Polarities & Focussing: a journey from Realisability to Automated Reasoning

Stéphane Graham-Lengrand

Polarities & Focussing: a journey from Realisability to Automated Reasoning

Stéphane Graham-Lengrand

Dissertation submitted towards the degree of
Habilitation à Diriger des Recherches
Université Paris-Sud

Thesis prepared at École Polytechnique, with support from CNRS and INRIA,

and publicly defended on 17th December 2014 before

Wolfgang Ahrendt	Referee
Hugo Herbelin	Referee
Frank Pfenning	Referee
Sylvain Conchon	Examiner
David Delahaye	Examiner
Didier Galmiche	Examiner
Laurent Regnier	Examiner
Christine Paulin-Mohring	Examiner

Abstract

This dissertation explores the roles of polarities and focussing in various aspects of Computational Logic.

These concepts play a key role in the the interpretation of proofs as programs, a.k.a. the Curry-Howard correspondence, in the context of classical logic. Arising from linear logic, they allow the construction of meaningful semantics for cut-elimination in classical logic, some of which relate to the Call-by-Name and Call-by-Value disciplines of functional programming. The first part of this dissertation provides an introduction to these interpretations, highlighting the roles of polarities and focussing. For instance: proofs of positive formulae provide structured data, while proofs of negative formulae consume such data; focussing allows the description of the interaction between the two kinds of proofs as pure pattern-matching. This idea is pushed further in the second part of this dissertation, and connected to realisability semantics, where the structured data is interpreted algebraically, and the consumption of such data is modelled with the use of an orthogonality relation. Most of this part has been proved in the Coq proof assistant.

Polarities and focussing were also introduced with applications to logic programming in mind, where computation is proof-search. In the third part of this dissertation, we push this idea further by exploring the roles that these concepts can play in other applications of proof-search, such as theorem proving and more particularly automated reasoning. We use these concepts to describe the main algorithm of SAT-solvers and SMT-solvers: DPLL. We then describe the implementation of a proof-search engine called Psyche. Its architecture, based on the concept of focussing, offers a platform where smart techniques from automated reasoning (or a user interface) can safely and trustworthily be implemented via the use of an API.

Acknowledgements

It is difficult to determine when to write an habilitation thesis, what it will include and which format it will have. In the case of a Ph.D. thesis, such issues are often entirely resolved by the end of the Ph.D. funding and the input of the Ph.D. adviser. Therefore, encouragements to write an habilitation thesis are all the more important, and for this reason I am grateful to Olivier Bournez, head of our research laboratory (LIX), who first planted the idea in my head, as well as to Benjamin Werner, head of Polytechnique's C.S. department, for his friendly support.

In fact I surprisingly found, in the habilitation process, no other obstacles than those I encountered on my own, as everybody that I interacted with helped me overcome them:

In particular, I am indebted to my team leader Dale Miller and to my Ph.D. adviser Roy Dyckhoff, who recommended me when I enrolled. I am grateful to Stéphanie Druetta and Marie-Christine Mignier[1] at the Paris-Sud administration, as well as to Dominique Gouyou-Beauchamps, whose work at the C.S. School of Doctoral Studies guarantees the high standards of the degree towards which this dissertation is submitted. They all worked impressively efficiently, especially given the tight schedules that characterised my application process.

Christine Paulin-Mohring was kind enough to sponsor my application and be one of the first people to look into my dissertation. I thank her, as well as Sylvain Conchon, David Delahaye, Didier Galmiche and Laurent Regnier, for accepting to sit on the defence panel. For the same reason, but also for having reported on my dissertation, I wish to thank Wolfgang Ahrendt, Hugo Herbelin and Frank Pfenning: they honoured me with their time and interest.

The work described in this dissertation benefitted from many years of interactions within my departement and elsewhere: I am very grateful to all past and present assistants at LIX[2] and of course I am also very grateful to all of my colleagues (whether or not they are inclined towards Logic) for making Polytechnique's C.S. department and my research community such a great working environment. I am particularly thankful to Assia Mahboubi for the many years of scientific interaction and friendship that have passed since we studied at ENS Lyon.

I understand that an *Habilitation à Diriger des Recherches* also has to do with student supervision. Students in general have been a very important part of my work ever since I came back to France. The first part of this dissertation results from my teaching at MPRI. The Psyche engine was inspired by a few lines of code from a undergraduate students' project at ESIEA, while its latest development results from Damien Rouhling's internship at LIX. I also learnt a lot from interacting, in very different styles, with my two Ph.D. students Mahfuza Farooque and Alexis Bernadet; congratulations again for your work and for having successfully defended your theses before mine. In brief, I am very grateful to all of the students I have worked with.

Besides Academia, I thank my parents and my friends who supported me in the habilitation process, particularly the Rémi(e)s who repeatedly dared to ask me for news updates, no matter how uninteresting to them the topics of *Polarities and Focussing* must have been. Finally, no-one supported me more than my wife Claire, whose patience I challenged by opening my laptop most nights and on every holiday: for your unwavering love and cups of coffee I am forever grateful.

[1] who successfully convinced me I once was a student at Paris-Sud in the 90s, which I had no recollection of

[2] This includes Martine Thirion for her help in making the defence happen.

To all of those who are not computer scientists or logicians,
here is to inverted witch-hats and squash courts.

Table of Contents

Introduction

This dissertation concerns two fundamental ways in which mathematical proofs relate to computation: *proof-normalisation* and *proof-search*.

- The key idea, in the view of "computation as proof-normalisation", is that mathematical proofs can be composed in a modular way and that composed proofs can (sometimes) be "simplified" into *normal forms* by a normalisation procedure. The most well-known tool to compose proofs (though not the only one) is a specific reasoning step known as *cut* in the proof formalism of *Sequent Calculus* and known as *cut* or *detour* in the proof formalism of *Natural Deduction* [Gen35]. The normalisation process that turns proofs with cuts into cut-free proofs, known as *cut-elimination*, strongly relates to the computational paradigm of Functional Programming, as shown by the Curry-Howard correspondence [CF58, How80].
- The view of "computation as proof-search", on the other hand, considers a mathematical formula as the input of computation, and a proof of that formula as its output. This strongly relates to the computational paradigm of Logic programming, as described for instance by the seminal paper on *uniform proofs* [MNPS91].

Interestingly enough, investigating normal forms for proofs is useful for both views: for the former, to understand to which proofs all other proofs should reduce; for the latter, to only search for proofs in normal form and thus restrict the search space in efficient ways. For example, both kinds of computation are often taken to produce cut-free proofs (though not always).

In fact, two key concepts in the study of normal forms have proved useful for both views: *polarities* and *focussing*. In proof-search, they were used to design variants and generalisations of logic programming languages [AP89, And92, LM09]. In proof-normalisation, they were used to understand the semantics of, and design meaningful variants of, cut-elimination procedures [Lau02, LQdF05, MM09] (building on previous work [DJS95, DJS97]).

Roughly speaking, polarities and focussing generalise the idea that a formula of the form

$$\forall x_1 \exists y_1 \forall x_2 \exists y_2 \ldots \forall x_i \exists x_i \ldots$$

suggests a two-player game: the opponent gets to choose x_1, and depending on x_1 the proponent gets to choose y_1, after which the opponent gets to choose x_2, etc until some criterion (determined by the final '$\ldots$') decides who has won, given all the choices that have been made. Of course, what the exact rules of the game are, what a winning strategy is, etc depends on the logic considered and its proofs (for instance in classical logic, one can backtrack on a previous choice, using the input of the adversary). Polarities and focussing generalise this idea to all connectives (not only quantifiers), with some *positive connectives* "corresponding to" proponent's moves and *negative connectives* "corresponding to" opponent's moves.

The range of fields that build on, or benefit from, the two computational aspects of mathematical proofs, is broad. This dissertation engages in two of them which may seem distant from each other: program semantics and theorem proving. More specifically, it proposes the use of polarities and focussing as the core concepts to approach a field of topics ranging from *realisability semantics* to *automated reasoning*. We can briefly illustrate how polarities impact those two areas.

Realisability semantics (see e.g. [Kle45, VO02]) is a way to interpret a mathematical formula (in a broad sense, including a program type) as a *specification* that an object of a certain kind (such as a computer program or a mathematical proof) may or may not satisfy. This interpretation as specifications (i.e. what it means for an object to satisfy them) is defined by induction on the syntax of formulae, and refers to the object either by its internal structure or by the way it behaves when placed in a well-chosen environment. Realisability semantics have been studied for various logics and systems, and a particular approach emerged from classical logic and Girard's *linear logic* [Gir87], namely *orthogonality*. This approach is sometimes described as *classical realisability* [DK00, Kri01] (even though it may be used for other logics than classical logic).

This dissertation aims at the very essence of orthogonality-based realisability, by building an abstract semantics only based on polarities and focussing:

- if a formula starts with a positive connective, then the criterion determining whether an object satisfies the formula's specification refers primarily to the object's internal structure;
- if a formula starts with a negative connective, then the criterion refers to the object's behaviour when placed in a well-chosen environment.

Automated reasoning (see e.g. [RV01]) concerns the numerous algorithmic techniques by which the validity or the satisfiability of mathematical formulae can be determined. Since a formula is valid if and only if it has a proof, an obvious approach to automated reasoning is proof-search. The basic core of logic programming, for instance, can be understood as proof-search on *Horn clauses*, and in that respect it can be seen as a very specific area of automated reasoning. Now the reason why proof-search on Horn clauses also provides a meaningful computational paradigm is because this class of formulae makes a simple goal-directed proof-search strategy logically complete, with well-identified backtrack points and a reasonable covering of the proof-search space. This still holds when the class is extended to *hereditary Harrop formulae* [MNPS91], and can hold on a wider class of formulae if logical connectives (and atoms) are tagged with polarities:

- Negative connectives can be decomposed with *invertible* inference rules: a goal-directed proof-search strategy performs the bottom-up application of those rules as basic proof-search steps, without loss of generality;[3] in other words, no backtracking is necessary on the application of such steps, even though other steps were possible.
- Positive connectives are the (De Morgan's) duals of negative connectives, and their decomposition rules are not necessarily invertible, so a goal-directed proof-search procedure creates backtrack points when applying them bottom-up.

To what extent these ideas can be useful for a wider area of automated reasoning (than logic programming) remains a recent field, with numerous open questions but already with a

[3] If the goal was provable, it remains provable after applying the step.

couple of implementations available, such as Imogen [MP08] (using the *inverse method*) and Tac [BMS10] (using bottom-up proof-search). This dissertation explores a particular take on this, with its own implementation: PSYCHE [Psy].

This dissertation is therefore a journey through the above topics, trying to connect them with e.g. common formalisms. It is organised in three parts.

Part I of this dissertation is a short introduction to the adaptation, to classical logic, of the Curry-Howard correspondence, already mentioned above in the view of "computation as proof-normalisation". Also known as the "proofs-as-programs paradigm", the correspondence emerged with a strong flavour of constructive mathematics, so its adaptation to classical logic only emerged in the past 25 years [Gri90]. This part explores (some of) the contributions that have been made in that period, where we shall see the important roles of polarities and focussing. While it starts from Parigot's $\lambda\mu$-calculus [Par92] and ends with a Zeilberger-style system [Zei08a, Zei08b], this part mostly uses Curien and Herbelin's System L [CH00] as a common framework to express and connect the concepts pertaining to the computational interpretations of classical proofs.

Chapter 1 describes the basic set-up, viewing classical proofs as programs. In particular, we give an overview of how classical reasoning corresponds to the use of control operators [Rey72, SW00, Fel87] that let programs capture the contexts within which they are being evaluated. We show standard ways of building meaningful operational and denotational semantics for cut-elimination, which correspond to the Call-by-Name and Call-by-Value evaluation strategies in programming [Plo75], and to control and co-control categories in category theory [Sel01].

Chapter 2 explores the concepts and techniques based on orthogonality: orthogonality models form the classical version of realisability semantics [DK00, Kri01], as well as providing methodology to prove strong normalisation results [Par97, LM08], i.e. the termination of well-typed programs. We also illustrate another use of orthogonality models for extracting, out of a classical proof of an existential formula of arithmetic (more precisely, a Σ^0_1-formula), a term witnessing the existence; this technique is due to Miquel [Miq09, Miq11]. Out of orthogonality techniques we shall see the notion of polarity naturally emerge.

Chapter 3 formalises this concept, inspired by a discussion on η-conversion and observational equivalence. A new semantics for the evaluation of classical programs is inferred from the use of polarities (as in [MM09]), and three different notions of normal forms are identified, out of which the concept of focussing naturally emerges. The strongest version of focussing, namely system LKF [LM09], organises each proof into an alternation of phases (similar to the alternation between proponent's moves and opponent's moves in the intuitive view of the formula $\forall x_1 \exists y_1 \forall x_2 \exists y_2 \ldots \forall x_i \exists x_i \ldots$). The chapter then describes how each phase can be collapsed into one inference step, giving rise to a presentation of LKF in the style of "big-step focussing". It then describes the computational interpretation of this in terms of pattern-matching, along the lines of [Zei08a, Zei08b].

Part II of this dissertation takes this last idea further and presents new material. Stripping focussed systems off the concept of connective and off the inductive structure of mathematical formulae, we only keep the core mechanisms of focussing to define a highly abstract system for big-step focussing, called LAF, whose computational interpretation is pure pattern-matching.

One of the main goals is to formalise the strong ties between Zeilberger-style systems and orthogonality models.

Chapter 4 presents the syntax and the typing system of a quantifier-free version of the LAF abstract system, which is modular in its syntax for atoms and formulae, in its logical connectives, in the logic used, and in the implementation of variables. The chapter shows how the abstract system can be instantiated to capture existing focussed systems such as LKF and its intuitionistic variant LJF.

Chapter 5 presents the extension with quantifiers of that abstract focussed system LAF. Different approaches may lead to either a treatment of quantifiers along the lines of the ω-rule [Hil31, Sch50] (where a formula "$\forall n \in \mathbb{N}, A(n)$" may be proved by providing a proof for each natural number), or a treatment that forces to prove a universal formula in a uniform way: for this we need to extend LAF with a mechanism that generalises eigenvariables.

Chapter 6 presents the realisability models of the abstract system LAF, finally formalising the connection between big-step focussing and orthogonality models: indeed we lift the orthogonality models of Chapter 2 to our abstract framework, and we prove the Adequacy Lemma, that relates typing to realisability, i.e. syntax to semantics. We present instances of orthogonality models such that the Adequacy Lemma immediately provides the logical consistency of the LAF system.

Chapter 7 explores proof transformations in the abstract system LAF; we start with an abstract machine to perform head-reduction, thus revealing the actual pattern-matching mechanism of the proof-term calculus. Using the realisability models of Chapter 6, we show that head-normalisation terminates on typed proof-terms. We then describe, via a notion similar to that of free variables, how to identify the parts of a sequent that have actually been used in its proof (which is useful if the proof is to be re-used for a sequent that is similar). We also use this to extend the abstract machine into a big-step operational semantics that evaluates a proof-term as a normal form that is cut-free. Adapting the orthogonality model to this big-step operational semantics, we prove that every typed proof-term does evaluate as a cut-free form, and conclude the cut-elimination result for the LAF system.

Part III of this dissertation concerns the roles that polarities and focussing can have in automated reasoning and more generally theorem proving (i.e. proof construction may also be interactive). Originally aiming at classical logic (and therefore departing from the Imogen [MP08] and Tac [BMS10] provers), we investigated one of the most popular automated reasoning techniques for classical propositional logic (a.k.a. SAT-solving): DPLL [DP60, DLL62], as well as its extension known as DPLL($\mathcal{T}$) [NOT06] for solving SAT-modulo-theories problems (SMT).

Chapter 8 aims at describing and simulating DPLL($\mathcal{T}$) runs as bottom-up proof-search in a focussed system for classical logic. We therefore present a extension of system LKF that allows atoms to be assigned polarities *on-the-fly* during proof-search, and that integrates the possibility to call a procedure that decides whether a conjunction of (ground) atoms is consistent with a given input theory $\mathcal{T}$. The resulting system, LK$^p(\mathcal{T})$, is used to establish a bisimulation result between proof-search and DPLL($\mathcal{T}$) runs. Based on the fact that LKF can be seen as an instance of LAF, the chapter then discusses how LK$^p(\mathcal{T})$ could be seen as an instance of a generalisation of LAF that could work *modulo* the theory $\mathcal{T}$.

Chapter 9 describes a small prototype called PSYCHE [Psy] implementing bottom-up proof-search in an extension of LK$^p(\mathcal{T})$ with quantifiers and meta-variables. Highly modular with

respect to the decision procedure and the proof-search strategy it can run with, PSYCHE comes with a strategy *plugin* that implements the simulation of DPLL($\mathcal{T}$) described in Chapter 8, and can also perform pure first-order reasoning.

Chapter 10 concludes this dissertation, in particular by giving an informal description of PSYCHE's mechanisms for quantifiers, and hinting at what could be achieved with them, in particular in the combination of first-order reasoning with decision procedures. It finally presents the LAF system as the theoretical foundations for the next version of PSYCHE.

Note that the whole of Part II is admittedly technical, which is due to two reasons:

- The first reason is the systematic search for the greatest generality (and therefore strength) in the definitions and theorems. It was a goal in itself to determine exactly which ingredients are necessary and which are disposable for the system to make sense, for the models to be built, and for the theorems to be proved. Hypotheses are systematically weakened and structures are systematically parameterised to achieve this. The result of course is a highly parameterised framework with complex yet precise specifications. In order to digest this technicality with confidence, most of the proofs have been formalised [GL14] in the proof assistant Coq [Coq], which was particularly useful to refine the definitions and theorems according to the above methodology.
- The second reason is that the development of this abstract framework was not only done for the sake of it, but also to provide the foundations of (the next version of) our PSYCHE implementation. Abstraction in the theoretical framework translates to genericity in the code, making the implementation more versatile, decomposing its architecture into smaller modules that could more easily be shown to be correct. Therefore, when introducing as a mathematical structure a tuple such as
$$(\mathbb{S}, \mathsf{Lab}_e, \mathbb{T}, \Vdash, \mathbb{A}, \mathbb{M}, \equiv, \mathsf{Lab}_+, \mathsf{Lab}_-, \mathbb{R}, \mathsf{Co}, \mathsf{Pat}, \vdash)$$
satisfying a long list of axioms, we really have in mind an OCaml module providing the corresponding types and functions and satisfying the corresponding specifications. Hence the verbosity of our axiomatic structures in Part II.

Personal note

This section aims at relating this dissertation to the papers I have published in the recent years.

Firstly, this dissertation lies within the very broad field of *Computational Logic*, on which Didier Galmiche and I edited a special issue of the Journal of Logic and Computation, in honour of Roy Dyckhoff [GGL14].

My interest for the topics developed in this dissertation can be traced back to the first paper I wrote as the sole author [Len03], relating Curien and Herbelin's work on the computational interpretations of classical logic [CH00] to Urban's [Urb00].[4] However, such topics stayed in the slow-cooker at the back of my mind, as my next contributions mostly concerned intuitionistic systems, which could more simply be related to the λ-calculus, the Curry-Howard correspondence, and Type Theory:

In [KL08] we explored a Call-by-Name cut-elimination procedure for the intuitionistic sequent calculus, in [DL07] we explored the focussed sequent calculus LJQ, and in [Len08] I

[4] Despite its critical typos, it surprisingly appears to be my most cited paper.

proved some conjecture about the termination of a Call-by-Value λ-calculus. Although these contributions are not directly included in this dissertation, the work that I did around that time greatly contributed to my understanding of focussing and cut-elimination strategies.

Still in intuitionistic logic, two more recent contributions broached the topics that this dissertation approaches under the focussing angle, namely realisability and automated reasoning: In [BGL12] we develop a simple presentation of Hyland's effective topos [Hyl82] which is based on realisability concepts; in [LDM11] we developed a focussed sequent calculus that can describe proof-search in the type theory behind the proof-assistant Coq [Coq].

More directly included in this dissertation are the publications [LM08] and [BL11b, BL11c, BGL13], all of which formalise proofs of strong normalisation with orthogonality techniques, aiming at genericity. In [LM08] we compare Barbanera and Berardi's technique based on symmetric reducibility candidates [BB96] with the basic orthogonality technique. In [BL11b, BL11c, BGL13], we formalise an abstract notion of orthogonality model and describe, as instances of this notion, several variants of proofs for the strong normalisation of System F [Gir72].

Chapter 2 of this dissertation covers these contributions with a systematic orthogonality model construction. It also uses the same orthogonality framework to describe an interesting application of classical realisability to witness extraction (due to Miquel [Miq09, Miq11]).

Over the recent years, a greater proportion of my research was devoted to the use of focussing for proof-search, in the context of our ANR-funded project on *Proof-Search Control in Interaction with domain-specific methods* [PSI]. Since the concept of focussing emerged with motivations for logic programming, it was natural to explore whether the concept could also impact automated reasoning. We first explored propositional problem solving, which can be seen either as proof-search or as satisfiability (SAT) solving: More precisely, we described in [FLM12b, FLM12a, FGLM13] how one of the main procedures, namely DPLL [DP60, DLL62], can be seen as the gradual construction of proof-trees in a focussed sequent calculus. We actually did this in combination with decision procedures, so as to describe SMT-solving in terms of proof-search; this required the extension of sequent calculus with such procedures [FL11, FGL13]. All this was put together in Farooque's Ph.D. thesis [Far13] which I supervised, and where another class of automated reasoning techniques, namely *tableaux* methods, are also simulated in the same focussed sequent calculus.

In the present dissertation, the above contributions are not developed in as many details as in [Far13]. However, they form the theoretical basis of the PSYCHE prototype [Psy], of which I am the main developer; and the software is the topic of Chapter 9, which covers the system description [GL13].

The material presented in this dissertation does not only come from publications. Some of it relates to an active teaching activity: in particular, Part I approximately covers the material that I teach at M.Sc. level in Paris, with the most advanced parts inspired by Munch-Maccagnoni's work relating focussing and classical realisability [MM09], and Zeilberger's work on big-step focussing and pattern-matching interpretations [Zei08a, Zei08b].

Part II presents entirely new material, rather than published work (or survey thereof), that builds on those two inspirational topics: Zeilberger's framework seemed particularly appropriate to relate focussing and classical realisability at a particulary abstract level. The proposal is to make this the theoretical foundation of PSYCHE's next version, and in that it connects to Part III this dissertation.

On the other hand, several publications do not (yet) relate (or only very remotely) to this dissertation, since they are too disconnected from its topic: In [GL08, GL09], we developed a λ-calculus inspired by Nominal Logic [Pit03], with a special construct in order to represent binding in data-structures; the goal is to allow incomplete terms within the scope of binders, without blocking α-conversion or computation. In [BL11a] we studied intersection type systems [CD78] for the λ-calculus, in a non-idempotent version similar to de Carvalho's [dC05, dC09], but such that the length of the longest β-reduction sequence starting from a strongly normalising term, can be directly read from its typing tree.

Finally, the careful reader will note that this dissertation not only has little material in common with my Ph.D. thesis [Len06], but it is not even in its direct continuation: I do not present here refinements or developments of its contributions, but rather a thesis that complements my doctoral work in the topics of my interest.

Notations and prerequisites

In this dissertation, we assume the reader to be already familiar with some areas and concepts of logic and computer science. Unless specifically given, the notations and definitions used in this dissertation are rather standard, and formally follow [Len06]. The areas and concepts are:

- Set theory; see e.g. [Kri71].
  In particular we will use the concepts of, and notations for, subsets, power sets, union, intersection and difference of sets, relations, functions, injectivity, surjectivity, etc. Our notation for the power set of A is $\mathbb{P}(A)$. Our notation for the set of total functions from A to B is denoted $A \to B$; the set of partial functions from A to B is denoted $A \rightharpoonup B$. We also assume the reader to be familiar with natural numbers, lists and trees.
- The standard difference between object-level and meta-level.
  In particular, variables of the meta-level are called meta-variables and (unless otherwise stated) "rules" and "systems" are meta-level devices (i.e. a rule has no existence at the object level, but its instances do -and the collection of them, for example).
- Trees and derivations.
  We use inference rules and systems to define sets of (valid) derivations and derivability of judgements, as well as partial derivations; when we state that a rule is derivable/admissible/invertible (in a system) we actually mean that its instances are derivable/admissible/invertible (in the collection of derivations defined by the system).
- Rewriting (first-order and higher-order); see e.g. [Ter03].
  In particular, the notations $\longrightarrow^n$, $\longrightarrow^+$, $\longrightarrow^*$, $\longleftrightarrow^*$, denote the composition n times of a (binary) relation $\longrightarrow$, the transitive closure, the transitive and reflexive closure, and the transitive, reflexive and symmetric closure, of the relation $\longrightarrow$, respectively.

  We assume that the reader is familiar with the properties of confluence and Church-Rosser, weak normalisation, strong normalisation, and the usual techniques to prove them, in particular the simulation techniques.

  Following [Len06], the notation

  $$(\gamma) \quad M \longrightarrow N$$

  introduces a rewrite rule whose contextual closure (or more precisely, the contextual closure of its instances) is denoted $M \longrightarrow_\gamma N$. We also use this notation when γ is a system of rules.

  Our languages will often be made of terms whose syntax is defined by a BNF-grammar. Some of its syntactic categories may contain *variables*. We assume the reader is familiar with variable binding, α-conversion and *equivariance*; specifying binders and their scopes automatically defines what the *free variables* of a term, denoted $\mathsf{FV}(t)$, are; *capture-*

avoiding substitution of u for x in t is denoted

$$\{^{u}/_{x}\}\, t$$

where x is a variable of some syntactic category with variables and u is a term of that syntactic category.

- Basic proof theory; see e.g. [TS00]. In particular, standard proof formalisms such as Natural Deduction and Sequent Calculus, for intuitionistic and classical logic (propositional and first-order).

Part I

The Curry-Howard view of classical logic - a short introduction

Chapter 1

Classical proofs as programs

Contents

The Curry-Howard correspondence [CF58, How80] has been one of the most fruitful connections between proofs and computation: As one of the embodiments of constructivism, where mathematical proofs bear computational content, the correspondence naturally emerged in the context of minimal and intuitionistic logic, and gave rise to the field of Type Theory [ML82, ML84].

Despite the non-constructive character of proofs in classical logic, arising from the Law of Excluded Middle, or the Double Negation Elimination etc, it is natural to investigate what part of the Curry-Howard correspondence can still be built for that logic.

In this chapter we review the foundations of the correspondence in the framework of classical logic, along the main lines of investigation that were explored over the past 25 years since Griffin's seminal work [Gri90].

The first step in this programme is to turn a proof format for classical logic into a typing system for a language. For such a language to be of computational nature, an operational semantics and/or a denotational semantics has to be designed.

Section 1.1 reviews the basic concepts of the Curry-Howard correspondence, both in their original framework and at a more abstract level. Section 1.2 present some concepts in programming, namely continuations and control, which will prove useful to understand classical proofs as programs. Section 1.3 presents early formalisations of the above concepts as proof-term calculi for classical logic while Section 1.4 presents in more details one of the most convenient ones, which relates to Gentzen's classical sequent calculus. Section 1.6 uses continuations to describe the evaluation strategies known as Call-by-Name and Call-by-Value while Section 1.7 explains how this can be used to build semantics for classical proofs.

1.1 Curry-Howard correspondence: concepts and instances

The correspondence relates logic to programming languages, and is sometimes taken to involve a third aspect, namely category theory (as it forms a popular framework to build the semantics of programming languages). Table 1.1 gives a high-level view of the correspondence,[1] which operates at several levels: mathematical formulae, or propositions, correspond to the types of a given programming language; proofs of such propositions correspond to programs that can be given the corresponding type; the way proofs can be composed corresponds to the way programs can be composed/applied; finally (and this is where we adopt the view of computation as proof-normalisation), cut-elimination corresponds to program execution.

Logic	**Programming language**	**Categories**
Propositions	Types	Objects
Proofs	Typed programs	Morphisms
Cut/Composition	Program composition	Morphism composition
Cut-elimination	Program execution	Equality of morphisms (commuting diagrams)

Table 1.1: High-level view of the Curry-Howard correspondence

The rest of this section gives a brief overview of the correspondence in the framework of minimal and intuitionistic logic. An in-depth presentation of the correspondence can be found in the book [SU06].

1.1.1 Simply-typed combinators

The original instance of the correspondence was given in the study of combinators [CF58], which yields a simple language made of three basic programs **I**, **K**, **S** and with program application as its only construct:

DEFINITION 1 (The (I, K, S)-combinatoric system)

The syntax is given by the following grammar:

$$M, N, \ldots ::= \mathbf{I} \,|\, \mathbf{K} \,|\, \mathbf{S} \,|\, M\ N$$

The last construct, *program application*, is associative to the left, i.e. $(M\ N)\ P$ can be abbreviated as $M\ N\ P$.

[1] We use the expression (Curry-Howard) "correspondence" rather than the popular (Curry-Howard) "isomorphism", as it is difficult to specify what the isomorphism exactly is before specifying exactly what formal systems we intend to relate.

and its operational semantics is given by the following first-order rewrite system

$$\begin{array}{lcl} \mathbf{I}\ M & \longrightarrow & M \\ \mathbf{K}\ M\ N & \longrightarrow & M \\ \mathbf{S}\ M\ N\ P & \longrightarrow & M\ P\ (N\ P) \end{array}$$

※

Clearly, the operational semantics defines **I** as the identity, while **K** provides erasure and **S** provides duplication. The reduction relation is confluent and defines a model of computation that turns out to be Turing-complete. At the cost of losing that property, the language can be given an intuitive typing system, using *simple types*:

Definition 2 (Simple types) *Simple types* are defined by the following grammar:

$$A, B, \ldots ::= a \mid A \to B$$

where a ranges over a fixed set of elements called *atomic types*. The symbol $\to$ is associative to the right, i.e. $A \to (B \to C)$ can be abbreviated as $A \to B \to C$. ※

The typing system is defined as follows:

Definition 3 (Simple types for the (I, K, S)-combinatoric system)

Typing is a binary relation between terms and simple types, denoted with expressions such as $\vdash M : A$. That relation is defined for the combinators as follows:

$$\begin{array}{l} \vdash \mathbf{I} : A \to A \\ \vdash \mathbf{K} : A \to B \to A \\ \vdash \mathbf{S} : (A \to (B \to C)) \to (A \to B) \to (A \to C) \end{array}$$

and program application is typed by the following rule:

$$\frac{\vdash M : A \to B \qquad \vdash N : A}{\vdash M\ N : B}$$

Derivability of the typing statement $\vdash M : A$, from the above axioms and using the program application rule, is denoted $\vdash_{\mathsf{stC}} M : A$. ※

The reduction defined by the rewrite system preserves types, a property called Subject Reduction:

Theorem 1 (Subject reduction for simply-typed combinatoric system)

If $\vdash M : A$ and $M \longrightarrow N$ then $\vdash N : A$. ※

Proof: By induction on the derivation of $M \longrightarrow N$, with the base cases corresponding to the 3 rewrite rules themselves. □

The essence of the Curry-Howard correspondence, is the simple remark that, viewing the functional type construct $\to$ as the logical symbol for implication, simples types are isomorphic[2] to the syntax of formulae for propositional minimal logic [Joh36] and that typing derivations are isomorphic to proofs in a particular Frege-Hilbert system [Fre79, Hil28] for minimal logic.

[2]The isomorphism with simple types assumes that atomic types are isomorphic to atomic formulae.

DEFINITION 4 (Propositional minimal logic)

Formulae of minimal logic are defined by the following grammar:

$$A, B, \ldots ::= \; a \mid A \Rightarrow B$$

where a ranges over a fixed set of elements called *atomic formulae*.

Proofs are the derivations built with the *Modus Ponens* rule

$$\frac{\vdash A \Rightarrow B \qquad \vdash A}{\vdash B}$$

from the axioms:

$$A \Rightarrow A$$
$$A \Rightarrow B \Rightarrow A$$
$$(A \Rightarrow (B \Rightarrow C)) \Rightarrow (A \Rightarrow B) \Rightarrow (A \Rightarrow C)$$

The symbol $\Rightarrow$ is associative to the right.

Derivability of $\vdash A$, from the above axioms and using the program application rule, is denoted $\vdash_{\mathsf{FH}\Rightarrow} A$. ※

Via the correspondence, the Subject Reduction property allows the view of the reduction relation as a proof-transforming procedure.

1.1.2 Simply-typed λ-calculus

Curry's view about minimal logic was extended by Howard to intuitionistic first-order arithmetic [How80]. A different format was also proposed, both for proofs and for programs, and became perhaps the most popular setting for the Curry-Howard correspondence: Natural Deduction [Gen35] was the formalism used for proofs, and the λ-calculus [Chu41] was the formalism used for programs. This instance of the Curry-Howard correspondence that we present is a version of natural deduction using *sequents* and a version of the simply-typed λ-calculus using typing contexts.

DEFINITION 5 (λ-calculus) The syntax of the λ-calculus is given by the following grammar:

$$M, N, \ldots ::= \; x \mid \lambda x.M \mid M \; N$$

where x ranges over a denumerable set of *variables*, and the construct $\lambda x.M$ binds x in M.[3]

Standard conventions are used for parentheses [Bar84]: the scopes of binders extend as much as parentheses allow (i.e. $\lambda x.M \; N$ abbreviates $\lambda x.(M \; N)$); program application is associative to the left (i.e. $M \; N \; P$ abbreviates $(M \; N) \; P$); moreover, binder can be grouped, so that $\lambda xy.M$ abbreviates $\lambda x.\lambda y.M$.

The following rewrite rules

$$\begin{array}{lll} (\beta) & (\lambda x.M) \; N & \longrightarrow \; \left\{ {}^{N}\!/_{x} \right\} M \\ (\eta) & \lambda x.M \; x & \longrightarrow \; M \qquad \text{if } x \notin \mathsf{FV}(M) \end{array}$$

define the reduction relations $\longrightarrow_\beta$, $\longrightarrow_\eta$ and $\longrightarrow_{\beta\eta}$. ※

As for the combinatoric system from Section 1.1.1, the reduction relations are confluent:

THEOREM 2 (Confluence) $\longrightarrow_\beta$, $\longrightarrow_\eta$ and $\longrightarrow_{\beta\eta}$ are confluent. ※

[3] As mentioned in the section about notations, specifying binders and their scopes automatically defines free variables, α-conversion, capture-avoiding substitution, etc.

Proof: See for instance [Bar84]. □

Definition 6 (Simply-typed λ-calculus) *Typing contexts* are finite maps from variables to simple types, with () denoting the empty context (sometimes the notation () is completely omitted), Γ, Γ' denoting the union of contexts Γ and Γ' (assuming it is defined), and $x:A$ denoting the singleton context mapping variable x to the simple type A.

The typing rules of the simply-typed λ-calculus are given in Fig. 1.

Derivability of the typing statement $\Gamma \vdash M:A$ in that system is denoted $\Gamma \vdash_{\mathsf{st}\lambda} M:A$. ※

$$\frac{}{\Gamma, x:A \vdash x:A} \qquad \frac{\Gamma, x:A \vdash M:B}{\Gamma \vdash \lambda x.M : A \to B} \qquad \frac{\Gamma \vdash M : A \to B \quad \Gamma \vdash N : A}{\Gamma \vdash M\ N : B}$$

Figure 1: Simply-typed λ-calculus

As for the combinatoric system from Section 1.1.1, the reduction relations satisfy Subject Reduction:

Theorem 3 (Subject reduction for simply-typed λ-calculus)

1. If $\Gamma, x:A \vdash M:B$ and $\Gamma \vdash N:A$ then $\Gamma \vdash \left\{ {}^{N}/_{x} \right\} M:B$.
2. If $\Gamma \vdash M:A$ and $M \longrightarrow_{\beta\eta} N$ then $\Gamma \vdash N:A$.

※

Proof: See for instance [Bar84]. □

Our second instance of the Curry-Howard correspondence relates the simply-typed λ-calculus with the Natural Deduction system $\mathsf{NJ}_{\Rightarrow}$ for minimal logic.

Definition 7 (Natural Deduction for minimal logic - $\mathsf{NJ}_{\Rightarrow}$)

System $\mathsf{NJ}_{\Rightarrow}$ is the inference system given in Fig. 2, where

- A, B range over formulae of minimal logic;
- Γ stands for a "collection" of formulae. By collection we mean either set or multiset,[4] with Γ, Γ' denoting the union of Γ and Γ', A denoting either the formula A itself or the singleton $\{A\}$ (or $\{\!\{A\}\!\}$), while the empty set (or multiset) is sometimes ommitted;
- $\Gamma \vdash A$ is a structure called *sequent*.

Derivations in that system are called *proofs* in $\mathsf{NJ}_{\Rightarrow}$.

Derivability in $\mathsf{NJ}_{\Rightarrow}$ of a sequent $\Gamma \vdash A$ is denoted $\Gamma \vdash_{\mathsf{NJ}_{\Rightarrow}} A$. ※

$$\frac{}{\Gamma, A \vdash A} \qquad \frac{\Gamma, A \vdash B}{\Gamma \vdash A \Rightarrow B} \qquad \frac{\Gamma \vdash A \Rightarrow B \quad \Gamma \vdash A}{\Gamma \vdash B}$$

Figure 2: Natural Deduction for minimal logic - $\mathsf{NJ}_{\Rightarrow}$

[4] That choice will change the number of proofs of a given formula.

Comparing Fig. 1 and Fig. 2 reveals our second instance of the Curry-Howard correspondence, although the exact meaning of the word 'correspondence' is in this case more subtle than for our first instance:

Clearly, the bijective aspect that pertains to the word "isomorphism" is jeopardised as $\dfrac{}{x:A, y:A \vdash x:A}$ and $\dfrac{}{x:A, y:A \vdash y:A}$ are clearly two distinct typing derivations which would both 'correspond to' the proof $\dfrac{}{A, A \vdash A}$ (whether we use sets or multisets). Moreover, binding introduces an ambiguity in the way we count typing derivations: is there one or infinitely many derivations of $\vdash \lambda x.x : A \to A$?[5]

For this reason, this dissertation takes the view that the interesting aspects of the Curry-Howard correspondence do not include the bijective aspect of an encoding from one system into another, but rather its compositionality (for trees), and the soundness and completeness properties:

In the present case, the forgetful encoding that maps every typing derivation to a proof is compositional with respect to the tree-structure of derivations; its surjectivity provides completeness of type inhabitation -whether there exists a λ-term of a given type- with respect to the provability of the corresponding formula; soundness is simply the fact that the tree obtained by forgetting variables and terms from a typing derivation is a correct proof.

These are the properties that we will aim at when investigating the variants of the Curry-Howard correspondence.

As for the combinatoric system from Section 1.1.1, the Subject Reduction property allows the view of the reduction relations $\longrightarrow_\beta$, $\longrightarrow_\eta$ and $\longrightarrow_{\beta\eta}$ as proof-transforming procedures.

In summary, the most well-known settings for the Curry-Howard correspondence are:

Frege-Hilbert system	$\leftrightarrow$	Combinators (S,K,I)	[CF58]
Natural Deduction	$\leftrightarrow$	Typed λ-terms	[How80]

1.1.3 The categorical aspect

We now briefly mention what is sometimes considered a third aspect of the Curry-Howard corespondence, in category theory.

Categories can be used to shed a semantical light on the Curry-Howard correspondence. In our case, a particular kind of category provides models of the simply-typed λ-calculus: cartesian closed categories (CCC). In brief, CCC feature a terminal object, products, and exponential objects. We start with a few notational conventions:

NOTATION 8 (Category)

The class of morphisms from object A to object B is denoted $\mathsf{hom}(A, B)$, and the expression $f\colon A \longrightarrow B$ denotes that f is a morphism from A to B. Identity morphisms are denoted Id_A, and the composition of $f\colon A \longrightarrow B$ and $g\colon B \longrightarrow C$ is denoted $f \cdot g\colon A \longrightarrow C$.

[5] Formally, the typing system allows infinitely many premisses for that typing judgement, depending on the variable that we pick to place in the typing context with type A.

In a cartesian closed category (CCC),

- the *terminal object* is denoted 1, with morphisms $1_A : A \longrightarrow 1$
- the *product* of A and B is denoted $A \times B$, with projections denoted $\pi_1 : A \times B \longrightarrow A$ and $\pi_2 : A \times B \longrightarrow B$ (and more generally, the i^{th} projection from n objects is denoted $\pi_{i/n}$) and morphism pairing denoted $\langle f_1, f_2 \rangle : C \longrightarrow A \times B$ for every $f_1 : C \longrightarrow A$ and $f_2 : C \longrightarrow B$.
- the *exponential* of A and B is denoted B^A, with morphisms $\mathsf{eval} : B^A \times A \longrightarrow B$ and a currified morphism $\Lambda g : X \longrightarrow B^A$ for every $g : X \times A \longrightarrow B$.

※

For a formal definition of the above concepts, see Appendix A.

Definition 9 (Semantics of the simply-typed λ-calculus in a CCC)

Consider a cartesian closed category.

Consider a mapping that interprets every atomic type a as an object $[\![a]\!]$ of the CCC, and extend it to all simple types by defining $[\![A{\to}B]\!]$ as $[\![B]\!]^{[\![A]\!]}$.

Consider a total order on the λ-calculus variables; when writing a typing context as $x_1 : A_1, \ldots, x_n : A_n$ we now follow the convention that $x_1, \ldots, x_n$ is an increasing sequence; we then define the semantics of any typing context by

$$[\![x_1 : A_1, \ldots, x_n : A_n]\!] := 1 \times [\![A_1]\!] \times \cdots \times [\![A_n]\!]$$

The semantics of a typing derivation π for the typing judgement $\Gamma \vdash M : A$ is defined according to Fig. 3, by induction on π, as a morphism $[\![\pi]\!] : [\![\Gamma]\!] \longrightarrow [\![A]\!]$. ※

$$\left[\!\!\left[\frac{}{x_1 : A_1, \ldots, x_n : A_n \vdash x_i : A_i} \right]\!\!\right] := \pi_{i+1/n+1} : 1 \times [\![A_1]\!] \times \cdots \times [\![A_n]\!] \longrightarrow [\![A_i]\!]$$

$$\left[\!\!\left[\frac{\begin{array}{c}\vdots\,\pi \\ \Gamma, x : A \vdash M : B\end{array}}{\Gamma \vdash \lambda x.M : A{\to}B} \right]\!\!\right] := \Lambda g : [\![\Gamma]\!] \longrightarrow [\![B]\!]^{[\![A]\!]}$$

$$\text{where} \quad g = \left[\!\!\left[\begin{array}{c}\vdots\,\pi \\ \Gamma, x : A \vdash M : B\end{array} \right]\!\!\right] : [\![\Gamma]\!] \times [\![A]\!] \longrightarrow [\![B]\!]$$

$$\left[\!\!\left[\frac{\begin{array}{cc}\vdots\,\pi_1 & \vdots\,\pi_2 \\ \Gamma \vdash M : A{\to}B & \Gamma \vdash N : A\end{array}}{\Gamma \vdash M\ N : B} \right]\!\!\right] := \langle g_1, g_2 \rangle \cdot \mathsf{eval} : [\![\Gamma]\!] \longrightarrow [\![B]\!]$$

$$\text{where} \quad g_1 = \left[\!\!\left[\begin{array}{c}\vdots\,\pi_1 \\ \Gamma \vdash M : A{\to}B\end{array} \right]\!\!\right] : [\![\Gamma]\!] \longrightarrow [\![B]\!]^{[\![A]\!]}$$

$$\text{and} \quad g_2 = \left[\!\!\left[\begin{array}{c}\vdots\,\pi_2 \\ \Gamma \vdash N : A\end{array} \right]\!\!\right] : [\![\Gamma]\!] \longrightarrow [\![A]\!]$$

Figure 3: Semantics of the simply-typed λ-calculus in a CCC

Note that in the case of a λ-abstraction, we assume that x is (strictly) greater than any

variable in Γ. Any derivation in which this is not the case can easily be turned into one satisfying this condition: variables can always be renamed[6] so that they are introduced in the typing context in increasing order.[7]

Now as mentioned earlier, we can relate the reductions in the simply-typed λ-calculus to the equality of morphisms in CCC:

THEOREM 4 (Soundness and completeness)

Assume $\Gamma \vdash_{\mathsf{st}\lambda} M : A$ (with derivation π) and $\Gamma \vdash_{\mathsf{st}\lambda} N : A$ (with derivation π').
$M \longleftrightarrow^*_{\beta\eta} N$ if and only if in every CCC we have $[\![\pi]\!] = [\![\pi']\!]$. ※

The equality theorems that can be derived from the axioms of CCC (and that therefore hold in every CCC) are reflected syntactically in the simply-typed λ-calculus, and the simply-typed λ-calculus is therefore said to form an *internal language* for CCC.

Note that it is easy to define the semantics of the (**I**, **K**, **S**)-combinatoric system (with simple types) in a CCC, for instance by encoding combinators as simply-typed λ-terms:

THEOREM 5 (Semantics of the (I, K, S)-combinatoric system)

The encoding of Fig. 4 satisfies satisfies the following properties:

- If $\vdash_{\mathsf{stC}} M : A$ then $\vdash_{\mathsf{st}\lambda} \overline{M} : A$, with a function $\pi \mapsto \overline{\pi}$ transforming a derivation of the former into a derivation of the latter.
- If $M \longrightarrow M'$ then $\overline{M} \longrightarrow_\beta M'$.

The above properties allow the definition of the semantics $[\![\pi]\!]$ of a typing derivation π of $\vdash_{\mathsf{stC}} M : A$ as the morphism $[\![\overline{\pi}]\!] : 1 \longrightarrow [\![A]\!]$, such that the following holds:
If $\pi \longleftrightarrow^* \pi'$ then $[\![\pi]\!] = [\![\pi']\!]$. ※

$$
\begin{array}{lcl}
\overline{\mathsf{I}} & := & \lambda x.x \\
\overline{\mathsf{K}} & := & \lambda xy.x \\
\overline{\mathsf{S}} & := & \lambda xyz.x\ z\ (y\ z) \\
\overline{M\ N} & := & \overline{M}\ \overline{N}
\end{array}
$$

Figure 4: (**I**, **K**, **S**)-combinators as λ-terms

1.1.4 Applying the methodology to other systems

The approach of the Curry-Howard correspondence can be, and has been, generalised with the following methodology:

- The first step is to decorate proofs with proof-terms: $\Gamma \vdash A$ becomes $\Gamma' \vdash M : A$, with Γ' being a typing context whose co-domain (i.e. the types which have been assigned to variables) is Γ;

[6] Using equivariance of typing derivations.

[7] Alternative presentations of the simply-typed λ-calculus may be more convenient to define its semantics in a CCC: the use of De Bruijn indices (see e.g. [Bar84]) instead of named variables provides a natural way of ordering the objects in the interpretation of a typing environment (without resorting to ordering the set of variables); if variables carry their own type (or when each type comes with its own set of variables), the interpretation can be defined on the terms themselves rather than their typing derivations.

- the second is to express proof transformations in terms of proof-term reduction, denoted $M \longrightarrow_{\mathcal{S}} N$, often given by a rewrite system $\mathcal{S}$.

The desired properties of reduction are

- *Progress*, i.e. any term containing "undesirable structures" can be reduced.
- *Subject reduction* property, i.e. preservation of typing:
  If $\Gamma \vdash M : A$ and $M \longrightarrow_{\mathcal{S}} N$ then $\Gamma \vdash N : A$
- possibly *Confluence*, programs are deterministic.
- possibly *Normalisation*, i.e. the fact that the execution of programs terminates.

The notion of "undesirable structures" is of course one of the concepts to identify in an interesting way; for instance in the simply-typed λ-calculus, a structure of the form $(\lambda x.M)\ N$ corresponds to the introduction of implication followed by the elimination of the introduced implication, a situation which we may consider undesirable from a proof-theoretic point of view.

To illustrate this methodology, we show how the correspondence from Section 1.1.2 can be extended to intuitionistic logic with both the implication connective and the logical constant $\bot$.

First note that by identifying $\bot$ simply as one of the atomic formulae, intuitionistic negation can be defined as follows: $\neg A := A \Rightarrow \bot$. With this definition, the following rules are instances of those of the simply-typed λ-calculus:

$$\frac{\Gamma, x : A \vdash M : \bot}{\Gamma \vdash \lambda x.M : \neg A} \qquad \frac{\Gamma \vdash M : \neg A \quad \Gamma \vdash N : A}{\Gamma \vdash M\ N : \bot}$$

and these reflect the usual Natural Deduction rules for negation, but what is missing, to have intuitionistic logic, is the rule named *Ex falso quodlibet* (EFQ):

$$\frac{\Gamma \vdash \bot}{\Gamma \vdash A}$$

We now see what that rule may become in the Curry-Howard correspondence.

Example 1 (Extension of the Curry-Howard correspondence for $\mathbf{NJ}_{\Rightarrow,\bot}$)

We extend the syntax of the λ-calculus with the following construct:

$$M, N, \ldots ::= \ \ldots \ \big| \ \mathsf{abort}(M)$$

and we add to the simply-typed λ-calculus a typing rule corresponding to EFQ:

$$\frac{\Gamma \vdash M : \bot}{\Gamma \vdash \mathsf{abort}(M) : A}$$

We may then add to the λ-calculus a rule such as

$$(\flat) \quad \mathsf{abort}(M)\ N \longrightarrow \mathsf{abort}(M)$$

to computationally interpret the new construct as a greedy consumer of arguments, and $\longrightarrow_\flat$, $\longrightarrow_{\beta\flat}$, $\longrightarrow_{\eta\flat}$, and $\longrightarrow_{\beta\eta\flat}$ are all confluent.

With these definitions, we still have Subject Reduction:
If $\Gamma \vdash M : A$ and $M \longrightarrow_{\beta\eta\flat} N$ then $\Gamma \vdash N : A$.

We can also interpret EFQ in category theory by requiring from a CCC the extra axiom that there is an *initial object* $\bot$, i.e. an object such that, from every object A there is a unique morphism $0_A : \bot \longrightarrow A$ (which is the dual of the terminal object 1 of the CCC). ※

From there, it is natural to try to extend the above correspondence to classical logic, which can be obtained from intuitionistic logic by adding any one of the three axiom schemes:

Elimination of double negation (EDN):	$(\neg\neg A)\Rightarrow A$
Peirce's law (PL):	$((A\Rightarrow B)\Rightarrow A)\Rightarrow A$
Law of excluded middle (LEM):	$A \vee \neg A$

In presence of EFQ (in short, the axiom scheme $\bot\Rightarrow A$), the above schemes are all equivalent in terms of formula provability. Interestingly enough and as noted in [AH03], without EFQ, we only have the following implications between the schemes:

$$\text{EDN}\Rightarrow\text{PL}\Rightarrow\text{LEM}$$
$$\text{EDN}\Rightarrow\text{EFQ}$$

Alternatives to adding axiom schemes is to add inference rules such as

$$\frac{\Gamma, \neg A \vdash \bot}{\Gamma \vdash A} \text{ for EDN}$$

or even to change the structure of the proof formalism, for instance by using the classical sequent calculus [Gen35] with *right-contraction*.

When thinking about classical logic, we have a tendency to identify a formula A with $\neg\neg A$, as suggested not only by the elimination of double negation but also by models of classical provability in boolean algebras.

Now, attempts to apply the Curry-Howard methodology to, say, the above axiom schemes or inference rule, are limited by the following fact:

A CCC with initial object $\bot$ and such that every object A is naturally isomorphic to $\bot^{\bot^A}$, collapses to a boolean algebra: there is at most 1 morphism between any 2 objects (see the proof in [LS86] or [Str11]).

That means that such a category would not distinguish two proofs of the same theorem, which is rather useless for a theory of proofs, or for the proofs-as-programs paradigm.

At that point, the natural question to ask is whether classical logic has computational content? To that question, and based on the above remarks, the book *Proofs and Types* [GTL89] answers in 1989:

"[The Curry-Howard] interpretation is not possible with classical logic: there is no sensible way of considering proofs as algorithms. In fact, classical logic has no denotational semantics, except the trivial one which identifies all the proofs of the same type."

In the rest of this chapter we explore the alternative answers that have been given to the question since then.

1.2 Continuations and control

For this we start with some concepts which at first sight may seem unrelated: continuations and control.

A freeze-frame shot taken at one point of a program's execution flow could be represented, in a high-level view, as follows:

$\downarrow$	code P that has been executed, producing data v
v	its output
$\downarrow$	code E that remains to be executed, consuming data v

The code E that remains to be executed, and more generally the programming environment or programming context within which some code is executed, is called *continuation*.

The concept is also useful for compiling recursive calls: consider the following pseudo code

```
myfunction(a1,...,an){
  some code;
  x = myfunction(a1',...,an');
  some code possibly using x;
}
```

When executing the recursive call, the code that remains to be executed (i.e. `some code possibly using x`), together with the values of the local variables, needs to be stored in order to resume computation after the recursive call has returned with a value for `x`. But this is not needed in the case of *tail recursion*, in which `some code possibly using x` just returns (the value for) `x`.

The above code can be transformed into a tail-recursive code by modelling the remaining code `some code possibly using x` as a "continuation" function `c'` taking the value of `x` as input:

```
myfunction(a1,...,an,c){
  some code;
  return myfunction(a1',...,an',c');
}
```

Now we see what these concepts become in the case where the programming language is the λ-calculus. An instance of the above program execution flow picture

$\downarrow$ code P that has been executed, producing data v
v its output
$\downarrow$ code E that remains to be executed, consuming data v

can be seen by considering

- P to be a λ-term that is reduced,
- v to be the value to which P reduces,
- E to be the context, in the syntactic sense: a term with a hole $E[\ ]$ (with the original λ-term being $E[P]$, i.e. the context $E[\ ]$ whose hole has been filled with the λ-term P).

This is only a general idea: whether that view accurately describes program execution depends on the evaluation strategy for λ-terms; in particular, whether λ-terms are reduced inside-out, what notion of "value" is considered (is it a normal form?), what grammar for contexts $E[\ ]$ ranges over, etc.

But in pure λ-calculus, it is clear that P has no knowledge of $E[\ ]$ while being evaluated.

Control is about letting a program know and manipulate its evaluation context. Originally, the concept was used to model `goto` instructions, and other features that are not pure functional programming.

In the case of λ-calculus, the evaluation context $E[\ ]$ within which a term is evaluated gives rise to a continuation function $\lambda x.E[x]$ (for a fresh variable x) that could be passed as an argument.

Reynolds [Rey72], Strachey-Wadsworth [SW00] (re-edition of 74) explored continuations and control along those lines, letting a program capture its evaluation context with a feature known as *call-with-current-continuation* (call-cc): `cc`. This was added to the programming language `Scheme`.

Felleisen's PhD work [Fel87] on the Syntactic Theory of Control introduced another control operator: $\mathcal{C}$.

The general idea of these control operators is given by the following reduction rules:

$$\begin{array}{lcl} E[\mathsf{cc}\ M] & \longrightarrow & E[M\ (\lambda x.E[x])] \\ E[\mathcal{C}\ M] & \longrightarrow & M\ (\lambda x.E[x]) \end{array}$$

In presence of $\mathsf{abort}(\ )$ and its (slightly modified) rule

$$E[\mathsf{abort}(M)] \longrightarrow M$$

cc and $\mathcal{C}$ are interdefinable:

$$\begin{array}{lcll} \mathcal{C}\ M & := & \mathsf{cc}\ (\lambda k.\mathsf{abort}(M\ k)) & k \notin \mathsf{FV}(M) \\ \mathsf{cc}\ M & := & \mathcal{C}\ (\lambda k.k\ (M\ k)) & k \notin \mathsf{FV}(M) \end{array}$$

Indeed,

$$\begin{array}{rcl} E[\mathsf{cc}\ M] & = & E[\mathcal{C}\ (\lambda k.k\ (M\ k))] \\ & \longrightarrow & (\lambda k.k\ (M\ k))\ (\lambda x.E[x]) \\ & \longrightarrow & (\lambda x.E[x])\ (M\ \lambda x.E[x]) \\ & \longrightarrow & E[M\ (\lambda x.E[x])] \\ E[\mathcal{C}\ M] & = & E[\mathsf{cc}\ (\lambda k.\mathsf{abort}(M\ k))] \\ & \longrightarrow & E[(\lambda k.\mathsf{abort}(M\ k))\ (\lambda x.E[x])] \\ & \longrightarrow & E[\mathsf{abort}(M\ \lambda x.E[x])] \\ & \longrightarrow & M\ (\lambda x.E[x]) \end{array}$$

Of course, the above rules are not "standard" rewrite rules (clearly not first-order rewrite rules) and remain informal because we have not specified what $E[\ ]$ exactly stands for or ranges over.

More fundamentally, a central question about control is: what kind of continuation can be captured by a control operator and how? Is the capture delimited? undelimited? etc.

But what is interesting is that the above intuitions are sufficient to start seeing a connection with classical logic, as initiated by Griffin in [Gri90]:

$$\begin{array}{ll} \mathsf{cc} \text{ can be typed by PL:} & ((A{\to}B){\to}A){\to}A \\ \mathcal{C} \text{ can be typed by EDN:} & (\neg\neg A){\to}A \end{array}$$

With these types, the rewrite rules satisfy the following Subject Reduction properties: The following (again, informal) typing tree for $E[\mathsf{cc}\ M]$

$$\dfrac{\dfrac{}{\Gamma \vdash \lambda x.E[x] : A{\to}B} \qquad \dfrac{\dfrac{}{\Gamma \vdash \mathsf{cc} : ((A{\to}B){\to}A){\to}A} \qquad \Gamma \vdash M : (A{\to}B){\to}A}{\Gamma \vdash \mathsf{cc}\ M : A}}{\Gamma \vdash E[\mathsf{cc}\ M] : B}$$

can be transformed into a typing tree for $E[M\ (\lambda x.E[x])]$

$$\dfrac{\dfrac{}{\Gamma \vdash \lambda x.E[x] : A{\to}B} \qquad \dfrac{\Gamma \vdash M : (A{\to}B){\to}A \qquad \dfrac{}{\Gamma \vdash \lambda x.E[x] : A{\to}B}}{\Gamma \vdash M\ (\lambda x.E[x]) : A}}{\Gamma \vdash E[M\ (\lambda x.E[x])] : B}$$

while the following (again, informal) typing tree for $E[\mathcal{C}\ M]$

$$\dfrac{\dfrac{}{\Gamma \vdash \lambda x.E[x] : A{\to}\bot} \qquad \dfrac{\dfrac{}{\Gamma \vdash \mathcal{C} : ((A{\to}\bot){\to}\bot){\to}A} \quad \Gamma \vdash M : (A{\to}\bot){\to}\bot}{\Gamma \vdash \mathcal{C}\ M : A}}{\Gamma \vdash E[\mathcal{C}\ M] : \bot}$$

can be transformed into a typing tree for $E[M\ (\lambda x.E[x])]$

$$\dfrac{\dfrac{}{\Gamma \vdash M : (A{\to}\bot){\to}\bot} \qquad \dfrac{}{\Gamma \vdash \lambda x.E[x] : A{\to}\bot}}{\Gamma \vdash M\ (\lambda x.E[x]) : \bot}$$

We can already see that, for the Subject Reduction property to hold in the case of $\mathcal{C}$, the context $E[\ ]$ cannot be any context: it has to produce something of type $\bot$. Similarly, for the generalised abort rule to satisfy Subject Reduction, the context $E[\ ]$ also needs to produce something of type $\bot$.

In fact, $\mathcal{C}$ generalises $\mathsf{abort}(\_)$, since we can define

$$\mathsf{abort}(M) := \mathcal{C}\ (\lambda x.M)$$

where x is a fresh (and therefore dummy) variable.

This reflects what we have already seen in pure logic: EDN $\Leftrightarrow$ (PL $\wedge$ EFQ)

1.3 Contributions in the 90s

One possible formalisation of the above informal concepts was proposed by Parigot [Par92] in the form of the $\lambda\mu$-calculus.

Definition 10 ($\lambda\mu$-calculus) The syntax of terms extends that of λ-calculus as follows:

$$\begin{array}{lrcl} \text{Terms} & M, N, P \ldots & ::= & x \mid \lambda x.M \mid M\ N \mid \mu\alpha.c \\ \text{Commands} & c & ::= & [\alpha]M \end{array}$$

where α ranges over a new set of variables called *continuation variables*, and $\mu\alpha.c$ binds α in c. The scope of this binder, as well as that of the unique *command* construct $[\alpha]M$, extend as much as parentheses allow, so that $\mu\alpha.M\ N$ stands for $\mu\alpha.(M\ N)$ and $[\alpha]M\ N$ stands for $[\alpha](M\ N)$.

The typing rules extend those of λ-calculus as follows:

$$\dfrac{}{\Gamma, x : A \vdash x : A; \Delta}$$

$$\dfrac{\Gamma, x : A \vdash M : B; \Delta}{\Gamma \vdash \lambda x.M : A{\to}B; \Delta} \qquad \dfrac{\Gamma \vdash M : A{\to}B; \Delta \quad \Gamma \vdash N : A; \Delta}{\Gamma \vdash M\ N : B; \Delta}$$

$$\dfrac{c : (\Gamma \vdash\ ; \alpha : A, \Delta)}{\Gamma \vdash \mu\alpha.c : A; \Delta} \qquad \dfrac{\Gamma \vdash M : A; \alpha : A, \Delta}{[\alpha]M : (\Gamma \vdash\ ; \alpha : A, \Delta)}$$

where Γ is a typing context for term-variables and Δ is a typing context for continuation-variables. Derivability of sequents in this system is respectively denoted $\Gamma \vdash_{\lambda\mu} M : A\ ;\ \Delta$ and $c : (\Gamma \vdash_{\lambda\mu}\ ; \Delta)$.

The reduction rules extend the β-reduction of λ-calculus as follows:

$$\begin{array}{lcl}(\lambda x.M)\ N & \longrightarrow & \left\{{}^{N}\!/_{x}\right\} M \\ (\mu\alpha.c)\ N & \longrightarrow & \mu\beta.\left\{{}^{[\beta]M\ N}\!/_{[\alpha]M}\right\} c \\ [\beta]\mu\alpha.c & \longrightarrow & \left\{{}^{\beta}\!/_{\alpha}\right\} c\end{array}$$

where $\left\{{}^{[\beta]M\ N}\!/_{[\alpha]M}\right\} c$ is an unconventional substitution operation, consisting in replacing, in c, every subcommand (i.e. subterm that is a command) of the form $[\alpha]M$ by $[\beta]M\ N$, with the usual capture-avoiding conditions pertaining to substitution.

The rules define a reduction relation $\longrightarrow_{\lambda\mu}$ on both terms and commands. ※

A basic intuition of the syntax is that each continuation variable α represents a "place" where various sub-terms of a given type (that of α) can be "stored" with a construct such as $[\alpha]M$. The construct $\mu\alpha.c$ retrieves what is stored under the continuation variable α and presents it as if it was a simple term. The second rewrite rule distributes for instance an argument to every sub-term stored under the variable α.

This calculus provides a computational interpretation of classical logic. Indeed, the typing system, when forgetting variables and terms, turns into the proof system of Fig. 5, where Γ and Δ now stand for sets or multisets of formulae. We see that the system generalises $\mathsf{NJ}_{\Rightarrow}$, in particular with a more general form of sequent: $A_1, \ldots, A_n \vdash A; B_1, \ldots, B_m$, and a new form of sequent $A_1, \ldots, A_n \vdash ; B_1, \ldots, B_m$.

$$\frac{}{\Gamma, A \vdash A; \Delta}$$

$$\frac{\Gamma, A \vdash B; \Delta}{\Gamma \vdash A \Rightarrow B; \Delta} \qquad \frac{\Gamma \vdash A \Rightarrow B; \Delta \quad \Gamma \vdash A; \Delta}{\Gamma \vdash B; \Delta}$$

$$\frac{\Gamma \vdash ; A, \Delta}{\Gamma \vdash A; \Delta} \qquad \frac{\Gamma \vdash A; A, \Delta}{\Gamma \vdash ; A, \Delta}$$

Figure 5: The proof system corresponding to the simply-typed $\lambda\mu$-calculus

Just like a sequent $A_1, \ldots, A_n \vdash A$ of $\mathsf{NJ}_{\Rightarrow}$ can be interpreted as the formula $(A_1 \wedge \cdots \wedge A_n) \Rightarrow A$, the two sequent forms above can respectively be interpreted as the formulae $(A_1 \wedge \cdots \wedge A_n) \Rightarrow (A \vee B_1 \vee \cdots \vee B_m)$ and $(A_1 \wedge \cdots \wedge A_n) \Rightarrow (B_1 \vee \cdots \vee B_m)$. With this interpretation, the system of Fig. 5 can be easily checked to be sound with respect to classical logic, and for completeness we can see that

- the rules of $\mathsf{NJ}_{\Rightarrow}$ are particular instances of the first three rules;
- the system features a right-contraction rule, which allows Peirce's Law to be proved, as we see below.

As with the simply-typed λ-calculus, the rewrite rules satisfy Subject Reduction, which allows the $\lambda\mu$-calculus to describe a proof-transforming procedure for the system of Fig. 5:

Theorem 6 (Subject reduction for the simply-typed $\lambda\mu$-calculus)

1. If $\Gamma \vdash_{\lambda\mu} M : A; \Delta$ and $M \longrightarrow_{\lambda\mu} M'$ then $\Gamma \vdash_{\lambda\mu} M' : A; \Delta$.
2. If $c : (\Gamma \vdash_{\lambda\mu} ; \Delta)$ and $c \longrightarrow_{\lambda\mu} c'$ then $c' : (\Gamma \vdash_{\lambda\mu} ; \Delta)$.

※

Remark 7 This calculus integrates Peirce's law: By defining

$$\mathsf{cc} := \lambda x.\mu\alpha.[\alpha](x\ \lambda y.\mu\beta.[\alpha]y)$$

we can build the following typing tree:

$$\dfrac{\dfrac{\dfrac{\dfrac{}{x:(A\to B)\to A \vdash x:(A\to B)\to A;\alpha:A} \quad \dfrac{\dfrac{\dfrac{\dfrac{}{x:(A\to B)\to A, y:A \vdash y:A;\alpha:A,\beta:B}}{[\alpha]y:(x:(A\to B)\to A, y:A \vdash ;\alpha:A,\beta:B)}}{x:(A\to B)\to A, y:A \vdash \mu\beta.[\alpha]y:B;\alpha:A}}{x:(A\to B)\to A \vdash \lambda y.\mu\beta.[\alpha]y:A\to B;\alpha:A}}{x:(A\to B)\to A \vdash x\ \lambda y.\mu\beta.[\alpha]y:A;\alpha:A}}{[\alpha](x\ \lambda y.\mu\beta.[\alpha]y):(x:(A\to B)\to A \vdash ;\alpha:A)}}{x:(A\to B)\to A \vdash \mu\alpha.[\alpha](x\ \lambda y.\mu\beta.[\alpha]y):A;}}{\vdash \mathsf{cc}:((A\to B)\to A)\to A;}$$

Now, consider that contexts are of the form $E[\ ] = [\gamma]([\ ]\ N_1 \dots N_n)$. We can perform the following reduction:

$$\begin{array}{rcl} E[\mathsf{cc}\ M] & = & [\gamma](\lambda x.\mu\alpha.[\alpha](x\ \lambda y.\mu\beta.[\alpha]y))\ M\ N_1 \dots N_n \\ & \longrightarrow & [\gamma](\mu\alpha.[\alpha](M\ \lambda y.\mu\beta.[\alpha]y))\ N_1 \dots N_n \\ & \longrightarrow & [\gamma](\mu\alpha.[\alpha](M\ \lambda y.\mu\beta.[\alpha]y\ N_1)\ N_1)\ N_2 \dots N_n \\ & \longrightarrow & \dots \\ & \longrightarrow & [\gamma]\mu\alpha.[\alpha](M\ \lambda y.\mu\beta.[\alpha]y\ N_1 \dots N_n)\ N_1 \dots N_n \\ & \longrightarrow & [\gamma](M\ \lambda y.\mu\beta.[\gamma]y\ N_1 \dots N_n)\ N_1 \dots N_n \\ & = & E[M\ (\lambda y.\mu\beta.E[y])] \end{array}$$

※

Notice that what is passed to M as an argument is not exactly $\lambda y.E[y]$, since $E[\ ]$ forms a command and $\lambda y.E[y]$ is not correct syntax, but $\mu\beta.E[y]$ turns the command $E[y]$ into a term (of any type).

Remark 8 If given a top-level continuation variable $\mathsf{top} : \bot$ (Ariola-Herbelin [AH03, AH08]), then the $\lambda\mu$-calculus integrates *Ex falso quodlibet* and the elimination of double negation: We can build the following typing tree:

$$\dfrac{\dfrac{\dfrac{\dfrac{}{x:\bot \vdash x:\bot;\alpha:A}}{[\mathsf{top}]x:(x:\bot \vdash ;\alpha:A)}}{x:\bot \vdash \mu\alpha.[\mathsf{top}]x:A;}}{\vdash \lambda x.\mu\alpha.[\mathsf{top}]x:\bot\to A;}$$

and perform the following reduction:

$$\begin{array}{rcl} E[(\lambda x.\mu\alpha.[\mathsf{top}]x)\ M] & = & [\gamma](\lambda x.\mu\alpha.[\mathsf{top}]x)\ M\ N_1 \dots N_n \\ & \longrightarrow & [\gamma](\mu\alpha.[\mathsf{top}]M)\ N_1 \dots N_n \\ & \longrightarrow & [\gamma](\mu\alpha.[\mathsf{top}]M)\ N_2 \dots N_n \\ & \longrightarrow & \dots \\ & \longrightarrow & [\gamma](\mu\alpha.[\mathsf{top}]M) \\ & \longrightarrow & [\mathsf{top}]M \end{array}$$

And by defining

$$\mathcal{C} := \ \lambda x.\mu\alpha.[\mathsf{top}](x\ \lambda y.\mu\beta.[\alpha]y)$$

we can build the following typing tree:

$$\dfrac{\dfrac{}{x:\neg\neg A \vdash x:\neg\neg A;\alpha:A} \qquad \dfrac{\dfrac{\dfrac{\dfrac{}{x:\neg\neg A, y:A \vdash y:A;\alpha:A,\beta:\bot}}{[\alpha]y:(x:\neg\neg A, y:A \vdash\ ;\alpha:A,\beta:\bot)}}{x:\neg\neg A, y:A \vdash \mu\beta.[\alpha]y:\bot;\alpha:A}}{x:\neg\neg A \vdash \lambda y.\mu\beta.[\alpha]y:\neg A;\alpha:A}}{\dfrac{\dfrac{\dfrac{x:\neg\neg A \vdash x\ \lambda y.\mu\beta.[\alpha]y:\bot;\alpha:A}{[\mathsf{top}](x\ \lambda y.\mu\beta.[\alpha]y):(x:\neg\neg A \vdash\ ;\alpha:A)}}{x:\neg\neg A \vdash \mu\alpha.[\mathsf{top}](x\ \lambda y.\mu\beta.[\alpha]y):A;}}{\vdash \lambda x.\mu\alpha.[\mathsf{top}](x\ \lambda y.\mu\beta.[\alpha]y):(\neg\neg A)\to A;}}$$

and we can perform the following reduction:

$$\begin{array}{rcl} E[\mathcal{C}\ M] & = & [\gamma](\lambda x.\mu\alpha.[\mathsf{top}](x\ \lambda y.\mu\beta.[\alpha]y))\ M\ N_1 \dots N_n \\ & \longrightarrow & [\gamma](\mu\alpha.[\mathsf{top}](M\ \lambda y.\mu\beta.[\alpha]y))\ N_1 \dots N_n \\ & \longrightarrow & [\gamma](\mu\alpha.[\mathsf{top}](M\ \lambda y.\mu\beta.[\alpha]y\ N_1))\ N_2 \dots N_n \\ & \longrightarrow & \dots \\ & \longrightarrow & [\gamma]\mu\alpha.[\mathsf{top}]M\ \lambda y.\mu\beta.[\alpha]y\ N_1 \dots N_n \\ & \longrightarrow & [\mathsf{top}]M\ \lambda y.\mu\beta.[\gamma]y\ N_1 \dots N_n \\ & = & [\mathsf{top}]M\ (\lambda y.\mu\beta.E[y]) \end{array}$$

※

Notice that what is eventually produced by the rewrites is not M and $M\ (\lambda y.\mu\beta.E[y])$, respectively, but $[\mathsf{top}]M$ and $[\mathsf{top}]M\ (\lambda y.\mu\beta.E[y])$, since reducing a command has to produce a command. But since M is of type $\bot$ (respectively produces a term of type $\bot$), it can be stored in the top-level continuation top.

Now, when thinking about classical logic, we often have in mind concepts of symmetry or duality:

Inversing the order in a boolean algebra provides another boolean algebra where e.g. the top and bottom elements have been swapped, the meet and join operations have been swapped.

Very related to this are De Morgan's rules, which show a duality, via negation, between $\wedge$ and $\vee$:

$$\begin{array}{ll} \neg(A\wedge B) & = \neg A\vee\neg B \\ \neg(A\vee B) & = \neg A\wedge\neg B \end{array}$$

In terms of proof formalisms, the classical sequent calculus LK [Gen35] shows a symmetry between the left-hand side and the right-hand of sequents, of the form $A_1, \ldots, A_n \vdash B_1, \ldots, B_m$: whatever can be done on the left-hand side can be done on the right-hand side, and vice versa. For instance, the left-introduction rule for $\wedge$ is symmetric to the right-introduction rule for $\vee$ and vice versa; left-contraction symmetric to right-contraction (and this is very different from intuitionistic logic).

But so far, such symmetries and dualities are not explicitly reflected in our proof-term approach to classical proofs.

However, before even Griffin made the connection between control operators and classical logic, Filinski [Fil89] formalised a duality between

- functions as values
- functions as continuations

in the form of a "symmetric λ-calculus", with explicit conversions from one view of functions to the other. Yet there was no explicit connection with classical logic.

In [BB96], Barbanera and Berardi formalised their own symmetric λ-calculus, with a typing system providing a Curry-Howard interpretation of classical proofs. The classical proof system depicted by their calculus is a one-sided version of the classical sequent calculus [Gen35], with a proof of normalisation for typed terms (we will see such a proof in Chapter 2).

Since then, two calculi emerged to provide Curry-Howard interpretations of the two-sided sequent calculus LK (or variants thereof), with the reduction rules describing the famous proof-transformation procedure known as *cut-elimination*:

- Urban's calculus [Urb00],
- Curien and Herbelin's $\overline{\lambda}\mu\widetilde{\mu}$ [CH00] for $\Rightarrow$,
  later extended by Wadler [Wad03] for $\wedge$ and $\vee$ (explicitly connecting the symmetries of the calculus to De Morgan's duality).

These two independent (sets of) contributions had different aims: Curien and Herbelin's was to expose, as the syntactic symmetry of the classical sequent calculus, a *duality* in computation based on Filinski's ideas about continuations and on the call-by-value and call-by-name evaluation strategies; they gave semantics to their calculus, but with no proof of normalisation. Urban's aim was to have a typing system as close as possible to LK and have a reduction system as close as possible to basic cut-elimination procedures; his Ph.D. adapted Barbanera and Berardi's proof of strong normalisation to his calculus, but gave no (denotational) semantics.

Several papers formalise the links between the various proof calculi for classical logic: in particular, [Len03] relates $\overline{\lambda}\mu\widetilde{\mu}$ and Urban's calculus, [Roc05] relates $\overline{\lambda}\mu\widetilde{\mu}$ and $\lambda\mu$, and [Her05] presents an extensive exploration of the relations between the various calculi.

In the rest of this chapter, we focus on Curien and Herbelin's calculus to explore some more semantical concepts, but many of them can be transposed to other calculi for classical logic (in particular, Parigot's $\lambda\mu$-calculus).

1.4 System L

System L is the new name of Curien and Herbelin's $\overline{\lambda}\mu\widetilde{\mu}$, extended with other connectives. We start with its syntax:

DEFINITION 11 (Syntax) The syntax of L is made of three syntactic categories:

$$\begin{array}{lcl} \text{terms} & t & ::= x \mid \mu\beta.c \mid \lambda x.t \mid (t_1, t_2) \mid \mathsf{inj}_i(t) \\ \text{continuations} & e & ::= \alpha \mid \mu x.c \mid t::e \mid (e_1, e_2) \mid \mathsf{inj}_i(e) \\ \text{commands} & c & ::= \langle t \mid e \rangle \end{array}$$

where i ranges over $\{1, 2\}$, x and α respectively range over term variables and continuation variables,[8] $\mu\beta.c$ binds β in c, $\mu x.c$ binds x in c, and $\lambda x.t$ binds x in t. The scope of binders extends as much as parentheses allow. ※

A (somewhat shallow) intuition of the syntax can be given as follows:

Term variables x, y, ...	denote inputs	
Continuation variables α, β, ...	denote outputs	
A *term* has	one main output	
	some inputs	(free term variables)
	some "alternative" outputs	(free continuation variables)
A *continuation* has	one main input	
	some "additional" inputs	(free term variables)
	some possible outputs	(free continuation variables)
A *command* is	a term facing a continuation	(the interaction is computation)

DEFINITION 12 (Typing)

We consider the following grammar for *types* (extending that of simple types):

$$A, B, \ldots ::= a \mid A \to B \mid A \wedge B \mid A \vee B$$

where a ranges over a fixed set of elements called *atomic types*. The symbol $\to$ is associative to the right, i.e. $A \to (B \to C)$ can be abbreviated as $A \to B \to C$.

The typing system for System L is given for three kinds of sequents corresponding to the three syntactic categories of the syntax:

$$\Gamma \vdash t : A \;;\; \Delta \qquad \Gamma \;;\; e : A \vdash \Delta \qquad c : (\Gamma \vdash \Delta)$$

where Γ is a typing context for term-variables and Δ is a typing context for continuation-variables.

The system is presented in Fig. 6. Derivability of sequents in this system is respectively denoted $\Gamma \vdash_{\mathsf{L}} t : A \;;\; \Delta$, $\Gamma \;;\; e : A \vdash_{\mathsf{L}} \Delta$, and $c : (\Gamma \vdash_{\mathsf{L}} \Delta)$. ※

As we can see, forgetting about variables and proof-terms does not give the sequent calculus LK exactly as we know it from [Gen35] or as the popular variants described in [TS00] (for this one can look at Urban's Ph.D. [Urb00]), if only because there are three types of sequents. However, it is a variant with a bit more structure, which defines the same notion of provability as LK, and which will prove useful for the computational interpretation of classical logic.

An intuition about this interpretation can be given as follows: similarly to the Curry-Howard correspondence in intuitionistic logic, each connective in the syntax of formulae corresponds to a type construct in programming; term constructs offer basic ways in which such

[8] As in Parigot's $\lambda\mu$-calculus [Par92].

$$\frac{}{\Gamma, x{:}A \vdash x{:}A \;;\; \Delta} \qquad \frac{}{\Gamma \;;\; \alpha{:}A \vdash \alpha{:}A, \Delta}$$

$$\frac{\Gamma, x{:}A \vdash t{:}B \;;\; \Delta}{\Gamma \vdash \lambda x.t{:}A{\to}B \;;\; \Delta} \qquad \frac{\Gamma \vdash t{:}A \;;\; \Delta \quad \Gamma \;;\; e{:}B \vdash \Delta}{\Gamma \;;\; t{::}e{:}A{\to}B \vdash \Delta}$$

$$\frac{\Gamma \vdash t_1{:}A_1 \;;\; \Delta \quad \Gamma \vdash t_2{:}A_2 \;;\; \Delta}{\Gamma \vdash (t_1, t_2){:}A_1 \wedge A_2 \;;\; \Delta} \qquad \frac{\Gamma \;;\; e{:}A_i \vdash \Delta}{\Gamma \;;\; \mathsf{inj}_i(e){:}A_1 \wedge A_2 \vdash \Delta}$$

$$\frac{\Gamma \vdash t{:}A_i \;;\; \Delta}{\Gamma \vdash \mathsf{inj}_i(t){:}A_1 \vee A_2 \;;\; \Delta} \qquad \frac{\Gamma \;;\; e_1{:}A_1 \vdash \Delta \quad \Gamma \;;\; e_2{:}A_2 \vdash \Delta}{\Gamma \;;\; (e_1, e_2){:}A_1 \vee A_2 \vdash \Delta}$$

$$\frac{c{:}(\Gamma \vdash \alpha{:}A, \Delta)}{\Gamma \vdash \mu\alpha.c{:}A \;;\; \Delta} \qquad \frac{c{:}(\Gamma, x{:}A \vdash \Delta)}{\Gamma \;;\; \mu x.c{:}A \vdash \Delta}$$

$$\frac{\Gamma \vdash t{:}A \;;\; \Delta \quad \Gamma \;;\; e{:}A \vdash \Delta}{\langle t \mid e \rangle \,{:}(\Gamma \vdash \Delta)}$$

Figure 6: Typing system for L

types can be inhabited, while continuation constructs offer basic ways in which inhabitants of such types are consumed:

- A conjunction $A_1 \wedge A_2$ corresponds to a product type, so basic inhabitants are pairs (t_1, t_2) of terms (with the first component inhabiting A_1 and the second inhabiting A_2); basic continuations that consume such a pair start by extracting either the first or the second component (in other words, they start with one of the two projections), which corresponds to the continuation constructs $\mathsf{inj}_1(e)$ and $\mathsf{inj}_2(e)$.
- A disjunction $A_1 \vee A_2$ corresponds to a sum type, so basic inhabitants are the injections $\mathsf{inj}_1(t)$ and $\mathsf{inj}_2(t)$ (with t inhabiting A_1 or inhabiting A_2, respectively); basic continuations that consume such an injection must handle both cases, so the case analysis leads to providing a pair $\langle e_1, e_2 \rangle$ of two continuations: the former can consume inhabitants of A_1 and the latter can consume inhabitants of A_2.
- An implication $A_1 \Rightarrow A_2$ corresponds to a function type, with the basic inhabitants being constructed with λ-abstractions just like in the λ-calculus; we do not have the construct that directly applies a function to an argument, but a basic way in which a continuation consumes a function is to offer an argument t as the input of the function, together with a continuation e that can consume the output of the function; hence the continuation construct $t{::}e$ (which is simply the usual stacking construct that can be found in abstract machines to implement computation in the λ-calculus).

This intuition will be strengthened by the reduction rules for System L, but we first start with an example.

The following story is borrowed from Phil Wadler [Wad03] (who might have borrowed it from Peter Selinger), and illustrates the computational contents of classical proofs:

EXAMPLE 2 (The devil, the fool, and the $1,000,000)

> The Devil meets a man and says:
> "- I have an offer for you! I promise you that
> *either I offer you $1,000,000 or, if you give me $1,000,000, then I will grant you any wish.*
> Actually, I choose to offer you the latter."
> The man then goes back home and, motivated by the Devil's promise, strives to gather $1,000,000. Ten years later, he finally succeeds; he goes back to the Devil and, handing him the money, says:
> "- Here's $1,000,000! I want immortality."
> The Devil takes the money and says:
> "- Well done and thank you!
> Actually, I've changed my mind. I've now decided to fulfil my promise by offering you $1,000,000. Here is your money back!" ※

The reason why that short story illustrates the computational contents of classical logic is that the Devil behaves as a proof of the Law of Excluded Middle: Imagine that

- the money ($1,000,000) can be seen as an atomic proposition a
- the part of the promise "If you give me $1,000,000, I'll grant you any wish" can be seen as the formula $a \Rightarrow \bot$, i.e. $\neg a$;

the Devil is then the proof of $a \vee \neg a$ shown in Fig. 7. Indeed, following the bottom-up

$$\dfrac{\dfrac{\dfrac{\dfrac{\dfrac{\dfrac{}{y{:}a \vdash y{:}a \;;\; \alpha{:}a\vee\neg a, \beta{:}\bot}}{y{:}a \vdash \mathsf{inj}_1(y){:}a\vee\neg a \;;\; \alpha{:}a\vee\neg a, \beta{:}\bot} \quad \dfrac{}{y{:}a \;;\; \alpha{:}a\vee\neg a \vdash \alpha{:}a\vee\neg a, \beta{:}\bot}}{\langle \mathsf{inj}_1(y) \mid \alpha\rangle : (y{:}a \vdash \alpha{:}a\vee\neg a, \beta{:}\bot)}}{y{:}a \vdash \mu\beta.\langle \mathsf{inj}_1(y) \mid \alpha\rangle : \bot \;;\; \alpha{:}a\vee\neg a}}{\vdash \lambda y.\mu\beta.\langle \mathsf{inj}_1(y) \mid \alpha\rangle : \neg a \;;\; \alpha{:}a\vee\neg a}}{\vdash \mathsf{inj}_2(\lambda y.\mu\beta.\langle \mathsf{inj}_1(y) \mid \alpha\rangle) : a\vee\neg a \;;\; \alpha{:}a\vee\neg a} \qquad \dfrac{}{\;;\; \alpha{:}a\vee\neg a \vdash \alpha{:}a\vee\neg a}$$

$$\dfrac{\langle \mathsf{inj}_2(\lambda y.\mu\beta.\langle \mathsf{inj}_1(y) \mid \alpha\rangle) \mid \alpha\rangle : (\vdash \alpha{:}a\vee\neg a)}{\vdash \mu\alpha.\langle \mathsf{inj}_2(\lambda y.\mu\beta.\langle \mathsf{inj}_1(y) \mid \alpha\rangle) \mid \alpha\rangle : a\vee\neg a \;;}$$

Figure 7: A proof of LEM

construction of the left-hand branch, we see that

- the proof (the Devil) starts by choosing to prove $\neg a$, as reflected by the $\mathsf{inj}_2(\_)$ construct;
- that requires an input of type a (the $\$1,000,000$ earned by the fool), namely y, as reflected by the $\lambda y.\_$ construct;
- given the impossibility to prove $\bot$ directly (the immortality wish, or for that matter, any wish), the proof re-attacks the original formula to prove, namely $a\vee\neg a$ (the Devil returns to his original promise), but this time with the input $y{:}A$ (the $\$1,000,000$ that the fool gave him);
- this time, the proof chooses to prove a, which is trivially done by returning y (the Devil chooses to give $\$1,000,000$, by returning the money that the man earned).

  We see here that the proof works because of the possibility to construct an inhabitant of

$a \vee \neg a$, twice along the same branch (we inhabit it the first time with the second injection, then with the first one), which is technically allowed by the *right-contraction* implicitly featured in the bottom two steps of the proof. While in intuitionistic logic it is possible to contract on the left but not contract on the right, classical logic allows both symmetrically.

This allows to also build a proof-term of type PL and, allowing again (as in Parigot's $\lambda\mu$) a top-level continuation variable $\mathtt{top}$ of type $\bot$, we can build proof-terms for *Ex falso quodlibet* and the elimination of double negation.

In summary, we have seen that it is easy enough to introduce proof-terms to represent classical proofs, such that the symmetry of classical logic reflects the symmetry between programs and continuations.

The use of classical reasoning corresponds to the use of control features allowing programs to capture their continuation, as we now see by looking at reductions:

Definition 13 (Reductions) The reductions are given by the following rewrite system:

$$\begin{array}{lccc}
(\to) & \langle \lambda x.t_1 \mid t_2 :: e\rangle & \longrightarrow & \langle t_2 \mid \mu x.\langle t_1 \mid e\rangle\rangle \\
(\wedge) & \langle (t_1,t_2) \mid \mathsf{inj}_i(e)\rangle & \longrightarrow & \langle t_i \mid e\rangle \\
(\vee) & \langle \mathsf{inj}_i(t) \mid (e_1,e_2)\rangle & \longrightarrow & \langle t \mid e_i\rangle \\
(\overleftarrow{\mu}) & \langle \mu\beta.c \mid e\rangle & \longrightarrow & \{^{e}/_{\beta}\}\, c \\
(\overrightarrow{\mu}) & \langle t \mid \mu x.c\rangle & \longrightarrow & \{^{t}/_{x}\}\, c
\end{array}$$

※

Now, while it was very clear that Parigot's $\lambda\mu$ forms an extension of λ-calculus, we should emphasise the fact that the λ-calculus can be encoded in System L:

Definition 14 (Encoding of λ-calculus)

We encode λ-terms as terms of System L by first encoding values, then all terms:

$$\begin{array}{lcl}
\overline{x}^{\mathrm{V}} & := & x \\
\overline{\lambda x.M}^{\mathrm{V}} & := & \lambda x.\overline{M}
\end{array}
\qquad
\begin{array}{l}
\overline{V\ M_1\ \ldots\ M_n} := \mu\alpha.\left\langle \overline{V}^{\mathrm{V}} \mid \overline{M_1} :: \ldots \overline{M_n} :: \alpha \right\rangle \\
\qquad \text{where } V \text{ is not an application and } n \geq 0
\end{array}$$

※

Lemma 9 (Simulation of λ-calculus)

1. $\mu\alpha.\left\langle \overline{M} \mid \overline{M_1} :: \ldots \overline{M_n} :: \alpha \right\rangle \longrightarrow \overline{M\ M_1\ \ldots\ M_n}$.
2. $\left\{^{\overline{N}}/_{x}\right\} \overline{M} \longrightarrow^{*} \overline{\{^{N}/_{x}\}\, M}$.
3. If $M \longrightarrow_{\beta} N$ then $\overline{M} \longrightarrow^{*} \overline{N}$.

※

Proof: The first point is a simple $\overleftarrow{\mu}$-reduction, the second point is by induction on M, the third point is by induction on the rewrite derivation. □

Lemma 10 (Preservation of simple types)

1. If $\Gamma \vdash_{\mathsf{st}\lambda} V : A$ then $\Gamma \vdash_{\mathsf{L}} \overline{M}^{\mathrm{V}} : A$;
2. If $\Gamma \vdash_{\mathsf{st}\lambda} M : A$ then $\Gamma \vdash_{\mathsf{L}} \overline{M} : A$;

※

Since System L contains cuts, a proof of LEM, and it can encode simply-typed λ-terms, it is clearly complete for classical logic (in the same sense as for the $\lambda\mu$-calculus: EDN and EFQ require the presence of a top-level continuation variable $\mathsf{top} : \bot$). Soundness can be trivially

checked by checking that all the inference rules are sound when forgetting about variables and proof-terms.

Substitution behaves well with respect to typing:

THEOREM 11 (Substitution Lemma)

1. If $c : (\Gamma, x : A \vdash_{\mathsf{L}} \Delta)$ and $\Gamma \vdash_{\mathsf{L}} t : A \ ; \Delta$ then $\{^{t}/_{x}\}\, c : (\Gamma \vdash_{\mathsf{L}} \Delta)$.
2. If $c : (\Gamma \vdash_{\mathsf{L}} \alpha : A, \Delta)$ and $\Gamma \ ; e : A \vdash_{\mathsf{L}} \Delta$ then $\{^{e}/_{\alpha}\}\, c : (\Gamma \vdash_{\mathsf{L}} \Delta)$.

※

Proof: By induction on c, simultaneously proving the two analogous properties for both terms and continuations. □

And the reduction relation satisfies Subject Reduction:

THEOREM 12 (Subject reduction for System L)

1. If $c : (\Gamma \vdash_{\mathsf{L}} \Delta)$ and $c \longrightarrow c'$ then $c' : (\Gamma \vdash_{\mathsf{L}} \Delta)$.
2. If $\Gamma \vdash_{\mathsf{L}} t : A \ ; \Delta$ and $t \longrightarrow t'$ then $\Gamma \vdash_{\mathsf{L}} t' : A \ ; \Delta$.
3. If $\Gamma \ ; e : A \vdash_{\mathsf{L}} \Delta$ and $e \longrightarrow e'$ then $\Gamma \ ; e' : A \vdash_{\mathsf{L}} \Delta$.

※

Proof: Straightforward induction on the rewrite derivations. □

Again, Subject Reduction allows the rewrite system to describe a proof transformation procedure in the classical sequent calculus, and in this case it is *cut-elimination* [Gen35].

Let us see the other properties we mentioned when introducing the Curry-Howard methodology:

Progress depends of course on what we consider an "undesirable structure". In the case of sequent calculus, the natural concept of undesirable structure is the cut, which in the typing system of L is (at least at first sight) represented as the bottom-most rule of Fig. 6. And at this point we notice that some cuts cannot be reduced, as no rewrite rule applies to their proof-terms, namely those of the form $\langle x \mid e \rangle$ and $\langle t \mid \alpha \rangle$ when e is not of the form $\mu x.c$ and t is not of the form $\mu\alpha.c$. We may think progress fails (in terms of cut-elimination), but we should also notice that cuts of that form are very peculiar: they do nothing but respectively implement a left-contraction or a right-contraction, two rules that the extra structure of the system requires for completeness (compared to e.g. G3ii [TS00]):

$$\frac{\overline{\Gamma, x : A \vdash x : A \ ; \Delta} \qquad \Gamma, x : A \ ; e : A \vdash \Delta}{\langle x \mid e \rangle : (\Gamma, x : A \vdash \Delta)} \qquad \frac{\Gamma \vdash t : A \ ; \alpha : A, \Delta \qquad \overline{\Gamma \ ; \alpha : A \vdash \alpha : A, \Delta}}{\langle t \mid \alpha \rangle : (\Gamma \vdash \alpha : A, \Delta)}$$

We actually used two of these special "cuts" in the proof of LEM showed in Fig. 7, and we would not expect to eliminate them (unless we had specific constructs for contractions and for the axiom represented as $\langle x \mid \alpha \rangle$).

Concerning normalisation, it can be proved that typed commands (resp. terms, continuations) are strongly normalising. This was inferred from Urban's calculus in [Len03], but can be more simply obtained as the direct application of Barbanera and Berardi's technique, as shown in [Pol04] for a variant of System L with explicit substitutions. This will be the topic of Chapter 2.

Finally, we look at the confluence property.

1.5 Non-confluence of cut-elimination in classical logic

As the reduction relation of System L specifies a cut-elimination procedure, we should note that cut-elimination in classical logic, at a purely logical level, can easily be defined as a non-confluent transformation procedure. A typical example of this is Lafont's example, which we first express in the original sequent calculus LK, with explicit rules for weakenings and contractions and "context-splitting" rules (see e.g. [TS00]):

Example 3 (Lafont's example for non-confluence)

Consider the following cut that we would like to eliminate

$$\dfrac{\dfrac{\begin{array}{c}\vdots\,\pi\\ \Gamma \vdash \Delta\end{array}}{\Gamma \vdash \Delta, A} \qquad \dfrac{\begin{array}{c}\vdots\,\pi'\\ \Gamma' \vdash \Delta'\end{array}}{\Gamma', A \vdash \Delta'}}{\Gamma, \Gamma' \vdash \Delta, \Delta'}$$

There are two ways to eliminate the cut:

$$\dfrac{\begin{array}{c}\vdots\,\pi\\ \Gamma \vdash \Delta\end{array}}{\Gamma, \Gamma' \vdash \Delta, \Delta'} \quad \text{or} \quad \dfrac{\begin{array}{c}\vdots\,\pi'\\ \Gamma' \vdash \Delta'\end{array}}{\Gamma, \Gamma' \vdash \Delta, \Delta'}$$

※

This obviously leads to non-confluence as soon as π and π' are two distinct proofs (say, cut-free). Note that we could, somewhat artificially, avoid the choice between π and π' by considering the following *mix* rule [FR94]:

$$\dfrac{\Gamma \vdash \Delta \quad \Gamma' \vdash \Delta'}{\Gamma, \Gamma' \vdash \Delta, \Delta'}$$

which would allow the symmetric combination of π and π' into a single proof (and what would be the semantics of this combination?). The question is whether we accept this derivation as a normal proof. Let us look at the same example in a sequent calculus (such as G3ii [TS00]) where rules are "context-sharing":

Example 4 (Lafont's example in a context-sharing sequent calculus)

The following cut:

$$\dfrac{\dfrac{\begin{array}{c}\vdots\,\pi\\ \Gamma \vdash \Delta\end{array}}{\Gamma \vdash \Delta, A} \qquad \dfrac{\begin{array}{c}\vdots\,\pi'\\ \Gamma \vdash \Delta\end{array}}{\Gamma, A \vdash \Delta}}{\Gamma \vdash \Delta}$$

can be reduced to:

$$\begin{array}{c}\vdots\,\pi\\ \Gamma \vdash \Delta\end{array} \quad \text{or} \quad \begin{array}{c}\vdots\,\pi'\\ \Gamma \vdash \Delta\end{array}$$

※

What is even more striking in this example is that π and π' are two proofs of the same sequent, which we probably do not want to consider denotationally equal and whose combination via the following rule

$$\frac{\Gamma \vdash \Delta \quad \Gamma \vdash \Delta}{\Gamma \vdash \Delta}$$

looks even more artificial than with the context-splitting mix. Again, do we want this derivation as a normal proof?

Unsatisfying though the mix may seem, it does technically solve Lafont's non-confluence problem based on two weakenings. Unfortunately, it cannot solve the even more problematic example obtained with contractions instead of weakenings:

EXAMPLE 5 (Example with contractions) The following cut

$$\dfrac{\dfrac{\vdots\,\pi}{\Gamma \vdash \Delta, A, A}}{\Gamma \vdash \Delta, A} \qquad \dfrac{\vdots\,\pi'}{\Gamma, A, A \vdash \Delta}}{\Gamma \vdash \Delta}$$

can be reduced to

$$\begin{array}{c}\vdots\,\pi' \quad \vdots\,\pi' \\ \bullet \quad\quad \bullet \\ \vdots\,\pi(\simeq) \\ \Gamma \vdash \Delta\end{array} \quad \text{or} \quad \begin{array}{c}\vdots\,\pi \quad \vdots\,\pi \\ \bullet \quad\quad \bullet \\ \vdots\,\pi'(\simeq) \\ \Gamma \vdash \Delta\end{array}$$

where $\pi(\simeq)$ (resp. $\pi'(\simeq)$) denotes the proof π (resp. π') modified by the propagation of π' (resp. π) into its structure.

We can give a concrete instance of the above:

$$\frac{(A \to B) \to A \vdash A \qquad A, A \to C, A \to D \vdash C \wedge D}{(A \to B) \to A, A \to C, A \to D \vdash C \wedge D}$$

Peirce's Law requires a right-contraction on the cut-formula A while the right-hand side proof requires a left-contraction on the cut-formula A. ※

Coming back to the proof-term side, both examples would appear in System L as instances of the following scheme:

$$\frac{c : (\Gamma \vdash \alpha : A, \Delta)}{\Gamma \vdash \mu\alpha.c : A \;;\; \Delta} \qquad \frac{c' : (\Gamma, x : A \vdash \Delta)}{\Gamma \;;\; \mu x.c' : A \vdash \Delta}}{\langle \mu\alpha.c \mid \mu x.c' \rangle : (\Gamma \vdash \Delta)}$$

α (resp. x) could be used 0 (weakening), 1, or several (contraction) times in c (resp. c').

That cut could be reduced to

$$\dfrac{c : (\Gamma \vdash \alpha : A, \Delta) \qquad \dfrac{c' : (\Gamma, x : A \vdash \Delta)}{\Gamma \;;\; \mu x.c' : A \vdash \Delta}}{\left\{ {}^{\mu x.c'}\!/_{\alpha} \right\} c : (\Gamma \vdash \Delta)} \quad \text{or} \quad \dfrac{\dfrac{c : (\Gamma \vdash \alpha : A, \Delta)}{\Gamma \vdash \mu\alpha.c : A \;;\; \Delta} \qquad c' : (\Gamma, x : A \vdash \Delta)}{\{ {}^{\mu\alpha.c}\!/_{x} \}\, c' : (\Gamma \vdash \Delta)}$$

where dotted lines do not represent primitive inference rules, but inference rules that have been shown admissible in the typing system (Lemma 11).

In the case of weakening, and reflecting Example 4, $\alpha \notin \mathsf{FV}(c)$ and $x \notin \mathsf{FV}(c')$ and we can reduce $\langle \mu\alpha.c \mid \mu x.c' \rangle$ to two arbitrary commands c and c' with the same type.

This makes it hard to give a denotational semantics of classical proofs or of typed proof-terms: if we require $\langle \mu\alpha.c \mid \mu x.c' \rangle$, $\left\{ {}^{\mu x.c'}\!/_{\alpha} \right\} c$, and $\{ {}^{\mu\alpha.c}\!/_{x} \}\, c'$, to have the same denotation,

Example 4 leads to giving the same denotation to every proof of the same sequent.

This of course relates to the fact that we have already mentioned: a CCC with initial object where every object A naturally isomorphic to $\neg\neg A$ collapses to a boolean algebra. As identified in [Str11], there are three natural ways to resolve this:

1. Break the symmetry between $\wedge$ and $\vee$
2. Break the cartesian product (as studied for instance in [FP06, LS05, Lam07, Str11])
3. Break curryfication (as studied for instance in [DP04, CS09])

In this dissertation, we break the symmetry between $\wedge$ and $\vee$, since out of the three solutions it is the one for which the Curry-Howard correspondence with programs is best understood.

One way of breaking the non-confluence problem

$$\dfrac{\begin{array}{cc}\vdots\,\pi & \vdots\,\pi' \\ \Gamma \vdash \Delta, A & \Gamma, A \vdash \Delta\end{array}}{\Gamma \vdash \Delta}$$

is simply to give systematic priority to

- the right (push π into π')
- or to the left (push π' into π)

Almost by definition, both solutions make the calculus confluent.

They also break the $\wedge\vee$ symmetry: Giving systematic priority to the right, say, makes a term t of type $A\wedge B$ have the same behaviour as $(\mathsf{inj}_1(t), \mathsf{inj}_2(t))$, whereas a continuation e of type $A\vee B$ will not necessarily have the same behaviour as $(\mathsf{inj}_1(e), \mathsf{inj}_2(e))$.

More details on this will be given in Chapter 3, but we shall also see by semantical means that the $\wedge\vee$ symmetry is broken. The two reduction strategies suggest to construct two denotational semantics $[\![c]\!]_{\mathsf{N}}$ and $[\![c]\!]_{\mathsf{V}}$ with the hope that:
$[\![c_0]\!]_{\mathsf{N}} = [\![c_1]\!]_{\mathsf{N}}$ iff "$c_0 \longleftrightarrow^* c_1$ with systematic priority to the right"
$[\![c_0]\!]_{\mathsf{V}} = [\![c_1]\!]_{\mathsf{V}}$ iff "$c_0 \longleftrightarrow^* c_1$ with systematic priority to the left"

The use of the letters N and V reflects the fact that the strategies relate to the notions of Call-by-name and Call-by-value, as investigated for instance by Plotkin [Plo75], Moggi [Mog89], and others.

In conclusion of this section, we have seen that it is easy enough to give a rewrite system on proof-terms to represent cut-elimination (and the system follows the intuitions of continuations and control), but it gives a non-confluent calculus because cut-elimination is non-confluent in classical logic (via the Curry-Howard correspondence, because programs and continuations fight for the control of computation).

The rest of this chapter is devoted to the construction of the above CBN and CBV semantics.

1.6 Continuations, Call-by-Name and Call-by-Value

Call-by-name and call-by-value are two strategies for evaluating programs. Imagine the definition of a function (in pseudo-code):

```
MyFavoriteFunction(x){
    ... x ...
}
```

and later a call to that function with argument `A`

```
MyFavoriteFunction(A)
```

The main question is whether `A` should be evaluated before entering the (code of the) function (CBV) or when it is actually used (CBN)? This is a question of interpretation or compilation of programs and, especially in presence of side-effects, knowing which of the two the compiler implements, is vital for the determinism of evaluation.

In general, we call *values* what evaluation should produce (e.g. booleans true, false). In functional programming, functions are particular values and can be passed as arguments. In general, functions are therefore not reduced.

The λ-calculus is both a core functional language and a theory of functions.

As a core functional language, it is equipped with an operational semantics, close to implementation, which can be expressed by an evaluation strategy that selects a unique β-redex to reduce:

- Never reduce a λ-abstraction, as it is a "value" (this is called *weak reduction*)
- Always reduce M first in an application $M\ N$. Then:
  - If M is an abstraction: reduce the β-redex first (CBN)
    reduce N first (CBV)
  - Otherwise, reduce N (never happens with closed terms)

We denote those strategies $\longrightarrow_{\mathsf{CBN}}$ and $\longrightarrow_{\mathsf{CBV}}$.[9]

As a theory of functions, the λ-calculus is equipped with a denotational semantics close to the mathematical notion of functions: in particular, equalities are congruences (e.g. if $M = N$ then $\lambda x.M = \lambda x.N$) and reductions are congruences (this is called *strong reduction*). In [Plo75], Plotkin investigated the concepts of call-by-name and call-by-value by identifying particular λ-terms as values:

DEFINITION 15 (Value, β_v, Call-by-Name and Call-by-Value)

λ-terms of the form $\lambda x.M$ and x are called *values*, and denoted V, V', etc, while λ-terms of the form MN are not values.[10]

- "Call-by-name" evaluation is given by general β-reduction
$$(\beta) \qquad (\lambda x.M)\ N \longrightarrow \left\{{}^{N}\!/_{x}\right\} M$$
- "Call-by-value" evaluation is given by the restriction of β-reduction where the argument is a value
$$(\beta_v) \qquad (\lambda x.M)\ V \longrightarrow \left\{{}^{V}\!/_{x}\right\} M$$

※

Now, natural questions to raise are

CBN: whether there is a relation between $\longrightarrow_{\mathsf{CBN}}$ and $\longrightarrow_{\beta}$;

[9] For instance, Haskell implements CBN while OCaml implements CBV.

[10] The intuition is that, by evaluating MN, you may get a λ-term of a completely different shape.

CBV: whether there is a relation between $\longrightarrow_{\mathsf{CBV}}$ and $\longrightarrow_{\beta_v}$?

Clearly, $\longrightarrow_{\mathsf{CBN}} \subseteq \longrightarrow_\beta$ and $\longrightarrow_{\mathsf{CBV}} \subseteq \longrightarrow_{\beta_v}$, but what about the converse? A bridge between weak and strong reductions was given by Plotkin [Plo75]:

THEOREM 13 (CBN and CBV: weak and strong reductions)

CBN: $\longrightarrow_\beta^*$ is the closure of $\longrightarrow_{\mathsf{CBN}}^*$ under $\dfrac{M_1 \longrightarrow_{\mathsf{CBN}}^* C[M_2] \qquad M_2 \longrightarrow M_3}{M_1 \longrightarrow C[M_3]}$

CBV: $\longrightarrow_{\beta_v}^*$ is the closure of $\longrightarrow_{\mathsf{CBV}}^*$ under $\dfrac{M_1 \longrightarrow_{\mathsf{CBV}}^* C[M_2] \qquad M_2 \longrightarrow M_3}{M_1 \longrightarrow C[M_3]}$

where $C[\,]$ ranges over any kind of context (i.e. λ-term with a hole). ※

The point of this result is that we shall now call CBN and CBV, not some operational semantics of some functional programming language, but some rewriting theories in λ-calculus.

As we have already mentioned compilation in reference to CBN/CBV, it is interesting to see that λ-calculus can be compiled into (a fragment of) itself: this is based on the idea of the program transformation presented in Section 1.2 using continuations. As continuations are passed as an extra argument to every call, such transformations are known as *Continuation Passing Style* (CPS)-translations.

DEFINITION 16 (CPS-translations)

Two important CPS-translations were defined for CBN and CBV:

CBN-translation (Plotkin [Plo75])		CBV-translation (Reynolds [Rey72])	
$\underline{x}$	$:= \lambda k.x\ k$	$\overline{x}$	$:= \lambda k.k\ x$
$\underline{\lambda x.M}$	$:= \lambda k.k\ (\lambda x.\underline{M})$	$\overline{\lambda x.M}$	$:= \lambda k.k\ (\lambda x.\lambda k'.\overline{M}\ k')$
$\underline{M\ N}$	$:= \lambda k.\underline{M}\ (\lambda y.y\ \underline{N}\ k)$	$\overline{M\ N}$	$:= \lambda k.\overline{M}\ (\lambda y.\overline{N}\ (\lambda z.y\ z\ k))$

where the variables k and k' are always chosen to be fresh. ※

One main feature of these translations is their target fragment of the λ-calculus: in this fragment, arguments are always values! This fragment is stable under $\longrightarrow_\beta$ and $\longrightarrow_{\beta_v}$, which actually coincide. The evaluation of a CPS-translated term is *strategy-indifferent*. How this evaluation relates to the evaluation of the original term is given by the following simulation properties:

THEOREM 14 (CPS-translations preserve reductions)

Soundness:

CBN If $M \longrightarrow_\beta N$ then $\underline{M} \longrightarrow_\beta^* \underline{N}$

CBV If $M \longrightarrow_{\beta_v} N$ then $\overline{M} \longrightarrow_\beta^* \overline{N}$

Completeness:

CBN If $\underline{M} \longleftrightarrow_\beta^* \underline{N}$ then $M \longleftrightarrow_\beta^* N$

CBV It is **not** the case for CBV, that if $\overline{M} \longleftrightarrow_\beta^* \overline{N}$ then $M \longleftrightarrow_{\beta_v}^* N$.

※

Proof: It is interesting to look at soundness to see how or why the CPS-translations make sense; for complete proofs, see [Plo75].

$$\begin{array}{rcl}
\underline{(\lambda x.M)\ N} & = & \lambda k.(\lambda k'.(k'\ (\lambda x.\underline{M})))\ (\lambda y.y\ \underline{N}\ k) \\
& \longrightarrow & \lambda k.(\lambda y.y\ \underline{N}\ k)\ (\lambda x.\underline{M}) \\
& \longrightarrow & \lambda k.(\lambda x.\underline{M})\ \underline{N}\ k \\
& \longrightarrow & \lambda k.(\underline{\left\{{}^{N}\!/_{x}\right\}\underline{M}})\ k \\
& = & \lambda k.(\underline{\left\{{}^{N}\!/_{x}\right\}M})\ k \\
& \longrightarrow & \underline{\left\{{}^{N}\!/_{x}\right\}M}
\end{array}$$

The second equality of course relies on the property that the CPS-translation behaves well with substitution: $(\underline{\left\{{}^{N}\!/_{x}\right\}\underline{M}} = \underline{\left\{{}^{N}\!/_{x}\right\}M})$. The last rewrite is an instance of β-reduction because $\underline{\left\{{}^{N}\!/_{x}\right\}M}$ necessarily starts with a λ-abstraction (i.e. we are not using η-reduction).

$$\begin{array}{rcll}
\overline{(\lambda x.M)\ V} & = & \lambda k.(\lambda k'.(k'\ (\lambda x k''.\overline{M}\ k'')))\ (\lambda y.\overline{V}\ (\lambda z.y\ z\ k)) & \\
& \longrightarrow & \lambda k.(\lambda y.\overline{V}\ (\lambda z.y\ z\ k))\ (\lambda x k''.\overline{M}\ k'') & \\
& \longrightarrow & \lambda k.\overline{V}\ (\lambda z.(\lambda x k''.\overline{M}\ k'')\ z\ k) & \\
& \longrightarrow & \lambda k.\overline{V}\ (\lambda z.(\lambda k''.\{{}^{z}\!/_{x}\}\,\overline{M}\ k'')\ k) & \\
& = & \lambda k.\overline{V}\ (\lambda x.(\lambda k''.\overline{M}\ k'')\ k) & \\
& \longrightarrow & \lambda k.\overline{V}\ (\lambda x.\overline{M}\ k) & \\
& \longrightarrow & \lambda k.(\lambda x.\overline{M}\ k)\ H & \text{where } \overline{V} = \lambda k.k\ H \\
& \longrightarrow & \lambda k.(\underline{\left\{{}^{H}\!/_{x}\right\}\overline{M}})\ k & \\
& = & \underline{\lambda k.(\{{}^{V}\!/_{x}\}\,M)\ k} & \\
& \longrightarrow & \underline{\{{}^{V}\!/_{x}\}\,M} &
\end{array}$$

The second equality is simply the renaming of z into x; the third one relies again on the property that the CPS-translation behaves well with substitution by values $(\underline{\left\{{}^{H}\!/_{x}\right\}\underline{M}} = \underline{\left\{{}^{V}\!/_{x}\right\}M}$ if $\underline{V} = \lambda k.kH)$. The last rewrite is again an instance of β-reduction, not η-reduction.

The point here is to realise that if V had not been a value, then $\overline{V}$ would not be of the form $\lambda k.kH$, and the simulation of this specific β-reduction would be stuck. □

The above simulations give some intuition about the encodings: the translation of any term M starts with a λ-abstract on a fresh variable k that is used exactly once. The variable k stands for the current continuation (hence the expression *continuation-passing style*). In the encoding of an abstraction $\lambda x.M$ (which is a value), the current continuation is applied to the encoding of the body M under a λ-abstraction on x. In case of an application $M\ N$, the current continuation is not directly applied, but wrapped in a bigger continuation that is passed as an argument to the encoding of M; what this wrapping exactly is depends on whether we do CBN or CBV and will determine whether we reflect the evaluation of N as a value V before we reflect the reduction of $M\ N$.

Now, the fact that $\overline{M} \longleftrightarrow^*_\beta \overline{N}$ does not imply $M \longleftrightarrow^*_{\beta_v} N$ is slightly disappointing: one way to look at it is to consider that $M \longleftrightarrow^*_{\beta_v} N$ is too weak, or incomplete, for Call-by-Value. Indeed, the inspiration from monads, and Moggi's monadic λ-calculus [Mog89], has allowed the extension of the Call-by-Value λ-calculus into a sound and complete calculus with respect to the CBV CPS-translation (see for instance [Len06]).

We now turn to the behaviour of the CPS-translations with respect to typing: Assume we have $\Gamma \vdash M : A$. Do we have: $\Gamma' \vdash \underline{M} : A'$ (for some Γ', A') and $\Gamma'' \vdash \overline{M} : A''$ (for some Γ'', A'')?

The CPS-translations reveal two classes of terms in the target: *values* & *continuations* (like k). The types of values and continuations in the translated terms depend on CBN or CBV:

Definition 17 (CPS-translations of simple types)

We choose or we add a particular atomic type R, called the *response type*, then we define

CBN		CBV	
$\underline{a}$	$:= a$	$\overline{a}$	$:= a$
$\underline{A \to B}$	$:= ((\underline{A}\to R)\to R)\to(\underline{B}\to R)\to R$	$\overline{A \to B}$	$:= \overline{A}\to(\overline{B}\to R)\to R$

※

Intuitively, a type A in the original calculus will give rise to a type $\underline{A}$ (resp. $\overline{A}$) of "A-values"; continuations are functions consuming those and returning something in the response type R (which is abstract in the sense that we will never need to know what it is), so continuations will therefore be of type $\underline{A}\to R$ (resp. $\overline{A}\to R$).

The encoding $\underline{M}$ (resp. $\overline{M}$) of a term M of type A will take the current continuation, of type $\underline{A}\to R$ (resp. $\overline{A}\to R$), and using that continuation, it will eventually output a response in the response type (as we have seen, the encoding starts with $\lambda k.\ldots$). It will therefore be of type $(\underline{A}\to R)\to R$ (resp. $(\overline{A}\to R)\to R$).

This is formalised as the following theorem:

Theorem 15 (CPS-translations preserve types)

If $\Gamma \vdash M : A$ then $(\underline{\Gamma}\to R)\to R \vdash \underline{M} : (\underline{A}\to R)\to R$ and $\overline{\Gamma} \vdash \overline{M} : (\overline{A}\to R)\to R$,
where $\overline{x_1 : A_1, \ldots, x_n : A_n}$ stands for $x_1 : \overline{A_1}, \ldots, x_n : \overline{A_n}$
and $((\underline{x_1 : A_1, \ldots, x_n : A_n})\to R)\to R$ stands for $x_1 : (\underline{A_1}\to R)\to R, \ldots, x_n : (\underline{A_n}\to R)\to R$. ※

Proof: Straightforward induction on M. □

Variants of CPS-translations exist, of which we mention two that are related to CBN and CBV:

Definition 18 (Variants)

- Fischer's translation for CBV [Fis72]

$$\begin{array}{rcl} \overline{x} & := & \lambda k.k\ x \\ \overline{\lambda x.M} & := & \lambda k.(k\ (\lambda k'.\lambda x.\overline{M}\ k')) \\ \overline{M\ N} & := & \lambda k.\overline{M}\ (\lambda y.\overline{N}\ (\lambda z.y\ k\ z)) \end{array} \qquad \begin{array}{rcl} \overline{a} & := & a \\ \overline{A \to B} & := & (\overline{B}\to R) \to \overline{A}\to R \end{array}$$

- Hofmann & Streicher's translation for CBN [HS97], using product types

$$\begin{array}{rcl} \underline{x} & := & \lambda k.x\ k \\ \underline{\lambda x.M} & := & \lambda(x,k).\underline{M}\ k \\ \underline{M\ N} & := & \lambda k.\underline{M}\ (\underline{N}, k) \end{array} \qquad \begin{array}{rcl} \underline{\underline{a}} & := & a\to R \\ \underline{\underline{A \to B}} & := & (\underline{\underline{A}}\to R) \times \underline{\underline{B}} \end{array}$$

※

Fischer's CBV-translation is very similar to Reynolds's: they only differ in the order in which arguments are passed in the encoding of λ-abstractions and applications (e.g. for the abstraction, Reynolds's translation binds x first, then binds the continuation variable k', whereas Fischer's binds k' first, then x). This is reflected in the encoding of the function type $A\to B$: the two arguments are swapped.

Hofmann & Streicher's CBN-translation differs more importantly from Plotkin's, as fewer "continuation wrappings" are introduced, reflected in the number of $\cdots\to R$ in the encoding

of types: that encoding works "negatively", as $\underline{\underline{A}}$ is directly the type of A-continuations which are not necessarily functions consuming an A-value and returning in the response type.

Again, these translations allow the same simulations as Plotkin's and Reynolds's, and of course preserve typing, with a slightly different formulation in the case of CBN:

THEOREM 16 (Hofmann & Streicher's CBN-translation preserves types)
If $\Gamma \vdash M : A$ then $\underline{\Gamma}\to R \vdash \underline{M} : \underline{A}\to R$
where $(\underline{\underline{x_1 : A_1, \ldots, x_n : A_n}})\to R$ stands for $x_1 : \underline{\underline{A_1}}\to R, \ldots, x_n : \underline{\underline{A_n}}\to R$. ※

Let us now look at CPS-translations with respect to denotational semantics: Remember that simply-typed λ-terms have a semantics in a Cartesian Closed Category. CPS-translations compile the simply-typed λ-calculus into itself (preserving types in the sense of Theorem 15), so we can now assign to a simply-typed λ-term M, the semantics (in a CCC) of $\underline{M}$ or $\overline{M}$ (so that semantics now depends on CBN/CBV). By the simulation theorem (Theorem 14), reductions are sound w.r.t. their corresponding semantics.

More interestingly, notice that we do not need the whole structure of a CCC to build those two semantics, as $\underline{M}$ or $\overline{M}$ live in a fragment of the simply-typed λ-calculus (the CPS-fragment), where in particular the types of $\underline{M}$ or $\overline{M}$ are functional types. More than this, every functional type that we ever need for that fragment is of the form $A\to R$.[11] Therefore, in order to build the categorical semantics of the CPS-fragment, we do not need as strong axioms as those of a CCC: on top of asking for cartesian products we only require the existence of exponentials objects of the form R^A. This is called a *response category*.

Now given a response category, the sub-category made of the objects of the form R^A is called a *continuation category*, a.k.a. *control category* (Selinger [Sel01]). Such a category turns out to have a rich structure that proves very useful for classical logic: not only it is a CCC (with exponential objects $(R^A)^{(R^B)}$ defined as $R^{A\times(R^B)}$) but objects of the form $R^{A\times B}$, denoted $R^A \parr R^B$, will play an important role.

1.7 Classical logic and CBN/CBV

We now relate classical logic to the above notions. We first review known translations from classical logic into intuitionistic logic: The intuition is that we can always turn P into P' by adding (enough) double negations, to get the property that

$$\text{If } \vdash_c P \text{ then } \vdash_i P'.$$

where $\vdash_c$ denotes classical provability and $\vdash_i$ denotes intuitionistic provability. Obviously, $\vdash_c P \leftrightarrow P'$, since the two formulae only differ by some double negations.

A potential question is then: If it suffices to add double negations in a classically provable formula to make it intuitionistically provable, are the two logics *really* different? Well, they differ at least in the sense that double negations break some of the nice properties of intuitionistic logic:

$$\text{If } \vdash_i A_1 \vee A_2 \text{ then either } \vdash_i A_1 \text{ or } \vdash_i A_2.$$
$$\text{If } \vdash_i \exists x A \text{ then there is } t \text{ such that } \vdash_i \{t/x\}\, A$$

[11] To be precise, we did use types such as $A_1\to\cdots\to A_n\to R$, but if we have products we can consider this to be the type $(A_1 \times \cdots \times A_n)\to R$.

Getting t from the proof of $\vdash_i \exists x A$ is called *witness extraction*. This can also be done in some theories, like (Heyting) arithmetics:

$$\text{If } HA \vdash_i \exists x A \text{ then there is } t \text{ such that } HA \vdash_i \{^t/_x\}\, A.$$

But in the most general case we cannot have the same properties when $\vdash_i \neg\neg(A_1 \vee A_2)$ or $\vdash_i \neg\neg\exists x A$. So what to do with a classical proof of $\vdash \exists x A$ is unclear. However, it is known that if A satisfies some specific property, a witness may be obtained from a classical proof of $\vdash \exists x A$; this is called classical witness extraction, and we will see this in Chapter 2.

The principle of inserting double negations gives rise to double negation translations (or $\neg\neg$-translations), of which we present two, remembering that $\neg A$ is $A \Rightarrow \bot$:

Definition 19 (Double negation translations)

$$\begin{array}{lcllcl} a^\bullet & := & a & a^\star & := & a \\ (A \Rightarrow B)^\bullet & := & ((A^\bullet \Rightarrow \bot) \Rightarrow \bot) \Rightarrow (B^\bullet \Rightarrow \bot) \Rightarrow \bot & (A \Rightarrow B)^\star & := & A^\star \Rightarrow (B^\star \Rightarrow \bot) \Rightarrow \bot \end{array}$$

※

We realise here that these translations, via the Curry-Howard correspondence, are exactly the translations of types from Definition 17 that make Plotkin's and Reynolds's translations "preserve types": The response type previously denoted R corresponds to the formula $\bot$, and a continuation is a proof of negation.

1.7.1 Identifying CBN and CBV in System L

The fact that double negation translations allow the construction of an intuitionistic proof of $A^\bullet$ (resp. $A^\star$) from a classical proof of A, suggests that we can adapt the CPS-translations of Definitions 16 and 18 to encode classical proof-terms, say of System L, into the simply-typed λ-calculus. If this encoding not only preserves types but also reductions (as in e.g. Theorem 14), then we could assign to a classical proof-term the categorical semantics of its CPS-encoding (which, as a simply-typed λ-term, is well-understood).

It remains to identify which reductions of System L will be reflected in the CPS-encoding.

Inspired by Theorem 14, we remark that the β-reductions that can be reflected by the CBV-encoding are of the form β_v, i.e. those reductions where every substitution that is computed substitute a variable by a value. In System L, we can impose similar restrictions: a CBV-reduction should only allow a substitution $\{^t/_x\}\, c$ to be computed if t is a "value"; and by symmetry, we could expect CBN-reduction to only allow a substitution $\{^e/_\alpha\}\, c$ to be computed if e is a "continuation value". But we still need to identify what the notions of values and continuation values are for System L. Considering the non-confluence situation $\langle \mu\alpha.c \mid \mu x.c' \rangle$ described in Section 1.4 (which causes so much difficulty for building semantics for System L), ruling out $\mu\alpha.c$ as value and ruling out $\mu x.c'$ as continuation value solves the problem: CBN-reduction would allow the reduction $\langle \mu\alpha.c \mid \mu x.c' \rangle \longrightarrow \{^{\mu\alpha.c}/_x\}\, c'$ and disallow $\langle \mu\alpha.c \mid \mu x.c' \rangle \longrightarrow \left\{^{\mu x.c'}/_\alpha\right\} c$, while CBV-reduction would allow the reduction $\langle \mu\alpha.c \mid \mu x.c' \rangle \longrightarrow \left\{^{\mu x.c'}/_\alpha\right\} c$ and disallow $\langle \mu\alpha.c \mid \mu x.c' \rangle \longrightarrow \{^{\mu\alpha.c}/_x\}\, c'$. In other words, CBV-reduction gives priority to the right while CBV-reduction gives priority to the left.

We can formalise this as the following definition:

DEFINITION 20 (CBN and CBV for System L -first attempt)

We identify the following notions of term values and continuation values:

$$\begin{array}{lrcl} \text{Term values} & V & ::= & x \,|\, \lambda x.t \,|\, (t_1, t_2) \,|\, \mathsf{inj}_i(t) \\ \text{Continuation values} & E & ::= & \alpha \,|\, t::e \,|\, (e_1, e_2) \,|\, \mathsf{inj}_i(e) \end{array}$$

The reduction relations $\longrightarrow_{\mathsf{CBN}}$ and $\longrightarrow_{\mathsf{CBV}}$ are defined as the contextual closures of the (groups of) rules in Fig. 9. ※

$$\begin{array}{lrcl} (\to) & \langle \lambda x.t_1 \mid t_2::e \rangle & \longrightarrow & \langle t_2 \mid \mu x.\langle t_1 \mid e \rangle \rangle \\ (\wedge) & \langle (t_1, t_2) \mid \mathsf{inj}_i(e) \rangle & \longrightarrow & \langle t_i \mid e \rangle \\ (\vee) & \langle \mathsf{inj}_i(t) \mid (e_1, e_2) \rangle & \longrightarrow & \langle t \mid e_i \rangle \end{array}$$

$$\begin{array}{llll} (\overleftarrow{\mu}_{\mathsf{N}}) & \langle \mu\beta.c \mid E \rangle \longrightarrow \left\{ {}^{E}\!/_{\beta} \right\} c & (\overleftarrow{\mu}) & \langle \mu\beta.c \mid e \rangle \longrightarrow \{ {}^{e}\!/_{\beta} \}\, c \\ (\overrightarrow{\mu}) & \langle t \mid \mu x.c \rangle \longrightarrow \{ {}^{t}\!/_{x} \}\, c & (\overrightarrow{\mu}_{\mathsf{V}}) & \langle V \mid \mu x.c \rangle \longrightarrow \left\{ {}^{V}\!/_{x} \right\} c \end{array}$$

CBN CBV

Figure 8: CBN and CBV reduction in System L (first attempt)

The fact that CBV-reduction keeps $(\overleftarrow{\mu})$ and restricts $(\overrightarrow{\mu})$ into $(\overrightarrow{\mu}_{\mathsf{V}})$ (and vice versa for CBN), is the formalisation of what we described before.

Doing this "works" in the sense that both CBN and CBV reductions are confluent systems (as higher-order orthogonal rewrite systems).

Unfortunately, these restrictions are not sufficient to build the denotational semantics of those systems according to methodology of CPS-translations, at least if we are to re-use the CPS-translations of types that we have seen in Section 1.6: Indeed, the simulation property that would be, for System L, the equivalent of Theorem 14, fails.

This was noticed in an erratum of [CH00], which also notices that the simulation does work on two specific fragments of System L: Concentrating on the implicational fragment,

- CBN-reduction can be simulated in the λ-calculus (according to Hofmann and Streicher's translation of types [HS97]) when every continuation of the form $t::e$ is such that t is a term value;
- CBV-reduction can be simulated in the λ-calculus (according to Reynold's or Fischer's translation of types [Rey72, Fis72]) when every continuation of the form $t::e$ is such that e is a continuation value.

Note that this makes sense because the former and the latter fragments are stable under CBN and CBV reduction, respectively.

This also suggests how to refine the notions of term values and continuation values as follows: instead of these notions concerning only the top-level construct of a term or a continuation, our new and more appropriate notions of values will be recursively defined.

This *strong* notion of value is taken primarily from [Wad03]:[12]

DEFINITION 21 (CBN and CBV for System L)

[12] Inspired by [MM09], we make a change about implication in the case of CBV, for which we *also* restrict continuation values, since this will make CBV normal forms correspond to proofs in LKQ [DJS95, DJS97], as we shall see in Chapter 3.

In CBN, we identify the following notion of *continuation values*:

$$\text{CBN continuation values} \quad E ::= \alpha \,|\, t::E \,|\, (E_1, E_2) \,|\, \mathsf{inj}_i(E)$$

In CBV, we identify the following notion of *term values* and *continuation values*:

$$\begin{array}{lrcl} \text{CBV term values} & V & ::= & x \,|\, \lambda x.t \,|\, (V_1, V_2) \,|\, \mathsf{inj}_i(V) \\ \text{CBV continuation values} & F & ::= & \alpha \,|\, V::F \,|\, (F_1, F_2) \,|\, \mathsf{inj}_i(F) \,|\, \mu x.c \end{array}$$

The reduction relations $\longrightarrow_{\mathsf{CBN}}$ and $\longrightarrow_{\mathsf{CBV}}$ are the contextual closures of the rules in Fig. 9. ※

$(\overleftarrow{\mu}_{\mathsf{N}})$	$\langle \mu\beta.c \mid E\rangle$	$\longrightarrow \left\{{}^{E}/_{\beta}\right\} c$	$(\overleftarrow{\mu})$	$\langle \mu\beta.c \mid e\rangle$	$\longrightarrow \{{}^{e}/_{\beta}\}\, c$
$(\overrightarrow{\mu})$	$\langle t \mid \mu x.c\rangle$	$\longrightarrow \{{}^{t}/_{x}\}\, c$	$(\overrightarrow{\mu}_{\mathsf{V}})$	$\langle V \mid \mu x.c\rangle$	$\longrightarrow \left\{{}^{V}/_{x}\right\} c$
(ζ_{N})	$R[e]$	$\longrightarrow \langle \mu\alpha.R[\alpha] \mid e\rangle$	(ζ_{V})	$S[t]$	$\longrightarrow \langle t \mid \mu x.S[x]\rangle$
			(ζ_{V})	$T[e]$	$\longrightarrow T[\mu x.\langle x \mid e\rangle]$
$(\to_{\mathsf{N}})$	$\langle \lambda x.t_1 \mid t_2::E\rangle$	$\longrightarrow \langle \{{}^{t_2}/_{x}\}\, t_1 \mid E\rangle$	$(\to_{\mathsf{V}})$	$\langle \lambda x.t_1 \mid V::e\rangle$	$\longrightarrow \left\langle \left\{{}^{V}/_{x}\right\} t_1 \mid e\right\rangle$
$(\wedge_{\mathsf{N}})$	$\langle (t_1, t_2) \mid \mathsf{inj}_i(E)\rangle$	$\longrightarrow \langle t_i \mid E\rangle$	$(\wedge_{\mathsf{V}})$	$\langle (V_1, V_2) \mid \mathsf{inj}_i(e)\rangle$	$\longrightarrow \langle V_i \mid e\rangle$
$(\vee_{\mathsf{N}})$	$\langle \mathsf{inj}_i(t) \mid (E_1, E_2)\rangle$	$\longrightarrow \langle t \mid E_i\rangle$	$(\vee_{\mathsf{V}})$	$\langle \mathsf{inj}_i(V) \mid (e_1, e_2)\rangle$	$\longrightarrow \langle V \mid e_i\rangle$
	CBN			CBV	

where R, S, and T range over contexts of the following grammar:

$$\begin{array}{lrcl} \text{CBN continuation contexts} & R & ::= & \langle t \mid t'::[\,]\rangle \;|\; \langle t \mid ([\,], e)\rangle \;|\; \langle t \mid (E, [\,])\rangle \;|\; \langle t \mid \mathsf{inj}_i([\,])\rangle \\ \text{CBV term contexts} & S & ::= & \langle V \mid [\,]::e\rangle \;|\; \langle ([\,], t) \mid e\rangle \;|\; \langle (V, [\,]) \mid e\rangle \;|\; \langle \mathsf{inj}_i([\,]) \mid e\rangle \\ \text{CBV continuation contexts} & T & ::= & \langle V \mid V'::[\,]\rangle \;|\; \langle V \mid ([\,], e)\rangle \;|\; \langle V \mid (F, [\,])\rangle \;|\; \langle V \mid \mathsf{inj}_i([\,])\rangle \end{array}$$

the (ζ_{N}) only applies under the condition that e is not a (CBN-) continuation value,
the (ζ_{V})-rules only apply under the condition that t is not a (CBV-) term value and e is not a (CBV-) continuation value.

Figure 9: CBN and CBV reduction in System L

In this version, we kept the CBV-rules $(\overleftarrow{\mu})$ and $(\overrightarrow{\mu}_{\mathsf{V}})$, and the CBN-rules $(\overleftarrow{\mu}_{\mathsf{N}})$ and $(\overrightarrow{\mu})$.

The (ζ_{V})-rules (resp. the (ζ_{N})-rule) are new: they were introduced in a slightly more general version in [Wad03], while the version we take here more closely follows [MM09]. These rules are due to our strong restriction on term values (resp. continuation values): the fact that a term is a value is not just the fact that it is not of the form $\mu\alpha.c$, as term values are recursively defined. Therefore if a term t is not of the form $\mu\alpha.c$ but one of its (say direct) subterms is, then t is not a value and there is no CBV-rule to reduce $\langle t \mid \mu x.c\rangle$. Progress then fails if we do not add the (ζ_{V})-rules to pull the first subterm of t that is not a value to the top-level.

These ζ rules also impact rules $(\to)$, $(\wedge)$, and $(\vee)$, which now have to be restricted in order to preserve confluence: for instance in CBV, the fact that $(\mu\alpha.c, t)$ is not a term value means that, when facing a continuation e, $\mu\alpha.c$ will be extracted from the pair and will have the control of computation

$$\langle (\mu\alpha.c, t) \mid e\rangle \longrightarrow_{\zeta_{\mathsf{N}}} \langle \mu\alpha.c \mid \mu x.\langle (x, t) \mid e\rangle\rangle \longrightarrow_{\overleftarrow{\mu}} \left\{{}^{\mu x.\langle (x,t)\mid e\rangle}/_{\alpha}\right\} c$$

and therefore it is clear that, should e be of the form $\mathsf{inj}_2(e')$, the original application of rule $(\wedge)$ would have a totally different semantics:

$$\langle (\mu\alpha.c, t) \mid e\rangle \longrightarrow_{\wedge} \langle t \mid e'\rangle$$

Hence the restriction of $(\to)$, $(\wedge)$, and $(\vee)$ to $(\to_{\mathsf{N}})$, $(\wedge_{\mathsf{N}})$, and $(\vee_{\mathsf{N}})$ in the CBN case, and to $(\to_{\mathsf{V}})$, $(\wedge_{\mathsf{V}})$, and $(\vee_{\mathsf{V}})$ in the CBV case.

Note that in CBN, we decided to make $(\to_{\mathsf{N}})$ collapse the two reduction steps

$$\langle \lambda x.t_1 \mid t_2 :: e\rangle \longrightarrow_{\to} \langle t_2 \mid \mu x.\langle t_1 \mid e\rangle\rangle \longrightarrow_{\vec{\mu}} \langle \{t_2/x\}\, t_1 \mid e\rangle$$

into one step, because $(\vec{\mu})$ has priority anyway.[13] The rule $(\to_{\mathsf{V}})$ is designed by symmetry, collapsing the two steps

$$\langle \lambda x.t_1 \mid V :: e\rangle \longrightarrow_{\to} \langle V \mid \mu x.\langle t_1 \mid e\rangle\rangle \longrightarrow_{\vec{\mu}} \left\langle \left\{V/x\right\} t_1 \mid e\right\rangle$$

and noticing that if t_2 is not a term value, then the original rule $(\to)$ is recovered as follows:

$$\langle \lambda x.t_1 \mid t_2 :: e\rangle \longrightarrow_{\zeta_{\mathsf{V}}} \langle t_2 \mid \mu y.\langle \lambda x.t_1 \mid y :: e\rangle\rangle \longrightarrow_{\to_{\mathsf{V}}} \langle t_2 \mid \mu x.\langle t_1 \mid e\rangle\rangle$$

Of course the extra rules satisfy Subject Reduction, so that we have:

THEOREM 17 (Subject reduction for System L: CBN & CBV)
If $c : (\Gamma \vdash \Delta)$ and either $c \longrightarrow_{\mathsf{CBN}} c'$ or $c \longrightarrow_{\mathsf{CBV}} c'$ then $c' : (\Gamma \vdash \Delta)$.
And similarly for terms and continuations. ※

Proof: Straightforward induction on the rewrite derivation. □

THEOREM 18 (Confluence) $\longrightarrow_{\mathsf{CBN}}$ and $\longrightarrow_{\mathsf{CBV}}$ are confluent. ※

Proof: They are orthogonal higher-order rewrite systems.[14] □

Finally, one could be puzzled by what seems like an asymmetry between CBN and CBV, the latter having more rules and requiring a notion of continuation value while the former does not need a notion of term value. This asymmetry is not due to CBN vs. CBV, but is due to the implication: its main continuation construct $t :: e$ has a term as a direct sub-term, while no term contruct has a continuation as a direct sub-term. It would be the case if we considered the De Morgan dual of implication, namely *subtraction* (see for instance [Cro04]), which would make CBN completely symmetric to CBV.

1.7.2 Two stable fragments

Now it is easy to connect the $\longrightarrow_{\mathsf{CBN}}$ and $\longrightarrow_{\mathsf{CBV}}$ reduction relations of Definition 21 to those of our first attempt in Definition 20, if we concentrate on the two fragments:[15]

DEFINITION 22 (LK^{N} and LK^{V}) Let LK^{N} and LK^{V} be the fragments of System L consisting of $\longrightarrow_{\zeta_{\mathsf{N}}}$-normal forms and $\longrightarrow_{\zeta_{\mathsf{V}}}$-normal forms, respectively. ※

[13] We shall see that it makes the $\mu x.\_$ construct superfluous (in the sense that the fragment without this construct is logically complete, and stable under reduction).

[14] The rewrite system presented in Fig. 9 is a standard (higher-order) rewrite system: we did use a non-standard formulation for the ζ rules based on a grammar for continuation contexts and term contexts as well as on side-conditions ("t is not a term value and e is not a continuation value"), but we could equally have formulated all the cases as standard (but numerous!) rewrite rules.

[15] Namely, those fragments where the reductions of Definition 20 are actually simulated by (the adaptation to System L of) the CPStranslations.

Remark 19

1. Concerning implication only, LK^{N} is the fragment where every continuation of the form $t::e$ is such that e is a continuation value.
2. Concerning implication only, LK^{V} is the fragment where every continuation of the form $t::e$ is such that t is a term value;
3. Also, $\longrightarrow_{\zeta_{\mathsf{N}}}$ and $\longrightarrow_{\zeta_{\mathsf{V}}}$ are terminating reduction relations, so it is easy to normalise a command into one of these fragments, using cuts.
4. Moreover, LK^{N} and LK^{V} are respectively stable under $\longrightarrow_{\mathsf{CBN}}$ and $\longrightarrow_{\mathsf{CBV}}$, so the cuts can be reduced while staying in the fragments.
5. Furthermore, in LK^{N} and LK^{V}, $\longrightarrow_{\mathsf{CBN}}$ and $\longrightarrow_{\mathsf{CBV}}$ of Definition 21 respectively coincide with those of Definition 20.[16]
6. Notice that the encoding of λ-calculus in System L in Definition 14, actually only reaches the fragment LK^{N}, and the simulation lemma (Lemma 9) only involves CBN-reduction.

※

1.7.3 Denotational semantics of CBN and CBV

As anticipated, it is now possible to define CPS-translations of terms, continuations, and commands, respectively denoted $\underline{t}$, $\underline{e}$, $\underline{c}$ for CBN, and $\overline{t}$, $\overline{e}$, $\overline{c}$ for CBV, in a way that preserves reductions:

Theorem 20 (Preservation of reduction)

CBN: If $c_1 \longrightarrow_{\mathsf{CBN}} c_2$ then $\underline{c_1} \longrightarrow_{\beta}^{*} \underline{c_2}$

CBV: If $c_1 \longrightarrow_{\mathsf{CBV}} c_2$ then $\overline{c_1} \longrightarrow_{\beta}^{*} \overline{c_2}$

※

We do not give the details here, which are just technical, but they can be found in e.g. [Wad03].

And these encodings also preserve typing, if Hofmann and Streicher's encoding of types for CBN, and Fischer's encoding of types for CBV, are considered not just for $\to$ (i.e. $\Rightarrow$) but also $\times$ (i.e. $\wedge$) and $+$ (i.e. $\vee$):

Theorem 21 (Preservation of typing)

Assume

$$\Gamma \vdash t : A \;;\; \Delta$$
$$\Gamma \;;\; e : A \vdash \Delta$$
$$c : (\Gamma \vdash \Delta)$$

[16] Almost, since in Definition 21 the rule $(\to_{\mathsf{N}})$ (resp. $(\to_{\mathsf{V}})$) collapses the two steps $\longrightarrow_{\to} \cdot \longrightarrow_{\mu}^{\to}$ (resp. $\longrightarrow_{\to} \cdot \longrightarrow_{\mu_{\mathsf{V}}}^{\to}$) of $\longrightarrow_{\mathsf{CBN}}$ (resp. $\longrightarrow_{\mathsf{CBV}}$) from Definition 20: but in LK^{N} (resp. LK^{V}), there is no other choice than $\longrightarrow_{\mu}^{\to}$ (resp. $\longrightarrow_{\mu_{\mathsf{V}}}^{\to}$) for a top-level reduction that can follow $\longrightarrow_{\to}$ in CBN-reduction (resp. CBV-reduction).

Then

$$\underline{\underline{\Gamma}}\to R, \underline{\underline{\Delta}} \vdash \underline{\underline{t}} : \underline{\underline{A}}\to R \qquad \overline{\overline{\Gamma}}, \overline{\overline{\Delta}}\to R \vdash \overline{\overline{t}} : (\overline{\overline{A}}\to R)\to R$$
$$\underline{\underline{\Gamma}}\to R, \underline{\underline{\Delta}} \vdash \underline{\underline{e}} : (\underline{\underline{A}}\to R)\to R \qquad \overline{\overline{\Gamma}}, \overline{\overline{\Delta}}\to R \vdash \overline{\overline{e}} : \overline{\overline{A}}\to R$$
$$\underline{\underline{\Gamma}}\to R, \underline{\underline{\Delta}} \vdash \underline{\underline{c}} : R \qquad \overline{\overline{\Gamma}}, \overline{\overline{\Delta}}\to R \vdash \overline{\overline{c}} : R$$

Using Hofmann & Streicher's translation of types [HS97]

Using Fischer's translation of types [Fis72]

※

As already mentioned, we can now use these CPS-translations to define categorical semantics for classical proofs:

DEFINITION 23 (Semantics of System L in a response category)

Assume $c : (x_1 : A_1, \ldots, x_n : A_n \vdash \alpha_1 : B_1, \ldots, \alpha_m : B_m)$.

Define the semantics $[\![c]\!]^r_{\mathsf{N}} := [\![\underline{\underline{c}}]\!]$ and $[\![c]\!]^r_{\mathsf{V}} := [\![\overline{\overline{c}}]\!]$, where $[\![t]\!]$ is the semantics, in a response category, of a λ-term t in the CPS-fragment, as defined by the rules of Fig. 3.

Writing K_A for the object corresponding to $\underline{\underline{A}}$, and C_A for R^{K_A}, we have

$$[\![c]\!]^r_{\mathsf{N}} : (C_{A_1} \times \ldots \times C_{A_n} \times K_{B_1} \times \ldots \times K_{B_m}) \longrightarrow R$$

Writing V_A for the object corresponding to $\overline{\overline{A}}$, K_A and for R^{V_A}, we have

$$[\![c]\!]^r_{\mathsf{V}} : (V_{A_1} \times \ldots \times V_{A_n} \times K_{B_1} \times \ldots \times K_{B_m}) \longrightarrow R$$

※

Now, remember that a *control category* is the sub-category of a response category $\mathcal{C}$ whose objects are in $\{R^A | A \in \mathcal{C}\}$, and that $R^A \parr R^B$ denotes $R^{A\times B}$.

DEFINITION 24 (Semantics of System L in control and co-control categories)

Assume $c : (x_1 : A_1, \ldots, x_n : A_n \vdash \alpha_1 : B_1, \ldots, \alpha_m : B_m)$.

CBN The semantics $[\![c]\!]^r_{\mathsf{N}}$ in a response category gives rise, by curryfication, to a morphism

$$[\![c]\!]_{\mathsf{N}} : C_{A_1} \times \ldots \times C_{A_n} \longrightarrow C_{B_1} \parr \ldots \parr C_{B_m}$$

in a control category.

CBV The semantics $[\![c]\!]^r_{\mathsf{V}}$ in a response category gives rise, by curryfication, to a morphism

$$[\![c]\!]_{\mathsf{V}} : K_{B_1} \times \ldots \times K_{B_m} \longrightarrow K_{A_1} \parr \ldots \parr K_{A_n}$$

in a control category.

As the arrow looks "reversed", from the original typing of c, it is more natural to interpret c as the corresponding morphism

$$[\![c]\!]_{\mathsf{V}} : K_{A_1} \otimes \ldots \otimes K_{A_n} \longrightarrow K_{B_1} + \ldots + K_{B_m}$$

in a *co-control category*, the dual of a control category where $\otimes$ denotes the dual of $\parr$ (and the co-product $+$ denotes the dual of the product $\times$).

※

This formalises the idea that CBN-reduction corresponds to a denotational semantics in control categories, while CBV-reduction corresponds to a denotational semantics in co-control categories.

Indeed, from Theorem 20 we get that the semantics validate the reductions:

THEOREM 22 (Soundness of CBN and CBV in control and co-control categories)
If $c \longrightarrow_{\mathsf{CBN}} c'$ then $[\![c]\!]_{\mathsf{N}} = [\![c']\!]_{\mathsf{N}}$
If $c \longrightarrow_{\mathsf{CBV}} c'$ then $[\![c]\!]_{\mathsf{V}} = [\![c']\!]_{\mathsf{V}}$ ※

And we did this by breaking the symmetry between $\wedge$ and $\vee$: Indeed in a control category, $\parr$ is not the dual of $\times$ (equivalently in a co-control category, $+$ is not the dual of $\otimes$).

Conclusion

The work on control and co-control categories is due to Selinger [Sel01] for Parigot's $\lambda\mu$-calculus, showing a duality between CBN and CBV in the categorical sense of duality, and even before a syntactic duality between CBN and CBV was displayed by System L and its variants. It follows preliminary works by Hofmann, Streicher, Reus [HS97, SR98], etc, on the semantics of continuations (where the question of duality between CBV and CBN -in $\lambda\mu$- is conjectured).

In conclusion, many variants of classical proof calculi have been studied; in particular,

- the De Morgan dual of implication in classical logic, namely *subtraction*, can also be given a computational interpretation, as shown for instance by [Cro04];
- variants of Parigot's $\lambda\mu$ have different properties with respect to observational equivalence, separation and η-conversion, etc... [Sau05, Sau08, HZ09, Sau10c, Sau10b, Sau12];
- *control delimiters* can be used to limit the scope of the context that can be captured by a term via control operators, allowing for instance the capture of the *shift* and *reset* operators of [DF89, DF90]; these give rise to *delimited continuations* and can be given a proof-theoretic interpretation [AHS09, HG08, Sau10a];
- other reduction strategies than CBV and CBN have been investigated under the light of the duality of computation, such as Call-by-Need [AF97, AHS11, ADH+12].

Chapter 2

Orthogonality: models for normalisation and witness extraction

Contents

In this chapter we present the concept of *orthogonality* and two applications of it that are useful for classical proof-term calculi: strong normalisation and witness extraction.

Orthogonality was used by Girard in the context of *linear logic* [Gir87] to prove normalisation of cut-elimination and it lies at the heart of its proof semantics based on *coherent spaces*.

The concept of orthogonality has also proved a key concept in the proof theory of classical logic, as it features, just like linear logic does, a duality that is most immediately seen in the form of De Morgan's laws. Indeed, orthogonality is the basis of *classical realisability* [DK00, Kri01], which can be seen as a semantical approach to the Curry-Howard correspondence for classical logic. It also provided a new tool for models of classical proofs, and for proving properties of programs [Par97, MV05, LM08], most notoriously the strong normalisation property. Furthermore, orthogonality shed a new light on the theory of *polarisation* and *focussing* for classical logic, as revealed for instance in [MM09] and explored in Chapter 3.

In this chapter we start by illustrating how proofs of strong normalisation relate to orthogonality. Summarising [BL11b], Section 2.1 rephrases and modularises, with the notion of *orthogonality models*, the well-known techniques by Tait [Tai67, Tai75], Reynolds [Rey72] and Girard [Gir72] for proving the strong normalisation of the simply-typed λ-calculus and

System F. Section 2.2 shows how such models allow the adaptation, to classical proof-term calculi, of strong normalisation proofs, both in the case of a confluent calculus and non-confluent calculus. Section 2.3 then shows how orthogonality models can be used for classical witness extraction, using a technique due to Miquel [Miq09, Miq11].

2.1 Revisiting Proofs of Strong Normalisation for System F

This section presents concepts and a methodology developed in [BL11b]: in particular, we approach the strong normalisation of System F with the notion of *orthogonality model*, adapting the Tait-Girard methodology [Tai75, Gir72]. Although System F is an intuitionistic system, this approach will form the starting point from which the adaptation to classical logic will be explored.

DEFINITION 25 (System F)

The types of System F are given by the following grammar:

$$A, B, \ldots ::= \alpha \mid A \to B \mid \forall \alpha A$$

where α ranges over a denumerable set of elements called *type variables*, and the construct $\forall \alpha A$ binds α in A.

Typing contexts are defined as in Definition 6, using System F types instead of simple types; they will be denoted Γ, Δ, etc.

The *free type variables* of a type A (resp. a typing context Γ) will be denoted $ftv(A)$ (resp. $ftv(\Gamma)$).

The typing system of System F is given in Fig. 10. Derivability of a sequent in System F is denoted $\Gamma \vdash_{\mathsf{F}} M : A$. ※

$$\frac{}{\Gamma, x : A \vdash x : A} \qquad \frac{\Gamma, x : A \vdash M : B}{\Gamma \vdash \lambda x.M : A \to B}$$

$$\frac{\Gamma \vdash M : A \to B \quad \Gamma \vdash N : A}{\Gamma \vdash M\ N : B}$$

$$\frac{\Gamma \vdash M : A}{\Gamma \vdash M : \forall \alpha A}\ \alpha \notin ftv(\Gamma) \qquad \frac{\Gamma \vdash M : \forall \alpha A}{\Gamma \vdash M : \left\{ {}^{B}/_{\alpha} \right\} A}$$

Figure 10: System F

The method to prove strong normalisation is to build a model with

- an interpretation for terms as elements of a set $\mathcal{E}$
- an interpretation for types as (interesting) subsets of $\mathcal{E}$
- such that
  - the interpretation of a term of type A is in the interpretation of A
  - if the interpretation of a term is in there, the term is strongly normalising.

2.1.1 Orthogonality models and the Adequacy Lemma

Definition 26 (Orthogonality models)

An *orthogonality model* is a 4-tuple $(\mathcal{D}, \bot, \mathcal{E}, [\![\_]\!]\_)$ where

- $\mathcal{D}$ is a set of elements called *values*;
- $\bot$ is a relation between values and lists of values called the *orthogonality relation*;
- $\mathcal{E}$ is a superset of $\mathcal{D}$;
- $[\![\_]\!]\_$ is a function mapping every λ-term M (typed or untyped) to an element $[\![M]\!]_\rho$ of $\mathcal{E}$, where ρ is a parameter called *semantic context* mapping term variables to values.
- the following axioms are satisfied:

  (A1) For all ρ, $\vec{v}$, x, if $\rho(x) \bot \vec{v}$ then $[\![x]\!]_\rho \bot \vec{v}$.

  (A2) For all ρ, $\vec{v}$, M_1, M_2, if $[\![M_1]\!]_\rho \bot ([\![M_2]\!]_\rho :: \vec{v})$ then $[\![M_1\ M_2]\!]_\rho \bot \vec{v}$.

  (A3) For all ρ, $\vec{v}$, x, M and for all values u, if $[\![M]\!]_{\rho, x\mapsto u} \bot \vec{v}$ then $[\![\lambda x.M]\!]_\rho \bot (u :: \vec{v})$.

※

In fact, $\mathcal{D}$ and $\bot$ are already sufficient to interpret any System F type A as a set $[\![A]\!]^+$ of values (see Definition 28 below): if types are seen as logical formulae, we can see this construction as a way of building some of their realisability / set-theoretical models.

There is no notion of computation pertaining to values, but the interplay between the interpretation of terms and the orthogonality relation is imposed by the axioms so that the Adequacy Lemma (which relates typing to semantics) holds:

$$\text{If } \vdash_{\mathsf{F}} M : A \text{ then } [\![M]\!] \in [\![A]\!]^+$$

We now assume that we are given an orthogonality model $(\mathcal{D}, \bot, \mathcal{E}, [\![\_]\!]\_)$.

Notation 27 By $\mathcal{D}^*$ we denote the set of lists of values.

If $X \subseteq \mathcal{D}$ and $Y \subseteq \mathcal{D}^*$ let

$$\begin{array}{lcl} X :: Y & := & \{u :: \vec{v} \mid u \in X, \vec{v} \in Y\} \\ X^\bot & := & \{\vec{v} \in \mathcal{D}^* \mid \forall u \in X, u \bot \vec{v}\} \\ Y^\bot & := & \{u \in \mathcal{D} \mid \forall \vec{v} \in Y, u \bot \vec{v}\} \end{array}$$

※

Remark 23 The usual properties of orthogonality hold:

$X \subseteq X^{\bot\bot}$, $X^{\bot\bot\bot} = X^\bot$, and if $X \subseteq X'$ then $X'^\bot \subseteq X^\bot$ ※

We now define the interpretation of types. The intuition is the same as that of Krivine's classical realisability [DK00, Kri01]:

- we first interpret a formula A as a set of "counter-proofs", with the basic constructs that we expect to use in order to refute the formula: for instance the basic way to refute $A_1 \to A_2$ is to provide a "proof" of A_1 and a "counter-proof" of A_2; similarly, the basic way to refute $\forall \alpha A_1$ is to find a suitable interpretation of α and produce a "counter-proof" of A_1 under this interpretation; the set of counter-proofs for atomic formulae is then naturally given by a *valuation*;
- we then define the "proofs" ("realisers" would be a better term) of a formula A as any value that is able to "face all counter-proofs", this latter concept being what the orthogonality relation is precisely there to specify.

DEFINITION 28 (Interpretation of types)

A *valuation* is a function, denoted σ, σ',..., from type variables to subsets of $\mathcal{D}^*$.

Two interpretation of types are defined by simultaneous induction of types, a *positive interpretation* and a *negative interpretation*:

$$\llbracket A \rrbracket^+_\sigma := [A]^{-\perp}_\sigma \qquad \begin{array}{lll} [\alpha]^-_\sigma & := & \sigma(\alpha) \\ [A{\to}B]^-_\sigma & := & \llbracket A \rrbracket^+_\sigma :: [B]^-_\sigma \\ [\forall\alpha A]^-_\sigma & := & \bigcup_{Y\subseteq\mathcal{D}^*} [A]^-_{\sigma,\alpha\mapsto Y} \end{array}$$

We then define the interpretation of typing contexts:

$$\llbracket \Gamma \rrbracket_\sigma := \{\rho \mid \forall (x\!:\!A) \in \Gamma, \rho(x) \in \llbracket A \rrbracket^+_\sigma\}$$

※

This approach begs the question: why is it the case that counter-proofs are defined more primitively than proofs? As described for instance in [MM09] (and in the rest of this thesis), this is simply a coincidence about the two type constructs we use in System F: they both have a "negative polarity", if we see them in the more general context of *polarised logic*, as we shall discuss in Chapter 3. Second-order quantification is often used in the field of realisability (and elsewhere) to encode other logical connectives, which made the negative approach prevalent in that field, with counter-proofs being defined first and proofs being defined by orthogonality. With primitive connectives that have a "positive polarity", such as intuitionistic disjunction, proofs would be defined first and counter-proofs would be defined by orthogonality. We shall come back to that discussion in the next chapters.

REMARK 24 We have the usual properties of substitutions:

$$\left[\left\{{}^{B}\!/_{\alpha}\right\} A\right]^-_\sigma = [A]^-_{\sigma,\alpha\mapsto[B]^-_\sigma} \quad \text{and} \quad \left[\!\left[\left\{{}^{B}\!/_{\alpha}\right\} A\right]\!\right]^+_\sigma = \llbracket A \rrbracket^+_{\sigma,\alpha\mapsto[B]^-_\sigma}$$

Also notice that the *for all* quantifier is interpreted as an intersection:

$$\llbracket \forall\alpha A \rrbracket^+_\sigma = \bigcap_{Y\subseteq\mathcal{D}^*} \llbracket A \rrbracket^+_{\sigma,\alpha\mapsto Y}$$

※

With these definitions we can prove the Adequacy Lemma:

LEMMA 25 (Adequacy Lemma)

If $\Gamma \vdash_{\mathsf{F}} M\!:\!A$, then for all valuations σ and for all mappings $\rho \in \llbracket \Gamma \rrbracket_\sigma$ we have $\llbracket M \rrbracket_\rho \in \llbracket A \rrbracket^+_\sigma$.

※

Proof: By induction on the derivation of $\Gamma \vdash M\!:\!A$, using axioms (A1), (A2) and (A3). See [BL11b]. □

2.1.2 Applicative orthogonality models and Strong Normalisation

DEFINITION 29 (Applicative orthogonality model)

An *applicative orthogonality model* is a 4-tuple $(\mathcal{D}, \mathcal{E}, @, \llbracket \_ \rrbracket\_)$ where:

- $\mathcal{D}$ is a set, $\mathcal{E}$ is a superset of $\mathcal{D}$, $@$ is a (total) function from $\mathcal{E}\times\mathcal{E}$ to $\mathcal{E}$, and $\llbracket \_ \rrbracket\_$ is a function (parameterised by a semantic context) from λ-terms to $\mathcal{E}$.
- $(\mathcal{E}, \mathcal{D}, \perp, \llbracket \_ \rrbracket\_)$ is an orthogonality model,
  where the relation $u \perp \vec{v}$ is defined as $(u@\vec{v}) \in \mathcal{D}$
  (writing $u@\vec{v}$ for $(\ldots(u@v_1)@\ldots@v_n)$ if $\vec{v} = v_1{::}\ldots v_n{::}[\,]$).

※

Remark 26 Axioms (A1) and (A2) are ensured provided that $[\![M\ N]\!]_\rho = [\![M]\!]_\rho @ [\![N]\!]_\rho$ and $[\![x]\!]_\rho = \rho(x)$. These conditions can hold by definition (as in the following example), or can be proved. ※

We now give an applicative orthogonality model to conclude strong normalisation of System F; this will capture, in essence, the Tait-Girard proof methodology [Tai75, Gir72]. The model is here a *term model*, in that $\mathcal{E}$ is the set of all λ-terms and a λ-term is interpreted as itself.

Example 6 (A term-model for Strong Normalisation)

Let $\mathcal{D}$ be the set of strongly-normalising λ-terms, and let $\mathcal{E}$ be set of all λ-terms. We define $u \perp \vec{v}$ as $(u@\vec{v}) \in \mathsf{SN}$, and the interpretation of terms as follows:

$$\begin{array}{lcl} [\![x]\!]_\rho & := & \rho(x) \\ [\![M_1\ M_2]\!]_\rho & := & [\![M_1]\!]_\rho\ [\![M_2]\!]_\rho \\ [\![\lambda x.M]\!]_\rho & := & \lambda x.[\![M]\!]_{\rho, x\mapsto x} \end{array}$$

Requirement 3 is a consequence of anti-reduction:
If $\left\{{}^{P}\!/_{x}\right\} M\ \vec{N} \in \mathsf{SN}$ and $P \in \mathsf{SN}$ then $(\lambda x.M)\ P\ \vec{N} \in \mathsf{SN}$.
Note that for all $\vec{N} \in \mathsf{SN}^*$ and all term variables x, $x \perp \vec{N}$.
Hence, for all valuations σ and all types A, $x \in [\![A]\!]^+_\sigma$.
We apply the Adequacy Lemma (Lemma 25):
If $\Gamma \vdash M : A$, then for all valuation σ and all mapping $\rho \in [\![\Gamma]\!]_\sigma$ we have $[\![M]\!]_\rho \in \mathsf{SN}$.
Hence, $M \in \mathsf{SN}$. ※

In summary, we have defined a family of models for the (polymorphically) typed λ-calculus, and presented one instance with which strong normalisation could be inferred. In [BL11b] we presented other instances of orthogonality models, based for instance on intersection types. Unlike usual models (e.g. CCC), orthogonality models do not necessarily equate terms up to β-reduction (if $M \longrightarrow_\beta N$, we do not necessarily have $[\![M]\!] = [\![N]\!]$). This allows us to build a model where $[\![M]\!] = M$, from which we can infer strong normalisation of typed terms (an instance of CCC would be useless for this).

2.2 Adapting the approach to classical calculi

Orthogonality was used by Parigot to prove strong normalisation of CBN $\lambda\mu$-calculus [Par97]. For their non-confluent calculus, Barbanera & Berardi [BB96] adapted the Tait-Girard reducibility technique with "symmetric reducibility candidates". The key idea in both cases is still that a type A is interpreted as a pair of two orthogonal sets:[1]

- a set $[\![A]\!]^+$ of proof(-terms)
- a set $[\![A]\!]^-$ of counter-proof(-terms)

...satisfying some saturation property (like reducibility candidates do).

In the proof of strong normalisation of System F that we presented in the previous section, the notion of saturation that holds for $[\![A]\!]^+$ is that it is closed under bi-orthogonal ($[\![A]\!]^{+\perp\perp} = [\![A]\!]^+$). In particular, in an applicative term model, the fact that $[\![A]\!]^+$ is closed under bi-orthogonal allows to derive, from axiom (A3) of the orthogonality relation, the property that

[1] Two sets $\mathcal{U}$ and $\mathcal{V}$ are *orthogonal* if $\forall t \in \mathcal{U}, \forall u \in \mathcal{V}, t \perp u$.

if $(\left\{{}^{N}/_{x}\right\} M)\ N_1 \dots N_n \in [\![A]\!]^+$ and $N, N_1, \dots, N_n \in \mathcal{D}$, then $(\lambda x.M)N\ N_1 \dots N_n \in [\![A]\!]^+$. Although not explicitly used in our proof of strong normalisation (Example 6), this property lies in the background and is often explicitly used in more traditional presentations of the reducibility technique [Tai75, Gir72]. In brief, the technique works because the interpretation of a type is closed under "head-anti-reduction".

This is also the approach for classical proof-term calculi, in particular for a confluent calculus such as the Parigot's $\lambda\mu$ [Par97].

2.2.1 The case of a confluent calculus

In this section we take the example of the LK^{N} fragment of System L (Definition 22), with $\Rightarrow$ as the only connective, and considering the reduction relation CBN. We can also prove strong normalisation by building a term model based on orthogonality:

Three rewrite rules apply:

$$\begin{array}{lccc} (\to) & \langle \lambda x.t_1 \mid t_2 :: E\rangle & \longrightarrow & \langle \left\{{}^{t_2}/_{x}\right\} t_1 \mid E\rangle \\ (\overleftarrow{\mu}_{\mathsf{N}}) & \langle \mu\beta.c \mid E\rangle & \longrightarrow & \left\{{}^{E}/_{\beta}\right\} c \\ (\overrightarrow{\mu}) & \langle t \mid \mu x.c\rangle & \longrightarrow & \left\{{}^{t}/_{x}\right\} c \end{array}$$

Therefore we can adapt the axiom (A3) of Definition 26 as follows:

DEFINITION 30 (Orthogonality model for System LK^{N})

An *orthogonality model* for System LK^{N} is given by $(\mathcal{D}_t, \mathcal{D}_e, \bot)$ where $\mathcal{D}_t$ is a set of terms, $\mathcal{D}_e$ is a set of continuations, and $\bot$ is a relation between $\mathcal{D}_t$ and $\mathcal{D}_e$, which can be seen as a set of commands, and satisfying the following *saturation requirements*:

If $\langle \left\{{}^{t_2}/_{x}\right\} t_1 \mid e\rangle\, \rho \in \bot$ then $\langle \lambda x.t_1 \mid t_2 :: e\rangle\, \rho \in \bot$

If $(\left\{{}^{E}/_{\beta}\right\} c)\rho \in \bot$ and $E \in \mathcal{D}_e$ then $\langle \mu\beta.c \mid E\rangle\, \rho \in \bot$

If $(\left\{{}^{t}/_{x}\right\} c)\rho \in \bot$ and $t \in \mathcal{D}_t$ then $\langle t \mid \mu x.c\rangle\, \rho \in \bot$

where $c\rho$ denotes the capture-avoiding application, to c, of a substitution ρ (mapping term variables to terms and continuation variables to continuations). ※

DEFINITION 31 (Interpretation of types for System LK^{N})

A *valuation* is a function, denoted σ, σ',..., from type variables to subsets of $\mathcal{D}_e$ that only contain value continuations.

Two interpretation of types are defined by simultaneous induction of types, a *positive interpretation* and a *negative interpretation*:

$$\begin{array}{rcl} & & [\alpha]^-_\sigma := \sigma(\alpha) \\ & & [A{\to}B]^-_\sigma := [\![A]\!]^+_\sigma :: [\![B]\!]^-_\sigma \\ [\![A]\!]^+_\sigma := [A]^{-\bot}_\sigma & & [\![A]\!]^-_\sigma := [A]^{-\bot\bot}_\sigma \end{array}$$

where $X :: Y$ denotes $\{u :: E \mid u \in X, E \in Y\}$ for any $X \subseteq \mathcal{D}_t$ and $Y \subseteq \mathcal{D}_e$.

We then define the interpretation of a typing context (i.e. a pair of a typing context for term variables and a typing context for continuation variables):

$$[\![\Gamma, \Delta]\!]_\sigma := \{\rho \mid \forall (x : A) \in \Gamma, \rho(x) \in [\![A]\!]^+_\sigma, \text{ and } \forall (\alpha : A) \in \Delta, \rho(\alpha) \in [\![A]\!]^-_\sigma\}$$

※

Lemma 27 (Adequacy Lemma for System LK^{N})

1. If $\Gamma \vdash_{\mathsf{L}} t : A \;;\; \Delta$, then for all valuations σ and for all $\rho \in [\![\Gamma, \Delta]\!]_\sigma$ we have $t\rho \in [\![A]\!]^+_\sigma$.
2. If $\Gamma \;;\; e : A \vdash_{\mathsf{L}} \Delta$, then for all valuations σ and for all $\rho \in [\![\Gamma, \Delta]\!]_\sigma$ we have $e\rho \in [\![A]\!]^-_\sigma$.
3. If $c : (\Gamma \vdash_{\mathsf{L}} \Delta)$, then for all valuations σ and for all $\rho \in [\![\Gamma, \Delta]\!]_\sigma$ we have $c\rho \in \bot$.

where $t\rho$, $e\rho$, and $c\rho$ denotes the capture-avoiding application of ρ, seen as a substitution, to t, e, and c, respectively. ※

Proof: By simultaneous induction on the typing derivations, using the axioms about $\bot$. □

Example 7 (Strong Normalisation of System LK^{N})

We define $\mathcal{D}_t$ to be the set of strongly normalising terms and $\mathcal{D}_e$ to be the set of strongly normalising continuations. We define the orthogonality relation $\bot$ between $\mathcal{D}_t$ and $\mathcal{D}_e$ as those commands that are strongly normalising.[2]

We can check that the saturation requirements are met by purely syntactical/rewriting reasoning, but it only works because there is at most one way to reduce the top-level command.

Take σ to map every type variable to $\mathcal{D}_e$. Notice that term variables are in every $[\![A]\!]^+_\sigma$ and continuation variables are in every $[\![A]\!]^-_\sigma$, and that the identity substitution ρ is in every $[\![\Gamma, \Delta]\!]_\sigma$.

We then apply the Adequacy Lemma with σ and ρ, and get that every typed term, continuation, and command is strongly normalising for $\longrightarrow_{\mathsf{CBN}}$. ※

In this section, we have proved the strong normalisation of the confluent calculus LK^{N} for classical logic, a CBN-fragment of System L. We could have done it along the same lines for the full syntax of System L (but still with the confluent reduction $\longrightarrow_{\mathsf{CBN}}$), but dealing with the extra constructs and extra reductions (ζ_{N}) would have meant a heavier machinery (along the lines of [MM09, CMM10, MM13]). We aimed instead at simplicity, which emphasises the connection with the orthogonality models for System F, and those that we present in the next section.

In summary, in a confluent calculus such as the LK^{N}, building the positive and negative interpretations of a type A can be described as follows:

	Sets of terms	Sets of continuations
Stage 1		$Y_0 := [A]^-$
Stage 2	$X_1 := Y_0^\bot$	$Y_1 := Y_0^{\bot\bot}$
Finished	$[\![A]\!]^+ := X_1$	$[\![A]\!]^- := Y_1$

The construction is finished in 2 steps, because the sets X_1 and Y_1, which are orthogonal, already have all of the saturation properties required to contain all the terms and continuations of type A, which is checked when proving the Adequacy Lemma.

In other words, closure under bi-orthogonality provides adequate saturation properties.

2.2.2 The case of a non-confluent calculus

Let us now consider the situation of a non-confluent calculus such as System L with its original reduction system

[2] The notion of strong normalisation in the definition of $\mathcal{D}_t$, $\mathcal{D}_e$ and $\bot$ is of course considered for $\longrightarrow_{\mathsf{CBN}}$.

$$\begin{array}{lccl}(\to) & \langle \lambda x.t_1 \mid t_2 :: e\rangle & \longrightarrow & \langle t_2 \mid \mu x.\langle t_1 \mid e\rangle\rangle \\ (\overleftarrow{\mu}) & \langle \mu\beta.c \mid e\rangle & \longrightarrow & \{^{e}/_{\beta}\}\, c \\ (\overrightarrow{\mu}) & \langle t \mid \mu x.c\rangle & \longrightarrow & \{^{t}/_{x}\}\, c\end{array}$$

The Adequacy Lemma might still work if we had the saturation requirements:

If $\langle t_2 \mid \mu x.\langle t_1 \mid e\rangle\rangle\, \rho \in \bot$ then $\langle \lambda x.t_1 \mid t_2 :: e\rangle\, \rho \in \bot$
If $(\{^{e}/_{\beta}\}\, c)\rho \in \bot$ and $e \in \mathcal{D}_e$ then $\langle \mu\beta.c \mid e\rangle\, \rho \in \bot$
If $(\{^{t}/_{x}\}\, c)\rho \in \bot$ and $V \in \mathcal{D}_t$ then $\langle t \mid \mu x.c\rangle\, \rho \in \bot$

But in any case, because of non-confluence, these requirements are not met if we define $\mathcal{D}_t$ to be the set of strongly normalising terms and $\mathcal{D}_e$ to be the set of strongly normalising continuations, and $\bot$ the set of strongly normalising commands.[3]

This means that because of non-confluence, we need to change our notion of saturation, so that $[\![A]\!]^+$ and $[\![A]\!]^-$ respectively contain enough terms and continuations for the Adequacy Lemma to hold, and because of that change, the pair $([\![A]\!]^+, [\![A]\!]^-)$ will not be constructed in 2 steps as in the confluent case, but in infinitely many steps:

	Sets of terms		Sets of continuations	
Stage 1	X_0	$\bot$	Y_0	not saturated
Stage 2	X_1	$\bot$	Y_1	not saturated
Stage 3	X_2	$\bot$	Y_2	not saturated
	$\dots$	$\bot$	$\dots$	$\dots$
Stage ∞	X_∞	$\bot$	Y_∞	saturated
Finished	$[\![A]\!]^+$	$\bot$	$[\![A]\!]^-$	saturated

We get a saturated pair of sets in infinitely many steps (via a fixpoint construct). In [LM08], we showed that the fixpoint construct could not be captured by a bi-orthogonal completion step.

We now see the details of the technique. In the rest of this section, we fix $\bot$ to be the set of strongly normalising commands.

DEFINITION 32 (Orthogonality and saturation)

Let Lab_t denote the set of term variables and Lab_e denote the set of continuation variables.

Given a set $\mathcal{U}$ of terms and a set $\mathcal{V}$ of continuations, the pair $(\mathcal{U}, \mathcal{V})$ is

- *orthogonal* if $\forall t \in \mathcal{U}, \forall u \in \mathcal{V}, t \perp u$
- *saturated* if the following two conditions hold
  1. $\mathsf{Lab}_t \subseteq \mathcal{U}$ and $\mathsf{Lab}_e \subseteq \mathcal{V}$
  2. $\{\mu\alpha.c \,|\, \forall e \in \mathcal{V}, \{^{v}/_{x}\}\, c \in \bot\} \subseteq \mathcal{U}$ and $\{\mu x.c \,|\, \forall t \in \mathcal{U}, \{^{v}/_{x}\}\, t \in \bot\} \subseteq \mathcal{V}$.

A set of terms (resp. continuations) is said to be *simple* if it is non-empty and it contains no term of the form $\mu\alpha.c$ (resp. $\mu x.c$).

For every set X of terms (set Y of continuations), we define a function

$$\Phi_X(\mathcal{W}) := X \cup \mathsf{Lab}_t \cup \{\mu\alpha.c \,|\, \forall e \in \mathcal{W}, \{^{e}/_{\alpha}\}\, c \in \bot\}$$

resp.

$$\Phi_Y(\mathcal{W}) := Y \cup \mathsf{Lab}_e \cup \{\mu x.c \,|\, \forall t \in \mathcal{W}, \{^{t}/_{x}\}\, c \in \bot\}$$

※

[3] The notion of strong normalisation in the definition of $\mathcal{D}_t$, $\mathcal{D}_e$ and $\bot$ is now considered for the full reduction relation $\longrightarrow$.

LEMMA 28 Given a set of terms X_0 and a set of continuations Y_0,

1. Φ_{X_0} and Φ_{Y_0} are anti-monotonic.[4]
2. Hence, $\Phi_{X_0} \circ \Phi_{Y_0}$ is monotonic and admits a fixpoint X_∞, with $\Phi_{X_0}(\Phi_{Y_0}(X_\infty)) = X_\infty$.
3. Writing Y_∞ for $\Phi_{Y_0}(X_\infty)$, we clearly have
$$X_\infty = X \cup \mathsf{Lab}_t \cup \{\mu\alpha.c \,|\, \forall e \in Y_\infty, \{e/\alpha\}\, c \in \bot\}$$
$$Y_\infty = Y \cup \mathsf{Lab}_e \cup \{\mu x.c \,|\, \forall t \in X_\infty, \{t/x\}\, c \in \bot\}$$
4. So (X_∞, Y_∞) is saturated, and a quick case analysis shows that it is orthogonal if X_0 and Y_0 are simple and orthogonal to each other.
5. Finally, $X_0 \subseteq X_\infty$ and $Y_0 \subseteq Y_\infty$.

We finally define $\mathsf{satur}(X_0, Y_0)$ as (X_∞, Y_∞). ※

DEFINITION 33 (Interpretation of types for System L)

A *valuation* is a function, denoted σ, σ',..., from type variables to orthogonal pairs of simple sets.

Two interpretation of types are defined by simultaneous induction of types, a *positive interpretation* and a *negative interpretation*:

$$\begin{array}{rcl} ([a]^+_\sigma, [a]^-_\sigma) & := & \sigma(\alpha) \\ ([A{\to}B]^+_\sigma, [A{\to}B]^-_\sigma) & := & (\{\lambda x.t \mid \lambda x.t \in ([\![A]\!]^+_\sigma :: [\![B]\!]^-_\sigma)^\bot\}, [\![A]\!]^+_\sigma :: [\![B]\!]^-_\sigma) \\ \\ ([\![A]\!]^+_\sigma, [\![A]\!]^-_\sigma) & := & \mathsf{satur}([A]^+_\sigma, [A]^-_\sigma) \end{array}$$

Again, we define

$$[\![\Gamma, \Delta]\!]_\sigma := \{\rho \mid \forall (x\!:\!A) \in \Gamma, \rho(x) \in [\![A]\!]^+_\sigma, \text{ and } \forall (\alpha\!:\!A) \in \Delta, \rho(x) \in [\![A]\!]^-_\sigma\}$$

※

Now notice the difference with Definition 31: the definition of $[A{\to}B]^-_\sigma$ is the same but if we just took $[\![A{\to}B]\!]^+_\sigma$ to be its orthogonal, the pair $([\![A{\to}B]\!]^+_\sigma, [A{\to}B]^-_\sigma)$ would not be saturated, as we have already seen; so instead we take all of the abstractions in the orthogonal of $[A{\to}B]^-_\sigma$ and form an orthogonal (but not saturated) pair of simple sets $([A{\to}B]^+_\sigma, [A{\to}B]^-_\sigma)$. Then we saturate that pair into $([\![A{\to}B]\!]^+_\sigma, [\![A{\to}B]\!]^-_\sigma)$, which is orthogonal and saturated:

LEMMA 29 (Interpretations of types are orthogonal and saturated)

For all valuations σ and all types A, $([\![A]\!]^+_\sigma, [\![A]\!]^-_\sigma)$ is orthogonal and saturated. ※

The rest is now just as in the CBN case:

LEMMA 30 (Adequacy Lemma for System L)

1. If $\Gamma \vdash_{\mathsf{L}} t\!:\!A \,;\, \Delta$, then for all valuations σ and for all $\rho \in [\![\Gamma, \Delta]\!]_\sigma$ we have $t\rho \in [\![A]\!]^+_\sigma$.
2. If $\Gamma \,;\, e\!:\!A \vdash_{\mathsf{L}} \Delta$, then for all valuations σ and for all $\rho \in [\![\Gamma, \Delta]\!]_\sigma$ we have $e\rho \in [\![A]\!]^-_\sigma$.
3. If $c\!:\!(\Gamma \vdash_{\mathsf{L}} \Delta)$, then for all valuations σ and for all $\rho \in [\![\Gamma, \Delta]\!]_\sigma$ we have $c\rho \in \bot$.

where $t\rho$, $e\rho$, and $c\rho$ denotes the capture-avoiding application of ρ, seen as a substitution, to t, e, and c, respectively. ※

Proof: By simultaneous induction on the typing derivations, using the Lemma 29. □

[4] In other words for Φ_{X_0}, if $\mathcal{W} \subseteq \mathcal{W}'$ then $\Phi_{X_0}(\mathcal{W}) \supseteq \Phi_{X_0}(\mathcal{W}')$. And similarly for Φ_{Y_0}.

THEOREM 31 (Strong Normalisation of System L)

Take σ to map every type variable to the orthogonal pair $(\mathsf{Lab}_t, \mathsf{Lab}_e)$ of simple sets. Notice again that the identity substitution ρ is in every $[\![\Gamma, \Delta]\!]_\sigma$.

We then apply the Adequacy Lemma with σ and ρ, and get that every typed term, continuation, and command is strongly normalising for $\longrightarrow$. ※

The points to remember are

- As for System F, we have proved strong normalisation by building a term model
  - which does not equate terms up to reduction (non-confluence would make that very problematic)
  - where axiom (A3) is replaced by a saturation property.
- Because of non-confluence,
  - saturation has to be a property of pairs $([\![A]\!]^+_\sigma, [\![A]\!]^-_\sigma)$, not a property of each component separately;
  - saturating is difficult (adding terms in one component of the pair affects the other component), and obtained by a fixpoint construction.

  As shown in [LM08], the saturation process is not just a bi-orthogonality completion: if $(\mathcal{U}, \mathcal{V})$ is orthogonal, then $(\mathcal{U}^{\perp\perp}, \mathcal{V}^{\perp\perp})$ is orthogonal but not necessarily saturated.

2.3 Orthogonality models for extracting witnesses from classical proofs

We now show how to extract a witness from a classical proof of a Σ^0_1-formula, i.e. a closed formula of the form $\exists a A(a)$ where $A(a)$ is a quantifier-free formula of arithmetics.

The technique is due to Miquel [Miq09, Miq11], we simply adapted it to our proof-term calculus for classical logic, and somewhat simplified it using the concepts and notations of the previous sections.

We work in a particular setting where such a formula is expressed in the shape of $\neg\forall a \neg \mathsf{isnull}(e(a))$, the grammar of formulae being defined as follows:

DEFINITION 34 (Expressions and Formulae)

$$\begin{array}{lllll} \textbf{Expressions} & u, u', \ldots & ::= & a \,|\, \mathsf{0} \,|\, \mathsf{s}(u) \,|\, u + u' \,|\, u \times u' \,|\, u \leq u' \\ \textbf{Formulae} & A, B, \ldots & ::= & \mathsf{isnull}(u) \,|\, A \to B \,|\, \forall a A \end{array}$$

We represent integers as expressions: let $\overline{0} := \mathsf{0}$ and, for all integers n, let $\overline{n+1} := \mathsf{s}(\overline{n})$.

We define $\neg A := A \to \mathsf{isnull}(\overline{1})$. ※

This shape for a Σ^0_1-formula brings no loss of generality. Moreover, such an expression as $u(a)$, with one free variable a, expresses a primitive recursive function from $\mathbb{N}$ to $\mathbb{N}$.

We will now build an orthogonality model that we will use for witness extraction. As in the previous sections, each formula A will be interpreted as a set $[\![A]\!]^+_\sigma$ of terms and a set $[\![A]\!]^-_\sigma$ of continuations, terms and continuations being those of LK^N.

The extraction mechanism itself will be given by the reductions of LK^{N}, and more precisely by **root** CBN-reduction, which we denote $\longrightarrow_{\mathsf{CBNr}}$.[5]

In other words, from a proof of $\neg\forall a \neg \mathsf{isnull}(u(a))$ in (the extension to arithmetic of) System L, we will perform $\longrightarrow_{\mathsf{CBNr}}$ -reduction until we reach (in a provably finite number of steps) a command where we can directly read a witness.

For this we need to express numerals as proof-terms. We simply use Church's numerals in λ-calculus (see e.g. [Bar84]) and encode them in LK^{N} with Definition 14:

Definition 35 (Church's numerals as terms)

$$\begin{array}{lcl} c_0 & := & \langle x \mid \alpha \rangle \\ c_{n+1} & := & \langle f \mid (\mu\alpha.c_n) :: \alpha \rangle \\ \underline{n} & := & \lambda x.\lambda f.\mu\alpha.c_n \end{array}$$

※

Remark 32 Doing the same thing with the λ-terms for the successor function and the recursion function, we get two LK^{N} terms **s** and **rec** such that, for all t, u_0, u_1, for all value continuations E, and all integer n,

$$\begin{array}{lcl} \langle \mathbf{s} \mid \underline{n} :: t :: E \rangle & \longrightarrow^*_{\mathsf{CBNr}} & \langle t \mid \underline{n+1} :: E \rangle \\ \langle \mathbf{rec} \mid u_0 :: u_1 :: \underline{0} :: E \rangle & \longrightarrow^*_{\mathsf{CBNr}} & \langle u_0 \mid E \rangle \\ \langle \mathbf{rec} \mid u_0 :: u_1 :: \underline{n+1} :: E \rangle & \longrightarrow^*_{\mathsf{CBNr}} & \langle u_1 \mid \underline{n} :: (\mu\alpha.\langle \mathbf{rec} \mid u_0 :: u_1 :: \underline{n} :: \alpha \rangle) :: E \rangle \end{array}$$

using the simulation of β-reduction by $\longrightarrow_{\mathsf{CBN}}$.[6]

Let $\mathbf{ifz} := \lambda n x_0 x_1.\mu\alpha.\langle \mathbf{rec} \mid x_0 :: (\lambda y_0 y_1.x_1) :: n :: \alpha \rangle$, so that

$$\begin{array}{lcl} \langle \mathbf{ifz} \mid \underline{0} :: u_0 :: u_1 :: E \rangle & \longrightarrow^*_{\mathsf{CBNr}} & \langle u_0 \mid E \rangle \\ \langle \mathbf{ifz} \mid \underline{n+1} :: u_0 :: u_1 :: E \rangle & \longrightarrow^*_{\mathsf{CBNr}} & \langle u_1 \mid E \rangle \end{array}$$

※

Definition 36 (Orthogonality semantics)

Let $\bot$ be an arbitrary set of commands, stable under anti-reduction (if $c \longrightarrow_{\mathsf{CBNr}} c'$ and $c' \in \bot$ then $c \in \bot$).

A valuation σ is a mapping from expression variables (a, etc) to integers.

Given a valuation σ, Fig. 11 defines the interpretation of an expression u as an integer $[\![u]\!]_\sigma$ and a formula A as a set $[\![A]\!]^+_\sigma$ of terms and a set $[\![A]\!]^-_\sigma$ of continuations. ※

Remark 33

1. Clearly, $[\![\overline{n}]\!]_\sigma = n$ for all n and σ.
2. By induction on u we get $\left[\!\left[\left\{ {}^{\overline{n}}/_{a} \right\} u \right]\!\right]_\sigma = [\![u]\!]_{\sigma, a \mapsto n}$, and by induction on A we get $\left[\left\{ {}^{\overline{n}}/_{a} \right\} A \right]^-_\sigma = [A]^-_{\sigma, a \mapsto n}$ and $\left[\!\left[\left\{ {}^{\overline{n}}/_{a} \right\} A \right]\!\right]^+_\sigma = [\![A]\!]^+_{\sigma, a \mapsto n}$.

※

Now, for simplicity we do not specify the exact proof system for arithmetic, nor do we give a typing system corresponding to it through the Curry-Howard correspondence. We assume that it could be built as an extension of Fig. 6, and that the Adequacy Lemma can be proved (along the lines of Lemma 27 for LK^{N}):

[5] The fact that we use CBN-reduction is important to make sure that reduction **can** produce a witness; the fact that we only use root reduction is not, but in order to implement the extraction mechanism deterministically, it is convenient to never have to choose the next redex to reduce.

[6] And the fact that we can do this with root-reduction only is rather clear.

$$
\begin{array}{lcll}
[\![a]\!]_\sigma & := & \sigma(a) & \\
[\![\mathsf{0}]\!]_\sigma & := & 0 & \\
[\![\mathsf{s}(u)]\!]_\sigma & := & [\![u]\!]_\sigma + 1 & \\
[\![u_1 + u_2]\!]_\sigma & := & [\![u_1]\!]_\sigma + [\![u_2]\!]_\sigma & \\
[\![u_1 \times u_2]\!]_\sigma & := & [\![u_1]\!]_\sigma \times [\![u_2]\!]_\sigma & \\
[\![u_1 \leq u_2]\!]_\sigma & := & 1 & \text{if } [\![u_1]\!]_\sigma \leq [\![u_2]\!]_\sigma \\
[\![u_1 \leq u_2]\!]_\sigma & := & 0 & \text{if } [\![u_1]\!]_\sigma > [\![u_2]\!]_\sigma
\end{array}
$$

$$
\begin{array}{lcll}
[\mathsf{isnull}(u)]^-_\sigma & := & \mathcal{E} & \text{if } [\![u]\!]_\sigma \neq 0 \\
[\mathsf{isnull}(u)]^-_\sigma & := & \emptyset & \text{if } [\![u]\!]_\sigma = 0 \\
[A{\to}B]^-_\sigma & := & [\![A]\!]^+_\sigma :: [\![B]\!]^-_\sigma & \\
[\forall a A]^-_\sigma & := & \bigcup_{n\in\mathbb{N}} (\{\underline{n}\} :: [\![A]\!]^-_{\sigma,a\mapsto n}) &
\end{array}
$$

$$
[\![A]\!]^+_\sigma := [A]^{-\perp}_\sigma \qquad [\![A]\!]^-_\sigma := [A]^{-\perp\perp}_\sigma
$$

where $\mathcal{E}$ is the set of all value continuations.

Figure 11: Semantics of expressions and formulae

A closed proof t_0 of a formula $\neg\forall a \neg \mathsf{isnull}(u(a))$ is such that,
for all possible $\perp$ closed under "anti-reduction" (the inverse of $\longrightarrow_{\mathsf{CBNr}}$),
$t_0 \in [\![\neg\forall a \neg \mathsf{isnull}(u(a))]\!]$.

We thus start with such a term t_0.

We now define a term that can check whether an integer is a witness of the property and, depending on this check, continue with one term or another:

Definition 37 (Witness checker)

Let f be the primitive recursive function defined by: for any integer n, $f(n) := [\![u(a)]\!]_{a\mapsto n}$.
Let $\underline{f}$ be a term representing f in the sense that, for any integer n, and term t and any continuation E,

$$\left\langle \underline{f} \mid \underline{n} :: t :: E \right\rangle \longrightarrow^*_{\mathsf{CBNr}} \left\langle t \mid \underline{f(n)} :: E \right\rangle$$

Such a term can be constructed from **s** and **rec**, as the projections, composition, etc are all available in System L.
We define the *witness checker* as follows:

$$d_f := \lambda nxy.\mu\alpha.\left\langle \underline{f} \mid n :: (\lambda p.\mu\alpha_1.\langle \mathbf{ifz} \mid p :: x :: y :: \alpha_1\rangle) :: \alpha \right\rangle$$

※

Lemma 34 (Witness checker property)

For any integer n, any u_0 and u_1 and E, we have
$\langle d_f \mid \underline{n} :: u_0 :: u_1 :: E\rangle \longrightarrow^*_{\mathsf{CBNr}} \langle u_0 \mid E\rangle$ if $f(n) = 0$
$\langle d_f \mid \underline{n} :: u_0 :: u_1 :: E\rangle \longrightarrow^*_{\mathsf{CBNr}} \langle u_1 \mid E\rangle$ if $f(n) \neq 0$ ※

Proof:

$$\begin{array}{rcl}
\langle d_f \mid \underline{n}::u_0::u_1::E\rangle & \longrightarrow^*_{\mathsf{CBNr}} & \left\langle \mu\alpha.\left\langle \underline{f} \mid \underline{n}::(\lambda p.\mu\alpha_1.\langle \mathbf{ifz} \mid p::u_0::u_1::\alpha_1\rangle)::\alpha\right\rangle \mid E\right\rangle \\
& \longrightarrow^*_{\mathsf{CBNr}} & \left\langle \underline{f} \mid \underline{n}::(\lambda p.\mu\alpha_1.\langle \mathbf{ifz} \mid p::u_0::u_1::\alpha_1\rangle)::E\right\rangle \\
& \longrightarrow^*_{\mathsf{CBNr}} & \left\langle \lambda p.\mu\alpha_1.\langle \mathbf{ifz} \mid p::u_0::u_1::\alpha_1\rangle \mid \underline{f(n)}::E\right\rangle \\
& \longrightarrow^*_{\mathsf{CBNr}} & \left\langle \mu\alpha_1.\left\langle \mathbf{ifz} \mid \underline{f(n)}::u_0::u_1::\alpha_1\right\rangle \mid E\right\rangle \\
& \longrightarrow^*_{\mathsf{CBNr}} & \left\langle \mathbf{ifz} \mid \underline{f(n)}::u_0::u_1::E\right\rangle
\end{array}$$

If $f(n)=0$, this reduces to $\langle u_0 \mid E\rangle$. Otherwise, this reduces to $\langle u_1 \mid E\rangle$. □

Definition 38 (Orthogonality and contradicter)

Let **stop** be an arbitrary term and **go** be an arbitrary continuation.

We now take a particular orthogonality set defined by

$$\bot := \{c \mid \text{ there exists } n \text{ such that } f(n)=0 \text{ and } c \longrightarrow^*_{\mathsf{CBNr}} \langle \mathbf{stop} \mid \underline{n}::\mathbf{go}\rangle\}$$

It is closed under anti-CBNr-reduction.

We now define a "contradicter":[7] Let $t_1 := \lambda nx.\mu\alpha.\langle d_f \mid n::(\mu\alpha_0.\langle \mathbf{stop} \mid n::\mathbf{go}\rangle)::x::\alpha\rangle$. ※

Lemma 35 (Behaviour of the contradicter)

For all integer n, and all continuation E in $[\![\neg\mathsf{isnull}(u(\overline{n}))]\!]^-$, we have $t_1 \perp \underline{n}::E$. ※

Proof: We have

$$\langle t_1 \mid \underline{n}::E\rangle \longrightarrow^*_{\mathsf{CBNr}} \langle \lambda x.\mu\alpha.\langle d_f \mid \underline{n}::(\mu\alpha_0.\langle \mathbf{stop} \mid \underline{n}::\mathbf{go}\rangle)::x::\alpha\rangle \mid E\rangle$$

To prove that this is an orthogonal command, we only have to show, as $E \in [\neg\mathsf{isnull}(u(\overline{n}))]^{-\perp\perp}$, that the left-hand side term is orthogonal to every continuation in $[\neg\mathsf{isnull}(u(\overline{n}))]^-$.

Consider such a continuation; it is of the form $t::E'$ with $t \in [\![\mathsf{isnull}(u(\overline{n}))]\!]^+$.

If $f(n) \neq 0$ then $[u(\overline{n})] \neq 0$, so $[\mathsf{isnull}(u(\overline{n}))]^- = \mathcal{E}$ and t is orthogonal to every continuation, in particular E'. So we have

$$\begin{array}{rcl}
 & & \langle \lambda x.\mu\alpha.\langle d_f \mid \underline{n}::(\mu\alpha_0.\langle \mathbf{stop} \mid \underline{n}::\mathbf{go}\rangle)::x::\alpha\rangle \mid t::E'\rangle \\
& \longrightarrow^*_{\mathsf{CBNr}} & \langle d_f \mid \underline{n}::(\mu\alpha_0.\langle \mathbf{stop} \mid \underline{n}::\mathbf{go}\rangle)::t::E'\rangle \\
& \longrightarrow^*_{\mathsf{CBNr}} & \langle t \mid E'\rangle
\end{array}$$

which is in $\bot$.

If $f(n)=0$ then we have

$$\begin{array}{rcl}
 & & \langle \lambda x.\mu\alpha.\langle d_f \mid \underline{n}::(\mu\alpha_0.\langle \mathbf{stop} \mid \underline{n}::\mathbf{go}\rangle)::x::\alpha\rangle \mid t::E'\rangle \\
& \longrightarrow^*_{\mathsf{CBNr}} & \langle d_f \mid n::(\mu\alpha_0.\langle \mathbf{stop} \mid \underline{n}::\mathbf{go}\rangle)::t::E'\rangle \\
& \longrightarrow^*_{\mathsf{CBNr}} & \langle \mu\alpha_0.\langle \mathbf{stop} \mid \underline{n}::\mathbf{go}\rangle \mid E'\rangle \\
& \longrightarrow^*_{\mathsf{CBNr}} & \langle \mathbf{stop} \mid \underline{n}::\mathbf{go}\rangle
\end{array}$$

□

Corollary 36 (Classical witness extraction)

$\langle t_0 \mid t_1::\mathbf{go}\rangle \longrightarrow^*_{\mathsf{CBNr}} \langle \mathbf{stop} \mid \underline{n}::\mathbf{go}\rangle$ for some integer n such that $f(n)=0$. ※

Proof:

From Lemma 35 we get that $t_1 \in [\![\forall a \neg\mathsf{isnull}(u(a))]\!]^+$ and therefore $t_1::\mathbf{go} \in [\neg\forall a \neg\mathsf{isnull}(u(a))]^-$.

Since we have assumed $t_0 \in [\![\neg\forall a \neg\mathsf{isnull}(u(a))]\!]^+$, we have $t_0 \perp t_1::\mathbf{go}$, from which we conclude. □

[7] In the sense that it will contradict what the proof t_0 claims.

In other words, once given a classical proof, we match it against the continuation $t_1 :: \mathbf{go}$ and we are certain that CBNr will produce $\langle \mathbf{stop} \mid \underline{n} :: \mathbf{go} \rangle$ in a finite number of steps, with n being a witness.

For a comparison with other techniques of classical witness extraction, see [Miq09, Miq11].

Conclusion

In summary, we have seen in this chapter a fundamental concept for model construction, namely orthogonality. We built several orthogonality models for various purposes: rephrase strong normalisation proofs for System F, prove the strong normalisation of a confluent proof-term calculus for classical logic as well as a non-confluent calculus (thereby proving cut-elimination), and finally extract witnesses from classical proofs of Σ_1^0-formulae.

In each of those model constructions, we have interpreted formulae first with basic inhabitants (terms or continuations), and then closed their interpretation by a completion process that could simply be taking the bi-orthogonal, in the case of confluent calculi, or a more complex fixpoint, in the case of a non-confluent calculus.

Whether in those constructions we first define the negative interpretation of a formula (as a set of "counter-proofs") or its positive interpretation (as a set of "proofs"), is a question that depends on the formula's *polarity*. This is the topic of the next chapter.

Chapter 3

Polarisation and focussing

Contents

In the previous chapters, we have seen that

- A proof-term syntax, together with a typing system, can be used to represent classical proofs (e.g. System L), such that the symmetry of classical logic is reflected by the symmetry between programs and continuations. The use of classical reasoning corresponds to letting a program capture its continuation.
- A rewrite system on proof-terms can be given to represent cut-elimination, following the intuitions of continuations and control. This gives a non-confluent calculus because (unrestricted) cut-elimination is non-confluent in classical logic, reflected by the fact that programs and continuations compete for the control of computation.
- Still, the rewrite system is strongly normalising on typed proof-terms (i.e. those representing real proofs), showing that cuts are admissible. The proof of strong normalisation was the occasion to introduce orthogonality techniques, although non-confluence requires more, namely specific saturation properties.
- The semantics of classical proofs, or typed proof-terms, is problematic until confluence is recovered in some way.

Back to the main issue, a CCC with $\neg\neg A \simeq A$ collapses, and out of the 3 natural ways to avoid the collapse, namely

1. break the symmetry between $\wedge$ and $\vee$,

2. break the cartesian product,
3. break the curryfication,

we investigate the breaking of symmetry between $\wedge$ and $\vee$.

In Chapter 1 we saw how to break the $\wedge\vee$-symmetry by the CBV/CBN approach. In this chapter, we break the $\wedge\vee$ symmetry by *polarisation*.

3.1 Recovering confluence by polarisation

3.1.1 Symmetry, asymmetry, and η-expansions

We start this section by coming back to a fundamental question: What is symmetrical about Classical Logic? There is definitely a symmetry based on the duality of negation / De Morgan's duality. It can be seen in the truth semantics of formulae, in e.g. truth tables or more generally in a boolean algebra: meet / join and top / bottom are swapped when flipping the order upside-down, and all the axioms of a boolean algebra are preserved.

At the level of proofs, there is also a symmetry that can be seen for instance in the two-sided sequent calculus: the left-introduction rule of a connective is symmetric to the right-introduction rule of its dual connective (in other words, the rules are preserved under duality flipping).

Cut-elimination is symmetrical (e.g. the rewrite system of Fig. 6), but to make semantical sense of it, one breaks the symmetry by making a choice between CBN and CBV that is completely arbitrary.

More interestingly, the following example reveals something asymmetric between the left-introduction of $\vee$ and the right-introduction of $\vee$:

$$\frac{\Gamma, A \vdash \Delta \quad \Gamma, B \vdash \Delta}{\Gamma, A\vee B \vdash \Delta} \qquad \frac{\Gamma, A \vdash \Delta}{\Gamma, A\vee B \vdash \Delta} \qquad \frac{\Gamma, B \vdash \Delta}{\Gamma, A\vee B \vdash \Delta}$$

Of course, we have never claimed that there is a symmetry between the left-introduction of $\vee$ and the right-introduction of $\vee$, but an interesting question is raised by the following situation: it is known (see e.g. [TS00]) that in the sequent calculus, the axiom rule

$$\frac{}{A \vdash A}$$

(say in a context-splitting setting) can be restricted, without losing logical completeness, to the *atomic* axiom rule

$$\frac{}{a \vdash a}$$

Every instance of the general instance can be replaced by a small proof only using atomic axioms, which is proved by induction on A: in particular, transforming the axiom $\dfrac{}{A\vee B \vdash A\vee B}$ into a proof with atomic axioms, we produce

$$\dfrac{\dfrac{\dfrac{\dots}{A \vdash A}}{A \vdash A\vee B} \qquad \dfrac{\dfrac{\dots}{B \vdash B}}{B \vdash A\vee B}}{A\vee B \vdash A\vee B}$$

and then recursively transform $A \vdash A$ and $B \vdash B$ (until all of the used axioms are atomic).

So the interesting question is whether there is a fundamental reason why $\vee$ is decomposed on the left before being decomposed on the right (looking at the bottom-up construction of the proof). Starting the decomposition on the right would have failed.

A related situation occurs with η-expansion in λ-calculus:

In λ-calculus, the use of an axiom corresponds to a variable in the proof-term. Typing the term

$$\lambda z^{A\to B}.z$$

(where we indicate the types of variables as superscripts) uses an axiom on $A\to B$. Typing its η-expansion

$$\lambda z^{A\to B}.\lambda y^{A}.z\ y$$

uses, strictly speaking, an axiom on $A\to B$ and an axiom on A, but as z is immediately applied and its type $A\to B$ immediately destructed, the η-expansion only uses, "morally" speaking, axioms on the smaller formulae A and B. Turning this moral intuition into something formal can be done by taking a proof-term calculus for sequent calculus (rather than natural deduction), as we shall see below, but still: we first have the λ-abstraction, and underneath it the application. Why again do they have to appear in that order?

Indeed, in a classical calculus such a System L, the axiom on $A\to B$ is represented as

$$\frac{}{z:A\to B\vdash z:A\to B\ ;}$$

The η-expansion of z is:

$$\dfrac{\dfrac{}{z:A\to B,y:A\vdash z:A\to B\ ;\alpha:B}\qquad\dfrac{\dfrac{}{z:A\to B,y:A\vdash y:A\ ;\alpha:B}\qquad\dfrac{}{z:A\to B,y:A\ ;\alpha:B\vdash\alpha:B}}{z:A\to B,y:A\ ;(y::\alpha):A\to B\vdash\alpha:B}}{\dfrac{\dfrac{\langle z\mid y::\alpha\rangle:(z:A\to B,y:A\vdash\alpha:B)}{z:A\to B,y:A\vdash\mu\alpha.\langle z\mid y::\alpha\rangle:B\ ;}}{z:A\to B\vdash\lambda y.\mu\alpha.\langle z\mid y::\alpha\rangle:A\to B\ ;}}$$

and then we can recursively transform the axioms on $y:A$ and $\alpha:B$ (until axioms are atomic). Of course, this η-expansion still features the use of $z:A\to B$, but only to implement a contraction (or even more precisely to implement the placing of the formula $A\to B$ where it can be decomposed), not to implement a proper axiom.

Now the η-expansion we used in the λ-calculus to illustrate our point is only one particular instance of η-expansion: the general form

$$M\longrightarrow_\eta\ \lambda y^{A}.M\ y\qquad y\notin\mathsf{FV}(M)$$

can be recovered, by the capture avoiding substitution of M for z, from the axiomatic η-expansion on axiom z:

$$z\longrightarrow_\eta\ \lambda y^{A}.z\ y$$

And in λ-calculus, no matter M (where y is not free), M and $\lambda y.M\ y$ have the same computational behaviour (with respect to β-reduction). In technical terms, M and $\lambda y.M\ y$ cannot be separated (even using untyped terms) [Bar84].

In System L, the η-expansion on axiom z

$$z\longrightarrow_\eta\ \lambda y.\mu\alpha.\langle z\mid y::\alpha\rangle$$

also provides, after instantiation of z by t (where y and α are not free), a general form of η-expansion:

$$t \longrightarrow_\eta \ \lambda y.\mu\alpha.\langle t \mid y::\alpha\rangle$$

which transforms

$$\Gamma \vdash t : A{\to}B \ ; \ \Delta$$

into

$$\dfrac{\dfrac{\Gamma \vdash t : A{\to}B \ ; \ \Delta \qquad \dfrac{\overline{\Gamma, y:A \vdash y:A \ ; \ \alpha:B, \Delta} \qquad \overline{\Gamma, y:A \ ; \ \alpha:B \vdash \alpha:B, \Delta}}{\Gamma, y:A \ ; \ (y::\alpha):A{\to}B \vdash \alpha:B, \Delta}}{\langle t \mid y::\alpha\rangle : (\Gamma, y:A \vdash \alpha:B, \Delta)}}{\dfrac{\Gamma, y:A \vdash \mu\alpha.\langle t \mid y::\alpha\rangle : B \ ; \ \Delta}{\Gamma \vdash \lambda y.\mu\alpha.\langle t \mid y::\alpha\rangle : A{\to}B \ ; \ \Delta}}$$

3.1.2 Towards polarised System L

Now the above general η-expansion can be instantiated with $t = \mu\beta.c$:

$$\mu\beta.c \longrightarrow_\eta \ \lambda y.\mu\alpha.\langle \mu\beta.c \mid y::\alpha\rangle$$

If we put those two terms in context, e.g. facing a continuation $\mu x.c'$, we get that

- $\langle \mu\beta.c \mid \mu x.c'\rangle$ rewrites to
$$\left\{ {}^{\mu\beta.c}\!/_{x} \right\} c' \qquad \text{or} \qquad \left\{ {}^{\mu x.c'}\!/_{\beta} \right\} c$$
- **but** $\langle \lambda y.\mu\alpha.\langle \mu\beta.c \mid y::\alpha\rangle \mid \mu x.c'\rangle$ rewrites only to
$$\left\{ {}^{\lambda y.\mu\alpha.\langle \mu\beta.c \mid y::\alpha\rangle}\!/_{x} \right\} c'$$

If η-convertible terms should have undistinguishable computational behaviour, we must forbid $\left\langle \mu\beta^{A_1\to A_2}.c \mid \mu x^{A_1\to A_2}.c' \right\rangle \longrightarrow \left\{ {}^{\mu x^{A_1\to A_2}.c'}\!/_{\beta} \right\} c$

The grounds for breaking the symmetry in such a way is that $\mu\beta^{A_1\to A_2}.c$ can be η-expanded, but $\mu x^{A_1\to A_2}.c'$ cannot, which reflects the fact that the right-introduction rule for $A_1 \to A_2$ is invertible while its left-introduction rule is not.

In short, when encountering

$$\dfrac{\dfrac{c : (\Gamma \vdash \beta : A_1{\to}A_2, \Delta)}{\Gamma \vdash \mu\beta.c : A_1{\to}A_2 \ ; \ \Delta} \qquad \dfrac{c' : (\Gamma, x : A_1{\to}A_2 \vdash \Delta)}{\Gamma \ ; \ \mu x.c' : A_1{\to}A_2 \vdash \Delta}}{\langle \mu\beta.c \mid \mu x.c'\rangle : (\Gamma \vdash \Delta)}$$

we could consider that the term $\mu\beta.c$ is a "cheater" in the sense that its type $A_1{\to}A_2$ could be proved or inhabited in another way (e.g. with the η-expansion of $\mu\beta.c$), avoiding the critical pair, and solving the non-confluence problem.

In particular, if β is used 0 times in c, or more than once, we can understand the typing tree

$$\dfrac{c : (\Gamma \vdash \beta : A_1{\to}A_2, \Delta)}{\Gamma \vdash \mu\beta.c : A_1{\to}A_2 \ ; \ \Delta}$$

as finishing with a weakening or a contraction. What η-expansion proves is that the proof can be transformed into a proof that finishes with a proper introduction of the implication.

In our earlier example about the connective $\vee$, it is the contrary: its left-introduction rule is invertible while its right-introduction rules are not.

This leads to considering a notion that arose from linear logic [Gir87]: *polarities*.

The intuition for *positive connectives* is that we expect no particular property of their right-introduction rules. These rules are called *synchronous*. In particular for goal-directed proof-search, applying such a rule bottom-up is *a priori* a choice which we may have to backtrack to if we fail to finish the proof. For logical completeness, (right-)weakenings or (right-)contractions may be necessary on a formula with a positive connective at its root.

The intuition for *negative connectives* is that their right-introduction rules *are* invertible. These rules are called *asynchronous*. In goal-directed proof-search we may apply such rules without loss of generality and therefore without creating backtrack points. Also, (right-)weakenings and (right-)contractions (on formulae that have a negative connective at their roots) are superfluous as far as logical completeness is concerned. On the other hand, the right-introduction rules "must interact well with the left-introduction rules" (or the right-introduction rules of the dual connective), in cut-elimination as well as in the expansion of axioms that we described in this section.

Just as in λ-calculus you can *always* inhabit a (non-empty) function type with a λ-abstraction, you can always η-expand an inhabitant of a type whose main connective is negative. Considering the η-expansion rules that we can apply in System L, we can derive the polarities of the three connectives we considered:

$$\begin{array}{llcl} \text{negative} & A{\to}B & t & \longrightarrow & \lambda y.\mu\alpha.\langle t \mid y :: \alpha\rangle \\ \text{negative} & A \wedge B & t & \longrightarrow & (\mu\alpha.\langle t \mid \mathsf{inj}_1(\alpha)\rangle, \mu\gamma.\langle t \mid \mathsf{inj}_2(\gamma)\rangle) \\ \text{positive} & A \vee B & e & \longrightarrow & (\mu x.\langle \mathsf{inj}_1(x) \mid e\rangle, \mu z.\langle \mathsf{inj}_2(z) \mid e\rangle) \end{array}$$

Now in order to solve the confluence problem, we also need to determine how to reduce $\langle \mu\beta.c \mid \mu x.c'\rangle$ when the cut-formula is atomic. This leads to splitting the set of the atomic formulae into positives and negatives as well. Unlike non-atomic formulae, the choice of polarity for each atom is arbitrary, and sometimes called the *bias* [LM09].

Now we can use these ideas to layer System L with polarities:

Definition 39 (Polarised System L) The polarised syntax of formulae is defined as

$$\begin{array}{rcl} P, P', \ldots & ::= & a^+ \mid A{\vee}B \\ N, N', \ldots & ::= & a^- \mid A{\wedge}B \mid A{\to}B \\ A, B, \ldots & ::= & P \mid N \end{array}$$

The syntax for proof-terms, together with their associated forms of typing judgements, is given below:

$$\begin{array}{llcll} -\text{terms} & t^- & ::= & x^- \mid \lambda x.t \mid (t_1, t_2) \mid \mathsf{inj}_i(t) \mid \mu\beta^-.c & \Gamma \vdash t^- : N \,;\, \Delta \\ +\text{terms} & t^+ & ::= & x^+ \mid \mathsf{inj}_i(t) \mid \mu\beta^+.c & \Gamma \vdash t^+ : P \,;\, \Delta \\ \text{terms} & t & ::= & t^+ \mid t^- & \Gamma \vdash t : A \,;\, \Delta \\ & & & & \\ -\text{continuations} & e^- & ::= & \alpha^- \mid t :: e \mid \mathsf{inj}_i(e) \mid \mu x^-.c & \Gamma \,;\, e^- : N \vdash \Delta \\ +\text{continuations} & e^+ & ::= & \alpha^+ \mid (e_1, e_2) \mid \mu x^+.c & \Gamma \,;\, e^+ : P \vdash \Delta \\ \text{continuations} & t & ::= & e^+ \mid e^- & \Gamma \,;\, e : A \vdash \Delta \\ & & & & \\ \text{commands} & c & ::= & \langle t^+ \mid e^+\rangle \mid \langle t^- \mid e^-\rangle & c : (\Gamma \vdash \Delta) \end{array}$$

writing x for either x^+ or x^-. ※

Now that polarities explicitly appear in the syntax of proof-terms, it is easy to reduce $\langle \mu\alpha.c \mid \mu x.c'\rangle$:

$$\langle \mu\alpha^+.c \mid \mu x^+.c'\rangle \longrightarrow \left\{ {}^{\mu x^+.c'}\!/_{\alpha^+} \right\} c \quad \text{and} \quad \langle \mu\alpha^-.c \mid \mu x^-.c'\rangle \longrightarrow \left\{ {}^{\mu\alpha^-.c}\!/_{x^-} \right\} c'$$

This turns into the following rewrite system:

DEFINITION 40 (Reductions in the polarised System L) Again, we define values:

$$\begin{array}{lrcl} \text{term values} & V & ::= & x \,|\, \mathsf{inj}_i(V) \,|\, t^- \\ \text{continuation values} & E & ::= & \alpha \,|\, V::E \,|\, \mathsf{inj}_i(E) \,|\, e^+ \end{array}$$

The reduction relation $\longrightarrow_{\mathsf{F}}$ is defined as the contextual closure of the rules in Fig. 12. ※

$$\begin{array}{lccl} (\to) & \langle \lambda x.t \mid V::E \rangle & \longrightarrow & \left\langle \left\{ {}^{V}\!/_{x} \right\} t \mid E \right\rangle \\ (\wedge) & \langle (t_1, t_2) \mid \mathsf{inj}_i(E) \rangle & \longrightarrow & \langle t_i \mid E \rangle \\ (\vee) & \langle \mathsf{inj}_i(V) \mid (e_1, e_2) \rangle & \longrightarrow & \langle V \mid e_i \rangle \\ & & & \\ (\overleftarrow{\mu}_-) & \langle \mu\beta^-.c \mid E \rangle & \longrightarrow & \left\{ {}^{E}\!/_{\beta^+} \right\} c \\ (\overrightarrow{\mu}) & \langle t^- \mid \mu x^-.c \rangle & \longrightarrow & \left\{ {}^{t^-}\!/_{x^-} \right\} c \\ (\overleftarrow{\mu}) & \langle \mu\beta^+.c \mid e^+ \rangle & \longrightarrow & \left\{ {}^{e^+}\!/_{\beta^+} \right\} c \\ (\overrightarrow{\mu}_+) & \langle V \mid \mu x^+.c \rangle & \longrightarrow & \left\{ {}^{V}\!/_{x^+} \right\} c \\ & & & \\ (\zeta_{\mathsf{F}}) & \langle t^- \mid t^+::e \rangle & \longrightarrow & \langle t^+ \mid \mu x^+.\langle t^- \mid x^+::e \rangle \rangle \\ (\zeta_{\mathsf{F}}) & \langle t^- \mid V::e^- \rangle & \longrightarrow & \langle \mu\alpha.\langle t^- \mid V::\alpha \rangle \mid e^- \rangle \\ (\zeta_{\mathsf{F}}) & \langle t^- \mid \mathsf{inj}_i(e^-) \rangle & \longrightarrow & \langle \mu\alpha.\langle t^- \mid \mathsf{inj}_i(\alpha) \rangle \mid e^- \rangle \\ (\zeta_{\mathsf{F}}) & \langle \mathsf{inj}_i(t^+) \mid e^+ \rangle & \longrightarrow & \langle \mu x^+.\langle \mathsf{inj}_i(x^+) \mid e^+ \rangle \mid t^+ \rangle \end{array}$$

where the (ζ_{F})-rules apply only under the condition that t^+ and e^- are not values.

Figure 12: Rewrite system for polarised System L

As in the CBN and CBV cases, we have:

THEOREM 37 (Confluence and Subject Reduction)
$\longrightarrow_{\mathsf{F}}$ is confluent and satisfies Subject Reduction. ※

Notice that the notion of value is slightly different from that of Definition 21: Indeed, if $\wedge$ is to be taken to be negative, as the dual of (the obviously positive) $\vee$, we can take every pair to be a value (in Definition 21 we stuck to Wadler's presentation [Wad03]); this also removes the need for ζ-rules for pairs. On the other hand, for a continuation $t::e$ to be a value, we require it to be of the form $V::E$, as we no longer recover confluence by opposing left vs. right (terms vs. continuations) but by opposing positives vs. negatives.

Precisely because we now no longer make any distinction based on the left vs. right opposition (terms vs. continuations opposition), this system could equally be given as a one-sided system, merging the syntaxes of terms and continuations, but keeping of course the distinction between positive terms and negative terms.[1] At the level of formulae, we would get 4 connectives $\vee^+$ and $\wedge^+$ of positive polarity, and $\vee^-$ and $\wedge^-$ of negative polarity:

$$\begin{array}{llll} A \vee B & \text{becomes} & A \vee^+ B & \qquad A \to B \quad \text{becomes} \quad A^\perp \vee^- B \\ A \wedge B & \text{becomes} & A \wedge^- B & \qquad (A \to B)^\perp \quad \text{becomes} \quad A \wedge^+ B^\perp \end{array}$$

where $(A \to B)^\perp$ represents the dual of implication: *subtraction* (see e.g. [Cro04]).

[1] Otherwise we would get back to Barbanera and Berardi's symmetric (and non-confluent) λ-calculus, with unclear denotational semantics.

This is what we will do in Section 3.2.

3.1.3 Focussing

Now, the polarised System L presented above, which has been studied at length by Munch-Maccagnoni [MM09, CMM10, MM13], solves non-confluence, not by giving systematic priority to the left (CBV) or to the right (CBN), but by giving priority to the non-invertible side (depending on the connective).

So the system takes advantage of the invertibility properties of the asynchronous rules (right-introduction of negative connectives, left-introduction of positive connectives). Invertibility entails that, in terms of proof-search, you can *chain* the decomposition of every formula of the sequent that has an asynchronous introduction rule, before doing anything else, without loss of generality (i.e. without losing logical completeness).

Now in [AP89, And92], Andreoli proved a more surprising result: *focussing*, that says that, once you have chosen to decompose by a synchronous rule a particular formula in the sequent,[2] you can also chain without loss of generality (i.e. without losing logical completeness) the recursive decomposition of its subformulae by synchronous rules until you reveal a subformula of the opposite polarity (whose decomposition can then be done by asynchronous rules again).

This was in the context of linear logic, whence polarities have come, but it is now understood in other polarised logics (classical or intuitionistic). This result can be expressed as the completeness of a sequent calculus with a *focus* device, which syntactically highlights a formula in the sequent and forces the next proof-search step to decompose it with a synchronous rule, keeping the focus on its newly revealed subformulae. In terms of proof-search, focussing considerably reduces the search space, otherwise heavily redundant when Gentzen-style inference rules are used.

Focussed proofs are proofs that implement such a chaining of synchronous decompositions. The main idea is that focussed proofs are those whose proof-terms systematically use term values and continuation values, in other words, the normal forms for ζ-rules. Of course, such normal forms may feature cuts (ζ-rules introduce cuts), but one should notice the following properties:

Remark 38
Just like $\longrightarrow_{\zeta_{\mathsf{N}}}$ and $\longrightarrow_{\zeta_{\mathsf{V}}}$ (from Definition 21), the relation $\longrightarrow_{\zeta_{\mathsf{F}}}$ is terminating. ※

Definition 41 (LK^{F})
Let LK^{F} be the fragment of System L consisting of $\longrightarrow_{\zeta_{\mathsf{F}}}$-normal forms. ※

Remark 39 Just like LK^{N} and LK^{V} are stable under $\longrightarrow_{\mathsf{CBN}}$ and $\longrightarrow_{\mathsf{CBV}}$, the fragment LK^{F} is stable under $\longrightarrow_{\mathsf{F}}$. ※

Remark 40 These normal form fragments relate to calculi of the literature:

1. LK^{N} is exactly the calculus called LKT [DJS95, DJS97];
2. LK^{V} is exactly the calculus called LKQ [DJS95, DJS97];
3. LK^{F} relates to Liang and Miller's LKF [LM09], and this will be the object of Section 3.2.

※

[2] Positive formula on the right-hand side of a sequent, or a negative formula on its left-hand side

We therefore have 3 versions of the focussing result in classical logic, an unfocussed proof c can be turned into a focussed proof c' (in the sense of LK^{N}, LK^{V}, or LK^{F}) by normalising it with respectively $\longrightarrow_{\zeta_{\mathsf{N}}}$, $\longrightarrow_{\zeta_{\mathsf{V}}}$, or $\longrightarrow_{\zeta_{\mathsf{F}}}$, and normalising it by respectively $\longrightarrow_{\mathsf{CBN}}$, $\longrightarrow_{\mathsf{CBV}}$, or $\longrightarrow_{\mathsf{F}}$ to eliminate cuts and finally obtain a cut-free focussed proof.

3.1.4 Weak η-conversion

Now, the notion of η-conversion that we used as an introduction to polarities in Section 3.1.2, is a *strong* notion of η-conversion:

Inspired by the way we can reduce an axiom on a non-atomic formula into a proof using axioms on smaller formulae, we considered the η-expansion of variables x and α:

$$\begin{array}{rcl} x & \longrightarrow & \lambda y.\mu\alpha.\langle x \mid y :: \alpha\rangle \\ x & \longrightarrow & (\mu\alpha.\langle x \mid \mathsf{inj}_1(\alpha)\rangle, \mu\gamma.\langle x \mid \mathsf{inj}_2(\gamma)\rangle) \\ \alpha & \longrightarrow & (\mu x.\langle \mathsf{inj}_1(x) \mid \alpha\rangle, \mu z.\langle \mathsf{inj}_2(z) \mid \alpha\rangle) \end{array}$$

which we sought to generalise to

$$\begin{array}{rcl} t & \longrightarrow & \lambda y.\mu\alpha.\langle t \mid y :: \alpha\rangle \\ t & \longrightarrow & (\mu\alpha.\langle t \mid \mathsf{inj}_1(\alpha)\rangle, \mu\gamma.\langle t \mid \mathsf{inj}_2(\gamma)\rangle) \\ e & \longrightarrow & (\mu x.\langle \mathsf{inj}_1(x) \mid e\rangle, \mu z.\langle \mathsf{inj}_2(z) \mid e\rangle) \end{array}$$

for *any* term t and any continuation e.

This led to a polarity-based reduction relation that contrasts with the CBN and CBV reduction relations from Chapter 1. But that does not mean that CBN and CBV are incompatible with the concept of η-conversion: they just require *weaker* notions of η-conversion than that discussed above.

The notion of η-conversion that is suitable for CBN are

$$\begin{array}{rcl} t & \longrightarrow & \lambda y.\mu\alpha.\langle t \mid y :: \alpha\rangle \\ t & \longrightarrow & (\mu\alpha.\langle t \mid \mathsf{inj}_1(\alpha)\rangle, \mu\gamma.\langle t \mid \mathsf{inj}_2(\gamma)\rangle) \\ E & \longrightarrow & (\mu x.\langle \mathsf{inj}_1(x) \mid E\rangle, \mu z.\langle \mathsf{inj}_2(z) \mid E\rangle) \end{array}$$

where α has not been substituted by any continuation e but *only* by a continuation value E.

The notion of η-conversion that is suitable for CBV are

$$\begin{array}{rcl} V & \longrightarrow & \lambda y.\mu\alpha.\langle V \mid y :: \alpha\rangle \\ V & \longrightarrow & (\mu\alpha.\langle V \mid \mathsf{inj}_1(\alpha)\rangle, \mu\gamma.\langle V \mid \mathsf{inj}_2(\gamma)\rangle) \\ e & \longrightarrow & (\mu x.\langle \mathsf{inj}_1(x) \mid e\rangle, \mu z.\langle \mathsf{inj}_2(z) \mid e\rangle) \end{array}$$

where x has not been substituted by any term t but *only* by a term value V.

Including these notions of CBN-η-conversion and CBV-η-conversion in the CBN and CBV notions of reduction, is actually necessary if these are to capture the semantics of classical proofs in control and co-control categories, respectively: just like in Theorem 4 we needed η-conversion to make the simply-typed λ-calculus sound and complete with respect to the semantics given by CCC, here we would need the above notions of CBN-η-conversion and CBV-η-conversion in order to turn the implications of Theorem 22 (soundness) into equivalences (soundness and completeness). This is actually what Selinger proved [Sel01] in the context of the $\lambda\mu$-calculus.

3.1.5 Related works

The role of polarities and focussing in classical proof theory has been investigated by a substantial literature, inspired by Girard's linear logic [Gir87]. Following this work and Andreoli's on focussing [AP89, And92], Girard developed in [Gir91] a sequent calculus LC for classical logic with more structure than Gentzen's LK [Gen35], based on an assignment of polarities to classical formulae. Danos, Joinet and Schellinx [DJS95, DJS97] studied semantically meaningful ways to make cut-elimination confluent in the classical sequent calculus, introducing

- the calculi LKT and LKQ mentioned above
- a version of the sequent calculus called LK^{tq} where a *colour* t or q on each formula indicates whether a cut on that formula should be pushed to the right or to the left,
- more restricted versions thereof,

all inspired by the various translations of classical logic into linear and intuitionistic logics. Out of that field, which includes the duality betweenn CBN and CBV,[3] *polarised classical logic* emerged, developed as such by Laurent et al. [Lau02, LQdF05]. It develops and enriches Girard's work on LC, in particular by explaining the proof theory of classical formulae as given by LC as a combination of

- an encoding from classical formulae to polarised classical formulae
- a proof theory for polarised classical logic.

Closer to Andreoli's original line of research, which was motivated by logic programming, Liang and Miller then formalised LKF as a more strongly focussed calculus than that called LK^{F} above; we will study it in the next section.

A useful introduction to that literature can be found in Chapter 2 of Farooque's thesis [Far13].

More recently, Munch-Maccagnoni approached the concept of focussing via orthogonality models [MM09]. He built for the polarised version of System L the same kind of orthogonality model as the one we presented in Section 2.2.1 for the LK^{N}-case, with an interpretation $[\![A]\!]^+$ of a formula A built as the orthogonal or bi-orthogonal of a more basic set of (counter-)proof-terms. He essentially shows that the interpretation $[\![A]\!]^+$ of a formula is generated from its values, in the sense that $[\![A]\!]^+ = ([\![A]\!]^+ \cap \mathcal{V})^{\perp\perp}$ where $\mathcal{V}$ denotes the set of term values (and symmetrically $[\![A]\!]^- = ([\![A]\!]^- \cap \mathcal{E})^{\perp\perp}$ where $\mathcal{E}$ denotes the set of continuation values).

This sheds an interesting light on our definition of

$$[A{\to}B]^-_\sigma := \ [\![A]\!]^+_\sigma :: [\![B]\!]^-_\sigma$$

in our orthogonality models of LK^{N} (Definitions 31 and 36): While in LK^{N} the above construct only considers those inhabitants of $[\![B]\!]^-_\sigma$ that are continuation values anyway, there is in the general case of System L a question of whether we want continuations of the form $t::\mu x.c$,[4] which are not focussed (in the sense of LK^{N} or LK^{F}), in the interpretation $[A{\to}B]^-_\sigma$. The result that $[\![A{\to}B]\!]^- = ([\![A{\to}B]\!]^- \cap \mathcal{E})^{\perp\perp}$ means that an "unfocussed" counter-proof such as $t::\mu x.c$ would be accepted after the bi-orthogonal completion (i.e. in $[\![A{\to}B]\!]^-_\sigma = ([A{\to}B]^-_\sigma)^{\perp\perp}$).

So far we have taken advantage of focussing, i.e. the chaining of synchronous rules, to identify complete fragments LK^{N}, LK^{V}, and LK^{F} of classical sequent calculus proofs.

Although we have discussed invertibility of asynchronous rules, in order to introduce the notion of polarity, we have not forced our proofs to apply asynchronous rules eagerly, before

[3] As revealed by Curien and Herbelin's System L [CH00] and Selinger's control categories [Sel01].

[4] (with $t \in [\![A]\!]^+_\sigma$ and $\mu x.c \in [\![B]\!]^-_\sigma$)

applying other rules (in LK^{F}, $\mu\alpha^{-}.c$ is still an accepted proof of a negative formula such as $A{\to}B$).

This is the main difference with the focussed proof systems in the style of e.g. Liang and Miller [LM09], where e.g. all proofs of $A{\to}B$ finish with the right-introduction of $\to$. In terms of proof-terms, it means that all proof-terms are in η-long normal forms (we can transform every proof-term into a proof-term in that form by a series of η-expansions, but it is always tricky to control the termination of η-expansion without having the types explicitly in the terms).

Coming back to the purely logical level, focussed proofs in the tradition of Miller et al. can be described in terms of "big-step focussing": going up a branch of the proof is an alternation of sychronous and asynchronous phases, which we may consider to be atomic. The next section shows a computational interpretation of that strongly focussed formalism.

3.2 Computational interpretation of a focussed calculus

The starting point of this section is Liang and Miller's LKF [LM09], a variant of the system LK^{F} described in the previous section that forces the asynchronous decomposition of formulae. It is described in purely logical terms and we will see how, by formalising the concept of *big-step focussing*, a Curry-Howard interpretation can be given to LKF, following Zeilberger's work [Zei08a, Zei08b].

We start with the formulae of *polarised classical logic*.

DEFINITION 42 (Polarised formulae)

The syntax of formulae is given by the following grammar

$$\begin{array}{lrcl} \textit{Positive formulae} & P & ::= & a \,|\, A_1 \wedge^{+} A_2 \,|\, A_1 \vee^{+} A_2 \\ \textit{Negative formulae} & N & ::= & a^{\perp} \,|\, A_1 \wedge^{-} A_2 \,|\, A_1 \vee^{-} A_2 \\ \textit{Formulae} & A & ::= & P \,|\, N \end{array}$$

where a ranges over a fixed set of elements called *positive atoms*, and $a^{\perp}$ ranges over a bijective copy of that set ($a \mapsto a^{\perp}$ is the bijection), whose elements are called *negative atoms*.

We extend the bijection between positive and negative atoms into an involutive bijection, called *negation*, between positive and negative formulae:

$(a)^{\perp}$	$:= a^{\perp}$	$(a^{\perp})^{\perp}$	$:= a$
$(A_1 \wedge^{+} A_2)^{\perp}$	$:= A_1^{\perp} \vee^{-} A_2^{\perp}$	$(A_1 \wedge^{-} A_2)^{\perp}$	$:= A_1^{\perp} \vee^{+} A_2^{\perp}$
$(A_1 \vee^{+} A_2)^{\perp}$	$:= A_1^{\perp} \wedge^{-} A_2^{\perp}$	$(A_1 \vee^{-} A_2)^{\perp}$	$:= A_1^{\perp} \wedge^{+} A_2^{\perp}$

※

Following the suggestion made in the previous section, we fold LKF into a 1-sided sequent calculus (hence our interest for the involutive negation), as it is traditional in the field arising from linear logic [Gir87].

EXAMPLE 8 For instance, Peirce's, law, which in the previous chapters and sections we wrote as $((a{\to}b){\to}a){\to}a$, is now $((a^{\perp} \vee^{-} b) \wedge^{+} a^{\perp}) \vee^{-} a$. ※

Definition 43 (Liang-Miller's LKF)

The rules of LKF are given in Fig. 13 for two kinds of sequents:
$\vdash \Theta \Downarrow A$ focussed sequent
$\vdash \Theta \Uparrow \Gamma$ unfocussed sequent
where A is an arbitrary formula, Θ is a multiset of either negative atoms or positive formulae and Γ is a multiset of arbitrary formulae.

Derivability in LKF of the sequents $\vdash \Theta \Downarrow A$ and $\vdash \Theta \Uparrow \Gamma$ is respectively denoted $\vdash_{\mathsf{LKF}} \Theta \Downarrow A$ and $\vdash_{\mathsf{LKF}} \Theta \Uparrow \Gamma$. ※

Synchronous phase	$\dfrac{\vdash \Theta \Downarrow A_1 \qquad \vdash \Theta \Downarrow A_2}{\vdash \Theta \Downarrow A_1 \wedge^+ A_2}$ $\qquad$ $\dfrac{\vdash \Theta \Downarrow A_i}{\vdash \Theta \Downarrow A_1 \vee^+ A_2}$
End of synchronous phase	$\dfrac{\vdash \Theta \Uparrow N}{\vdash \Theta \Downarrow N}$ $\qquad$ $\dfrac{}{\vdash \Theta \Downarrow a}\ a^\perp \in \Theta$
Asynchronous phase	$\dfrac{\vdash \Theta \Uparrow A_1, \Gamma \qquad \vdash \Theta \Uparrow A_2, \Gamma}{\vdash \Theta \Uparrow A_1 \wedge^- A_2, \Gamma}$ $\qquad$ $\dfrac{\vdash \Theta \Uparrow A_1, A_2, \Gamma}{\vdash \Theta \Uparrow A_1 \vee^- A_2, \Gamma}$
End of asynchronous phase	$\dfrac{\vdash \Theta, P \Uparrow \Gamma}{\vdash \Theta \Uparrow P, \Gamma}$ $\qquad$ $\dfrac{\vdash \Theta, p^\perp \Uparrow \Gamma}{\vdash \Theta \Uparrow p^\perp, \Gamma}$ $\qquad$ $\dfrac{\vdash \Theta, P \Downarrow P}{\vdash \Theta, P \Uparrow}$

Figure 13: LKF

Liang and Miller showed in [LM09] that

- various cut-rules are admissible;
- the polarities of atoms and connectives do not change the provability of an unfocussed sequent, but they change the shape of its proofs;
- with the admissibility of cut-rules, the system is complete for classical logic, no matter which polarities are placed on connectives and literals.

To prove cut-admissibility, they do not explicitly formalise a cut-elimination procedure, but it could probably be inferred from the proof.

Note that soundness of the system with respect to classical logic is trivially checked, rule by rule, forgetting about polarities and the structure of sequents.

Polarities in classical logic raise interesting questions: $A \wedge^+ B$ and $A \wedge^- B$ are equiprovable, and so are $A \vee^+ B$ and $A \vee^- B$. But, while the difference between the (direct) proofs of $A \vee^+ B$ and (direct) proofs of $A \vee^- B$ is clear,[5] one may wonder what the real difference is between (direct) proofs of $A \wedge^+ B$ and (direct) proofs of $A \wedge^- B$, given that the two rules look very much alike.

The difference lies not in their structure, but in the way they will behave in cut-elimination:

- from a proof of $A \wedge^- B$ (facing a proof of $A^\perp \vee^+ B^\perp$), only one sub-proof is used while the other is thrown away,
- from a proof of $A \wedge^+ B$ (facing a proof of $A^\perp \vee^- B^\perp$), both sub-proofs are used.

[5] Direct proofs of $A \vee^+ B$ choose one side and throw away the other, while direct proofs of $A \vee^- B$ keep the two sides.

Metaphorically, proving either conjunction is like picking 1 boy name and 1 girl name, when your couple is pregnant: proving a negative conjunction is picking the two names when you are expecting one baby (not knowing whether it is a boy or a girl), while proving the positive conjunction is picking the two names when expecting twins (a boy and a girl). On the paper, you have the same job to do, but you will probably approach the problem very differently.

3.2.1 Informal relation to System L

It may not be obvious, but system LKF roughly expresses, without proof-terms, some derivations of System L in a 1-sided format.[6]

Indeed, think of

- a focussed sequent $\vdash \Theta \Downarrow A$ as a typing judgement for a term value V: $\quad \vdash V : A \;;\; \Theta$.
- an unfocussed sequent $\vdash \Theta \Uparrow \Gamma$ as a typing judgement for a command c: $\quad c : (\vdash \;;\; \Theta, \Gamma)$.

with Γ being the part of the typing context for negative continuation variables α^- with non-atomic types (which can be asynchronously decomposed), and Θ the rest of it (typing negative continuation variables α^- with atomic types, and typing positive continuation variables α^+).

To derive a focussed sequent $\vdash \Theta \Downarrow A$:

- the two rules of the group 'Synchronous phase' correspond to the typing rules for $V :: V'$ and for $\mathsf{inj}_i(V')$;
- the first rule of the group 'End of synchronous phase' does not correspond to a rule of System L but simply the realisation that the value V is a negative term t^-;
- the second rule of that group is when V is a variable.

To derive an unfocussed sequent $\vdash \Theta \Uparrow \Gamma$:

- the two rules of the group 'Asynchronous phase' correspond to the typing of (t_1, t_2) and $\lambda\alpha.t$;
- the first two rules of the group 'End of asynchronous phase' are not reflected in System L (they just move formulae that cannot be asynchronously decomposed from Γ to Θ)
- the third rule of that phase corresponds to the typing of $\langle V \mid \alpha^+ \rangle$.

Roughly speaking, we should think of LKF as typing those proof-terms of (a 1-sided version of) System L that are η-long $\longrightarrow_{\mathsf{F}}$ -normal forms. These are described by the following grammar:

$$\begin{array}{lcl} V, V' & ::= & \alpha^+ \,|\, V :: V' \,|\, \mathsf{inj}_i(V) \,|\, t^- \,|\, \mu\alpha^-.c \\ t^-, \ldots & ::= & (t_1, t_2) \,|\, \lambda\alpha.t \\ t, \ldots & ::= & \mu\alpha^+.c \,|\, t^- \,|\, \mu\alpha^-.c \\ c, \ldots & ::= & \langle V \mid \alpha^+ \rangle \;|\; \langle t^- \mid \alpha^- \rangle \end{array}$$

where the type of every $\mu\alpha^-.c$ is atomic.

It is only "roughly speaking", because in Liang-Miller's LKF:

1. when a formula is asynchronously decomposed, no copy of the formula is kept in the sequent (which means in the above grammar that in every command $\langle t^- \mid \alpha^- \rangle$ we impose $\alpha^- \notin \mathsf{FV}(t^-)$),

[6] A 1-sided version of System L would merge terms and continuations into terms, so that $V :: V'$ is a term value, and merge term variables and continuation variables into continuation variables.

2. when the focus is placed on a formula, all the formulae that could be asynchronously decomposed have already been asynchronously decomposed (which means in the above grammar that in every command $\langle V \mid \alpha^+ \rangle$, V has no free variable of the form α^+).
3. finally, the order in which formulae are decomposed in the asynchronous phase, is less deterministic than that imposed by the above grammar.

This is because, in Liang and Miller's view of focussing, and more generally in the tradition of linear logic, "what happens in the asynchronous phase stays in the asynchronous phase", in the sense that the details of the asynchronous phase (e.g. the order in which formulae are decomposed) are meaningless and should not impact the semantics of the proof.

This is difficult to reflect at the level of System L's proof-terms.

Therefore, we will now develop a Curry-Howard interpretation for that particular view of focussing, along the lines of Zeilberger's work [Zei08a, Zei08b, Zei10]: in order to forget about the inner details of the asynchronous phase, we formalise the idea of compacting each phase (asynchronous and even synchronous) into one atomic inference. This is called big-step focussing.

3.2.2 Identifying phases as atomic steps

We start by showing an example of how positive connectives are decomposed.

Example 9 Trying to prove $\vdash \Theta \Downarrow N_1 \wedge^+ (a \vee^+ N_2)$ we can build:

either

$$\dfrac{\dfrac{\dfrac{\text{End of synch phase}}{\vdash \Theta \Uparrow N_1}}{\vdash \Theta \Downarrow N_1} \qquad \dfrac{\dfrac{\dfrac{\text{End of synch phase}}{a^\perp \in \Theta}}{\vdash \Theta \Downarrow a}}{\vdash \Theta \Downarrow a \vee^+ N_2}}{\vdash \Theta \Downarrow N_1 \wedge^+ (a \vee^+ N_2)}$$

or

$$\dfrac{\dfrac{\dfrac{\text{End of synch phase}}{\vdash \Theta \Uparrow N_1}}{\vdash \Theta \Downarrow N_1} \qquad \dfrac{\dfrac{\dfrac{\text{End of synch phase}}{\vdash \Theta \Uparrow N_2}}{\vdash \Theta \Downarrow N_2}}{\vdash \Theta \Downarrow a \vee^+ N_2}}{\vdash \Theta \Downarrow N_1 \wedge^+ (a \vee^+ N_2)}$$

The whole synchronous phase can be expressed in just one step:

$$\dfrac{\vdash \Theta \Uparrow N_1 \quad a^\perp \in \Theta}{\vdash \Theta \Downarrow N_1 \wedge^+ (a \vee^+ N_2)} \quad \text{or} \quad \dfrac{\vdash \Theta \Uparrow N_1 \quad \vdash \Theta \Uparrow N_2}{\vdash \Theta \Downarrow N_1 \wedge^+ (a \vee^+ N_2)}$$

In other words:

$$\dfrac{\forall N \in \Gamma, \quad \vdash \Theta \Uparrow N \qquad \forall a \in \Gamma, \quad a^\perp \in \Theta}{\vdash \Theta \Downarrow N_1 \wedge^+ (a \vee^+ N_2)}$$

with $\Gamma = N_1, a$ or $\Gamma = N_1, N_2$

In either case, we say that Γ "is a positive decomposition of" $N_1 \wedge^+ (a \vee^+ N_2)$, which we denote: $N_1, a \Vdash^+ N_1 \wedge^+ (a \vee^+ N_2)$ and $N_1, N_2 \Vdash^+ N_1 \wedge^+ (a \vee^+ N_2)$. ※

Now we generalise this example into a formal definition:

DEFINITION 44 (Decomposition of positive connectives)

The *positive decomposition relation* is the binary relation, defined by the rules of Fig. 14, where $\Gamma, \Gamma_1, \Gamma_2$ are sets of positive atoms or negative formulae.

The *one-step synchronous phase* is the rule:

$$\frac{\Gamma \Vdash^+ A \qquad \forall N \in \Gamma,\ \vdash \Theta \Uparrow N \qquad \forall a \in \Gamma, \quad a^\perp \in \Theta}{\vdash \Theta \Downarrow A}\ \mathsf{synch}$$

※

$$\frac{}{N \Vdash^+ N} \qquad \frac{}{a \Vdash^+ a}$$

$$\frac{\Gamma_1 \Vdash^+ A_1 \qquad \Gamma_2 \Vdash^+ A_2}{\Gamma_1, \Gamma_2 \Vdash^+ A_1 \wedge^+ A_2} \qquad \frac{\Gamma \Vdash^+ A_i}{\Gamma \Vdash^+ A_1 \vee^+ A_2}$$

Figure 14: Positive decomposition relation

Notice the syntax we use for the synch rule: the symbols $\forall$ and $\in$ are **meta-level** symbols: the number of premisses is the cardinal of Γ (plus one if you count $\Gamma \Vdash^+ A$).

We now show an example of how negative connectives are decomposed.

EXAMPLE 10

$$\dfrac{\dfrac{\dfrac{\dfrac{\dfrac{\text{End of asynch phase}}{\vdash \Theta, P_1, a^\perp \Uparrow} \qquad \dfrac{\dfrac{\text{End of asynch phase}}{\vdash \Theta, P_1, P_2 \Uparrow}}{\vdash \Theta \Uparrow P_1, P_2}}{\vdash \Theta \Uparrow P_1, a^\perp}}{\vdash \Theta \Uparrow P_1, (a^\perp \wedge^- P_2)}}{\vdash \Theta \Uparrow P_1 \vee^- (a^\perp \wedge^- P_2)}$$

The whole asynchronous phase can be expressed in just one step:

$$\frac{\vdash \Theta, P_1, a^\perp \Uparrow \qquad \vdash \Theta, P_1, P_2 \Uparrow}{\vdash \Theta \Uparrow P_1 \vee^- (a^\perp \wedge^- P_2)}\ \mathsf{asynch}$$

In other words

$$\frac{\forall \Delta, \quad \vdash \Theta, \Delta \Uparrow}{\vdash \Theta \Uparrow P_1 \vee^- (a^\perp \wedge^- P_2)}$$

where Δ ranges over $\{\ \{P_1, a^\perp\}\ ,\ \{P_1, P_2\}\ \}$

In either case, we say that Δ "is a negative decomposition of" $P_1 \vee^- (a^\perp \wedge^- P_2)$, which we denote $P_1, a^\perp \Vdash^- P_1 \vee^- (a^\perp \wedge^- P_2)$ and $P_1, P_2 \Vdash^- P_1 \vee^- (a^\perp \wedge^- P_2)$. ※

Now we generalise this example into a formal definition:

DEFINITION 45 (Decomposition of negative connectives)

The *negative decomposition relation* is the binary relation, defined by the rules of Fig. 15, where where $\Delta, \Delta_1, \Delta_2$ are sets of negative atoms or positive formulae.

The *one-step asynchronous phase* is the rule:

$$\frac{\forall \Delta, (\Delta \Vdash^- A) \Rightarrow (\vdash \Theta, \Delta \Uparrow)}{\vdash \Theta \Uparrow A}$$

※

$$\frac{}{P \Vdash^- P} \qquad \frac{}{a^\perp \Vdash^- a^\perp}$$

$$\frac{\Delta \Vdash^- A_i}{\Delta \Vdash^- A_1 \wedge^- A_2} \qquad \frac{\Delta_1 \Vdash^- A_1 \qquad \Delta_2 \Vdash^- A_2}{\Delta_1, \Delta_2 \Vdash^- A_1 \vee^- A_2}$$

Figure 15: Negative decomposition relation

Again, notice that the syntax we use for the asynch rule uses the **meta-level** symbols $\forall$ and $\Rightarrow$: the number of premisses is the number of Δ satisfying $\Delta \Vdash^- A$ for the given A.

We now put everything together in the style of Zeilberger [Zei08a, Zei08b, Zei10].

Definition 46 (Big-step LKF, v1)

The big-step LKF system is given in Fig. 16, where Θ, Δ are sets of negative atoms or positive formulae and Γ is a set of positive atoms or negative formulae. ※

$$\frac{\Gamma \Vdash^+ A \qquad \forall N \in \Gamma, \vdash \Theta \Uparrow N \qquad \forall a \in \Gamma, \quad a^\perp \in \Theta}{\vdash \Theta \Downarrow A}\ \text{synch}$$

$$\frac{\vdash \Theta, P \Downarrow P}{\vdash \Theta, P \Uparrow}\ \text{focus} \qquad \frac{\forall \Delta, (\Delta \Vdash^- A) \Rightarrow (\vdash \Theta, \Delta \Uparrow)}{\vdash \Theta \Uparrow A}\ \text{asynch}$$

Figure 16: Big-step LKF, v1

Remark 41

Sequents of the form $\vdash \Theta \Downarrow N$ and sequents of the form $\vdash \Theta \Uparrow P$ are never present in the premisses of the rules. Such sequents can only appear as the very conclusion of a whole proof-tree.

Hence, we can equivalently present the big-step LKF system as the system of Fig. 17, and declare $\vdash \Theta \Uparrow P$ as syntactic sugar for $\vdash \Theta, P \Uparrow$, and $\vdash \Theta \Downarrow N$ as syntactic sugar for $\vdash \Theta \Uparrow N$. ※

$$\frac{\Gamma \Vdash^+ P \qquad \forall N \in \Gamma, \vdash \Theta \Uparrow N \qquad \forall a \in \Gamma, \quad a^\perp \in \Theta}{\vdash \Theta \Downarrow P}$$

$$\frac{\vdash \Theta, P \Downarrow P}{\vdash \Theta, P \Uparrow} \qquad \frac{\forall \Delta, (\Delta \Vdash^- N) \Rightarrow (\vdash \Theta, \Delta \Uparrow)}{\vdash \Theta \Uparrow N}$$

Figure 17: Big-step LKF, v2

Now we should notice a complete symmetry, and therefore some redundancy, in the two decomposition relations that we have defined:

REMARK 42 $\Gamma \Vdash^+ P$ if and only if $\Gamma^\perp \Vdash^- P^\perp$. ※

Hence, we can define in Fig. 18 a simplified version of big-step LKF, where this redundancy is eliminated.

$$\frac{\Gamma \Vdash P \qquad \forall N \in \Gamma,\ \vdash \Theta \Uparrow N \qquad \forall a \in \Gamma, \quad a^\perp \in \Theta}{\vdash \Theta \Downarrow P}$$

$$\frac{\vdash \Theta, P \Downarrow P}{\vdash \Theta, P \Uparrow} \qquad \frac{\forall \Gamma, (\Gamma \Vdash N^\perp) \Rightarrow (\vdash \Theta, \Gamma^\perp \Uparrow)}{\vdash \Theta \Uparrow N}$$

where $\Gamma \Vdash P$ is $\Gamma \Vdash^+ P$.

Figure 18: Big-step LKF, v3

Now notice that the rules for negative connectives are never used in the system! Due to the duality in the syntax, given by the involutive negation, we should be able to remove negative connectives altogether. We just need to introduce a marker in the syntax of a formula, to denote every change of polarity.

Let us write $\neg$ for this marker.

DEFINITION 47 (Syntax with positive connectives only)

Formulae are now defined by the following syntax:

$$\begin{array}{rcl} P & ::= & a \,|\, A_1 \wedge^+ A_2 \,|\, A_1 \vee^+ A_2 \\ A & ::= & P \,|\, \neg P \end{array}$$

with the following involutive negation:

$$\begin{array}{rcl} P^\perp & := & \neg P \\ (\neg P)^\perp & := & P \end{array}$$

※

REMARK 43 The previous grammar can be encoded into that one:

$$\begin{array}{rcl} \overline{a} & := & a \\ \overline{A \wedge^+ B} & := & \overline{A} \wedge^+ \overline{B} \\ \overline{A \vee^+ B} & := & \overline{A} \vee^+ \overline{B} \end{array} \qquad \overline{N} := \neg(\overline{N^\perp})$$

※

In Fig. 19 we reformulate big-step LKF with this syntax for formulae.

$$\frac{\Gamma \Vdash P \qquad \forall \neg P' \in \Gamma,\ \vdash \Theta \Uparrow \neg P' \qquad \forall a \in \Gamma, \neg a \in \Theta}{\vdash \Theta \Downarrow P}$$

$$\frac{\vdash \Theta, P \Downarrow P}{\vdash \Theta, P \Uparrow} \qquad \frac{\forall \Gamma, (\Gamma \Vdash P) \Rightarrow (\vdash \Theta, \Gamma^\perp \Uparrow)}{\vdash \Theta \Uparrow \neg P}$$

where $\Gamma \Vdash P$ is $\Gamma \Vdash^+ P$.

Figure 19: Big-step LKF, v4

Finally, we notice that it is more natural to write Γ on the left-hand side of a sequent:

Definition 48 (Big-step LKF, v5)

The big-step LKF system v5 is given in Fig. 20, where Γ is a set of atoms a or formulae of the form $\neg P$ and Θ is a set of negated atoms $\neg a$ or formulae of the form P. ※

$$\frac{\Gamma \Vdash P \qquad \forall \neg P' \in \Gamma,\ \Gamma_0 \vdash \Uparrow \neg P' \qquad \forall a \in \Gamma,\ a \in \Gamma_0}{\Gamma_0 \vdash \Downarrow P}$$

$$\frac{\Gamma_0, \neg P \vdash \Downarrow P}{\Gamma_0, \neg P \vdash \Uparrow} \qquad \frac{\forall \Gamma, (\Gamma \Vdash P) \Rightarrow (\Gamma_0, \Gamma \vdash \Uparrow)}{\Gamma_0 \vdash \Uparrow \neg P}$$

where $\Gamma \Vdash P$ is $\Gamma \Vdash^+ P$.

Figure 20: Big-step LKF, v5

The lesson to be remembered from this formulation of big-step LKF, is that (the big-step version of) asynchronous rules happens to **coincide** with a rule inferred from (the big-step version of) synchronous rules. This will make cut-elimination work, and it formalises (at least in classical logic) the concept known in philosophical logic as *harmony* [Ten78, Rea00, Rea10] (expressed originally between the introduction rules and elimination rules of Natural Deduction, or between left-introduction rules and right-introduction rules of Sequent Calculus).

Now we can consider that both synchronous and asynchronous rules are defined primitively, and notice the somewhat "miraculous" coincidence, or we can adopt the view that only synchronous rules are defined primitively; asynchronous phases then work by duality from the way synchronous phases work.

In other words,

- positive connectives are "defined" by their introduction rules;
- negative connectives are "defined" by duality, from the introduction rules of their positive duals.

We may not even need to bother representing their rules.

In this view, and via the Curry-Howard correspondence, we should define how to inhabit a type $A \to B$ with (proof-)terms, from the way we inhabit A with terms and B with continuations (with $\to$ being a negative connective). Writing $\lambda x.M$ with a variable $x : A$ and a body $M : B$, would then only be a mere representation for (or a mere even implementation of) an inhabitant of $A \to B$ that pre-exists the syntactical notation.

This is of course expressed *semantically* in orthogonality models (say in Definitions 28, 31 and 36) by the fact that we first define an interpretation for positive formulae (mentioning the syntax of their basic inhabitants, such as the construct $t :: e$), and in a second step we define the interpretation of negative formulae simply as the orthogonal of the interpretation of their dual formula. The defininition for negative formulae does not even mention the syntax of their inhabitants (such as $\lambda x.M$), but if we have a syntax for them, they "happen to live" (somewhat miraculously) in the interpretation.

We now formalise a way to express this *syntactically*, as a proof-term calculus for big-step LKF.

3.2.3 Functional interpretation as pattern-matching

Earlier we wrote that "the proofs of negatives *must interact well with* the proofs of the positive dual". The intuition we formalise is that

- the proofs of a positive connective (i.e. of some $\Gamma_0 \vdash \Downarrow P$) are some data that can be *pattern-matched*;
- the proofs of a negative connective (i.e. of some $\Gamma_0 \vdash \Uparrow \neg P$) are *functions* that consume data by *pattern-matching*.

The fact that the proofs of a negative are determined by duality from the proofs of the positive dual, is reflected by the fact that the shape of a pattern-matching function is indeed completely determined by the data-type of its argument.

So the Curry-Howard interpretation of big-step LKF is an abstract system of pattern-matching.

The "proof-terms" for the decomposition of (positive) connectives are *patterns*. For instance for the connectives $\wedge^+, \vee^+$:

DEFINITION 49 (Patterns for $\wedge^+, \vee^+$) Patterns are defined by the following syntax:

$$p ::= x^+ \,|\, x^- \,|\, (p_1, p_2) \,|\, \mathsf{inj}_i(p)$$

Their typing rules are presented in Fig. 21. ※

$$\frac{}{x^- : \neg P \Vdash x^- : \neg P} \qquad \frac{}{x^+ : a \Vdash x^+ : a}$$

$$\frac{\Gamma_1 \Vdash p_1 : A_1 \qquad \Gamma_2 \Vdash p_2 : A_2}{\Gamma_1, \Gamma_2 \Vdash (p_1, p_2) : A_1 \wedge^+ A_2} \qquad \frac{\Gamma \Vdash p : A_i}{\Gamma \Vdash \mathsf{inj}_i(p) : A_1 \vee^+ A_2}$$

Figure 21: Decomposition with patterns

We now give the proof-terms for big-step LKF:

DEFINITION 50 (Pattern-matching calculus)

Let Pat be a set of elements called *patterns*, and denoted p, p', ...

The syntax of proof-terms is given by the following grammar:

$$\begin{array}{llll} \textbf{Positive terms} & t^+ & ::= & p.\sigma \\ \textbf{Negative terms} & t^- & ::= & f \\ \textbf{Commands} & c & ::= & \langle x^- \mid t^+ \rangle \,|\, \langle f \mid t^+ \rangle \end{array}$$

where

- σ is a substitution from negative variables such as x^- to negative terms, and from positive variables such as x^+ to positive terms;
- f is a function from patterns to commands.

Let $\mathbb{A}$ and $\mathbb{M}$ be two sets of elements called *atoms* and *molecules* and denoted p and P, respectively.

Let *typing contexts* be functions mapping negative variables to molecules (written $x^- : \neg P$) and positive variables to atoms (written $x^+ : a$).

Let $\Gamma \Vdash p : P$ be a typing relation where p is a pattern, P is a molecule, and Γ is a typing context.

The typing rules for proof-terms are presented in Fig. 22.
There is just one cut-elimination rule:

$$\text{(pat-match)} \qquad \langle f \mid p.\sigma \rangle \longrightarrow (f(p))\sigma$$

where $c\sigma$ denotes the application of substitution σ to the command c. ※

$$\frac{\Gamma \Vdash p : P \qquad \forall (x^- : \neg P) \in \Gamma, \quad \Gamma_0 \vdash \Uparrow \sigma(x^-) : \neg P \qquad \forall (x^+ : a) \in \Gamma, \quad (\sigma(x^+) : a) \in \Gamma_0}{\Gamma_0 \vdash \Downarrow p.\sigma : P}$$

$$\frac{\forall \Gamma, (\Gamma \Vdash p : P) \Rightarrow f(p) : (\Gamma_0, \Gamma \vdash \Uparrow)}{\Gamma_0 \vdash \Uparrow f : \neg P}$$

$$\frac{\Gamma_0, x^- : \neg P \vdash \Downarrow p.\sigma : P}{\langle x^- \mid p.\sigma \rangle : (\Gamma_0, x^- : \neg P \vdash \Uparrow)} \qquad \frac{\Gamma_0 \vdash \Uparrow f : \neg P \qquad \Gamma_0 \vdash \Downarrow t^+ : P}{\left\langle f \mid t^+ \right\rangle : (\Gamma_0 \vdash \Uparrow)}$$

where Γ_0, Γ, ... are typing contexts.

Figure 22: Typing for the pattern-matching calculus

The one cut-elimination rule is the very standard mechanism of pattern-matching, with the command $\langle f \mid t^+ \rangle$ representing what we could informally write as:
"`match` t^+ `with` $\underbrace{\ldots \mapsto \ldots}_{f}$"

Remark 44

1. Notice how negative terms are not really terms, but functions of the meta-level (or meta-level functions that are reified in the term syntax); this is a higher-order definition, and we do not give any concret syntax for such functions.

   Strictly speaking, our definition depends on the notion of function space that we take for the definition of negative terms (We could for instance restrict it to computable functions, but so far we do not specify such things).

   Also, with such a definition, it may not be clear exactly what the contextual closure of the rule (pat-match) is. By $\longrightarrow_{(\mathsf{pat-match})}$ we therefore denote the reduction relation where (pat-match) is applied at the top-level of a given command (no contextual closure).
2. Also notice how we emphasised that the definition of the proof-term calculus is *independent* from the syntax of patterns and the typing system for them.

   Definition 49 and Fig. 21 give one example (where atoms are positive atomic formulae and molecules are positive formulae).

   But the construction of the Curry-Howard interpretation for big-step focussing is modular in those notions.

   This is a gain of genericity / abstraction that we will further develop in the next Chapters. ※

Theorem 45 (Subject Reduction)

If $c : (\Gamma \vdash \Uparrow)$ and $c \longrightarrow_{(\mathsf{pat-match})} c'$ then $c' : (\Gamma \vdash \Uparrow)$. ※

Reviewing the various properties that are desirable for an instance of the Curry-Howard correspondence, we find that Progress (in this case, cut-elimination), depends on how we may reduce functions (so far, $\longrightarrow_{(\mathsf{pat-match})}$ only applies at the root); Confluence does not make sense here because, until we define how to reduce functions, there is at most one redex to reduce; and for Normalisation, we need to explore models (e.g. orthogonality models) of such a calculus.

Conclusion

The study of orthogonality models for the pattern-matching calculus should be particularly interesting:

In [MM09], Munch-Maccagnoni already explored the construction of orthogonality models for polarised System L with an emphasis on focussing properties. Big-step LKF and its underlying pattern-matching calculus seems to be an even more appropriate framework to look at the connection between focussing and orthogonality models, since this framework reflects at the syntactical level what orthogonality models describe at the semantical level, namely the fact that we first declare what the "inhabitants of positive formulae" are, and then we define the "inhabitants of negative formulae" by duality as those inhabitants that "interact well with" the inhabitants of the dual (positive) formula. In case of an orthogonality model, to "interact well with" means to "be orthogonal to"; in the case of pattern-matching, it means to "be able to consume". To be more precise:

- Inhabitants of positive types have *structure*: in an orthogonality model we need an algebraic structure to interpret positive constructs such as _::_ or $\mathsf{inj}\_(\_)$; in big-step LKF, these inhabitants come as the combination of a pattern (e.g. _::_ or $\mathsf{inj}\_(\_)$) and a substitution that fills its holes.
- Inhabitants of negative formulae may lack any structure, but they come with a *behaviour*: in an orthogonality model, they can range over any abstract set for which the orthogonality relation with positive inhabitants is defined; in big-step LKF, they range over any abstract set of functions (we do not specify which) that can consume patterns.

So, in order to formalise the connections that are informally described above, the second part of this dissertation explores orthogonality models for big-step focussing systems. We shall strip anything that is not essential off the constructions we make, systematically seeking the greatest generality, and aiming at the cores of orthogonality models and focussing systems. Doing so reveals the essential difference between realisability and typing:

- in realisability, checking whether a given negative inhabitant "interacts well with" an arbitrary inhabitant of a positive formula, requires the computation of an interaction that explores the positive inhabitant's structure to an **arbitrary depth** (as nothing restricts the criterion given by orthogonality);
- in typing, checking whether a given negative inhabitant "interacts well with" an arbitrary inhabitant of a positive formula, only requires the computation of an interaction that explores the positive inhabitant's structure to a **bounded depth** (as the negative inhabitant is a function that performs a case analysis on the positive inhabitant's top-level pattern and the interaction has to uniformly treat the rest of the inhabitant's structure).

In case each positive formula comes with a finite number of patterns for it, the above distinction is what makes typing decidable and realisability undecidable (in general).

Part II

Abstract focussing

Introduction

The second part of this dissertation presents unpublished material, on the theme of *abstract focussing*.

In the previous chapters we have seen the use of polarities and focussing in the proof theory of classical logic, where a focussed proof is a tree that alternates *synchronous phases* with *asynchronous phases*.

A level of abstraction is reached by *big-step focussing*, which compacts each phase into one inference step and thus allows the inner details of phases to be "forgotten". As revealed by Zeilberger's formulation of big-step focussing [Zei08a, Zei08b], the computational interpretation of this is pattern-matching.

In parallel to this, Munch-Maccagnoni [MM09] formalised the connection between focussing and the orthogonality techniques, which were presented in Chapter 2 for strong normalisation proofs and witness extraction.

The origin of the material presented in this second part of this dissertation is the idea that this deep connection could be revealed at a more abstract level if a Zeilberger-style system was used: For instance, the fact that, in such a system, the inhabitants of negative formulae live in an abstract function space and are not made of any syntax reflects the fact that, in orthogonality models, inhabitants of negative formulae can range over an abstract set and have no algebraic structure. The second part of this dissertation therefore started as a formalisation, in the proof-assistant Coq [Coq], of orthogonality models for a Zeilberger-style system, culminating with the Adequacy Lemma that connects the big-step focussing proof system with the orthogonality approach.

Doing this formalises the connection at a level of abstraction that forgets about the syntax or structure not only of the inhabitants of negative formulae (as suggested above) but also of positive formulae, abstracting over the logical connectives and the very syntax of formulae.

In the abstract framework that we present here, called LAF, an extra step of abstraction is also reached (compared to [Zei08a, Zei08b]) over the construction of (typing) contexts, which allows the same framework to capture both classical and intuitionistic systems. More substantially, the treatment of quantifiers is also new.

As the material developed and expanded, it also appeared that our framework, together with its machine-checked formalisation, could be directly implemented and serve as the theoretical foundations for a new version of the Psyche system, discussed in Part III of this dissertation. It could perhaps even serve as the basis for a formal proof of the system's correctness. Thinking along those lines oriented the design of the LAF framework with implementation issues in mind (e.g. using De Bruijn's indices or De Bruijn's levels), and resulted in the formalisation of mathematical structures behind which the OCaml modules can clearly

be seen.

This Coq formalisation and the implementability concern also resulted in a presentation of the material that is admittedly technical, with e.g. numerous parameters and long specifications, which was also fuelled by the desire to identify the connection between focussing and orthogonality at the "purest" level: every design choice or ingredient of the framework that was not essential to establishing the connection was systematically turned into a parameter of the framework, with an axiomatisation for it that we sought to be as weak as possible for the theory to hold.

Chapter 4 presents a description of the proof-term system for big-step focussing that is more formal than that with which we concluded Part I of this dissertation. This formalisation, called LAF, is essentially a reformulation of the ideas in [Zei08a, Zei08b], with no substantial difference but the modular description of typing contexts. This allows classical and intuitionistic systems to be instances of the same parameterised system LAF, as we describe at the end of the chapter.

Chapter 5, on the other hand, presents a substantial extension: the LAF system with quantifiers. It therefore subsumes Chapter 4, but giving the version of LAF with quantifiers straight away would be a bit harsh on the reader.

Chapter 6 explores realisability models for LAF, based on orthogonality, and its contents was the original motivation for the development of this material, as the Adequacy Lemma connects big-step focussing with orthogonality, typing with realisability, syntax with semantics. We apply this methodology to derive the consistency of LAF systems.

Chapter 7 then investigates the operational semantics of LAF, which interprets the proof-terms for big-step focussing as a pattern-matching calculus. We first present a small-step semantics by means of an abstract machine for head-reduction. Adapting the methodology of Chapter 2, we apply the orthogonality models of Chapter 6 to prove the normalisation of typed terms with respect to this abstract machine. Then we develop the abstract machine into a big-step operational semantics, for which a new application of orthogonality models provides the cut-elimination result for LAF.

Chapter 4

An abstract focussed sequent calculus - without quantifiers

Contents

In this chapter, we show how Zeilberger's ideas [Zei08a, Zei08b], as presented in Chapter 3, can be developed into an *abstract focussed sequent calculus* called LAF, and whose instances express the big-step versions of standard focussed sequent calculi.

The system of Chapter 3 is already abstract in the relation $\Vdash$ that decomposes a positive formula into a collection of positive atoms and negative formulae. Correspondingly, it is also abstract in the notion of *pattern* whose typing judgement is given by the relation $\Vdash$.

We push this abstraction further:

- Since this decomposition relation $\Vdash$ was the only ingredient of the system that used the syntax of formulae, we do not even have to assume that formulae are syntax, i.e. have an inductive structure, nor do we have to assume that "positive atoms" are particular kinds of formulae; positive atoms and formulae could literally be two arbitrary sets. We shall now respectively call them *atoms* and *molecules*.
- Moreover, a typing context Γ could be extended in an asynchronous step into Γ, Δ, where Δ is the result of decomposing some positive formula according to some pattern p and the

decomposition relation $\Vdash$. We have in fact no reason to assume that Γ and Δ are of the same nature and that Γ, Δ corresponds to set union (or whatever standard combination of typing contexts one usually considers). Therefore, "typing contexts" such as Γ will form an abstract notion, namely an algebra equipped with specific functions among which an arbitrary asymmetric construction $\Gamma; \Delta$ that replaces the above.[1]

Something that is difficult to treat formally at this abstract level is the use of a non-deterministic way of naming variables, and then having to deal with α-conversion, in particular when we formalise our framework LAF and its meta-theory in the proof-assistant Coq. Therefore we adopt a deterministic way of naming variables (now called *labels* since they are not subject to α-conversion), but we remain abstract in the exact system that we use for naming them: this approach will capture for instance De Bruijn's indices as well as De Bruijn's levels.

Section 4.1 presents LAF. Section 4.3 describes how to tune (i.e. instantiate) the abstract parameters so as to capture different logics (or logical systems). Section 4.4 provide instances illustrating different implementations of labels corresponding to De Bruijn's indices and De Bruijn's levels.

4.1 Presentation of the system

This section presents the quantifier-free version of system LAF, a highly modular / parameterised sequent calculus for big-step focussing.

An instance of LAF is given by a tuple of parameters

$$(\mathbb{A}, \mathbb{M}, \mathsf{Lab}_+, \mathsf{Lab}_-, \mathsf{Co}, \mathsf{Pat}, \Vdash)$$

where each parameter is described below.

4.1.1 Atoms, molecules, typing decompositions and typing contexts

The first group of parameters $(\mathbb{A}, \mathbb{M})$ specifies what the instance of LAF, as a logical system, talks about. A typical example is when $\mathbb{A}$ and $\mathbb{M}$ are respectively the sets of (positive) atoms and the set of formulae from a polarised logic. We will see in the next sections how our level of abstraction allows for some interesting variants. In the Curry-Howard view, $\mathbb{A}$ and $\mathbb{M}$ are our sets of types.

DEFINITION 51 (Atoms & molecules)

LAF is parameterised by two sets $\mathbb{A}$ and $\mathbb{M}$, whose elements are respectively called *atoms* (denoted a, a',...), and *molecules* (denoted M, M',...). ※

We then aim at defining *typing contexts*, those structures denoted Γ in a typing judgement of the form $\Gamma \vdash \ldots$.

Intuitively, we expect Γ to "contain" atoms and molecules, or more precisely to declare some variables as having atoms and molecules as their types.

For this it will be useful (e.g. to build models of LAF) to define contexts more generically, mapping variables to elements of two sets $\mathcal{A}$ and $\mathcal{B}$.

[1] Following Zeilberger's style, Δ itself will not be a typing context but will have a tree structure that may reflect the way a positive formula is decomposed into it.

Contexts will be extendable (in the case of typing contexts, we may want to extend Γ with a new type declaration for a fresh variable), and the following data-structure formalises what generic contexts will be extended with.

Definition 52 (Generic decomposition algebras)

Given two sets $\mathcal{A}$ and $\mathcal{B}$, the $(\mathcal{A},\mathcal{B})$-*decomposition algebra* $\mathbb{D}_{\mathcal{A},\mathcal{B}}$, whose elements are called $(\mathcal{A},\mathcal{B})$-*decompositions*, is the free algebra defined by the following grammar:

$$\Delta, \Delta_1, \ldots ::= \; a \,\big|\, {\sim}b \,\big|\, \bullet \,\big|\, \Delta_1, \Delta_2$$

where a (resp. b) ranges over $\mathcal{A}$ (resp. $\mathcal{B}$).

Let $\mathbb{D}_{\mathsf{st}}$ abbreviate $\mathbb{D}_{\mathtt{unit},\mathtt{unit}}$, whose elements we call *decomposition structures*.

The *(decomposition) structure* of an $(\mathcal{A},\mathcal{B})$-decomposition Δ, denoted $|\Delta|$, is its obvious homomorphic projection in $\mathbb{D}_{\mathsf{st}}$. ※

Intuitively, a $(\mathcal{A},\mathcal{B})$-decomposition Δ is simply the packaging of elements of $\mathcal{A}$ and elements of $\mathcal{B}$; we could flatten this packaging by seeing $\bullet$ as the empty set (or multiset), and Δ_1, Δ_2 as the union of the two sets (or multisets) Δ_1 and Δ_2.

Note that the coercion from $\mathcal{B}$ into $\mathbb{D}_{\mathcal{A},\mathcal{B}}$ is denoted with $\sim$. It helps distinguishing it from the coercion from $\mathcal{A}$ (e.g. when $\mathcal{A}$ and $\mathcal{B}$ intersect each other), and in many instances of LAF it will remind us of the presence of an otherwise implicit negation. But so far it has no logical meaning, and in particular $\mathcal{B}$ is not equipped with an operator $\sim$ of syntactical or semantical nature.

Definition 53 (Generic contexts)

LAF is parameterised by two sets Lab_+ and Lab_-, of elements called *positive labels* and *negative labels*, respectively.

Given two sets $\mathcal{A}$ and $\mathcal{B}$, an $(\mathcal{A},\mathcal{B})$-*context algebra* is an algebra of the form

$$\left(\mathcal{G}, \begin{pmatrix} \mathcal{G}\times\mathsf{Lab}_+ \rightharpoonup \mathcal{A} \\ (\Gamma, x^+) \mapsto \Gamma\left[x^+\right] \end{pmatrix}, \begin{pmatrix} \mathcal{G}\times\mathsf{Lab}_- \rightharpoonup \mathcal{B} \\ (\Gamma, x^-) \mapsto \Gamma\left[x^-\right] \end{pmatrix}, \begin{pmatrix} \mathcal{G}\times\mathbb{D}_{\mathcal{A},\mathcal{B}} \to \mathcal{G} \\ (\Gamma, \Delta) \mapsto \Gamma;\Delta \end{pmatrix}\right)$$

whose elements are called $(\mathcal{A},\mathcal{B})$-*contexts*.

As $(\Gamma, x^+) \mapsto \Gamma\left[x^+\right]$ and $(\Gamma, x^-) \mapsto \Gamma\left[x^-\right]$ are partial functions, we denote by $\mathsf{dom}^+(\Gamma)$ (resp. $\mathsf{dom}^-(\Gamma)$) the subset of Lab_+ (resp. Lab_-) where $\Gamma\left[x^+\right]$ (resp. $\Gamma\left[x^-\right]$) is defined. ※

We choose to call elements of Lab_+ and Lab_- "labels", rather than "variables", because "variable" suggests an object identified by a name that "does not matter" and somewhere subject to α-conversion. For instance in the following typing rule for the (simply-typed) λ-calculus

$$\frac{\Gamma, x\!:\!A \vdash t\!:\!B}{\Gamma \vdash \lambda x.t\!:\!A\to B}$$

the α-convertibility of the variable x bound in $\lambda x.t$ relates to a non-deterministic choice of name for the variable used to extend the context Γ into $\Gamma, x\!:\!A$.[2] It turns out that such non-determinism in context extension is quite tricky to adapt (though probably not impossible) to the level of abstraction of LAF, and in practice would not be used in an implementation of proof-search, where a deterministic choice of name would be performed ("first fresh name" picking, etc).

[2]The fact that the non-deterministic choice does not matter, a.k.a. *equivariance*, is covered at length in nominal logic [Pit03] and other works formalising binding.

Therefore, we decide to present LAF without the non-determinism related to α-conversion, yet without committing to using De Bruijn's indices or De Bruijn's levels. Hence the use of "labels", that will accommodate both systems (and others, as long as the concept of context extension $\Gamma;\Delta$ is a proper function, i.e. remains deterministic).

DEFINITION 54 (Typing decompositions and typing contexts)

The *typing decomposition algebra*, denoted $\mathbb{D}$, whose elements are called *typing decompositions*, is the $(\mathbb{A},\mathbb{M})$-decomposition algebra.

LAF is then parameterised by an $(\mathbb{A},\mathbb{M})$-context algebra Co, whose elements are called *typing contexts*. ※

4.1.2 Logical connectives

Finally, the last group of parameters (Pat, $\Vdash$) specifies the structure of molecules. If $\mathbb{M}$ is a set of formulae featuring logical connectives, those parameters specify the introduction rules for the connectives.

DEFINITION 55 (Patterns & decomposition relation)

LAF is parameterised by a *pattern algebra*, an algebra of the form

$$\left(\mathsf{Pat}, \begin{pmatrix} \mathsf{Pat}\to\mathbb{D}_{\mathsf{st}} \\ p \mapsto |p| \end{pmatrix}\right)$$

whose elements are called *patterns*, and by a *decomposition relation*, i.e. a set of elements

$$(\_ \Vdash \_ : \_) : (\mathbb{D} \times \mathsf{Pat} \times \mathbb{M})$$

such that if $\Delta \Vdash p : M$ then the structure of Δ is $|p|$. ※

The intuition behind the terminology is that the decomposition relation $\Vdash$ decomposes a molecule, according to a pattern, into a typing decomposition which, as a first approximation, can be seen as a "collection of atoms and (hopefully smaller) molecules".

4.1.3 Definition of the system

DEFINITION 56 (Proof-Terms)

Proof-terms are defined by the following syntax:

Positive terms	Terms^+	$t^+ ::= pd$
Decomposition terms	$\mathsf{Terms}^{\mathsf{d}}$	$d ::= x^+ \mid f \mid \bullet \mid d_1, d_2$
Commands	Terms	$c ::= \langle x^- \mid t^+ \rangle \mid \langle f \mid t^+ \rangle$

where p ranges over Pat, x^+ ranges over Lab_+, x^- ranges over Lab_-, and f ranges over the partial function space $\mathsf{Pat} \rightharpoonup \mathsf{Terms}$. ※

We can finally present the typing system LAF:

DEFINITION 57 (LAF) LAF is the inference system of Fig. 23 defining the derivability of three kinds of sequents

$$\begin{array}{lcl} (\_ \vdash [\_ : \_]) & : & (\mathsf{Co} \times \mathsf{Terms}^+ \times \mathbb{M}) \\ (\_ \vdash \_ : \_) & : & (\mathsf{Co} \times \mathsf{Terms}^{\mathsf{d}} \times \mathbb{D}) \\ (\_ \vdash \_) & : & (\mathsf{Co} \times \mathsf{Terms}) \end{array}$$

We further impose in rule async that the domain of function f be exactly those patterns that can decompose M (if $p \in \mathsf{Dom}(f)$ then there exists Δ such that $\Delta \Vdash p : M$).

$\mathsf{LAF}^{\mathsf{cf}}$ is the inference system LAF without the cut-rule. ※

$$\frac{\Delta \Vdash p : M \quad \Gamma \vdash d : \Delta}{\Gamma \vdash [pd : M]}\ \mathsf{sync}$$

$$\frac{}{\Gamma \vdash \bullet : \bullet} \qquad \frac{\Gamma \vdash d_1 : \Delta_1 \quad \Gamma \vdash d_2 : \Delta_2}{\Gamma \vdash d_1, d_2 : \Delta_1, \Delta_2}$$

$$\frac{\Gamma \left[x^+\right] = a}{\Gamma \vdash x^+ : a}\ \mathsf{init} \qquad \frac{\forall p, \forall \Delta, \quad \Delta \Vdash p : M \quad \Rightarrow \quad \Gamma ; \Delta \vdash f(p)}{\Gamma \vdash f : {\sim}M}\ \mathsf{async}$$

$$\frac{\Gamma \vdash [t^+ : \Gamma \left[x^-\right]]}{\Gamma \vdash \left\langle x^- \mid t^+ \right\rangle}\ \mathsf{select} \qquad \frac{\Gamma \vdash f : {\sim}M \qquad \Gamma \vdash [t^+ : M]}{\Gamma \vdash \left\langle f \mid t^+ \right\rangle}\ \mathsf{cut}$$

Figure 23: LAF

An intuition of LAF can be given in terms of proof-search:

When we want to "prove" a molecule, we first need to decompose it into a collection of atoms and (refutations of) molecules (rule sync). Each of those atoms must be found in the current typing context (rule init). Each of those molecules must be refuted, and the way to do this is to consider all the possible ways that this molecule could be decomposed, and for each of those decompositions, prove the inconsistency of the current typing context extended with the decomposition (rule async). This can be done by proving one of the molecules refuted in the typing context (rule select) or refuted by a complex proof (rule cut). Then a new cycle begins again.

Typing decompositions and decomposition terms organise the packaging of the proofs of atoms and (refuted) molecules decomposed by rule sync. Typing decompositions could here be taken to be a multiset of atoms and (refuted) molecules, but keeping a dedicated structure for the packaging will be more convenient when we add quantifiers: giving decompositions an inductive structure allows a lossless modelling of quantifiers' scopes.

4.2 Capturing existing systems

The above intuitions may become clearer when we instantiate the parameters of LAF with actual literals, formulae, etc in order to capture existing systems:

In the rest of this chapter we illustrate system LAF by specifying different instances, providing each time the long list of parameters, that capture different focussed sequent calculus systems.

By "capture", we mean of course a stronger result than just the equivalence between the notions of provability. In order to strengthen such a weak property between two systems, it is relevant to consider the notions of *adequacy* as defined in [Nig09, NM10]:

The shallowest level of adequacy, *relative completeness*, or adequacy of level -1, requires that a sequent is provable in one system if and only if the sequent to which it is mapped is provable in the other system. Level -2 of adequacy, *full completeness of proofs*, requires that there be a one-to-one correspondence between their (complete) proofs. Level -3 of adequacy, *full completeness of derivations* (a word used in [Nig09, NM10] for incomplete proofs), requires a one-to-one correspondence between the derivations in one system and those of the other system.

Strictly speaking, level -2 adequacy does not say more than level -1 as soon as the sequent has infinitely and denumerably many proofs. With level -3 adequacy, we aim at capturing much more. The simplest way to formalise its informal description above, for a function ϕ that maps the sequents of system $\mathcal{A}$ into the sequents of system $\mathcal{B}$, is probably as follows:

For every sequent $\mathcal{S}$ and multiset $\{\!\{\mathcal{S}_1, \ldots, \mathcal{S}_n\}\!\}$ of sequents in $\mathcal{A}$, there is a one-to-one correspondence $\phi_{\mathcal{S},\{\!\{\mathcal{S}_1,\ldots,\mathcal{S}_n\}\!\}}$ between

- *the partial proofs in $\mathcal{A}$ whose conclusion is $\mathcal{S}$ and whose multiset of open leaves is $\{\!\{\mathcal{S}_1, \ldots, \mathcal{S}_n\}\!\}$*
- *the partial proofs in $\mathcal{B}$ whose conclusion is $\phi(\mathcal{S})$ and whose multiset of open leaves is $\{\!\{\phi(\mathcal{S}_1), \ldots, \phi(\mathcal{S}_n)\}\!\}$*

The above is a symmetric property when ϕ is itself a one-to-one correspondence between sequents, but can also make sense if it is not. However, the above property needs to be adapted

- when in either of the two systems, we are interested not in each individual application of the inference rules but rather in groupings of rules: for instance in a focussed calculus, we may want to consider the grouping of a synchronous phase followed by a asynchronous phase (a.k.a. a *macro-rule* decomposing a *synthetic connective*) as a single step whose internal details should be ignored by the correspondence (this is what happens in [Nig09, NM10]);
- when either of the two systems features proof-terms, as the notion of incomplete proof is polluted by the presence, in the sequent, of a proof-term denoting a complete proof (unless we start considering incomplete proof-terms as well).

Both situations jeopardise the bijective aspect of each $\phi_{\mathcal{S},\{\!\{\mathcal{S}_1,\ldots,\mathcal{S}_n\}\!\}}$: in the former situation, we probably want to quotient proofs in some way so that the internal details of a rule grouping do not lead to multiple proofs that are not reflected in the other system ([NM10] mentions for instance "up to the permutation of asynchronous rules"); in the latter situation, proof-term annotations would provide for instance two proofs of $x:A, y:A \vdash ?:A$ while we only count one proof of $A, A \vdash A$ (whether A, A denotes a set or a multiset).

Another issue with the above notion of adequacy is that it fails to impose any notion of compositionality (when derivations are "plugged into" the open leaves of another derivation) about the family $(\phi_{\mathcal{S},\{\!\{\mathcal{S}_1,\ldots,\mathcal{S}_n\}\!\}})_{\mathcal{S},\{\!\{\mathcal{S}_1,\ldots,\mathcal{S}_n\}\!\}}$, something which we may have in mind when thinking about the "deepest level of adequacy".

System LAF features both focussing and proof-terms. Rather than trying to adapt to these concepts, and strengthen with compositionality, the above formalisation of level -3 adequacy, we opt for a version of level -3 adequacy that drops the use of bijections and the quantitative aspects that they provide (we no longer try to count proofs). On the other hand, we retain from level -3 adequacy, and formalise, the fact that the **structure** of proofs in one system matches the structure of proofs in the other system.

Definition 58 (Structural adequacy)

- Let $\mathcal{A}$ be an inference system providing a notion of proof-trees for elements called "sequents", and let $\mathcal{P}$ be a set of sequents.

  Given a proof-tree π in $\mathcal{A}$, the multiset of $\mathcal{P}$-*immediate sequents* of π is defined recursively on π: it contains the conclusions that are in $\mathcal{P}$ of the direct sub-trees of π, as well as the $\mathcal{P}$-immediate sequents of the direct sub-trees of π whose conclusions are not in $\mathcal{P}$.
- Let $\mathcal{A}$ and $\mathcal{B}$ be two inference systems as in the previous point, and $\mathcal{R}$ be a relation between the sequents of $\mathcal{A}$ and the sequents of $\mathcal{B}$, with domain $\mathcal{D}$ and co-domain $\mathcal{C}$.

  $\mathcal{R}$ satisfies *structural adequacy* if, whenever $\mathcal{S}\mathcal{R}\mathcal{S}', \mathcal{S}_1\mathcal{R}\mathcal{S}'_1 \dots, \mathcal{S}_n\mathcal{R}\mathcal{S}'_n$,

  there is in $\mathcal{A}$ a proof of $\mathcal{S}$ with $\mathcal{D}$-immediate sequents $\{\!\{S_1, \dots, S_n\}\!\}$
  if and only if
  there is in $\mathcal{B}$ a proof of $\mathcal{S}'$ with $\mathcal{C}$-immediate sequents $\{\!\{S'_1, \dots, S'_n\}\!\}$

※

Structural adequacy clearly entails level -1 adequacy (by induction on a proof in $\mathcal{A}$, recursively finding its $\mathcal{D}$-immediate sequents, we recompose a proof in $\mathcal{B}$), but implies neither level -2 nor level -3 since we are not counting proofs. Also notice that we do not require anything about incomplete proofs that cannot be completed.

Every instance below relates to a traditional system, as we define an encoding satisfying structural adequacy. While LAF is defined as a typing system (in other words with proof-terms decorating proofs in the view of the Curry-Howard correspondence), most traditional systems that we capture below are purely logical, with no proof-term decorations. The encoding therefore needs to erase proof-term annotation, and for this it is useful to project the notion of typing context as follows:

Definition 59 (Referable atoms and molecules)

Let $\mathsf{Im}^+(\Gamma)$ (resp. $\mathsf{Im}^-(\Gamma)$) be the image set of $x^+ \mapsto \Gamma\left[x^+\right]$ (resp. $x^+ \mapsto \Gamma\left[x^+\right]$), i.e. the set of atoms (resp. molecules) that can be refered to, in Γ, by the use of a positive (resp. negative) label. ※

4.3 Examples in propositional logic

The parameters of LAF will be specified so as to capture: the one-sided version of LKF [LM09, LM11], its two-sided version, and LJF [LM09].

4.3.1 Polarised classical logic - one-sided

In this sub-section we define the instance LAF_{K1} corresponding to the one-sided focused sequent calculus LKF for polarised classical logic [LM09, LM11].

Definition 60 (Literals, formulae, patterns, decomposition)

Let $\mathbb{L}$ be a set of elements called *literals*, equipped with an involutive function called *negation* mapping every literal l to a literal $l^\perp$.

Let $\mathbb{A}$ be a *polarisation set*, i.e. a subset of $\mathbb{L}$ such that $l \in \mathbb{A}$ if and only if $l^\perp \notin \mathbb{A}$. Elements of $\mathbb{A}$ will be ranged over by $a, a', \dots$.

$$\frac{}{\bullet \Vdash \bullet : \top^+} \qquad \frac{}{\sim N^\perp \Vdash \_^- : N} \qquad \frac{}{a \Vdash \_^+ : a}$$

$$\frac{\Delta_1 \Vdash p_1 : A_1 \quad \Delta_2 \Vdash p_2 : A_2}{\Delta_1, \Delta_2 \Vdash (p_1, p_2) : A_1 \wedge^+ A_2} \qquad \frac{\Delta \Vdash p : A_i}{\Delta \Vdash \mathsf{inj}_i(p) : A_1 \vee^+ A_2}$$

Figure 24: Decomposition relation for LAF_{K1}

Let $\mathbb{M}$ be the set defined by the first line of the following grammar for (polarised) formulae of classical logic:

$$\begin{array}{lllll} \text{Positive formulae} & P, \dots & ::= & a \,|\, \top^+ \,|\, \bot^+ \,|\, A \wedge^+ B \,|\, A \vee^+ B \\ \text{Negative formulae} & N, \dots & ::= & a^\perp \,|\, \top^- \,|\, \bot^- \,|\, A \wedge^- B \,|\, A \vee^- B \\ \text{Unspecified formulae} & A & ::= & P \,|\, N \end{array}$$

Negation is extended to formulae as follows:

$$\begin{array}{llllll} \top^{+\perp} & := & \bot^- & \top^{-\perp} & := & \bot^+ \\ \bot^{+\perp} & := & \top^- & \bot^{-\perp} & := & \top^+ \\ (A \wedge^+ B)^\perp & := & A^\perp \vee^- B^\perp & (A \wedge^- B)^\perp & := & A^\perp \vee^+ B^\perp \\ (A \vee^+ B)^\perp & := & A^\perp \wedge^- B^\perp & (A \vee^- B)^\perp & := & A^\perp \wedge^+ B^\perp \end{array}$$

and we extend it to sets or multisets of formulae pointwise.

The set Pat of *pattern* is defined by the following grammar:

$$p, p_1, p_2, \dots ::= \_^+ \,|\, \_^- \,|\, \bullet \,|\, (p_1, p_2) \,|\, \mathsf{inj}_i(p)$$

The decomposition relation $(\_ \Vdash \_ : \_) : (\mathbb{D} \times \mathsf{Pat} \times \mathbb{M})$ is the restriction to molecules of the relation of $\mathbb{D} \times \mathsf{Pat} \times \mathbb{F}$ defined inductively for all formulae by the inference system of Fig. 24.

The map $p \mapsto |p|$ can be inferred from the decomposition relation. ※

Keeping the sync rule of LAF_{K1} in mind, we can already see in Fig. 24 the traditional introduction rules of positive connectives in polarised classical logic. The rest of this sub-section formalises that intuition and explains how LAF_{K1} manages the introduction of negative connectives, etc.

But in order to finish the instantiation of LAF for propositional polarised classical logic (1-sided), we need to define typing contexts, i.e. give Lab_+, Lab_-, and Co. In particular, we have to decide how to refer to elements of the typing context. To avoid getting into aspects that may be considered as implementation details, we will stay rather generic and only assume the following property:

DEFINITION 61 (Typing contexts) We assume

$$\begin{array}{llll} \mathsf{Im}^+(\Gamma; a) & = \mathsf{Im}^+(\Gamma) \cup \{a\} & \mathsf{Im}^-(\Gamma; a) & = \mathsf{Im}^-(\Gamma) \\ \mathsf{Im}^+(\Gamma; \sim M) & = \mathsf{Im}^+(\Gamma) & \mathsf{Im}^-(\Gamma; \sim M) & = \mathsf{Im}^-(\Gamma) \cup \{M\} \\ \mathsf{Im}^\pm(\Gamma; \bullet) & = \mathsf{Im}^\pm(\Gamma) & \mathsf{Im}^\pm(\Gamma; (\Delta_1, \Delta_2)) & = \mathsf{Im}^\pm(\Gamma; \Delta_1; \Delta_2) \end{array}$$

where $\pm$ stands for either $+$ or $-$. ※

In section 4.4 we present several implementations satisfying the above.

We now relate (cut-free) $\mathsf{LAF}^{\mathsf{cf}}_{K1}$ and the LKF system of [LM09, LM11].

DEFINITION 62 (Flattening typing decompositions) Let $\overline{\Delta}$ be the flattening of a typing decomposition as a multiset of positive literals and negative formulae, i.e.

$$\begin{array}{llll} \overline{a} & := & \{\!\{a\}\!\} & \overline{\sim P} \quad := \quad \{\!\{P^\perp\}\!\} \\ \overline{\bullet} & := & \emptyset & \overline{\Delta_1, \Delta_2} \quad := \quad \overline{\Delta_1} \cup \overline{\Delta_2} \end{array}$$

※

REMARK 46

- Notice that, for all formulae A and typing decomposition Δ, there exists $p \in \mathsf{Pat}$ such that $\Delta \Vdash p : A$ if and only if $A \downarrow \overline{\Delta}$ as defined in [LM11].
- Our assumption about typing contexts implies that, for all Γ and Δ,
$$\mathsf{Im}^+(\Gamma;\Delta) \cup \mathsf{Im}^-(\Gamma;\Delta)^\perp = \mathsf{Im}^+(\Gamma) \cup \mathsf{Im}^-(\Gamma)^\perp \cup \overline{\Delta}$$

※

DEFINITION 63 (Mapping sequents)

We encode the sequents of LAF_{K1} (regardless of derivability) to those of LKF as follows:

$$\begin{array}{lll} \phi(\Gamma \vdash c) & := & \vdash \mathsf{Im}^+(\Gamma)^\perp, \mathsf{Im}^-(\Gamma) \Uparrow \\ \phi(\Gamma \vdash x^+ : a) & := & \vdash \mathsf{Im}^+(\Gamma)^\perp, \mathsf{Im}^-(\Gamma) \Downarrow a \\ \phi(\Gamma \vdash f : {\sim}P) & := & \vdash \mathsf{Im}^+(\Gamma)^\perp, \mathsf{Im}^-(\Gamma) \Downarrow P^\perp \\ \phi(\Gamma \vdash [t^+ : P]) & := & \vdash \mathsf{Im}^+(\Gamma)^\perp, \mathsf{Im}^-(\Gamma) \Downarrow P \end{array}$$

※

THEOREM 47 (Adequacy between $\mathsf{LAF}^{\mathsf{cf}}_{K1}$ and LKF)

ϕ satisfies structural adequacy between $\mathsf{LAF}^{\mathsf{cf}}_{K1}$ and LKF. ※

Proof: The Lemmata 2 and 3 of [LM11] (for the particular case of LKF) provide the correspondence with the big-step rules of $\mathsf{LAF}^{\mathsf{cf}}_{K1}$:

async Clearly, a derivation in $\mathsf{LAF}^{\mathsf{cf}}_{K1}$ concludes $\Gamma \vdash f : {\sim}P$ for some term f if and only if it is of the form

$$\frac{\Gamma;\Delta_1 \vdash c_1 \quad \dots \quad \Gamma;\Delta_n \vdash c_n}{\Gamma \vdash f : {\sim}P}$$

for some commands $\{c_1, \dots, c_n\}$, and where $\{\Delta_1, \dots, \Delta_n\} = \{\Delta \mid \exists p, \Delta \Vdash p : P\}$.

Correspondingly, Lemma 2 of [LM11][3] entails that a derivation in LKF concludes $\vdash \mathsf{Im}^+(\Gamma)^\perp, \mathsf{Im}^-(\Gamma) \Downarrow P^\perp$ if and only if it is of the form

$$\dfrac{\dfrac{\vdots \qquad\qquad \vdots}{\vdash \mathsf{Im}^+(\Gamma)^\perp, \mathsf{Im}^-(\Gamma) \Uparrow P^\perp}}{\vdash \mathsf{Im}^+(\Gamma)^\perp, \mathsf{Im}^-(\Gamma) \Downarrow P^\perp} \quad \begin{array}{c} \vdash \mathsf{Im}^+(\Gamma)^\perp, \mathsf{Im}^-(\Gamma), {\Phi_1}^\perp \Uparrow \\ \vdots \\ \vdash \mathsf{Im}^+(\Gamma)^\perp, \mathsf{Im}^-(\Gamma), {\Phi_n}^\perp \Uparrow \end{array}$$

where $\{\Phi_1, \dots, \Phi_n\} = \{\Phi \mid P \downarrow \Phi\}$.

Writing ϕ for the bijection from $1, \dots, n$ to itself such that $\overline{\Delta_i} = \Phi_{\phi(i)}$, we notice that every sequent $\Gamma;\Delta_i \vdash c_i$ is mapped to the sequent $\vdash \mathsf{Im}^+(\Gamma)^\perp, \mathsf{Im}^-(\Gamma), {\Phi_{\phi(i)}}^\perp \Uparrow$. Indeed, Remark 46.2 entails that

[3] slightly reworded using its Lemma 4 as well

$$\mathsf{Im}^+(\Gamma;\Delta_i)^\perp \cup \mathsf{Im}^-(\Gamma;\Delta_i) = \mathsf{Im}^+(\Gamma)^\perp \cup \mathsf{Im}^-(\Gamma) \cup {\Phi_{\phi(i)}}^\perp$$

sync

Clearly, a derivation in $\mathsf{LAF}^{\mathsf{cf}}_{K1}$ concludes $\Gamma \vdash [t^+ : P]$ for some term t^+ if and only if it is of the form

$$\dfrac{\Delta \Vdash p : P \qquad \dfrac{\dfrac{\Gamma \vdash t_1^- : u_1}{\vdots} \quad \dfrac{\Gamma \vdash t_n^- : u_n}{\vdots}}{\Gamma \vdash \sigma : \Delta}}{\Gamma \vdash [p\sigma : P]}$$

for some Δ, p, σ, $t_1^-, \ldots, t_n^-$, and where $\overline{\Delta} = \{u_1, \ldots, u_n\}$.

Correspondingly, Lemma 3 of [LM11] entails that a derivation in LKF concludes $\vdash \mathsf{Im}^+(\Gamma)^\perp, \mathsf{Im}^-(\Gamma) \Downarrow P$ if and only if it is of the form

$$\dfrac{\dfrac{\vdash \mathsf{Im}^+(\Gamma)^\perp, \mathsf{Im}^-(\Gamma) \Downarrow u_1}{\vdots} \quad \dfrac{\vdash \mathsf{Im}^+(\Gamma)^\perp, \mathsf{Im}^-(\Gamma) \Downarrow u_n}{\vdots}}{\vdash \mathsf{Im}^+(\Gamma)^\perp, \mathsf{Im}^-(\Gamma) \Downarrow P}$$

for some $P \downarrow \{u_1, \ldots, u_n\}$.

init Clearly, a derivation in $\mathsf{LAF}^{\mathsf{cf}}_{K1}$ concludes $\Gamma \vdash x^+ : a$ for some positive label x^+ if and only if it is of the form

$$\dfrac{}{\Gamma \vdash x^+ : a}$$

with $a \in \mathsf{Im}^+(\Gamma)$.

Correspondingly, a derivation in LKF concludes $\vdash \mathsf{Im}^+(\Gamma)^\perp, \mathsf{Im}^-(\Gamma) \Downarrow a$ if and only if it is of the form

$$\dfrac{}{\vdash \mathsf{Im}^+(\Gamma)^\perp, \mathsf{Im}^-(\Gamma) \Downarrow a}$$

with $a \in \mathsf{Im}^+(\Gamma)$.

select Clearly, a derivation in $\mathsf{LAF}^{\mathsf{cf}}_{K1}$ concludes $\Gamma \vdash c$ for some command c if and only if it is of the form

$$\dfrac{\Gamma \vdash [t^+ : P]}{\Gamma \vdash \left\langle x^- \mid t^+ \right\rangle}$$

with $P \in \mathsf{Im}^-(\Gamma)$.

Correspondingly, a derivation in LKF concludes $\vdash \mathsf{Im}^+(\Gamma)^\perp, \mathsf{Im}^-(\Gamma) \Uparrow$ if and only if it is of the form

$$\dfrac{\vdash \mathsf{Im}^+(\Gamma)^\perp, \mathsf{Im}^-(\Gamma) \Downarrow P}{\vdash \mathsf{Im}^+(\Gamma)^\perp, \mathsf{Im}^-(\Gamma) \Uparrow}$$

with $P \in \mathsf{Im}^-(\Gamma)$.

□

COROLLARY 48 (Equivalence of provability)

The provability of a sequent in $\mathsf{LAF}^{\mathsf{cf}}_{K1}$ is the same as that of its encoding in LKF. ※

The proof may raise the question of why, in the definition of LAF, we gave a structure to typing decompositions, instead of directly using a flattened version (e.g. multiset). The reason is to allow the parametrisation of the system so as to capture logics for which the structure of typing decomposition may be important; if only for first-order logic, the scope of eigenvariables is more easily managed with a structure; this is even more true in higher-order logic.

4.3.2 Polarised classical logic - two-sided

Having seen how an instance of LAF captures a one-sided sequent calculus, we could see LAF itself as a sequent calculus that is intrinsically one-sided, considering as a notational idiosyncrasy our writing the typing environments on the left of the turnstyle.

Here, we show that, by enriching the atoms and molecules with a "side information", we can also capture a two-sided version of LKF.

Definition 64 (Literals, formulae, patterns, decomposition)

Let $\mathbb{L}^+$ (resp. $\mathbb{L}^-$) be a set of elements called positive (resp. negative) literals, and ranged over by $l^+, l_1^+, l_2^+, \ldots$ (resp. $l^-, l_1^-, l_2^-, \ldots$).

Formulae are defined by the following grammar:

$$\begin{array}{lllll} \text{Positive formulae} & P, \ldots & ::= & l^+ \,|\, \top^+ \,|\, \bot^+ \,|\, A \wedge^+ B \,|\, A \vee^+ B \,|\, \neg^+ A \\ \text{Negative formulae} & N, \ldots & ::= & l^- \,|\, \top^- \,|\, \bot^- \,|\, A \wedge^- B \,|\, A \vee^- B \,|\, \neg^- A \\ \text{Unspecified formulae} & A & ::= & P \,|\, N \end{array}$$

We *position* a literal or a formula on the left-hand side or the right-hand side of a sequent by combining it with an element, called *side information*, of the set $\{\mathsf{l}, \mathsf{r}\}$: we define

$$\begin{array}{lll} \mathbb{A} & := & \{(l^+, \mathsf{r}) \,|\, l^+ \text{ positive literal}\} \cup \{(l^-, \mathsf{l}) \,|\, l^- \text{ negative literal}\} \\ \mathbb{M} & := & \{(P, \mathsf{r}) \,|\, P \text{ positive formula}\} \cup \{(N, \mathsf{l}) \,|\, N \text{ negative formula}\} \end{array}$$

The set Pat of *patterns* is defined by the following grammar:

$$\begin{array}{rcl} p, p_1, p_2, \ldots & ::= & \__{\mathsf{r}}^+ \,|\, \__{\mathsf{r}}^- \,|\, \bullet_{\mathsf{r}} \,|\, (p_1, p_2) \,|\, \mathsf{inj}_i(p) \,|\, \curvearrowleft(p) \\ & | & \__{\mathsf{l}}^+ \,|\, \__{\mathsf{l}}^- \,|\, \bullet_{\mathsf{l}} \,|\, [p_1, p_2] \,|\, \pi_i(p) \,|\, \curvearrowright(p) \end{array}$$

The decomposition relation $(\_ \Vdash \_ : \_) : (\mathbb{D} \times \mathsf{Pat} \times \mathbb{M})$ is the restriction to molecules of the relation of $\mathbb{D} \times \mathsf{Pat} \times (\mathbb{F} \times \{\mathsf{l}, \mathsf{r}\})$ defined inductively for all positioned formulae by the inference system of Fig. 25.

Again, since we want to capture classical logic, we assume the same property about $(\mathsf{Lab}_+, \mathsf{Lab}_-, \mathsf{Co})$ as we did in Definition 61. ※

Keeping the sync rule of LAF_{K2} in mind, we see in Fig. 25 the traditional right-introduction rules of positive connectives and left-introduction rules of negative connectives.

A deeper intuition can be given by encoding LAF_{K2} sequents as two-sided sequents, just like we encoded LAF_{K1} sequents as one-sided LKF sequents:

$$\frac{}{\sim(N,\mathsf{l}) \Vdash \_^-_{\mathsf{r}} : (N,\mathsf{r})} \qquad \frac{}{(l^+,\mathsf{r}) \Vdash \_^+_{\mathsf{r}} : (l^+,\mathsf{r})}$$

$$\frac{}{\bullet \Vdash \bullet_{\mathsf{r}} : (\top^+,\mathsf{r})} \qquad \frac{\Delta \Vdash p : (A,\mathsf{l})}{\Delta \Vdash \curvearrowleft(p) : (\neg^+ A,\mathsf{r})}$$

$$\frac{\Delta_1 \Vdash p_1 : (A_1,\mathsf{r}) \quad \Delta_2 \Vdash p_2 : (A_2,\mathsf{r})}{\Delta_1,\Delta_2 \Vdash (p_1,p_2) : (A_1 \wedge^+ A_2,\mathsf{r})} \qquad \frac{\Delta \Vdash p : (A_i,\mathsf{r})}{\Delta \Vdash \mathsf{inj}_i(p) : (A_1 \vee^+ A_2,\mathsf{r})}$$

$$\frac{}{\sim(P,\mathsf{r}) \Vdash \_^-_{\mathsf{l}} : (P,\mathsf{l})} \qquad \frac{}{(l^-,\mathsf{l}) \Vdash \_^+_{\mathsf{l}} : (l^-,\mathsf{l})}$$

$$\frac{}{\bullet \Vdash \bullet_{\mathsf{l}} : (\bot^-,\mathsf{l})} \qquad \frac{\Delta \Vdash p : (A,\mathsf{r})}{\Delta \Vdash \curvearrowright(p) : (\neg^- A,\mathsf{l})}$$

$$\frac{\Delta_1 \Vdash p_1 : (A_1,\mathsf{l}) \quad \Delta_2 \Vdash p_2 : (A_2,\mathsf{l})}{\Delta_1,\Delta_2 \Vdash [p_1,p_2] : (A_1 \vee^- A_2,\mathsf{l})} \qquad \frac{\Delta \Vdash p : (A_i,\mathsf{l})}{\Delta \Vdash \pi_i(p) : (A_1 \wedge^- A_2,\mathsf{l})}$$

Figure 25: Decomposition relation for LAF_{K2}

DEFINITION 65 (LAF_{K2} sequents as two-sided sequents)

1. First, when $\pm$ is either $+$ or $-$, we define
$$\mathsf{Im}^{\pm\mathsf{r}}(\Gamma) := \{A \mid (A,\mathsf{r}) \in \mathsf{Im}^{\pm}(\Gamma)\}$$
$$\mathsf{Im}^{\pm\mathsf{l}}(\Gamma) := \{A \mid (A,\mathsf{l}) \in \mathsf{Im}^{\pm}(\Gamma)\}$$
2. Then we define the encoding:
$$\phi(\Gamma \vdash c) \quad := \quad \mathsf{Im}^{+\mathsf{r}}(\Gamma), \mathsf{Im}^{-\mathsf{l}}(\Gamma) \vdash \mathsf{Im}^{+\mathsf{l}}(\Gamma), \mathsf{Im}^{-\mathsf{r}}(\Gamma)$$

※

Keeping the above interpretation of sequents in mind, we should now see how to develop the details of the correspondence (similar to that expressed in Theorem 47) between $\mathsf{LAF}^{\mathsf{cf}}_{K2}$ and the two-sided version of LKF (which may actually not be written down in the literature).

As we can see, the decomposition relation, and the whole inference system described by LAF_{K2}, is completely symmetric.

4.3.3 Polarised intuitionistic logic

DEFINITION 66 (Literals, formulae, patterns, decomposition)

Let $\mathbb{L}^+$ (resp. $\mathbb{L}^-$) be a set of elements called positive (resp. negative) literals, and ranged over by $l^+, l_1^+, l_2^+, \ldots$ (resp. $l^-, l_1^-, l_2^-, \ldots$).

Formulae are defined by the following grammar:

Positive formulae	$P, \ldots$	$::=$	$l^+ \mid \top^+ \mid \bot^+ \mid A \wedge^+ B \mid A \vee B$
Negative formulae	$N, \ldots$	$::=$	$l^- \mid \top^- \mid \bot^- \mid A \wedge^- B \mid A \Rightarrow B \mid \neg A$
Unspecified formulae	A	$::=$	$P \mid N$

We *position* a literal or a formula on the left-hand side or the right-hand side of a sequent by combining it with an element, called *side information*, of the set $\{\mathsf{l}, \mathsf{r}\}$: we define

$$\begin{array}{rcl} \mathbb{A} & := & \{(l^+, \mathsf{r}) \,|\, l^+ \text{ positive literal}\} \cup \{(l^-, \mathsf{l}) \,|\, l^- \text{ negative literal}\} \cup \{(\bot^-, \mathsf{l})\} \\ \mathbb{M} & := & \{(P, \mathsf{r}) \,|\, P \text{ positive formula}\} \cup \{(N, \mathsf{l}) \,|\, N \text{ negative formula}\} \end{array}$$

In the rest of this sub-section v stands for either a negative literal l^- or $\bot^-$.

The set Pat of *pattern* is defined by the following grammar:

$$\begin{array}{rcl} p, p_1, p_2, \ldots & ::= & \_^+_{\mathsf{r}} \,|\, \_^-_{\mathsf{r}} \,|\, \bullet_{\mathsf{r}} \,|\, (p_1, p_2) \,|\, \mathsf{inj}_i(p) \\ & | & \_^+_{\mathsf{l}} \,|\, \_^-_{\mathsf{l}} \,|\, \bullet_{\mathsf{l}} \,|\, p_1 :: p_2 \,|\, \pi_i(p) \,|\, \curvearrowright(p) \end{array}$$

The decomposition relation $(\_ \Vdash \_ : \_) : (\mathbb{D} \times \mathsf{Pat} \times \mathbb{M})$ is the restriction (to molecules) of the relation of $\mathbb{D} \times \mathsf{Pat} \times (\mathbb{F} \times \{\mathsf{l}, \mathsf{r}\})$ defined inductively for all positioned formulae by the inference system of Fig. 26. ※

$$\frac{}{\sim(N, \mathsf{l}) \Vdash \_^-_{\mathsf{r}} : (N, \mathsf{r})} \qquad \frac{}{(l^+, \mathsf{r}) \Vdash \_^+_{\mathsf{r}} : (l^+, \mathsf{r})}$$

$$\frac{}{\bullet \Vdash \bullet_{\mathsf{r}} : (\top^+, \mathsf{r})}$$

$$\frac{\Delta_1 \Vdash p_1 : (A_1, \mathsf{r}) \quad \Delta_2 \Vdash p_2 : (A_2, \mathsf{r})}{\Delta_1, \Delta_2 \Vdash (p_1, p_2) : (A_1 \wedge^+ A_2, \mathsf{r})} \qquad \frac{\Delta \Vdash p : (A_i, \mathsf{r})}{\Delta \Vdash \mathsf{inj}_i(p) : (A_1 \vee A_2, \mathsf{r})}$$

$$\frac{}{\sim(P, \mathsf{r}) \Vdash \_^-_{\mathsf{l}} : (P, \mathsf{l})} \qquad \frac{}{(l^-, \mathsf{l}) \Vdash \_^+_{\mathsf{l}} : (l^-, \mathsf{l})}$$

$$\frac{}{(\bot^-, \mathsf{l}) \Vdash \bullet_{\mathsf{l}} : (\bot^-, \mathsf{l})} \qquad \frac{\Delta \Vdash p : (A, \mathsf{r})}{\Delta, (\bot^-, \mathsf{l}) \Vdash \curvearrowright(p) : (\neg A, \mathsf{l})}$$

$$\frac{\Delta_1 \Vdash p_1 : (A_1, \mathsf{r}) \quad \Delta_2 \Vdash p_2 : (A_2, \mathsf{l})}{\Delta_1, \Delta_2 \Vdash p_1 :: p_2 : (A_1 \Rightarrow A_2, \mathsf{l})} \qquad \frac{\Delta \Vdash p : (A_i, \mathsf{l})}{\Delta \Vdash \pi_i(p) : (A_1 \wedge^- A_2, \mathsf{l})}$$

Figure 26: Decomposition relation for LAF_J

Again, we can already see in Fig. 26 the traditional right-introduction rules of positive connectives and left-introduction rules of negative connectives.

A few words about the connectives: compared to LAF_{K2}, we have dropped the positive negation and we have replaced the negative disjunction by the implication, also negative (the negative negation and the positive disjunction are consequently written $\neg$ and $\vee$, respectively). Since in (polarised) classical logic, $A \Rightarrow B$ can be seen as an abbreviation for $(\neg^- A) \vee^- B$, the decomposition rule for $(A \Rightarrow B, \mathsf{l})$ is simply the combination of the $K2$ rules for $\neg^-$ and $\vee^-$.

With implication as a primitive connective, we could actually remove the (negative) negation from the system, since it can in turn be seen as the combination of implication and absurdity ($\neg A$ can be seen as the abbreviation for $A \Rightarrow \bot^-$) and its decomposition rule reflects this. Notice that the decomposition rule for $\bot^-$ (and therefore that of $\neg$) are slightly modified compared to $K2$. To understand this, we should start by making the following remark:

REMARK 49

1. Whenever $\Delta \Vdash p:(A,\mathsf{r})$, Δ contains no items of the form $\sim(P,\mathsf{r})$ or (v,l).
2. Whenever $\Delta \Vdash p:(A,\mathsf{l})$, Δ contains **exactly one** item of the form $\sim(P,\mathsf{r})$ or (v,l).

(v stands for either a negative literal l^- or $\bot^-$). ※

The first point corresponds to the fact that, when we have a right-hand side focus in intuitionistic logic, the focus never switches to the left-hand side when looking at a proof-tree bottom-up. Notice that this would be false in presence of the positive negation, which would precisely switch the focus to the left-hand side as in $K2$.

Now the second point would not hold if we kept the negative disjunction from $K2$, since its decomposition rule would create a branching with two premisses of the form (v,l). Hence its replacement with implication, whose decomposition rule has only one premiss of that form, so that, in every derivation of the above inference system, at most one branch keeps decomposing formulae on the left. And that would be true with the $K2$ rules for $\bot^-$ and $\neg^-$. The reason to tweak them is to get point 2 with *exactly one* rather than *at most one*, and it is for this tweak that we added $(\bot^-,\mathsf{l})$ to $\mathbb{A}$ (compared to the $K2$ version).

To see why Remark 49.2 is so important for intuitionistic logic, we should now interpret LAF_{K2} sequents as intuitionistic sequents (from e.g. LJF [LM09]):

DEFINITION 67 (LAF_J sequents as two-sided LJF sequents)

1. First, when $\pm$ is either $+$ or $-$, we define
$$\begin{aligned}\mathsf{Im}^{\pm\mathsf{r}}(\Gamma) &:= \{A \mid (A,\mathsf{r}) \in \mathsf{Im}^{\pm}(\Gamma)\}\\ \mathsf{Im}^{+\mathsf{l}}(\Gamma) &:= \{l^- \mid (l^-,\mathsf{l}) \in \mathsf{Im}^{+}(\Gamma)\}\\ \mathsf{Im}^{-\mathsf{l}}(\Gamma) &:= \{N \mid (N,\mathsf{l}) \in \mathsf{Im}^{-}(\Gamma)\}\end{aligned}$$
2. Then we define the encoding:
$$\begin{aligned}\phi(\Gamma \vdash c) &:= [\mathsf{Im}^{+\mathsf{r}}(\Gamma),\mathsf{Im}^{-\mathsf{l}}(\Gamma)] \longrightarrow [\mathsf{Im}^{+\mathsf{l}}(\Gamma),\mathsf{Im}^{-\mathsf{r}}(\Gamma)]\\ \phi(\Gamma \vdash x^+:(l^-,\mathsf{l})) &:= [\mathsf{Im}^{+\mathsf{r}}(\Gamma),\mathsf{Im}^{-\mathsf{l}}(\Gamma)] \xrightarrow{l^-} [\mathsf{Im}^{+\mathsf{l}}(\Gamma),\mathsf{Im}^{-\mathsf{r}}(\Gamma)]\\ \phi(\Gamma \vdash f:\sim(P,\mathsf{r})) &:= [\mathsf{Im}^{+\mathsf{r}}(\Gamma),\mathsf{Im}^{-\mathsf{l}}(\Gamma)] \xrightarrow{P} [\mathsf{Im}^{+\mathsf{l}}(\Gamma),\mathsf{Im}^{-\mathsf{r}}(\Gamma)]\\ \phi(\Gamma \vdash [t^+:(N,\mathsf{l})]) &:= [\mathsf{Im}^{+\mathsf{r}}(\Gamma),\mathsf{Im}^{-\mathsf{l}}(\Gamma)] \xrightarrow{N} [\mathsf{Im}^{+\mathsf{l}}(\Gamma),\mathsf{Im}^{-\mathsf{r}}(\Gamma)]\\ \phi(\Gamma \vdash x^+:(l^+,\mathsf{r})) &:= [\mathsf{Im}^{+\mathsf{r}}(\Gamma),\mathsf{Im}^{-\mathsf{l}}(\Gamma)] -_{l^+}\!\!\rightarrow\\ \phi(\Gamma \vdash f:\sim(N,\mathsf{l})) &:= [\mathsf{Im}^{+\mathsf{r}}(\Gamma),\mathsf{Im}^{-\mathsf{l}}(\Gamma)] -_{N}\!\!\rightarrow\\ \phi(\Gamma \vdash [t^+:(P,\mathsf{r})]) &:= [\mathsf{Im}^{+\mathsf{r}}(\Gamma),\mathsf{Im}^{-\mathsf{l}}(\Gamma)] -_{P}\!\!\rightarrow\end{aligned}$$

In the first four cases, we require $\mathsf{Im}^{+\mathsf{l}}(\Gamma),\mathsf{Im}^{-\mathsf{r}}(\Gamma)$ to be a singleton (or be empty). ※

The first line of the encoding is the same as for LAF_{K2} (Definition 65), but for the fact that we require $\mathsf{Im}^{+\mathsf{l}}(\Gamma),\mathsf{Im}^{-\mathsf{r}}(\Gamma)$ to be a singleton (or be empty), since we are to capture an intuitionistic system such as LJF. We also see in the last three cases (when there is a right-hand side focus), that the encoding forgets $\mathsf{Im}^{+\mathsf{l}}(\Gamma),\mathsf{Im}^{-\mathsf{r}}(\Gamma)$ altogether. If it is not empty, then it should definitely play no further role in the proof of the LAF_J sequent.

The issue arises in particular when analysing the select rule:

In LJF, placing the focus on a formula on the left-hand side does not affect the formula stored on the right-hand side; on the contrary, placing the focus on the right-hand side formula removes it from the right-hand side (no backup copy is made).

This is an important feature of intuitionistic logic: if a backup copy of the formula was kept, we could place again the focus on it further up in the proof, and we could thus prove formulae such as $A \vee \neg A$ or the drinker's theorem; indeed this amounts to authorising contraction on the right.

Now looking at the select rule of LAF_J, notice that the typing context Γ is **unchanged** by the rule: placing the focus on a right-hand side formula (i.e. a formula from $\mathsf{Im}^{+\mathsf{l}}(\Gamma), \mathsf{Im}^{-\mathsf{r}}(\Gamma)$) does not remove it from the typing context.

We could therefore fear that, because of this feature, LAF forces the presence of contraction (left and right) and is therefore intrinsically classical. Fortunately, this is not the case:

After selecting a right-hand side formula for focus, it is decomposed according to the rules of the decomposition relation. As mentioned in Remark 49.1, the focus never switches to the left-hand side and we are therefore left to prove a collection of sequents of the form $\Gamma \vdash x^+ : (v, \mathsf{r})$ or $\Gamma \vdash f : {\sim}(N, \mathsf{l})$ for some x^+ or f to be found. In the former case, the part of Γ that stores the unfortunate backup copy of the right-hand side formula that was selected for focus, does not affect whether $(v, \mathsf{r}) \in \mathsf{Im}^+(\Gamma)$. In the latter case, only rule async can be applied and a sequent of the form $\Gamma; \Delta \vdash c$ is left to be proved (for some c to be found) for every Δ that can decompose (N, l). For the adequacy with intuitionistic logic to work, it suffices that for every such Δ, the operation $\Gamma; \Delta$ erases from Γ the unfortunate backup copy of the right-hand side formula that was selected for focus. According to Remark 49.2, every such Δ contains **exactly one** item of the form (v, l) or ${\sim}(P, \mathsf{r})$, i.e. a new right-hand side formula which can overwrite the old one. At least, provided that $(\mathsf{Lab}_+, \mathsf{Lab}_-, \mathsf{Co})$ are defined to do that job.

Having tweaked the decomposition rule for $(\bot^-, \mathsf{l})$ to guarantee Remark 49.2, $(\mathsf{Lab}_+, \mathsf{Lab}_-, \mathsf{Co})$ should also make sure that, for any Γ, the (focussed) sequent $\Gamma \vdash [t^+ : (\bot^-, \mathsf{l})]$ can still be proved for some t^+ to be found, i.e. the sequent $\Gamma \vdash x^+ : (\bot^-, \mathsf{l})$ can be proved for some x^+ to be found, i.e. $(\bot^-, \mathsf{l}) \in \mathsf{Im}^+(\Gamma)$ (even when Γ is interpreted as something completely empty). This is easy to do, by having a permanent and special label $x^+_{(\bot^-, \mathsf{l})} \in \mathsf{Lab}_+$ mapped to $(\bot^-, \mathsf{l})$ in every Γ. This is the same as permanently adding $\bot^-$ on the right-hand side of intuitionistic sequents (as some kind of multi-conclusion), lest that right-hand side ever gets empty: it is harmless to both the intuitionistic provability and the structural theory of proofs (none are added, non are removed).

Definition 68 (Typing contexts)

We assume that we always have $(\bot^-, \mathsf{l}) \in \mathsf{Im}^+(\Gamma)$ and that

$$\begin{array}{llll}
\mathsf{Im}^+(\Gamma; (l^+, \mathsf{r})) & = \mathsf{Im}^+(\Gamma) \cup \{(l^+, \mathsf{r})\} & \mathsf{Im}^-(\Gamma; a) & = \mathsf{Im}^-(\Gamma) \\
\mathsf{Im}^+(\Gamma; {\sim}M) & = \mathsf{Im}^+(\Gamma) & \mathsf{Im}^-(\Gamma; {\sim}(N, \mathsf{l})) & = \mathsf{Im}^-(\Gamma) \cup \{(N, \mathsf{l})\} \\
\mathsf{Im}^\pm(\Gamma; \bullet) & = \mathsf{Im}^\pm(\Gamma) & \mathsf{Im}^\pm(\Gamma; (\Delta_1, \Delta_2)) & = \mathsf{Im}^\pm(\Gamma; \Delta_1; \Delta_2)
\end{array}$$

$$\begin{array}{ll}
\mathsf{Im}^+(\Gamma; (v, \mathsf{l})) & = \{(l^+, \mathsf{r}) \mid (l^+, \mathsf{r}) \in \mathsf{Im}^+(\Gamma)\} \cup \{(v, \mathsf{l}), (\bot^-, \mathsf{l})\} \\
\mathsf{Im}^-(\Gamma; {\sim}(P, \mathsf{r})) & = \{(N, \mathsf{l}) \mid (N, \mathsf{l}) \in \mathsf{Im}^-(\Gamma)\} \cup \{(P, \mathsf{r})\}
\end{array}$$

where again $\pm$ stands for either $+$ or $-$ and v stands for either a negative literal l^- or $\bot^-$. ※

The first three lines are the same as those assumed for $K1$ and $K2$, except it is restricted to those cases where we do not try to add to Γ an atom or a molecule that is interpreted as going to the right-hand side of a sequent. When we want to do that, this atom or molecule should overwrite the previous atom(s) or molecule(s) that was (were) interpreted as being on the right-hand side; this is done in the last two lines, where $\mathsf{Im}^{+\mathsf{l}}(\Gamma), \mathsf{Im}^{-\mathsf{r}}(\Gamma)$ is completely

erased.

THEOREM 50 (Adequacy between $\mathsf{LAF}^{\mathsf{cf}}_J$ and LJF)

ϕ satisfies structural adequacy between $\mathsf{LAF}^{\mathsf{cf}}_J$ and LJF. ※

Proof: The details are similar to those of Theorem 47, relying again on the LJF properties expressed in [LM09, LM11] and following the series of remarks and design decisions that were made above. □

4.4 Examples of labels implementation: De Bruijn's indices and levels

In this section we give some concrete implementations of labels to completely specify the typing context algebras used in the examples of the previous section.

4.4.1 Labels for classical logic

In the instances LAF_{K1} and LAF_{K2}, we have simply made some assumptions on the typing context algebra (in Definition 61). We now give it a full definition satisfying these assumptions and using of De Bruijn's indices.

In fact, we generically build an $(\mathcal{A},\mathcal{B})$-context for each pair of sets $\mathcal{A}$ and $\mathcal{B}$, and the typing context algebra will simply be the instance where $\mathcal{A}=\mathbb{A}$ and $\mathcal{B}=\mathbb{M}$.

DEFINITION 69 (Generic context algebras with De Bruijn's indices - classical)

Given two sets $\mathcal{A}$ and $\mathcal{B}$, we define an $(\mathcal{A},\mathcal{B})$-context algebra $\mathsf{Co}_{\mathcal{A},\mathcal{B}}$ as follows:

An $(\mathcal{A},\mathcal{B})$-context Γ is a pair (Γ^+,Γ^-) where Γ^+ is a list of elements of $\mathcal{A}$ and Γ^- is a list of elements of $\mathcal{B}$.

Extensions are defined as follows:

$$\begin{array}{llll}(\Gamma^+,\Gamma^-);a & := & (a{::}\Gamma^+,\Gamma^-) & (\Gamma^+,\Gamma^-);{\sim}b := (\Gamma^+,b{::}\Gamma^-)\\ (\Gamma^+,\Gamma^-);\bullet & := & (\Gamma^+,\Gamma^-) & (\Gamma^+,\Gamma^-);(\Delta_1,\Delta_2) := (\Gamma^+,\Gamma^-);\Delta_1;\Delta_2\end{array}$$

Positive labels and negative labels are two disjoint copies of the set of integers, with elements denoted n^+ and n^-, and we define
$(\Gamma^+,\Gamma^-)\,[n^+]$ as the $(n+1)^{th}$ element of Γ^+
$(\Gamma^+,\Gamma^-)\,[n^-]$ as the $(n+1)^{th}$ element of Γ^-. ※

These are indeed De Bruijn's indices, since the element accessed by label 0^+ (resp. 0^-) is the head of the list Γ^+ (resp. Γ^-), i.e. the element of the list that has last been added.

Alternatives using De Bruijn's indices are possible:

- the choice we made, when extending a context with (Δ_1,Δ_2), of first extending the context with Δ_1 and then extending the result with Δ_2, was completely arbitrary, we could have defined
$$(\Gamma^+,\Gamma^-);(\Delta_1,\Delta_2) := (\Gamma^+,\Gamma^-);\Delta_2;\Delta_1$$
- we could have defined an $(\mathcal{A},\mathcal{B})$-context Γ as one single list of atoms and molecules, with $\mathsf{Lab}_+=\mathsf{Lab}_-=\mathbb{N}$ and $\Gamma\,[n^+]$ (resp. $\Gamma\,[n^-]$) being defined only on those integers mapped to atoms (resp. molecules);

- we could have defined an $(\mathcal{A}, \mathcal{B})$-context Γ as a list of $(\mathcal{A}, \mathcal{B})$-decompositions, with
$$\Gamma; \Delta := \Delta :: \Gamma$$
and then a positive or negative label would be a pair (n, i), where the integer n identifies the n^{th} element Δ of the list and i is a string of 0 and 1 representing the path from the root of Δ (seen as a tree) to one of its leaves.

  But we can also use De Bruijn's levels, rather than indices.

One of the drawbacks of the implementation with De Bruijn's indices, is that the name of a label, declared with a type in a typing context Γ, "changes" when Γ is extended with some typing decomposition Δ. For instance if $\Gamma\,[0^+]$ is an atom a because a is the head of the list Γ^+, then in $\Gamma; \Delta$, a may no longer be the head of $(\Gamma; \Delta)^+$ and it will be refered to with an updated label name.

Depending on how the computations of $\Gamma\,[x^+]$ and $\Gamma\,[x^-]$ are implemented (imagine we have a HashMap for this), it could be problematic to have to update all the label names at every extension. We could do this update lazily, or we could also go for De Bruijn's levels: once it has been introduced in a typing context, a label will remain unchanged by the subsequent extensions of the context.

Definition 70 (Context algebras with De Bruijn's levels - classical)

Positive labels and negative labels are two disjoint copies of the set of integers, with elements denoted n^+ and n^-, and we define
$(\Gamma^+, \Gamma^-)\,[n^+]$ as the $(|\Gamma^+| - n)^{th}$ element of Γ^+
$(\Gamma^+, \Gamma^-)\,[n^-]$ as the $(|\Gamma^-| - n)^{th}$ element of Γ^-. ※

In other words, the difference between De Bruijn's indices and De Bruijn's levels is that we are counting from the bottom of the list rather than from the head.

All of the above alternatives work for LAF_{K1} and LAF_{K2}, in that the assumptions of Definition 61 are clearly satisfied.

Choosing between them is really a question of implementation, with no theoretical impact.

4.4.2 Labels for intuitionistic logic

In the instance LAF_J, we have made some different assumptions on the typing context algebra (in Definition 68). We adapt our definition of the typing context algebra accordingly.

This time, we directly define it rather than go through the generic definition of an $(\mathcal{A}, \mathcal{B})$-context algebra for each $\mathcal{A}$ and $\mathcal{B}$, since the assumptions in Definition 68 (unlike those in Definition 61) make a case analysis on the kind of atom (resp. on the kind of molecule) that is added to the typing context. That case analysis would not make sense for arbitrary sets $\mathcal{A}$ and $\mathcal{B}$.

Definition 71 (Context algebras with De Bruijn's indices - intuitionistic)

The typing context algebra Co is defined as follows:

A typing context Γ is a triple (Γ^+, Γ^-, R) where Γ^+ is a list of atoms, Γ^- is a list of molecules, and R is either an atom of the form (v, l) or a molecule of the form (P, r).[4]

[4] Intuitively, R represents the right-hand side of the LJF sequent.

Extensions are defined as follows:

$$
\begin{array}{llll}
(\Gamma^+, \Gamma^-, R); (l^+, \mathsf{r}) & := & ((l^+, \mathsf{r}) :: \Gamma^+, \Gamma^-, R) & (\Gamma^+, \Gamma^-, R); \sim(N, \mathsf{l}) := (\Gamma^+, (N, \mathsf{l}) :: \Gamma^-, R) \\
(\Gamma^+, \Gamma^-, R); (v, \mathsf{l}) & := & (\Gamma^+, \Gamma^-, (v, \mathsf{l})) & (\Gamma^+, \Gamma^-, R); \sim(P, \mathsf{r}) := (\Gamma^+, \Gamma^-, (P, \mathsf{r})) \\
(\Gamma^+, \Gamma^-, R); \bullet & := & (\Gamma^+, \Gamma^-) & (\Gamma^+, \Gamma^-, R); (\Delta_1, \Delta_2) := (\Gamma^+, \Gamma^-, R); \Delta_1; \Delta_2
\end{array}
$$

Again, we use for labels two disjoint copies $\mathbb{N}^+$ and $\mathbb{N}^-$ of the set of integers:

A positive label is either some $n^+ \in \mathbb{N}^+$ or one of two special labels $\star^+$ and $x^+_{(\perp^-, \mathsf{l})}$.

A negative label is either some $n^- \in \mathbb{N}^-$ or the special label $\star^-$.

And we define

$(\Gamma^+, \Gamma^-, R)\,[n^+]$ as the $(n+1)^{th}$ element of Γ^+

$(\Gamma^+, \Gamma^-, R)\,[\star^+]$ as R if it is of the form (v, l) (undefined if not)

$(\Gamma^+, \Gamma^-, R)\left[x^+_{(\perp^-, \mathsf{l})}\right]$ as $(\perp^-, \mathsf{l})$

$(\Gamma^+, \Gamma^-, R)\,[n^-]$ as the $(n+1)^{th}$ element of Γ^-

$(\Gamma^+, \Gamma^-, R)\,[\star^-]$ as R if it is of the form (P, r) (undefined if not). ※

Clearly, the above definition of the typing context algebra satisfies the assumptions in Definition 68.

And again, there are many alternatives for the above definition, including the use of De Bruijn's levels, etc. Choosing between them would again simply be a question of implementation.

In brief, this section (as well as other parts of this dissertation) shows that the theory is able to handle diverse implementations, instead of having the theory commit to a particular choice of formalisation, and then having an implementation depart from it. Here, we can directly see the OCaml modules and module signatures that we can or should implement.

Chapter 5

An abstract focussed sequent calculus - with quantifiers

Contents

In this chapter we extend the LAF sequent calculus to handle quantifiers.

First, we should notice that the calculus we presented in Chapter 4 can already "handle quantifiers", in the way the ω-rule does [Hil31, Sch50]. Indeed, we can adapt and extend system LAF_{K1} with an extra rule for the decomposition relation such as

$$\frac{\Delta \Vdash p : \{^{r}/_{x}\}\, A}{\Delta \Vdash (r,p) : \exists x A}$$

capturing the positive behaviour of the existential quantifier in the synchronous rule.

But this will also determine the asynchronous treatment of the universal quantifier: Ignoring proof-terms for the moment, proving the refutation $\Gamma \vdash {\sim}\exists x N$ (i.e. intuitively proving $\forall x N^{\perp}$) requires the use of rule async, with sub-proofs for each of the sequents

$$\Gamma, {\sim}\{^{t}/_{x}\}\, N^{\perp} \vdash$$

where t ranges over all potential witnesses for x, which is the behaviour of the ω-rule.

In particular if N is of the form $\forall y\ P$, each of those premisses can then be derived by a proof of the form

$$\frac{\dfrac{\Gamma \vdash {\sim}\left\{^{t,t'}/_{x,y}\right\} P}{\Gamma \vdash [\exists y \left\{^{t}/_{x}\right\} P^{\perp}]}}{\Gamma, {\sim}\exists y \left\{^{t}/_{x}\right\} P^{\perp} \vdash}$$

where t' is witness for y whose choice may depend (possibly in a non-uniform way) on the instance t of x.

So, in order to recover a standard rule for $\forall$-introduction, which uses something like an eigenvariable, we need to enrich LAF, which will now be given by a tuple of parameters

$$(\mathbb{S}, \mathbb{T}, \mathbb{C}, \Vdash, \mathbb{A}, \mathbb{M}, \equiv, \mathsf{Lab}_+, \mathsf{Lab}_-, \mathsf{Co}, \mathsf{Pat}, \vdash)$$

where each parameter is described in Section 5.1.

Section 5.2 then provides an instance illustrating first-order quantification.

5.1 Presentation of the system

5.1.1 Quantifying structure

The first group of parameters $(\mathbb{S}, \mathbb{T}, \mathbb{C}, \Vdash)$ specifies the objects that LAF quantifies over. For logics with quantifiers, the following definition provides a rather general setting: the terms that can be provided as witnesses are multi-sorted, and the sorting may depend on a local sorting context (as we would need for higher-order logic, dependent types, etc).

DEFINITION 72 (Quantifying structure)

LAF is parameterised by a *quantifying structure* $(\mathbb{S}, \mathbb{T}, \mathbb{C}, \Vdash)$, made of

- a fixed set $\mathbb{S}$ of elements called *sorts*, denoted s, s_1, etc.
- a fixed set $\mathbb{T}$ of elements called *terms*, denoted r, r_1, etc,
- a fixed set $\mathbb{C}$ of elements called *sorting contexts*, denoted Σ, Σ_1, etc,
- a *sorting relation*, i.e. a set of elements $(\_ \Vdash \_ : \_) : (\mathbb{C} \times \mathbb{T} \times \mathbb{S})$

※

5.1.2 Atoms and Molecules

The next group of parameters $(\mathbb{A}, \mathbb{M}, \equiv)$ adapts the notions of atoms and molecules to the presence of quantifiers.

Atoms and molecules now need more structure than in the propositional case, because intuitively, we would like atoms and molecules to refer to terms. Whether it is in the decomposition relation or elsewhere in the LAF inference system, we will have a rule where witnesses for existential variables are picked. This usually involves substituting the witness for the existential variable in the premiss of the rule.

Two reasons suggest to go for a different approach:

First, this requires us to specify how substitutions affect atoms and molecules; which then requires us to specify what variables and terms are for the abstract notions of atoms and molecules; then we would probably need to specify how substitutions affect the decomposition relation. All of which are rather heavy in our abstract setting.

Second, an implementation of proof-search would probably depart from such a rule anyway, as it could be costly to traverse the whole sequent, or even just some of its atoms and molecules, to compute the substitution every time a witness is picked. The substitution would more efficiently be done lazily, keeping the fact that the existential variable "is in fact the witness" in a separate data-structure to be looked up when we finally need the information.

Hence, we will formalise what would be actually closer to implementation, expressing an atom (it will be the same for a molecule) as a pair $(a, \mathbf{r})$ where a is a structure not (explicitly) refering to terms and $\mathbf{r}$ is a list of terms: the former is a *parameterised atoms* and the latter is its list of parameters. The list of parameters $\mathbf{r}$ can be seen as a delayed substitution, in the view that a refers to its parameters by either using something like De Bruijn's indices, or by having a series of λ-abstractions at its top-level.

For instance, to represent the atoms of first-order logic, we could use a pair

$$(p(4, \#1, \#2), x::5::[\,])$$

(where p is a ternary predicate symbol and x is an eigenvariable) to represent the atom usually written as $p(4, x, 5)$.

A parameterised atom (such as $p(4, \#1, \#2)$) comes with a notion of *arity*: a list of sorts l describing the sorts of its parameters numbered from 1 to $|l|$ (the arity of $p(4, \#1, \#2)$ would here be a list of length at least 2).

This leads to the following definition.

Definition 73 (Atoms & molecules)

LAF is parameterised by two sets $\mathbb{A}$ and $\mathbb{M}$, whose elements are respectively called *(parameterised) atoms* (denoted a, a',...) and *(parameterised) molecules* (denoted M, M',...), each of which is equipped with a function that maps every atom a (resp. molecule M) to a list of sorts denoted $|a|$ (resp. $|M|$) and called its *arity*.

The set $\mathbb{A}_{\downarrow}$ (resp. $\mathbb{M}_{\downarrow}$) of *instanciated atoms* (resp. *instanciated molecules*) is the set of pairs of the form $(a, \mathbf{r})$ (resp. $(M, \mathbf{r})$), where a is an atom (resp. M is a molecule) of arity l and $\mathbf{r}$ is a list of terms of length $|l|$.[1]

LAF is also parameterised by an equivalence relation $\equiv$ over $\mathbb{A}_{\downarrow}$, which we call *equality*. ※

The equality relation is what replaces the computation of substitutions: using the previous example, if $(p(4, \#1, \#2), x::5::[\,])$ "represents" the atom usually written as $p(4, x, 5)$, so do the pairs $(p(4, \#2, \#1), 5::x::[\,])$, $(p(4, \#1), x::[\,])$ and $(p(\#3, \#1, \#2), x::5::4::[\,])$. The equality relation on instantiated atoms is then used to declare all of these pairs be equal.

Interestingly enough, this equality relation will only be used at the leaves of proof-trees when proof-search has to compare two instantiated atoms to close the branch. More surprisingly, there is no need to have a similar equality relation between molecules; they are never compared during proof-search.

5.1.3 Typing decompositions and typing contexts

As we have seen, if the choice of witnesses for existential variables is made in the decomposition relation, the asynchronous phase treats universal variables in the style of the ω-rule. To avoid this, the choice of witnesses cannot be made in the decomposition relation; instead, we "leave a hole" and delay its filling until we inhabit typing decompositions.

Therefore, the notion of typing decomposition itself needs to be enriched with a new construct, denoted $s.\Delta$, that we use to mark a place where an existential variable of sort s was found while decomposing a molecule: For instance we can use the construct in the decomposition relation with a rule (again, forgetting proof-terms) such as

[1] We do not relate the sorts specified in l to the sorts of the terms, which only make sense in a specific sorting context.

$$\frac{\Delta \Vdash \left\{ {}^{\#1}\!/_{x} \right\} A}{s.\Delta \Vdash \exists x^s A}$$

where #1 is a temporary name for the hole (you may think of it as a De Bruijn's index), or with the equivalent rule

$$\frac{\Delta \Vdash A}{s.\Delta \Vdash \exists^s A}$$

if De Bruijn's indices are already used in formulae to represent bound variables.

The choice of witness will then be made when proving/inhabiting $s.\Delta$.

Correspondingly, we extend the notion of typing decompositions as follows:

DEFINITION 74 (Typing decompositions)

The *typing decomposition algebra*, denoted $\mathbb{D}$, is the family of sets $(\mathbb{D}_l)_l$ defined by the following grammar:

$$\Delta^l, \Delta^l_1, \ldots ::= \; a^l \,|\, {\sim}M^l \,|\, \bullet \,|\, \Delta^l_1, \Delta^l_2 \,|\, s.\Delta^{s::l}$$

where $\Delta^l, \Delta^l_1, \ldots$ range over $\mathbb{D}_l$, a^l ranges over parameterised atoms of arity l and M^l ranges over parameterised molecules of arity l (and s still ranges over $\mathbb{S}$).

Elements of $\mathbb{D}_l$ are called *typing decompositions* of arity l.

The set $\mathbb{D}_\downarrow$ of *instanciated typing decompositions* is the set of pairs of the form $(\Delta^l, \mathbf{r})$ where Δ^l is a typing decomposition of arity l and $\mathbf{r}$ is a list of terms of length $|l|$. ※

Here, the construct $s.\Delta^{s::l}$ declares a new "hole" of sort s so that the atoms and molecules in $\Delta^{s::l}$ may now depend on this extra parameter.

But notice that typing decompositions (unless instantiated) never mention terms; they will be used to decompose *parameterised* molecules, rather than *instantiated* molecules: this is because, intuitively, the decomposition of a molecule into a typing decomposition only concerns the logical structure of the molecule, not the terms that it contains.[2]

Now, as in the quantifier-free case, typing decompositions will be used to extend typing contexts, but we do expect the types, in the typing declarations of a typing context, to be *instantiated* atoms and molecules.

This raises two questions when extending a typing context Γ with a typing decomposition Δ^l:

- how do the parameterised atoms and molecules of Δ^l turn into instantiated atoms and molecules in the extended environment?
- how should the new construct $s.\Delta^{s::l}$ impact the extension?

To answer these questions, we anticipate that, as in the quantifier-free case, the typing context Γ gets extended in the async rule. In our setting with quantifiers, that rule will be used to refute an *instantiated* molecule. This extension thus takes place in presence of the list of parameters $\mathbf{r}$ of the molecule we are refuting, and the length of this list will match the arity l of Δ^l.

Therefore, the operation that we need to extend environments is of the form $\Gamma; (\Delta^l, \mathbf{r})$, hence the notion of instanciated typing decomposition.

[2] This requires the distinction between the two to be clear, which will prevent us from modelling higher-order logic.

Definition 75 (Typing contexts)

LAF is parameterised by two sets Lab_+ and Lab_-, of elements called *positive labels* and *negative labels*, respectively.

LAF is then parameterised by an algebra Co of the form

$$\left(\mathsf{Co}, \begin{pmatrix} \mathsf{Co}\times\mathsf{Lab}_+ \to \mathbb{A}_\downarrow \\ (\Gamma, x^+) \mapsto \Gamma\left[x^+\right] \end{pmatrix}, \begin{pmatrix} \mathsf{Co}\times\mathsf{Lab}_- \to \mathbb{M}_\downarrow \\ (\Gamma, x^-) \mapsto \Gamma\left[x^-\right] \end{pmatrix}, \begin{pmatrix} \mathsf{Co} \to \mathbb{C} \\ \Gamma \mapsto \Gamma^e \end{pmatrix}, \begin{pmatrix} \mathsf{Co}\times\mathbb{D}_\downarrow \to \mathsf{Co} \\ (\Gamma, (\Delta^l, \mathbf{r})) \mapsto \Gamma;(\Delta^l, \mathbf{r}) \end{pmatrix}\right)$$

whose elements are called *typing contexts*.

We denote by $\mathsf{dom}^+(\Gamma)$ (resp. $\mathsf{dom}^-(\Gamma)$) the subset of Lab_+ (resp. Lab_-) where $\Gamma\left[x^+\right]$ (resp. $\Gamma\left[x^-\right]$) is defined. ※

Of course, nothing in the above definition specifies the behaviour of the operation $\Gamma;(\Delta^l, \mathbf{r})$.

This will not be problematic to define the LAF system, nor to define its realisability models; but in order to *build* those, it will be easier if we also know that the typing context algebra satisfies more specific properties. In Section 5.2 we give an example of LAF instance where the behaviour of $\Gamma;(\Delta^l, \mathbf{r})$ is specified.

Also, note the presence of the operation Γ^e that extracts from Γ a sorting context, which will be used in the LAF system to constrain the pick of witnesses.

Notice in the above two definitions (74 and 75) that, in contrast to what we did in the quantifier-free case, we have here directly defined typing decompositions and typing contexts instead of defining them as particular instances of generic decompositions and generic contexts. This is due to the need of taking into account, in those data structures, the parameters $\mathbf{r}$ that are specific to atoms and molecules and non-existant for arbitrary sets $\mathcal{A}$ ad $\mathcal{B}$. However, we shall still define generic decompositions and generic contexts, as these will be used for instance to build models of LAF, and also more immediately to define the *structure* of a typing derivation (as we did in the quantifier-free case).

Definition 76 (Generic decomposition algebras and decomposition structures)

Given three set $\mathcal{A}$, $\mathcal{B}$, and $\mathcal{C}$, the $(\mathcal{A},\mathcal{B},\mathcal{C})$-*decomposition algebra* $\mathbb{D}_{\mathcal{A},\mathcal{B},\mathcal{C}}$, whose elements are called $(\mathcal{A},\mathcal{B},\mathcal{C})$-*decompositions*, is the free algebra defined by the following grammar:

$$\Delta, \Delta_1, \ldots ::= a \,|\, {\sim}b \,|\, \bullet \,|\, \Delta_1, \Delta_2 \,|\, c.\Delta$$

where a (resp. b, c) ranges over $\mathcal{A}$ (resp. $\mathcal{B}$, $\mathcal{C}$).

Let $\mathbb{D}_{\mathsf{st}}$ abbreviate $\mathbb{D}_{\mathtt{unit},\mathtt{unit},\mathtt{unit}}$, whose elements we call *decomposition structures*.

The *(decomposition) structure* of an $(\mathcal{A},\mathcal{B},\mathcal{C})$-decomposition Δ, denoted $|\Delta|$, is its obvious homomorphic projection in $\mathbb{D}_{\mathsf{st}}$.

The *(decomposition) structure* of a typing decomposition Δ^l, denoted $\left|\Delta^l\right|$, is defined as follows:

$$\begin{array}{lcllcl} \left|a^l\right| & := & () & \left|{\sim}M^l\right| & := & () \\ |\bullet| & := & () & \left|\Delta_1^l, \Delta_2^l\right| & := & \left|\Delta_1^l\right|, \left|\Delta_2^l\right| \\ \left|s.\Delta^{s::l}\right| & := & ().\left|\Delta^{s::l}\right| & & & \end{array}$$

※

Here, we see that the typing decomposition algebra is more subtle than the $(\mathbb{A},\mathbb{M},\mathbb{S})$-decomposition algebra, because arities are taken into account.

Similarly, we here define generic contexts, which will be used in the next chapters.

DEFINITION 77 (Generic contexts)

Given four sets $\mathcal{A}$, $\mathcal{B}$, $\mathcal{C}$ and $\mathcal{D}$, an $(\mathcal{A},\mathcal{B},\mathcal{C},\mathcal{D})$-*context algebra* is an algebra of the form

$$\left(\mathcal{G}, \begin{pmatrix} \mathcal{G}\times\mathsf{Lab}_+ \rightharpoonup \mathcal{A} \\ (\Gamma, x^+) \mapsto \Gamma\left[x^+\right] \end{pmatrix}, \begin{pmatrix} \mathcal{G}\times\mathsf{Lab}_- \rightharpoonup \mathcal{B} \\ (\Gamma, x^-) \mapsto \Gamma\left[x^-\right] \end{pmatrix}, \begin{pmatrix} \mathcal{G}\to\mathcal{D} \\ \Gamma\mapsto\Gamma^e \end{pmatrix}, \begin{pmatrix} \mathcal{G}\times\mathbb{D}_{\mathcal{A},\mathcal{B},\mathcal{C}}\to\mathcal{G} \\ (\Gamma,\Delta) \mapsto \Gamma;\Delta \end{pmatrix}\right)$$

whose elements are called $(\mathcal{A},\mathcal{B},\mathcal{C},\mathcal{D})$-*contexts*.

Again, we denote by $\mathsf{dom}^+(\Gamma)$ (resp. $\mathsf{dom}^-(\Gamma)$) the subset of Lab_+ (resp. Lab_-) where $\Gamma\left[x^+\right]$ (resp. $\Gamma\left[x^-\right]$) is defined. ※

5.1.4 Logical connectives

The concepts of patterns and decomposition relations are unchanged, except they rely on the enriched concepts of atoms, molecules and typing decompositions.

DEFINITION 78 (Patterns & decomposition relation)

LAF is parameterised by a *pattern algebra*, an algebra of the form

$$\left(\mathsf{Pat}, \begin{pmatrix} \mathsf{Pat}\to\mathbb{D}_{\mathsf{st}} \\ p \mapsto |p| \end{pmatrix}\right)$$

whose elements are called *patterns*, and by a *decomposition relation* (for every l), i.e. a set of elements

$$(\_ \Vdash \_ : \_) : (\mathbb{D}_l \times \mathsf{Pat} \times \mathbb{M}_l)$$

such that if $\Delta \Vdash p : M$ then the structure of Δ is $|p|$. ※

5.1.5 Definition of the system

DEFINITION 79 (Proof-Terms) Proof-terms are defined by the following grammar:

Positive terms	Terms^+	$t^+ ::= pd$
Decomposition terms	$\mathsf{Terms}^{\mathsf{d}}$	$d ::= x^+ \mid f \mid \bullet \mid d_1, d_2 \mid r.d$
Commands	Terms	$c ::= \langle x^- \mid t^+ \rangle \mid \langle f \mid t^+ \rangle$

where p ranges over Pat, x^+ ranges over Lab_+, x^- ranges over Lab_-, and f ranges over $\mathsf{Pat} \rightharpoonup \mathsf{Terms}$. ※

We can finally present the typing system LAF:

DEFINITION 80 (LAF)

LAF is the inference system of Fig. 27 defining the derivability of three kinds of sequents

$$\begin{array}{lcl} (\_ \vdash [\_:\_]) & : & (\mathsf{Co} \times \mathsf{Terms}^+ \times \mathbb{M}_\downarrow) \\ (\_ \vdash \_:\_) & : & (\mathsf{Co} \times \mathsf{Terms}^{\mathsf{d}} \times \mathbb{D}_\downarrow) \\ (\_ \vdash \_) & : & (\mathsf{Co} \times \mathsf{Terms}) \end{array}$$

We further impose in rule async that the domain of function f be exactly those patterns that can decompose M (if $p \in \mathsf{Dom}(f)$ then there exists Δ such that $\Delta \Vdash p : M$).

$\mathsf{LAF}^{\mathsf{cf}}$ is the inference system LAF without the cut-rule. ※

5.2 Extending LAF_{K1} with quantifiers

In this section we extend to multi-sorted first-order logic the example of polarised classical logic (one-sided version LAF_{K1}) in Section 4.3. Such a first-order extension could also be done

$$\frac{\Delta \Vdash p : M \quad \Gamma \vdash d : (\Delta, \mathbf{r})}{\Gamma \vdash [pd : (M, \mathbf{r})]}\ \mathsf{sync}$$

$$\frac{}{\Gamma \vdash \bullet : (\bullet, \mathbf{r})} \qquad \frac{\Gamma \vdash d_1 : (\Delta_1, \mathbf{r}) \quad \Gamma \vdash d_2 : (\Delta_2, \mathbf{r})}{\Gamma \vdash d_1, d_2 : ((\Delta_1, \Delta_2), \mathbf{r})} \qquad \frac{\Gamma^e \Vdash r' : s \quad \Gamma \vdash d : (\Delta, r' :: \mathbf{r})}{\Gamma \vdash r'.d : s.(\Delta, \mathbf{r})}$$

$$\frac{\Gamma\left[x^+\right] \equiv (a, \mathbf{r})}{\Gamma \vdash x^+ : (a, \mathbf{r})}\ \mathsf{Init} \qquad \frac{\forall p, \forall \Delta, \quad \Delta \Vdash p : M \quad \Rightarrow \quad \Gamma; (\Delta, \mathbf{r}) \vdash f(p)}{\Gamma \vdash f : (\sim M, \mathbf{r})}\ \mathsf{async}$$

$$\frac{\Gamma \vdash [t^+ : \Gamma\,[x^-]]}{\Gamma \vdash \left\langle x^- \mid t^+ \right\rangle}\ \mathsf{Select} \qquad \frac{\Gamma \vdash f : (\sim M, \mathbf{r}) \qquad \Gamma \vdash [t^+ : (M, \mathbf{r})]}{\Gamma \vdash \left\langle f \mid t^+ \right\rangle}\ \mathsf{cut}$$

Figure 27: LAF

for the two-sided versions of polarised classical logic or intuitionistic logic.

To handle quantifiers, we make a clear separation between *bound variables* and *eigenvariables*: the intuition being that in order to prove $\forall x\ p(x)$ we prove $p(\mathfrak{x})$ for "an arbitrary $\mathfrak{x}$", using an eigenvariable $\mathfrak{x}$.

Actually, the reasons why we used "labels" instead of "variables" in the quantifier-free system also apply to eigenvariables: both in the perspective of an implementation and for the formalisation of such an abstract system as LAF, it will be convenient to have a deterministic way to name eigenvariables with no notion of α-conversion or equivariance. We will therefore call them *eigenlabels*.

LAF is more flexible regarding bound variables, which could be named and subject to α-conversion. However, already using De Bruijn's indices to represent binding in the syntax of formulae will be convenient since, as already mentioned in Section 5.1.3, we can simply write

$$\frac{\Delta \Vdash A}{s.\Delta \Vdash \exists^s A}$$

instead of

$$\frac{\Delta \Vdash \left\{ {}^{\#1}\!/_{x} \right\} A}{s.\Delta \Vdash \exists x^s A}$$

saving us the trouble of defining the substitution operation on formulae.

Definition 81 (Literals, formulae, patterns, decomposition)

Let $\mathbb{S}$ be a set of *sorts* and Ξ be an $\mathbb{S}$-signature in the sense of multi-sorted first-order logic. Predicate arities are represented as lists of sorts, denoted $l, l', \ldots$.

Given such an arity l, the set of *l-literals* is the set of literals over Ξ (well-sorted atomic propositions and their negations) whose free variables are among $\#1, \ldots, \#|l|$ with (respective) sorts given by l.

Consider a subset of the set of predicate symbols, whose elements are called *positive predicate symbols*; predicate symbols that are not in that set are called *negative*.

Let the set $\mathbb{A}_l$ of *parameterised atoms of arity l* be the set of l-literals that are either of the form $p(t_1, \ldots, t_n)$, with p being a positive predicate symbol, or of the form $\neg p(t_1, \ldots, t_n)$, with p being a negative predicate symbol.

Similarly to Definition 60, let the set $\mathbb{M}_l$ of *parameterised molecules of arity l* be the set defined by the first line of the following grammar for (polarised) formulae of classical logic:

$$\begin{array}{lll} \text{Positive } l\text{-formulae} & P^l, \ldots & ::= a^l \mid \top^+ \mid \bot^+ \mid A^l \wedge^+ B^l \mid A^l \vee^+ B^l \mid \exists^s A^{s::l} \\ \text{Negative } l\text{-formulae} & N^l, \ldots & ::= a^{l^\perp} \mid \top^- \mid \bot^- \mid A^l \wedge^- B^l \mid A^l \vee^- B^l \mid \forall^s A^{s::l} \\ \text{Unspecified } l\text{-formulae} & A^l & ::= P^l \mid N^l \end{array}$$

with a^l ranging over $\mathbb{A}_l$ and $a^{l^\perp}$ ranging over l-literals that are not in $\mathbb{A}_l$.

Similarly to Definition 60, let negation be the involutive function defined as follows:

$$\begin{array}{llll} (p(t_1, \ldots, t_n))^\perp & := \neg p(t_1, \ldots, t_n) & & \\ (\neg p(t_1, \ldots, t_n))^\perp & := p(t_1, \ldots, t_n) & & \\ \top^{+\perp} & := \bot^- & \top^{-\perp} & := \bot^+ \\ \bot^{+\perp} & := \top^- & \bot^{-\perp} & := \top^+ \\ (A \wedge^+ B)^\perp & := A^\perp \vee^- B^\perp & (A \wedge^- B)^\perp & := A^\perp \vee^+ B^\perp \\ (A \vee^+ B)^\perp & := A^\perp \wedge^- B^\perp & (A \vee^- B)^\perp & := A^\perp \wedge^+ B^\perp \end{array}$$

and we extend it to sets or multisets of formulae pointwise.

The set Pat of *patterns* extends that of Definition 60 according to the following grammar:

$$p, p_1, p_2, \ldots ::= \_^+ \mid \_^- \mid \bullet \mid (p_1, p_2) \mid \mathsf{inj}_i(p) \mid \exists p$$

The decomposition relation $(\_ \Vdash \_ : \_) : (\mathbb{D} \times \mathsf{Pat} \times \mathbb{M})$ is the extension of that of Definition 60, as shown in Fig. 28. ※

$$\frac{}{\bullet \Vdash \bullet : \top^+} \qquad \frac{}{\sim N^\perp \Vdash \_^- : N} \qquad \frac{}{a \Vdash \_^+ : a}$$

$$\frac{\Delta_1 \Vdash p_1 : A_1 \quad \Delta_2 \Vdash p_2 : A_2}{\Delta_1, \Delta_2 \Vdash (p_1, p_2) : A_1 \wedge^+ A_2} \qquad \frac{\Delta \Vdash p : A_i}{\Delta \Vdash \mathsf{inj}_i(p) : A_1 \vee^+ A_2} \qquad \frac{\Delta \Vdash p : A}{s.\Delta \Vdash \exists p : \exists^s A}$$

Figure 28: Decomposition relation for LAF_{K1}

Several concepts are still missing to define an instance of LAF: we need to define the set $\mathbb{T}$ of terms, the set $\mathbb{C}$ of sorting contexts, the sorting relation $\Vdash\!\!\!\Vdash$, the equality relation $\equiv$ on instantiated atoms, and the typing context algebra Co.

DEFINITION 82 (Terms, sorting and equality)

Let Lab_e be a copy of the set of natural numbers, whose elements are called *eigenlabels* and denoted n^e, n_1^e, ...

The set $\mathbb{T}$ of *terms*, denoted $r, r', \ldots$, is defined as the set of first-order terms whose variables are eigenlabels and whose function symbols are those declared in the signature Ξ.

The set $\mathbb{C}$ of *sorting contexts*, denoted $\Sigma, \Sigma', \ldots$, is $\mathsf{Lab}_e \rightharpoonup \mathbb{S}$.
We write $\Sigma \Vdash r : s$ when the term r is of sort s in the sorting context Σ, according to the signature Ξ.

We define the equality relation as follows: $(a^l, \mathbf{r}) \equiv (a'^{l'}, \mathbf{r}')$ if the literal $\left\{ \mathbf{r} /_{\#1,\ldots,\#|l|} \right\} a^l$ [3] is syntactically equal to the literal $\left\{ \mathbf{r}' /_{\#1,\ldots,\#|l'|} \right\} a'^{l'}$.[4] ※

The last task is to define the typing context algebra Co. We do this by adapting Definitions 69 and 70. We will use De Bruijn's levels for eigenlabels, because as explained in Section 4.4, De Bruijn's levels do not need to be updated once they are introduced (in contrast to De Bruijn's indices).

Definition 83 (Typing context algebra)

We define the support set of Co as the set of triples $(\Gamma^+, \Gamma^-, \Gamma^e)$ where Γ^+ is a list of elements of $\mathbb{A}_\downarrow$, Γ^- is a list of elements of $\mathbb{M}_\downarrow$, and Γ^e is a list of elements of $\mathbb{T}$.
As in Definition 69, two disjoint copies Lab_+ and Lab_- of the set of natural numbers are used for positive labels and negative labels, respectively denoted n^+ and n^-, and we define
$(\Gamma^+, \Gamma^-, \Gamma^e)\,[n^+]$ as the $(|\Gamma^+| - n)^{th}$ element of Γ^+
$(\Gamma^+, \Gamma^-, \Gamma^e)\,[n^-]$ as the $(|\Gamma^-| - n)^{th}$ element of Γ^-.

We now also define $(\Gamma^+, \Gamma^-, \Gamma^e)\,[n^e]$ as the $(|\Gamma^e| - n)^{th}$ element of Γ^e, for an eigenlabel n^e.
We turn the resulting structure

$$\left(\mathsf{Co}, \begin{pmatrix} \mathsf{Co} \times \mathsf{Lab}_+ \rightharpoonup \mathbb{A}_\downarrow \\ (\Gamma, n^+) \mapsto \Gamma\,[n^+] \end{pmatrix}, \begin{pmatrix} \mathsf{Co} \times \mathsf{Lab}_- \rightharpoonup \mathbb{M}_\downarrow \\ (\Gamma, n^-) \mapsto \Gamma\,[n^-] \end{pmatrix}, \begin{pmatrix} \mathsf{Co} \to (\mathsf{Lab}_e \rightharpoonup \mathbb{T}) \\ \Gamma \mapsto (n^e \mapsto \Gamma\,[n^e]) \end{pmatrix}\right)$$

into a typing context algebra, by adding a notion of typing context extension

$$\begin{pmatrix} \mathsf{Co} \times \mathbb{D}_\downarrow \to \mathsf{Co} \\ (\Gamma, (\Delta^l, \mathbf{r})) \mapsto \Gamma; (\Delta^l, \mathbf{r}) \end{pmatrix}$$

which we define as follows:

$$\begin{array}{lcl}
(\Gamma^+, \Gamma^-, \Gamma^e); (a^l, \mathbf{r}) & := & ((a^l, \mathbf{r}) :: \Gamma^+, \Gamma^-, \Gamma^e) \\
(\Gamma^+, \Gamma^-, \Gamma^e); (\sim M^l, \mathbf{r}) & := & (\Gamma^+, (M^l, \mathbf{r}) :: \Gamma^-, \Gamma^e) \\
(\Gamma^+, \Gamma^-, \Gamma^e); \bullet & := & (\Gamma^+, \Gamma^-, \Gamma^e) \\
(\Gamma^+, \Gamma^-, \Gamma^e); ((\Delta_1^l, \Delta_2^l), \mathbf{r}) & := & (\Gamma^+, \Gamma^-, \Gamma^e); (\Delta_1^l, \mathbf{r}); (\Delta_2^l, \mathbf{r}) \\
(\Gamma^+, \Gamma^-, \Gamma^e); (s.\Delta^{s::l}, \mathbf{r}) & := & (\Gamma^+, \Gamma^-, s :: \Gamma^e); (\Delta^{s::l}, |\Gamma^e|^e :: \mathbf{r})
\end{array}$$

※

The operation of typing context extension adapts to the presence of quantifiers the operation defined in Definition 69 for the quantifer-free case.

The only difference is the third component Γ^e of the typing context, which records the declared eigenlabels together with their sorts. This sorting context is extended whenever the typing context is extended with an instantiated typing decomposition of the form $(s.\Delta^{s::l}, \mathbf{r})$, which creates a new eigenlabel of sort s, which becomes the new head of the sorting context. As we use De Bruijn's levels rather than De Brujn's indices, the new eigenlabel is therefore $|\Gamma^e|^e$ (and it gets added to the current list of terms). This can be seen as picking the "first available name" for the eigenlabel to be created, a process that is often used in implementations of such systems.

[3] i.e. the substitution of $\mathbf{r}$ for $\#1, \ldots, \#|l|$ in a^l
[4] i.e. the substitution of $\mathbf{r}'$ for $\#1, \ldots, \#|l'|$ in $a'^{l'}$

With De Bruijn's indices rather than De Bruijn's levels, the new eigenlabel would be 0^e, but the price to pay for this is that the previously declared eigenlabels have "changed names", i.e. would be referred to as $(n+1)^e$ instead of n^e. Every structure referring to those previously declared eigenlabels (namely, the instanciated atoms and molecules in Γ^+ and Γ^-, as well as $\mathbf{r}$ itself) would then need to be updated with the name change.

We could easily define variants of the above system to quantify over other objects than first-order terms, as most of the definitions are rather modular: For example we could quantify over simply-typed λ-terms to design a LAF instance similar to the type theory $\lambda\Pi$ (see e.g. [Bar91]), except our proof-terms do not have the same syntax and and typing properties as the λ-terms we quantify over.

For this we take the same definitions as for multi-sorted first-order logic, except in Definition 81 we take $\mathbb{S}$ be the set of *simple types*, and in Definition 82 we take terms to be λ-terms, we take $\Sigma \Vdash r : s$ to be the typing relation of the simply-typed λ-calculus, and we define atom equality with β- (or $\beta\eta$-) conversion instead of syntactic equality: $(a^l, \mathbf{r}) \equiv (a'^{l'}, \mathbf{r}')$ if $(\lambda\#1 \dots \#|l|.a^l)\ \mathbf{r} \quad \longleftrightarrow_\beta^* \quad (\lambda\#1 \dots \#|l'|.b'^{l'})\ \mathbf{r}'$ (or similarly with $\beta\eta$).

All of the other definitions are the same.

Chapter 6

Realisability models of abstract focussing

Contents

In this chapter we investigate the semantics of LAF. More precisely, we investigate models of proofs / typing derivations with the *Adequacy Lemma* as the main objective: In very generic terms, if t is of type A then in the model we want the interpretation of t to be in the interpretation of A (if that is a set, or we want the interpretation of t to satisfy the interpretation of A, if that is a predicate).

Of course there are many models satisfying the above, starting with the uninformative ones where everything is collapsed.[1] So we investigate here a class of models, as large as possible, and prove the Adequacy Lemma generically for that class; then we will show interesting models in that class for which the Adequacy Lemma (that we now have for free) is informative (e.g. concludes the consistency of LAF, despite the presence of cuts and without proving cut-elimination).

This class of models is that of *abstract realisability algebras*; the specifications that we require of such an algebra do depend on the instance of LAF that we want to model - they will be different if we are to model for instance LAF_{K1}, LAF_{K2}, or LAF_{J}; but we can give those specifications parametrically and prove the Adequacy Lemma generically.

Hence in this chapter we start by assuming we are given an instance of LAF

[1] Interpret every type by the same singleton set and every inhabitant of that type by the inhabitant of the singleton set, and the Adequacy Lemma trivially holds but does not provide any useful information.

$$(\mathbb{S}, \mathbb{T}, \mathbb{C}, \Vdash, \mathbb{A}, \mathbb{M}, \equiv, \mathsf{Lab}_+, \mathsf{Lab}_-, \mathsf{Co}, \mathsf{Pat}, \Vdash)$$

Section 6.1 gives the specifications needed to interpret terms, Section 6.2 gives the specifications needed to interpret types and proves the Adequacy Lemma. Finally, Section 6.5 exhibits a simple model from which we derive the consistency of LAF.

6.1 Model structures and the interpretation of proof-terms

In this section we interpret the proof-terms of LAF in a realisability algebra, and for this we introduce the notion of *model structure*.

DEFINITION 84 (Model structure)

A *model structure* is an algebra of the form

$$\left(\mathscr{T}, \mathscr{C}, \mathscr{L}, \mathscr{P}, \mathscr{N}, \perp, \tilde{\mathsf{Co}}, \begin{pmatrix} \mathsf{Pat} \to (\mathbb{D}_{\mathscr{L},\mathscr{N},\mathscr{T}} \to \mathscr{P}) \\ p \mapsto \tilde{p} \end{pmatrix}, \begin{pmatrix} \mathbb{T} \times \mathscr{C} \rightharpoonup \mathscr{T} \\ (r,\sigma) \mapsto [\![r]\!]_\sigma \end{pmatrix}, \begin{pmatrix} (\mathsf{Pat} \rightharpoonup \mathsf{Terms}) \times \tilde{\mathsf{Co}} \rightharpoonup \mathscr{N} \\ (f,\rho) \mapsto [\![f]\!]_\rho \end{pmatrix}\right)$$

where

- $\mathscr{T}, \mathscr{C}, \mathscr{L}, \mathscr{P}, \mathscr{N}$ are five arbitrary sets of elements called *term denotations*, *valuations*, *label denotations*, *positive denotations*, *negative denotations*, respectively;
- $\perp$ is a relation between negative and positive denotations ($\perp \subseteq \mathscr{N} \times \mathscr{P}$), called the *orthogonality relation*;
- $\tilde{\mathsf{Co}}$ is a $(\mathscr{L}, \mathscr{N}, \mathscr{T}, \mathscr{C})$-context algebra, whose elements, denoted $\rho, \rho', \ldots$, are called *semantic contexts*.

We extend the notation $[\![r]\!]_\sigma$ to apply to a list of terms $\mathbf{r}$: $[\![\mathbf{r}]\!]_\sigma$, using the standard map function on lists.

The $(\mathscr{L}, \mathscr{N}, \mathscr{T})$-decomposition algebra $\mathbb{D}_{\mathscr{L},\mathscr{N},\mathscr{T}}$ is abbreviated $\tilde{\mathbb{D}}$; its elements, denoted $\mathfrak{d}$, $\mathfrak{d}' \ldots$, are called *semantic decompositions*. ※

Given a model structure, we can define the interpretation of proof-terms. The model structure already gives an interpretation for the partial functions f from patterns to commands. We extend it to all proof-terms as follows

DEFINITION 85 (Interpretation of proof-terms)

Positive terms (in Terms^+) are interpreted as positive denotations (in $\mathscr{P}$),
decomposition terms (in $\mathsf{Terms}^{\mathsf{d}}$) are interpreted as semantic decompositions (in $\tilde{\mathbb{D}}$),
and commands (in Terms) are interpreted as pairs in $\mathscr{N} \times \mathscr{P}$ (that may or may not be orthogonal), according to the following definition:

$$[\![pd]\!]_\rho := \tilde{p}([\![d]\!]_\rho)$$

$$\begin{array}{lcl} [\![x^+]\!]_\rho & := & \rho\,[x^+] \\ [\![f]\!]_\rho & := & [\![f]\!]_\rho \text{ [2]} \\ [\![\bullet]\!]_\rho & := & \bullet \\ [\![d_1, d_2]\!]_\rho & := & [\![d_1]\!]_\rho, [\![d_2]\!]_\rho \\ [\![r.d]\!]_\rho & := & [\![r]\!]_{\rho^e}.[\![d]\!]_\rho \end{array} \qquad \begin{array}{lcl} [\![\langle x^- \mid t^+ \rangle]\!]_\rho & := & (\rho\,[x^-], [\![t^+]\!]_\rho) \\ [\![\langle f \mid t^+ \rangle]\!]_\rho & := & ([\![f]\!]_\rho, [\![t^+]\!]_\rho) \end{array}$$

※

[2] as given by the model structure

6.2 Realisability algebras, interpretation of types & Adequacy

Again, let us keep in mind the Adequacy Lemma: if t is of type A then the interpretation of t to be in the interpretation of A. We have already defined the interpretation of proof-terms in a model structure. We now proceed to define the interpretation of types.

In system LAF, there are three concepts of "type inhabitation" for atoms and molecules:

- "proving" an atom by finding a suitable positive label in the typing context (the inhabitant is in Lab_+);
- "proving" a molecule by choosing a way to decompose it into a typing decomposition (the inhabitant is in Terms^+);
- "refuting" a molecule by case analysing all the possible ways of decomposing it into a typing decomposition (the inhabitant is in $\mathsf{Pat} \rightharpoonup \mathsf{Terms}$).

As positive labels are interpreted in $\mathscr{L}$, positive proof-terms are interpreted in $\mathscr{P}$ and functions in $\mathsf{Pat} \rightharpoonup \mathsf{Terms}$ are interpreted in $\mathscr{N}$, we will correspondingly

- have an interpretation of every atom as a particular subset of $\mathscr{L}$;
- have a *positive* interpretation of every molecule as a particular subset of $\mathscr{P}$;
- have a *negative* interpretation of every molecule as a particular subset of $\mathscr{N}^{\mathsf{V}}$.

To make sure that we capture, in our notion of abstract realisability algebra, a wide class of models, the first of the three above interpretations will be left as a parameter; we barely impose any specification on this parameter. The other two, however, will be *defined* notions.

Also, we have in LAF a notion of *sorting* for terms, whose counter-part in an abstract realisability algebra is also left as a parameter to be fixed *ad libitum*. This leads to the following definition:

Definition 86 (Realisability algebra)

A *realisability algebra* is

- a model structure
- together with three functions

$$\left(\begin{array}{l}\mathbb{S}\to\mathbb{P}(\mathscr{T})\\ s\mapsto[\![s]\!]\end{array}\right),\qquad \left(\begin{array}{l}\mathbb{C}\to\mathbb{P}(\mathscr{C})\\ \Sigma\mapsto[\![\Sigma]\!]\end{array}\right),\qquad \left(\begin{array}{l}\mathbb{A}_l\to(\mathscr{T}^{|l|}\to\mathbb{P}(\mathscr{L}))\\ a^l\mapsto[\![a^l]\!]\end{array}\right)$$

  satisfying
  - if $\Sigma \Vdash r : s$ and $\sigma \in [\![\Sigma]\!]$ then ($[\![r]\!]_\sigma$ is defined and) $[\![r]\!]_\sigma \in [\![s]\!]$;
  - if $(a, \mathbf{r}) \equiv (a', \mathbf{r}')$ then for all $\sigma : \mathscr{C}$ we have $[\![a]\!]([\![\mathbf{r}]\!]_\sigma) = [\![a']\!]([\![\mathbf{r}']\!]_\sigma)$.[3]

※

Now notice in LAF that the derivability of the typing judgements for atoms and molecules is defined inductively together with the derivability of a typing judgement for typing decompositions; inhabitants of those are decomposition terms.

Therefore, we will also define an interpretation for typing decompositions, as subsets of $\mathbb{D}_{\mathscr{L},\mathscr{N},\mathscr{T}}$. For this we need to specify how to "lift relations to typing decompositions":

[3] if both sides are defined

DEFINITION 87 (Lifting relations) Given

- two relations $\mathcal{R}_1 \subseteq \mathbb{A}_l \times \mathscr{T}^{|l|} \times \mathscr{L}$ and $\mathcal{R}_2 \subseteq \mathbb{M}_l \times \mathscr{T}^{|l|} \times \mathscr{N}$ (for every arity l)
- a relation $\mathcal{R}_3 \subseteq \mathbb{S} \times \mathscr{T}$,
- an arity l and a list of term denotations $\mathfrak{rl}$ of length $|l|$,
- a typing decomposition Δ^l of arity l and a semantic decomposition $\mathfrak{d}$

we say that Δ^l $\mathfrak{rl}$-*relates to* $\mathfrak{d}$ *according to* $\mathcal{R}_1$, $\mathcal{R}_2$ *and* $\mathcal{R}_3$ if the relation $(\Delta^l, \mathfrak{rl})\ \mathcal{R}\ \mathfrak{d}$ can be derived by the following rules:

$$\frac{(a^l, \mathfrak{rl})\ \mathcal{R}_1\ \mathfrak{l}}{(a^l, \mathfrak{rl})\ \mathcal{R}\ \mathfrak{l}} \qquad \frac{(M^l, \mathfrak{rl})\ \mathcal{R}_2\ \mathfrak{n}}{(\sim M^l, \mathfrak{rl})\ \mathcal{R}\ \mathfrak{n}} \qquad \frac{}{(\bullet, \mathfrak{rl})\ \mathcal{R}\ \bullet}$$

$$\frac{(\Delta_1^l, \mathfrak{rl})\ \mathcal{R}\ \mathfrak{d}_1 \qquad (\Delta_2^l, \mathfrak{rl})\ \mathcal{R}\ \mathfrak{d}_2}{((\Delta_1^l, \Delta_2^l), \mathfrak{rl})\ \mathcal{R}\ \mathfrak{d}_1, \mathfrak{d}_2} \qquad \frac{s\ \mathcal{R}_3\ \mathfrak{r} \qquad (\Delta^{s::l}, \mathfrak{r}::\mathfrak{rl})\ \mathcal{R}\ \mathfrak{d}}{(s.\Delta^{s::l}, \mathfrak{rl})\ \mathcal{R}\ \mathfrak{r}.\Delta'}$$

※

Obviously in that case Δ^l and $\mathfrak{d}$ have the same decomposition structure.

The interpretation of types will be defined by simultaneous induction on molecules and typing decompositions. This induction needs to follow a well-founded relation:

DEFINITION 88 (Well-founded LAF instance)

We write $M^{l'} \leqslant M^l$ if there are Δ^l and p such that $\Delta^l \Vdash p : M^l$ and $M^{l'}$ is a leaf of Δ^l.
The LAF instance is *well-founded* if $\leqslant$ is well-founded. ※

It could be the case that the LAF instance is not well-founded, e.g. if molecules contain fixpoints.

Notice 1 In the rest of this chapter, we will assume LAF instances to be well-founded.

Under this assumption, the following interpretations of types are well-defined:

DEFINITION 89 (Interpretation of types and typing contexts) A realisability algebra already provides the interpretation of a parameterised atom of arity l in $(\mathscr{T}^{|l|} \to \mathbb{P}(\mathscr{L}))$.
We now define
the positive interpretation of a parameterised molecule of arity l in $(\mathscr{T}^{|l|} \to \mathbb{P}(\mathscr{P}))$;
the negative interpretation of a parameterised molecule of arity l in $(\mathscr{T}^{|l|} \to \mathbb{P}(\mathscr{N}))$;
the interpretation of a typing decomposition of arity l is in $(\mathscr{T}^{|l|} \to \mathbb{P}(\mathbb{D}_{\mathscr{L},\mathscr{N},\mathscr{T}}))$:

$$\begin{array}{lcll}
[\![M^l]\!]^+(\mathfrak{rl}) & := & \{\tilde{p}(\mathfrak{d}) \in \mathscr{P} & \mid \mathfrak{d} \in [\![\Delta^l]\!](\mathfrak{rl}),\ \text{and } \Delta^l \Vdash p : M^l\} \\
[\![M^l]\!]^-(\mathfrak{rl}) & := & \{\mathfrak{n} \in \mathscr{N} & \mid \forall \mathfrak{p} \in [\![M]\!]^+(\mathfrak{rl}), \mathfrak{n} \perp \mathfrak{p}\} \\
[\![\Delta^l]\!](\mathfrak{rl}) & := & \{\mathfrak{d} \in \tilde{\mathbb{D}} & \mid \Delta^l\ \mathfrak{rl}\text{-relates to } \mathfrak{d} \text{ according to } \{(a^l, \mathfrak{rl}, \mathfrak{l}) \mid \mathfrak{l} \in [\![a^l]\!](\mathfrak{rl})\} \\
 & & & \qquad \{(M^l, \mathfrak{rl}, \mathfrak{n}) \mid \mathfrak{n} \in [\![M^l]\!]^-(\mathfrak{rl})\} \\
 & & & \qquad \text{and } \{(s, \mathfrak{r}) \mid \mathfrak{r} \in [\![s]\!]\} \quad \}
\end{array}$$

We now define the semantics of instanciated atoms, molecules and typing decompositions:

$$\begin{array}{lcllcl}
[\![(a^l, \mathbf{r})]\!]_\sigma & := & [\![a^l]\!]([\![\mathbf{r}]\!]_\sigma) & [\![(M^l, \mathbf{r})]\!]^+_\sigma & := & [\![M^l]\!]^+([\![\mathbf{r}]\!]_\sigma) \\
[\![(\Delta^l, \mathbf{r})]\!]_\sigma & := & [\![\Delta^l]\!]([\![\mathbf{r}]\!]_\sigma) & [\![(M^l, \mathbf{r})]\!]^-_\sigma & := & [\![M^l]\!]^-([\![\mathbf{r}]\!]_\sigma)
\end{array}$$

We finally define the interpretation of a typing context:[4]

$$\begin{aligned}\llbracket\Gamma\rrbracket := \{\rho \in \tilde{\mathsf{Co}} \mid\ & \rho^e \in \llbracket\Gamma^e\rrbracket \\ & \forall x^+ \in \mathsf{dom}^+(\rho),\ \rho\,[x^+] \in \llbracket\Gamma\,[x^+]\rrbracket_{\rho^e} \\ & \forall x^- \in \mathsf{dom}^-(\rho),\ \rho\,[x^-] \in \llbracket\Gamma\,[x^-]\rrbracket^-_{\rho^e} \quad \}\end{aligned}$$

※

Now that we have defined the interpretation of terms and the interpretation of types, we prove the Adequacy Lemma.

Lemma 51 (Adequacy for LAF) We assume the following hypotheses:

Well-foundedness:
The LAF instance is well-founded.

Typing correlation:
If $\rho \in \llbracket\Gamma\rrbracket$ and $\mathfrak{d} \in \llbracket(\Delta^l, \mathbf{r})\rrbracket_{\rho^e}$ then $(\rho; \mathfrak{d}) \in \llbracket\Gamma; (\Delta^l, \mathbf{r})\rrbracket$.

Stability:
If $\mathfrak{d} \in \llbracket(\Delta^l, \mathbf{r})\rrbracket_\sigma$ for some $\Delta^l, \sigma, \mathbf{r}$ and $\llbracket f(p)\rrbracket_{\rho;\mathfrak{d}} \in \bot$, then $\llbracket f\rrbracket_\rho \perp \tilde{p}(\mathfrak{d})$.

We conclude that, for all $\rho \in \llbracket\Gamma\rrbracket$,

1. if $\Gamma \vdash [t^+ : (M^l, \mathbf{r})]$ then $\llbracket t^+\rrbracket_\rho \in \llbracket(M^l, \mathbf{r})\rrbracket^+$;
2. if $\Gamma \vdash d : (\Delta^l, \mathbf{r})$ then $\llbracket d\rrbracket_\rho \in \llbracket(\Delta^l, \mathbf{r})\rrbracket$;
3. if $\Gamma \vdash t$ then $\llbracket t\rrbracket_\rho \in \bot$.

※

Proof: See the proof in Coq [GL14]. □

6.3 A more concrete class of LAF instances

Looking at the Adequacy Lemma, the stability condition is traditional: it is the generalisation, to that level of abstraction, of the usual condition on the orthogonality relation in orthogonality models (those realisability models that are defined in terms of orthogonality, usually to model classical proofs [Gir87, DK00, Kri01, MM09, LM08]): orthogonality is "closed under anti-reduction". Here, we have not yet defined a notion of reduction on LAF proof-terms, but intuitively, we would expect, in order to reduce cuts, to rewrite $\langle f \mid pd\rangle$ to $f(p)$ "substituted by d".

On the other hand, the typing correlation property is new, and is due to the level of abstraction we operate at: there is no reason why our data structure for typing contexts would relate to our data structure for semantic contexts, and the extension operation, in both of them, has so far been completely unspecified. Hence, we clearly need such an assumption to relate the two.

However, one may wonder when and why the typing correlation property should be satisfied. Looking at the example of LAF_{K1}, one may anticipate how typing correlation could hold for this instance of LAF: at least in the quantifier-free case (Sections 4.3.1 and 4.4.1), we

[4] In this definition we implicitly require that $\mathsf{dom}^+(\rho) = \mathsf{dom}^+(\Gamma)$, $\mathsf{dom}^-(\rho) = \mathsf{dom}^-(\Gamma)$ and for all $x^+ \in \mathsf{dom}^+(\rho)$ (resp. $x^- \in \mathsf{dom}^-(\rho)$) $\llbracket\Gamma\,[x^+]\rrbracket_{\rho^e}$ (resp. $\llbracket\Gamma\,[x^-]\rrbracket^-_{\rho^e}$) is defined.

have a generic definition of $(\mathcal{A},\mathcal{B})$-contexts with a parametric operation of extension, which we can use for both typing contexts and semantic contexts.

In this section we generalise this example to a class of LAF systems (and later we identify a corresponding subclass of realisability algebras), where Adequacy holds under a hypothesis that simplifies (and entails) typing correlation, and that is satisfied in particular when the extension of contexts is defined parametrically.

6.3.1 LAF instances with eigenlabels

In this class, the extension of a typing context $\Gamma;(\Delta^l,\mathbf{r})$ is expressed in terms of the extension operation $\Gamma';\Delta'$ of an $(\mathbb{A}_\downarrow,\mathbb{M}_\downarrow,\mathbb{S},\mathbb{C})$-context algebra.

This is done along the same lines as in the example of LAF_{K1} in Section 5.2, i.e. with a notion of eigenlabel and the understanding of sorting contexts (in $\mathbb{C}$) a functions mapping eigenlabels to sorts. Hence, we update and refine previous definitions (and introduce new ones) with this understanding of sorting contexts.

DEFINITION 90 (Three-parameter contexts)

Assume we have three disjoint sets Lab_+, Lab_- and Lab_e, the union of which ($\mathsf{Lab}_+\cup\mathsf{Lab}_-\cup\mathsf{Lab}_e$) we denote Lab.

Given three sets $\mathcal{A}$, $\mathcal{B}$, $\mathcal{C}$, we abreviate the terminology $(\mathcal{A},\mathcal{B},\mathcal{C},\mathsf{Lab}_e \rightharpoonup \mathcal{C})$-context into $(\mathcal{A},\mathcal{B},\mathcal{C})$-context.

We also abbreviate $\Gamma^e(x)$ as $\Gamma\,[x]$, for an $(\mathcal{A},\mathcal{B},\mathcal{C})$-context Γ and an element $x\in\mathsf{Lab}_e$, writing $\mathsf{dom}^e(\Gamma)$ for $\mathsf{Dom}(\Gamma^e)$. Finally, we abbreviate $\mathsf{dom}^+(\Gamma)\cup\mathsf{dom}^-(\Gamma)\cup\mathsf{dom}^e(\Gamma)$ as $\mathsf{dom}(\Gamma)$, and we say that Γ is *empty* if $\mathsf{dom}(\Gamma)=\emptyset$. ※

We now introduce the lifting of relations to generic decompositions and contexts:

DEFINITION 91 (Lifting relations)

Assume we are given three relations $\mathcal{R}_1\subseteq\mathcal{A}\times\mathcal{A}'$, $\mathcal{R}_2\subseteq\mathcal{B}\times\mathcal{B}'$ and $\mathcal{R}_3\subseteq\mathcal{C}\times\mathcal{C}'$.

We say that *an* $(\mathcal{A},\mathcal{B},\mathcal{C})$*-decomposition* Δ *relates to an* $(\mathcal{A}',\mathcal{B}',\mathcal{C}')$*-decomposition* Δ' *according to* $\mathcal{R}_1$, $\mathcal{R}_2$ *and* $\mathcal{R}_3$ if the relation $\Delta\ \mathcal{R}\ \Delta'$ can be derived by the following rules:

$$\frac{a\ \mathcal{R}_1\ a'}{a\ \mathcal{R}\ a'}\qquad\frac{b\ \mathcal{R}_2\ b'}{b\ \mathcal{R}\ b'}\qquad\frac{}{\bullet\ \mathcal{R}\ \bullet}\qquad\frac{\Delta_1\ \mathcal{R}\ \Delta_1'\qquad\Delta_2\ \mathcal{R}\ \Delta_2'}{\Delta_1,\Delta_2\ \mathcal{R}\ \Delta_1',\Delta_2'}\qquad\frac{c\ \mathcal{R}_3\ c'\qquad\Delta\ \mathcal{R}\ \Delta'}{c.\Delta\ \mathcal{R}\ c'.\Delta'}$$

We say that *an* $(\mathcal{A},\mathcal{B},\mathcal{C})$*-context* Γ *relates to an* $(\mathcal{A}',\mathcal{B}',\mathcal{C}')$*-context* Γ' *according to* $\mathcal{R}_1$, $\mathcal{R}_2$ *and* $\mathcal{R}_3$ if[5]

$$\begin{array}{rr}\forall x^+\in\mathsf{Lab}_+, & \Gamma\,[x^+]\ \mathcal{R}_1\ \Gamma'\,[x^+]\\ \forall x^-\in\mathsf{Lab}_-, & \Gamma\,[x^-]\ \mathcal{R}_2\ \Gamma'\,[x^-]\\ \forall x\in\mathsf{Lab}_e, & \Gamma\,[x]\ \mathcal{R}_3\ \Gamma'\,[x]\end{array}$$

Assume we are now given three functions $f_1:\mathcal{A}\to\mathcal{A}'$, $f_2:\mathcal{B}\to\mathcal{B}'$ and $f_3:\mathcal{C}\to\mathcal{C}'$.
we say that Γ' *is a map of* Γ *according to* f_1, f_2 *and* f_3 if it relates to Γ according to the relations $\{(f_1(a),a)\mid a\in\mathcal{A}\}$, $\{(f_2(b),b)\mid b\in\mathcal{B}\}$ and $\{(f_3(c),c)\mid c\in\mathcal{C}\}$. ※

Using the above two definition, we can now say what a LAF instance with eigenlabels is:

[5] By writing the three conditions we implicitly request $\mathsf{dom}^+(\Gamma)=\mathsf{dom}^+(\Gamma')$, $\mathsf{dom}^-(\Gamma)=\mathsf{dom}^-(\Gamma')$ and $\mathsf{dom}^e(\Gamma)=\mathsf{dom}^e(\Gamma')$.

DEFINITION 92 (LAF instance with eigenlabels)

A LAF *instance with eigenlabels* is given by the following tuple:

$$(\mathbb{S}, \mathsf{Lab}_e, \mathbb{T}, \Vdash, \mathbb{A}, \mathbb{M}, \equiv, \mathsf{Lab}_+, \mathsf{Lab}_-, \mathsf{Co}, \mathsf{Pat}, \Vdash, \pi^{\mathcal{V}}_{\Delta}, \mathsf{st}^{\mathcal{V}}_{\Delta})$$

where

- $\mathbb{S}$ is as in Definition 72 (a set of elements called *sorts*, denoted s, s_1, etc);
- Lab_e is a set of elements called *eigenlabels*, denoted x, x'_1, etc;
- $\mathbb{T}$ is a set of elements called *terms*, denoted r, r_1, etc,
  - that extends the set Lab_e of eigenlabels,
  - and with a systematic way of lifting a function $\mathsf{Lab}_e \to \mathsf{Lab}_e$ to $\mathbb{T} \to \mathbb{T}$;

  We can then apply a function $\pi : \mathsf{Lab}_e \to \mathsf{Lab}_e$ to lists of terms (using the standard map function on lists);
- $\Vdash$ is a *sorting relation*, i.e. a set of elements $(\_ \Vdash \_ : \_) : ((\mathsf{Lab}_e \rightharpoonup \mathbb{S}) \times \mathbb{T} \times \mathbb{S})$, with
  - $\Sigma \Vdash x : s$ if and only if $s = \Sigma(x)$
  - for all $\pi : \mathsf{Lab}_e \to \mathsf{Lab}_e$, if $\Sigma \circ \pi \Vdash r : s$ then $\Sigma \Vdash \pi(r) : s$;
- $\mathbb{A}$, $\mathbb{M}$, Lab_+ and Lab_- are as in Definitions 73 and 75;

  We can apply a function $\pi : \mathsf{Lab}_e \to \mathsf{Lab}_e$ to instantiated atoms and molecules using

  $$\pi(a^l, \mathbf{r}) := (a^l, \pi(\mathbf{r})) \text{ and } \pi(M^l, \mathbf{r}) := (M^l, \pi(\mathbf{r}));$$

  We then impose that equality on instantiated atoms be stable under any such function $\pi : \mathsf{Lab}_e \to \mathsf{Lab}_e$: If $(a, \mathbf{r}) \equiv (a', \mathbf{r}')$ then $\pi(a, \mathbf{r}) \equiv \pi(a', \mathbf{r}')$.
- Co is an $(\mathbb{A}_{\downarrow}, \mathbb{M}_{\downarrow}, \mathbb{S})$-context algebra, called the *typing context algebra*, equipped with a *map* operation that associates, to a context Γ and two functions $f_1 : \mathbb{A}_{\downarrow} \to \mathbb{A}_{\downarrow}$, $f_2 : \mathbb{M}_{\downarrow} \to \mathbb{M}_{\downarrow}$, a context $(f_1, f_2) \circ \Gamma$ that is a map of Γ according to f_1, f_2 and the identity on sorts;
- Pat and $\Vdash$ are as in Definition 78.
- We have two functions, respectively called the *renaming policy* and the *fresh naming policy*, of the form

  $$\begin{pmatrix} \mathbb{P}(\mathsf{Lab}) \times \mathbb{D}_{\mathsf{st}} \to (\mathsf{Lab}_e \to \mathsf{Lab}_e) \\ (\mathcal{V}, \Delta) \quad \mapsto \pi^{\mathcal{V}}_{\Delta} \end{pmatrix}, \begin{pmatrix} \mathbb{P}(\mathsf{Lab}) \times \mathbb{D}_{\mathsf{st}} \to \mathbb{D}_{\mathtt{unit},\mathtt{unit},\mathsf{Lab}_e} \\ (\mathcal{V}, \Delta) \quad \mapsto \mathsf{st}^{\mathcal{V}}_{\Delta} \end{pmatrix}$$

  We abbreviate $\mathsf{st}^{\mathsf{dom}(\Gamma)}_{|\Delta|}$ as $\mathsf{st}^{\Gamma}_{\Delta}$ and $\pi^{\mathsf{dom}(\Gamma)}_{|\Delta|}$ as π^{Γ}_{Δ}.

※

Most of the above definition is rather straightforward when thinking of sorting contexts as assigning sorts to eigenlabels. What is probably more cryptic is the naming policies $\pi^{\mathcal{V}}_{\Delta}$ and $\mathsf{st}^{\mathcal{V}}_{\Delta}$ (as well as the map operation of typing contexts): they compensate for the fact that the extension operation of the typing context algebra Co is more basic than in Definition 78. In short, $\mathsf{st}^{\Gamma}_{\Delta}$ and π^{Γ}_{Δ} describe which labels are used in an extended typing context $\Gamma; \Delta$ (especially regarding the labels used in Γ). Section 6.3.3 explains this in details, but we first start with the example of LAF_{K1}.

6.3.2 LAF_{K1} is a LAF instance with eigenlabels

In this section we illustrate the above concept by giving an alternative definition for system LAF_{K1} (from Section 5.2), this time as a LAF instance with eigenvariables.

Among the parameters

$$(\mathbb{S}, \mathsf{Lab}_e, \mathbb{T}, \Vdash, \mathbb{A}, \mathbb{M}, \equiv, \mathsf{Lab}_+, \mathsf{Lab}_-, \mathsf{Co}, \mathsf{Pat}, \Vdash)$$

of a LAF instance with eigenvariables, the context algebra Co is perhaps the least obvious to identify for LAF_{K1}. We do this now, by going via the definition of a generic family of context algebras as we had done in the quantifier-free version of LAF_{K1} (Sections 4.3.1 and 4.4.1).

DEFINITION 93 (A generic family of context algebras)

Given three sets $\mathcal{A}$, $\mathcal{B}$ and $\mathcal{C}$, we define $\mathcal{G}_{\mathcal{A},\mathcal{B},\mathcal{C}}$ as the set of elements of the form $(\Gamma^+, \Gamma^-, \Gamma^e)$ where Γ^+ (resp. Γ^-, Γ^e) is a list of elements of $\mathcal{A}$ (resp. $\mathcal{B}$, $\mathcal{C}$).

As in Definition 83, three disjoint copies Lab_+, Lab_- and Lab_e of the set of natural numbers are used for positive labels, negative labels and eigenlabels, respectively denoted n^+, n^- and n^e, and we define
$(\Gamma^+, \Gamma^-, \Gamma^e)\,[n^+]$ as the $(n+1)^{th}$ element of Γ^+
$(\Gamma^+, \Gamma^-, \Gamma^e)\,[n^-]$ as the $(n+1)^{th}$ element of Γ^-
$(\Gamma^+, \Gamma^-, \Gamma^e)\,[n^e]$ as the $(n+1)^{th}$ element of Γ^e.

We turn the resulting structure

$$\left(\mathcal{G}_{\mathcal{A},\mathcal{B},\mathcal{C}}, \begin{pmatrix} \mathcal{G}_{\mathcal{A},\mathcal{B},\mathcal{C}} \times \mathsf{Lab}_+ \rightharpoonup \mathcal{A} \\ (\Gamma, n^+) \quad \mapsto \Gamma\,[n^+] \end{pmatrix}, \begin{pmatrix} \mathcal{G}_{\mathcal{A},\mathcal{B},\mathcal{C}} \times \mathsf{Lab}_- \rightharpoonup \mathcal{B} \\ (\Gamma, n^-) \quad \mapsto \Gamma\,[n^-] \end{pmatrix}, \begin{pmatrix} \mathcal{G}_{\mathcal{A},\mathcal{B},\mathcal{C}} \to (\mathsf{Lab}_e \rightharpoonup \mathcal{C}) \\ \Gamma \quad \mapsto (n^e \mapsto \Gamma\,[n^e]) \end{pmatrix}\right)$$

into an $(\mathcal{A}, \mathcal{B}, \mathcal{C})$-context algebra, by defining the notion of extension as follows:

$$\begin{pmatrix} \mathcal{G}_{\mathcal{A},\mathcal{B},\mathcal{C}} \times \mathbb{D}_{\mathcal{A},\mathcal{B},\mathcal{C}} \to \mathcal{G}_{\mathcal{A},\mathcal{B},\mathcal{C}} \\ (\Gamma, \Delta) \quad \mapsto \Gamma; \Delta \end{pmatrix}$$

$$\begin{array}{llll} (\Gamma^+, \Gamma^-, \Gamma^e); a & := (a :: \Gamma^+, \Gamma^-, \Gamma^e) & (\Gamma^+, \Gamma^-, \Gamma^e); \sim b & := (\Gamma^+, b :: \Gamma^-, \Gamma^e) \\ (\Gamma^+, \Gamma^-, \Gamma^e); \bullet & := (\Gamma^+, \Gamma^-, \Gamma^e) & (\Gamma^+, \Gamma^-, \Gamma^e); (\Delta_1, \Delta_2) & := (\Gamma^+, \Gamma^-, \Gamma^e); \Delta_1; \Delta_2 \\ (\Gamma^+, \Gamma^-, \Gamma^e); c.\Delta & := (\Gamma^+, \Gamma^-, c :: \Gamma^e); \Delta & & \end{array}$$

※

We can now give the parameters that present LAF_{K1} as a LAF instance with eigenlabels:

DEFINITION 94 (LAF_{K1} as a LAF instance with eigenlabels)

$\mathbb{S}$ is the set of sorts as in Definition 81.

Lab_e is a copy of the set of natural numbers and $\mathbb{T}$ is the set of terms, as in Definition 82. Notice that $\mathbb{T}$ does extend Lab_e, and substitution of eigenlabels for terms gives a systematic way to lift a function $\mathsf{Lab}_e \to \mathsf{Lab}_e$ to a function $\mathbb{T} \to \mathbb{T}$.

$\Vdash$ is the sorting relation as in Definition 82. Notice that $\Sigma \Vdash x : s$ if and only if $s = \Sigma(x)$, and that for all $\pi : \mathsf{Lab}_e \to \mathsf{Lab}_e$, if $\Sigma \circ \pi \Vdash r : s$ then $\Sigma \Vdash \pi(r) : s$.

$\mathbb{A}$, $\mathbb{M}$ and $\equiv$ are as in Definition 81.

Lab_+ and Lab_- are as in Definition 83, and the context algebra Co is the instance $\mathcal{G}_{\mathbb{A}_\downarrow, \mathbb{M}_\downarrow, \mathbb{T}}$ of the generic family of contexts from Definition 93.

Given two functions $f_1 : \mathbb{A}_\downarrow \to \mathbb{A}_\downarrow$, $f_2 : \mathbb{M}_\downarrow \to \mathbb{M}_\downarrow$ and an $(\mathbb{A}_\downarrow, \mathbb{M}_\downarrow, \mathbb{T})$-context $(\Gamma^+, \Gamma^-, \Gamma^e)$, the result of the map operation $(f_1, f_2) \circ (\Gamma^+, \Gamma^-, \Gamma^e)$ is defined as $(f_1(\Gamma^+), f_2(\Gamma^-), \Gamma^e)$, where $f_1(\Gamma^+)$ and $f_2(\Gamma^-)$ are defined with the standard map operation on lists.

We define $\pi^{\mathcal{V}}_{\Delta}$ as the identity (note that we have $\Gamma\,[x] = (\Gamma;\Delta)\,[x]$).

We define $\mathsf{st}^{\mathcal{V}}_{\Delta}$ as the element $\mathsf{st}(\Delta, \mathsf{sup}(\mathcal{V}), (n, \Pi) \mapsto \Pi)$ of $\mathbb{D}_{\mathtt{unit},\mathtt{unit},\mathsf{Lab}_e}$, where $\mathsf{st}(\Delta, n, f)$ is defined in continuation-passing style[6] as follows:

$$\begin{array}{lcl}
\mathsf{st}((), n, f) & := & f(n, ()) \\
\mathsf{st}(\sim(), n, f) & := & f(n, \sim()) \\
\mathsf{st}(\bullet, n, f) & := & f(n, \bullet) \\
\mathsf{st}((().\Delta), n, f) & := & \mathsf{st}(\Delta, n+1, (n', \Pi) \mapsto f(n', (n+1)^e.\Pi)) \\
\mathsf{st}((\Delta_1, \Delta_2), n, f) & := & \mathsf{st}(\Delta_1, n, (n_1, \Pi_1) \mapsto \mathsf{st}(\Delta_2, n_1, (n_2, \Pi_2) \mapsto f(n_2, (\Pi_1, \Pi_2))))
\end{array}$$

Finally, Pat and $\Vdash$ are as in Definition 81. ※

The only subtle things in the above definition are that:

- we defined $\pi^{\mathcal{V}}_{\Delta}$ as the identity, since the use of De Bruijn's levels avoids the need to update labels with new names every time a context is extended;[7]
- we defined $\mathsf{st}^{\mathcal{V}}_{\Delta}$ as a data-structure that does nothing but remember the fresh eigenlabels that will be used for each construct $s.\Delta'$ within Δ.

From this alternative definition of LAF_{K1} we now have to describe how the original definition of LAF_{K1} can be recovered. More precisely, the context algebra $\mathcal{G}_{\mathbb{A}_\downarrow,\mathbb{M}_\downarrow,\mathbb{T}}$ of Definitions 93 and 94 yields the typing context algebra of Definition 83.

This is a particular case of a more general construction that turns every LAF instance with eigenlabels into a LAF instance, which we now present.

6.3.3 LAF instances with eigenlabels are LAF instances

We now show how a LAF instance with eigenlabels forms a LAF instance.

As expected, *sorting contexts* are now partial functions from eigenlabels to sorts (i.e. $\mathbb{C} = \mathsf{Lab}_e \rightharpoonup \mathbb{S}$).

What remains to do is to turn the $(\mathbb{A}_\downarrow, \mathbb{M}_\downarrow, \mathbb{S})$-context algebra Co into a proper typing context algebra in the sense of Definition 75. What is missing is the notion of extension:

$$\left(\begin{array}{l}
\mathsf{Co} \times \mathbb{D}_\downarrow \to \mathsf{Co} \\
(\Gamma, (\Delta^l, \mathbf{r})) \mapsto \Gamma;(\Delta^l, \mathbf{r})
\end{array}\right)$$

We define such an extension from the notion of extension that is available in the $(\mathbb{A}_\downarrow, \mathbb{M}_\downarrow, \mathbb{S})$-context algebra Co

$$\left(\begin{array}{ll}
\mathsf{Co} \times \mathbb{D}_{\mathbb{A}_\downarrow,\mathbb{M}_\downarrow,\mathbb{S}} \to \mathsf{Co} \\
\quad (\Gamma', \Delta') \quad \mapsto \Gamma';\Delta'
\end{array}\right)$$

and from the naming policies $(\mathcal{V}, \Delta) \mapsto \pi^{\mathcal{V}}_{\Delta}$ and $(\mathcal{V}, \Delta) \mapsto \mathsf{st}^{\mathcal{V}}_{\Delta}$.

More precisely, $\Gamma;(\Delta^l, \mathbf{r})$ is defined as $\Gamma';\Delta'$, where Γ' is an $(\mathbb{A}_\downarrow, \mathbb{M}_\downarrow, \mathbb{S})$-context and Δ' is an $(\mathbb{A}_\downarrow, \mathbb{M}_\downarrow, \mathbb{S})$-decomposition, obtained from Γ and $(\Delta^l, \mathbf{r})$ by using the two new functions. These functions allow us to describe the intricacies of the operation $\Gamma;(\Delta^l, \mathbf{r})$ that is completely unspecified in the abstract LAF system:

[6] i.e. for a continuation $f : (\mathbb{N} \times \mathbb{D}_{\mathtt{unit},\mathtt{unit},\mathsf{Lab}_e}) \to \mathbb{D}_{\mathtt{unit},\mathtt{unit},\mathsf{Lab}_e}$

[7] With De Bruijn's indices we would there have the opportunity to specify how the eigenlabels in a context Γ should be updated when Γ is extended into $\Gamma;\Delta$; namely, the indices should be raised by the number of new eigenlabels that Δ introduces.

- for instance, imagining that the eigenlabel x is mapped to s by Γ^e, we will need to know what happens to this mapping when Γ is extended into $\Gamma;(\Delta^l,\mathbf{r})$
- also, when Δ^l contains a typing decomposition of the form $s.\Delta^{s::l}$, we expect a new eigenlabel to be mapped to s and we will need to know which one it is.

The two naming policies provide this information:

- in the first example, the renaming policy $\pi^\Gamma_{\Delta^l}(x)$ provides the eigenlabel corresponding to x and mapped to s in the extended environment $\Gamma;(\Delta^l,\mathbf{r})$;[8]
- in the second example, the fresh naming policy $\mathsf{st}^\Gamma_{\Delta^l}$ provides the names of the newly introduced eigenlabels, placed in a $(\mathtt{unit},\mathtt{unit},\mathsf{Lab}_e)$-decomposition with the same structure as Δ^l; hence, it will contain a decomposition of the form $x.\Pi$ to indicate that x is the eigenlabel we are looking for (mapped to s in the extended environment).

Building the $(\mathbb{A}_\downarrow,\mathbb{M}_\downarrow,\mathbb{S})$-decomposition Δ' from $(\Delta^l,\mathbf{r})$ thus relies on the following instantiation mechanism:

DEFINITION 95 (Instantiation of typing decompositions)

The *instantiation* $\downarrow^\Pi_{\mathbf{r}}\ \Delta^l$ of a typing decomposition Δ^l is defined for a list of terms $\mathbf{r}$ of length $|l|$ and a $(\mathtt{unit},\mathtt{unit},\mathsf{Lab}_e)$-decomposition Π that has the same structure as Δ^l, as follows:

$$\begin{array}{llll}
\downarrow^{()}_{\mathbf{r}}\ a^l & := (a^l,\mathbf{r}) & \downarrow^{()}_{\mathbf{r}}\ (\sim M^l) & := \sim(M^l,\mathbf{r}) \\
\downarrow^{\bullet}_{\mathbf{r}}\ \bullet & := \bullet & \downarrow^{\Pi_1,\Pi_2}_{\mathbf{r}}\ (\Delta^l_1,\Delta^l_2) & := (\downarrow^{\Pi_1}_{\mathbf{r}}\ \Delta^l_1),(\downarrow^{\Pi_1}_{\mathbf{r}}\ \Delta^l_2) \\
\downarrow^{x.\Pi}_{\mathbf{r}}\ (s.\Delta^{s::l}) & := s.(\downarrow^{\Pi}_{x::\mathbf{r}}\ \Delta^{s::l}) & &
\end{array}$$

※

DEFINITION 96 (Typing contexts in the sense of LAF instances)

The $(\mathbb{A}_\downarrow,\mathbb{M}_\downarrow,\mathbb{S})$-context algebra Co of a LAF instance with eigenlabels, is turned into a typing context in the sense of LAF instances by defining the following extention operation:

Given a typing context Γ, a typing decomposition Δ^l of arity l and a list of terms $\mathbf{r}$ of length $|l|$, we define

$$\Gamma;(\Delta^l,\mathbf{r}) \ :=\ ((\pi^\Gamma_{\Delta^l},\pi^\Gamma_{\Delta^l})\circ\Gamma);(\downarrow^{\mathsf{st}^\Gamma_{\Delta^l}}_{\pi^\Gamma_{\Delta^l}(\mathbf{r})}\ \Delta^l)$$

※

The way we perform this extension can be explained as follows:

- first, the extension will rename the eigenlabels that were declared in Γ; these eigenlabels are mentioned in the parameters of the instantiated atoms and molecules in Γ, so we use the renaming policy $\pi^\Gamma_{\Delta^l}$ to update with the new names these instantiated atoms and molecules; the result is the context
$$\Gamma' := (\pi^\Gamma_{\Delta^l},\pi^\Gamma_{\Delta^l})\circ\Gamma$$
- second, we turn Δ^l into a $(\mathbb{A}_\downarrow,\mathbb{M}_\downarrow,\mathbb{S})$-decomposition as follows: the instantiated atoms and molecules at the leaves of this decomposition to produce will have their parameters based on $\mathbf{r}$; but the terms in $\mathbf{r}$ may mention the eigenlabels declared in Γ, which are now renamed, so we update $\mathbf{r}$ into $\pi^\Gamma_{\Delta^l}(\mathbf{r})$; then a parameterised atom a (resp. molecule M) at a leaf of Δ^l has an arity of the form $s_1::\ldots s_n::l$, and turns into the instantiated atom

[8] In other words, the eigenlabel x has been renamed $\pi^\Gamma_{\Delta^l}(x)$ in the extended environment; depending on how labels are implemented, it might be the case that x keeps its name and $\pi^\Gamma_{\Delta^l}$ is simply the identity.

$(a, x_1 :: \ldots x_n :: \pi^{\Gamma}_{\Delta^l}(\mathbf{r}))$ (resp. molecule $(M, x_1 :: \ldots x_n :: \pi^{\Gamma}_{\Delta^l}(\mathbf{r}))$), where $x_1, \ldots, x_n$ are new eigenlabels whose names we get from the fresh naming policy $\mathsf{st}^{\Gamma}_{\Delta^l}$; this results in the $(\mathbb{A}_{\downarrow}, \mathbb{M}_{\downarrow}, \mathbb{S})$-decomposition

$$\Delta' := \downarrow^{\mathsf{st}^{\Gamma}_{\Delta^l}}_{\pi^{\Gamma}_{\Delta^l}(\mathbf{r})} \Delta^l$$

- third, we extend Γ' with Δ'.

THEOREM 52 (The case of LAF_{K1}) The LAF instance LAF_{K1}, defined according to the above methodology from its definition as a LAF instance with eigenlabels (Definition 94), coincides with the direct definition of LAF_{K1} as a LAF instance (Sections 4.3.1 and 4.4.1). ※

Proof: Clearly we have

$$\Gamma; (\Delta^l, \mathbf{r}) \;=\; \Gamma; (\downarrow^{\mathsf{st}^{\Gamma}_{\Delta^l}}_{\mathbf{r}} \Delta^l)$$

with the left-hand side being defined in Definition 83 and the right-hand side being defined in Definitions 93 and 94. □

As we have seen, $\pi^{\mathcal{V}}_{\Delta}(x)$ and $\mathsf{st}^{\mathcal{V}}_{\Delta}(x)$ form a naming policy for the eigenlabels used after a (typing) context extension. More generally, the fact that an $(\mathcal{A}, \mathcal{B}, \mathcal{C})$-context algebra respects this naming policy can be expressed as follows:

DEFINITION 97 (Respecting naming policies) An $(\mathcal{A}, \mathcal{B}, \mathcal{C})$-context algebra $\mathcal{G}$ *respects the naming policies* $(\mathcal{V}, \Delta) \mapsto \pi^{\mathcal{V}}_{\Delta}$ and $(\mathcal{V}, \Delta) \mapsto \mathsf{st}^{\mathcal{V}}_{\Delta}$ if for all ρ and v we have

1. $\rho\,[x] = (\rho; v)\left[\pi^{\mathsf{dom}(\rho)}_{|v|}(x)\right]$ for all eigenlabel $x \in \mathsf{dom}^e(\rho)$;
2. and $\mathsf{st}^{\mathsf{dom}(\rho)}_{|v|}$ relates to v according to $(\mathtt{unit} \times \mathbb{A}_{\downarrow})$, $(\mathtt{unit} \times \mathbb{M}_{\downarrow})$, and $\{(x, (\rho; v)\,[x]) \mid x \in \mathsf{dom}^e(\rho; v)\}$.

※

6.4 A more concrete class of realisability algebras

Now the whole point of introducing the subclass of LAF instances that we call "with eigenlabels", is to have a tighter Adequacy Lemma that relies on a weaker (and more systematically derivable) correlation property than typing correlation.

For this we identify a class of realisability algebras that naturally form models for LAF instances with eigenlabels.

In brief, a realisability algebra with eigenlabels is a realisability algebra where valuations are functions mapping eigenlabels to term denotations.

Assume we have a LAF instance with eigenlabels

$$(\mathbb{S}, \mathsf{Lab}_e, \mathbb{T}, \Vdash, \mathbb{A}, \mathbb{M}, \equiv, \mathsf{Lab}_+, \mathsf{Lab}_-, \mathsf{Co}, \mathsf{Pat}, \vdash, \pi^{\mathcal{V}}_{\Delta}, \mathsf{st}^{\mathcal{V}}_{\Delta})$$

DEFINITION 98 (Realisability algebras with eigenlabels)

A *model structure with eigenlabels* is a model structure where $\mathscr{C} = \mathsf{Lab}_e \rightharpoonup \mathscr{T}$, satisfying

- for all $x \in \mathsf{Lab}_e$, $\sigma : \mathsf{Lab}_e \rightharpoonup \mathscr{T}$, we have $[\![x]\!]_{\sigma} = \sigma(x)$;
- for all $r \in \mathbb{T}$, $\sigma : \mathsf{Lab}_e \rightharpoonup \mathscr{T}$ and $\pi : \mathsf{Lab}_e \rightharpoonup \mathsf{Lab}_e$, we have $[\![r]\!]_{\sigma\circ\pi} = [\![\pi(r)]\!]_{\sigma}$;

and where the semantic context algebra respects the naming policies.

A *realisability algebra with eigenlabels* is a realisability algebra whose model structure is a model structure with eigenlabels and where, for all $\Sigma : \mathsf{Lab}_e \rightharpoonup \mathbb{S}$ and $\sigma : \mathsf{Lab}_e \rightharpoonup \mathscr{T}$,

$\sigma \in [\![\Sigma]\!]$ if and only if for all $x \in \mathsf{Lab}_e$ we have $\sigma(x) \in [\![\Sigma(x)]\!]$ (and $\mathsf{Dom}(\sigma) = \mathsf{Dom}(\Sigma)$).

※

DEFINITION 99 (Generic correlation)

Given three relations $\mathcal{R}_1 \subseteq \mathcal{A} \times \mathcal{A}'$, $\mathcal{R}_2 \subseteq \mathcal{B} \times \mathcal{B}'$ and $\mathcal{R}_3 \subseteq \mathcal{C} \times \mathcal{C}'$,
we say that an $(\mathcal{A}, \mathcal{B}, \mathcal{C})$-context algebra $\mathcal{G}$ and an $(\mathcal{A}', \mathcal{B}', \mathcal{C}')$-context algebra $\mathcal{G}'$ satisfy the *correlation property* for $\mathcal{R}_1$, $\mathcal{R}_2$ and $\mathcal{R}_3$ if the following holds:

For all $\Gamma \in \mathcal{G}$, $\Gamma' \in \mathcal{G}'$, $\Delta \in \mathbb{D}_{\mathcal{A},\mathcal{B},\mathcal{C}}$ and $\Delta' \in \mathbb{D}_{\mathcal{A}',\mathcal{B}',\mathcal{C}'}$
if Γ relates to Γ' according to $\mathcal{R}_1$, $\mathcal{R}_2$ and $\mathcal{R}_3$
and Δ relates to Δ' according to $\mathcal{R}_1$, $\mathcal{R}_2$ and $\mathcal{R}_3$
then $\Gamma; \Delta$ relates to $\Gamma'; \Delta'$ according to $\mathcal{R}_1$, $\mathcal{R}_2$ and $\mathcal{R}_3$;

※

DEFINITION 100 (Correlation with eigenlabels) Given a realisability algebra with eigenlabels (for our LAF instance with eigenlabels), we define three relations[9]

$$\begin{array}{lll} \mathcal{R}_1^\sigma & := & \{(\mathfrak{l}, (a, \mathbf{r})) \mid \mathfrak{l} \in [\![(a, \mathbf{r})]\!]_\sigma\} \quad \subseteq \mathscr{L} \times \mathbb{A}_\downarrow \\ \mathcal{R}_2^\sigma & := & \{(\mathfrak{n}, (M, \mathbf{r})) \mid \mathfrak{n} \in [\![(M, \mathbf{r})]\!]_\sigma\} \quad \subseteq \mathscr{N} \times \mathbb{M}_\downarrow \\ \mathcal{R}_3 & := & \{(\mathfrak{r}, s) \mid \mathfrak{r} \in [\![s]\!]\} \quad \subseteq \mathscr{T} \times \mathbb{S} \end{array}$$

for any given $\sigma : \mathsf{Lab}_e \rightharpoonup \mathscr{T}$.

We say that $\tilde{\mathsf{Co}}$ and Co satisfy the *correlation with eigenlabels* property if for all $\sigma : \mathsf{Lab}_e \rightharpoonup \mathscr{T}$, they satisfy the correlation property for $\mathcal{R}_1^\sigma$, $\mathcal{R}_2^\sigma$ and $\mathcal{R}_3$. ※

REMARK 53 $\rho \in [\![\Gamma]\!]$ if and only if ρ relates to Γ according to $\mathcal{R}_1^{\rho^e}$, $\mathcal{R}_2^{\rho^e}$ and $\mathcal{R}_3$. ※

LEMMA 54 (Correlation with eigenlabels implies typing correlation)

If $\tilde{\mathsf{Co}}$ and Co satisfy the correlation with eigenlabels property, then they satisfy the typing correlation property: if $\rho \in [\![\Gamma]\!]$ and $\mathfrak{d} \in \left[\!\left[(\Delta^l, \mathbf{r})\right]\!\right]_{\rho^e}$ then $(\rho; \mathfrak{d}) \in \left[\!\left[\Gamma; (\Delta^l, \mathbf{r})\right]\!\right]$. ※

Proof: See the proof in Coq [GL14]. The main lines are as follows:

From $\rho \in [\![\Gamma]\!]$ we get that ρ relates to Γ according to $\mathcal{R}_1^{\rho^e}$, $\mathcal{R}_2^{\rho^e}$ and $\mathcal{R}_3$.

Then ρ relates to $(\pi^\Gamma_{\Delta^l}, \pi^\Gamma_{\Delta^l}) \circ \Gamma$ according to $\mathcal{R}_1^{(\rho;\mathfrak{d})^e}$, $\mathcal{R}_2^{(\rho;\mathfrak{d})^e}$ and $\mathcal{R}_3$.

From $\mathfrak{d} \in \left[\!\left[(\Delta^l, \mathbf{r})\right]\!\right]_{\rho^e}$ we get that $\mathfrak{d}$ relates to $\downarrow^{\mathsf{st}^\Gamma_{\Delta^l}}_{\pi^\Gamma_{\Delta^l}(\mathbf{r})} \Delta^l$ according to $\mathcal{R}_1^{(\rho;\mathfrak{d})^e}$, $\mathcal{R}_2^{(\rho;\mathfrak{d})^e}$ and $\mathcal{R}_3$.

Then correlation with eigenlabels provides that $\rho; \mathfrak{d}$ relates to $\Gamma; (\Delta^l, \mathbf{r})$ according to $\mathcal{R}_1^{(\rho;\mathfrak{d})^e}$, $\mathcal{R}_2^{(\rho;\mathfrak{d})^e}$ and $\mathcal{R}_3$, which means that $(\rho; \mathfrak{d}) \in \left[\!\left[\Gamma; (\Delta^l, \mathbf{r})\right]\!\right]$.

At some point in the above proof we use the fact that the semantic context algebra respects the naming policies. □

[9] In this definition we implicitly require $[\![(a, \mathbf{r})]\!]_\sigma$ and $[\![(M, \mathbf{r})]\!]_\sigma$ to be defined.

Lemma 55 (Adequacy for LAF with eigenlabels)

We assume the following hypotheses:

Well-foundedness:

The LAF instance with eigenlabels is well-founded.

Correlation with eigenlabels:

$\tilde{\mathsf{Co}}$ and Co satisfy the correlation with eigenlabels property.

Stability:

If $\mathfrak{d} \in \left[\!\left[(\Delta^l, \mathbf{r})\right]\!\right]_\sigma$ for some $\Delta^l, \sigma, \mathbf{r}$ and $\left[\!\left[f(p)\right]\!\right]_{\rho;\mathfrak{d}} \in \bot$, then $\left[\!\left[f\right]\!\right]_\rho \perp \tilde{p}(\mathfrak{d})$.

We conclude that, for all $\rho \in \left[\!\left[\Gamma\right]\!\right]$,

1. if $\Gamma \vdash [t^+ : (M^l, \mathbf{r})]$ then $\left[\!\left[t^+\right]\!\right]_\rho \in \left[\!\left[(M^l, \mathbf{r})\right]\!\right]^+$;
2. if $\Gamma \vdash d : (\Delta^l, \mathbf{r})$ then $\left[\!\left[d\right]\!\right]_\rho \in \left[\!\left[(\Delta^l, \mathbf{r})\right]\!\right]$;
3. if $\Gamma \vdash t$ then $\left[\!\left[t\right]\!\right]_\rho \in \bot$

※

Proof: Corollary of Lemmata 51 and 54. □

This Adequacy Lemma looks similar to Lemma 51, but the correlation assumption is much "weaker": all the job is done in the extra structure with eigenlabels that we have required from terms, sorting contexts, typing contexts and valuations (and the finer-grained specifications we have imposed on them).

Indeed, correlation with eigenlabels often holds as a particular case of the more general correlation property for all relations $\mathcal{R}_1$, $\mathcal{R}_2$, $\mathcal{R}_3$, typically when Co and $\tilde{\mathsf{Co}}$ are respectively defined as the two instances $\mathcal{G}_{\mathbb{A}_\downarrow,\mathbb{M}_\downarrow,\mathbb{T}}$ and $\mathcal{G}_{\mathscr{L},\mathscr{N},\mathscr{T}}$ of a generic family $(\mathcal{G}_{\mathcal{A},\mathcal{B},\mathcal{C}})_{\mathcal{A},\mathcal{B},\mathcal{C}}$ of $(\mathcal{A},\mathcal{B},\mathcal{C})$-context algebras whose definition is "sufficiently parametric". In particular for LAF_{K1}:

Remark 56 Generic correlation always holds for the family $(\mathcal{G}_{\mathcal{A},\mathcal{B},\mathcal{C}})_{\mathcal{A},\mathcal{B},\mathcal{C}}$ of $(\mathcal{A},\mathcal{B},\mathcal{C})$-context algebras defined for LAF_{K1} (Definition 93).

In particular, correlation with eigenlabels holds for that system and for any of its realisability algebras where $\tilde{\mathsf{Co}} = \mathcal{G}_{\mathscr{L},\mathscr{N},\mathscr{T}}$. If stability also holds for that instance and that realisability algebra, then the conclusions of the Adequacy Lemma hold. ※

The same remark would hold of any LAF instance and any realisability algebra where Co and $\tilde{\mathsf{Co}}$ are defined from a similarly parametric family of context algebras.

6.5 Example: boolean models to prove Consistency

We now exhibit models to prove the consistency of LAF systems.

Assume we have a LAF instance with eigenlabels

$$(\mathbb{S}, \mathsf{Lab}_e, \mathbb{T}, \Vdash, \mathbb{A}, \mathbb{M}, \equiv, \mathsf{Lab}_+, \mathsf{Lab}_-, \mathsf{Co}, \mathsf{Pat}, \vdash)$$

Definition 101 (Boolean realisability algebras)

A *boolean realisability algebra* is a realisability algebra where $\bot = \emptyset$. ※

The terminology comes from the remark that in a boolean realisability algebra, $\left[\!\left[(M^l, \mathbf{r})\right]\!\right]^-_\sigma$ can only take one of two values: $\emptyset$ or $\mathscr{N}$, depending on whether $\left[\!\left[(M^l, \mathbf{r})\right]\!\right]^+_\sigma$ is empty or not.

Remark 57 A boolean realisability algebra satisfies Stability. ※

THEOREM 58 (Consistency of LAF instances with eigenvariables)

Assume the LAF instance with eigenlabel is well-founded and assume that we have a boolean realisability algebra with eigenlabels where

- there is an empty semantic context $\rho_\emptyset$;
- $\tilde{\mathsf{Co}}$ and Co satisfy the correlation with eigenlabels property.

Then there is no empty typing context $\Gamma_\emptyset$ and command t such that $\Gamma_\emptyset \vdash t$. ※

Proof: The previous remark provides Stability. If there was such a $\Gamma_\emptyset$ and t, then we would have $\rho_\emptyset \in [\![\Gamma_\emptyset]\!]$, and the Adequacy Lemma (Lemma 55) would conclude $[\![t]\!]_{\rho_\emptyset} \in \emptyset$. □

We provide such a realisability model that works with all parametric LAF instances with eigenlabels:

DEFINITION 102 (Trivial model for parametric LAF instances with eigenlabels)

Assume that Co is the instance $\mathcal{G}_{\mathbb{A}_\downarrow,\mathbb{M}_\downarrow,\mathbb{T}}$ of a family of context algebras $(\mathcal{G}_{\mathcal{A},\mathcal{B},\mathcal{C}})_{\mathcal{A},\mathcal{B},\mathcal{C}}$.

The *trivial boolean model* for it is:

$$\begin{array}{lll}
\mathscr{T} := \mathscr{L} := \mathscr{P} := \mathscr{N} := \mathtt{unit} & & \\
\bot := \emptyset & & \\
\tilde{\mathsf{Co}} := \mathcal{G}_{\mathtt{unit},\mathtt{unit},\mathtt{unit}} & & \\
\text{and therefore} & & \\
\quad \forall \rho \in \tilde{\mathsf{Co}}, \forall x^+ \in \mathsf{dom}^+(\rho), & \rho\,[x^+] & := \ () \\
\quad \forall \rho \in \tilde{\mathsf{Co}}, \forall x^- \in \mathsf{dom}^-(\rho), & \rho\,[x^-] & := \ () \\
\quad \forall \rho \in \tilde{\mathsf{Co}}, \forall x \in \mathsf{dom}^e(\rho), & \rho\,[x] & := \ () \\
\forall \mathfrak{d} \in \tilde{\mathbb{D}}, & \tilde{p}(\mathfrak{d}) & := \ () \\
\forall r \in \mathbb{T}, \forall \sigma \in \mathscr{C}, & [\![r]\!]_\sigma & := \ () \\
\forall f : \mathsf{Pat} \rightharpoonup \mathsf{Terms}, \forall \rho \in \tilde{\mathsf{Co}}, & [\![f]\!]_\rho & := \ () \\
 & & \\
\forall s \in \mathbb{S}, & [\![s]\!] & := \ \mathtt{unit} \\
\forall a^l \in \mathbb{A}_l, \forall \mathfrak{rl} \in \mathscr{T}^l, & [\![a^l]\!]\,(\mathfrak{rl}) & := \ \mathtt{unit}
\end{array}$$

※

It is straightforward to check that the above definition does satisfy the specification of a realisability algebra with eigenlabels.

Note that, not only can $[\![(M^l, \mathbf{r})]\!]^-_\sigma$ only take one of the two values $\emptyset$ or $\mathtt{unit}$, but $[\![(M^l, \mathbf{r})]\!]^+_\sigma$ can also only take one of the two values $\emptyset$ or $\mathtt{unit}$.

We can now use such a structure to derive consistency for a large class of systems:

COROLLARY 59 (Consistency for parametric LAF instances with eigenlabels)

Assume that the LAF instance with eigenlabels is well-founded and that

- Co is the instance $\mathcal{G}_{\mathbb{A}_\downarrow,\mathbb{M}_\downarrow,\mathbb{T}}$ of a family of context algebras $(\mathcal{G}_{\mathcal{A},\mathcal{B},\mathcal{C}})_{\mathcal{A},\mathcal{B},\mathcal{C}}$,
- Any two context algebras of the family $(\mathcal{G}_{\mathcal{A},\mathcal{B},\mathcal{C}})_{\mathcal{A},\mathcal{B},\mathcal{C}}$ satisfy the correlation property for all $\mathcal{R}_1$, $\mathcal{R}_2$, $\mathcal{R}_3$.
- There is an empty $(\mathtt{unit}, \mathtt{unit}, \mathtt{unit})$-context in $\mathcal{G}_{\mathtt{unit},\mathtt{unit},\mathtt{unit}}$,

Then there is no empty typing context $\Gamma_\emptyset$ and command t such that $\Gamma_\emptyset \vdash t$.

In particular, this is the case for LAF_{K1}. ※

The system LAF_J does not fall in the above category since the operation of context extension is not parametric enough: when computing $\Gamma;(a^l,\mathbf{r})$ (resp. $\Gamma;(\sim M^l,\mathbf{r})$), we have to make a case analysis on whether a^l is of the form (l^+,r) or (v,l) (resp. whether M^l is of the form (N,l) or (P,r)).

But we can easily adapt the above trivial model into a not-as-trivial-but-almost model:

DEFINITION 103 (Trivial model for LAF_J) The *trivial boolean model for LAF_J* is:

$\mathscr{T} := \mathscr{P} := \mathtt{unit}$
$\mathscr{L} := \mathscr{N} := \{\mathsf{l},\mathsf{r}\}$
$\bot := \emptyset$
$\tilde{\mathsf{Co}}$ has semantics contexts of the form (m^+,m^-,m^e,R),
where $m^+,m^-,m^e \in \mathbb{N}$ and $R \in \{0,1\}$
and an extension operation defined as follows

$$\begin{array}{llll}
(m^+,m^-,m^e,R);\mathsf{r} & := (m^+ + 1,m^-,R) & (m^+,m^-,m^e,R);\mathsf{l} & := (m^+,m^-,m^e,0)\\
(m^+,m^-,m^e,R);\sim\mathsf{l} & := (m^+,m^- + 1,R) & (m^+,m^-,m^e,R);\sim\mathsf{r} & := (m^+,m^-,m^e,1)\\
(m^+,m^-,m^e,R);\bullet & := (m^+,m^-,m^e,R) & & \\
(m^+,m^-,m^e,R);().\Delta & := (m^+,m^-,m^e + 1,R) & & \\
(m^+,m^-,m^e,R);(\Delta_1,\Delta_2) & := (m^+,m^-,m^e,R);\Delta_1;\Delta_2 & &
\end{array}$$

and we define

$$\begin{array}{llll}
(m^+,m^-,m^e,R)\,[n^+] & := \mathsf{r} \text{ if } n^+ < m^+ & (m^+,m^-,m^e,R)\,[n^-] & := \mathsf{l} \text{ if } n^- < m^-\\
(m^+,m^-,m^e,R)\,[n^+] & \text{undefined otherwise} & (m^+,m^-,m^e,R)\,[n^-] & \text{undefined otherwise}\\
(m^+,m^-,m^e,0)\,[\star^+] & := \mathsf{l} & (m^+,m^-,m^e,0)\,[\star^-] & \text{undefined}\\
(m^+,m^-,m^e,1)\,[\star^+] & \text{undefined} & (m^+,m^-,m^e,1)\,[\star^-] & := \mathsf{r}\\
(m^+,m^-,m^e,R)\left[x^+_{(\bot^-,\mathsf{l})}\right] & := \mathsf{l} & &
\end{array}$$

$$\begin{array}{ll}
(m^+,m^-,m^e,R)\,[n^e] & := () \text{ if } n^e < m^e\\
(m^+,m^-,m^e,R)\,[n^e] & \text{undefined otherwise}
\end{array}$$

$$\begin{array}{lll}
\forall \mathfrak{d} \in \tilde{\mathbb{D}}, & \tilde{p}(\mathfrak{d}) & := ()\\
\forall r \in \mathbb{T}, \forall \sigma \in \mathscr{C}, & [\![r]\!]_\sigma & := ()\\
\forall f : \mathsf{Pat} \rightharpoonup \mathsf{Terms}, \forall \rho \in \tilde{\mathsf{Co}}, & [\![f]\!]_\rho & := \mathsf{l} \text{ if every } p \in \mathsf{Dom}(f) \text{ is of the form}\\
 & & \qquad _^+_{\mathsf{r}} \mid _^-_{\mathsf{r}} \mid \bullet_{\mathsf{r}} \mid (p_1,p_2) \mid \mathsf{inj}_i(p)\\
 & & := \mathsf{r} \text{ if not}
\end{array}$$

$$\begin{array}{lll}
\forall s \in \mathbb{S}, & [\![s]\!] & := \mathtt{unit}\\
\forall (l^+,\mathsf{r}) \in \mathbb{A}_l, \forall \mathfrak{r}\mathfrak{l} \in \mathscr{T}^l, & [\![(l^+,\mathsf{r})]\!]\,(\mathfrak{r}\mathfrak{l}) & := \{\mathsf{r}\}\\
\forall (v,\mathsf{l}) \in \mathbb{A}_l, \forall \mathfrak{r}\mathfrak{l} \in \mathscr{T}^l, & [\![(v,\mathsf{l})]\!]\,(\mathfrak{r}\mathfrak{l}) & := \{\mathsf{l}\}
\end{array}$$

※

It is straightforward to check that the above definition does satisfy the specification of a realisability algebra with eigenlabels. Moreover, Co and $\tilde{\mathsf{Co}}$ satisfy the correlation property with eigenlabels.

We can now use such a structure to derive consistency for LAF_J:

THEOREM 60 (Consistency of LAF_J)

There is no command t such that $(\Gamma^+,[\,],\Gamma^e,(v,\mathsf{l},\mathbf{r})) \vdash t$ in LAF_J. ※

Proof: Take the trivial boolean model for LAF_J; we have Stability. Take $\rho := (|\Gamma^+|,0,|\Gamma^e|,\mathsf{l});$

clearly $\rho \in [\![(\Gamma^+, [\,], \Gamma^e, (v, \mathsf{l}, \mathbf{r}))]\!]$, and the Adequacy Lemma (Lemma 55) would conclude $[\![t]\!]_\rho \in \emptyset$. □

Chapter 7

Transforming proofs in the abstract focussed sequent calculus

Contents

In this chapter we investigate how LAF proofs can be transformed.

First and foremost, we have in mind the key process of structural proof theory: *cut-elimination*. The process is all the more interesting as it relates, through the Curry-Howard correspondence [How80], to the paradigm of computation in functional programming; and in the case of LAF systems, cut-elimination strongly relates to the very concept of pattern-matching, following [Zei09].

Let us also remember that originally, admissibility of cuts was a property used by Gentzen [Gen35] to relate the sequent calculus with cuts, which can easily be proved complete, to the cut-free sequent calculus, which is easily proved consistent. Even though we already have consistency results for LAF systems (with cuts) obtained by semantical methods (see Section 6.5), we are still interested in cut admissibility to get completeness of cut-free LAF systems. Indeed, we identifiedde LAF systems with the perspective of using them as the basis of proof-search implementations, and knowing this property will help organising the exploration of the search-space.

The prospect of implementing proof-search also motivates the study of another kind of proof-transformation: As we shall seek to memoise the proof-search process (tabling all the proofs and sub-proofs we complete to re-use them as often as possible), we will often seek to *adapt* a previously obtained proof to a new sequent to be proved (provided of course this new sequent contains all the necessary ingredients for the proof to be replayed).

In Section 7.1, we identify a notion of *abstract machine* to reduce the proof-terms of LAF, implementing in effect a notion of *head reduction*. In Section 7.2 we prove that this reduction terminates on typed terms, for which the realisability models of Chapter 6 will play

a key role. In Section 7.3 we investigate the re-usability of proofs, by identifying with the concept of *free label* the atoms and molecules of a proved sequent that are necessary for the proof to be replayed on another sequent to prove. In Section 7.4, we will investigate how the transformations explored in the previous sections can be used to prove cut-elimination in LAF. In Section 7.5 we discuss the possibility of more general notions of reduction and the issue of Strong Normalisation.

The proof transformations explored in this chapter will prove particularly useful when using LAF in automated reasoning (see the third part of this dissertation).

7.1 Head reduction

It is natural to want to reduce $\langle f \mid pd \rangle$ to $f(p)$ "substituted by d". Indeed, this would be the evaluation rule of pattern-matching: we can think of p as a pattern and d as a way to fill its holes, while f is a pattern-matching function; the rule then selects the branch of f corresponding to p and depending on the pattern's holes, and computation continues with the code in that branch where the holes have been substituted according to d.

Such a notion of substitution, however, is not yet defined. And so far d is a decomposition term: we can easily imagine using it to extend a context, but it is not a context itself.

Now following the view that "there is no such thing as a free variable" (what is thought of as free in in fact bound somewhere else), we can accept that reducing $\langle f \mid pd \rangle$ is in fact done in a context ρ that assigns "values" to the "free labels" of f and d. This view is actually quite natural when thinking of evaluating programs by an abstract machine: evaluation is performed within an "environment" that maps variables to values such as *closures*.

In the case of LAF, this view helps understanding how the term decomposition d can be involved in reductions, as it can now be used to extend the local context ρ in which the command is evaluated:

$$\langle\!\langle\ \langle f \mid pd \rangle\ \mid\ \rho\ \rangle\!\rangle \longrightarrow \langle\!\langle\ f(p)\ \mid\ \rho; d'\ \rangle\!\rangle$$

where d' is "d in the context ρ". This we could think as simply the pairing (d, ρ), were it not for the fact that the extension $\rho; d'$ needs d' to be a decomposition, not a pair. Hence, d' will rather be the distribution of ρ down to each leaf of d.

This is formalised as follows:

DEFINITION 104 (Abstract machine for LAF)

Assume we have four sets $\mathbb{V}_+$, $\mathbb{V}_-$, $\mathbb{T}$, $\mathbb{S}$, and a $(\mathbb{V}_+, \mathbb{V}_-, \mathbb{T}, \mathbb{S})$-context algebra with support set $\mathcal{G}$ such that the set $\mathbb{C} := (\mathsf{Pat} \rightharpoonup \mathsf{Terms}) \times \mathcal{G}$ is a subset of $\mathbb{V}_-$.

Elements of $\mathbb{C}$ are called *closures* and denoted $\langle\!\langle f \mid \rho \rangle\!\rangle$ (where $f : \mathsf{Pat} \rightharpoonup \mathsf{Terms}$ and $\rho \in \mathcal{G}$), while elements of $\mathcal{G}$ are called *evaluation contexts*.

An *evaluation decomposition* is a $(\mathbb{V}_+, \mathbb{V}_-, \mathbb{T})$-decomposition.

An *evaluation triple* is a triple denoted $\langle v \mid pd \rangle$ (overloading the notation for commands) where $v \in \mathbb{V}_-$, $p \in \mathsf{Pat}$ and d is an evaluation decomposition.

A *contextualised command* is a pair denoted $\langle\!\langle\ t\ \mid\ \rho\ \rangle\!\rangle$ where t is a command and ρ is an evaluation context.

We assume we have an *instantiation function*

$$\begin{pmatrix} \mathbb{T} \times \mathbb{S} \rightharpoonup \mathbb{T} \\ (r,\sigma) \mapsto \langle\!\langle r \rangle\!\rangle_\sigma \end{pmatrix}$$

We define the *distribution of an evaluation context ρ over a term decomposition d*, denoted $\langle\!\langle d \rangle\!\rangle_\rho$, as the following evaluation decomposition:

$$\begin{array}{lll} \langle\!\langle x^+ \rangle\!\rangle_\rho & := & \rho\,[x^+] \\ \langle\!\langle f \rangle\!\rangle_\rho & := & \langle\!\langle f \mid \rho \rangle\!\rangle \\ \langle\!\langle \bullet \rangle\!\rangle_\rho & := & \bullet \\ \langle\!\langle d_1, d_2 \rangle\!\rangle_\rho & := & \langle\!\langle d_1 \rangle\!\rangle_\rho, \langle\!\langle d_2 \rangle\!\rangle_\rho \\ \langle\!\langle r.d \rangle\!\rangle_\rho & := & \langle\!\langle r \rangle\!\rangle_{\rho^e} . \langle\!\langle d \rangle\!\rangle_\rho \end{array}$$

The reduction relation is defined in two steps: the reduction of a contextualised command to an evaluation triple, and the reduction of an evaluation triple to a contextualised command:

$$\begin{array}{lll} (\mathsf{head}_1) & \langle\!\langle\, \langle x^- \mid pd \rangle \mid \rho \,\rangle\!\rangle & \longrightarrow \langle \rho\,[x^-] \mid p \langle\!\langle d \rangle\!\rangle_\rho \rangle \\ (\mathsf{head}_2) & \langle\!\langle\, \langle f \mid pd \rangle \mid \rho \,\rangle\!\rangle & \longrightarrow \langle \langle\!\langle f \mid \rho \rangle\!\rangle \mid p \langle\!\langle d \rangle\!\rangle_\rho \rangle \\ \\ (\mathsf{head}_3) & \langle\!\langle f \mid \rho \rangle\!\rangle \mid pd \rangle & \longrightarrow \langle\!\langle\, f(p) \mid \rho; d \,\rangle\!\rangle \end{array}$$

We will write $\longrightarrow^*_{\mathsf{head}_{123}}$ for $\longrightarrow^*_{\mathsf{head}_1,\mathsf{head}_2,\mathsf{head}_3}$, which will always be an alternation of $\longrightarrow_{\mathsf{head}_1,\mathsf{head}_2}$ and $\longrightarrow_{\mathsf{head}_3}$.

There are no contextualised commands in normal form and evaluation triples in normal form are those of the form $\langle x^- \mid pd \rangle$.

If a contextualised command or an evaluation triple reduces by $\longrightarrow^*_{\mathsf{head}_{123}}$ to such a normal form, we say that it *head-normalises*. ※

EXAMPLE 11 (Syntactic abstract machine)

Standard examples of abstract machine are *syntactic abstract machines*, where $\mathbb{V}_+ := \mathsf{Lab}_+$ and $\mathbb{T} := \mathbb{T}$, and $\mathbb{V}_- = \mathsf{Lab}_- \cup \mathbb{C}$. In other words, computation can substitute positive labels for positive labels, and substitute either negative labels or closures for negative labels.

Note however that this makes $\mathbb{V}_-$ and $\mathbb{C}$ mutually dependent,[1] so their exact definition can hardly be defined at this abstract level.

But for instance with LAF_{K1}, we can adapt Definition 93 to define $\mathbb{C}$, $\mathbb{V}_-$ and the set $\mathcal{G}$ of evaluation contexts by simultaneous induction:

- $\mathbb{C} := (\mathsf{Pat} \rightharpoonup \mathsf{Terms}) \times \mathcal{G}$
- $\mathbb{V}_- := \mathsf{Lab}_- \cup \mathbb{C}$
- $\mathcal{G}$ is the set of elements of the form $(\Gamma^+, \Gamma^-, \Gamma^e)$ where Γ^+ (resp. Γ^-, Γ^e) is a list of elements of Lab_+ (resp. $\mathbb{V}_-$, $\mathbb{T}$).

Once the set $\mathcal{G}$ of evaluation contexts is defined, the full evaluation context algebra is simply $\mathcal{G}_{\mathsf{Lab}_+,\mathbb{V}_-,\mathbb{T}}$ (using the notation of Definition 93).

[1] Remember that $\mathbb{C}$ is $(\mathsf{Pat} \rightharpoonup \mathsf{Terms}) \times \mathcal{G}$, where $\mathcal{G}$ is (the support set of) a $(\mathbb{V}_+, \mathbb{V}_-, \mathbb{T}, \mathbb{S})$-context algebra.

Similarly, the set $\mathbb{S}$ and the function

$$\begin{pmatrix} \mathbb{T} \times \mathbb{S} \rightharpoonup \mathbb{T} \\ (r, \sigma) \mapsto \langle\!\langle r \rangle\!\rangle_\sigma \end{pmatrix}$$

can hardly be defined at the abstract level. But for first-order logic it is natural to define $\mathbb{S}$ as the set $\mathsf{Lab}_e \to \mathbb{T}$ of substitutions, and $\langle\!\langle r \rangle\!\rangle_\sigma$ is simply the application of substitution σ to the first-order term r. ※

7.2 Head normalisation

In this section we show that the abstract machine from Definition 104 terminates, when starting from typed proof-terms.

Mimicking the use of orthogonality models to prove strong normalisation result as in Chapter 2, we prove normalisation of the abstract machine by the use of a realisability model, in the sense of Chapter 6.

DEFINITION 105 (A realisability model for head-normalisation)

Assume we have an abstract machine defined by four sets $\mathbb{V}_+$, $\mathbb{V}_-$, $\mathbb{T}$, $\mathbb{S}$, an evaluation context algebra $\mathcal{G}$, and an instantiation function $(r, \sigma) \mapsto \langle\!\langle r \rangle\!\rangle_\sigma$.

The *head-normalisation model* for this abstract machine is

$$\begin{array}{lll}
\mathscr{C} & := & \mathbb{S} \\
\mathscr{T} & := & \mathbb{T} \\
\mathscr{L} & := & \mathbb{V}_+ \\
\mathscr{P} & := & \mathsf{Pat} \times \mathbb{D}_{\mathbb{V}_+, \mathbb{V}_-, \mathbb{T}} \\
\mathscr{N} & := & \mathbb{V}_- \\
v \perp pd & \text{if the evaluation triple } \langle v \mid pd \rangle \text{ head-normalises}^2 & \\
\tilde{\mathsf{Co}} & := & \mathcal{G} \\
\forall \mathfrak{d} \in \tilde{\mathbb{D}}, & \tilde{p}(\mathfrak{d}) & := p\mathfrak{d} \\
\forall r \in \mathbb{T}, \forall \sigma \in \mathscr{C}, & [\![r]\!]_\sigma & := \langle\!\langle r \rangle\!\rangle_\sigma \\
\forall f : \mathsf{Pat} \rightharpoonup \mathsf{Terms}, \forall \rho \in \tilde{\mathsf{Co}}, & [\![f]\!]_\rho & := \langle\!\langle f \mid \rho \rangle\!\rangle \\
\forall s \in \mathbb{S}, & [\![s]\!] & := \mathbb{T} \\
\forall \Sigma \in \mathbb{C}, & [\![\Sigma]\!] & := \mathbb{S} \\
\forall a^l \in \mathbb{A}_l, \forall \mathfrak{tl} \in \mathscr{T}^l, & [\![a^l]\!](\mathfrak{tl}) & := \mathbb{V}_+
\end{array}$$

※

REMARK 61 Notice that $\langle\!\langle\ t \mid \rho\ \rangle\!\rangle \longrightarrow_{\mathsf{head}_1, \mathsf{head}_2} [\![t]\!]_\rho$. ※

THEOREM 62 (Head-normalisation of an abstract machine)

We assume the following hypotheses:

Well-foundedness:

The LAF instance is well-founded.

Typing correlation:

If $\rho \in [\![\Gamma]\!]$ and $\mathfrak{d} \in [\![(\Delta^l, \mathbf{r})]\!]$ then $(\rho; \mathfrak{d}) \in [\![\Gamma; (\Delta^l, \mathbf{r})]\!]$.

We conclude that, for all $\rho \in [\![\Gamma]\!]$, if $\Gamma \vdash t$ then $\langle\!\langle\ t \mid \rho\ \rangle\!\rangle$ head-normalises. ※

[2] for the reduction relation defined in Definition 104

Proof: Stability is obvious for the head-normalisation model:

Assume $[\![f(p)]\!]_{\rho;d} \in \bot$. Following the previous remark, this entails that $\langle\!\langle\ f(p)\ \mid\ \rho; d\ \rangle\!\rangle$ head-normalises. Hence, $\langle \langle\!\langle f\ \mid\ \rho \rangle\!\rangle \mid pd \rangle$ head-normalises, which is literally what $\in [\![f]\!]_\rho \perp \tilde{p}(d)$ means.

So we can apply the Adequacy Lemma (Lemma 51), and obtain that $[\![t]\!]_\rho$ head-normalises, from which get that $\langle\!\langle\ t\ \mid\ \rho\ \rangle\!\rangle$ head-normalises. □

We now see how this applies to a syntactic abstract machine. Assume we have a well-founded LAF instance, and a syntactic abstract machine for it that features identity evaluation contexts, i.e. a family of contexts id satisfying $\mathsf{id}\,[x^+] = x^+$ and $\mathsf{id}\,[x^-] = x^-$.

Corollary 63 (Head normalisation) Assume that the evaluation context algebra $\mathcal{G}$ and Co satisfy the typing correlation.

If $\Gamma \vdash t$ then $\langle\!\langle\ t\ \mid\ \mathsf{id}\ \rangle\!\rangle$ head-normalises.[3] ※

Proof: The valuation id^e is in $[\![\Gamma^e]\!] = \mathbb{S}$.

Every positive label x^+ is in $[\![(a^l, \mathbf{r})]\!]_\sigma = \mathbb{V}_+ = \mathsf{Lab}_+$ (for every σ, a^l and $\mathbf{r}$).

Every negative label x^- is in $[\![(M^l, \mathbf{r})]\!]^-_\sigma$ (for every σ, M^l and $\mathbf{r}$),
since x^- is in $\mathscr{N} = \mathbb{V}_- = \mathsf{Lab}_- \cup \mathbb{C}$ and $x^- \perp pd$ for all $(p, d) \in [\![(M^l, \mathbf{r})]\!]^+_\sigma$.[4]

Hence, the identity evaluation context id is in $[\![\Gamma]\!]$. □

In particular, LAF_{K1}, LAF_{K2}, LAF_J are all head normalising.

7.3 Re-using proofs

Now, in order to have strong normalisation, and even just cut-elimination itself, our notion of abstract machine above is too weak, as it only (and deterministically) performs "head reduction".

A state of a syntactic machine such as $\langle v \mid p\, d \rangle$ could almost be read back as a real command, if only we could *compute* closures such as $\langle\!\langle f\ \mid\ \rho \rangle\!\rangle$, which we never do: just as in the weak reduction in λ-calculus, we never propagate the evaluation context ρ (which can be seen as a substitution) into f (in other words propagate it under the abstraction represented by the meta-level function f).

We could compute a closure $\langle\!\langle f\ \mid\ \rho \rangle\!\rangle$ as a function $\langle\!\langle f \rangle\!\rangle_\rho : \mathsf{Pat} \rightharpoonup \mathsf{Terms}$ such that

$$\langle\!\langle f \rangle\!\rangle_\rho(p) = \langle\!\langle f(p) \rangle\!\rangle_{\rho';\mathsf{idd}}$$

with the recursively defined propagation of an evaluation context ρ into a command c denoted $\langle\!\langle c \rangle\!\rangle_\rho$, and where

- idd is an "identity decomposition term", to create identity bindings for the labels in $f(p)$ introduced by the application of f to p;
- ρ' is the update of ρ, providing the same bindings as ρ but taking care that the labels might have changed after the context extension with idd.

[3] for the evaluation context id with $\mathsf{dom}^+(\mathsf{id}) = \mathsf{dom}^+(\Gamma)$ and $\mathsf{dom}^+(\mathsf{id}) = \mathsf{dom}^+(\Gamma)$

[4] Indeed, $\langle x^- \mid pd \rangle$ is head-normalising since it cannot be reduced by the abstract machine.

But so far a LAF instance does not tell us how to infer idd and ρ' from ρ.

This is exactly the same situation as with the eigenlabels for which a LAF instance with eigenlabels provided two functions $\mathsf{st}^\Gamma_\Delta$ and π^Γ_Δ to do exactly that.

We therefore enrich the concept of a LAF instance with eigenlabels as follows:

DEFINITION 106 (LAF instance with explicit label updates)

A LAF *instance with explicit label updates* is given by the following tuple:

$$(\mathbb{S}, \mathsf{Lab}_e, \mathbb{T}, \Vdash, \mathbb{A}, \mathbb{M}, \equiv, \mathsf{Lab}_+, \mathsf{Lab}_-, \mathsf{Co}, \mathbb{R}, \mathsf{Pat}, \vdash, \pi^{\mathcal{V}}_\Delta, \mathsf{st}^{\mathcal{V}}_\Delta)$$

whose components are exactly as in the definition of a LAF instance with eigenlabels, except that

- The map operation of the typing context algebra Co satisfies the following property:

  For all $f_1 : \mathbb{A}_\downarrow \to \mathbb{A}_\downarrow$ and $f_2 : \mathbb{M}_\downarrow \to \mathbb{M}_\downarrow$, all $(\mathbb{A}_\downarrow, \mathbb{M}_\downarrow, \mathbb{S})$-decompositions Δ and Δ', and all typing contexts Γ,

  If Δ relates to Δ' according to $\{(a, f_1(a)) \mid a \in \mathbb{A}_\downarrow\}$ $\{(m, f_2(m)) \mid m \in \mathbb{M}_\downarrow\}$ and the identity relation on sorts,
  then $(f_1, f_2) \circ (\Gamma; \Delta)$ relates to $((f_1, f_2) \circ \Gamma); \Delta'$ according to the identity relations.
- There is a $(\mathsf{Lab}_+, \mathsf{Lab}_-, \mathsf{Lab}_e)$-context algebra $\mathbb{R}$ called the *renaming context algebra*, and equipped with a *renaming composition* that combines two renaming contexts π and π' into $\pi \circ \pi'$ so that $\pi \circ \pi'[x] = \pi[\pi'[x]]$ (resp. $\pi \circ \pi'[x^+] = \pi[\pi'[x^+]]$ and $\pi \circ \pi'[x^-] = \pi[\pi'[x^-]]$);
- we require the naming policies $\pi^{\mathcal{V}}_\Delta$ and $\mathsf{st}^{\mathcal{V}}_\Delta$ to give information not only on eigenlabels, but also on positive and negative labels:

  $$\begin{pmatrix} \mathbb{P}(\mathsf{Lab}) \times \mathbb{D}_{\mathsf{st}} & \to & \mathbb{R} \\ (\mathcal{V}, \Delta) & \mapsto & \pi^{\mathcal{V}}_\Delta \end{pmatrix} \qquad \begin{pmatrix} \mathbb{P}(\mathsf{Lab}) \times \mathbb{D}_{\mathsf{st}} & \to & \mathbb{D}_{\mathsf{Lab}_+,\mathsf{Lab}_-,\mathsf{Lab}_e} \\ (\mathcal{V}, \Delta) & \mapsto & \mathsf{st}^{\mathcal{V}}_\Delta \end{pmatrix}$$

  Clearly, we can extract from those naming policies the policies in the sense of Definition 92 (with types $(\mathbb{P}(\mathsf{Lab}) \times \mathbb{D}_{\mathsf{st}} \to (\mathsf{Lab}_e \to \mathsf{Lab}_e))$ and $(\mathbb{P}(\mathsf{Lab}) \times \mathbb{D}_{\mathsf{st}} \to \mathbb{D}_{\mathsf{unit},\mathsf{unit},\mathsf{Lab}_e})$).
  Finally, we require that $\mathbb{R}$ respect those naming policies.

※

REMARK 64 It is straightforward to define $\mathbb{R}$, $\pi^{\mathsf{dom}(\Gamma)}_{|\Delta|}$, and $\mathsf{st}^{\mathsf{dom}(\Gamma)}_{|\Delta|}$ in LAF_{K1} and LAF_{K2} to make them LAF instances with explicit label updates. ※

With this information, we can now properly define the free labels of a proof-term, something which we surprinsingly did not need so far, but that will indicate which parts of a typing environment are actually used in a proof.

DEFINITION 107 (Free labels) The free labels of a positive term (resp. decomposition term, command) that is typed in a typing context Γ, are defined by the rules of Fig. 29. ※

Knowing what free variables are, we are now able, given a proof of a sequent, to replay the proof for any other sequent whose typing context contains the atoms and molecules that type the free variables of the original proof.

For this we define the renaming of a term:

DEFINITION 108 (Renaming) The renaming, denoted $\pi \cdot t^+$ (resp. $\pi \cdot d$, $\pi \cdot t$), by a renaming context π, of a positive term (resp. decomposition term, command) that is typed in a typing context Γ, is defined by the rules of Fig. 30. ※

$$
\begin{array}{lcl}
\mathsf{FL}(pd) & := & \mathsf{FL}(d) \\
\\
\mathsf{FL}(x^+) & := & \{x^+\} \\
\mathsf{FL}(f) & := & \bigcup_{p \in \mathsf{Dom}(f)} \pi^{-1}(\mathsf{FL}(f(p))) \\
\mathsf{FL}(\bullet) & := & \emptyset \\
\mathsf{FL}(d_1, d_2) & := & \mathsf{FL}(d_1) \cup \mathsf{FL}(d_2) \\
\mathsf{FL}(r.d) & := & \mathsf{FL}(r) \cup \mathsf{FL}(d) \\
\\
\mathsf{FL}(\langle x^- \mid t^+ \rangle) & := & \{x^-\} \cup \mathsf{FL}(t^+) \\
\mathsf{FL}(\langle f \mid t^+ \rangle) & := & \mathsf{FL}(f) \cup \mathsf{FL}(t^+)
\end{array}
$$

where π is the function in $(\mathsf{Lab}_+ \to \mathsf{Lab}_+) \cup (\mathsf{Lab}_- \to \mathsf{Lab}_-) \cup (\mathsf{Lab}_e \to \mathsf{Lab}_e)$ mapping every $x^+ \in \mathsf{Lab}_+$ to $\pi^{\mathsf{dom}(\Gamma)}_{|p|}[x^+]$ (resp. $x^- \in \mathsf{Lab}_-$ to $\pi^{\mathsf{dom}(\Gamma)}_{|p|}[x^-]$, and $x \in \mathsf{Lab}_e$ to $\pi^{\mathsf{dom}(\Gamma)}_{|p|}[x]$).

Figure 29: Free labels

$$
\begin{array}{lcl}
\pi \cdot pd & := & p(\pi \cdot d) \\
\\
\pi \cdot x^+ & := & \pi\,[x^+] \\
\pi \cdot f & := & p \mapsto \left(\left(\pi^{\mathsf{dom}(\Gamma)}_{|p|} \circ \pi\right); \mathsf{st}^{\mathsf{dom}(\Gamma)}_{|p|}\right) \cdot f(p) \\
\pi \cdot \bullet & := & \bullet \\
\pi \cdot (d_1, d_2) & := & (\pi \cdot d_1), (\pi \cdot d_2) \\
\pi \cdot (r.d) & := & \pi^e(r).(\pi \cdot d) \\
\\
\pi \cdot \langle x^- \mid t^+ \rangle & := & \langle \pi\,[x^-] \mid (\pi \cdot t^+) \rangle \\
\pi \cdot \langle f \mid t^+ \rangle & := & \langle (\pi \cdot f) \mid (\pi \cdot t^+) \rangle
\end{array}
$$

Figure 30: Renaming

In the renaming of a function f, $\pi^{\mathsf{dom}(\Gamma)}_{|p|} \circ \pi$ updates the co-domain of π as we went "through a binding", and composing with $\mathsf{st}^{\mathsf{dom}(\Gamma)}_{|p|}$ adds the "identity bindings" for the labels introduced by the application of f to p.

Now as mentioned before, when we have a proof for a particular sequent, we want to identify when it can be replayed for another sequent. For this we define what it means for a typing context Γ' to at least contain the instantiated atoms and molecules of a typing context Γ: this is done by identifying a renaming π that will map the labels in Γ' to some labels in Γ' that have the same type.

Definition 109 (Context embedding)

We say that Γ embeds into Γ' along a renaming context π, written $\Gamma \sqsubseteq_\pi \Gamma'$, if for all x (resp. x^+, x^-) in $\mathsf{dom}^e(\pi)$ (resp. $\mathsf{dom}^+(\pi)$, $\mathsf{dom}^-(\pi)$) we have $\Gamma\,[x] = \Gamma'\,[\pi\,[x]]$ (resp. $\Gamma\,[x^+] = \Gamma'\,[\pi\,[x^+]]$, $\Gamma\,[x^-] = \Gamma'\,[\pi\,[x^-]]$). ※

Notice that the domain of π might be smaller than that of Γ, so that π does not necessarily map *every* label declared in Γ. This is a feature (rather than a bug) that will allow us to ignore those instantiated atoms and molecules in Γ that are not used in the proof that we

want to replay (the renaming π may only be defined on those labels that are free in the proof-term).

THEOREM 65 (Replaying a proof) Assume the following property:

Renaming correlation:

For all Γ, Γ', π, if $\Gamma \sqsubseteq_\pi \Gamma'$ then $\Gamma; \Delta \sqsubseteq_{\pi'} \Gamma'; \Delta$, where $\pi' = \left(\pi^{\mathsf{dom}(\Gamma)}_{|p|} \circ \pi\right); \mathsf{st}^{\mathsf{dom}(\Gamma)}_{|p|}$.

We conclude that, for all $\pi \in \mathbb{R}$ such that $((\rho^e, \rho^e) \circ \Gamma) \sqsubseteq_\pi \Gamma'$,

1. if $\mathsf{FL}(t^+) \subseteq \mathsf{Dom}(\pi)$ and $\Gamma \vdash [t^+ : (M^l, \mathbf{r})]$ then $\Gamma' \vdash [(\pi \cdot t^+) : (M^l, \mathbf{r})]$
2. if $\mathsf{FL}(d) \subseteq \mathsf{Dom}(\pi)$ and $\Gamma \vdash d : (\Delta^l, \mathbf{r})$ then $\Gamma' \vdash (\pi \cdot d) : (\Delta^l, \mathbf{r})$
3. if $\mathsf{FL}(t) \subseteq \mathsf{Dom}(\pi)$ and $\Gamma \vdash t$ then $\Gamma' \vdash (\pi \cdot t)$

※

Proof: See the Coq proof [GL14]. □

A LAF instance with explicit label updates thus allows us to apply a renaming to a proof to get a proof of a new sequent. This will be used heavily in an implementation of proof-search that memoises proofs in order to paste them as often as possible.

7.4 Cut-elimination

We now show how to use substitution and the substitution lemma to define a normalisation procedure, in a LAF instance with explicit label updates, to produce cut-free terms.

DEFINITION 110 (Normalisation) We take a syntactic abstract machine, whose evaluation context algebra is equipped with a *renaming operation* that associates, to a renaming context π and an evaluation context ρ, an evaluation context $\pi \circ \rho$ such that

- for all $x \in \mathsf{dom}^e(\rho)$, we have $\pi \circ \rho\,[x] = \pi^e(\rho\,[x])$
- for all $x^+ \in \mathsf{dom}^+(\rho)$, we have $\pi \circ \rho\,[x^+] = \pi\,[\rho\,[x^+]]$
- for all $x^- \in \mathsf{dom}^-(\rho)$, we have
  - if $\rho\,[x^-] = y^-$ then $\pi \circ \rho\,[x^-] = \pi\,[y^-]$
  - if $\rho\,[x^-] = \langle\!\langle\, f \mid \rho' \,\rangle\!\rangle$ then $\pi \circ \rho\,[x^-] = \langle\!\langle\, f \mid \pi \circ \rho' \,\rangle\!\rangle$

We define the *big-step semantics* of the LAF instance with explicit label updates as two relations

- one denoted $d \Downarrow d'$ between an evaluation decomposition d and a cut-free decomposition term d',
- one denoted $\langle\!\langle\, t \mid \rho \,\rangle\!\rangle \Downarrow t'$ between a contextualised command $\langle\!\langle\, t \mid \rho \,\rangle\!\rangle$ and a cut-free command t',

defined by simultaneous induction by the rules of Fig. 31.

We say that a contextualised command $\langle\!\langle\, t \mid \rho \,\rangle\!\rangle$ (resp. an evaluation decomposition d) *normalises* if there is some t' such that $\langle\!\langle\, t \mid \rho \,\rangle\!\rangle \Downarrow t'$ (resp. some d' such that $d \Downarrow d'$). Notice in that case that t' (resp. d') is cut-free. We also say that an evaluation triple $\langle v \mid pd \rangle$ *normalises* if $\langle v \mid pd \rangle \longrightarrow_{\mathsf{head}_{123}} \langle x^- \mid p'd' \rangle$ and d' normalises. ※

In order to show that this forms a cut-elimination procedure, we need to

$$\frac{\forall p \in \mathsf{Dom}(f), \qquad \langle\!\langle\ f(p)\ \mid\ \left(\left(\pi_{|p|}^{\mathsf{dom}(\rho)} \circ \rho\right) ; \mathsf{st}_{|p|}^{\mathsf{dom}(\rho)}\right) \rangle\!\rangle \Downarrow f'(p)}{\langle\!\langle\ f\ \mid\ \rho \rangle\!\rangle \Downarrow f'} \qquad \frac{}{x^- \Downarrow x^-}$$

$$\frac{}{x^+ \Downarrow x^+} \qquad \frac{}{\bullet \Downarrow \bullet} \qquad \frac{d_1 \Downarrow d_1' \qquad d_2 \Downarrow d_2'}{d_1, d_2 \Downarrow d_1', d_2'} \qquad \frac{d \Downarrow d'}{r.d \Downarrow r.d'}$$

$$\frac{\langle\!\langle\ t\ \mid\ \rho \rangle\!\rangle \longrightarrow_{\mathsf{head}_{123}}^{*} \langle x^- \mid pd \rangle \qquad d \Downarrow d'}{\langle\!\langle\ t\ \mid\ \rho \rangle\!\rangle \Downarrow \langle x^- \mid pd' \rangle}$$

Figure 31: Cut-elimination

- give typing rules for contextualised commands, evaluation decomposition, and evaluation triples;
- show that $\longrightarrow_{\mathsf{head}_{123}}^{*}$ satisfies Subject Reduction with these rules;
- show that every typed contextualised command normalises.

Definition 111 (Typing the elements of a syntactic abstract machine)

The typing rules for the elements of a syntactic abstract machine are given in Fig. 32, where $\vdash_{\mathsf{LAF}}$ denotes the derivability of sequents in LAF (Fig. 27). ※

Theorem 66 (Subject Reduction)

1. If $\Gamma \vdash \langle\!\langle\ t\ \mid\ \rho \rangle\!\rangle$ and $\langle\!\langle\ t\ \mid\ \rho \rangle\!\rangle \longrightarrow_{\mathsf{head}_1, \mathsf{head}_2} \langle v \mid pd \rangle$ then $\Gamma \vdash \langle v \mid pd \rangle$.
2. If $\Gamma \vdash \langle v \mid pd \rangle$ and $\langle v \mid pd \rangle \longrightarrow_{\mathsf{head}_3} \langle\!\langle\ t\ \mid\ \rho \rangle\!\rangle$ then $\Gamma \vdash \langle\!\langle\ t\ \mid\ \rho \rangle\!\rangle$.
3. If $\Gamma \vdash d : (\Delta, \mathbf{r})$ and $d \Downarrow d'$ then $\Gamma \vdash_{\mathsf{LAF}} d' : (\Delta, \mathbf{r})$.
4. If $\Gamma \vdash \langle\!\langle\ t\ \mid\ \rho \rangle\!\rangle$ and $\langle\!\langle\ t\ \mid\ \rho \rangle\!\rangle \Downarrow t'$ then $\Gamma \vdash_{\mathsf{LAF}} t'$.

※

Proof: The first two points are given by a simple rearrangement of the sub-derivation trees. The last two points are proved by induction on the normalisation derivations. □

Finally, we adapt the realisability model for head normalisation (Definition 105) to prove cut-elimination:

Definition 112 (A realisability model for normalisation)

Evaluation decompositions

$$\frac{}{\Gamma \vdash \bullet : (\bullet, \mathbf{r})} \qquad \frac{\Gamma \vdash d_1 : (\Delta_1, \mathbf{r}) \quad \Gamma \vdash d_2 : (\Delta_2, \mathbf{r})}{\Gamma \vdash d_1, d_2 : ((\Delta_1, \Delta_2), \mathbf{r})} \qquad \frac{\Gamma^e \Vdash r' : s \quad \Gamma \vdash d : (\Delta, r' :: \mathbf{r})}{\Gamma \vdash r'.d : s.(\Delta, \mathbf{r})}$$

$$\frac{\Gamma \left[x^+\right] \equiv (a, \mathbf{r})}{\Gamma \vdash x^+ : (a, \mathbf{r})} \qquad \frac{\Gamma \vdash \rho : \Gamma' \quad \Gamma' \vdash_{\mathsf{LAF}} f : (\sim M, \mathbf{r})}{\Gamma \vdash \langle\!\langle f \mid \rho \rangle\!\rangle : (\sim M, \mathbf{r})}$$

Evaluation triples

$$\frac{\Delta \Vdash p : M \quad \Gamma \vdash d : (\Delta, \mathbf{r})}{\Gamma \vdash \langle x^- \mid pd \rangle} \, \Gamma\left[x^-\right] = (M, \mathbf{r})$$

$$\frac{\Gamma \vdash \langle\!\langle f \mid \rho \rangle\!\rangle : (\sim M, \mathbf{r}) \quad \Delta \Vdash p : M \quad \Gamma \vdash d : (\Delta, \mathbf{r})}{\Gamma \vdash \langle \langle\!\langle f \mid \rho \rangle\!\rangle \mid pd \rangle}$$

Contextualised commands

$$\frac{\Gamma \vdash \rho : \Gamma' \quad \Gamma' \vdash_{\mathsf{LAF}} t}{\Gamma \vdash \langle\!\langle \, t \mid \rho \, \rangle\!\rangle}$$

Evaluation contexts

$$\frac{\begin{array}{ll} (\forall x \in \mathsf{dom}^e(\Gamma'), & \Gamma^e \Vdash \rho\left[x\right] : \Gamma'\left[x\right]) \\ (\forall x^+ \in \mathsf{dom}^+(\Gamma'), & \Gamma\left[\rho\left[x^+\right]\right] \equiv \Gamma'\left[x^+\right] \\ \left(\begin{array}{l} \forall x^- \in \mathsf{dom}^+(\Gamma'), \\ \quad \text{either } \Gamma\left[y^-\right] = \Gamma'\left[x^-\right] \qquad \text{if } \rho\left[x^-\right] = y^- \\ \quad \text{ or } \Gamma \vdash \langle\!\langle f \mid \rho' \rangle\!\rangle : (\sim M, \mathbf{r}) \quad \text{if } \rho\left[x^-\right] = \langle\!\langle f \mid \rho' \rangle\!\rangle \text{ and } \Gamma'\left[x^-\right] = (M, \mathbf{r}) \end{array} \right) \end{array}}{\Gamma \vdash \rho : \Gamma'}$$

Figure 32: Typing a syntactic abstract machine

The *normalisation model* for this syntactic abstract machine is

$$\begin{array}{lll} \mathscr{C} & := & \mathbb{S} \\ \mathscr{T} & := & \mathbb{T} \\ \mathscr{L} & := & \mathsf{Lab}_+ \\ \mathscr{P} & := & \mathsf{Pat} \times \mathbb{D}_{\mathsf{Lab}_+, \mathsf{Lab}_- \cup \mathbb{C}, \mathbb{T}} \\ \mathscr{N} & := & \mathsf{Lab}_- \cup \mathbb{C} \\ v \perp (p, d) & \text{if the evaluation triple } \langle v \mid pd \rangle & \text{normalises} \\ \tilde{\mathsf{Co}} & := & \mathcal{G} \\ \forall \mathfrak{d} \in \tilde{\mathbb{D}}, & \tilde{p}(\mathfrak{d}) & := (p, \mathfrak{d}) \\ \forall r \in \mathbb{T}, \forall \sigma \in \mathscr{C}, & [\![r]\!]_\sigma & := \langle\!\langle r \rangle\!\rangle_\sigma \\ \forall f : \mathsf{Pat} \rightharpoonup \mathsf{Terms}, \forall \rho \in \tilde{\mathsf{Co}}, & [\![f]\!]_\rho & := \langle\!\langle f \mid \rho \rangle\!\rangle \\ \forall s \in \mathbb{S}, & [\![s]\!] & := \mathbb{T} \\ \forall \Sigma \in \mathbb{C}, & [\![\Sigma]\!] & := \mathbb{S} \\ \forall a^l \in \mathbb{A}_l, \forall \mathfrak{rl} \in \mathscr{T}^l, & [\![a^l]\!](\mathfrak{rl}) & := \mathsf{Lab}_+ \end{array}$$

※

Remark 67

Notice this is the same definition as Definition 105, except for the orthogonality relation which we have strengthened by requiring normalisation instead of head-normalisation.

Of course we still have $\langle\!\langle\ t\ \mid\ \rho\ \rangle\!\rangle \longrightarrow_{\mathsf{head}_{12}} [\![t]\!]_\rho$. ※

Theorem 68 (Normalisation of a syntactic abstract machine)

We assume the following hypotheses:

Well-foundedness:

The LAF instance is well-founded.

Correlation with eigenlabels:

$\tilde{\mathsf{Co}}$ and Co satisfy the correlation with eigenlabels property.

We conclude that, for all $\rho \in [\![\Gamma]\!]$, if $\Gamma \vdash t$ then $\langle\!\langle\ t\ \mid\ \rho\ \rangle\!\rangle$ normalises. ※

Proof: Stability is obvious for the normalisation model:

Assume $[\![f(p)]\!]_{\rho;d} \in \bot$. Following the previous remark, this entails that $\langle\!\langle\ f(p)\ \mid\ \rho; d\ \rangle\!\rangle$ normalises. Hence, $\langle \langle\!\langle f\ \mid\ \rho \rangle\!\rangle \mid pd\rangle$ normalises, which is literally what $\in [\![f]\!]_\rho \perp \tilde{p}(d)$ means.

So we can apply the Adequacy Lemma (Lemma 55), and obtain that $[\![t]\!]_\rho$ normalises, from which get that $\langle\!\langle\ t\ \mid\ \rho\ \rangle\!\rangle$ normalises. □

Again, assume we have a LAF instance with explicit label updates, a syntactic machine for it that features identity evaluation contexts, i.e. a family of contexts id satisfying $\mathsf{id}\,[x^+] = x^+$, $\mathsf{id}\,[x^-] = x^-$ and $\mathsf{id}\,[x] = x$.

Lemma 69 (Normalisation and renaming)

If d (resp. t) normalises then $\pi \cdot d$ (resp. $\pi \cdot t$) normalises. ※

Proof: By induction on the normalisation derivation. □

Lemma 70 (Evaluation decompositions in the model are normalising)

1. For all typing decomposition Δ^l, for all $\mathbf{r}$, σ and all $d \in \mathbb{D}_{\mathsf{Lab}_+,\mathsf{Lab}_-,\mathsf{Lab}_e}$ with the same structure as Δ^l, $d \in [\![(\Delta^l, \mathbf{r})]\!]_\sigma$.
2. For all molecules M^l, for all $\mathbf{r}$, σ and all $\langle\!\langle f\ \mid\ \rho \rangle\!\rangle$ in $[\![(M^l, \mathbf{r})]\!]^-_\sigma$, $\langle\!\langle f\ \mid\ \rho \rangle\!\rangle$ normalises.
3. For all typing decomposition Δ^l, for all $\mathbf{r}$, σ and all d in $[\![(\Delta^l, \mathbf{r})]\!]_\sigma$, d normalises.
4. For all molecules M^l, for all $\mathbf{r}$, σ and all negative labels x^-, $x^- \in [\![(M^l, \mathbf{r})]\!]^-_\sigma$.

※

Proof: By simultaneous induction on Δ^l and M^l, using the well-founded property of the LAF instance.

For point 1: by induction on Δ^l, the base case being point 4.

For point 2: by unfolding the definition of $[\![(M^l, \mathbf{r})]\!]^-_\sigma$, $\langle\!\langle f\ \mid\ \pi^{\mathsf{dom}(\rho)}_{|p|} \circ \rho \rangle\!\rangle$ is orthogonal to $(p, \mathsf{st}^{\mathsf{dom}(\rho)}_{|p|})$ (using point 1).

For point 3: by induction on Δ^l, the base case being point 2.

For point 4: for all $(p, d) \in [\![(M^l, \mathbf{r})]\!]^+_\sigma$, we have the evaluation decomposition d in some $[\![(\Delta^l, \mathbf{r})]\!]_\sigma$, and by point 3 d normalises; hence $\langle x^- \mid pd\rangle$ normalises (i.e. $x^- \perp pd$), so x^- is in $[\![(M^l, \mathbf{r})]\!]^-_\sigma$. □

COROLLARY 71 (Cut-elimination) Assume that the evaluation context algebra $\mathcal{G}$ and Co satisfy the correlation with eigenlabels property.

If $\Gamma \vdash t$ then $\langle\!\langle\ t\ \mid\ \mathsf{id}\ \rangle\!\rangle$ normalises.[5]

Therefore the LAF instance with explicit label updates admits cuts. ※

Proof: Every eigenlabel x is in $[\![s]\!] = \mathbb{T} = \mathbb{T}$ (for every s).

Every positive label x^+ is in $\left[\!\left[(a^l, \mathbf{r})\right]\!\right]_\sigma = \mathbb{V}_+ = \mathsf{Lab}_+$ (for every σ, a^l and $\mathbf{r}$).

Every negative label x^- is in $\left[\!\left[(M^l, \mathbf{r})\right]\!\right]^-_\sigma$ (previous lemma).

Hence, the identity evaluation context id is in $[\![\Gamma]\!]$, and we can apply the previous theorem.

Combined with Subject Reduction (Theorem 66), we can transform every proof with cuts into a cut-free proof. □

In particular, LAF_{K1} and LAF_{K2} admit cuts.

7.5 Conclusion and further work: Strong normalisation

Now, we have proved cut-elimination but not strong normalisation, as we have used a big-step operational semantics to reduce proof-terms to cut-free forms, but we still have not defined a non-deterministic reduction relation for which strong normalisation might be interesting. For this we would definitely need to compute closures (which we still have avoided so far), by pushing down evaluation contexts with the rules of Fig. 33.

$$
\begin{array}{lcl}
\rho \cdot pd & := & p(\rho \cdot d) \\
\\
\rho \cdot x^+ & := & \pi\left[x^+\right] \\
\rho \cdot f & := & p \mapsto \left(\left(\pi^{\mathsf{Dom}(\Gamma)}_{|p|} \circ \rho\right) ; \mathsf{st}^{\mathsf{Dom}(\Gamma)}_{|p|}\right) \cdot f(p) \\
\rho \cdot \bullet & := & \bullet \\
\rho \cdot (d_1, d_2) & := & (\rho \cdot d_1), (\rho \cdot d_2) \\
\rho \cdot (r.d) & := & \pi^e(r).(\rho \cdot d) \\
\\
\rho \cdot \langle x^- \mid t^+ \rangle & := & \langle \pi\left[x^-\right] \mid (\rho \cdot t^+) \rangle \\
\rho \cdot \langle f \mid t^+ \rangle & := & \langle (\rho \cdot f) \mid (\rho \cdot t^+) \rangle
\end{array}
$$

Figure 33: Substitution

Using this to define a non-deterministic reduction relation, Subject Reduction for the latter would rely on a typability result for the substitution operation, similar to Theorem 65 for renamings. On the other hand, we would then avoid introducing all the extra typing rules of Fig. 32, as the reduction relation would not rely on the constructs of an abstract machine but would directly operate on terms and commands.

We conjecture that the normalisation model of Definition 112 would work, exactly as it is, to show that typed terms and commands are strongly normalising.[6] However, we would probably need to prove the equivalent, for LAF, of the substitution lemma in λ-calculus:

[5] for an identity evaluation context id with the same domains as Γ

[6] Well, not exactly "as it is", since instead of closures we would directly take functions from $\mathsf{Pat} \rightharpoonup \mathsf{Terms}$.

$$\left\{{}^{P}\!/_{y}\right\}\left\{{}^{N}\!/_{x}\right\}M = \left\{\left\{{}^{P}\!/_{y}\right\}{}^{N}\!/_{x}\right\}\left\{{}^{P}\!/_{y}\right\}M$$

which we would also need if we are to prove confluence of the (non-deterministic) reduction relation. Such a lemma would broach the topic of equality between LAF proof-terms, a question that we carefully managed to avoid so far as it involves considering which equality we take on the meta-level functions $f : \mathsf{Pat} \rightharpoonup \mathsf{Terms}$ (extensional, intensional?).

We therefore leave all of these questions for future work.

Part III

Theorem proving

Introduction

The Sequent Calculus, even in Gentzen's original formulation [Gen35], is not only a formalism to represent complete proofs, but it also specifies a natural, non-deterministic proof-search procedure: the gradual completion of incomplete proof-trees, starting from the one-node tree carrying a sequent to be proved, and extending the incomplete branches step-by-step until a complete proof-tree is obtained. This is called *bottom-up proof-search*, or *root-first proof-search*. It is the basic mechanism of e.g. *tableaux methods* (see e.g. [DGHP99, BG01]), and it was also used to describe and extend the logic programming paradigm [MNPS91].

As mentioned in Chapter 3, focussing was originally introduced [AP89, And92] in the framework of linear logic [Gir87], with motivations for logic programming. In other words, focussing helped designing proof-search procedures. As described in Chapter 3 (and in Part II), focussing had a important impact on the theory of complete proofs (and their semantics). We now come back to the view of focussing as an algorithmic methodology for completing incomplete proof-trees.

In [LDM11] we used a focussed sequent calculus to describe type inhabitation / proof-construction in Pure Type Systems [Bar92] (and higher-order unification), which provides (the basis for) the type theory behind several proof assistants such as Coq [Coq] or Twelf [Twe].

In this dissertation, we illustrate how the above methodology can apply to theorem proving in classical logic. This was mostly the object of our PSI project [PSI], with contributions shared with my student Mahfuza Farooque [FLM12b, FLM12a, FGL13, FGLM13, Far13].

Analytic tableaux probably form the proof-search procedures that are closest to the sequent calculus. Being much more procedure-oriented than the sequent calculus (whose theory handles complete proofs), tableaux offer an important difference in their explicit management of existential variables during search (variables that may be instantiated to conclude provability or refutability of the input), for instance via *first-order unification* in the case of pure first-order logic.

Clause tableaux provide variants of tableaux procedures that exploit a clausal formulation of the formulae to be refuted (which they share with resolution-based techniques or even SAT-solving techniques). In Farooque's Ph.D. [Far13], clause tableaux were shown to be simulated (in a strong sense) by root-first proof-search in the focussed sequent calculus LKF (see e.g. Chapter 3 or [LM09]), when the latter is extended with the ability to change the polarity of atoms *on-the-fly* during proof-search. More interestingly, clause tableaux that satisfy *connection* properties (strong and weak connections) were shown to correspond to the construction of LKF proofs that abide by specific polarisation policies:

Connections require that, when a branch of an incomplete proof-tree (or tableau) is extended by expanding on a clause $l_1 \vee \cdots \vee l_n$, thus creating n new sub-branches, then at least

one of them is closed immediately by connecting its corresponding literal l_i with a literal that was obtained earlier on the branch.

Similarly, when root-first proof-search in LKF focusses on (the negation of)[7] the clause $l_1 \vee \cdots \vee l_n$, a policy that forces the polarity of $\vee$ to be negative and forces the polarity of one literal among $l_1 \dots l_n$ to also be negative, may be used so that the synchronous phase of LKF forces the immediate closing of the branch corresponding to that literal, by "connecting it" to a previously obtained literal.

This was formalised in [Far13], which also broached the topic of reasoning *modulo a theory*. Building on the idea of (clause) *tableaux-modulo-theories* suggested by Tinelli [Tin07] in connection with SAT-modulo-theories solving (SMT-solving), we developed in the PSI project [PSI] an extension of LKF with a decision procedure, and showed its application to SMT-solving.

This is what is presented in Chapter 8. The motivation behind it is to propose a focussed sequent calculus framework where different techniques for automated (or interactive!) theorem proving can be simulated: tuning the polarities or the polarisation policies determines (or contributes to determining) the proof-search strategies that capture the said techniques, switching for instance from a tableau procedure to an SMT-procedure (such as DPLL($\mathcal{T}$)) by a simple change of polarity policy.

This aim gave rise to the implementation of the Psyche prototype, which is still in the early development phase, which is the object of the system description [GL13] and which is presented in Chapter 9. The system is designed as a platform for implementing the proof-search strategies that capture different theorem proving techniques. Doing so raises the question of trust, and of the correctness of an output produced by any of these implemented strategies. What the platform offers is an architecture that lets various strategies and techniques be experimented, and implemented as *plugins* via an API with Psyche's *kernel*, while guaranteeing the correctness of the output. This is obtained by a somewhat transformed LCF-architecture [GMW79]. A potential application of such a platform is to offer it as a backend prover for the proof obligations produced by verification tools such as Why3 [BFM+13, FP13]; the strategy programming and experimenting facilities of Psyche could then be used to tune the behaviour of the proof-search to the specific kind of proof obligations that need to be proved, without worrying about correctness.

We then conclude this dissertation in Chapter 10 with the perspectives of Psyche's development as impacted by the material developed in this dissertation, and an opening to the numerous connections with the automated reasoning literature that remain to be investigated.

[7] In sequent calculus we try to prove the negation of the formulae that tableaux methods seek to refute.

Chapter 8

DPLL($\mathcal{T}$) as proof-search in a focussed sequent calculus

Contents

This chapter focusses on automated techniques for solving the *Satisfiability Modulo Theories (SMT)* family of problems, illustrating how these can be available in a system based on goal-directed proof-search. Such problems generalise propositional SAT-problems: instead of considering the satisfiability of conjunctive normal forms (CNF) over propositional variables, SMT problems concern the satisfiability of CNF over atomic propositions from a theory $\mathcal{T}$ such as linear arithmetic or bit vectors. Given a procedure deciding the consistency -with respect to $\mathcal{T}$- of a conjunction of atoms or negated atoms, SMT-solving organises a cooperation between this procedure and SAT-solving techniques, thus providing a decision procedure for SMT-problems. This smart extension of the successful SAT-solving techniques opened a prolific area of research and led to the implementation of ever-improving tools, namely SMT-solvers, now crucial to a number of applications in software verification. The architecture of SMT-solvers is based on the extension of the Davis, Putnam, Logemann and Loveland (DPLL) procedure [DP60, DLL62] for solving SAT-problems to a procedure called DPLL($\mathcal{T}$) [NOT06] addressing SMT-problems.

This chapter does not try to improve the DPLL($\mathcal{T}$) technique itself, or current SMT-solvers based on it, but makes a step towards the integration of the technique into a sequent calculus

framework. More precisely, we investigate how we can perform each of the *steps* of DPLL($\mathcal{T}$) as bottom-up proof-search in sequent calculus. This allows the DPLL($\mathcal{T}$) algorithm to be applied *up-to-a-point*, where a switch to another technique can be made (depending on the newly generated goals).[1] This simulation can be seen as a first step toward a better proof-theoretical understanding of how different proof-search strategies (e.g. tableaux, resolution, DPLL($\mathcal{T}$),...), geared toward different logical fragments, could efficiently cooperate inside a common platform for theorem proving.

The polarities and the focussing properties of the sequent calculus we use allow us to derive a stronger result than the mere simulation of DPLL($\mathcal{T}$): the proofs that are the images of DPLL($\mathcal{T}$) runs finishing on UNSATcan be characterised by a simple criterion only involving the way polarities are assigned to literals and the way formulae are placed into the focus of sequents. From this criterion we directly get a simple proof-search strategy that is bisimilar to DPLL($\mathcal{T}$) runs: that which performs the depth-first completion of incomplete proof-trees (starting with the leftmost open leaf), using any inference steps satisfying the given criterion on polarities and focusing. The bisimulation ensures that bottom-up proof-search in sequent calculus can be as efficient as the DPLL($\mathcal{T}$) procedure.

Section 8.1 presents the variant of System LKF (from Section 3.2) that we use to describe DPLL($\mathcal{T}$) in terms of proof-search. Section 8.2 describes the details of how DPLL($\mathcal{T}$) is captured: we first identify an *elementary* version of DPLL($\mathcal{T}$) that is the direct extension of the *Classical DPLL procedure* to a background theory $\mathcal{T}$, as well as being a restriction of the full *Abstract DPLL DPLL Modulo Theories* system,[2] both of which can be found in [NOT06]; then we prove the bisimulation result and discuss the DPLL($\mathcal{T}$) mechanisms that are not in our elementary version. Section 8.3 concludes by connecting the above to the abstract LAF system(s) developed in Part II of this dissertation.

8.1 A version of LKF to work modulo a theory: LK^p($\mathcal{T}$)

8.1.1 Background

Clearly in root-first proof-search, asynchronous rules can be applied eagerly (i.e. can be chained, without creating backtrack points and losing completeness), since they are invertible. Focussing says that applying synchronous rules (although possibly creating backtrack points) can also be chained without losing completeness. This forced chaining of synchronous rules can be seen for instance in the LKF system of Section 3.2, where sequent may feature a formula in its *focus*.

A sequent with a positive atom in focus must be proved immediately by an axiom on that atom; hence, the polarity of atoms greatly affects the shape of proofs. As illustrated in e.g. [LM09], the following sequent expresses the *Fibonacci* logic program (in some language where addition is primitive) and a goal fib(n, p) (where n and p are closed terms):

[1] In contrast, other approaches to integrating SAT- or SMT-solving to a wider theorem proving framework usually rely on the automated technique to perform a *full* run; this is the case of [Web11, AFG+11, BCP11, BBP11] and Lescuyer's solver [LC09] that runs within the Coq proof assistant thanks to its *reflection* ability.

[2] that allows more advanced features such as *backjumping* and *clause learning*

$$\begin{array}{l}\mathsf{fib}(0,0),\\ \mathsf{fib}(1,1),\\ \forall i p_1 p_2(\mathsf{fib}(i,p_1)\Rightarrow\mathsf{fib}(i+1,p_2)\Rightarrow\mathsf{fib}(i+2,p_1+p_2))\\ \vdash \mathsf{fib}(n,p)\end{array}$$

The goal will be proved with backward-reasoning if the fib atoms are negative (yielding a proof of exponential size in n), and forward-reasoning if they are positive (yielding many proofs, one of which being linear).

In classical logic, polarities of connectives and atoms do not affect the provability of formulae, but still greatly affect the shape of proofs, and hence the basic proof-construction steps. This chapter shows how the $\mathsf{DPLL}(\mathcal{T})$ steps correspond to proof-construction steps for an appropriate management of polarities. For this we use a variant of the LKF system [LM09] presented in Section 3.2: the sequent calculus $\mathsf{LK}^p(\mathcal{T})$ [FGLM13, Far13].

In order to make logical sense of e.g. the primitive addition in the Fibonacci example above, we only enrich LKF with the ability to call a decision procedure to decide the consistency of conjunctions of literals w.r.t. a theory (i.e. the same as for $\mathsf{DPLL}(\mathcal{T})$): for a theory that equates $1+1$ and 2, a call to the procedure proves $p(2), p^\perp(1+1)\vdash$ in one step (unlike LKF's syntactic checks).

System LKF also assumes that all atoms come with a pre-determined polarity, whereas $\mathsf{LK}^p(\mathcal{T})$ allows *on-the-fly* polarisation of atoms: the root of a proof-tree might have none of its atoms polarised, but atoms may become positive or negative as progress is made in the proof-search.

8.1.2 Definitions

In this section we present the quantifer-free fragment of the focussed sequent calculus $\mathsf{LK}^p(\mathcal{T})$ [FGLM13, Far13]. This fragment concerns propositional classical logic *modulo a theory* and will be sufficent for the simulation of $\mathsf{DPLL}(\mathcal{T})$.

This sequent calculus (and this logic) involves a notion of *literal* and a notion of *theory*. The reader can safely see behind this terminology the standard notions from proof theory and automated reasoning. However at this point, very little is required from or assumed about those two notions.

Definition 113 (Literals)

Let $\mathcal{L}$ be a set of elements called *literals*, equipped with an involutive function called *negation* from $\mathcal{L}$ to $\mathcal{L}$. In the rest of this chapter, a possibly primed or indexed lowercase l always denotes a literal, and $l^\perp$ its negation. ※

Another ingredient of $\mathsf{LK}^p(\mathcal{T})$ is a *theory* $\mathcal{T}$, given in the form of an *inconsistency predicate*, a notion that we now introduce:

Definition 114 (Inconsistency predicates)

An *inconsistency predicate* is a predicate over sets of literals

- satisfied by the set $\{l, l^\perp\}$ for every literal l;
- that is upward closed (if a subset of a set satisfies the predicate, so does the set);
- such that if the sets $\mathcal{P}, l$ and $\mathcal{P}, l^\perp$ satisfy it then so does $\mathcal{P}$.

The smallest inconsistency predicate is called the *syntactical inconsistency* predicate[3]. If a set $\mathcal{P}$ of literals satisfies the syntactically inconsistency predicate, we say that $\mathcal{P}$ is *syntactically inconsistent*, denoted $\mathcal{P} \models$. Otherwise $\mathcal{P}$ is *syntactically consistent*.

The theory $\mathcal{T}$ in the notation $\mathsf{LK}^p(\mathcal{T})$ is described by means of an(other) inconsistency predicate, called the *semantical inconsistency predicate*, which will be a formal parameter of the inference system defining $\mathsf{LK}^p(\mathcal{T})$.

If a set $\mathcal{P}$ of literals satisfies the semantical inconsistency predicate, we say that $\mathcal{P}$ is *semantically inconsistent* or *inconsistent modulo theory*, denoted by $\mathcal{P} \models_{\mathcal{T}}$. Otherwise $\mathcal{P}$ is *semantically consistent* or *consistent modulo theory*. ※

DEFINITION 115 (Formulae, negation)

Let $\mathcal{L}$ be a set of literals. The formulae of propositional polarised classical logic are given by the following grammar:

$$\begin{array}{rrcll} \text{Formulae} & A, B, \ldots & ::= & l & \text{where } l \text{ ranges over } \mathcal{L} \\ & & \mid & A \wedge^+ B \mid A \vee^+ B \mid \top^+ \mid \bot^+ & \\ & & \mid & A \wedge^- B \mid A \vee^- B \mid \top^- \mid \bot^- & \end{array}$$

The size of a formula A, denoted $\sharp(A)$, is its size as a tree (number of nodes).

Let $\mathcal{P} \subseteq \mathcal{L}$ be syntactically consistent. Intuitively, it represents the set of literals declared to be positive.

We define $\mathcal{P}$-positive formulae and $\mathcal{P}$-negative formulae as the formulae generated by the following grammars:

$$\begin{array}{rrcl} \mathcal{P}\text{-positive formulae} & P, \ldots & ::= & p \mid A \wedge^+ B \mid A \vee^+ B \mid \top^+ \mid \bot^+ \\ \mathcal{P}\text{-negative formulae} & N, \ldots & ::= & p^\perp \mid A \wedge^- B \mid A \vee^- B \mid \top^- \mid \bot^- \end{array}$$

where p ranges over $\mathcal{P}$.

Let $\mathsf{U}_{\mathcal{P}}$ be the set of all $\mathcal{P}$-*unpolarised literals*, i.e. literals that are neither $\mathcal{P}$-positive nor $\mathcal{P}$-negative.

Negation is recursively extended into an involutive map on formulae as follows:

$(A \wedge^+ B)^\perp$	$:= A^\perp \vee^- B^\perp$	$(A \wedge^- B)^\perp$	$:= A^\perp \vee^+ B^\perp$
$(A \vee^+ B)^\perp$	$:= A^\perp \wedge^- B^\perp$	$(A \vee^- B)^\perp$	$:= A^\perp \wedge^+ B^\perp$
$(\top^+)^\perp$	$:= \bot^-$	$(\top^-)^\perp$	$:= \bot^+$
$(\bot^+)^\perp$	$:= \top^-$	$(\bot^-)^\perp$	$:= \top^+$

※

REMARK 72 Note that, given a syntactically consistent set $\mathcal{P}$ of literals, negations of $\mathcal{P}$-positive formulae are $\mathcal{P}$-negative and vice versa. ※

NOTATION 116 A possibly primed or indexed Γ always denotes a set of formulae. By Γ_{lit} we denote the subset of elements of Γ that are literals, and we write $l \in \Gamma$ if l or $l^\perp$ appears in Γ. By $\mathsf{lit}_{\mathcal{P}}(\Gamma)$ we denote the sub-multiset of Γ consisting of its $\mathcal{P}$-positive literals (i.e. $\mathcal{P} \cap \Gamma$ as a set). ※

[3] It is the predicate that is true of a set $\mathcal{P}$ of literals iff $\mathcal{P}$ contains both l and $l^\perp$ for some $l \in \mathcal{L}$.

Definition 117 (System $\mathsf{LK}^p(\mathcal{T})$)

The system $\mathsf{LK}^p(\mathcal{T})$ is the sequent calculus defined by the rules of Fig. 34, which fall into three categories: *synchronous*, *asynchronous*, and *structural* rules, and manipulate two kinds of sequents:
$\Gamma \vdash^{\mathcal{P}} [A]$ where the formula A is in the *focus* of the sequent
$\Gamma \vdash^{\mathcal{P}} \Gamma'$
where $\mathcal{P}$ is a syntactically consistent set of literals declared to be positive.
A sequent of the second kind where Γ' is empty is called *developed*. ※

Synchronous rules

$$(\wedge^+)\frac{\Gamma \vdash^{\mathcal{P}} [A] \qquad \Gamma \vdash^{\mathcal{P}} [B]}{\Gamma \vdash^{\mathcal{P}} [A \wedge^+ B]} \qquad (\vee^+)\frac{\Gamma \vdash^{\mathcal{P}} [A_i]}{\Gamma \vdash^{\mathcal{P}} [A_1 \vee^+ A_2]}$$

$$(\top^+)\frac{}{\Gamma \vdash^{\mathcal{P}} [\top^+]} \qquad (\mathsf{Init}_1)\frac{\mathsf{lit}_{\mathcal{P}}(\Gamma), l^\perp \models_{\mathcal{T}}}{\Gamma \vdash^{\mathcal{P}} [l]}\ l \in \mathcal{P} \qquad (\mathsf{Release})\frac{\Gamma \vdash^{\mathcal{P}} N}{\Gamma \vdash^{\mathcal{P}} [N]}\ N \text{ is not } \mathcal{P}\text{-positive}$$

Asynchronous rules

$$(\wedge^-)\frac{\Gamma \vdash^{\mathcal{P}} A, \Delta \qquad \Gamma \vdash^{\mathcal{P}} B, \Delta}{\Gamma \vdash^{\mathcal{P}} A \wedge^- B, \Delta} \qquad (\vee^-)\frac{\Gamma \vdash^{\mathcal{P}} A_1, A_2, \Delta}{\Gamma \vdash^{\mathcal{P}} A_1 \vee^- A_2, \Delta}$$

$$(\bot^-)\frac{\Gamma \vdash^{\mathcal{P}} \Delta}{\Gamma \vdash^{\mathcal{P}} \Delta, \bot^-} \qquad (\top^-)\frac{}{\Gamma \vdash^{\mathcal{P}} \Delta, \top^-} \qquad (\mathsf{Store})\frac{\Gamma, A^\perp \vdash^{\mathcal{P};A^\perp} \Delta}{\Gamma \vdash^{\mathcal{P}} A, \Delta}\ \begin{array}{l} A \text{ is a literal} \\ \text{or is } \mathcal{P}\text{-positive} \end{array}$$

Structural rules

$$(\mathsf{Select})\frac{\Gamma, P^\perp \vdash^{\mathcal{P}} [P]}{\Gamma, P^\perp \vdash^{\mathcal{P}}}\ P \text{ is not } \mathcal{P}\text{-negative} \qquad (\mathsf{Init}_2)\frac{\mathsf{lit}_{\mathcal{P}}(\Gamma) \models_{\mathcal{T}}}{\Gamma \vdash^{\mathcal{P}}}$$

where $\mathcal{P}; A := \mathcal{P}, A$ if $A \in \mathsf{U}_{\mathcal{P}}$
$\mathcal{P}; A := \mathcal{P}$ if not

Figure 34: System $\mathsf{LK}^p(\mathcal{T})$

The gradual proof-tree construction defined by the bottom-up application of the inference rules of $\mathsf{LK}^p(\mathcal{T})$, is a goal-directed mechanism whose intuition can be given as follows:

Asynchronous rules are invertible: $(\wedge^-)$ and $(\vee^-)$ are applied eagerly when trying to construct the proof-tree of a given sequent; (Store) is applied when hitting a positive formula or a negative literal on the right-hand side of a sequent, storing its negation on the left.

When the right-hand side of a sequent becomes empty (i.e. the sequent is *developed*), a sanity check can be made with (Init_2) to check the semantical consistency of the stored literals (w.r.t. the theory), otherwise a choice must be made to place a positive formula in focus, using rule (Select), before applying synchronous rules like $(\wedge^+)$ and $(\vee^+)$. Each such rule decomposes the formula in focus, keeping the revealed sub-formulae in the focus of the corresponding premises, until a positive literal or a non-positive formula is obtained: the former case must be closed immediately with (Init_1) calling the decision procedure, and the latter case uses the (Release) rule to drop the focus and start applying asynchronous

rules again. The synchronous and the structural rules are in general not invertible,[4] so each application of those yields in general a backtrack point in the proof-search.

Notice that an invariant of such a proof-tree construction process is that the left-hand side of a sequent only contains negative formulae and positive literals.

NOTATION 118 When F is a formula of unpolarised propositional logic and Ψ is a set of such formulae, $\Psi \models F$ means that Ψ entails F in propositional classical logic. Given a theory $\mathcal{T}$ (given by a semantical inconsistency predicate), we define the set of all *theory lemmas* as $\Psi_{\mathcal{T}} := \{l_1 \vee \cdots \vee l_n \mid l_1^{\perp}, \cdots, l_n^{\perp} \models_{\mathcal{T}}\}$ and generalise the notation $\models_{\mathcal{T}}$ to write $\Psi \models_{\mathcal{T}} F$ when $\Psi_{\mathcal{T}}, \Psi \models F$. In that case we say that F is a *semantical consequence* of Ψ. For any polarised formula A, let $\underline{A}$ be the unpolarised formula obtained by removing all polarities on connectives. ※

THEOREM 73 (Cut-elimination and Completeness of $\mathsf{LK}^p(\mathcal{T})$)

- The following rules are admissible in $\mathsf{LK}^p(\mathcal{T})$:

$$(\mathsf{Pol})\ \frac{\Gamma \vdash^{\mathcal{P}, l}}{\Gamma \vdash^{\mathcal{P}}}\ l \in \Gamma \text{ and } \mathsf{lit}_{\mathcal{P}}(\Gamma), l^{\perp} \models_{\mathcal{T}} \qquad (\mathsf{cut})\ \frac{\Gamma \vdash^{\mathcal{P}} l \quad \Gamma \vdash^{\mathcal{P}} l^{\perp}}{\Gamma \vdash^{\mathcal{P}}}\ l \in \Gamma$$

  provided the bottom sequent satisfies some property called *safety* [FGL13, Far13].
- If $\models_{\mathcal{T}} F$, then for all A such that $\underline{A} = F$, we can prove $\vdash^{\emptyset} A$ in $\mathsf{LK}^p(\mathcal{T})$.

※

The meta-theory of $\mathsf{LK}^p(\mathcal{T})$, in particular the proofs of the above, can be found in [FGL13, Far13].

8.2 Bisimulation with the DPLL($\mathcal{T}$) procedure

8.2.1 The elementary DPLL($\mathcal{T}$) procedure

Intuitively, DPLL($\mathcal{T}$) aims at proving the inconsistency of a set of *clauses* with respect to a theory. We therefore retain from the previous section the notion of literal and inconsistencies, and introduce clauses:

DEFINITION 119 (Clause)

A *clause* is a finite set of literals, which can be seen as their disjunction.

In the rest of the chapter, a possibly indexed upper cased C always denotes a clause. The empty clause is denoted by $\perp$. The number of literals in a clause C is denoted $\sharp(C)$. The possibly indexed symbol ϕ always denotes finite sets of clauses $\{C_1, \ldots, C_n\}$, which can also be seen as a Conjunctive Normal Form (CNF). We use $\sharp(\phi)$ to denote the sum of the sizes of the clauses in ϕ. Finally $\mathsf{lit}(\phi)$ denotes the set of literals that appear in ϕ or whose negations appear in ϕ.

Viewing clauses as disjunctions of literals and sets of clauses as CNF, we will generalise Notation 118, writing for instance $\phi \models C^{\perp}$ or $\phi \models C$, as well as $\phi \models_{\mathcal{T}} C^{\perp}$ or $\phi \models_{\mathcal{T}} C$. ※

[4] (but they may be so, e.g. $(\wedge^+)$)

DEFINITION 120 (Decision literals and sequences)

We consider a (single) copy of the set $\mathcal{L}$ of literals, denoted $\mathcal{L}^d$, whose elements are called *decision literals*, which are just tagged clones of the literals in $\mathcal{L}$. Decision literals are denoted[5] by l^d.

We use the possibly indexed symbol Δ to denote a finite sequence of possibly tagged literals, with $\emptyset$ denoting the empty sequence. We also use Δ_1, Δ_2 and Δ_1, l, Δ_2 to denote the suggested concatenation of sequences.

For such a sequence Δ, we write $|\Delta|$ for the subset of $\mathcal{L}$ containing all the literals in Δ with their potential tags removed. The sequences that DPLL($\mathcal{T}$) will construct will always be duplicate-free, so the difference between Δ and $|\Delta|$ is just a matter of tags and ordering. When the context is unambiguous, we will sometimes use Δ when we mean $|\Delta|$.

We define $\mathsf{Sat}(\Delta) := \{l \mid \Delta, l^\perp \models_\mathcal{T}\}$, the closure of a sequence Δ by semantical entailment. For any set of clauses ϕ, the set of literals occuring in ϕ that are semantically entailed by Δ is denoted by $\mathsf{Sat}_\phi(\Delta) := \mathsf{Sat}(\Delta) \cap \mathsf{lit}(\phi)$. ※

REMARK 74 Semantical consequences are the analogues of the consequences of a partial boolean assignment in the context of a DPLL procedure for propositional logic without theory.
Obviously, if $l \in \Delta$, then $l \in \mathsf{Sat}(\Delta)$. If $\phi_1 \subseteq \phi_2$, then for any Δ, $\mathsf{Sat}_{\phi_1}(\Delta) \subseteq \mathsf{Sat}_{\phi_2}(\Delta)$. ※

We can now describe the elementary DPLL($\mathcal{T}$) procedure as a transition system between states.

DEFINITION 121 (Elementary DPLL($\mathcal{T}$))

A *state* of the DPLL($\mathcal{T}$) procedure is either the state UNSAT, or a pair denoted $\Delta \| \phi$, where ϕ is a set of clauses and Δ is a sequence of possibly tagged literals. The *transition rules* of the elementary DPLL($\mathcal{T}$) procedure are given in Fig. 35. ※

Decide	$\Delta \| \phi$	$\Rightarrow \Delta, l^d \| \phi$	where $l \in \mathsf{lit}(\phi)$	and $l \notin \Delta$ and $l^\perp \notin \Delta$.
Propagate	$\Delta \| \phi, C \vee l$	$\Rightarrow \Delta, l \| \phi, C \vee l$	where $\Delta \models C^\perp$	and $l \notin \Delta$ and $l^\perp \notin \Delta$.
Propagate$_\mathcal{T}$	$\Delta \| \phi$	$\Rightarrow \Delta, l \| \phi$	where $l \in \mathsf{Sat}_\phi(\Delta)$	and $l \notin \Delta$ and $l^\perp \notin \Delta$.
Fail	$\Delta \| \phi, C$	$\Rightarrow$ UNSAT,	where $\Delta \models C^\perp$	and there is no decision literal in Δ.
Fail$_\mathcal{T}$	$\Delta \| \phi$	$\Rightarrow$ UNSAT,	where $\Delta \models_\mathcal{T}$	and there is no decision literal in Δ.
Backtrack	$\Delta_1, l^d, \Delta_2 \| \phi, C$	$\Rightarrow \Delta_1, l^\perp \| \phi, C$	where $\Delta_1, l, \Delta_2 \models C^\perp$	and there is no decision literal in Δ_2.
Backtrack$_\mathcal{T}$	$\Delta_1, l^d, \Delta_2 \| \phi$	$\Rightarrow \Delta_1, l^\perp \| \phi$	where $\Delta_1, l, \Delta_2 \models_\mathcal{T}$	and there is no decision literal in Δ_2.

Figure 35: Elementary DPLL($\mathcal{T}$)

This transition system is an extension of the *Classical DPLL procedure*, as presented in [NOT06], to the background theory $\mathcal{T}$.[6] The first four rules are explicitly taken from the Abstract DPLL Modulo Theories system of [NOT06].[7] The other rules of that system (namely $\mathcal{T}$-Backjump, $\mathcal{T}$-Learn, $\mathcal{T}$-Forget, etc), are not considered here in their full generality, but specific cases and combinations are covered by the rest of our elementary DPLL($\mathcal{T}$) sys-

[5] This exponent tag is a standard notation, standing for "decision".
[6] We removed the Pure Literal rule, in general unsound in presence of a theory $\mathcal{T}$.
[7] Unit Propagate and Theory Propagate are renamed as Propagate and Propagate$_\mathcal{T}$ for consistency with the other rule names.

tem, so that it is logically complete.[8] Note that this transition system is not deterministic: for instance the Decide rule can be applied from any state and it furthermore does not enforce a strategy for picking the literal to be tagged among the eligible elements of $\mathsf{lit}(\phi)$. At the level of implementation, this (non deterministic) transition system is turned into a deterministic algorithm, whose efficientcy crucially relies on the strategies adopted to perform the choices left unspecified by DPLL($\mathcal{T}$).

We illustrate those rules, in the theory $\mathcal{T}$ of *Linear Rational Arithmetic*, with the two basic examples of elementary DPLL($\mathcal{T}$) runs presented in Fig. 36 (where Δ and ϕ always refer to the current state $\Delta \| \phi$).

$\emptyset$	$x>0, (x+y>0)^\bot, (y>0 \vee x=-1)$	Propagate
$x>0$	$x>0, (x+y>0)^\bot, (y>0 \vee x=-1)$	Propagate
$x>0, (x+y>0)^\bot$	$x>0, (x+y>0)^\bot, (y>0 \vee x=-1)$	Propagate$_\mathcal{T}$ $((y>0)^\bot \in \mathsf{Sat}_\phi(\Delta))$
$x>0, (x+y>0)^\bot, (y>0)^\bot$	$x>0, (x+y>0)^\bot, (y>0 \vee x=-1)$	Propagate$_\mathcal{T}$ $((x=-1)^\bot \in \mathsf{Sat}_\phi(\Delta))$
$x>0, (x+y>0)^\bot, (y>0)^\bot, (x=-1)^\bot$	$x>0, (x+y>0)^\bot, (y>0 \vee x=-1)$	Fail on clause $(y>0 \vee x=-1)$
UNSAT		

$\emptyset$	$x>0, (x+y>0)^\bot, (y>0 \vee x=-1)$	Propagate
$x>0$	$x>0, (x+y>0)^\bot, (y>0 \vee x=-1)$	Propagate
$x>0, (x+y>0)^\bot$	$x>0, (x+y>0)^\bot, (y>0 \vee x=-1)$	Propagate$_\mathcal{T}$ $((y>0)^\bot \in \mathsf{Sat}_\phi(\Delta))$
$x>0, (x+y>0)^\bot, (y>0)^\bot$	$x>0, (x+y>0)^\bot, (y>0 \vee x=-1)$	Propagate $(x=-1)$ from $(y>0 \vee x=-1)$
$x>0, (x+y>0)^\bot, (y>0)^\bot, (x=-1)$	$x>0, (x+y>0)^\bot, (y>0 \vee x=-1)$	Fail$_\mathcal{T}$ $x>0, x=-1$ inconsistent with $\mathcal{T}$
UNSAT		

Figure 36: Examples of elementary DPLL($\mathcal{T}$) runs

A reason to introduce rule Fail$_\mathcal{T}$ is to allow the second run to finish with the same output as the first: Indeed, the last Propagate step has created a $\mathcal{T}$-inconsistency from which we could not derive UNSAT without a Fail$_\mathcal{T}$ step.[9]

8.2.2 Simulation of the elementary DPLL($\mathcal{T}$) procedure in LK^p($\mathcal{T}$)

The aim of this section is to describe how the elementary DPLL($\mathcal{T}$) procedure can be transposed into a proof-search process for sequents of the LK^p($\mathcal{T}$) calculus. A complete and successful run of the DPLL($\mathcal{T}$) procedure is a sequence of transitions $\emptyset \| \phi \Rightarrow^* \mathsf{UNSAT}$, which ensures that the set of clauses ϕ is inconsistent modulo the theory. Hence, we are devising a proof-search process aiming at building an LK^p($\mathcal{T}$) proof-tree for sequents of the form $\phi' \vdash$, where ϕ' represents the set of clauses ϕ as a sequent calculus structure, in the following

[8] Backtrack is a restricted version of $\mathcal{T}$-Backjump (this holds on the basis that the full system satisfies some basic invariant -Lemma 3.6 of [NOT06]), Fail$_\mathcal{T}$ (resp. Backtrack$_\mathcal{T}$) is a combination of $\mathcal{T}$-Learn, Fail (resp. Backtrack), and $\mathcal{T}$-Forget steps.

[9] (or, alternatively, a $\mathcal{T}$-Learn step in [NOT06])

sense:

Definition 122 (Representation of clauses as formulae)

An $\mathsf{LK}^p(\mathcal{T})$ formula C' *represents* a $\mathsf{DPLL}(\mathcal{T})$ clause $\{l_j\}_{j=1\ldots p}$ if $C' = l_1 \vee^- \ldots \vee^- l_p \vee^- \bot^-$.

A set of formulae ϕ' *represents* a set of clauses ϕ if there is a bijection f from ϕ to ϕ' such that for all clauses C in ϕ, $f(\phi)$ represents C. ※

Remark 75 If C' represents C, then $\sharp(C') \leq 2\sharp(C)$ (there are fewer symbols $\vee^-$ than there are literals in C). ※

Note here that we carefully use the negative disjunction connective to translate $\mathsf{DPLL}(\mathcal{T})$ clauses. This is crucial not only to mimic $\mathsf{DPLL}(\mathcal{T})$ without duplicating formulae but more generally to control the search space.

Now, in order to construct a proof of $\phi' \vdash$ from a run $\emptyset \| \phi \Rightarrow^* \mathsf{UNSAT}$, we proceed gradually by considering the intermediate steps of the $\mathsf{DPLL}(\mathcal{T})$ run:

$$\emptyset \| \phi \Rightarrow^* \Delta \| \phi \Rightarrow^* \mathsf{UNSAT}$$

In the intermediate $\mathsf{DPLL}(\mathcal{T})$ state $\Delta \| \phi$, the sequence Δ is a log of both the search space explored so far (in $\emptyset \| \phi \Rightarrow^* \Delta \| \phi$) and the search space that remains to be explored (in $\Delta \| \phi \Rightarrow^* \mathsf{UNSAT}$). In this log, a tagged decision literal l^d indicates a point where the procedure has made an exploratory choice (the case where l is true has been/is being explored, the case where $l^\bot$ is true remains to be explored), while untagged literals in Δ are predictable consequences of the decisions made so far and of the set of clauses ϕ to be falsified.

If we are to express the $\mathsf{DPLL}(\mathcal{T})$ procedure as the gradual construction of a $\mathsf{LK}^p(\mathcal{T})$ proof-tree, we should get from $\emptyset \| \phi \Rightarrow^* \Delta \| \phi$ a proof-tree that is not yet complete and get from $\Delta \| \phi \Rightarrow^* \mathsf{UNSAT}$ some (complete) proof-tree(s) that can be "plugged into the holes" of the incomplete tree. We should read in Δ the "interface" between the incomplete tree that has been constructed and the complete sub-trees to be constructed.

We use the plural here since there can be more than one sub-tree left to construct: $\Delta \| \phi \Rightarrow^* \mathsf{UNSAT}$ contains the information to build not only a proof of $|\Delta|, \phi' \vdash$, but also proofs of the sequents corresponding to the other parts of the search space to be explored, characterised by the tagged literals in Δ. For instance, a run from $l_1, l_2^d, l_3, l_4^d \| \phi \Rightarrow^* \mathsf{UNSAT}$ contains the information to build a proof of $l_1, l_2, l_3, l_4, \phi' \vdash$ but also the proofs of $l_1, l_2, l_3, l_4^\bot, \phi' \vdash$ and $l_1, l_2^\bot, \phi' \vdash$. Those extra sequents are obtained by collecting from a sequence Δ its "backtrack points" as follows:

Definition 123 (Backtrack points)

The *backtrack points* $[\![\Delta]\!]$ of a sequence Δ of possibly tagged literals is the list of sets of untagged literals recursively defined by the following rules, where $[\,]$ and :: are the standard list constructors.

$$\begin{array}{lcl} [\![()]\!] & := & [\,] \\ [\![\Delta, l]\!] & := & [\![\Delta]\!] \\ [\![\Delta, l^d]\!] & := & |\Delta, l^\bot| :: [\![\Delta]\!] \end{array}$$

※

Remark 76 The length of $[\![\Delta]\!]$ is the number of decision literals in Δ. ※

Now, coming back to the $\mathsf{DPLL}(\mathcal{T})$ transition sequence $\emptyset \| \phi \Rightarrow^* \Delta \| \phi$ and its intuitive counterpart in sequent calculus, we have to formalise the notion of *incomplete* proof-tree together with the notion of "filling its holes":

DEFINITION 124 (Incomplete proof-tree, extension)

An *incomplete proof-tree* in $\mathsf{LK}^p(\mathcal{T})$ is a tree labelled with sequents,

- whose leaves are tagged as either *open* or *closed*;
- whose open leaves are labelled with developed sequents;
- and such that every node that is not an open leaf, together with its children, forms an instance of the $\mathsf{LK}^p(\mathcal{T})$ rules.

The *size* of an incomplete proof-tree is its number of nodes.

An incomplete proof-tree π' is an *extension* of π, if there is a tree (edge and nodes preserving) homomorphism from π to π'. It is an *n-extension* of π, if moreover the difference of size between π' and π is less than or equal to n. ※

REMARK 77 An incomplete proof-tree that has no open leaf is (isomorphic to) a well-formed complete $\mathsf{LK}^p(\mathcal{T})$ proof of the sequent labelling its root. In that case, we say the proof-tree is *complete*. ※

The intuition that an intermediate $\mathsf{DPLL}(\mathcal{T})$ state describes an "interface" between an incomplete proof-tree and the complete proof-trees that should be plugged into its holes, is formalised as follows:

DEFINITION 125 (Correspondence)

An incomplete proof-tree π *corresponds to* a $\mathsf{DPLL}(\mathcal{T})$ state $\Delta \| \phi$ if:

- the length of $|\Delta| :: [\![\Delta]\!]$ is the number of open leaves of π;
- if Δ_i is the i^{th} element of $|\Delta| :: [\![\Delta]\!]$, then the i^{th} open leaf of π (taken left-to-right) is labelled by a developed sequent of the form $\Delta'_i, \phi'_i \vdash^{\Delta_i}$, where:
  - ϕ'_i represents ϕ (in the sense of Definition 122);
  - $\mathsf{Sat}_\phi(\Delta_i) = \mathsf{Sat}_\phi(\Delta'_i)$.

An incomplete proof-tree π *corresponds to* the state UNSAT if it has no open leaf. ※

REMARK 78 In the general case, different incomplete proof-trees might correspond to the same $\mathsf{DPLL}(\mathcal{T})$ state (just like different $\mathsf{DPLL}(\mathcal{T})$ runs may reach that state from the initial one).

Note that we do not require anything from the conclusion of an incomplete proof-tree corresponding to $\Delta \| \phi$: just as our correspondence says nothing about the $\mathsf{DPLL}(\mathcal{T})$ transitions taking place after $\Delta \| \phi$ (nor about the trees to be plugged into the open leaves), it says nothing about the transitions taking place before $\Delta \| \phi$ (nor about the incomplete proof-tree, except for its open leaves).

If an incomplete proof-tree π corresponds to a $\mathsf{DPLL}(\mathcal{T})$ state $\Delta \| \phi$ where there are no decision literals in Δ, then there is exactly one open leaf in π, and it is labelled by a sequent of the form $\Delta', \phi' \vdash^{|\Delta|}$, where ϕ' represents ϕ and $\mathsf{Sat}_\phi(\Delta) = \mathsf{Sat}_\phi(\Delta')$.

To the initial state $\emptyset \| \phi$ of a run of the $\mathsf{DPLL}(\mathcal{T})$ procedure corresponds the incomplete proof-tree consisting of one node (both root and open leaf) labelled with the sequent $\phi' \vdash$, where ϕ' represents ϕ. ※

The simulation theorem below provides a systematic way of interpreting any $\mathsf{DPLL}(\mathcal{T})$ transition as a completion of incomplete proof-trees that preserves the correspondence given in Definition 125 and controls the growth of the proof trees.

Theorem 79 (Simulation of DPLL($\mathcal{T}$) in LK^p($\mathcal{T}$))

If $\Delta \| \phi \Rightarrow \mathcal{S}_2$ is a valid DPLL($\mathcal{T}$) transition, and π_1 is an incomplete proof tree in LK^p($\mathcal{T}$) corresponding to $\Delta \| \phi$, then there exists a $(2\sharp(\phi) + 3)$-extension π_2 of π_1 that corresponds to $\mathcal{S}_2$. ※

Proof: See [FGLM13, Far13]. By case analysis on the nature of the transition, completing the leftmost open leaf of π_1. Basically:

- Fail using clause C corresponds to Select on $C^\perp$
- Fail$_\mathcal{T}$ corresponds to Init$_2$ rule
- Backtrack using clause C corresponds to Select on $C^\perp$
- Backtrack$_\mathcal{T}$ corresponds to Init$_2$ rule
- Propagate using clause C corresponds to Select on $C^\perp$
- Fail$_\mathcal{T}$ corresponds to Pol rule
- Decide corresponds to cut rule

□

Corollary 80

If $\emptyset \| \phi \Rightarrow^n \mathsf{UNSAT}$ and ϕ' represents ϕ then there is a complete proof in LK^p($\mathcal{T}$) of $\phi' \vdash$, of size smaller than $(2\sharp(\phi) + 3)n$. ※

8.2.3 Completing the bisimulation

Now the point of having mentioned quantitative information in Theorem 79, via the notion of n-extension, is to motivate the idea that performing proof-search directly in LK^p($\mathcal{T}$) is in essence not less efficient than running DPLL($\mathcal{T}$): we have a linear bound in the length of the DPLL($\mathcal{T}$) run (and the proportionality ratio is itself an affine function of the size of the original problem).

We also need to make sure that this final proof-tree is indeed found as efficiently as running DPLL($\mathcal{T}$), which can be done by identifying, in LK^p($\mathcal{T}$), a (complete) search space that is isomorphic to (and hence no wider than) that of DPLL($\mathcal{T}$). We analyse for this a proof-search strategy, in LK^p($\mathcal{T}$), that exactly captures the proof-extensions that we have used in the simulation of DPLL($\mathcal{T}$), i.e. the proof of Theorem 79:

DEFINITION 126 (DPLL($\mathcal{T}$)-extensions)

An incomplete proof tree π_2 is a *DPLL($\mathcal{T}$)-extension* of an incomplete proof tree π_1 if

1. it extends π_1 by replacing its leftmost open leaf with an incomplete proof-tree of one of the forms:

$$\dfrac{\dfrac{\cdots}{\Gamma, A^\perp \vdash^{\mathcal{P}} [A]}\,(b)}{\Gamma, A^\perp \vdash^{\mathcal{P}}}\,(a) \qquad \dfrac{\Gamma \vdash^{\mathcal{P}} l \quad \Gamma \vdash^{\mathcal{P}} l^\perp}{\Gamma \vdash^{\mathcal{P}}}\, l \in \Gamma$$

$$\dfrac{\Gamma \vdash^{\mathcal{P}, l}}{\Gamma \vdash^{\mathcal{P}}}\,(c) \qquad \dfrac{}{\Gamma \vdash^{\mathcal{P}}}\, \Gamma_{\mathsf{lit}} \models_{\mathcal{T}}$$

   where
   (a) A is a (positive) conjunction of literals that are all in $\mathcal{P}$ except maybe one that is $\mathcal{P}$-unpolarised
   (b) the only instances of (Pol) in the above proof are of the form $\dfrac{\Gamma \vdash^{\mathcal{P}, l^\perp} l}{\Gamma \vdash^{\mathcal{P}} l}$
   (c) $l \in \Gamma$ with $\Gamma_{\mathsf{lit}}, l^\perp \models_{\mathcal{T}}$
2. any incomplete proof-tree satisfying point 1. and extended by π_2 is π_2 itself.

※

Given a DPLL($\mathcal{T}$)-extension, we can now identify a DPLL($\mathcal{T}$) transition that the extension simulates, in the sense of Theorem 79:

THEOREM 81 (Simulation of the strategy back into DPLL($\mathcal{T}$))

If π_2 is a DPLL($\mathcal{T}$)-extension of π_1, and π_1 corresponds to $\Delta \| \phi$, then there is a (unique) DPLL($\mathcal{T}$) transition $\Delta \| \phi \Rightarrow \mathcal{S}_2$ such that π_2 corresponds to $\mathcal{S}_2$. ※

Proof: See [FGLM13, Far13]. □

If a complete proof-tree of LK^p($\mathcal{T}$), whose conclusion is an SMT-problem,[10] systematically uses the rules in the way described by the above shapes, then it is the image of a DPLL($\mathcal{T}$) run.

While it could be envisaged to simulate DPLL($\mathcal{T}$) in a Gentzen-style sequent calculus (with a variant of Theorem 79), the above definition and theorem reveal the advantage of using a focused sequent calculus for polarised logic: Definition 126 presents[11] different ways of *starting* the extension of an open branch (whose leaf sequent is developed), each one of them corresponding to a specific DPLL($\mathcal{T}$) transition; then *focussing* takes care of the following steps of the extension so that, when hitting developed sequents again, the exact simulation of the DPLL($\mathcal{T}$) transition has been performed.

In order for proof-search mechanisms to exactly match DPLL($\mathcal{T}$) transitions, focussing therefore provides the right level of granularity and (together with an appropriate management of polarities) the right level of determinism.

[10] i.e. it corresponds to an initial state of DPLL($\mathcal{T}$)
[11] mostly by specifying the management of polarities

Corollary 82 (Bisimulation) The correspondance relation (see Definition 125) between incomplete proof trees and DPLL($\mathcal{T}$) states is a bisimulation for the transition system defined on incomplete proof-trees of LK^p($\mathcal{T}$) by the strategy of DPLL($\mathcal{T}$)-extensions and on states by DPLL($\mathcal{T}$). ※

8.2.4 More advanced features

Finally, obtaining this tight result is the reason why we identified the *elementary DPLL($\mathcal{T}$)* system, a restriction of the Abstract DPLL Modulo Theories system of [NOT06]:

Modern SMT-solvers feature some mechanisms that are not part of our (logically complete) *elementary DPLL($\mathcal{T}$)* system but increase efficiency, such as *backjumping* and *lemma learning* (cf. rules $\mathcal{T}$-Backjump, $\mathcal{T}$-Learn in [NOT06]).

It is possible to simulate those rules in LK^p($\mathcal{T}$) by using general cuts, by extending with identical steps several open branches of incomplete proof-trees, and possibly by using explicit weakenings (depending on whether we adapt the correspondence between DPLL($\mathcal{T}$) states and incomplete proof-trees). Again, the details of this can be found in [FGLM13, Far13].

However, with such "parallel extensions" of incomplete proof-trees, it is not clear how to count the *sizes* of proofs and extensions in a meaningful way, so the quantitative aspects of Theorem 79 and Corollary 80 are compromised; neither is it clear which criterion on proof-trees (and on how to extend them) identifies the proof-construction strategy that is the exact image of a DPLL($\mathcal{T}$) procedure featuring those advanced mechanisms. In other words, it is not clear how to obtain such a tight correspondence.

Nonetheless, understaning backjumping and lemma learning in terms of "parallel extensions" of incomplete proof-trees, gives some concrete leads on how to integrate these features to a root-first proof-search procedure as described in this chapter. One of them is to use *memoisation* for the proof-search function. This is used to close, in one single step, any branch that would otherwise be closed by repeating the same steps as in a subproof that has already been found. In particular, doing this avoids repeating, several times, the proof-construction steps of a "parallel extension" corresponding to a single backjump.

Memoisation is also a way of performing clause-learning: a learnt clause C is a clause for which we know that $\phi \models_{\mathcal{T}} C$, and that is made available for Fail, Backtrack or Propagate. Such a clause corresponds to a key $\phi', C^{\perp} \vdash$ of the memoisation table, with its proof as value. A state where C can be used for Fail or Backtrack is necessarily a sequent weakening $\phi', C^{\perp} \vdash$ with extra formulae or literals, so the proof recorded in the memoisation table can be plugged there to close the current branch. When C can be used for Propagate, it suffices to make a cut on the missing literal: one branch will be closed by plugging-in the proof recorded in the memoisation table, while the other branch will continue the simulation.

In the next chapter, we expand on the implementation of the above results in the form of a specific *plugin* for the Psyche system.

8.3 Future work: Relation to abstract focussing

In this section, we give some hints as to how we could develop, in the abstract LAF system, the methodology of simulating DPLL($\mathcal{T}$) as bottom-up proof-search in a focussed sequent calculus. We already know that we can capture LKF as the LAF instance LAF$_{K1}$. Since we

$$\frac{}{\bullet \Vdash \bullet : (\top^+, \mathcal{P})} \qquad \frac{}{\sim(N^\perp, \mathcal{P}) \Vdash \_^- : (N, \mathcal{P})}\ N \text{ is } \mathcal{P}\text{-negative} \qquad \frac{}{a \Vdash \_^+ : (a, \mathcal{P})}\ a \in \mathcal{P}$$

$$\frac{\Delta_1 \Vdash p_1 : (A_1, \mathcal{P}) \quad \Delta_2 \Vdash p_2 : (A_2, \mathcal{P})}{\Delta_1, \Delta_2 \Vdash (p_1, p_2) : (A_1 \wedge^+ A_2, \mathcal{P})} \qquad \frac{\Delta \Vdash p : (A_i, \mathcal{P})}{\Delta \Vdash \mathsf{inj}_i(p) : (A_1 \vee^+ A_2, \mathcal{P})}$$

Figure 37: Decomposition relation for LAF_{K1p}

have slightly modified LKF for the purpose of capturing DPLL($\mathcal{T}$), it is natural to ask whether that fits the LAF framework as well.

8.3.1 On-the-fly polarisation

The first difference between LKF and $\mathsf{LK}^p(\mathcal{T})$, is that we have on-the-fly polarisation of atoms. This is a feature that can easily be integrated as another LAF instance:

DEFINITION 127 (The LAF instance for on-the-fly polarisation)

The definition of the instance LAF_{K1p} is the same as that of LAF_{K1}, except that

- molecules are now pairs $(A, \mathcal{P})$ made of a formula A and a polarisation set $\mathcal{P}$ such that A is $\mathcal{P}$-positive;
- atoms are literals.

The decomposition relation is defined in Fig. 37 (which adapts Fig. 24).

Typing contexts are defined similarly to Definition 69, but with the extra information about polarities:
A typing context is given by $(\Gamma^+, \Gamma^-, \mathcal{P})$, where Γ^+ is a list of literals and Γ^- is a list of formulae;
$(\Gamma^+, \Gamma^-, \mathcal{P})\,[n^+]$ is the $(n+1)^{th}$ element of Γ^+
$(\Gamma^+, \Gamma^-, \mathcal{P})\,[n^-]$ is $(A, \mathcal{P})$, where A is the $(n+1)^{th}$ element of Γ^-.
Context extension updates $\mathcal{P}$ just as in rule Store of $\mathsf{LK}^p(\mathcal{T})$, so that (for instance)

$$(\Gamma^+, \Gamma^-, \mathcal{P}); a = ((a :: \Gamma^+), \Gamma^-, (\mathcal{P}; a))$$

※

That instance being defined, it is however not completely clear how to integrate to LAF_{K1p} the two admissible rules of $\mathsf{LK}^p(\mathcal{T})$ (at least in their current form):

$$(\mathsf{Pol})\ \frac{\Gamma \vdash^{\mathcal{P}, l}}{\Gamma \vdash^{\mathcal{P}}}\ l \in \Gamma \text{ and } \mathsf{lit}_{\mathcal{P}}(\Gamma), l^\perp \models_{\mathcal{T}} \qquad (\mathsf{cut})\ \frac{\Gamma \vdash^{\mathcal{P}} l \quad \Gamma \vdash^{\mathcal{P}} l^\perp}{\Gamma \vdash^{\mathcal{P}}}\ l \in \Gamma$$

and how to use them in a bottom-up proof-search procedure based on LAF_{K1p}. This is future work.

8.3.2 Extending LAF to LAF($\mathcal{T}$)

The second difference between LKF and $\mathsf{LK}^p(\mathcal{T})$, is of course the theory $\mathcal{T}$ and its decision procedure, used in rules Init_1 and Init_2.

Notice that LAF can accommodate a weak form of "modulo theory", at least according to the definition of Chapter 5: The equality on atoms is a parameter that we can use to identify for instance $a(3+4)$ with $a(7)$, in particular in the rule typing positive labels

$$\frac{\Gamma\left[x^+\right] \equiv (a, \mathbf{r})}{\Gamma \vdash x^+ : (a, \mathbf{r})}\ \mathsf{init}$$

However in LAF, we cannot close a branch in one step by involving several atoms of Γ, as we do for instance in $\mathsf{LK}^p(\mathcal{T})$ when we call e.g. a simplex algorithm to check the consistency of (the positive literals of) Γ (which in LAF would be Γ^+).

For this we would need to extend LAF with a decision procedure. We could think of doing it in the following way:

- replace the notion of positive label by a notion of *focussed justification*, and abstract away the part Γ^+ of a typing context Γ, which is no longer a function from positive labels to instantiated atoms but an abstract data structure called a *positive typing context*;
- replace the notion of equality between instantiated atoms by a typing relation of the form $\Gamma^+ \models [s^+ : (a, \mathbf{r})]$, where Γ^+ is a positive typing context, and s^+ is a positive justification.
- add a notion of *justification* and a typing relation of the form $\Gamma^+ \models s$, where Γ^+ is a positive typing context, and s is a justification.

  We would then get:

Definition 128 (Proof-Terms) Let $\mathbb{J}$ be a set of elements called *justifications*, and $\mathbb{J}^+$ be a set of elements called *focussed justifications*.

Proof-terms are defined by the following syntax:

Positive terms	Terms^+	$t^+ ::= pd$
Decomposition terms	$\mathsf{Terms}^{\mathsf{d}}$	$d ::= s^+ \mid f \mid \bullet \mid d_1, d_2 \mid r.d$
Commands	Terms	$c ::= \{s\} \mid \langle x^- \mid t^+ \rangle \mid \langle f \mid t^+ \rangle$

where p ranges over patterns, s ranges over justifications, s^+ ranges over focused justifications, x^- ranges over Lab_-, f ranges over functions from patterns to commands. ※

And now we can give the LAF system parameterised by a "theory" given by the pair of typing relations $\_ \models [\_ : \_]$ and $\_ \models \_$, which we may call $\mathcal{T}$, and which plays the same role as the semantical inconsistency predicate in $\mathsf{LK}^p(\mathcal{T})$.

Definition 129 ($\mathsf{LAF}(\mathcal{T})$) Let Co^+ be the family of positive typing contexts.

Assume we are given a pair $\mathcal{T}$ of two relations
$(\_ \models [\_ : \_]) : (\mathsf{Co}^+ \times \mathbb{J}^+ \times \mathbb{A}_\downarrow)$ and $(\_ \models \_) : (\mathsf{Co}^+ \times \mathbb{J})$.

We define in Fig. 38 the derivability of three typing judgements

- $(\_ \vdash [\_ : \_]) \quad : \quad (\mathsf{Co} \times \mathsf{Terms}^+ \times \mathbb{M}_\downarrow)$
- $(\_ \vdash \_ : \_) \quad : \quad (\mathsf{Co} \times \mathsf{Terms}^{\mathsf{d}} \times \mathbb{D}_\downarrow)$
- $(\_ \vdash \_) \quad : \quad (\mathsf{Co} \times \mathsf{Terms})$

※

The empty theory could be recovered by having positive labels, having positive typing contexts as maps from positive labels to instantiated atoms, having focussed justifications be exactly positive labels, and by setting setting $\Gamma^+ \models [s^+ : (a, \mathbf{r})]$ if and only if $\Gamma^+(s^+) \equiv (a, \mathbf{r})$ and never having $\Gamma^+ \models s$.

Linear arithmetic could be defined by a relation $\Gamma^+ \models s$ that would check the consistency of the instantiated atoms in Γ^+, and a relation $\Gamma^+ \models [s^+ : (a, \mathbf{r})]$ that would check the consistency of the instantiated atoms in Γ^+ together with $(a^\perp, \mathbf{r})$.

$$\frac{\Delta \Vdash p : M \quad \Gamma \vdash d : (\Delta, \mathbf{r})}{\Gamma \vdash [pd : (M, \mathbf{r})]}\ \mathsf{sync}$$

$$\frac{}{\Gamma \vdash \bullet : (\bullet, \mathbf{r})} \qquad \frac{\Gamma \vdash d_1 : (\Delta_1, \mathbf{r}) \quad \Gamma \vdash d_2 : (\Delta_2, \mathbf{r})}{\Gamma \vdash d_1, d_2 : ((\Delta_1, \Delta_2), \mathbf{r})} \qquad \frac{\Gamma^e \Vdash r' : s \quad \Gamma \vdash d : (\Delta, r' :: \mathbf{r})}{\Gamma \vdash r'.d : s.(\Delta, \mathbf{r})}$$

$$\frac{\Gamma^+ \models [s^+ : (a, \mathbf{r})]}{\Gamma \vdash s^+ : (a, \mathbf{r})}\ \mathsf{Init}_1 \qquad \frac{\forall p, \forall \Delta, \quad \Delta \Vdash p : M \quad \Rightarrow \quad \Gamma; (\Delta, \mathbf{r}) \vdash f(p)}{\Gamma \vdash f : (\sim M, \mathbf{r})}\ \mathsf{async}$$

$$\frac{\Gamma^+ \models s}{\Gamma \vdash \{s\}}\ \mathsf{Init}_2 \qquad \frac{\Gamma \vdash [t^+ : \Gamma\,[x^-]]}{\Gamma \vdash \left\langle x^- \mid t^+ \right\rangle}\ \mathsf{Select} \qquad \frac{\Gamma \vdash f : (\sim M, \mathbf{r}) \qquad \Gamma \vdash [t^+ : (M, \mathbf{r})]}{\Gamma \vdash \left\langle f \mid t^+ \right\rangle}\ \mathsf{cut}$$

Figure 38: LAF($\mathcal{T}$)

For congruence closure we could have the same approach, or we could give a special role to $(a, \mathbf{r})$ in defining when $\Gamma^+ \models [s^+ : (a, \mathbf{r})]$ holds.

The abstract focussing system could be seen as a *functor* (in the programming language sense) $\mathcal{T} \mapsto \mathsf{LAF}(\mathcal{T})$ that takes a pair of typing relations (focussed, unfocussed) and returns a new pair of typing relations (focussed, unfocussed). In that view, the functor could be composed with others, and iterated. We conjecture that second-order logic or higher-order logic could be captured by the fixpoint of this functor, together with one that can convert an atom into a molecule, etc.

In every theory, the justifications could be dummy objects, if we do not have proof objects to produce when running the decision procedure. Or they could be as informative as one would like; in particular, it would be useful if s (resp. s^+) could at least indicate which part of Γ^+ is actually used to derive $\Gamma^+ \models [s^+ : (a, \mathbf{r})]$ (resp. $\Gamma^+ \models s$). This could be done via a notion of free labels, so that we can apply the same methodology as that of Section 7.3 to re-use proofs in different contexts.

The formal study of such a LAF system, together with the adaptation of its realisability models, is left for future work.

Chapter 9

The PSYCHE system

Contents

In this chapter, we describe PSYCHE [Psy], a system programmed in OCaml that implements, among other things, the ideas developed in the previous chapter(s). In particular, it uses polarities and focussing as a way to organise proof-search.

PSYCHE is a highly modular proof-search engine designed as a platform for either interactive or automated theorem proving, and the acronym stands for the *Proof-Search factorY for Collaborative Heuristics*. By platform, we mean that its architecture is organised around a *kernel* that interacts with *plugins* to be programmed via a specific API. The goal of this architecture is to allow the implementation of various theorem proving techniques while guaranteeing correctness of the output: whether an input formula is provable or not provable. As a platform, it can also be used to implement the collaboration of various techniques which, once programmed as plugins, share the same notion of *proof-search state*.

The aim is therefore to provide a high level of confidence about the output of the theorem proving process, no matter how programmers have implemented their plugins, which is done by adopting and somewhat transforming the LCF architecture [GMW79].

Finally, PSYCHE features the ability to call *decision procedures* such as those used in Sat-Modulo-Theories provers. We therefore illustrate PSYCHE by using it for SMT-solving.

In brief:

- The kernel is based on a proof-search engine *à la* Prolog, offering an API to perform incremental and goal-directed constructions of proof-trees in a focussed Sequent Calculus, which can be seen as a *tableaux method* [DGHP99].
- PSYCHE can produce proof objects.

- Plugins can be programmed to drive the kernel, using its API, through the search space towards an answer *provable* or *not provable*; correctness of the answer only relies on the kernel via the use of a private type for answers (similar to LCF's *theorem* type).
- Plugins can be interactive.
- PSYCHE offers a *memoisation* feature to help programming efficient plugins.
- The kernel is parameterised by a procedure deciding the consistency of collections of literals with respect to a background theory, just as in SAT-modulo-theories (SMT) solvers.

The current version 2.0 of PSYCHE is distributed

- with a kernel designed for first-order logic modulo a theory $\mathcal{T}$;
- with a plugin whose behaviour on quantifier-free problems is DPLL($\mathcal{T}$), using *watched literals* to propagate literals or close branches, and PSYCHE's memoisation feature to learn lemmas;
- with decision procedures for: pure propositional logic (for SAT-solving), pure first-order logic, quantifier-free Linear Rational Arithmetic (LRA), and Congruence Closure;
- with a DIMACS parser and an SMTLib2 parser[1];
- as a program of about 6700 lines of OCaml 4.00 (the kernel itself is only 800 lines), using hash-consing and Patricia tries for efficiency reasons.

PSYCHE does not claim to be a better SAT- or SMT-solver or first-order theorem prover than any existing one: for instance the heuristics for applying DPLL($\mathcal{T}$) rules in the aforementioned plugin are still basic, and so is the decision procedure for LRA (it is not incremental). What we offer here is a platform and its modularity: anyone with better (or different) heuristics and decision procedures can simply write them as OCaml modules of our predefined module types, and PSYCHE will seamlessly run with them, keeping the same LCF-style guarantees.

In Section 9.1, we give more motivation for the development of PSYCHE. In Section 9.2, we describe the general architecture of the system, in particular we explain how the guarantee of correctness is enforced, using the kernel API and a private type for answers. In Section 9.3, we briefly review how the kernel works, connecting to the theory decribed in the previous chapters. Section 9.4 then describes what the specifications required of a plugin, and the way our distributed plugin simulates DPLL($\mathcal{T}$) according to the results from Chapter 8. Section 9.5 describes the specifications of decision procedures and parsers, while Section 9.5 concludes with some tests and perspectives.

9.1 Motivation

PSYCHE's architecture is designed for the ambition of allowing various theorem proving techniques (generic or problem-specific) to collaborate on a common platform, whilst giving high confidence in the answers produced.

Interfacing the numerous techniques and tools available for theorem proving is legitimately receiving a lot of attention: Automated Theorem Provers, SAT-solvers, SMT-solvers, Proof assistants, etc. While trust is already an issue even for a single tool running on its own, it becomes even more of an issue when different tools interact. *Proof-checking* is one way of addressing this, i.e. being very permissive in the algorithms used for theorem proving,

[1]The latter is taken without modification from the Alt-Ergo SMT prover.

as long as they output some proof objects that can be checked. Another way is the LCF-style [GMW79], where only a small kernel of primitives needs to be trusted, and anything smarter (e.g. the interaction between sophisticated techniques) boils down to calls to the kernel's primitives.

In the context of proof-checking (such as in Coq [Coq]), a natural way to interact with different (already implemented) techniques, is the black box approach, where an external tool is called and its output is converted back into a proof that can be checked by the system [AFG+11, BCP11]. It is somewhat more surprising that, despite the highly programmable possibilities of the LCF architecture (from which the ML languages come), the most successful integration of automated reasoning techniques in an LCF-based proof assistant such as Isabelle [NPW02, Isa] seems to also use variants of the black box approach (as very impressively demonstrated by Sledgehammer) [Web11, PB12, BBP11].

Psyche aims at producing answers that are correct by construction, not having to rely on proof-checking; it therefore adopts the LCF philosophy (although it can produce proof objects), also because having a simple trusted kernel is a convenient starting point for different techniques to collaborate. But the goal here is to open the black boxes and program their algorithms directly with calls to the kernel's API, as plugins for Psyche.

Such a deeper level of integration opens up the perspective of interleaving the use of different techniques: An external tool requires an input problem that it can entirely treat; but implementing the *steps* of its algorithm as small progressions in the search-space covered by the main system, allows more possibilities, such as running the technique *up-to-a-point*, where a switch to another technique may be appropriate (e.g. depending on newly generated goals).

The challenge is for the kernel to offer an appropriate API of proof-search or proof-construction primitives, to allow the efficient implementation of theorem proving techniques as plugins. Most LCF-style systems offer primitives corresponding to the inference rules of Natural deduction, or a Hilbert-style system. This is a very fine-grained level, that leaves most (if not all) of the work to the plugin; requiring it to use the kernel's primitives is less of an aid and more of a constraint: it does ensure that, in case the output is provable, a proof has been constructed (at least theoretically), but it is a computational overhead for the plugin's work.

Psyche makes the choice of a bigger grain, and leaves to the kernel some real proof-search computation, but where no decision needs to be made. For this we use the focussed sequent calculus $\mathsf{LK}^p(\mathcal{T})$ [FGL13, FGLM13], whose quantifier-free version has been presented in Chapter 8. Not only can polarities and focussing be used to describe effective proof-search strategies in Sequent Calculus (narrowing the search-space offered by Gentzen's original rules), but in our case, they also specify a sensible division of labour between Psyche's kernel and Psyche's plugins, re-designing the standard LCF-style API.

This new design makes Psyche guarantee the correcteness of both types of answers: *provable* or *not provable*, while the traditional LCF style only guarantees the correctness of answers of the form *provable*.

9.2 Overview and general architecture

The kernel contains the mechanisms for exploring the proof-search space, taking into account branching and backtracking. It has no *a priori* regarding the order in which branches are explored, and this lack of intelligence makes its code rather short. If it reaches a proof, then that proof is correct by construction, and if the entire search space is explored and no proof is found, then the kernel correctly outputs that no proof exists.

The plugins then drive the kernel by specifying in which order the branches of the search space should be explored and to which depth, something that is expected to depend on the kind of problem that is being treated. The quality of the plugin is how fast it drives the kernel towards a answer *provable* or *not provable*.

This already departs from the traditional LCF-style in that some actual proof-search computation is performed in the kernel, not just atomic steps of proof-construction:

In traditional LCF, each inference rule of the logic

$$\frac{\mathsf{prem}_1 \quad \ldots \quad \mathsf{prem}_n}{\mathsf{conc}}\,\mathsf{name}$$

gives rise to a primitive of the kernel's API, whose type declares n arguments:

```
name:  thm -> ··· -> thm -> thm
```

In PSYCHE's kernel, such an inference rule is "wrapped" in the kernel's unique API primitive:

```
machine:  statement -> output
```

such that `search(conc)` will trigger the recursive calls `search(prem_1)`,..., `search(prem_n)`, as bottom-up proof-search should do.

PSYCHE's general architecture is illustrated by its main top-level call (slightly reworded for clarity):

```
Plugin.solve(Kernel.machine(Parser.parse input))
```

PSYCHE has a collection of parsers (currently one for DIMACS and one for SMTLib2) and calls the appropriate one on PSYCHE's input. The resulting abstract syntax tree is fed to the kernel's `machine` function that will initiate the search.[2] This produces a value of type `output` that is given to the plugin to work with, and the plugin must solve the problem by outputting an answer *provable* or *not provable*.

This could give the impression that the plugin performs computation after the kernel has finished his, but this is not quite true, as illustrated by the nature of type `output`:

```
type output = Jackpot of answer | InsertCoin of coin -> output
```

which describes the kernel as a *slot machine*: when it is run, it outputs

- either a definitive answer *provable* or *not provable*
- or an intermediate output that represents unfinished computation: in order for computation to continue, the plugin needs to "insert another coin in the slot machine"; depending on the kind of coin inserted, proof-search will resume in a certain way.

To summarise, the kernel performs proof-search as long as there is no decision to be made (on which backtrack may later be needed), and when it hits such a point, it stops and asks for another coin to indicate how to proceed next. The plugin drives the kernel in the exploration

[2] In fact, the kernel module is created with the choice of a background theory that is either guessed from the input or specified by the user on the command line.

of the proof-search space by inserting carefully chosen coins, hoping that one day the machine will stop with the "jackpot": a value of the form `Jackpot(...)`.

Now while this architecture somewhat departs from LCF, it does share with it the distrust of anything outside the kernel: when concerned with the soundness of the answer (whichever it be), the plugin is here considered as an adversary, so Psyche defines the type `answer` as a private type that only the kernel can inhabit (just like the `thm` type of LCF). Psyche's type

```
answer = private Provable of statement*proof | NotProvable of statement
```

can be read by the plugin and the top-level if need be, but cannot be inhabited by them. That way, a plugin cannot cheat about Psyche's answer: the worst it can do is of course to crash Psyche's runs or diverge. In Psyche as in traditional LCF, inhabitation of the abstract type (in case of Psyche, with a value of the form `Provable(...)`) explicitly or implicitly constructs a proof of the statement. But contrary to LCF, Psyche also gives guarantees when the output is *not provable*: it can only occur when the kernel has entirely explored the search-space unsuccessfully.

Such a use of typing prompted for an ML-language to implement Psyche, and we chose OCaml (4.0).

9.3 Psyche's Kernel

As described above, the kernel's API has the slot machine as its only primitive, controlled by the *coins* that are inserted in it. In order for efficient plugins to be conveniently programmed, the kernel's primitive needs to accept a rather expressive range of coins that can specify a smart exploration of the search-space. This depends on the inference system that is used in the kernel for the incremental and bottom-up construction of proof-trees, and on identifying the inference rules that the kernel will perform automatically from those that will pause computation and prompt the plugin for new directions.

This is where focussed sequent calculi for polarised logic(s) come in. Focussing is what we use for the division of labour between Psyche's kernel and Psyche's plugins:

The kernel applies the asynchronous steps automatically without any instruction from the plugin, and then stops and asks for another coin describing the next synchronous phase, where smart choices may have to be made (starting with the choice of the positive formula to work on).

An important consequence of this division of labour is that **every kernel call terminates**, because the length of each phase is bounded by the size of the formula(e) being decomposed. Therefore, infinite proof-search has to go through an infinite interaction between the kernel and the plugin (unless the plugin itself loops before inserting the next coin).

The choice of polarities on connectives and literals affects the kernel-plugin interaction. For instance the polarity of $\vee$ will determine whether it will be decomposed automatically by the kernel (second rule, asynchronous) or with a smart choice by the plugin (first rule, synchronous):

$$\frac{\Gamma \vdash A_i}{\Gamma \vdash A_1 \vee^+ A_2} \qquad \frac{\Gamma \vdash A_1, A_2, \Gamma'}{\Gamma \vdash A_1 \vee^- A_2, \Gamma'}$$

The polarity of literal being also crucial, Psyche offers the plugin the possibility to polarise literals *on-the-fly*, during the search (which is very useful for the plugins we implemented).

In PSYCHE 2.0, the kernel implements the sequent calculus $\mathsf{LK}^p(\mathcal{T})$ [FGLM13, Far13], whose quantifier-free version was presented in Chapter 8. But PSYCHE does implement the full system with quantifiers, with specific mechanisms dealing with eigenvariables (introduced when proving a universal formula) and meta-variables (introduced when proving an existential formula).

The different coins that the plugin can insert thus correspond to the smart application of the non-asynchronous inference rules of $\mathsf{LK}^p(\mathcal{T})$: a formula to select, a side to choose when decomposing $\vee^+$, a literal to polarise in a certain way, a cut to be made ($\mathsf{LK}^p(\mathcal{T})$ admits cuts), or a consistency check of the current sequent with the given background theory (a global parameter of the kernel).

Finally, the plugin can also instruct the kernel to move in the search-space: when it gets tired of investigating the current branch, it can abandon it temporarily and explore the next success/failure branch to the left/right.

The code of the kernel is rather small (around 800 lines) and purely functional. Continuation-Passing-Style (CPS) is used to minimise the use of the stack and provide a natural way to represent the progression of the kernel within the search space: the API function

```
machine: statement->output
```

actually wraps a real (tail-)recursive function

```
search: statement->(output->'a)->'a
```

with the identity continuation. Continuations are heavily used for branching and backtracking (e.g. when `search` applies a rule with several premises, it makes a recursive call on one of the branches and stacks up the others in the continuation that is passed; when the plugin chooses to explore one branch, the kernel records in a similar way the other branches that are not being explored yet -forcing in the end the entire exploration of the search-space), and naturally provide the values implementing a slot machine waiting for its coin.

9.4 Plugins

9.4.1 Specifications and implemented instances

A plugin is any OCaml module implementing the following identified module type (bearing in mind that `answer` is for the plugin a private type that it cannot inhabit by itself):

```
module type PluginType = sig
  ...
  solve: output->answer
end
```

However, it is likely that the sophisticated strategies/heuristics that the plugin is meant to implement rely on some clever choice of data-structures for formulae, sets of formulae, sets of literals. So the plugin and the kernel have to agree on those three data-structures that are communicated both ways during the interaction. In PSYCHE 2.0, the kernel provides the data-structure to represent formulae, but the plugin can embark in the data-structure the information that it needs to treat them efficiently. The data-structures implementing sets of

formulae and sets of literals, on the other hand, are parameters of the kernel, and the plugin provides them.[3]

We first tested PSYCHE's architecture with a basic plugin `Naive`, which

- implements sets (of formulae, literals) with OCaml's lists;
- inserts the first available coin in the slot machine, whenever asked.

This works fine for small tautologies, printable on a screen.

More recently, Jean-Marc Notin provided a module for interactive theorem proving, via a command-line interface: it still implements sets using OCaml's lists, but every time a coin needs to be inserted in the machine, the interface prompts the user for the coin to insert.

A more ambitious aim for automated reasoning was to capture in PSYCHE some propositional SAT and SAT-Modulo-Theories solving techniques, making DPLL($\mathcal{T}$) technology available in a generic tableau-like / Prolog-like / goal-directed proof-search framework like PSYCHE.

For this we implemented the simulation of DPLL($\mathcal{T}$), expressed rather canonically as a transition system [NOT06], as a simple bottom-up proof-construction mechanism in $\mathsf{LK}^p(\mathcal{T})$, as described in Chapter 8. More practically, every rule of DPLL($\mathcal{T}$) can be seen as the insertion of a particular coin in PSYCHE's slot machine.

We implemented this as two different plugins for PSYCHE: `DPLL_Pat` and `DPLL_WL`. These remain toy plugins, because, although it is now clear, from Chapter 8, how to perform each rule of DPLL($\mathcal{T}$) in PSYCHE, we still have to decide *which* rule to apply and *when*. So the two plugins

- embark, in the kernel's representation of formulae, a flat representation of them as sets of literals when the formulae happen to be clauses;
- implement sets (of formulae/clauses, literals) using Patricia tries;
- implement a basic strategy to apply DPLL($\mathcal{T}$) rules; in the case of propositional logic: apply Fail or Backtrack if possible, if not try Unit Propagate, if not do Decide on some random literal.

The two plugins differ in the way they look up for the applicability of Fail / Backtrack / Unit Propagate: `DPLL_Pat` looks it up using the Patricia tries implementing sets of clauses, while `DPLL_WL` looks it up using the technique of *watched literals* [MMZ+01] (keeping a small watching table in the plugin). This technique was originally implemented in PSYCHE by student Matthieu Vegreville, and the plugin seems on average 1.5 faster than that using Patricia tries.

9.4.2 Memoisation and lemma learning

Such plugins would not be efficient at all if no backjumping and clause learning was done while performing DPLL($\mathcal{T}$). In [FGLM13] we also show how to do this using general cuts, and either accept to extend several open branches of an open proof-tree with identical steps or depart from the bottom-up proof construction paradigm that we have used so far. We

[3] This is admittedly a security problem, since a bug in the plugin's data-structure for sets could affect the way a sequent is transformed by the kernel when it applies an inference rule bottom-up. The next version of PSYCHE will adopt a double representation of sets (one for the kernel, one for the plugin) to completely avoid the kernel relying on plugin code.

opted for a generic mechanism to avoid re-doing, for some open branch, the same steps as those used in a previously completed branch: *memoisation*. In Chapter 8 we explained how the use of memoisation emulates the use of a learnt clause for Fail, Unit Propagate, etc.

Indeed, nothing prevents a plugin from recording the sub-trees completed by the kernel, and proposing them later for another branch where the same proof-tree is relevant. PSYCHE 2.0 therefore offers a memoisation module, to be used by plugins to record values of (the abstract) type `answer`. And the kernel's slot machine accepts from the plugin, as a special coin carrying such a value, "here is an already found answer that also applies to the current goal". The kernel accepts the value as closing the current branch (one way or another) **without any proof-checking** (since the abstract type ensures the value came as an earlier output of the kernel); it only checks that the value applies to the current goal.

The memoisation table is filled-in by clause-learning: our plugin adds an entry whenever it builds a complete proof of some sequent $\Delta \vdash$ and no previous entry $\Delta' \vdash$ exists with $\Delta' \subseteq \Delta$, or whenever it concludes that some sequent $\Delta \vdash$ is not provable and no previous entry $\Delta' \vdash$ exists with $\Delta \subseteq \Delta'$.

Now for a memoised answer *Provable* to be reusable as often as possible, a pre-processing step is applied to a proof-tree before it enters the table: it is pruned from every formula that is not used in the proof. This is easy to do for the complete proofs of $\mathsf{LK}^p(\mathcal{T})$ (eager weakening are applied a posteriori by inspection of the inductive structure). PSYCHE's kernel actually performs the pruning on-the-fly whenever an inference is added to complete proofs, so that, whenever it outputs `Jackpot(sequent,proof)`, the sequent is already pruned.

Since proof-completion can be seen as finding a *conflict* (a situation where the current partial model contradicts the set of clauses), pruning by eager weakening is a *conflict analysis* process naturally provided by structural proof theory:

Conflict anaylsis is a process used in SAT- and SMT-solving aims at identifying, in a situation of Fail, Backtrack, etc, which literals of the current model are sufficient to contradict, when taken together, the set of clauses; the disjunction of their negations forms a new clause that can be learnt and re-used later. Techniques to compute this can be based on *graph analysis*; the kind of conflict analysis performed by the pruning mechanism of PSYCHE turns out to be a particular form the graph analysis mechanism.

Of course, just as in SMT-solving, the efficiency of conflict analysis relies on the efficiency of the decision procedure in providing a small inconsistent subset whenever it decides that a set of literals is inconsistent.

Another feature sometimes used in SMT-solving, in conjunction with clause learning, is the use of *restarts*: at some point of the $\mathsf{DPLL}(\mathcal{T})$ run, computation resumes with the empty model:

$$\Delta \| \phi \Rightarrow \emptyset \| \phi$$

This is only useful if the current set of clauses ϕ is different from the original one, i.e. some clauses have been learnt: in that case, restarting from the empty model but with all the learnt clauses, might be faster than closing all the branches that have been opened (corresponding to the decision literals in Δ).

In PSYCHE, restarts can be done the same way: since the plugin is in charge of computation, it can record the first output that the kernel produced, and later come back to it: the side effect that makes it different from the first run is that the memoisation table has been filled with valuable information; this information may allow the search to find a proof

more quickly than by closing all the open branches of the current incomplete proof-tree. This has been implemented in Psyche by students Zelda Mariet and Clément Pit-Claudel, with convincing experimental results.

9.5 Decision procedures

Decision procedures and parsers integrate Psyche's code the same way as plugins: we offer a module type for decision procedures and one for parsers. Someone with a decision procedure or a parser can implement a module of the corresponding type and run Psyche with it.

In the case of decision procedures for quantifier-free problems (i.e. with ground literals), the output of a decision procedure for the background theory $\mathcal{T}$ should be able to decide whether a conjunction of literals is consistent with $\mathcal{T}$ or not.

```
module type GroundDecProc = sig
  ...
  type literals
  ...
  consistency: literals set -> (literals set) option
end
```

The decision procedure provides the type of literals, so as to run efficiently, while the kernel accepts any type for literals since it will not inspect its values.

The output of the `consistency` function is not a boolean: it should be `None` if the input is a set of literals consistent with the theory $\mathcal{T}$, and `Some(s)` otherwise, with `s` being a subset of the input that is already inconsistent. Indeed, conflict analysis requires such subsets to be produced when an inconsistency is found, and the smaller the subsets, the more efficient clause learning and memoisation will be.

In the case of problems with quantifiers, the decision procedure should answer whether there is a way to instantiate meta-variables so as to make the conjunction of literals inconsistent with the theory: only in this case will the current branch be immediately closed, propagating the instantiation of meta-variables to the remaining open branches. In case such an instantiation fails another branch, we should backtrack to the current branch and propose another way of closing it. Therefore, the decision should not only be able to decide whether there is an instantiation of meta-variables that makes the literals inconsistent, but it should be able to *enumerate all* possible instantiations that make the literals inconsistent. Instead of a boolean answer, we thus expect a *stream* of solutions, each of which is a set of literals together with a working instantiation:

```
module type DecProc = sig
  ...
  type literals
  type constraints
  ...
  val consistency : literals set -> (literals set,constraints) stream
end
```

The idea of using streams of solutions is natural, and proposed in the form of *instance streams* in [Gie00] in a proof-search methodology that deliberately avoids backtracking. Although proof-search in PSYCHE does backtrack, we should investigate the connection between PSYCHE's methodology and that of [Gie00].

As evoked by the module type above, instantiations in PSYCHE are actually called *constraints*: in pure first-order logic, a constraint would simply be a (most general) unifier σ that makes two literals l_1 and l_2 of the input set such that $l_1\sigma = l_2^{\perp}\sigma$. It would be easy to enumerate all such constraints, by enumerating all pairs l_1 and l_2 of the input set that can be unified in the above sense: they are in finite numbers.

But for other theories we could imagine different kinds of constraints on meta-variables: for instance in Linear Rational Arithmetic we could imagine a constraint imposing that meta-variable `?X3` be in the interval $\left[0; \frac{3}{2}\right]$. Therefore, the decision procedure provides the notion of constraint that is appropriate for the background theory $\mathcal{T}$, while the type for constraints is abstract for the kernel, which will only propagate constraints from branch to branch, but not inspect them.

The exact specifications that should be met by the constraint structure so that proof-search using them is sound and complete with respect to the formalisation of the theory $\mathcal{T}$ without meta-variables, is the object of a paper being currently written with student Damien Rouhling.

Finally, PSYCHE is modular in its parsers: a *parser* is any module of a pre-defined module type, and should in particular implement a function

```
parse:  string->((statement option)*(boolean option))
```

that turns a string input into a statement to be proved (or `None` if no statement was parsed in the input), and possibly an expected result *Provable*/*NotProvable* that the input string may indicate.

Conclusion: Testing and perspectives

PSYCHE 2.0 is run from the command-line, taking as input one or more file(s) or directory(ies), or, if none indicated, the standard input:

```
psyche [OPTION]... [FILE/DIR]...

Available options are:
  -theory selects theory (among empty, lra, cc, first-order; default is empty)
  -gplugin selects generic plugin (among naive, dpll_pat, dpll_wl; default is dpll_wl)
  -latex allows latex output of proof-trees
  -alphasort treats input files in alphabetical order (default is from smaller to bigger)
  -examples treats theory examples instead of standard input
  -nocuts disallows cuts
  -fair ensures fairness between formulae for focus
  -noweakenings disables conflict analysis
  -nomemo disables memoisation
  -restarts selects a restart strategy
  -help  displays this list of options
```

As illustrated by the options, Psyche can produce proofs (of $\mathsf{LK}^p(\mathcal{T})$) and print them as inference trees in LaTeX format (but proofs can quickly get too big for LaTeX).

We ran Psyche on instances of SAT in DIMACS format, and `QF_LRA` instances in SMTLib2 format and the results are available on Psyche's website [Psy]. Psyche works well on small instances and its performance starts declining between 20Kb and 100Kb of input problem size (of course this is no appropriate measure of difficulty). This is of course very far from current SAT benchmarks, perhaps a bit less from SMTLib2 ones (our instances were download from the up-to-date library). But as we said, the current plugins and decision procedures are illustrative toys. Psyche is a platform where people knowing good and efficient techniques should be able to program them.

In the short-to-medium terms, we plan to

- improve the decision procedure for LRA (making it incremental, and returning smaller sets);
- implement other theories and combine them (congruence closure, Linear Integer Arithmetic, bit vectors, etc);
- improve $\mathsf{DPLL}(\mathcal{T})$ plugins to better handle non-clausal formulae;
- implement other theorem proving techniques as plugins: *analytic tableaux* are the closest to our sequent calculus, but theoretical developments have already shown that *clausal tableaux* (including *connection tableaux*) can also be done [Far13], as well as *resolution* proofs.

Finally, we can imagine using a proof assistant to prove Psyche's correctness, since the kernel seems small enough (800 lines) and the plugins need not be certified.

We will develop our long-term plans for Psyche in the conclusion of this dissertation.

Chapter 10

Conclusion and further work

Contents

10.1 Summary of the topics covered by this dissertation

In the first part of this dissertation, we reviewed the computational interpretation of proofs in terms of Call-by-Name and Call-by-Value evaluation of programs [CH00, Sel01]; we approached realisability semantics by a systematic construction of orthogonality models [Par97, DK00, Kri01, Miq11], used for instance to prove strong normalisation results or classical witness extraction. From this the concepts of polarities and focussing naturally emerged [MM09], and a computational interpretation of focussed proofs was given in terms of pattern-matching [Zei08a, Zei08b].

In the second part of this dissertation, we developed this approach into an abstract focussed sequent calculus LAF with proof-terms, of which several focussed calculi of the literature are instances, such as LKF and LJF [LM09]. We used this framework to formally relate classical realisability with the computational interpretation of focussing as pure pattern-matching, again via the construction of orthogonality models.

In the third part of the thesis we explored a specific approach to theorem proving benefitting from the use of polarities and focussing. We described how these concepts can contribute to the description of the DPLL($\mathcal{T}$) procedure for SMT-solving [NOT06] as a specific strategy for the bottom-up proof-search process specified by the sequent calculus. For this we extended the focussed sequent calculus LKF into LK^p($\mathcal{T}$), equipped with the ability to polarise atoms *on-the-fly* and call a decision procedure specific to the background theory $\mathcal{T}$.

We then described the implementation of a proof-search engine called Psyche [Psy] whose architecture is based on a kernel that interacts with plugins to be programmed via a specific API. This allows the implementation and experimentation of various reasoning techniques (among which DPLL($\mathcal{T}$)) and heuristics, without worrying about breaking the correctness of Psyche's output: this is guaranteed to be correct by the architecture, which develops a new variant of the LCF style [GMW79].

10.2 Impact of this dissertation of the development of PSYCHE, and further work

In conclusion of this dissertation we proffer two main directions in which the material of this dissertation will be developed and integrated to the next releases of PSYCHE.

The first one is a rather major change in the kernel of PSYCHE: instead of implementing bottom-up proof-search in the particular focussed sequent calculus called $\mathsf{LK}^p(\mathcal{T})$, which is specific to classical logic, PSYCHE 3.0 will implement proof-search in the abstract focussed sequent calculus LAF developed in Part II of this dissertation. This will allow the kernel to be decomposed into smaller components: the main module will have much fewer rules to implement than in $\mathsf{LK}^p(\mathcal{T})$, taking advantage of the "big-step presentation" of focussing; the decomposition of formulae into smaller formulae, given by the specific relation $\Vdash$ of LAF, will be moved to a specific module where the inductive syntax of formulae is implemented; another module will implement typing contexts with their notion of context extension that crucially determines which logic is being implemented, etc.

The advantages of modularising the kernel in this way are numerous:

- It will allow PSYCHE to run on different instances of LAF, thus handling different systems and logics;
- PSYCHE will then be equipped with proof-terms, which may be used as compact representations of proofs in memory (e.g. in the memoisation table); this will also allow the extraction of programs from proofs (in different logics);
- the code will also be simpler to understand and formally prove correct, as most of the specifications describing the roles of each component have already been identified in Part II of this dissertation, with most of the theorems already formalised in Coq [GL14];

In retrospect, this next move in the development of PSYCHE is also what motivated the detailed study of LAF, at the cost of presenting it in a rather technical way. However, theoretical work still needs to be done before this new basis for PSYCHE's implementation replaces the current one. Indeed, as described in details in Section 8.3, it is not clear how PSYCHE can take advantage of rules that are admissible for specific instances of LAF but not generically (e.g. specific forms of cuts, on-the-fly polarisation rules, etc); more importantly, in order to supersede the current implementation, LAF needs to be generalised into a system $\mathsf{LAF}(\mathcal{T})$ that can call a decision procedure for a theory $\mathcal{T}$. What conditions are required of such procedures for cut-elimination to work, etc, remains to be identified.

The second direction for further development is exploiting the machinery for quantifiers that has newly been introduced with the release of PSYCHE 2.0 on 20th September 2014.

As briefly described in Section 9.5, this machinery involves meta-variables which are introduced when breaking an existential quantifier sitting on top of a formula to be proved, and eigenvariables which are introduced when breaking a universal quantifier. Dependencies between them are recorded in the proof-search, so as to avoid the production of incorrect instances for meta-variables, from which no actual proof could be re-constituted.

On the note of dependencies, Skolemisation is often described as the transformation of a formula $\forall x_1 \dots \forall x_n \exists y A$ to be *refuted* (by tableaux methods, resolution, etc...) into the formula $\forall x_1 \dots \forall x_n \left\{ \mathsf{sk}_y(x_1,\dots,x_n) / y \right\} A$, where sk_y is a (new) *Skolem symbol* (specific to y). The

correctness of this transformation is often justified by semantical models involving the axiom of choice, and Skolemisation is often applied as a pre-processing step before refutational methods are applied. In bottom-up proof-search, Skolemisation occurs *on-the-fly* when the universal quantifier of $\exists x_1 \dots \exists x_n \forall y A^{\perp}$ is broken, and the skolem symbol sk_y is merely the eigenvariable Y introduced for y; the fact that the Skolem symbol is *applied* to $x_1,\dots,x_n$ (or in proof-search, to their corresponding meta-variables $?X_1,\dots,?X_n$) is a mere implementation trick to record the dependencies between eigenvariables and meta-variables: writing $Y(?X_1,\dots,?X_n)$ simply records that $?X_1,\dots,?X_n$ were introduced before Y and any correct instances for them cannot mention Y. This is a smart implementation of the dependencies in the case of pure first-order logic, inasmuch first-order unification will rule out incorrect instances "for free" thanks to the `occurs_check`.

As Psyche aims at working modulo theories, it is no longer clear that this specific implementation of dependencies will be as appropriate when another algorithm than first-order unification is run to close branches. A dual implementation of dependencies would consist for instance in recording, whenever a meta-variable $?X$ is introduced, the eigenvariables that existed at that point, among which any correct instance for $?X$ would need to find its free variables. This is for instance the choice in Coq [Coq]: instead of recording the dependencies that are disallowed (as in Skolemisation), one records the dependencies that are allowed. To avoid making any commitment on that choice of implementation, Psyche 2.0 is modular in the data-structure that implements dependencies.

Similarly, Psyche's kernel is agnostic in regard of the *constraints* imposed on the instantiation of meta-variables by closing branches: proving a sequent $\Gamma \vdash \Gamma'$ mentioning meta-variables should output (if successful) on constraint σ on these meta-variables, which in pure first-order logic could simply be a *first-order unifier*, but in other theories could be of a different nature (one could think of convex polytops for arithmetic, for instance). A branching rule such as

$$(\wedge^-)\frac{\Gamma \vdash A, \Delta \qquad \Gamma \vdash B, \Delta}{\Gamma \vdash A \wedge^- B, \Delta}$$

should eventually produce the *meet* $\sigma \wedge \sigma'$ of the two constraints σ and σ' returned by the recursive calls of the proof-search function on the two premisses, something we can write as

$$(\wedge^-)\frac{\Gamma \vdash A, \Delta \Rightarrow \sigma \qquad \Gamma \vdash B, \Delta \Rightarrow \sigma'}{\Gamma \vdash A \wedge^- B, \Delta \Rightarrow \sigma \wedge \sigma'}$$

The meet should of course exists (otherwise we should try to find other ways to prove the premisses), in other words $\sigma \wedge \sigma'$ should not be the unsatisfiable constraint $\perp$. There we see an algebraic structure like a semi-lattice appear as the natural concept to spell out the abstract specifications of how constraints work. To avoid the independent exploration of branches before realising they produce incompatible constraints, Psyche 2.0 implements the propagation of constraints from one branch to the next, which we could write as

$$(\wedge^-)\frac{\sigma_0 \Rightarrow \Gamma \vdash A, \Delta \Rightarrow \sigma \qquad \sigma \Rightarrow \Gamma \vdash B, \Delta \Rightarrow \sigma'}{\sigma_0 \Rightarrow \Gamma \vdash A \wedge^- B, \Delta \Rightarrow \sigma'}$$

where $\sigma_0 \geq \sigma \geq \sigma'$ for the semi-lattice ordering, breaking the symmetry of the introduction rule for conjunction depending on which of the two branches is actually explored first.

A paper with Damien Rouhling is currently being written on such systems of constraint propagation, which still allow proof-search to backtrack, with persistent data-structures for

constraints. Besides justifying the implementation of PSYCHE 2.0 and identifying the algebraic specifications that an implemented constraint module should satisfy, further development should investigate the connections with *constraint tableaux* [GH03] and *constraint logic programming* [JM94].

Now, what to do with the quantifier features of Psyche 2.0?

First, test them on standard benchmarks and possibly add a TPTP parser to PSYCHE.

Secondly, it was shown in [Far13] how to simulate, as proof-search in $\mathsf{LK}^p(\emptyset)$ with quantifiers, the techniques of *clause tableaux* and strong or weak *connection tableaux* (see e.g. [RV01]) for pure first-order logic (hence the empty theory $\emptyset$). How to simulate resolution is also known. So turning these simulations into the implementation of plugins for PSYCHE is the obvious next step, which would allow us to run tests and compare the plugins with other implementations.

Thirdly, we have an approach for mixing first-order reasoning with theories (or, said differently, perform instantiations in presence of a theory); with this we can:

- Investigate to what extent the standard *triggers-based* mechanisms of SMT-solvers for instantiations, can be described in our setting; in particular, Dross [DCKP12] provides theoretical foundations for the use of triggers:

  Intuitively, an existential formula $\exists x[k]A$ with a *trigger* $[k]$ allows, as only instantiations of x, those which turn k into a term that is already *known*, as if the notation represented a formula $\exists x(\mathsf{known}(k) \wedge A)$ together with a proof-search policy that forces to prove the left branch $\mathsf{known}(k)$ before the branch on formula A is explored. This strongly suggests a focussing approach in which the predicate $\mathsf{known}(\_)$ is positive, so that in the above case $\mathsf{known}(k)$ has to immediately be proved by an axiom. This should be formalised in our focussed calculi.
- More generally look at the various extensions of $\mathsf{DPLL}(\mathcal{T})$, in particular the systems with full first-order logic and/or equality as developed by e.g. [Bau00, BT08, BT11], and compare them to $\mathsf{LK}^p(\mathcal{T})$ with quantifiers and meta-variables. Note that the abstract setting of LAF might be appropriate for equality itself: since we have abstracted away from connectives, it may be the case that sequent calculi with equality just fit as LAF instances.

Bibliography

[ADH+12] Z. M. Ariola, P. Downen, H. Herbelin, K. Nakata, and A. Saurin. Classical call-by-need sequent calculi: The unity of semantic artifacts. In T. Schrijvers and P. Thiemann, editors, *Proc. of the 11th Int. Symp. Functional and Logic Programming (FLOPS'12)*, volume 7294 of *LNCS*, pages 32–46. Springer-Verlag, 2012. 49

[AF97] Z. M. Ariola and M. Felleisen. The call-by-need lambda calculus. *J. Funct. Programming*, 7(3):265–301, 1997. 49

[AFG+11] M. Armand, G. Faure, B. Grégoire, C. Keller, L. Théry, and B. Werner. A modular integration of SAT/SMT solvers to Coq through proof witnesses. In *Proc. of the 1st Int. Conf. on Certified Programs and Proofs (CPP'11)*, volume 7086 of *LNCS*, pages 135–150. Springer, 2011. 152, 169

[AH03] Z. M. Ariola and H. Herbelin. Minimal classical logic and control operators. In J. C. M. Baeten, J. K. Lenstra, J. Parrow, and G. J. Woeginger, editors, *Proc. of the 30th Intern. Col. on Automata, Languages and Programming (ICALP)*, volume 2719 of *LNCS*, pages 871–885. Springer-Verlag, 2003. 22, 27

[AH08] Z. M. Ariola and H. Herbelin. Control reduction theories: the benefit of structural substitution. *J. Funct. Programming*, 18(3):373–419, 2008. 27

[AHS09] Z. M. Ariola, H. Herbelin, and A. Sabry. A type-theoretic foundation of delimited continuations. *Higher-Order and Symbolic Computation*, 22(3):233–273, 2009. online from 2007. 49

[AHS11] Z. M. Ariola, H. Herbelin, and A. Saurin. Classical call-by-need and duality. In C. L. Ong, editor, *Proc. of the 10th Int. Conf. on Typed Lambda Calculus and Applications (TLCA'11)*, volume 6690 of *LNCS*, pages 27–44. Springer-Verlag, 2011. 49

[And92] J. M. Andreoli. Logic programming with focusing proofs in linear logic. *J. Logic Comput.*, 2(3):297–347, 1992. 1, 71, 73, 149

[AP89] J.-M. Andreoli and R. Pareschi. Logic programming with sequent systems, a linear logic approach. In P. Schroeder-Heister, editor, *Proc. of the Int. Work. on Extensions of Logic Programming*, LNCS, pages 1–30. Springer-Verlag, 1989. 1, 71, 73, 149

[Bar84] H. P. Barendregt. *The Lambda-Calculus, its syntax and semantics*. Studies in Logic and the Foundation of Mathematics. Elsevier, 1984. Second edition. 16, 17, 20, 61, 67

[Bar91] H. P. Barendregt. Introduction to generalized type systems. *J. Funct. Programming*, 1(2):125–154, 1991. 116

[Bar92] H. P. Barendregt. Lambda calculi with types. In S. Abramsky, D. M. Gabby, and T. S. E. Maibaum, editors, *Hand. Log. Comput. Sci.*, volume 2, chapter 2, pages 117–309. Oxford University Press, 1992. 149

[Bau00] P. Baumgartner. FDPLL - a first order Davis-Putnam-Longeman-Loveland procedure. In *Proc. of the 17th Int. Conf. on Automated Deduction (CADE'00)*, volume 1831 of *LNCS*, pages 200–219. Springer-Verlag, 2000. 182

[BB96] F. Barbanera and S. Berardi. A symmetric lambda-calculus for classical program extraction. *Inform. and Comput.*, 125(2):103–117, 1996. 6, 29, 55

[BBP11] J. C. Blanchette, S. Böhme, and L. C. Paulson. Extending Sledgehammer with SMT solvers. In *Automated Deduction*, volume 6803 of *LNCS*, pages 116–130. Springer-Verlag, 2011. 152, 169

[BCP11] F. Besson, P.-E. Cornilleau, and D. Pichardie. Modular SMT proofs for fast reflexive checking inside Coq. In J.-P. Jouannaud and Z. Shao, editors, *Certified Programs and Proofs*, volume 7086 of *LNCS*, pages 151–166. Springer-Verlag, 2011. 152, 169

[BFM+13] F. Bobot, J.-C. Filliâtre, C. Marché, G. Melquiond, and A. Paskevich. The Why3 platform 0.81. 2013. Tutorial and Reference Manual. Available at http://hal.inria.fr/hal-00822856 150

[BG01] H. Barendregt and H. Geuvers. Proof-assistants using dependent type systems. In Robinson and Voronkov [RV01], pages 1149–1238. 149

[BGL12] A. Bernadet and S. Graham-Lengrand. A simple presentation of the effective topos. Technical report, Laboratoire d'informatique de l'École Polytechnique - CNRS, France, 2012. Available at http://hal.archives-ouvertes.fr/hal-00844250 6

[BGL13] A. Bernadet and S. Graham-Lengrand. Non-idempotent intersection types and strong normalisation. *Logic. Methods Comput. Science*, 9(4), 2013. 6

[BL11a] A. Bernadet and S. Lengrand. Complexity of strongly normalising λ-terms via non-idempotent intersection types. In M. Hofmann, editor, *Proc. of the 14th Int. Conf. on Foundations of Software Science and Computation Structures (FOSSACS'11)*, volume 6604 of *LNCS*. Springer-Verlag, 2011. 7

[BL11b] A. Bernadet and S. Lengrand. Filter models: non-idempotent intersection types, orthogonality and polymorphism. In M. Bezem, editor, *Proc. of the 20th Annual Conf. of the European Association for Computer Science Logic (CSL'11)*, LIPIcs. Schloss Dagstuhl LCI, 2011. 6, 51, 52, 54, 55

[BL11c] A. Bernadet and S. Lengrand. Filter models: non-idempotent intersection types, orthogonality and polymorphism - long version. Technical report, LIX, CNRS-INRIA-Ecole Polytechnique, 2011. Available at http://hal.archives-ouvertes.fr/hal-00600070/en/ 6

[BMS10] D. Baelde, D. Miller, and Z. Snow. Focused inductive theorem proving. In J. Giesl and R. Hähnle, editors, *Proc. of the 5th Int. Joint Conf. on Automated Reasoning (IJCAR'10)*, volume 6173 of *LNCS*, pages 278–292. Springer-Verlag, 2010. 3, 4

[BT08] P. Baumgartner and C. Tinelli. The model evolution calculus as a first-order DPLL method. *Artificial Intelligence*, 172(4-5):591–632, 2008. 182

[BT11] P. Baumgartner and C. Tinelli. Model evolution with equality modulo built-in theories. In N. Bjørner and V. Sofronie-Stokkermans, editors, *Proc. of the 23rd Int. Conf. on Automated Deduction (CADE'11)*, volume 6803 of *LNCS*, pages 85–100. Springer-Verlag, 2011. 182

[CD78] M. Coppo and M. Dezani-Ciancaglini. A new type assignment for lambda-terms. *Arch. Math. Log.*, 19:139–156, 1978. 7

[CF58] H. B. Curry and R. Feys. *Combinatory Logic*, volume I. North-Holland, 1958. 1, 13, 14, 18

[CH00] P.-L. Curien and H. Herbelin. The duality of computation. In *Proc. of the* 5^{th} *ACM SIGPLAN Int. Conf. on Functional Programming (ICFP'00)*, pages 233–243. ACM Press, 2000. 3, 5, 29, 44, 73, 179

[Chu41] A. Church. *The Calculi of Lambda Conversion*. Princeton University Press, 1941. 16

[CMM10] P.-L. Curien and G. Munch-Maccagnoni. The duality of computation under focus. In C. S. Calude and V. Sassone, editors, *Theoretical Computer Science*, volume 323 of *IFIP Advances in Information and Communication Technology*, pages 165–181. Springer-Verlag, 2010. 57, 71

[Coq] The Coq Proof Assistant. Available at http://coq.inria.fr/ 5, 6, 87, 149, 169, 181

[Cro04] T. Crolard. A formulae-as-types interpretation of subtractive logic. *J. Logic Comput.*, 14(4):529–570, 2004. 46, 49, 70

[CS09] J. R. B. Cockett and L. Santocanale. On the word problem for ΣΠ-categories, and the properties of two-way communication. In E. Grädel and R. Kahle, editors, *Proc. of the 18th Annual Conf. of the European Association for Computer Science Logic (CSL'09)*, volume 5771 of *LNCS*, pages 194–208. Springer-Verlag, 2009. 37

[dC05] D. de Carvalho. Intersection types for light affine lambda calculus. *ENTCS*, 136:133–152, 2005. 7

[dC09] D. de Carvalho. Execution time of lambda-terms via denotational semantics and intersection types. *CoRR*, abs/0905.4251, 2009. 7

[DCKP12] C. Dross, S. Conchon, J. Kanig, and A. Paskevich. Reasoning with triggers. In P. Fontaine and A. Goel, editors, *10th Int. Work. on Satisfiability Modulo Theories, SMT 2012*, volume 20 of *EPiC Series*, pages 22–31. EasyChair, 2012. 182

[DF89] O. Danvy and A. Filinski. A functional abstraction of typed contexts. Technical Report 89/12, DIKU, University of Cophenhagen, 1989. 49

[DF90] O. Danvy and A. Filinski. Abstracting control. In *Proc. of the 1990 ACM Conf. on LISP and functional programming*, pages 151–160. ACM Press, 1990. 49

[DGHP99] M. D'Agostino, D. M. Gabbay, R. Hähnle, and J. Posegga. *Handbook of Tableau Methods*. Kluwer Academic Publishers, 1999. 149, 167

[DJS95] V. Danos, J.-B. Joinet, and H. Schellinx. LKQ and LKT: sequent calculi for second order logic based upon dual linear decompositions of classical implication. In J.-Y. Girard, Y. Lafont, and L. Regnier, editors, *Proc. of the Work. on Advances in Linear Logic*, volume 222 of *London Math. Soc. Lecture Note Ser.*, pages 211–224. Cambridge University Press, 1995. 1, 44, 71, 73

[DJS97] V. Danos, J.-B. Joinet, and H. Schellinx. A new deconstructive logic: Linear logic. *J. of Symbolic Logic*, 62(3):755–807, 1997. 1, 44, 71, 73

[DK00] V. Danos and J.-L. Krivine. Disjunctive tautologies as synchronisation schemes. In P. Clote and H. Schwichtenberg, editors, *Proc. of the 9th Annual Conf. of the European Association for Computer Science Logic (CSL'00)*, volume 1862 of *LNCS*, pages 292–301. Springer-Verlag, 2000. 2, 3, 51, 53, 121, 179

[DL07] R. Dyckhoff and S. Lengrand. Call-by-value λ-calculus and LJQ. *J. Logic Comput.*, 17:1109–1134, 2007. 5

[DLL62] M. Davis, G. Logemann, and D. W. Loveland. A machine program for theorem-proving. *Communications of the ACM*, 5(7):394–397, 1962. 4, 6, 151

[DP60] M. Davis and H. Putnam. A computing procedure for quantification theory. *J. of the ACM Press*, 7(3):201–215, 1960. 4, 6, 151

[DP04] K. Došen and Z. Petrić. *Proof-theoretical Coherence*. King's College Publications, 2004. 37

[Far13] M. Farooque. *Automated reasoning techniques as proof-search in sequent calculus*. PhD thesis, Ecole Polytechnique, 2013. 6, 73, 149, 150, 153, 156, 161, 162, 163, 172, 177, 182

[Fel87] M. Felleisen. *The Calculi of λ-v-CS Conversion: A Syntactic Theory of Control and State in Imperative Higher-Order Programming Languages*. PhD thesis, Department of Computer Science, Indiana University, Bloomington, Indiana, 1987. 3, 24

[FGL13] M. Farooque and S. Graham-Lengrand. Sequent calculi with procedure calls. Technical report, Laboratoire d'informatique de l'École Polytechnique - CNRS, Parsifal - INRIA Saclay, France, 2013. Available at http://hal.archives-ouvertes.fr/hal-00779199 6, 149, 156, 169

[FGLM13] M. Farooque, S. Graham-Lengrand, and A. Mahboubi. A bisimulation between DPLL(T) and a proof-search strategy for the focused sequent calculus. In A. Momigliano, B. Pientka, and R. Pollack, editors, *Proc. of the 2013 Int. Work. on Logical Frameworks and Meta-Languages: Theory and Practice (LFMTP 2013)*. ACM Press, 2013. 6, 149, 153, 161, 162, 163, 169, 172, 173

[Fil89] A. Filinski. Declarative continuations and categorical duality. Master's thesis, DIKU, Computer Science Department, University of Copenhagen, 1989. DIKU Rapport 89/11. 29

[Fis72] M. J. Fischer. Lambda calculus schemata. In *Proc. of the ACM Conf. on Proving Assertions about Programs*, pages 104–109. SIGPLAN Notices, Vol. 7, No 1 and SIGACT News, No 14, 1972. 41, 44, 48

[FL11] M. Farooque and S. Lengrand. A sequent calculus with procedure calls. Technical report, Laboratoire d'informatique de l'École Polytechnique - CNRS, Parsifal - INRIA Saclay, France, 2011. Available at http://hal.archives-ouvertes.fr/hal-00690577 6

[FLM12a] M. Farooque, S. Lengrand, and A. Mahboubi. Simulating the DPLL(T) procedure in a sequent calculus with focusing. Technical report, Laboratoire d'informatique de l'École Polytechnique - CNRS, Microsoft Research - INRIA Joint Centre, Parsifal & TypiCal - INRIA Saclay, France, 2012. Available at http://hal.inria.fr/hal-00690392 6, 149

[FLM12b] M. Farooque, S. Lengrand, and A. Mahboubi. Two simulations about DPLL(T). Technical report, Laboratoire d'informatique de l'École Polytechnique - CNRS, Microsoft Research - INRIA Joint Centre, Parsifal & TypiCal - INRIA Saclay, France, 2012. Available at http://hal.archives-ouvertes.fr/hal-00690044 6, 149

[FP06] C. Fürmann and D. Pym. Order-enriched categorical models of the classical sequent calculus. *J. Pure Appl. Algebra*, 204(1):21–78, 2006. 37

[FP13] J.-C. Filliâtre and A. Paskevich. Why3 - where programs meet provers. In M. Felleisen and P. Gardner, editors, *ESOP*, volume 7792 of *LNCS*, pages 125–128. Springer-Verlag, 2013. 150

[FR94] A. Fleury and C. Retoré. The mix rule. *Math. Structures in Comput. Sci.*, 4(2):273–285, 1994. 35

[Fre79] G. Frege. *Begriffsschrift, eine der arithmetischen nachgebildete Formelsprache des reinen Denkens*. Verlag von Louis Nebert, 1879. 15

[Gen35] G. Gentzen. Investigations into logical deduction. In *Gentzen collected works*, pages 68–131. Ed M. E. Szabo, North Holland, (1969), 1935. 1, 16, 22, 29, 30, 34, 73, 133, 149

[GGL14] D. Galmiche and S. Graham-Lengrand. Special issue on computational logic in honour of Roy Dyckhoff. *Journal of Logic and Computation*, 2014. 5

[GH03] M. Giese and R. Hähnle. Tableaux + constraints. In M. C. Mayer and F. Pirri, editors, *Proc. of the 16th Int. Conf. on Automated Reasoning with Analytic Tableaux and Related Methods (Tableaux'03)*, volume 2796 of *LNCS*. Springer-Verlag, 2003. 182

[Gie00] M. Giese. Proof search without backtracking using instance streams, position paper. In P. Baumgartner and H. Zhang, editors, *3rd Int. Workshop on First-Order Theorem Proving (FTP), St. Andrews, Scotland, TR 5/2000 Univ. of Koblenz*, pages 227–228, 2000. 176

[Gir72] J.-Y. Girard. *Interprétation fonctionelle et élimination des coupures de l'arithmétique d'ordre supérieur.* Thèse d'état, Université Paris 7, 1972. 6, 51, 52, 55, 56

[Gir87] J.-Y. Girard. Linear logic. *Theoret. Comput. Sci.*, 50(1):1–101, 1987. 2, 51, 68, 73, 74, 121, 149

[Gir91] J.-Y. Girard. A new constructive logic: Classical logic. *Math. Structures in Comput. Sci.*, 1(3):255–296, 1991. 73

[GL08] M. Gabbay and S. Lengrand. The lambda-context calculus. volume 196 of *ENTCS*, pages 19–35, 2008. Revision from the Second Int. Work. on Logical Frameworks and Meta-Languages: Theory and Practice (LFMTP 2007) (was *A(nother) NEW Calculus of Contexts*). 7

[GL09] M. Gabbay and S. Lengrand. The λ-context calculus. *Inform. and Comput.*, 207(12):1369–1400, 2009. 7

[GL13] S. Graham-Lengrand. Psyche: a proof-search engine based on sequent calculus with an LCF-style architecture. In D. Galmiche and D. Larchey-Wendling, editors, *Proc. of the 22nd Int. Conf. on Automated Reasoning with Analytic Tableaux and Related Methods (Tableaux'13)*, volume 8123 of *LNCS*, pages 149–156. Springer-Verlag, 2013. 6, 150

[GL14] S. Graham-Lengrand. Polarities & focussing: a journey from realisability to automated reasoning – Coq proofs of Part II. 2014. Available at http://www.lix.polytechnique.fr/~lengrand/Work/HDR/ 5, 121, 128, 140, 180

[GMW79] M. Gordon, R. Milner, and C. Wadsworth. *Edinburgh LCF: a mechanized logic of computation*, volume 78 of *LNCS*. Springer-Verlag, 1979. 150, 167, 169, 179

[Gri90] T. G. Griffin. A formulae-as-type notion of control. In P. Hudak, editor, *17th Annual ACM Symp. on Principles of Programming Languages (POPL'90)*, pages 47–58. ACM Press, 1990. 3, 13, 24

[GTL89] J.-Y. Girard, P. Taylor, and Y. Lafont. *Proofs and Types*, volume 7 of *Cambridge Tracts in Theoret. Comput. Sci.* Cambridge University Press, 1989. 22

[Her05] H. Herbelin. *C'est maintenant qu'on calcule: au coeur de la dualité.* Thèse d'habilitation à diriger des recherches, Université Paris 11, 2005. 29

[HG08] H. Herbelin and S. Ghilezan. An approach to call-by-name delimited continuations. In G. C. Necula and P. Wadler, editors, *Proc. of the 35th Annual ACM Symp. on Principles of Programming Languages (POPL'08)*, pages 383–394. ACM Press, 2008. 49

[Hil28] D. Hilbert. Die Grundlagen der Mathematik. *Abhandlungen aus dem Mathematischen Seminar der Universität Hamburg*, 6(1):65–85, 1928. 15

[Hil31] D. Hilbert. Die Grundlegung der elementaren Zahlenlehre. *Mathematische Annalen*, 104:485–494, 1931. 4, 107

[How80] W. A. Howard. The formulae-as-types notion of construction. In J. P. Seldin and J. R. Hindley, editors, *To H. B. Curry: Essays on Combinatory Logic, Lambda Calculus, and Formalism*, pages 479–490. Academic Press, 1980. Reprint of a manuscript written 1969. 1, 13, 16, 18, 133

[HS97] M. Hofmann and T. Streicher. Continuation models are universal for $\lambda\mu$-calculus. In *Proc. of the 12th Annual IEEE Symp. on Logic in Computer Science*, pages 387–397. IEEE Computer Society Press, 1997. 41, 44, 48, 49

[Hyl82] J. Hyland. The effective topos. In A. Troelstra and D. V. Dalen, editors, *The L.E.J. Brouwer Centenary Symposium*, pages 165–216. North Holland Publishing Company, 1982. 6

[HZ09] H. Herbelin and S. Zimmermann. An operational account of call-by-value minimal and classical λ-calculus in "natural deduction" form. In P. Curien, editor, *Proc. of the 9th Int. Conf. on Typed Lambda Calculus and Applications (TLCA'09)*, volume 5608 of *LNCS*, pages 142–156. Springer-Verlag, 2009. 49

[Isa] The Isabelle theorem prover. Available at http://isabelle.in.tum.de/ 169

[JM94] J. Jaffar and M. J. Maher. Constraint logic programming: a survey. *J. Logic Programming*, 19–20, Supplement 1(0):503 – 581, 1994. Special Issue: Ten Years of Logic Programming. 182

[Joh36] I. Johansson. Der Minimalkalkül, ein reduzierter intuitionistischer Formalismus. *Compositio Math.*, 4:119–136, 1936. 15

[KL08] K. Kikuchi and S. Lengrand. Strong normalisation of cut-elimination that simulates β-reduction. In R. Amadio, editor, *Proc. of the 11th Int. Conf. on Foundations of Software Science and Computation Structures (FOSSACS'08)*, volume 4962 of *LNCS*, pages 380–394. Springer-Verlag, 2008. 5

[Kle45] S. Kleene. On the interpretation of intuitionistic number theory. *J. of Symbolic Logic*, 10:109–124, 1945. 2

[Kri71] J.-L. Krivine. *Introduction to axiomatic set theory.* Dordrecht, Reidel, 1971. 9

[Kri01] J.-L. Krivine. Typed lambda-calculus in classical Zermelo-Frænkel set theory. *Arch. Math. Log.*, 40(3):189–205, 2001. 2, 3, 51, 53, 121, 179

[Lam07] F. Lamarche. Exploring the Gap between Linear and Classical Logic. *Theory and Applications of Categories*, 18(17):473–535, 2007. 37

[Lau02] O. Laurent. *Etude de la polarisation en logique*. Thèse de doctorat, Université Aix-Marseille II, 2002. 1, 73

[LC09] S. Lescuyer and S. Conchon. Improving Coq propositional reasoning using a lazy CNF conversion scheme. In *Proc. of the 7th Int. Conf. on Frontiers of combining systems (FroCoS'09)*, pages 287–303. Springer-Verlag, 2009. 152

[LDM11] S. Lengrand, R. Dyckhoff, and J. McKinna. A focused sequent calculus framework for proof search in Pure Type Systems. *Logic. Methods Comput. Science*, 7(1), 2011. 6, 149

[Len03] S. Lengrand. Call-by-value, call-by-name, and strong normalization for the classical sequent calculus. volume 86(4) of *ENTCS*. Elsevier, 2003. Revision from the 3rd Int. Work. on Reduction Strategies in Rewriting and Programming (WRS'03). 5, 29, 34

[Len06] S. Lengrand. *Normalisation & Equivalence in Proof Theory & Type Theory*. PhD thesis, Université Paris 7 & University of St Andrews, 2006. 7, 9, 40

[Len08] S. Lengrand. Termination of lambda-calculus with the extra call-by-value rule known as assoc. Technical report, LIX, CNRS-INRIA-Ecole Polytechnique, 2008. Available at http://hal.inria.fr/inria-00292029 5

[LM08] S. Lengrand and A. Miquel. Classical F_ω, orthogonality and symmetric candidates. *Ann. Pure Appl. Logic*, 153:3–20, 2008. 3, 6, 51, 58, 60, 121

[LM09] C. Liang and D. Miller. Focusing and polarization in linear, intuitionistic, and classical logics. *Theoret. Comput. Sci.*, 410(46):4747–4768, 2009. 1, 3, 69, 71, 74, 75, 95, 96, 102, 104, 149, 152, 153, 179

[LM11] C. Liang and D. Miller. A focused approach to combining logics. *Ann. Pure Appl. Logic*, 162(9):679–697, 2011. 95, 96, 97, 98, 104

[LQdF05] O. Laurent, M. Quatrini, and L. T. de Falco. Polarized and focalized linear and classical proofs. *Ann. Pure Appl. Logic*, 134(2-3):217–264, 2005. 1, 73

[LS86] J. Lambek and P. J. Scott. *Introduction to Higher Order Categorical Logic*. Cambridge University Press, 1986. 22

[LS05] F. Lamarche and L. Straßburger. Constructing free boolean categories. In P. Panangaden, editor, *Proc. of the 20th Annual IEEE Symp. on Logic in Computer Science*, pages 209–218. IEEE Computer Society Press, 2005. 37

[Miq09] A. Miquel. Relating classical realizability and negative translation for existential witness extraction. In P. Curien, editor, *Proc. of the 9th Int. Conf. on Typed Lambda Calculus and Applications (TLCA'09)*, volume 5608 of *LNCS*, pages 188–202. Springer-Verlag, 2009. 3, 6, 52, 60, 64

[Miq11] A. Miquel. Existential witness extraction in classical realizability and via a negative translation. *Logic. Methods Comput. Science*, 7(2), 2011. 3, 6, 52, 60, 64, 179

[ML82] P. Martin-Löf. Constructive mathematics and computer programming. In *Proc. of the Sixth Int. Congress for Logic, Methodology, and Philosophy of Science*, pages 153–175. North-Holland, 1982. 13

[ML84] P. Martin-Löf. *Intuitionistic Type Theory*. Number 1 in Studies in Proof Theory, Lecture Notes. Bibliopolis, 1984. 13

[MM09] G. Munch-Maccagnoni. Focalisation and classical realisability. In E. Grädel and R. Kahle, editors, *Proc. of the 18th Annual Conf. of the European Association for Computer Science Logic (CSL'09)*, volume 5771 of *LNCS*, pages 409–423. Springer-Verlag, 2009. 1, 3, 6, 44, 45, 51, 54, 57, 71, 73, 84, 87, 121, 179

[MM13] G. Munch-Maccagnoni. *Syntax and Models of a Non-Associative Composition of Programs and Proofs*. PhD thesis, Université Paris Diderot – Paris 7, 2013.57, 71

[MMZ+01] M. W. Moskewicz, C. F. Madigan, Y. Zhao, L. Zhang, and S. Malik. Chaff: Engineering an efficient SAT solver. In *DAC*, pages 530–535. ACM Press, 2001.173

[MNPS91] D. Miller, G. Nadathur, F. Pfenning, and A. Scedrov. Uniform proofs as a foundation for logic programming. *Ann. Pure Appl. Logic*, 51:125–157, 1991. 1, 2, 149

[Mog89] E. Moggi. Computational lambda-calculus and monads. In *Proc. of the 4th Annual IEEE Symp. on Logic in Computer Science*, pages 14–23. IEEE Computer Society Press, 1989. 37, 40

[MP08] S. McLaughlin and F. Pfenning. Imogen: Focusing the polarized inverse method for intuitionistic propositional logic. In I. Cervesato, H. Veith, and A. Voronkov, editors, *Proc. of the the 15th Int. Conf. on Logic for Programming Artificial Intelligence and Reasoning (LPAR'08)*, volume 5330 of *LNCS*, pages 174–181. Springer-Verlag, 2008. 3, 4

[MV05] P.-A. Melliès and J. Vouillon. Recursive polymorphic types and parametricity in an operational framework. In P. Panangaden, editor, *Proc. of the 20th Annual IEEE Symp. on Logic in Computer Science*, pages 82–91. IEEE Computer Society Press, 2005. 51

[Nig09] V. Nigam. *Exploiting non-canonicity in the sequent calculus*. PhD thesis, Ecole Polytechnique, 2009. 93, 94

[NM10] V. Nigam and D. Miller. A framework for proof systems. *J. of Automated Reasoning*, 45(2):157–188, 2010. 93, 94

[NOT06] R. Nieuwenhuis, A. Oliveras, and C. Tinelli. Solving SAT and SAT Modulo Theories: From an abstract Davis–Putnam–Logemann–Loveland procedure to DPLL(T). *J. of the ACM Press*, 53(6):937–977, 2006. 4, 151, 152, 157, 158, 163, 173, 179

[NPW02] T. Nipkow, L. C. Paulson, and M. Wenzel. *Isabelle/HOL — A Proof Assistant for Higher-Order Logic*, volume 2283 of *LNCS*. Springer-Verlag, 2002. 169

[Par92] M. Parigot. $\lambda\mu$-calculus: An algorithmic interpretation of classical natural deduction. In A. Voronkov, editor, *Proc. of the Int. Conf. on Logic Programming and Automated Reasoning (LPAR'92)*, volume 624 of *LNCS*, pages 190–201. Springer-Verlag, 1992. 3, 25, 30

[Par97] M. Parigot. Proofs of strong normalisation for second order classical natural deduction. *J. of Symbolic Logic*, 62(4):1461–1479, 1997. 3, 51, 55, 56, 179

[PB12] L. C. Paulson and J. C. Blanchette. Three years of experience with Sledgehammer, a practical link between automatic and interactive theorem provers. In G. Sutcliffe, S. Schulz, and E. Ternovska, editors, *IWIL 2010*, volume 2 of *EPiC Series*, pages 1–11. EasyChair, 2012. 169

[Pit03] A. M. Pitts. Nominal logic, a first order theory of names and binding. *Inform. and Control*, 186:165–193, 2003. 7, 91

[Plo75] G. D. Plotkin. Call-by-name, call-by-value and the lambda-calculus. *Theoret. Comput. Sci.*, 1:125–159, 1975. 3, 37, 38, 39

[Pol04] E. Polonovski. Strong normalization of $\lambda\mu\tilde{\mu}$-calculus with explicit substitutions. In I. Walukiewicz, editor, *Proc. of the 7th Int. Conf. on Foundations of Software Science and Computation Structures (FOSSACS'04)*, volume 2987 of *LNCS*, pages 423–437. Springer-Verlag, 2004. 34

[PSI] The PSI project. 2009-2013. http://www.lix.polytechnique.fr/~lengrand/PSI. 6, 149, 150

[Psy] Psyche: the Proof-Search factorY for Collaborative HEuristics. Available at http://www.lix.polytechnique.fr/~lengrand/Psyche 3, 4, 6, 167, 177, 179

[Rea00] S. Read. Harmony and autonomy in classical logic. *J. of Philosophical Logic*, 29(2):123–154, 2000. 81

[Rea10] S. Read. General-elimination harmony and the meaning of the logical constants. *J. of Philosophical Logic*, 39(5):557–576, 2010. 81

[Rey72] J. C. Reynolds. Definitional interpreters for higher-order programming languages. In *Proc. of the ACM annual Conf.* , pages 717–740, 1972. 3, 23, 39, 44, 51

[Roc05] J. Rocheteau. lambda-μ-calculus and duality: Call-by-name and call-by-value. In J. Giesl, editor, *Proc. of the 16th Int. Conf. on Rewriting Techniques and Applications (RTA'05)*, volume 3467 of *LNCS*, pages 204–218. Springer-Verlag, 2005. 29

[RV01] J. A. Robinson and A. Voronkov, editors. *Handbook of Automated Reasoning (in 2 volumes)*. Elsevier and The MIT Press, 2001. 2, 182, 184

[Sau05] A. Saurin. Separation with streams in the lambdaμ-calculus. In P. Panangaden, editor, *Proc. of the 20th Annual IEEE Symp. on Logic in Computer Science*, pages 356–365. IEEE Computer Society Press, 2005. 49

[Sau08] A. Saurin. On the relations between the syntactic theories of lambda-mu-calculi. In *Proc. of the 17th Annual Conf. of the European Association for Computer Science Logic (CSL'08)*, volume 5213 of *LNCS*, pages 154–168. Springer-Verlag, 2008. 49

[Sau10a] A. Saurin. A hierarchy for delimited continuations in call-by-name. In C. L. Ong, editor, *Proc. of the 13th Int. Conf. on Foundations of Software Science and Computation Structures (FOSSACS'10)*, volume 6014 of *LNCS*, pages 374–388. Springer-Verlag, 2010. 49

[Sau10b] A. Saurin. Standardization and böhm trees for lambdaμ-calculus. In M. Blume, N. Kobayashi, and G. Vidal, editors, *Proc. of the 10th Int. Symp. Functional and Logic Programming (FLOPS'10)*, volume 6009 of *LNCS*, pages 134–149. Springer-Verlag, 2010. 49

[Sau10c] A. Saurin. Typing streams in the lambdaμ-calculus. *ACM Transactions on Computational Logic*, 11(4), 2010. 49

[Sau12] A. Saurin. Böhm theorem and böhm trees for the $\lambda\mu$-calculus. *Theoret. Comput. Sci.*, 435:106–138, 2012. 49

[Sch50] K. Schütte. Beweistheoretische Erfassung der unendlichen Induktion in der Zahlentheorie. *Mathematische Annalen*, 122:369–389, 1950. 4, 107

[Sel01] P. Selinger. Control categories and duality: on the categorical semantics of the $\lambda\mu$-calculus. *Math. Structures in Comput. Sci.*, 11(2):207–260, 2001. 3, 42, 49, 72, 73, 179

[SR98] T. Streicher and B. Reus. Classical logic, continuation semantics and abstract machines. *J. Funct. Programming*, 8(6):543–572, 1998. 49

[Str11] L. Straßburger. *Towards a Theory of Proofs of Classical Logic*. Habilitation thesis, Université Denis Diderot – Paris 7, 2011. 22, 37

[SU06] M. H. B. Sørensen and P. Urzyczyn. *Lectures on the Curry-Howard Isomorphism*. Studies in Logic and the Foundations of Mathematics. Elsevier, 2006. 14

[SW00] C. Strachey and C. P. Wadsworth. Continuations: A mathematical semantics for handling fulljumps. *Higher-Order and Symbolic Computation*, 13:135–152, 2000. 3, 23

[Tai67] W. W. Tait. Intensional interpretations of functionals of finite type I. *J. of Symbolic Logic*, 32:198–212, 1967. 51

[Tai75] W. W. Tait. A realizability interpretation of the theory of species. In *Logic Colloquium*, volume 453 of *LNM*, pages 240–251. Springer-Verlag, 1975.51, 52, 55, 56

[Ten78] N. Tennant. *Natural logic*. Edinburgh University Press, 1978. 81

[Ter03] Terese. *Term Rewriting Systems*, volume 55 of *Cambridge Tracts in Theoret. Comput. Sci.* Cambridge University Press, 2003. 9

[Tin07] C. Tinelli. An abstract framework for satisfiability modulo theories. In N. Olivetti, editor, *Proc. of the 16th Int. Conf. on Automated Reasoning with Analytic Tableaux and Related Methods (Tableaux'07)*, volume 4548 of *LNCS*. Springer-Verlag, 2007. Invited talk, available at http://ftp.cs.uiowa.edu/pub/tinelli/talks/TABLEAUX-07.pdf. 150

[TS00] A. S. Troelstra and H. Schwichtenberg. *Basic Proof Theory*. Cambridge University Press, 2000. 10, 30, 34, 35, 66

[Twe] The Twelf Project. Available at http://twelf.org 149

[Urb00] C. Urban. *Classical Logic and Computation*. PhD thesis, University of Cambridge, 2000. 5, 29, 30

[VO02] J. VAN OOSTEN. Realizability: a historical essay. *Math. Structures in Comput. Sci.*, 12:239–263, 2002. 2

[Wad03] P. Wadler. Call-by-value is dual to call-by-name. In *Proc. of the 8th ACM SIGPLAN Int. Conf. on Functional programming (ICFP'03)*, volume 38(9), pages 189–201. ACM Press, 2003. 29, 31, 44, 45, 47, 70

[Web11] T. Weber. SMT solvers: New oracles for the HOL theorem prover. *International Journal on Software Tools for Technology Transfer (STTT)*, 13(5):419–429, 2011. 152, 169

[Zei08a] N. Zeilberger. Focusing and higher-order abstract syntax. In G. C. Necula and P. Wadler, editors, *Proc. of the 35th Annual ACM Symp. on Principles of Programming Languages (POPL'08)*, pages 359–369. ACM Press, 2008. 3, 6, 74, 77, 79, 87, 88, 89, 179

[Zei08b] N. Zeilberger. On the unity of duality. *Ann. Pure Appl. Logic*, 153(1-3):66–96, 2008. 3, 6, 74, 77, 79, 87, 88, 89, 179

[Zei09] N. Zeilberger. *The Logical Basis of Evaluation Order and Pattern-Matching*. PhD thesis, Carnegie Mellon University, 2009. 133

[Zei10] N. Zeilberger. Polarity and the logic of delimited continuations. In J.-P. Jouannaud, editor, *Proc. of the 25th Annual IEEE Symp. on Logic in Computer Science*, pages 219–227. IEEE Computer Society Press, 2010. 77, 79

Index

List of Figures

Appendix A

Basic definitions for categories

Definition 130 (Category) A *category* is the combination of

- a class of elements called *objects*, denoted A, B,...
- for every pair of objects A and B, a class $\mathsf{hom}(A, B)$ of elements called *morphisms from A to B*; the expression $f\colon A \longrightarrow B$ denotes that f is a morphism from A to B;
- for every object A, a morphism Id_A called *identity*;
- for every objects A, B, C, a binary operation called *composition* mapping every $f\colon A \longrightarrow B$ and $g\colon B \longrightarrow C$ to a morphism $f \cdot g\colon A \longrightarrow C$

such that the following properties holds

- composition is associative $((f \cdot g) \cdot h = f \cdot (g \cdot h))$
- identities are units for composition $(\mathsf{Id}_A \cdot f = f \cdot \mathsf{Id}_A = f)$.

Given a category, we often use *diagrams* to represent a collection of objects -represented as vertices- and morphisms -represented as labelled arrows between vertices. Using morphism composition, each path between any two given vertices unambiguously represents a morphism. A diagram *commutes* when for each pair of vertices A and B, all paths from A to B represent equal morphisms.

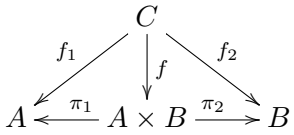
Two objects A and B are *isomorphic* when there are two morphisms $f\colon A \longrightarrow B$ and $g\colon B \longrightarrow A$ such that $f \cdot g = \mathsf{Id}_A$ and $g \cdot f = \mathsf{Id}_B$. ※

Standard examples of categories are: the categories of sets and functions (objects are sets and morphisms from A to B are functions from A to B), the category of sets and relations (objects are sets and morphisms from A to B are relations from A to B), etc. Groups form a particular kind of categories (where there is only one object, and elements of the group are the morphisms from that object to itself). Partially ordered sets form another particular kind of categories (where there is at most one morphism between any two objects), etc.

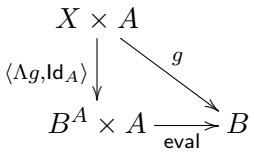
DEFINITION 131 (Cartesian Closed Category)

A *Cartesian Closed Category* (CCC), is a category such that

- there is an object, denoted 1 and called *terminal object*, such that, for each object A, there is a unique morphism $1_A \colon A \longrightarrow 1$;
- for every two objects A and B, there is an object, denoted $A \times B$ and called the *product* of A and B, together with two morphisms $\pi_1 \colon A \times B \longrightarrow A$ and $\pi_2 \colon A \times B \longrightarrow B$, called the first and second *projections*, satisfying the following property:
  for every object C and morphisms $f_1 \colon C \longrightarrow A$ and $f_2 \colon C \longrightarrow B$, there is a unique $f \colon C \longrightarrow A \times B$, denoted $\langle f_1, f_2 \rangle$, such that the following diagram commutes



- for every two objects A and B, there is an object, denoted B^A and called the *exponential* of A and B, together with a morphism $\mathsf{eval} \colon B^A \times A \longrightarrow B$, satisfying the following property:
  for every object X and morphism $g \colon X \times A \longrightarrow B$ there is a unique $f \colon X \longrightarrow B^A$, denoted Λg, such that the following diagram commutes



We choose the convention that products are associative to the left, i.e. $(A \times B) \times C$ can be abbreviated as $A \times B \times C$. A family of morphisms $\pi_{i/n} \colon A_1 \times \cdots \times A_n \longrightarrow A_i$, for $1 \leq i \leq n$, can be defined by composing the two projections in the obvious way:

$$\begin{array}{lll} \pi_{1/1} & := & \mathsf{Id}_{A_1} \\ \pi_{n/n} & := & \pi_2 \qquad\qquad \text{when } 1 < n \\ \pi_{p/n} & := & \pi_1 \cdot \pi_{p/n-1} \quad \text{when } p < n \end{array}$$

※

REMARK 83 One can quickly check that the terminal object, products and exponentials are unique up to isomorphism (i.e. two objects satisfying the property of the terminal object, or the product / exponential object for a given A and B, are isomorphic); hence the notations 1, $A \times B$, B^A. ※