculus without a suitable effect system.

We designed $(\!| \lambda |\!)$ to have a calculus in which evaluation (reduction) is independent of execution (the ordering of effects within computations). After the definition of $(\!| \lambda |\!)$ started to stabilize, we looked at ways of mechanizing the calculus so as to be able to use a computer to assist us in exploring the calculus. $(\!| \lambda |\!)$ is defined by a set of reduction rules and so we have used PLT Redex [42] to develop the mechanization of the calculus. Redex is a Racket library/domain-specific language for engineering and debugging reduction semantics. The formal language of terms and types, the typing rules and the reduction rules can all be written in a style close to the one used in academic papers. The formalization can then be used to automatically generate test cases for formal properties such as subject reduction, termination or progress.

We have formalized the syntax, typing rules and reduction rules using Redex. The source code can be found at:

`https://github.com/jirkamarsik/ling-eff/blob/master/redex/effects-and-handlers.rkt`

However, in order to be able to define the typing rules as judgment forms within Redex, we needed to include type annotations in our terms. Since $(\!| \lambda |\!)$ types include computation types, which are indexed by effect signatures, which contain more types, the type-annotated terms become unwieldy. We have therefore also developed a formalization which omits the type system but keeps the original syntax. This second formalization can be found at:

`https://github.com/jirkamarsik/ling-eff/blob/master/redex/untyped-bananas.rkt`

We have commented this formalization and we have also included the complete final grammar that we introduce in Chapter 8. The formalization can be used as a source of machine proofs for reductions in $(\!| \lambda |\!)$. For example, we can use the formalization to verify the predicted meaning of Example 42 from Appendix B.

However, this formalization was not practical for experimenting with the calculus since normalizing a large $(\!| \lambda |\!)$ term (such as the ones we saw in the linguistic examples throughout this dissertation) took about an hour when using Redex's reduction mechanisms. We have therefore also implemented a fast normalization procedure, which can be found at:

`https://github.com/jirkamarsik/ling-eff/blob/master/redex/fast-bananas.rkt`

In the rest of the appendix, we present an edited version of the implementation.

```
#lang racket
(require redex)
(require (for-syntax racket/syntax))

;; Below is a mechanization of the lambda-banana calculus defined in
;; 'Effects and Handlers in Natural Language'. The mechanization can be
;; consulted to verify the computations done in the dissertation and see a
;; formalized definition of the calculus and the grammar.  Beware that the
;; implementation of normalization is very inefficient and it can thus take
;; an hour to normalize a term large enough to represent an interesting
;; linguistic example.


;; Defining the Calculus
;; =====================
;;
;; These are the terms of the lambda-banana calculus, as defined in Section
;; 1.2 of the dissertation.
(define-language BANANA
  (e ::= x
         c
         (e e)
         (λ (x) e)
         (η e)
         (OP e (λ (x) e))
         ;; Since we cannot (easily) change the delimiters from parentheses
         ;; to banana brackets, we employ a different notation in this
         ;; implementation.
         (with (OP e) ... (η e) handle e)
         ;; DrRacket does not have a convenient shortcut for a cherry
         ;; symbol and so we use ♭.
         (♭ e)
         (C e))
  (x ::= variable-not-otherwise-mentioned)
  (c ::= variable-not-otherwise-mentioned)
  (OP ::= variable-not-otherwise-mentioned))

;; We then extend the set of terms with the constructions for sum types and
;; product types from Section 1.4 of the dissertation.
(define-extended-language BANANA+SP
  BANANA
  (e ::= ....
         ★
         (π1 e)
         (π2 e)
         (pair e e)
         (inl e)
         (inr e)
         (case e e e)
         (absurd e)))
```

```
;; Finally, we add the ambiguity operator that we introduced in Subsection 7.3.4
;; of the dissertation. Since semicolon is used by Racket to indicate
;; comments, we use || as the symbol.
(define-extended-language BANANA+SPA
  BANANA+SP
  (e ::= ....
         (e || e)))

;; We define a few necessary metafunctions on the terms of our calculus.

;; (no-match x (y ...)) is true iff x is different from all y ...
(define-metafunction BANANA+SPA
  no-match : any (any ...) -> #t or #f
  ...)

;; (free-in x e) is true iff x occurs free in e
(define-metafunction BANANA+SPA
  free-in : x e -> #t or #f
  ...)

;; (subst e x e_new) is the result of substituting e_new for all the free
;; occurrences of x in e (i.e. e[x := e_new])
(define-metafunction BANANA+SPA
  subst : e x e -> e
  ...)
```

```
;; We can now define the reduction relation of our calculus. This follows
;; closely the definitions given in Chapter 1 of the dissertation.
(define reduce
  (compatible-closure
    (reduction-relation BANANA+SPA #:domain e
      (--> ((λ (x) e_1) e_2)
           (subst e_1 x e_2)
           "β")
      (--> (λ (x) (e x))
           e
           (side-condition (not (term (free-in x e))))
           "η")
      (--> (with (OP_i e_i) ... (η e_p) handle (η e_v))
           (e_p e_v)
           "handle-η")
      (--> (with (OP_1 e_1) ... (OP_2 e_2) (OP_3 e_3) ... (η e_p)
            handle (OP_2 e_arg (λ (x) e_m)))
           ((e_2 e_arg) (λ (x_f) (with (OP_1 e_1) ...
                                       (OP_2 e_2)
                                       (OP_3 e_3) ...
                                       (η e_p)
                                 handle (subst e_m x x_f))))
           (side-condition (term (no-match OP_2 (OP_1 ...))))
           (fresh x_f)
           "handle-OP")
      (--> (with (OP_i e_i) ... (η e_p) handle (OP e_arg (λ (x) e_m)))
           (OP e_arg (λ (x_f) (with (OP_i e_i) ...
                                    (η e_p)
                              handle (subst e_m x x_f))))
           (side-condition (term (no-match OP (OP_i ...))))
           (fresh x_f)
           "handle-missing-OP")
      (--> (♭ (η e))
           e
           "♭")
      (--> (C (λ (x) (η e)))
           (η (λ (x) e))
           "C-η")
      (--> (C (λ (x) (OP e_a (λ (x_k) e_k))))
           (OP e_a (λ (x_k) (C (λ (x) e_k))))
           (side-condition (not (term (free-in x e_a))))
           "C-OP")
      (--> (π1 (pair e_1 e_2))
           e_1
           "β.×1")
      (--> (π2 (pair e_1 e_2))
           e_2
           "β.×2")
      (--> (case (inl e) e_l e_r)
           (e_l e)
           "β.+1")
      (--> (case (inr e) e_l e_r)
           (e_r e)
           "β.+2")) BANANA+SPA e))
```

```
;; Anaphora Resolution
;; ===================
;;
;; When computing the normal forms of the terms in our dissertation, we
;; often assume that the anaphora resolution operators sel_he, sel_she,
;; sel_it and selP choose some specific individual from the context, or in
;; the case of selP, recognize that the context does not contain any
;; suitable individual and reduce to some value which signals this. We will
;; want to use our mechanization to reduce lambda-banana terms to readable
;; truth-conditions and so we include reduction rules that implement a very
;; basic form of anaphora resolution into the reduction relation of our
;; calculus.

;; reduce-more extends the reduction relation 'reduce of our calculus. It
;; adds rules that concatenate contexts and look within them for anaphoric
;; antecedents.
(define reduce-more
  (compatible-closure
    (extend-reduction-relation reduce
      BANANA+SPAC ;; BANANA+SPAC extends BANANA+SPA with the gender markers
                  ;; 'masculine, 'feminine and 'neutral
      #:domain e
      (--> ((++ nil) e)
           e
           "++ nil")
      (--> ((++ ((::-ι e_h) e_t)) e_2)
           ((::-ι e_h) ((++ e_t) e_2))
           "++ ::-ι")
      (--> ((++ ((::-o e_h) e_t)) e_2)
           ((::-o e_h) ((++ e_t) e_2))
           "++ ::-o")
      (--> (sel-he e_context)
           e_referent
           (judgment-holds (sel masculine e_context e_referent))
           "sel-he")
      (--> (sel-she e_context)
           e_referent
           (judgment-holds (sel feminine e_context e_referent))
           "sel-she")
      (--> (sel-it e_context)
           e_referent
           (judgment-holds (sel neutral e_context e_referent))
           "sel-it")
      (--> ((selP e_property) e_context)
           (inl e_referent)
           (judgment-holds (sel-prop e_property e_context e_referent))
           "selP-found")
      (--> ((selP e_property) e_context)
           (inr ⋆)
           (judgment-holds (complete-ctx e_context))
           (side-condition (not (judgment-holds (sel-prop e_property
                                                          e_context
                                                          e_referent))))
           "selP-not-found")) BANANA+SPAC e))
```

```
;; Manipulating Terms
;; ==================
;;
;; Defined below are utility functions that allow us to normalize and
;; pretty-print terms.

;; simplify-logic is a reduction relation that implements some simple
;; logical rules.  Their point is to sanitize the logical formulas
;; generated by our system by, e.g., decoding the logical operators
;; ∀, ⇒ and ∨.
(define simplify-logic
  (compatible-closure
    (reduction-relation BANANA+SPA #:domain e
      (--> ((∧ e) ⊤)
           e)
      (--> ((∧ ⊤) e)
           e)
      (--> (¬ (¬ e))
           e)
      (--> (¬ (∃ (λ (x) e)))
           (∀ (λ (x) (¬ e))))
      (--> (¬ (∀ (λ (x) e)))
           (∃ (λ (x) (¬ e))))
      (--> (¬ ((∧ e_1) (¬ e_2)))
           ((⇒ e_1) e_2))
      (--> (¬ ((∧ e_1) (∀ (λ (x) (¬ e_2)))))
           ((⇒ e_1) (∃ (x) e_2)))
      (--> (¬ ((∧ (¬ e_1)) (¬ e_2)))
           ((∨ e_1) e_2))) BANANA+SPA e))

;; prettify-logic makes logical operators infix and translates
;; lambda-binders to quantifiers.
(define prettify-logic
  (context-closure
    (reduction-relation BANANA+SPA
      (--> ((∧ any_1) any_2)
           (any_1 ∧ any_2))
      (--> ((⇒ any_1) any_2)
           (any_1 ⇒ any_2))
      (--> ((∨ any_1) any_2)
           (any_1 ∨ any_2))
      (--> (∃ (λ (x) any))
           (∃ (x) any))
      (--> (∀ (λ (x) any))
           (∀ (x) any))
      (--> ((c_pred any_1 ...) any_2)
           (c_pred any_1 ... any_2)
           ;; We translate the convention of using boldface to typeset
           ;; logical predicates in the dissertation to the convention of
           ;; using symbols ending with * in this mechanization.
           (side-condition (string-suffix? (symbol->string (term c_pred)) "*"))))
    BANANA+SPAL
    context))
```

```
;; compute-truth-conditions combines all the steps necessary to go from
;; a lambda-banana term which encodes the meaning of a sentence to
;; human-readable truth-conditions of that sentence.
(define (compute-truth-conditions term)
  (let* ([normal-form (normalize-bottom-up reduce-more term)]
         [unblocked-form (resolve-blocked normal-form)]
         [simplified (normalize simplify-logic unblocked-form)]
         [pretty (normalize prettify-logic simplified)])
    pretty))



;; Common Combinators
;; ==================
;;
;; This part mirrors Section 1.6 of the dissertation. It introduces
;; syntactic shortcuts, combinators that we will make heavy use of.


;; We define the monadic bind (>>=) of our monad.
(define-metafunction BANANA+SPA
  >>= : e e -> e
  [(>>= e_m e_k)
   (with (η e_k) handle e_m)])

;; The op! syntax lets uses an operation with the trivial continuation
;; (lambda x. eta x).
(define-metafunction BANANA+SPA
  [(! OP)
   (λ (x) (OP x (λ (y) (η y))))]
  [(! OP e)
   (OP e (λ (x) (η x)))])

;; We functionalize OP, i.e. we turn the OP expression constructor into a
;; function expression. Also known as a generic effect.
(define-metafunction BANANA+SPA
  gen : OP -> e
  [(gen OP)
   (λ (x) (λ (k) (OP x (λ (y) (k y)))))])

;; This construction lets us omit the eta clause when writing
;; a handler. The default clause eta: (lambda x. eta x) is used.
(define-metafunction BANANA+SPA
  with-η : (OP e) ... handle e -> e
  [(with-η (OP e_h) ... handle e_arg)
   (with (OP e_h) ... (η (λ (x) (η x))) handle e_arg)])

;; We functionalize handlers. In lambda-banana, this corresponds to writing
;; a handler without giving its argument.
(define-metafunction BANANA+SPA
  handler : (OP e) ... (η e) -> e
  [(handler (OP e_h) ... (η e_p))
   ,(term-let ([x_f (variable-not-in (term (e_h ... e_p)) 'x)])
              (term (λ (x_f) (with (OP e_h) ... (η e_p) handle x_f))))])
```

```
;; We combine the two last abstractions to define a functionalized handler
;; expression with the default eta clause.
(define-metafunction BANANA+SPA
  handler-η : (OP e) ... -> e
  [(handler-η (OP e_h) ...)
   ,(term-let ([x_f (variable-not-in (term (e_h ...)) 'x)])
              (term (λ (x_f) (with-η (OP e_h) ... handle x_f))))])

;; We define a syntax for (n-ary) function composition.
(define-metafunction BANANA+SPA
  ∘ : e ... -> e
  ...)

;; We define function application for cases when the function is provided
;; by a computation.
(define-metafunction BANANA+SPA
  <<· : e e -> e
  [(<<· e_f e_x)
   ,(term-let ([x_f (variable-not-in (term e_x) 'f)])
              (term (>>= e_f (λ (x_f) (η (x_f e_x))))))])

;; We also define function application when the argument is the result of
;; a computation.
(define-metafunction BANANA+SPA
  ·>> : e e -> e
  [(·>> e_f e_x)
   ,(term-let ([x_f (variable-not-in (term e_f) 'x)])
              (term (>>= e_x (λ (x_f) (η (e_f x_f))))))])

;; Finally, we define function application for when both function and
;; argument are the results of computations. This is the <*> binary
;; operator of applicative functors.
(define-metafunction BANANA+SPA
  <<·>> : e e -> e
  [(<<·>> e_f e_x)
   ,(term-let ([x_f1 (variable-not-in (term e_x) 'f)]
               [x_f2 (variable-not-in (term x_f1) 'x)])
              (term (>>= e_f (λ (x_f1) (>>= e_x (λ (x_f2) (η (x_f1 x_f2))))))))])

;; When defining the open handler for dynamics (box), we will make use of
;; the following two combinators, introduced in Subsection 7.3.1 of
;; the dissertation.
(define-metafunction BANANA+SPA
  <<<· : e e -> e
  [(<<<· e_f e_x)
   ,(term-let ([x_f (variable-not-in (term e_x) 'f)])
              (term (>>= e_f (λ (x_f) (x_f e_x))))))])

(define-metafunction BANANA+SPA
  ∃>> : e -> e
  [(∃>> e_pred)
   (·>> ∃ (C e_pred))])
```

```
;; We will extend the above idea of applying an operation to computations
;; that yield the operands to other operations. We introduce a macro that,
;; given an operator 'op, defines the extended versions '<<op, 'op>> and
;; '<<op>>.
(define-syntax (extend-operator-to-computations stx)
  (syntax-case stx ()
    [(_ op)
     (with-syntax ([opl (format-id stx "<<~a" #'op)]
                   [opr (format-id stx "~a>>" #'op)]
                   [oplr (format-id stx "<<~a>>" #'op)])
       #'(begin
           (define-metafunction BANANA+SPA
             opl : e e -> e
             [(opl e_x e_y)
              ,(term-let ([x_f (variable-not-in (term e_y) 'x)])
                (term (>>= e_x (λ (x_f) (η ((op x_f) e_y))))))])
           (define-metafunction BANANA+SPA
             opr : e e -> e
             [(opr e_x e_y)
              ,(term-let ([x_f (variable-not-in (term e_x) 'y)])
                (term (>>= e_y (λ (x_f) (η ((op e_x) x_f))))))])
           (define-metafunction BANANA+SPA
             oplr : e e -> e
             [(oplr e_x e_y)
              ,(term-let ([x_f1 (variable-not-in (term e_y) 'x)]
                          [x_f2 (variable-not-in (term x_f1) 'y)])
                (term (>>= e_x (λ (x_f1) (>>= e_y (λ (x_f2) (η ((op x_f1) x_f2)))))))


;; We have conjunction,
(extend-operator-to-computations ∧)
;; disjunction,
(extend-operator-to-computations ∨)
;; implication,
(extend-operator-to-computations ⇒)
;; equality (on individuals),
(extend-operator-to-computations =)
;; adding an individual to a context,
(extend-operator-to-computations ::-ι)
;; adding a proposition to a context,
(extend-operator-to-computations ::-o)
;; and concatenating contexts.
(extend-operator-to-computations ++)
```

```
;; The next three definitions concern the expression of Boolean values
;; using sums (Subsection 1.5.4 in the dissertation). We define constants
;; for true...
(define-checked-term true
  (inl ⋆))
;; and false.
(define-checked-term false
  (inr ⋆))
;; Finally, we define if-then-else expressions using case analysis.
(define-metafunction BANANA+SPA
  ifte : e e e -> e
  [(ifte e_cond e_then e_else)
   (case e_cond (λ (,(variable-not-in (term e_then) '_)) e_then)
         (λ (,(variable-not-in (term e_else) '_)) e_else))])
```

```
;; Handlers
;; ========
;;
;; The rest of the program will cover the final grammar presented in
;; Chapter 8. We start by first regrouping the definitions of all the
;; handlers.

;; This is the box handler for dynamics, based on its presentation in
;; Section 8.1.  Note that the INTRODUCE operation has been decomposed into
;; FRESH and PUSH as in Subsection 8.5.1.
(define-checked-term box
  (λ (A)
    (<<<· ((handler
      (GET (λ (_) (λ (k)
             (η (λ (e) (GET ⋆ (λ (e_) (<<<· (k ((++ e) e_)) e))))))))
      (FRESH (λ (_) (λ (k)
               (η (λ (e) (∃>> (λ (x) (<<<· (k x) e))))))))
      (PUSH (λ (x) (λ (k)
              (η (λ (e) (<<<· (k ⋆) ((::-ι x) e)))))))
      (ASSERT (λ (p) (λ (k)
                (η (λ (e) (∧>> p (<<<· (k ⋆) ((::-o p) e))))))))
      (η (λ (_) (η (λ (e) (η ⊤)))))) A) nil)))

;; We have replaced INTRODUCE by FRESH and PUSH and so we express INTRODUCE
;; in terms of FRESH and PUSH.
(define-metafunction BANANA+SPA
  INTRODUCE : e e -> e
  [(INTRODUCE e_u e_k)
   ,(term-let ([x_f1 (variable-not-in (term e_k) 'x)]
               [x_f2 (variable-not-in (term (e_k x_f1)) '_)])
              (term (FRESH e_u (λ (x_f1)
                                 (PUSH x_f1 (λ (x_f2)
                                              (e_k x_f1)))))))])

;; The empty handler (Section 8.1) evaluates the discourse in an empty
;; anaphoric context.
(define-checked-term empty
  (handler-η (GET (λ (_) (λ (k) (k nil))))))

;; SI, which stands for Scope Island, is the handler for SCOPE effects,
;; which are used for quantification (Section 8.5).
(define-checked-term SI
  (handler-η (SCOPE (λ (c) (λ (k) (c k))))))

;; Next, we turn to presupposition (Section 8.2). We have the accommodate
;; handler, that accommodates presuppositions by introducing new discourse
;; referents.  Note that the predicate P is assumed to yield a computation
;; with DRT effects (GET, FRESH, PUSH, ASSERT) and not just a plain truth
;; value. This is to licence anaphoric binding from definite descriptions,
;; as in Subsection 8.6.2.
(define-checked-term accommodate
  (handler-η (PRESUPPOSE (λ (P) (λ (k)
               (INTRODUCE ⋆ (λ (x) (>>= (P x) (λ (_) (k x)))))))))
```

```
;; We will need to make a nondeterministic choice when trying to accommodate
;; a presupposition at different levels. The choose expression constructor,
;; which corresponds to the + operator in the dissertation (Section 8.2),
;; gives us a convenient syntax for the choice operation AMB : 1 → 2.
(define-metafunction BANANA+SPA
  choose : e e -> e
  [(choose e_1 e_2)
   ,(term-let ([x_f (variable-not-in (term (e_1 e_2)) 'b)])
              (term (AMB ⋆ (λ (x_f) (ifte x_f e_1 e_2)))))])

;; The maybeAccommodate handler uses choose to consider both projecting the
;; presupposition and accommodating it.
(define-checked-term maybeAccommodate
  (handler-η (PRESUPPOSE (λ (P) (λ (k)
              (choose (PRESUPPOSE P (λ (x) (k x)))
                      (INTRODUCE ⋆ (λ (x) (>>= (P x) (λ (_) (k x))))))))))

;; The find combinator is of the same type as (! PRESUPPOSE). It tries to
;; look for the missing entity in the context. If it cannot be found, it
;; projects the presupposition. Note that this find is the one from
;; Subsection 8.6.2, which expects dynamic predicates as arguments and uses
;; ♭ ∘ empty ∘ box to make them static.
(define-checked-term find
  (λ (P) (GET ⋆ (λ (e) (case ((selP (λ (x) (♭ (empty (box (P x)))))) e)
                        (λ (x) (η x))
                        (λ (_) (! PRESUPPOSE P)))))))

;; The useFind handler tries resolving the presuppositions within its
;; arguments using find.
(define-checked-term useFind
  (handler-η (PRESUPPOSE (λ (P) (λ (k) (>>= (find P) k))))))

;; maybeAccommodate introduces ambiguity via the AMB operator. The search
;; handler resolves the ambiguity by choosing which of the two
;; possibilities to pursue.  In the dissertation, we make this choice based
;; on whether or not the computations fail.  In this mechanization, we
;; leave the ambiguity operator unresolved.
(define-checked-term search
  (handler-η (AMB (λ (_) (λ (k) ((k true) || (k false)))))))

;; We incorporate the possibility to accommodate a presupposition in every
;; DRS on the projection line by introducing maybeAccommodate to the box
;; handler. We also add useFind so that presuppositions can be (preferably)
;; found within the context without having to be accommodated.
(define-checked-term dbox
  (∘ box maybeAccommodate useFind))
```

```
;; The next two handlers are the handlers for conventional implicature from
;; Section 8.3.  The asImplicature handler translates ASSERT to IMPLICATE
;; and INTRODUCE (i.e. FRESH and PUSH) to INTRODUCE_I (i.e. FRESH_I and
;; PUSH_I).
(define-checked-term asImplicature
  (handler-η (FRESH (gen FRESH_I))
             (PUSH (gen PUSH_I))
             (ASSERT (gen IMPLICATE))))
;; The withImplicatures handler reintegrates implicatures into the layer of
;; asserted meaning by reversing the translation done by asImplicature.
(define-checked-term withImplicatures
  (handler-η (FRESH_I (gen FRESH))
             (PUSH_I (gen PUSH))
             (IMPLICATE (gen ASSERT))))

;; withSpeaker is the handler for first-person pronouns from Section 8.4.
(define-checked-term withSpeaker
  (λ (s) (handler-η (SPEAKER (λ (_) (λ (k) (k s)))))))

;; Finally, we can compose all of the handlers to get an interpreter that
;; maps any computation in our fragment to a proposition.
(define-checked-term top
  (λ (s)
    (∘ search empty box accommodate useFind withImplicatures (withSpeaker s) SI)))
```

```
;; Dynamic Logic
;; =============
;;
;; This section introduces the logical operators that we will be using in
;; our grammar.  These are based on de Groote and Lebedeva's Type-Theoretic
;; Dynamic Logic.  Their lambda-banana definitions can be found in Section 8.1
;; of the dissertation.

(define-metafunction BANANA+SPA
  d∧ : e e -> e
  [(d∧ e_a e_b)
   (>>= e_a (λ (,(variable-not-in (term e_b) '_)) e_b))])

(define-metafunction BANANA+SPA
  d¬ : e -> e
  [(d¬ e_a)
   (>>= (dbox e_a) (λ (a) (! ASSERT (¬ a))))])

(define-metafunction BANANA+SPA
  d∃ : e -> e
  [(d∃ e_pred)
   ,(term-let ([x_f (variable-not-in (term e_pred) 'x)])
              (term (FRESH ⋆ (λ (x_f) (e_pred x_f)))))])

(define-metafunction BANANA+SPA
  d⇒ : e e -> e
  [(d⇒ e_a e_b)
   (d¬ (d∧ e_a (d¬ e_b)))])

(define-metafunction BANANA+SPA
  d∨ : e e -> e
  [(d∨ e_a e_b)
   (d¬ (d∧ (d¬ e_a) (d¬ e_b)))])

(define-metafunction BANANA+SPA
  d∀ : e -> e
  [(d∀ e_pred)
   ,(term-let ([x_f (variable-not-in (term e_pred) 'x)])
              (term (d¬ (d∃ (λ (x_f) (d¬ (e_pred x_f)))))))])
```

```
;; Grammar
;; =======
;;
;; What follows are lambda-banana terms which are the interpretations of
;; the lexical items that make up our grammar. This grammar combines all of
;; the phenomena discussed in Chapter 8 of the dissertation. In this edited
;; version, we only include the entries which feature in the example that
;; we study in Appendix B. For the complete grammar, see
;; https://github.com/jirkamarsik/ling-eff/blob/master/redex/untyped-bananas.rkt

;; it : NP
(define-checked-term it
  (GET ⋆ (λ (e)
  (η (sel-it e)))))

;; common-noun : N
;; Common nouns all have the same kind of meaning and so we define a macro
;; to facilitate the population of the lexicon.
(define-syntax-rule (define-common-noun abstract object)
  (define-checked-term abstract
    (λ (x) (! ASSERT (object x)))))

(define-common-noun dog dog*)

;; indef : N → NP
;; This is the semantics of the indefinite article. In the dissertation, we
;; call this constructor 'a. Here we give it a longer name so as to avoid
;; confusion with the variable 'a.
(define-checked-term indef
  (λ (n) (INTRODUCE ⋆ (λ (x)
         (>>= (n x) (λ (_)
         (η x)))))))

;; transitive-verb : NP → NP → S
(define-syntax-rule (define-transitive-verb abstract object)
  (define-checked-term abstract
    (λ (O) (λ (S) (SI (>>= (<<·>> (·>> object S) O) (! ASSERT)))))))

(define-transitive-verb loves love*)
(define-transitive-verb owns own*)

;; relational-noun : NP → NP
(define-syntax-rule (define-relational-noun abstract object)
  (define-checked-term abstract
    (λ (X) (>>= X (λ (x)
           (! PRESUPPOSE (λ (y) (! ASSERT ((object y) x)))))))))

(define-relational-noun best-friend best-friend*)

;; who_s : (NP → S) → NP → NP
(define-checked-term who_s
  (λ (C_) (λ (X) (>>= X (λ (x)
                 (>>= (asImplicature (C_ (η x))) (λ (_)
                 (η x)))))))))
```

```
;; me : NP
(define-checked-term me
  (! SPEAKER ⋆))

;; said_is : S → NP → S
(define-checked-term said_is
  (λ (C_) (λ (S) (SI (>>= (<<·>> (·>> say* S) (dbox C_)) (! ASSERT))))))

;; in-situ : QNP → NP
(define-checked-term in-situ
  (λ (Q) (>>= Q (λ (X) X))))

;; The following is a helper combinator for the lexical entry of
;; quantifiers.  (trace x) is a term that evaluates to x but also
;; introduces x to the context.
(define-checked-term trace
  (λ (x) (PUSH x (λ (_) (η x)))))

;; everyone : QNP
(define-checked-term everyone
  (SCOPE (λ (k) (d∀ k))
         (λ (x) (η (trace x)))))



;; Examples
;; ========
;;
;; We end this program with example abstract terms that can be
;; evaluated. The examples are taken from the dissertation and can
;; therefore be used to verify the calculations that are done "on paper" in
;; the dissertation. In this edited version, we include only
;; the example from Appendix B.

;; My best friend, who owns a dog, said it loves everyone.
;; (Section 8.7, Appendix B)
;; (compute-truth-conditions (term example-final))
;; '(∃ (x8) ((best-friend* x8 s) ∧
;;  (∃ (x1) ((dog* x1) ∧ ((own* x8 x1) ∧
;;          (say* x8 (∀ (x6) (love* x1 x6))))))))
(define-checked-term example-final
  (♭ ((top s) ((said_is ((loves (in-situ everyone)) it))
               ((who_s (owns (indef dog))) (best-friend me))))))
```

Résumé en français

La motivation de ces travaux de thèse est de proposer des traductions automatiques d'énoncés en langue naturelle en des représentations logiques. L'intérêt est double, d'un côté de permettre aux linguistes de définir une sémantique formelle de la langue et de l'autre côté de permettre aux informaticiens de les utiliser pour, par exemple, créer des systèmes de raisonnement automatique sur des données exprimées en langue naturelle.

De nombreux travaux de recherche cherchent à définir un système transformant des énoncés en anglais ou en d'autres langues en représentation sémantique. Mais ils se heurtent à plusieurs difficultés. Par exemple dans une phrase en anglais, on peut identifier plusieurs phénomènes linguistiques et leur associer des théories qui en rendent compte.

(1) She might still be dating that idiot.

Cet exemple est constitué :

1. d'expressions anaphoriques, comme le pronom *she*. On sait qu'il est possible de traduire des phrases avec des anaphores dans des structures de la Discourse Representation Theory (DRT) [64], des formules de la Dynamic Predicate Logic (DPL) [53] ou des λ-termes avec des continuations [38].

2. de l'auxiliaire modal *might* qu'on peut traduire comme un opérateur d'une logique modale ou une quantification existentielle sur des mondes possibles.

3. du temps présent progressif dans *be dating*. Comme avec l'auxiliaire modal, on peut le traduire en un opérateur d'une logique temporelle ou introduire une quantification existentielle sur des intervalles de temps pendant lesquels le fait de *dating* a lieu et postuler que le moment d'énonciation de la phrase se trouve dans cet intervalle.

4. du déclencheur de présupposition *still* qui implique que le sujet et l'objet sortaient déjà ensemble. Un mécanisme permettant de projeter cette présupposition en dehors de la portée de n'importe quel opérateur logique est alors nécessaire.[193] On peut utiliser la stratégie de Lebedeva [80] qui utilise une levé d'exception pour projeter la présupposition.

5. de l'épithète expressif *idiot*. En suivant la théorie des implicatures conventionnelles de Potts [108], élaborée par Gutzmann [54], on introduit une deuxième dimension du sens dans laquelle on va noter l'attitude négative de l'interlocuteur envers l'objet.

Les descriptions ci-dessus peuvent sembler être une bonne première approximation de l'exemple. Il est possible de les reprendre et intuitivement construire une représentation logique raisonnable du contenu informationnel. Mais comment formaliser ce processus complexe ? La plupart de ces théories utilisent leur propre système de représentation, leurs propres définitions, notations et opérations.

La DRT introduit un encodage spécifique des formules logiques et propose un algorithme qui les construits progressivement à partir de la phrase [64]. La logique des implicatures conventionnelles de Potts introduit des formules logiques bidimensionnelles et définit ses propres façons de les combiner [108].

[193] Dans le présent, on peut déduire que les deux sont *peut-être* en train de sortir ensemble (grâce à l'auxiliaire modal *might*), mais avec la présupposition, on peut déduire que dans le passé, ils ont *dû* sortir ensemble.

Les traitements compositionnels de l'intensionalité ou du temps ont tendance à utiliser le λ-calcul simplement typé [18, 40], comme c'est également le cas avec le traitement des anaphores de de Groote [38]. Pour son traitement des présuppositions, Lebedeva utilise une version modifiée du calcul de de Groote qui inclut aussi les exceptions [80].

Il semble clair que pour rendre compte des phénomènes présentés dans les descriptions 1 à 5, il est pertinent de proposer un cadre universel basé sur un langage formel.

Les monades

Notre langage universel est fondé sur le λ-calcul. Grâce au travail très influent de Richard Montague [98], le λ-calcul est un formalisme répandu en sémantique compositionnelle formelle.[194] Beaucoup de systèmes pour la sémantique utilisent le λ-calcul et les autres ont tendance à se traduire en λ-calcul (par exemple la λ-DRT [79] ou le traitement de dynamicité à partir des continuations de de Groote [38]).

Cependant, même si nous avons plusieurs théories qui sont toutes formalisées dans le λ-calcul, cela ne signifie pas nécessairement qu'elles sont compatibles ou que nous savons les combiner. Une théorie de l'intensionalité pourrait déclarer que les phrases sont traduites en termes de type $\sigma \to o$, le type des fonctions des mondes possibles vers les valeurs de vérité. D'un autre côté, une analyse des expressives suggère que des phrases devraient correspondre à des termes de type $o \times \epsilon$, le type des paires composées de valeurs de vérité (le contenu propositionnel) et de marqueurs expressifs (le contenu expressif). Les deux théories seraient compatibles au niveau du calcul utilisé mais pas au niveau des termes. Une fonction traitant des propositions intensionnelles ne seraient pas directement applicables à une proposition expressive.

Pour poursuivre notre recherche d'uniformité et de compatibilité des opérations sémantiques, nous avons examiné les termes et les types utilisés par les traitements sémantiques à partir du λ-calcul et nous avons essayé de trouver une structure sous-jacente commune. Nous remarquons que l'ensemble de ces approches partagent les caractéristiques suivantes :

1. Les types de certaines dénotations sont élargies. Par exemple, lorsqu'il s'agit de quantificateurs, le type des dénotations des syntagmes nominaux va de ι (des individus) à $(\iota \to o) \to o$ (des quantificateurs généralisés sur les individus) ; dans la sémantique intensionnelle, le type des dénotations des phrases va de o (des valeurs de vérité) à $\sigma \to o$ (des fonctions des mondes possibles vers des valeurs de vérité, c.-à-d. des ensembles de mondes possibles) ; et avec les expressives, le type des dénotations des phrases va de o à $o \times \epsilon$ (des valeurs de vérité associées à des marqueurs expressifs).

2. Il existe un processus pouvant transformer des dénotations de l'ancien type dans des dénotations du nouveau type. Dans l'exemple avec les quantificateurs, il s'agit de la fameuse opération de type «*type raising*». Dans l'exemple avec les intensions, il s'agit du combinateur **K** qui transforme une valeur de vérité en une fonction constante qui attribue cette valeur de vérité à tous les mondes, (une intension rigide). Dans l'exemple avec des expressifs, c'est la fonction qui couple une proposition avec un marqueur expressif neutre/vide.

3. Puis il y a d'autres termes de type étendu qui ne peuvent pas être obtenus en utilisant la fonction de *type raising* évoqué ci-dessus ; ce sont ceux pour lequel nous avons élargi le type. Les syntagmes nominaux quantificationnel tels que *everyone* ne sont pas les résultats d'un *type raising* sur un terme de type d'individu. Les propositions intensionnelles telles que *Hespherus is Phosphorus* ont des extensions qui varient d'un monde à l'autre. Les expressives tel que le diminutif *Wolfie* indiquent un individu et portent également un marqueur expressif qui transmet l'attitude du locuteur envers le référent.

4. Enfin, ces approches ont aussi une stratégie de composition des dénotations plus petites en des dénotations plus grandes et de traitement de la complexité ajoutée par les types plus élaborés. Lorsqu'on applique un verbe transitif à un sujet et à un objet quantificationnel, on laisse l'un (souvent

[194] Il est généralement motivé par le principe de compositionnalité de Frege qui dit que le sens d'une expression complexe doit être déterminé par le sens de ses constituants (c.-à-d. être une fonction de). Si un sens complexe est une fonction d'autres sens, il est logique d'utiliser un calcul sur les fonctions, comme le λ-calcul.

le sujet) prendre la portée et ensuite on laisse l'autre prendre la sienne. Lorsque nous appliquons le verbe aux arguments intensionnels, nous passons le monde où nous évaluons la phrase au sujet et à l'objet. Quand on l'applique à des arguments expressifs, on applique le verbe aux référents du sujet et de l'objet et on recueille le contenu expressif des deux.

Ce type de structure est en réalité très commun dans la programmation fonctionnelle et dans la sémantique dénotationnelle des langages de programmation. C'est la structure d'un *foncteur applicatif* [91]. Les exemples ci-dessus sont également des instances d'une structure plus spéciale appelée une *monade* [95].

Nous n'entrons pas dans cette section dans les détails de la définition d'une monade que l'on peut retrouver dans ce document, mais nous allons néanmoins en donner un aperçu. Une monade est un triplet $(T, \eta, \gg\!=)$ où T est une fonction sur les types (l'expansion des types que nous avons vu dans la section 1), η est une façon de transformer des valeurs simples en valeurs étendues (les fonctions de *type raising* dans 2) et $\gg\!=$ nous donne un moyen général de combiner des valeurs de ce type étendu (semblable aux exemples données dans 4).[195] Le triplet doit également satisfaire certaines propriétés algébriques qui garantissent que la composition des fonctions sur des types étendus est associative et que la fonction de *type raising* sert d'unité pour cet opérateur de composition.

Les analyses discutées ci-dessus sont toutes des exemples de monades. La prévalence des monades dans la sémantique de la langue a déjà été mise en avant par Shan dans [113]. Cependant, le défi consiste à combiner l'utilisation de plusieurs monades en même temps.

Les effets de bord linguistiques

Les monades apparaissent souvent dans la sémantique dénotationnelle des langages de programmation pour tenir compte des notions de calcul communément appelées *les effets de bord* [95]. Nous pouvons nous baser sur cette correspondance et considérer la structure monadique en langue naturelle comme des effets de bord linguistiques. Cette analogie a été poursuivie par Shan [115, 116] et Kiselyov [67] et est présente dans les travaux récents sur les monades dans la sémantique de la langue naturelle [48, 27]. Cependant, l'idée remonte avant que les monades soient introduites en informatique. Dans leur article de 1977, Hobbs et Rosenschein prennent une perspective computationnelle sur la logique intensionnelle de Montague [98] : les intensions correspondent aux programmes et les extensions correspondent aux valeurs. Un programme peut accéder aux variables globales qui décrivent l'état du monde.[196] Les opérateurs $\uparrow$ et $\downarrow$, qui traduisent les expressions dénotant les extensions et les expressions dénotant les intensions, correspondent respectivement aux opérateurs `quote` et `eval` dans le langage Lisp.

L'idée de traiter les expressions linguistiques comme des actions ou des programmes avec des effets est également pertinente pour la sémantique dynamique, qui traite le sens d'une phrase comme des instructions mettant à jour le «common ground» (ou un autre contexte linguistique).[197] La sémantique dynamique et l'anaphore sont parfois classées comme appartenant à la fois à la sémantique et à la pragmatique. C'est également le cas pour d'autres phénomènes que nous traitons comme des effets de bord dans cette thèse : la deixis, la présupposition, l'implicature conventionnelle. La pragmatique étudie la façon dont un langage se situe dans la communauté de ses utilisateurs, c.-à-d. comment il est réellement utilisé par ses locuteurs pour atteindre leurs objectifs. Il ne serait donc pas surprenant que la pragmatique corresponde bien aux effets de bord des langages de programmation, car les effets de bord eux-mêmes concernent la façon dont les programmes peuvent interagir avec le monde de leurs utilisateurs (par exemple, en faisant apparaître des choses sur l'écran ou en récupérant des entrées fournies par l'utilisateur).

[195] Cette façon de présenter une monade (un constructeur de type, η et $\gg\!=$) est particulière à la programmation fonctionnelle. Notez que cette présentation diffère de celle utilisée dans la théorie des catégories, qui remplace $\gg\!=$ par une transformation naturelle μ [88].

[196] La dépendance à un environnement d'un certain type σ est un effet de bord qui peut être décrit en utilisant la monade «reader». Cette monade transforme le type α vers le type $\sigma \to \alpha$. Cette transformation de types est exactement la même que celle décrite par les théories d'intensionalisation [18, 40].

[197] L'utilisation des monades pour encoder des effets dynamiques (l'anaphore) remonte à 2009 et aux travaux de Giorgolo et Unger [52, 129].

Les effets et les handlers

En prenant les différentes structures monadiques de la sémantique de la langue naturelle comme des effets de bord, nous pouvons appliquer des théories qui les combinent pour définir un formalisme capable d'exprimer différents aspects du langage en même temps. Les effets et les handlers sont pour nous un tel cadre théorique. Ici, les programmes sont interprétés comme des séquences d'instructions (ou plus généralement comme des arbres de décision).[198] Les instructions sont des symboles appelés des *opérations*, qui représentent les différents effets, les différentes façons dont les programmes peuvent interagir avec leur contexte. Dans notre application à la sémantique de la langue naturelle, voici quelques exemples d'opérations qui figurent dans nos définitions, ainsi que leur sémantique :[199]

- `introduce` introduit un nouveau référent de discours dans le contexte. C'est l'opération utilisée par des syntagmes nominaux comme l'indéfini *a man*.
- `presuppose` P présuppose l'existence d'une entité satisfaisant le prédicat P. Elle est utilisée par des descriptions définies *the* P et par des noms propres.
- `implicate` i indique que i est une implicature conventionnelle. Cette opération est utilisée par des constructions appositives telles que *John, who is my neighbor*.
- `speaker` demande au contexte l'identité du locuteur. Il est utilisé par le pronom de première personne pour trouver son référent.

Le calcul de la dénotation d'une expression linguistique est décomposé en ces opérations. Lorsque les expressions se combinent pour former des phrases et des discours, ces opérations finissent par être concaténées dans un grand programme qui effectue une série d'interactions avec son contexte. C'est à ce point que les handlers sont utilisés. Un *handler* est un interpréteur qui donne une définition aux symboles d'opération présents dans un programme. Les handlers peuvent être rendus modulaires[200] afin que l'interprète global de toutes les interactions avec le contexte puisse être défini comme la composition de plusieurs petits handlers, chacun traitant un aspect différent du langage (la dynamique, les implicatures, la deixis…).

Lorsque nous utilisons les effets et les handlers, nous commençons par énumérer l'ensemble de toutes les interactions possibles que les programmes (c.-à-d. les expressions linguistiques dans notre application) peuvent avoir avec leurs contextes. Ensuite, nous pouvons interpréter les expressions linguistiques comme des séquences d'instructions. Enfin, nous écrivons des handlers qui mettent en œuvre ces instructions et produisent une représentation sémantique appropriée. Cette approche suit donc de près le mantra donné par Lewis :

> *In order to say what a meaning is, we may first ask what a meaning does and then find something that does that.*
>
> Afin de dire ce que c'est le sens, nous pouvons d'abord demander ce que le sens fait et puis trouver quelque chose qui le fait.
>
> General Semantics, David Lewis [83]

Nous pouvons faire remonter les origines des effets et des handlers à deux moments. L'un est le travail de Cartwright et Felleisen sur les Extensible Denotational Language Specifications [24], dans lequel une technique de développement de la sémantique est présentée qui fait que lorsqu'un langage (de programmation) est étendu avec des nouvelles constructions (et des nouveaux effets de bord), les dénotations restent compatibles et peuvent être réutilisées. L'autre précurseur est le travail de Hyland, de Plotkin et de Power sur les effets algébriques [60], une technique catégorique pour étudier les programmes avec

[198] Plus précisément, nous interprétons des programmes dans une monade libre [123].
[199] Les opérations ne sont que des symboles et n'ont donc pas un sens inhérent.
[200] De la même manière que les monades peuvent être transformées en «monad transformers» («monad morphisms») puis composées [113, 134].

des effets de bord, qui fut ensuite étendu par Plotkin et Pretnar pour inclure les handlers [103, 110, 104]. La technique a gagné en popularité ces dernières années (depuis 2012). Elle trouve des applications à la fois dans l'encodage des effets dans les langages de programmation fonctionnels purs [71, 70, 63, 22] et dans la conception des langages de programmation [16, 87, 41, 73, 56]. Notre thèse explore l'applicabilité des effets et des handlers à la sémantique de la langue naturelle.

Résumé des résultats

Après avoir introduit la problématique dans laquelle se situe ces travaux, nous revenons sur une description du contenu du manuscrit.

Dans la partie I, nous introduisons $(\!| \lambda |\!)$, un calcul formel qui étend le λ-calcul simplement typé (STLC) avec des effets et des handlers.

La définition de $(\!| \lambda |\!)$ est donnée dans le chapitre 1. $(\!| \lambda |\!)$ introduit une nouvelle famille de types dans STLC, les types des calculs, et des nouveaux termes, qui comprennent des constructeurs et des destructeurs des calculs. Nous avons donné un système de types au calcul qui étend celui de STLC et une sémantique de réduction qui combine les réductions de STLC β et η avec les définitions des nouveaux symboles. Dans ce chapitre, nous maintenons deux perspectives sur la signification de ces termes : les calculs peuvent être considérés comme des programmes qui interagissent avec un système à travers un ensemble d'opérations spéciales ou ils peuvent être considérés comme des expressions algébriques construites à partir d'une signature algébrique infinie.

Dans le chapitre 2, nous avons donné un exemple d'utilisation du calcul $(\!| \lambda |\!)$. En plus d'introduire les notations et les réductions du calcul, l'exemple sert d'aperçu du genre d'ingénierie linguistique que nous faisons plus tard dans le manuscrit. Au cours du chapitre, nous développons une sémantique compositionnelle pour un langage informatique simple avec des erreurs et des variables. Ceci nous permet de démontrer la modularité de l'utilisation de notre monade de calcul, car nous pouvons ajouter des variables au langage sans avoir à modifier la sémantique des autres constructions.

La contribution principale de la partie I se trouve dans le chapitre 3, dans lequel nous avons développé la métathéorie de $(\!| \lambda |\!)$. Dans la section 3.1, les concepts qui sont primitifs dans d'autres langues (les handlers clos et l'opérateur $\gg\!=$) ont été définis dans $(\!| \lambda |\!)$ et leurs règles de typage et de réduction ont été dérivées à partir de celles de $(\!| \lambda |\!)$. Dans la section 3.3, nous avons ensuite relié le calcul à la théorie des monades en identifiant une monade dans la catégorie dans laquelle nous interprétons $(\!| \lambda |\!)$ avec notre *sémantique dénotationnelle*. Dans la section 3.2, nous avons prouvé la *préservation des types* («subject reduction») dans $(\!| \lambda |\!)$. Ce résultat donne une cohérence entre le système des types de $(\!| \lambda |\!)$ et sa sémantique de réduction, garantissant que les types sont conservés par la réduction. Ceci est complété par une preuve de *progression*, qui indique que les termes qui n'utilisent aucun des opérateurs partiels et qui ne peuvent plus être réduits doivent avoir une forme spécifique.

Nous avons enchaîné avec la preuve d'une autre propriété fondamentale : la *normalisation forte*. Sa preuve a été divisée en deux parties : *confluence* (prouvée dans la section 3.4) et *terminaison* (prouvée dans la section 3.5). Les preuves de confluence et de terminaison procèdent par des stratégies similaires : prouver la propriété pour le calcul sans la η-réduction en appliquant un résultat général, puis étendre la propriété au calcul complet. Dans le cas de la confluence, le résultat général est la confluence des systèmes de réduction combinatoires orthogonaux [76]. Dans le cas de la terminaison, nous nous appuyons sur deux techniques : la terminaison de la relation de réduction dans les systèmes de type de données inductives qui valident le schéma général [19] et l'étiquetage sémantique d'ordre supérieur [55], ce qui nous permet d'utiliser notre sémantique dénotationnelle pour étiqueter les termes de notre calcul afin de valider le schéma général.

Andrej Bauer a fait l'analogie dans laquelle les effets et les handlers sont aux continuations délimitées ce que les boucles while ou les constructions if-then-else sont aux instructions du type *goto* [15]. Les continuations elles-mêmes se sont avérées être un outil efficace dans la sémantique de la langue naturelle [36, 12, 115, 38, 11, 14]. Dans le chapitre 4, nous avons montré comment $(\!| \lambda |\!)$ peut simuler des continuations délimitées, à savoir les opérateurs de contrôle délimitée `shift/reset`. Nous avons présenté un λ-calcul avec appel par valeur et les opérateurs `shift` et `reset` et simulé ses types et ses réductions dans $(\!| \lambda |\!)$.

Dans la partie II, nous démontrons les applications de $(\!| \lambda |\!)$ aux problèmes de modélisation du sens des énoncés de la langue naturelle.

Après avoir examiné les bases de la sémantique formelle dans le chapitre 5, nous avons montré comment les calculs dans $(\!| \lambda |\!)$ peuvent être utilisés pour donner une sémantique compositionnelle à plusieurs phénomènes linguistiques "non compositionnels". Dans le chapitre 6, nous avons décrit comment introduire des calculs dans une sémantique compositionnelle tout en préservant les dénotations assignées par la sémantique dans la section 6.1. Nous avons ensuite présenté des analyses de plusieurs phénomènes linguistiques dans $(\!| \lambda |\!)$: la deixis (section 6.2), l'implicature conventionnelle à la Potts (section 6.3) et la quantification à la Montague (section 6.4). Nous avons ensuite décrit explicitement la méthodologie utilisée pour trouver le genre d'analyses que nous avons présentées dans le chapitre 6 afin d'encourager les chercheurs à élaborer d'autres analyses dans le même cadre.

Nous avons consacré le chapitre 7 à un phénomène particulièrement complexe : la dynamique. Dans les sections 7.1 et 7.2, nous avons montré comment une analyse $(\!| \lambda |\!)$ de la dynamique peut être extraite à partir de la Discourse Representation Theory. Cela donne un moyen de gérer la dynamique dans $(\!| \lambda |\!)$ et renforcer l'idée que les effets et les handlers sont des mécanismes appropriés pour traiter la langue naturelle. Nous avons également montré comment interpréter les calculs $(\!| \lambda |\!)$ comme les propositions dynamiques de la Type-Theoretic Dynamic Logic (TTDL) [38]. Dans sa thèse [80], Lebedeva a étendu TTDL avec des exceptions pour traiter les présuppositions et dans la section 7.3, nous avons intégré l'analyse de présuppositions de Lebedeva dans notre analyse $(\!| \lambda |\!)$ de la dynamique (nous avons comparé notre adaptation avec l'original dans 7.3.5). Dans la section 7.4, nous avons considéré une autre extension de TTDL pour la double négation [111] et nous avons montré que le *type raising* des dénotations qui y est utilisé ne peut pas être simulé dans $(\!| \lambda |\!)$.

Dans le chapitre 8, nous montrons comment l'utilisation d'effets et de handlers permet de combiner le traitement de phénomènes différents dans une seule grammaire. Nous avons commencé avec la grammaire dynamique développée dans le chapitre 7, répété dans les sections 8.1 et 8.2. Nous avons ensuite étendu cette grammaire avec les implicatures conventionnelles (8.3), la deixis (8.4) et la quantification (8.5) avec peu ou aucune modification de la sémantique originale. Nous terminons le chapitre en esquissant une analyse des subordonnées restrictives et leurs interactions avec les présuppositions dans la section 8.6.

Résumé

Ces travaux s'intéressent à la modélisation formelle de la sémantique des langues naturelles. Pour cela, nous suivons le principe de compositionalité qui veut que le sens d'une expression complexe soit une fonction du sens de ses parties. Ces fonctions sont généralement formalisées à l'aide du λ-calcul. Cependant, ce principe est remis en cause par certains usages de la langue, comme les pronoms anaphoriques ou les présuppositions. Ceci oblige à soit abandonner la compositionalité, soit modifier les structures du sens. Dans le premier cas, le sens n'est alors plus obtenu par un calcul qui correspond à des fonctions mathématiques, mais par un calcul dépendant du contexte, ce qui le rapproche des langages de programmation qui manipulent leur contexte avec des effets de bord. Dans le deuxième cas, lorsque les structures de sens sont ajustées, les nouveaux sens ont tendance à avoir une structure de monade. Ces dernières sont elles-mêmes largement utilisées en programmation fonctionnelle pour coder des effets de bord, que nous retrouvons à nouveau. Par ailleurs, s'il est souvent possible de proposer le traitement d'un unique phénomène, composer plusieurs traitements s'avère être une tâche complexe. Nos travaux proposent d'utiliser les résultats récents autour des langages de programmation pour parvenir à combiner ces modélisations par les effets de bord.

Pour cela, nous étendons le λ-calcul avec une monade qui implémente les *effects* et les *handlers*, une technique récente dans l'étude des effets de bord. Dans la première partie de la thèse, nous démontrons les propriétés fondamentales de ce calcul (préservation de type, confluence et terminaison). Dans la seconde partie, nous montrons comment utiliser le calcul pour le traitement de plusieurs phénomènes linguistiques : deixis, quantification, implicature conventionnelle, anaphore et présupposition. Enfin, nous construisons une unique grammaire qui gère ces phénomènes et leurs interactions.

Mots-clés : sémantique formelle, compositionalité, effets de bord, monades, grammaires catégorielles abstraites, sémantique dynamique.

Abstract

In formal semantics, researchers assign meanings to sentences of a natural language. This work is guided by the principle of compositionality: the meaning of an expression is a function of the meanings of its parts. These functions are often formalized using the λ-calculus. However, there are areas of language which challenge the notion of compositionality, e.g. anaphoric pronouns or presupposition triggers. These force researchers to either abandon compositionality or adjust the structure of meanings. In the first case, meanings are derived by processes that no longer correspond to pure mathematical functions but rather to context-sensitive procedures, much like the functions of a programming language that manipulate their context with side effects. In the second case, when the structure of meanings is adjusted, the new meanings tend to be instances of the same mathematical structure, the monad. Monads themselves being widely used in functional programming to encode side effects, the common theme that emerges in both approaches is the introduction of side effects. Furthermore, different problems in semantics lead to different theories which are challenging to unite. Our thesis claims that by looking at these theories as theories of side effects, we can reuse results from programming language research to combine them.

This thesis extends λ-calculus with a monad of computations. The monad implements effects and handlers, a recent technique in the study of programming language side effects. In the first part of the thesis, we prove some of the fundamental properties of this calculus: subject reduction, confluence and termination. Then in the second part, we demonstrate how to use the calculus to implement treatments of several linguistic phenomena: deixis, quantification, conventional implicature, anaphora and presupposition. In the end, we build a grammar that features all of these phenomena and their interactions.

Keywords: formal semantics, compositionality, side effects, monads, abstract categorial grammars, dynamic semantics.