PARAMETRIC POLYMORPHISM FOR XML PROCESSING LANGUAGES

Zhiwu Xu

▶ **To cite this version:**

Zhiwu Xu. PARAMETRIC POLYMORPHISM FOR XML PROCESSING LANGUAGES. Programming Languages [cs.PL]. Université Paris-Diderot - Paris VII, 2013. English. NNT : . tel-00858744

HAL Id: tel-00858744

https://theses.hal.science/tel-00858744

Submitted on 6 Sep 2013

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

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

UNIVERSITÉ PARIS DIDEROT-PARIS 7

ÉCOLE DOCTORALE DE
SCIENCES MATHEMATIQUES DE PARIS CENTRE

COTUTELLE DE THÈSE

pour obtenir le double diplôme de

DOCTEUR DE L'UNIVERSITÉ PARIS DIDEROT

et

DOCTEUR DE L'UNIVERSITÉ DE L'ACADÉMIE CHINOISE DES SCIENCES

Discipline: INFORMATIQUE

POLYMORPHISME PARAMÉTRIQUE POUR LE TRAITEMENT DE DOCUMENTS XML

Zhiwu XU

Soutenue le 30 Mai 2013, devant le jury composé de

Frederic BLANQUI	Examinateur
Giuseppe CASTAGNA	Directeur de thèse
Haiming CHEN	Co-Directeur de thèse
Mariangiola DEZANI	Rapportrice
Xinyu FENG	Rapporteur
Giorgio GHELLI	Rapporteur
Huiming LIN	Président du jury

UNIVERSITY OF PARIS DIDEROT-PARIS 7
UNIVERSITY OF CHINESE ACADEMY OF SCIENCES

JOINT DISSERTATION

for obtaining the dual degrees of

DOCTOR OF THE UNIVERSITY OF PARIS DIDEROT

and

DOCTOR OF UNIVERSITY OF CHINESE ACADEMY OF SCIENCES

Discipline: COMPUTER SCIENCE

PARAMETRIC POLYMORPHISM FOR XML PROCESSING LANGUAGES

Written by

Zhiwu XU

Supervised by

Giuseppe CASTAGNA University of Paris Diderot
Haiming CHEN Institute of Software Chinese Academy of Sciences

Laboratoire Preuves, Programmes et Systèmes
University of Paris Diderot
State Key Laboratory of Computer Science
Institute of Software Chinese Academy of Sciences

Acknowledgments

I am especially indebted to my supervisor Giuseppe Castagna, without whose supervision and support, this thesis is not possible. He exposed me to a wealth of knowledge in programming languages, and has taught me by example the skills of a good researcher. I also thank him for his help in livings when I arrived in Paris.

I am very grateful to my supervisor Haiming Chen, who supported me in starting my study in Institute of Software, Chinese Academic of Sciences. I thank Haiming for his supervision when I am in Beijing and for his support on my co-joint agreement.

I thank my collaborators, Kim Nguyen and Serguei Lenglet, for many fruitful and joyful discussions, which gave me a great deal of inspiration.

Many persons discussed the contents of this work with us and gave precious suggestions. Nino Salibra showed us how to prove that all powersets of infinite sets are convex models of the propositional logic. Christine Paulin suggested and proved a weaker condition for convex models. Pietro Abate implemented an early version of our prototype subtype checker. Bow-Yaw Wang drew our attention to the connection with convex theories. Claude Benzaken pointed out several useful results in combinatorics to us. Nils Gesbert spotted a subtle mistake in our subtyping algorithm. Special thanks to Véronique Benzaken, Mariangiola Dezani-Ciancaglini, Daniele Varacca, Philip Wadler, and Luca Padovani.

When I worked in PPS and SKLCS, I met a lot of wonderful friends — you know who you are — who share my enthusiasm for computer science and who gave me lots of help in work.

Finally, but not least, I must thank my family. Without you, this thesis would never have been done.

I dedicate this thesis to my family.

Abstract

XML (eXtensible Markup Language) is a current standard format for exchanging semi-structured data, which has been applied to web services, database, research on formal methods, and so on. For a better processing of XML, recently there emerge many statically typed functional languages, such as XDuce, ℂDuce, XJ, XTatic, XACT, XHaskell, OCamlDuce and so on. But most of these languages lack parametric polymorphism or present it in a limited form. While parametric polymorphism is needed by XML processing, this is witnessed by the fact that it has repeatedly been requested to and discussed in various working groups of standards (*e.g.*, RELAX NG and XQuery). We study in this thesis the techniques to extend parametric polymorphism into XML processing languages.

Our solution consists of two parts: a definition of a polymorphic semantic subtyping relation and a definition of a polymorphic calculus. In the first part, we define and study a polymorphic semantic subtyping relation for a type system with recursive, product and arrow types and set-theoretic type connectives (*i.e.*, intersection, union and negation). We introduce the notion of "convexity" on which our solution is built up and prove there exists at least one model that satisfies convexity. We also propose a sound, complete and terminating subtyping algorithm. The second part is devoted to the theoretical definition of a polymorphic calculus, which takes advance of the subtyping relation. The novelty of the polymorphic calculus is to decorate λ-abstractions with sets of type-substitutions and to lazily propagate type-substitutions at the moment of the reduction. The second part also explores a semi-decidable local inference algorithm to infer the set of type-substitutions as well as the compilation of the polymorphic calculus into a variety of ℂDuce.

Résumé

XML (eXtensible Markup Language) est un format standard pour l'échange de données semi-structurées, qui est utilisé dans services web, les bases de données, et comme format de sérialisation pour échanger des données entre applications. Afin d'avoir un meilleur traitement de données XML, plusieurs langages statiquement typés pour XML ont été récemment définis, tels XDuce, ℂDuce, XJ, XTatic, XACT, XHaskell, OCaml-Duce. Ces langages peuvent vérifier si un programme n'engendra d'erreurs de types à l'exécution. Mais la plupart de ces langages n'incluent pas le polymorphisme paramétrique ou l'incluent sous un forme très limitée. Cependant, le traitement de données XML nécessite du polymorphisme paramétrique, c'est pourquoi il a été demandé et discuté à plusieurs reprises dans diverses groupes de travail de standardisation (par example, RELAX NG et XQuery) .

Nous étudions dans cette thèse les techniques pour étendre par le polymorphisme paramétrique les langages de traitement XML. Notre solution se deroule sur deux étapes : (*i*) nous définissons et étudions une relation de sous-typage polymorphe sémantique pour un système de type avec types récursifs, types des produits, types des flèches, et types des ensemblistes (c'est-à-dire, l'union, l'intersection et la négation) ; et (*ii*) nous concevons et étudions un langage fonctionnel d'ordre supérieur qui tire pleinement parti des nouvelles fonctionnalités du système de type.

La solution que nous proposons dans cette thèse est générale. Ainsi elle a des domaines d'application autres que les langages pour le traitement de données XML.

Sous-typage polymorphe sémantique

Une approche naïf d'étendre l'approche monomorphisme avec polymorphisme consiste à réutiliser l'approche monomorphisme en considérant toutes les instances de sol de types polymorphes (c'est-à-dire, les substitutions des variables de type dans les types de sol), mais l'extension fournit une relation de sous-typage qui souffre de tant de problèmes qui se révèle être inutile. Le nœud du problème est le comportement de « bégaiement » de types indivisibles, c'est-à-dire, les types non vides dont seul sous-type strict est le type de vide. Généralement, un type indivisible est soit complètement à l'intérieur soit complètement à l'extérieur de tout autre type. Notre solution est de faire de types indivisibles « sécable » afin que les variables de type peuvent varier sur des sous-ensembles stricts de n'importe quel type. Formellement, nous définissons assignement sémantique pour les variables de type comme substitutions des variables de types dans sous-ensembles de n'importe quel type. Ainsi, nous définissons la relation de sous-typage comme l'inclusion des ensembles dénotés sous toutes les assignement pour

les variables de type.

Pour caractériser sémantiquement les modèles où le bégaiement est absent, nous introduisons une propriété de « convexité ». La convexité impose la relation de sous-typage pour avoir un comportement uniforme, donc nous estimons qu'il existe des liens solides entre la convexité et la paramétricité. Nous montrons aussi qu'il existe au moins un modèle de ensembliste qui est convexe. En fait, il existe beaucoup de modèles convexes car chaque modèle où tous les types de non vide dénotent qu'un ensemble infini est convexe.

Enfin, basée sur la théorie des ensembles et la convexité, nous proposons un algorithme de sous-typage, ce qui est correct, complète et décidable. L'idée est de remplacer les variables de type survenant au niveau supérieur par type structurel (c'est-à-dire, types de produits et de flèche) qui sont composées de nouvelles variables de type, et puis de décomposer les constructeurs de types au niveau supérieur afin de récursivité ou arrêter.

Calcul polymorphe

Le langage spéciale que nous cherchons à définir dans cette thèse est une version polymorphe de $\mathbb{C}$Duce. Dans $\mathbb{C}$Duce, λ-abstractions sont tapées par les intersections des types de flèche généralement pour taper surcharge. La nouveauté de la version polymorphe est de permettre à des variables de type à se produire dans les types et donc dans les types d'étiquetage dans les λ-abstractions. En conséquence, nous pouvons définir des fonctions polymorphes et appliquez-les. Avant d'appliquer une fonction polymorphe, il faut ré-étiqueter (instancier) via une substitution de type approprié. En raison de types d'intersection, nous pouvons renommer une fonction polymorphe via plusieurs substitutions de type, à savoir que via un *ensemble* de substitutions de type. Mais ce qui rend la voie de la tradition, qui applique un ensemble de substitutions de type à tous les types d'étiquetage dans une fonction, ne fonctionnent plus. Notre solution est d'effectuer un « paresseux » ré-étiquetage en décorant des λ-abstractions avec des ensembles de substitutions de type et de propager paresseusement les substitutions de type au moment de la réduction. Notre calcul est aussi un λ-calcul explicitement typé avec types d'intersection.

Deux types, le problème de correspondance de type est de vérifier s'il existe une substitution telle que l'instance du premier type est un sous-type de l'instance du second type. Nous considérons le problème de correspondance de type comme un problème de satisfaction de contraintes. Basée sur la théorie des ensembles et l'algorithme de sous-typage, nous proposons un algorithme de correspondance de type, ce qui est correct, complète et décidable. L'idée principale est d'isoler les variables de type survenant au niveau supérieur, ce qui donne des contraintes sur ces variables de type ; et puis d'extraire une substitution de ces contraintes.

En pratique, les ensembles de substitutions de type sont transparentes pour le programmeur. Pour cela, nous proposons un système d'inférence, ce qui infère où et quelles substitutions de type pouvant être insérées dans une expression pour la rendre bien typé. La difficulté réside dans commment taper une application, car il doit vérifier si les types de fonction et de son argument peuvent être rendues compatibles. Dans notre cadre,

pour faire une application bien typé, nous devons trouver *deux ensembles* de substitutions de type qui produisent des instances dont les intersections sont en la relation de sous-typage droit. Supposons que les cardinalités de ces deux ensembles sont fixes. Ensuite, nous pouvons réduire le problème de typage application à une instance de le problème de correspondance de type en l'α-renommage. Cela donne immédiatement une procédure semi-décision qui essaie toutes les cardinalités pour le système d'inférence. Par ailleurs, afin d'assurer la résiliation, nous proposons deux nombres heuristiques pour les cardinalités de ces deux ensembles qui sont établis selon les formes des types de la fonction et de son argument. L'intuition est que ces connecteurs de type sont ce qui nous font instancier un type polymorphe plusieurs fois.

Enfin, afin de fournir un modèle d'exécution pour notre calcul, nous étudions la traduction de notre calcul polymorphe dans un calcul monomorphisme (par exemple, une variante de $\mathbb{C}$Duce). L'idée principale est d'utiliser l'expression case type pour simuler les différentes ré-étiquetage sur l'expression du corps d'une λ-abstraction avec différents cas type. Nous montrons aussi que la traduction est correct.

Contents

List of Figures

List of Tables

List of Symbols

Part I

Introduction

Chapter 1

Introduction

1.1 XML

XML (Extensible Markup Language) is a simple but flexible text format to structure documents [XML], standardized by World Wide Web Consortium (W3C). XML is a markup language much like HTML [HTM99]. But unlike HTML, XML is extensible and allows users to create their own tags based on the data processed. Due to its flexibility and extensibility, XML has become popular for web services, database, research on formal methods, and other applications. For example, XML has come into common use for the interchange of data over the Internet.

An XML document is a text file where tags are inserted so that the whole document forms a tree structure. That is to say, an XML document can be viewed as a tree of variable arities in a natural way, where nodes (also called elements) are markups, leaves are raw text (also called PCDATA). For example, a simple XML document and its tree representation are given in Figure 1.1. This document represents an address book. In this document "`addrbook`", "`person`", "`name`", and "`email`" are tag names (labels). A node (element) is represented as a pair of an opening tag `<label>` and a closing tag `</label>`.

The XML specification defines an XML document as a text that is well-formed [XML]. For example, XML elements must be properly nested (*i.e.*, tags must be well parenthesized). Of course, the specification specifies the *syntactic* constraints. Besides the specification, there are some *semantic* constraints to impose on an XML document. Then a well-formed document which satisfies these semantic constraints is said to be valid. There are many *schema languages* to describe such constraints, such as DTD (Document Type Definitions) [DTD06], RELAX NG [CM01], XML-Schema [XSc], DSD (Document Structure Description) [KMS00], etc. With these schema languages, such additional constraints can be specified: restrain the set of possible tags, specify the order of appearance of tags, the content of a tag (*e.g.*, a tag `<a>` must only contain elements of tag `<b>` or characters) and so on. In XML processing, one can validate an XML document against a given schema (*e.g.*, DTD). For example, Figure 1.2 shows a DTD which accepts documents similar to the one of the address book in Figure 1.1.

```
<addrbook>
   <person>
      <name>Giuseppe Castagna</name>
      <email>Giuseppe.Castagna@pps.jussieu.fr</email>
      <email>gc@pps.jussieu.fr</email>
   </person>
   <person>
      <name>Zhiwu Xu</name>
      <email>Zhiwu.Xu@pps.jussieu.fr</email>
      <tel>06-12-34-56-78</tel>
   </person>
</addrbook>
```

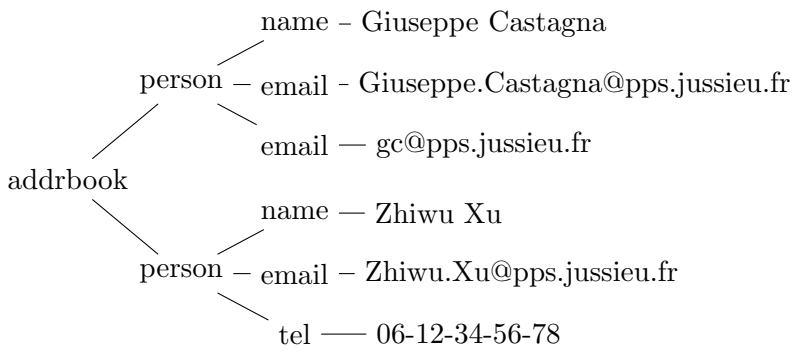

Figure 1.1: An XML document and its tree representation

```
<!ELEMENT addrbook (person*)>
<!ELEMENT person   (name, email*,tel?)>
<!ELEMENT name     #PCDATA>
<!ELEMENT email    #PCDATA>
<!ELEMENT tel      #PCDATA>
```

Figure 1.2: An example of DTD

1.2 Statically typed languages for XML

Many tools, libraries, and languages have been developed to help developing applications that manipulate XML documents. But most of them are error-prone or lack any flexibility. For a better programming with XML and due to the fact that DTD, Relax-NG or XML schema are some classes of regular tree languages [MLM01], recently there emerged several concrete statically typed languages for XML processing, taking XML documents as first-class values and XML schemas as XML types, such as XDuce [HP03],

XQuery [BCF+03b], ℂDuce [BCF03a], XJ [HRS+05], XTatic [GLPS05a], XACT [KM06], XHaskell [SZL07] and OCamlDuce [Fri06]. Moreover, programs are analyzed at compile time (*i.e.*, statically). When a program passes type-checking, it is guaranteed that the program runs well and generates outputs of the expected type. For example, if an XML transformation (*i.e.*, a function) has been checked to have type $t \to s$, then for any input of type t, the output of the transformation will always have type s, without any further validation.

XDuce [Hos01, HP01] was the first domain specific programming language with static type-checking of XML operations. In XDuce, XML documents are first class values and types are *regular expression types*. Regular expression types are nothing else but regular tree languages [CDG+97]. The most essential part of the type system is the definition of *semantic subtyping*, which is uniquely suited to finely check constraints on XML documents. In Figure 1.3, we can see a sample of code[1], which defines the address book document (value) and its corresponding DTD (type). Type constructors of the form `<label>[` R `]` classify tree nodes with the tag name `label`, where R denotes a regular expression on types.

```
type Addrbook = <addrbook>[Person*]
type Person   = <person>[Name,Email*,Tel?]
type Name     = <name>[String]
type Email    = <email>[String]
type Tel      = <tel>[String]

<addrbook>[
   <person>[
      <name>[Giuseppe Castagna]
      <email>[Giuseppe.Castagna@pps.jussieu.fr]
      <email>[gc@pps.jussieu.fr]
   ]
   <person>[
      <name>[Zhiwu Xu]
      <email>[Zhiwu.Xu@pps.jussieu.fr]
      <tel>[06-12-34-56-78]
   ]
]
```

Figure 1.3: ℂDuce definitions of address book and its type

Another important feature is the definition of *regular expression pattern matching*, a generalization of *pattern matching* as found in many functional programming languages (*e.g.*, SML, OCaml, and Haskell), which selects efficiently sub-parts of an input document in a precisely typed way.

[1] This code is actually a ℂDuce code, whose syntax is close to the one of XDuce.

XDuce has been further extended in many different directions. One extension is XTatic [GLPS05a], which aims at integrating the main ideas from XDuce into C♯. Another extension is the ℂDuce language [BCF03a], whose syntax is similar to that of XDuce. The ℂDuce language attempts to extend XDuce towards being a general-purpose functional language. To this end, ℂDuce extended semantic subtyping with arrow types, and added higher-order and overloaded functions (while in XDuce, functions are not first class values), yielding a more powerful type system. ℂDuce also generalized the notion of regular expression pattern matching by exposing all Boolean connectives to the programmer (while XDuce only provides union types).

1.3 Parametric polymorphism

All these XML processing languages achieved their goal to provide a way to statically and precisely type XML transformations, but most of them lack an important typing facility, namely parametric polymorphism[2] [Mil78]. Parametric polymorphism is a way to make a language more expressive, while still maintaining full static type-safety. It has been exploited in many programming languages, such as ML [MTHM97], Haskell [Sim92], OCaml [Oca], Java [GJSB05], C♯ [HWG03], etc.

Not surprisingly, this useful facility is needed by XML processing. Parametric polymorphism has repeatedly been requested to and discussed in various working groups of standards (*e.g.*, RELAX NG [CM01] and XQuery [DFF+07]), since it would bring not only the well-known advantages already demonstrated in existing languages (*e.g.*, the typing of `map`, `fold`, and other functions that are standard in functional languages or the use of generics in object-oriented languages), but also new usages peculiar to XML. A typical example is the Simple Object Access Protocol (SOAP) [SOAP] that provides XML "envelopes" to wrap generic XML data d as follows:

```
<Envelope>
    <Body>
     d
    </Body>
</Envelope>
```

The format of SOAP envelopes consists of an optional header (which contains application-specific information like authentication, authorization, and payment processing) and of a body (which contains the information that is the core of the SOAP message). This format is specified by the schema in [Env]. Using the notation of regular expression types (as defined in [BCF03a]) the core of this specification can be summarized as follows:

```
type Envelope = <Envelope>[ Header? Body ]
type Header = <Header>[ Any* ]
type Body = <Body>[ Any* ]
```

[2]Indeed, some of them do provide polymorphism but in a limited form, for example, XDuce supports polymorphism but not for higher-order functions [HFC09].

where `Any` is the type of any XML document and `<tag>[` R `]` classifies elements of the form `<tag>...</tag>` whose content is a sequence of elements described by the *regular expression* R on types. The definitions above, thus, state that documents of type `Envelope` are XML elements with tag `Envelope` and containing a sequence formed by an optional (as stated by the regular expression operator '`?`') element of type `Header` followed by an element of type `Body`. Elements of the former type are tagged by `Header` and contain a possibly empty sequence of elements of any type (as specified by the Kleene star '`*`'), and similarly for elements of the latter.

Envelopes are manipulated by API's which provide functions — such as `getBody`, `setBody`, `getHeader`, `setBodyElement`, ...— which are inherently polymorphic. So, for instance , the "natural" types of the first two functions are:

`setBody` : $\forall\,\alpha$. $\alpha \to$ `<Envelope>[ <Body>` α `]`
`getBody` : $\forall\,\alpha$. `<Envelope>[ Header? <Body>` α `]` $\to \alpha$

These types specify that `setBody` is a function that takes an argument of type α and returns an envelope whose body encapsulates a value of type α, while `getBody` extracts the content of type α from the body of an envelope, whatever this type is. Unfortunately, since polymorphism for XML types is not available, API's must either use the `Any` type instead of type variables, or resort to macro expansion tricks.

A subtler example involving polymorphic higher-order functions, bounded polymorphism, and XML types is given by Ocsigen [BVY09], a framework to develop dynamic web sites, where web-site paths (URIs) are associated to functions that take the URI parameters —the so-called "query strings" [BLFM05]— and return an XHTML page. The core of the dynamic part of Ocsigen system is the function `register_new_service` whose (moral) type is:

$$\forall\alpha \leq \texttt{Params}.(\texttt{Path} \times (\alpha \to \texttt{Xhtml})) \to \texttt{unit}$$

where `Params` is the XML type of all possible query strings. That is, it is a function that registers the association of its two parameters: a path in the site hierarchy and a function that is fed with a query string that matches the description α and then returns an XHTML page. By explicitly passing to `register_new_service` the type of the parameters of the page to register, we can force the type system to statically check that the function given as argument can safely handle them and we set a upper bound (with respect to the subtyping order) for the type of these parameters. Unfortunately, this kind of polymorphism is not available, yet, and the current implementation of `register_new_service` must bypass the type system (of OCaml [Oca], OCamlDuce [Fri06], and/or ℂDuce [BCF03a]), losing all the advantages of static verification.

So why despite all this interest and motivations does no satisfactory solution exist yet? The crux of the problem is that, despite several efforts (*e.g.*, [HFC09, Vou06]), a comprehensive polymorphic type system for XML was deemed unfeasible. This is mainly because (*i*) a purely semantic subtyping approach to polymorphism was believed to be impossible, and (*ii*) even if we are able to define a subtyping relation, we then need to solve "local type inference" [PT00] in the presence of intersection types. We explain these two reasons in the following.

Semantic subtyping

XML types (Schema) are essentially regular tree languages [MLM01]. As such they can be encoded by product types (to encode lists), union types (for regexp unions) and recursive types (for Kleene's star). To type higher order functions we need arrow types. We also need intersection and negation types since in the presence of arrows they can no longer be encoded. Therefore, studying polymorphism for XML types is equivalent to studying it for the types that are co-inductively (for recursion) produced by the following grammar:

$$t \ ::= \ b \ \mid \ t \times t \ \mid \ t \to t \ \mid \ t \vee t \ \mid \ t \wedge t \ \mid \ \neg t \ \mid \ \mathbb{0} \ \mid \ \mathbb{1} \tag{1.1}$$

where b ranges over basic types (*e.g.*, `Bool`, `Real`, `Int`, ...), and $\mathbb{0}$ and $\mathbb{1}$ respectively denote the empty (*i.e.*, that contains no value) and top (*i.e.*, that contains all values) types. In other terms, types are nothing but a propositional logic (with standard logical connectives: $\wedge, \vee, \neg$) whose atoms are $\mathbb{0}$, $\mathbb{1}$, basic, product, and arrow types.

Because an XML type can be interpreted as a set of documents which conform to the type (Schema) on one hand, and the subtyping relation can be defined simply as the inclusion of denoted sets on the other hand, some of current research on type systems for XML processing languages follows a semantic approach [AW93, Dam94] (*e.g.*, XDuce and $\mathbb{C}$Duce), that is, one starts with a model of the language and an interpretation of types as subsets of the model, then defines the subtyping relation as the inclusion of denoted sets. Accordingly, `Int` is interpreted as the set that contains the values `0`, `1`, `-1`, `2`, ...; `Bool` is interpreted as the set that contains the values `true` and `false`; and so on. In particular, then, unions, intersections, and negations (*i.e.*, type connectives) must have a set-theoretic semantics, and products and arrows (*i.e.*, type constructors) behave as set-theoretic products and function spaces. Formally, this corresponds to defining an interpretation function $[\![\_]\!]$ from types to the subsets of some domain $\mathcal{D}$ (for simplicity the reader can think of the domain as the set of all values of the language). Once such an interpretation has been defined, then the subtyping relation is naturally defined as the inclusion of denoted sets:

$$t_1 \leq t_2 \ \stackrel{\text{def}}{\iff} \ [\![t_1]\!] \subseteq [\![t_2]\!],$$

which, restricted to XML types, corresponds to the standard interpretation of subtyping as tree language containment.

However, it is not trivial to extend this approach with parametric polymorphism. To see the difficulty, assume that we allow type variables to occur in types [3] (hereafter we call types without type variables as ground types) and that we naively extend the semantic approach by considering all possible ground instances of types with type variables:

$$t \leq s \quad \stackrel{\text{def}}{\iff} \quad \forall \theta.\ [\![t\theta]\!] \subseteq [\![t\theta]\!] \tag{1.2}$$

where t, s are types with type variables and θ is a ground substitution, that is a finite map from type variables to ground types. Then let us consider the following subtyping

[3] We did not include any explicit quantification for type variables, since we focus on prenex parametric polymorphism where type quantification is meta-theoretic here.

statement borrowed from [HFC09]:

$$(t \times \alpha) \quad \leq \quad (t \times \neg t) \vee (\alpha \times t) \tag{1.3}$$

where t is a ground type and α is a type variable. According to (1.2), it is possible to see that (1.3) holds if and only if for all ground instances of α

$$t \leq \alpha \text{ or } \alpha \leq \neg t. \tag{1.4}$$

where or denotes the *disjunction* of classic propositional logic. In other terms, (1.3) holds if and only if t is an indivisible type, that is, a non-empty type whose only strict subtype is the empty type (*e.g.*, singleton types). Consequently, deciding the subtyping relation is at least as difficult as deciding the indivisibility of a type, which is a very hard problem [CDV08] that makes us believe more in the undecidability of the relation than in its decidability. Moreover, (1.3) is completely unintuitive[4] since α appears in unrelated positions in two related types and, as we discuss in Section 3.1, breaks the so-called parametricity [Rey83]. Actually, a purely semantic subtyping approach to polymorphism was believed to be impossible.

Local type inference

Even if we are able to solve the problem above and we define an intuitive subtyping relation, we then need to solve "local type inference" [PT00], namely, the problem of checking whether the types of a function and of its argument can be made compatible and, if so, of inferring the type of their result. In this framework functions are typed by intersections of arrow types typically to type overloading (see the grammar in (1.5)). Thus the difficulty mainly resides in the presence of the intersection types: an expression can be given different types either by subsumption (the expression is coerced into a super-type of its type) or by instantiation (the expression is used as a particular instance of its polymorphic type) and it is typed by the intersection of all these types. Therefore, in this setting, the problem is not just to find a substitution that unifies the domain type of the function with the type of its argument but, rather, a *set* of substitutions that produce instances whose intersections are in the right subtyping relation. Besides, the reminder that unrestricted intersection type systems are undecidable [CD78, Bak92] should further strengthen the idea of how difficult this is going to be.

1.4 Our work

In this thesis, we are going to add polymorphism to XML processing languages, namely, to define and study a language with higher-order polymorphic functions and recursive types with union, intersection, and negation connectives. The ultimate goal is to define a language in which the functions `setBody` and `getBody` described in the previous section, would have definitions looking similar to the following ones:

[4] For what concerns subtyping, a type variable can be considered as a special new user-defined basic type that is unrelated to any other atom but $\mathbb{0}$, $\mathbb{1}$ and itself.

```
setBody :: α -> <Envelope> [ <Body> α ]
setBody x = <Envelope> [ <Body> x ]

getBody :: <Envelope> [ Header? <Body> α ] ->  α
getBody (<Envelope> [ Header? <Body> x ]) = x
```

The functions above are mildly interesting. To design a general language and type system that allows definitions such as the above, we will use and study a different motivating example, namely the application of the functions `map` and `even` defined as follows:

```
map :: (α -> β) -> [α] -> [β]
map f l = case l of
           | [] -> []
           | (x : xs) -> (f x : map f xs)

even :: (Int -> Bool) ∧ ((γ\Int) -> (γ\Int))
even x = case x of
          | Int -> (x `mod` 2) == 0
          |   _ -> x
```

The first function is the classic `map` function defined in Haskell (we just used Greek letters to denote type variables). The second would be an Haskell function were it not for two oddities: its type contains type connectives (type intersection "∧" and type difference "\"); and the pattern in the `case` expression is a type, meaning that it matches if the value returned by the matched expression has that type. So what does the `even` function do? It checks whether its argument is an integer; if so it returns whether the integer is even or not, otherwise it returns its argument as it received it.

This thesis (in particular part III) aims to define a local type inference algorithm for such a language so that the type inferred for the application `map even` is (equivalent to) the following one:

```
map even :: ([Int] -> [Bool])  ∧
            ([γ\Int] -> [γ\Int]) ∧
            ([γ∨Int] -> [(γ\Int)∨Bool])
```

In words this type states that `map even` returns a function that when applied to a list of integers it returns a list of Booleans, when applied to a list that does not contain any integer then it returns a list of the same type (actually, it returns the same list), and when it is applied to a list that may contain some integers (*e.g.*, a list of reals), then it returns a list of the same type, without the integers but with some Booleans instead (in the case of reals, a list with Booleans and reals that are not integers).

The specific language we aim to define is a polymorphic version of ℂDuce, so first let us succinctly describe ℂDuce. The essence of ℂDuce is a λ-calculus with explicitly-typed functions, pairs and a type-case expression. Its types are defined by (1.1) and

its expressions are inductively defined by:

$$\begin{array}{rcll} e & ::= & c & \text{constant} \\ & | & x & \text{variable} \\ & | & (e,e) & \text{pair} \\ & | & \pi_i(e) & \text{projection } (i=1,2) \\ & | & e\,e & \text{application} \\ & | & \lambda^{\wedge_{i\in I} s_i\to t_i} x.\ e & \text{abstraction} \\ & | & e{\in}t\,?\,e:e & \text{type case} \end{array} \tag{1.5}$$

where c ranges over constants (*e.g.*, Booleans, integers, characters, and so on), $e{\in}t\,?\,e_1:e_2$ denotes the type-case expression that evaluates either e_1 or e_2 according to whether the value returned by e (if any) is of type t or not, and the λ-abstraction comes with an intersection of arrow types $\bigwedge_{i\in I} s_i \to t_i$ such that the whole λ-abstraction is explicitly typed by $\bigwedge_{i\in I} s_i \to t_i$ and intuitively such an abstraction is well-typed if for each $i \in I$ the body of the abstraction has type t_i when the parameter is assumed to have type s_i.

Our solution to define the polymorphic extension of such a language consists of two steps, each step being developed in one of the following parts of the thesis: (*i*) we define and study a polymorphic semantic subtyping relation for a type system with recursive, product and arrow types and set-theoretic type connectives (*i.e.*, union, intersection and negation); and (*ii*) we design and study a higher-order functional language that takes full advantage of the new capabilities of the type system. The solution we found is so general that we believe that its interest goes beyond the simple application to XML processing languages.

1.4.1 Polymorphic semantic subtyping (Part II)

We want to define a subtyping relation for types formed by type variables, products, arrows, and set-theoretic type connectives (*i.e.*, union, intersection and negation) such that it is (*i*) semantic, (*ii*) intuitive, and (*iii*) decidable. That is, the problem we want to solve is to extend the semantic subtyping defined for ground types to types with type variables.

Subtyping relation

As hinted in Section 1.3, the crux of the problem is the stuttering behaviour of indivisible types: they are either completely inside or completely outside any other type (*i.e.*, (1.4)). Our solution is to make indivisible types "splittable" so that type variables can range over strict subsets of any type. To that end, we first define a semantic assignment η for type variables as substitutions from type variables to subsets of a given domain (*e.g.* the set of all values). Then we add it to the interpretation function as a further parameter. Thus, we define the subtyping relation as the inclusion of denoted sets under all semantic assignments for the type variables:

$$t \leq s \quad \stackrel{\text{def}}{\Longleftrightarrow} \quad \forall \eta.\ [\![t]\!]\eta \subseteq [\![s]\!]\eta.$$

In this setting, every type t that denotes a set of at least two elements can be split by an assignment. That is, it is possible to define an assignment for which a type variable α denotes a set such that the interpretation of t is neither completely inside nor outside

the set. For example, assume that the interpretation of a ground type t is $\{d_1, d_2\}$. Then we can define an assignment η for a type variable α such that $\eta(\alpha) = \{d_1\}$. For such a type t, (1.4) does not hold and thus neither does (1.3). Then it is clear that the stuttering of (1.4) is absent in models where all non-empty types (indivisible types included) denote an infinite set. Therefore, infinite denotation for non-empty types is a possible and particular solution.

However, what we are looking for is not a particular solution. We are looking for a semantic characterization of models where the stuttering behaviour of indivisible types is absent. Let us consider (1.3) and (1.4) again in our setting. Applying our subtyping definition and according to set-theory, (1.3) holds if and only if

$$\forall\eta.\,((\llbracket t \wedge \alpha \rrbracket\eta = \emptyset)\ \mathsf{or}\ (\llbracket t \wedge \neg\alpha \rrbracket\eta = \emptyset)),$$

while (1.4) holds if and only if

$$(\forall\eta.\,\llbracket t \wedge \alpha \rrbracket\eta = \emptyset)\ \mathsf{or}\ (\forall\eta.\,\llbracket t \wedge \neg\alpha \rrbracket\eta = \emptyset).$$

Clearly, (1.3) holds if and only if t is an indivisible type, while (1.4) does not hold for all type t. That is to say, they are not equivalent now. Therefore, there exists a gap between (1.3) and (1.4) in our setting, which is due to the stuttering behaviour of indivisible types. To fill the gap (and thus avoid the stuttering behaviour), we require that (1.3) and (1.4) should be equivalent in our setting as well, that is we will consider only interpretation functions $\llbracket \_ \rrbracket$ such that

$$\forall\eta.\,((\llbracket t \wedge \alpha \rrbracket\eta = \emptyset)\ \mathsf{or}\ (\llbracket t \wedge \neg\alpha \rrbracket\eta = \emptyset)) \Longleftrightarrow (\forall\eta.\,\llbracket t \wedge \alpha \rrbracket\eta = \emptyset)\ \mathsf{or}\ (\forall\eta.\,\llbracket t \wedge \neg\alpha \rrbracket\eta = \emptyset).$$

Moreover, as we will use arbitrary number of Boolean connectives, we generalize the equation above to a finite set of types, which we call the *convexity* property:

$$\forall\eta.((\llbracket t_1 \rrbracket\eta{=}\varnothing)\ \mathsf{or}\ \cdots\ \mathsf{or}\ (\llbracket t_n \rrbracket\eta{=}\varnothing)) \iff (\forall\eta.\llbracket t_1 \rrbracket\eta{=}\varnothing)\ \mathsf{or}\ \cdots\ \mathsf{or}\ (\forall\eta.\llbracket t_n \rrbracket\eta{=}\varnothing) \quad (1.6)$$

Accordingly, convexity is the property we sought. Convexity states that, given a finite set of types, if every assignment makes some of these types empty, then there exists one particular type such that it is empty for all possible assignments. That is, convexity imposes a uniform behaviour to the zeros of the semantic interpretation. Indeed

$$s \leq t \Longleftrightarrow \llbracket s \rrbracket \subseteq \llbracket t \rrbracket \Longleftrightarrow \llbracket s \rrbracket \cap \overline{\llbracket t \rrbracket} \subseteq \varnothing \Longleftrightarrow \llbracket s \wedge \neg t \rrbracket = \varnothing$$

where the $\overline{S}$ denotes the complement of the set S. Consequently, checking whether $s \leq t$ is equivalent to checking whether the type $s \wedge \neg t$ is empty. Therefore, convexity imposes the subtyping relation to have a uniform behaviour and thus we believe that convexity is a semantic characterization of the "uniformity" that characterize *parametricity*.

Subtyping algorithm

Next we propose a subtyping algorithm for convex well-founded models. To illustrate our subtyping algorithm, let us consider the following example

$$(\alpha \times \mathtt{Int}) \wedge \alpha \leq ((\mathbb{1} \times \mathbb{1}) \times \mathtt{Int}). \quad (1.7)$$

According to set-theory, the subtyping checking problem can be transformed into an emptiness decision problem. So we are going to check

$$(\alpha \times \mathtt{Int}) \wedge \alpha \wedge \neg((\mathbb{1} \times \mathbb{1}) \times \mathtt{Int}) \leq \mathbb{0}.$$

Considering the interpretation of α, we can split it into two parts: one contains all the pair values and the other is the rest of the interpretation. So we substitute α with $(\alpha_1 \times \alpha_2) \vee \alpha_3$, yielding

$$((\alpha_3 \vee (\alpha_1 \times \alpha_2)) \times \mathtt{Int}) \wedge ((\alpha_1 \times \alpha_2) \vee \alpha_3) \wedge \neg((\mathbb{1} \times \mathbb{1}) \times \mathtt{Int}) \leq \mathbb{0},$$

where α_i's are fresh type variables. Moreover, since the second occurrence of α is intersected with some products, then whatever the interpretation of α is, the only part of its denotation that matters is the one that contains all the pair values, that is, $(\alpha_1 \times \alpha_2)$. In other words, the substitution of $(\alpha_1 \times \alpha_2)$ will suffice for the second occurrence of α. While for the first occurrence of α, we can not simplify the substitution, as it is not intersected with products and the interpretation of α may contain some values that are not pairs. Consequently, we can simply check

$$((\alpha_3 \vee (\alpha_1 \times \alpha_2)) \times \mathtt{Int}) \wedge (\alpha_1 \times \alpha_2) \wedge \neg((\mathbb{1} \times \mathbb{1}) \times \mathtt{Int}) \leq \mathbb{0}$$

which is equivalent to

$$((\alpha_3 \vee (\alpha_1 \times \alpha_2)) \times \mathtt{Int}) \wedge (\alpha_1 \times \alpha_2) \leq ((\mathbb{1} \times \mathbb{1}) \times \mathtt{Int}) \tag{1.8}$$

According to the set-theoretic properties of the interpretation function and our definition of subtyping, we then check

$$\forall \eta. [\![((\alpha_3 \vee (\alpha_1 \times \alpha_2)) \times \mathtt{Int})]\!]\eta \cap [\![(\alpha_1 \times \alpha_2)]\!]\eta \cap \overline{[\![((\mathbb{1} \times \mathbb{1}) \times \mathtt{Int})]\!]\eta} = \emptyset.$$

Based on the set-theory, we then reduce it to [5]

$$\begin{array}{rl} & \forall \eta. ([\![(\alpha_3 \vee (\alpha_1 \times \alpha_2)) \wedge \alpha_1]\!]\eta = \emptyset \text{ or } [\![\mathtt{Int} \wedge \alpha_2 \wedge \neg\mathtt{Int}]\!]\eta = \emptyset) \\ \text{and} & \forall \eta. ([\![(\alpha_3 \vee (\alpha_1 \times \alpha_2)) \wedge \alpha_1 \wedge \neg(\mathbb{1} \times \mathbb{1})]\!]\eta = \emptyset \text{ or } [\![\mathtt{Int} \wedge \alpha_2]\!]\eta = \emptyset) \end{array}$$

where and denotes the *conjunction* of classic propositional logic. We now apply the convexity property and distribute the quantification on η on each subformula of the or, yielding

$$\begin{array}{rl} & (\forall \eta. [\![(\alpha_3 \vee (\alpha_1 \times \alpha_2)) \wedge \alpha_1]\!]\eta = \emptyset) \text{ or } (\forall \eta. [\![\mathtt{Int} \wedge \alpha_2 \wedge \neg\mathtt{Int}]\!]\eta = \emptyset) \\ \text{and} & (\forall \eta. [\![(\alpha_3 \vee (\alpha_1 \times \alpha_2)) \wedge \alpha_1 \wedge \neg(\mathbb{1} \times \mathbb{1})]\!]\eta = \emptyset) \text{ or } (\forall \eta. [\![\mathtt{Int} \wedge \alpha_2]\!]\eta = \emptyset) \end{array}$$

which is equivalent to

$$\begin{array}{rl} & ((\alpha_3 \vee (\alpha_1 \times \alpha_2)) \wedge \alpha_1 \leq \mathbb{0}) \text{ or } (\mathtt{Int} \wedge \alpha_2 \wedge \neg\mathtt{Int} \leq \mathbb{0}) \\ \text{and} & ((\alpha_3 \vee (\alpha_1 \times \alpha_2)) \wedge \alpha_1 \wedge \neg(\mathbb{1} \times \mathbb{1}) \leq \mathbb{0}) \text{ or } (\mathtt{Int} \wedge \alpha_2 \leq \mathbb{0}) \end{array}$$

[5] Interested readers can refer to the decomposition rule in Lemma 4.3.9.

or equivalently[6]

$$\begin{array}{ll} & (\alpha_3 \vee (\alpha_1 \times \alpha_2)) \wedge \alpha_1 \leq \mathbb{0} \\ \mathsf{or} & \mathtt{Int} \wedge \alpha_2 \leq \mathbb{0} \\ \mathsf{or} & ((\alpha_3 \vee (\alpha_1 \times \alpha_2)) \wedge \alpha_1 \leq (\mathbb{1} \times \mathbb{1})) \ \mathsf{and} \ (\mathtt{Int} \wedge \alpha_2 \leq \mathtt{Int}) \end{array}$$

These are the expected conditions for the subtyping relation in (1.8).

Hence, we are able to reduce the problem into sub-problems on (the instances of) sub-terms. Then it is easy to find a substitution (*i.e.*, a finite map from type variables to types), for example, $\{\mathtt{Int}/\alpha_3, \mathtt{Int}/\alpha_1, \mathtt{Int}/\alpha_2\}$ such that all the subformulas of the or fail. Therefore, the subtyping relation in (1.7) does not hold.

Besides, if we substituted $(\alpha_1 \times \alpha_2)$ for the first occurrence of α as well, then the sub-problems we would check are

$$\begin{array}{ll} & (\alpha_1 \times \alpha_2) \wedge \alpha_1 \leq \mathbb{0} \\ \mathsf{or} & \mathtt{Int} \wedge \alpha_2 \leq \mathbb{0} \\ \mathsf{or} & ((\alpha_1 \times \alpha_2) \wedge \alpha_1 \leq (\mathbb{1} \times \mathbb{1})) \ \mathsf{and} \ (\mathtt{Int} \wedge \alpha_2 \leq \mathtt{Int}) \end{array}$$

Clearly, the last subformula holds and thus so does the subtyping relation in (1.7). This justifies that the tricky substitution $\{\alpha_3 \vee (\alpha_1 \times \alpha_2)/\alpha\}$ is necessary.

Similarly for the intersection of arrow types. Thanks to the regularity of our types, the subtyping algorithm we explain in details in Section 3.4 terminates on all types.

Consider `map even`: its expected type contains an arrow $[(\gamma \vee \mathtt{Int})] \to [((\gamma \setminus \mathtt{Int}) \vee \mathtt{Bool})]$. From that we deduce `even` should be typed by $(\gamma \vee \mathtt{Int}) \to ((\gamma \setminus \mathtt{Int}) \vee \mathtt{Bool})$, which we can not obtain by instantiation. So the only possible way to obtain it is by subsumption. Namely, we need to check the following subtyping relation:

$$(\mathtt{Int} \to \mathtt{Bool}) \wedge ((\gamma \setminus \mathtt{Int}) \to (\gamma \setminus \mathtt{Int})) \leq (\gamma \vee \mathtt{Int}) \to ((\gamma \setminus \mathtt{Int}) \vee \mathtt{Bool}) \qquad (1.9)$$

According to the subtyping algorithm, (1.9) holds. Hence, `even` has type $(\gamma \vee \mathtt{Int}) \to ((\gamma \setminus \mathtt{Int}) \vee \mathtt{Bool})$ as expected.

Convex model

Finally, we prove that there exists at least one set-theoretic model that is convex, that is, that satisfies (1.6). Actually, there exist a lot of convex models since every model for ground types with infinite denotations is convex. To grasp the intuition, let us consider the simple case of propositional logic:

$$t \ ::= \ \alpha \ \mid \ t \vee t \ \mid \ t \wedge t \ \mid \ \neg t \ \mid \ \mathbb{0} \ \mid \ \mathbb{1}$$

where α's are then propositional variables (the construction works also in the presence of basic types). It is relatively easy to prove by contrapositive that if all propositions (*i.e.*, types) are interpreted into infinite sets, then for all $n \geq 2$

$$\forall \eta \,.\, ((\llbracket t_1 \rrbracket \eta = \emptyset) \ \mathsf{or} \ \ldots \ \mathsf{or} \ (\llbracket t_n \rrbracket \eta = \emptyset)) \ \Rightarrow \ (\forall \eta \,.\, \llbracket t_1 \rrbracket \eta = \emptyset) \ \mathsf{or} \ \ldots \ \mathsf{or} \ (\forall \eta \,.\, \llbracket t_n \rrbracket \eta = \emptyset) \qquad (1.10)$$

[6]Since the intersection of `Int` and $\neg$`Int` is always empty, by simplifying we can consider only two cases: $((\alpha_3 \vee (\alpha_1 \times \alpha_2)) \wedge \alpha_1 \leq (\mathbb{1} \times \mathbb{1}))$ or $(\mathtt{Int} \wedge \alpha_2 \leq \mathbb{0})$. But to exemplify our algorithm more detail, we keep all the cases.

holds[7]. Without loss of generality, we can assume that

$$t_i = \bigwedge_{j \in P_i} \alpha_j \wedge \bigwedge_{j \in N_i} \neg \alpha_j \quad (P_i \cap N_i = \emptyset)$$

since, if t_i is a union, then by exiting the union from the interpretation we obtain a special case of (1.10) with a larger n, for instance, $[\![t_1 \vee t_2]\!]\eta = \emptyset$ is equivalent to $[\![t_1]\!]\eta = \emptyset$ or $[\![t_2]\!]\eta = \emptyset$. Then we proceed by contrapositive: suppose that there exists η_i such that $[\![t_i]\!]\eta_i \neq \emptyset$ for each i, then we can build $\bar{\eta}$ such that for all i, $[\![t_i]\!]\bar{\eta} \neq \emptyset$. The construction of $\bar{\eta}$ is done by iterating on the t_i's, picking at each iteration a particular element d of the domain, and recording this choice in a set s_0 so as to ensure that at each iteration a different d is chosen:

Start with $s_0 = \emptyset$ and for $i = 1..n$:

- choose $d \in ([\![\mathbb{1}]\!]\eta_i \setminus s_0)$ (which is possible since $[\![\mathbb{1}]\!]\eta_i$ *is infinite* and s_0 is finite)
- add d to s_0 (to record that we used it) and to $\bar{\eta}(\alpha_j)$ for all $j \in P_i$.

Notice that infinite denotation ensures that a fresh element always exists at each step of the iteration.

Therefore, to construct a convex model, it just suffices to take any model defined in [FCB08] and straightforwardly modify the interpretation of basic types such that they have infinite denotations.

1.4.2 Polymorphic calculus (Part III)

In part II we show how to extend semantic subtyping to polymorphic types. In part III we then show how to define a polymorphic calculus that takes full advantage of the new capabilities of the type system, which can then be easily extended to a full-fledged polymorphic functional language for processing XML documents. Namely, the calculus we want to define is a polymorphic version of CDuce.

An explicitly typed λ-calculus

The novelty of our work with respect to the current work on semantic subtyping and CDuce [FCB08] is to allow type variables to occur in the types and, thus, in the types labeling λ-abstractions. It becomes thus possible to define the polymorphic identity function as $\lambda^{\alpha \to \alpha} x.x$, while classic "auto-application" term is written as $\lambda^{((\alpha \to \beta) \wedge \alpha) \to \beta} x.xx$. The intended meaning of using a type variable, such as α, is that a (well-typed) λ-abstraction not only has the type specified in its label (and by subsumption all its super-types) but also all the types obtained by instantiating the type variables occurring in the label. So $\lambda^{\alpha \to \alpha} x.x$ has by subsumption both the type $\mathbb{0} \to \mathbb{1}$ (the type of all functions, which is a super-type of $\alpha \to \alpha$) and the type $\neg$`Int`, and by instantiation the types `Int`$\to$`Int`, `Bool`$\to$`Bool`, etc.

The use of instantiation in combination with intersection types has nasty consequences, for if a term has two distinct types, then it has also their intersection type. In the monomorphic case a term can have distinct types only by subsumption and, thus, intersection types are assigned transparently to terms by the type system via subsumption. But in the polymorphic case this is no longer possible: a term can be

[7] The other direction is straightforward, so we do not consider it here.

typed by the intersection of two distinct instances of its polymorphic type which, in general, are not in any subtyping relation with the latter: for instance, $\alpha{\to}\alpha$ is neither a subtype of $\texttt{Int}{\to}\texttt{Int}$ nor vice-versa, since the subtyping relation must hold for *all possible* instantiations of α (and there are infinitely many instances of $\alpha{\to}\alpha$ that are neither subtype nor super-type of $\texttt{Int}{\to}\texttt{Int}$).

Concretely, if we want to apply the polymorphic identity $\lambda^{\alpha\to\alpha}x.x$ to, say, 42, then the particular instance obtained by the type substitution $\{\texttt{Int}/\alpha\}$ (denoting the replacement of every occurrence of α by Int) must be used, that is $(\lambda^{\texttt{Int}\to\texttt{Int}}x.x)\texttt{42}$. We have thus to *relabel* the type decorations of λ-abstractions before applying them. In implicitly typed languages, such as ML, the relabeling is meaningless (no type decoration is used) while in their explicitly typed counterparts relabeling can be seen as a logically meaningful but computationally useless operation, insofar as execution takes place on type erasures. In the presence of type-case expressions, however, relabeling is necessary since the label of a λ-abstraction determines its type: testing whether an expression has type, say, $\texttt{Int}{\to}\texttt{Int}$ should succeed for the application of $\lambda^{\alpha\to\alpha\to\alpha}x.\lambda^{\alpha\to\alpha}y.x$ to 42 and fail for its application to true. This means that, in Reynolds' terminology, our terms have an *intrinsic* meaning [Rey03].

Notice that we may need to relabel/instantiate functions not only when they are applied but also when they are used as arguments. For instance consider a function that expects arguments of type $\texttt{Int}{\to}\texttt{Int}$. It is clear that we can apply it to the identity function $\lambda^{\alpha\to\alpha}x.x$, since the identity function *has* type $\texttt{Int}{\to}\texttt{Int}$ (feed it by an integer and it will return an integer). Before, though, we have to relabel the latter by the substitution $\{\texttt{Int}/\alpha\}$ yielding $\lambda^{\texttt{Int}\to\texttt{Int}}x.x$. As the identity has type $\texttt{Int}{\to}\texttt{Int}$ so it has type $\texttt{Bool}{\to}\texttt{Bool}$ and, therefore, the intersection of the two: $(\texttt{Int}{\to}\texttt{Int})\wedge(\texttt{Bool}{\to}\texttt{Bool})$. So we can apply a function that expects an argument of this intersection type to our identity function. The problem is now how to relabel $\lambda^{\alpha\to\alpha}x.x$ via two distinct type-substitutions $\{\texttt{Int}/\alpha\}$ and $\{\texttt{Bool}/\alpha\}$. Intuitively, we have to apply $\{\texttt{Int}/\alpha\}$ and $\{\texttt{Bool}/\alpha\}$ to the label of the λ-abstraction and replace it by the intersection of the two instances. This corresponds to relabel the polymorphic identity from $\lambda^{\alpha\to\alpha}x.x$ into $\lambda^{(\texttt{Int}\to\texttt{Int})\wedge(\texttt{Bool}\to\texttt{Bool})}x.x$. This is the solution adopted by this work, where we manipulate *sets* of type-substitutions — delimited by square brackets —. The application of such a set (*e.g.*, in the previous example $[\{\texttt{Int}/\alpha\},\{\texttt{Bool}/\alpha\}]$) to a type t, returns the intersection of all types obtained by applying each substitution in the set to t (*e.g.*, in the example $t\{\texttt{Int}/\alpha\}\wedge t\{\texttt{Bool}/\alpha\}$).

The polymorphic identity function is too simple since we do not need to relabel its body. Let us consider the relabeling of the "daffy" polymorphic identity function

$$\lambda^{\alpha\to\alpha}x.\ (\lambda^{\alpha\to\alpha}y.x)x$$

via the set of substitutions $[\{\texttt{Int}/\alpha\},\{\texttt{Bool}/\alpha\}]$. Assume we naively relabel the function deeply, that is, apply the set of type substitutions not only to the label of the function but also to the labels in the body of the function. So we have the following expression:

$$\lambda^{(\texttt{Int}\to\texttt{Int})\wedge(\texttt{Bool}\to\texttt{Bool})}x.\ (\lambda^{(\texttt{Int}\to\texttt{Int})\wedge(\texttt{Bool}\to\texttt{Bool})}y.x)x$$

which, however, is not well-typed. Let us try to type-check this expression. This corresponds to prove that the expression $(\lambda^{(\texttt{Int}\to\texttt{Int})\wedge(\texttt{Bool}\to\texttt{Bool})}y.x)x$ has type Int under

the hypothesis that x has type $\texttt{Int}$, and it has type $\texttt{Bool}$ under the hypothesis that x has type $\texttt{Bool}$. Both checks fail, since neither the expression $\lambda^{(\texttt{Int}\to\texttt{Int})\wedge(\texttt{Bool}\to\texttt{Bool})}y.x$ has type $\texttt{Int}\to\texttt{Int}$ when x has type $\texttt{Bool}$, nor it has type $\texttt{Bool}\to\texttt{Bool}$ when x has type $\texttt{Int}$. This example shows that in order to ensure that relabeling yields well-typed expressions, the relabeling of the body must change according to the type the parameter x is bound to. More precisely, $\lambda^{\alpha\to\alpha}y.x$ should be relabeled as $\lambda^{\texttt{Int}\to\texttt{Int}}y.x$ when x is of type $\texttt{Int}$, and as $\lambda^{\texttt{Bool}\to\texttt{Bool}}y.x$ when x is of type $\texttt{Bool}$.

Our solution to relabel a function via a set of type substitutions is to introduce a new technique that consists in performing a "lazy" relabeling of the bodies, which is obtained by *decorating* λ-abstractions by a finite set of explicit type-substitutions:

$$\lambda^{\wedge_{i\in I}t_i\to s_i}_{[\sigma_j]_{j\in J}}x.e$$

where $[\sigma_j]_{j\in J}$ is a set of type substitutions to relabel. This yields our calculus with explicit type-substitutions. So the "daffy" identity function can be decorated as

$$\lambda^{\alpha\to\alpha}_{[\{\texttt{Int}/\alpha\},\{\texttt{Bool}/\alpha\}]}x.\,(\lambda^{\alpha\to\alpha}y.x)x.$$

When type-checking λ-abstractions, we propagate each type-substitution σ_j to the bodies of abstractions separately and thus solve the type-checking problem. Consider the "daffy" identity function again. When x is assumed to has type $\texttt{Int}$ we propagate $\{\texttt{Int}/\alpha\}$ to the body, and similarly $\{\texttt{Bool}/\alpha\}$ is propagated to the body when x is assumed to have type $\texttt{Bool}$.

During reduction we also can not simply relabel an expression by a set of type-substitutions by propagating the set of type-substitutions into the expression deeply. For that, we also introduce a "lazy" *relabeling* operation @ which takes an expression and a set of type-substitutions and lazily applies the set of type-substitutions to all outermost λ-abstractions occurring in the expression. In particular, for λ-abstractions we have

$$(\lambda^{\wedge_{i\in I}t_i\to s_i}_{[\sigma_j]_{j\in J}}x.e)@[\sigma_k]_{k\in K}=\lambda^{\wedge_{i\in I}t_i\to s_i}_{[\sigma_j]_{j\in J}\circ[\sigma_k]_{k\in K}}x.e$$

where $\circ$ denotes the pairwise composition of all substitutions of the two sets.

Using the explicitly typed λ-calculus, in particular, with the type case, we can define $\texttt{map}$[8] and $\texttt{even}$ as

$$\begin{array}{rl}\texttt{map}:= & \mu^{(\alpha\to\beta)\to[\alpha]\to[\beta]}m\;\lambda f\,.\,\lambda^{[\alpha]\to[\beta]}\ell\,.\,\ell{\in}\texttt{nil}\,?\,\texttt{nil}:(f(\pi_1\ell),mf(\pi_2\ell))\\ \texttt{even}:= & \lambda^{(\texttt{Int}\to\texttt{Bool})\wedge((\gamma\setminus\texttt{Int})\to(\gamma\setminus\texttt{Int}))}x.x{\in}\texttt{Int}\,?\,(x\;\texttt{mod}\;2)=0:x\end{array}$$

where $\texttt{nil}$ denotes (the type of) the empty list. To make the application well-typed and to obtain the type we expect, we first instantiate the type of $\texttt{map}$ with $[\sigma_1,\sigma_2,\sigma_3]$, where $\sigma_1=\{\texttt{Int}/\alpha,\texttt{Bool}/\beta\}$, $\sigma_2=\{(\gamma\setminus\texttt{Int})/\alpha,(\gamma\setminus\texttt{Int})/\beta\}$, and $\sigma_3=\{(\gamma\vee\texttt{Int})/\alpha,((\gamma\setminus\texttt{Int})\vee\texttt{Bool})/\beta\}$. Namely, we relabel $\texttt{map}$ with $[\sigma_1,\sigma_2,\sigma_3]$, yielding $\texttt{map}_{[\sigma_1,\sigma_2,\sigma_3]}$. Then by subsumption (see the subtyping relation in (1.9)), we deduce $\texttt{even}$ also has the following type:

$$(\texttt{Int}\to\texttt{Bool})\wedge((\gamma\setminus\texttt{Int})\to(\gamma\setminus\texttt{Int}))\wedge((\gamma\vee\texttt{Int})\to((\gamma\setminus\texttt{Int})\vee\texttt{Bool})),$$

[8]The type case in $\texttt{map}$ should be a "binding" one and recursive functions can be added straightforwardly, which is introduced in Chapter 12.

which matches the domain of the type of $\texttt{map}_{[\sigma_1,\sigma_2,\sigma_3]}$. Therefore $(\texttt{map}_{[\sigma_1,\sigma_2,\sigma_3]})\,\texttt{even}$ is well-typed and the result type is

$$(\texttt{[Int]} \to \texttt{[Bool]}) \wedge (\texttt{[}\gamma \setminus \texttt{Int]} \to \texttt{[}\gamma \setminus \texttt{Int]}) \wedge (\texttt{[}\gamma \vee \texttt{Int]} \to \texttt{[}(\gamma \setminus \texttt{Int}) \vee \texttt{Bool]})$$

as expected.

Type substitutions inference

In practice, we want to program with expressions without explicitly giving type substitutions, that is, in the language defined by the grammar (1.5), where types in the λ-abstractions may contain type variables. For example, we would like to write the application of `map` to `even` as `map even` rather than $(\texttt{map}_{[\sigma_1,\sigma_2,\sigma_3]})\,\texttt{even}$. Hence we propose an inference system that infers where and whether type substitutions can be inserted in an expression to make it well-typed. The difficulty lies in how to type applications, as we need to check whether the types of a function and of its argument can be made compatible.

As shown above, we can instantiate a polymorphic type not only with a type substitution but also with a set of type substitutions. That is, a λ-abstraction can be relabeled not only by a type substitution but also by a set of type substitutions, such as $\texttt{map}_{[\sigma_1,\sigma_2,\sigma_3]}$. Accordingly, to make `map even` well-typed, that is, to tally the domain of the type of `map` against the type of `even`, we need to find two sets of type substitutions $[\sigma_i]_{i\in I}$ and $[\sigma'_j]_{j\in J}$ such that

$$\bigwedge_{i\in I}((\texttt{Int} \to \texttt{Bool}) \wedge ((\gamma \setminus \texttt{Int}) \to (\gamma \setminus \texttt{Int})))\sigma_i \leq \bigvee_{j\in J}(\alpha \to \beta)\sigma'_j \tag{1.11}$$

Note that the domain of an intersection of arrows is the *union* of the domains of each arrow in the intersection. Assume that the cardinalities of I and J are fixed, for example, $|I| = 1$ and $|J| = 3$. By general renaming, we can equivalently rewrite (1.11) into

$$((\texttt{Int} \to \texttt{Bool}) \wedge ((\gamma \setminus \texttt{Int}) \to (\gamma \setminus \texttt{Int})))\sigma \leq ((\alpha_1 \to \beta_1) \vee (\alpha_2 \to \beta_2) \vee (\alpha_3 \to \beta_3))\sigma \tag{1.12}$$

where α_i and β_i are fresh variables. This is an instance of the *type tallying problem* $\exists\sigma.\ s\sigma \leq t\sigma$, which is explained in the next section. Using the type tallying algorithm, we can solve (1.12) and a solution is a substitution σ_0 such as[9]:

$$\begin{array}{lll} \{ & \texttt{Int}/\alpha_1, \texttt{Bool}/\beta_1 & \text{(corresponding to } \sigma_1\text{)} \\ & (\gamma \setminus \texttt{Int})/\alpha_2, (\gamma \setminus \texttt{Int})/\beta_2 & \text{(corresponding to } \sigma_2\text{)} \\ & (\gamma \vee \texttt{Int})/\alpha_3, ((\gamma \setminus \texttt{Int}) \vee \texttt{Bool})/\beta_3 \quad \} & \text{(corresponding to } \sigma_3\text{)} \end{array}$$

In general, we can reduce the type substitution inference to the problem of deciding whether for two given types s and t there exist two sets of type substitutions $[\sigma_i]_{i\in I}$ and $[\sigma'_j]_{j\in J}$ such that

$$\bigwedge_{i\in I} s\sigma_i \leq \bigvee_{j\in J} t\sigma'_j.$$

[9]The other solutions are either equivalent to σ_0 or useless.

When the cardinalities of I and J are fixed, we can further reduce it to a type tallying problem. We prove that the type tallying problem has a sound, complete and terminating algorithm. This immediately yields a semi-decision procedure (that tries all the cardinalities) for the inference system.

In order to ensure termination we propose two heuristic numbers p and q for the cardinalities of I and J that are established according to the form of s and t. The intuition is that type connectives are what make us to instantiate a polymorphic type several times. Take `map even` for example. The function `even` can be invoked in `map` onto the lists (*i*) that contain only integers, (*ii*) that contain no integers, and (*iii*) that contain some integers and some non-integers. That is, `even` can be typed by $\texttt{Int} \to \texttt{Bool}$, by $(\gamma \setminus \texttt{Int}) \to (\gamma \setminus \texttt{Int})$, by $(\gamma \vee \texttt{Int}) \to ((\gamma \setminus \texttt{Int}) \vee \texttt{Bool})$ (which can be obtained from the first two by unions) and by the intersection of the three. So the heuristic number q for `map` is 3. While the domain of the type for `map` is a single arrow $\alpha \to \beta$, so the heuristic number p for `even` is 1.

Type tallying algorithm

Given two types t and s, the type tallying problem is to check whether there exists a substitution σ such that $t\sigma \leq s\sigma$. The type tallying problem can be consider as a constraint solving problem, where constraints are of the form $t \leq s$ or $t \geq s$. Constraints of the form $\alpha \geq s$ or $\alpha \leq s$ are called normalized constraints. To illustrate the type tallying algorithm, let us consider the following constraint:

$$(\beta \to \texttt{Int}) \to (\beta \to \texttt{Int}) \leq (\alpha \to \alpha) \to (\alpha \to \alpha).$$

First, according to the subtyping algorithm presented in Section 1.4.1, we can decompose this constraint into several ones on sub-terms:

$$(\alpha \to \alpha) \leq (\beta \to \texttt{Int}) \text{ and } (\beta \to \texttt{Int}) \leq (\alpha \to \alpha)$$

and thus, finally, a set of normalized constraints[10]

$$\{\alpha \geq \beta, \alpha \leq \texttt{Int}, \alpha \leq \beta, \alpha \geq \texttt{Int}\}.$$

Next, we can merge the normalized constraints with the same type variable. For $\alpha \geq \beta$ and $\alpha \geq \texttt{Int}$ (*i.e.* lower bounds for α), we can merge them by unions, that is, replace them by $\alpha \geq (\texttt{Int} \vee \beta)$. Similarly, we can replace $\alpha \leq \texttt{Int}$ and $\alpha \leq \beta$ (*i.e.* upper bounds for α) by $\alpha \leq (\texttt{Int} \wedge \beta)$. Then we get

$$\{\alpha \geq (\beta \vee \texttt{Int}), \alpha \leq (\beta \wedge \texttt{Int})\}.$$

If the type substitution σ we are looking for satisfies $\alpha\sigma \geq (\beta \vee \texttt{Int})\sigma$ and $\alpha\sigma \leq (\beta \wedge \texttt{Int})\sigma$, then it also satisfies $(\beta \vee \texttt{Int})\sigma \leq (\beta \wedge \texttt{Int})\sigma$. So we saturate the constraint set with $(\beta \vee \texttt{Int}) \leq (\beta \wedge \texttt{Int})$, which we also decompose into normalized constraints, yielding

$$\{\alpha \geq (\beta \vee \texttt{Int}), \alpha \leq (\beta \wedge \texttt{Int}), \beta \geq \texttt{Int}, \beta \leq \texttt{Int}\}.$$

[10]For simplicity, we omit the case that $\mathbb{0} \to \mathbb{1}$ is a supertype of any arrow. Besides, from this constraint set, we can immediate get the substitution $\{\texttt{Int}/\beta, \texttt{Int}/\alpha\}$. But to exemplify our algorithm, we keep on elaborating constraints.

After that, there are only one lower bound and only one upper bound for each type variable. Note that if there are no lower bounds (upper bounds resp.) for α, we can add the trivial constraint $\alpha \geq \mathbb{0}$ ($\alpha \leq \mathbb{1}$ resp.). We then can transform a set of normalized constraints into a set of equations: each pair of $\alpha \geq t$ and $\alpha \leq s$ is transformed into an equation $\alpha = (t \vee \alpha') \wedge s$, where α' is a fresh type variable. Thus the constraint set above is transformed into

$$\begin{aligned} \alpha = &\quad ((\beta \vee \mathtt{Int}) \vee \alpha') \wedge (\beta \wedge \mathtt{Int}) \\ \beta = &\quad (\mathtt{Int} \vee \beta') \wedge \mathtt{Int} \end{aligned}$$

where α' and β' are fresh type variables. Finally, using Courcelle's work on infinite trees [Cou83], we solve this equation system, which gives us the following substitution:

$$\{\mathtt{Int}/\beta, \mathtt{Int}/\alpha\}$$

which is a solution of the original type tallying problem.

Translation to $\mathbb{C}$Duce

Finally, to make our approach feasible in practice, we compile our polymorphic calculus into a variant of $\mathbb{C}$Duce (where types contain type variables and the subtyping relation is the polymorphic one) to provide an execution model for our calculus. The translation relies on an extension of the type case expression that features binding, which we write $x \in t \,?\, e_1 : e_2$ and can be encoded as:

$$(\lambda^{((s\wedge t)\to t_1)\wedge((s\wedge\neg t)\to t_2)} x.x \in t \,?\, e_1 : e_2)x$$

where s is the type for the argument x and t_i is the type of e_i. Note that the difference of the binding type case from the unbinding one is that the types for the occurrences of x in e_1 and e_2 are different. The key idea of the translation is to use the "binding" type case expression to simulate different relabeling on the body expression of λ-abstractions with different type cases. For example, the expression $\mathtt{map}[\sigma_1, \sigma_2]$ is translated into

$$\begin{aligned} f \in (\alpha \to \beta)\sigma_1 \wedge (\alpha \to \beta)\sigma_2 \,?\quad & \mathbb{C}[\![mb@[\sigma_1, \sigma_2]]\!] \\ f \in (\alpha \to \beta)\sigma_1 \,?\quad & \mathbb{C}[\![mb@[\sigma_1]]\!] \\ f \in (\alpha \to \beta)\sigma_2 \,?\quad & \mathbb{C}[\![mb@[\sigma_2]]\!] \\ f \in \mathbb{1} \,?\quad & \mathbb{C}[\![mb]\!] \end{aligned}$$

where mb is the body of `map` and $\mathbb{C}[\![e]\!]$ denotes the translation of e. The first branch simulates the case where both type substitutions σ_1 and σ_2 are selected and propagated to the body, that is, the parameter f belongs to the intersection of different instances of $\alpha \to \beta$ with these two type substitutions. The second and third branches simulate the case where exactly one substitution is used. Finally, the last branch denotes the case where no type substitutions are selected.

1.5 Contributions

The overall contribution of this thesis is the definition of a polymorphic semantic subtype relation for a type system with recursive types and union, intersection, and negation type connectives and the definition of a statically typed calculus with polymorphic higher-order functions and semantic subtyping. More in details:

1.5.1 Part II

The contributions of Part II are the following:

1. we define a polymorphic semantic subtyping relation for a type system with recursive types and union, intersection, and negation type connectives. We first define a semantic assignment for type variables, that is a substitution from type variables to subsets of any type, and then the subtyping relation is defined as the inclusion of denoted sets under all semantic assignments for the type variables. We also introduce a convexity property, which imposes the subtyping relation to have a uniform behaviour. Our definition is the first solution to the problem of defining a semantic subtyping relation for regular tree types with type variables.

2. we propose an algorithm for deciding the polymorphic semantic subtyping relation induced by a well-founded convex model, which is based on set-theoretic properties and the convexity property. We also prove the soundness, completeness, and termination properties of the algorithm.

3. we prove that there exists at least one set-theoretic model that is convex. Actually, there exist a lot of convex models since every model for ground types with infinite denotations is convex.

1.5.2 Part III

The contributions of Part III are the following:

1. we define an explicitly-typed λ-calculus with intersection (and union and negation) types, whose key idea consists in decorating λ-abstractions with types and type-substitutions that are lazily propagated at the moment of the reduction. This contrasts with current solutions in the literature which require the addition of new operators, stores and/or pointers. In doing that we singled out that the problem of defining an explicit typed version of intersection type systems resides in the fact that the relabeling of the body of a function is dependent on the actual type of the argument, a point that, in our knowledge, was not understood before.

2. we propose an algorithm that for any pair of polymorphic regular tree types t_1 and t_2 produces a sound and complete set of solutions to the problem whether there exists a substitution σ such that $t_1\sigma \leq t_2\sigma$. This is obtained by using the set-theoretic interpretation of types to reduce the problem to a unification problem on regular tree types. We prove the termination of the algorithm.

3. we propose an algorithm for local type inference for the calculus. Practically speaking this means that the programmer has to explicitly type function definitions, but that any other type information, in particular the instantiations and expansion of type variables is inferred by the system at compile time. The algorithm yields a semi-decision procedure for the typability of a λ-calculus with intersection types and with explicitly typed lambda expressions whose decidability is still an open issue.

4. we design a compilation of the polymorphic calculus into the monomorphic one. This is a non-trivial problem since the source polymorphic calculus includes a type-case expression. From a practical viewpoint it allows us to reuse the run-time engine developed for monomorphic $\mathbb{C}$Duce also for the polymorphic version with the sole modification of plugging the polymorphic subtyping relation in the execution of type-cases.

1.6 Outline of the thesis

The thesis is organized as follows. The rest of Part I contains a chapter, which succinctly describes the core of the programming language $\mathbb{C}$Duce. Part II and Part III give the definition of a polymorphic semantic subtyping relation and the definition of a polymorphic calculus respectively. Part IV concludes the thesis and presents some future work.

The contents of Part II and Part III are organized as follows. Chapter 3 illustrates an informal description of the main ideas and intuitions underlying the definition of polymorphic semantic subtyping relation. Chapter 4 describes the technical development that supports the results exposed in Chapter 3. Chapter 5 presents a practice subtyping algorithm which does not perform any type-substitution. Chapter 6 gives an overview of the definition of polymorphic calculus. Chapter 7 defines an explicitly-typed λ-calculus with sets of type-substitutions. Chapter 8 presents an equivalent type system with syntax-directed rules for the explicitly-typed calculus. Chapter 9 defines an implicitly-typed calculus and proposes an inference system that infers where and whether type-substitutions can be inserted in an implicitly-typed expression. Chapter 10 defines a type tallying problem and its algorithm, and proposes a semi-decision procedure for the inference system. Chapter 11 introduces the translation from the polymorphic calculus into a variant of the monomorphic calculus. Chapter 12 presents some extensions and design choices.

Publication

Part II is an extended version of [CX11]. Part III is an extended version of [CNXL12].

Chapter 2

ℂDuce

Since our work is an extension of ℂDuce and in order to make our work more easy to understand, we succinctly describe the core of the programming language ℂDuce and some of its key ideas at first. The interested reader can refer to [BCF03a, Fri04, FCB08] for more detailed definitions of the various concepts presented hereafter.

2.1 Types

We define the types in ℂDuce and their subtyping relation.

Definition 2.1.1. *A* type *in ℂDuce is a regular (infinite) tree, co-inductively (for recursion) produced by the following grammar:*

$$t \ ::= \ b \ \mid \ t \times t \ \mid \ t \to t \ \mid \ t \vee t \ \mid \ t \wedge t \ \mid \ \neg t \ \mid \ \mathbb{0} \ \mid \ \mathbb{1}$$

*where b ranges over basic types (*e.g.*, `Bool`, `Real`, `Int`, ...), and* $\mathbb{0}$ *and* $\mathbb{1}$ *respectively denote the empty (*i.e.*, that contains no value) and top (*i.e.*, that contains all values) types.*

In other terms, types are nothing but a propositional logic (with standard logical connectives: $\wedge, \vee, \neg$) whose atoms are $\mathbb{0}$, $\mathbb{1}$, basic, product, and arrow types. Let $\mathcal{T}$ denote the set of all types.

From a strictly practical viewpoint recursive types, products, and type connectives are used to encode regular tree types [Hos01], which subsume existing XML schema/types. For example, the following type (presented in Figure 1.3)

```
type Person   = <person>[Name,Email*,Tel?]
```

can be considered as a syntactic sugar for the following equations:

$$\begin{aligned} \texttt{Person} = \ & (\texttt{`book} \times (\texttt{Name} \times (X \times Y))) \\ X = \ & (\texttt{Email} \times X) \vee \texttt{nil} \\ Y = \ & (\texttt{Tel} \times \texttt{nil}) \vee \texttt{nil} \end{aligned}$$

where `nil` and `` `book `` are singleton types.

In order to preserve the semantics of XML types as sets of documents but also to give programmers a very intuitive interpretation of types, it is advisable to interpret a

type as the set of all values that have that type. Accordingly, `Int` is interpreted as the set that contains the values `0`, `1`, `-1`, `2`, ...; `Bool` is interpreted as the set that contains the values `true` and `false`; and so on. In particular, then, unions, intersections, and negations (*i.e.*, type *connectives*) must have a set-theoretic semantics. Formally, this corresponds to defining an interpretation function as follows:

Definition 2.1.2. *A* set-theoretic interpretation *of $\mathcal{T}$ is given by some domain $\mathcal{D}$ and a function $[\![\_]\!] : \mathcal{T} \to \mathscr{P}(\mathcal{D})$ such that, for all $t_1, t_2, t \in \mathcal{T}$:*

$$\begin{array}{rcl@{\qquad\qquad}rcl} [\![t_1 \vee t_2]\!] & = & [\![t_1]\!] \cup [\![t_2]\!] & [\![\mathbb{0}]\!] & = & \emptyset \\ [\![t_1 \wedge t_2]\!] & = & [\![t_1]\!] \cap [\![t_2]\!] & [\![\mathbb{1}]\!] & = & \mathcal{D} \\ [\![\neg t]\!] & = & \mathcal{D} \setminus [\![t]\!] & & & \end{array}$$

Note that this function says nothing about the interpretation of atoms (*i.e.*, basics, products and arrows). Indeed, by an induction on types, the set-theoretic interpretations with domain $\mathcal{D}$ correspond univocally to functions from atoms to $\mathscr{P}(\mathcal{D})$. Once such an interpretation has been defined, then the subtyping relation is naturally defined as the inclusion relation.

Definition 2.1.3. *Let $[\![\_]\!] : \mathcal{T} \to \mathscr{P}(\mathcal{D})$ be a set-theoretic interpretation. The* subtyping *relation $\leq_{[\![\,]\!]} \subseteq \mathcal{T}^2$ is defined as follows:*

$$t \leq_{[\![\,]\!]} s \quad \stackrel{\text{def}}{\iff} \quad [\![t]\!] \subseteq [\![s]\!]$$

When restricted to XML types, this definition corresponds to the standard interpretation of subtyping as tree language containment.

As long as basic and product types are the only atoms and a product type $(t_1 \times t_2)$ is standardly interpreted as the Cartesian product $[\![t_1]\!] \times [\![t_2]\!]$, all definitions above run quite smoothly. That is the setting of XDuce studied by Hosoya and Pierce [HP03]. But as soon as higher-order functions are added, that is, arrow types, the definitions above no longer work:

1. If we take $\mathcal{D}$ as the set of all values, then it must include also λ-abstractions. Therefore, to define the semantic interpretation of types we need to define the type of λ-abstractions (in particular of the applications that may occur in their bodies) which needs the subtyping relation, which needs the semantic interpretation. We fall on a circularity.
2. If we take $\mathcal{D}$ as some mathematical domain, then $t_1 \to t_2$ must be interpreted as the set of functions from $[\![t_1]\!]$ to $[\![t_2]\!]$. For instance if we consider functions as binary relations, then $[\![t_1 \to t_2]\!]$ could denote the set

$$\{\ f \subseteq \mathcal{D}^2 \mid (d_1, d_2) \in f \text{ and } d_1 \in [\![t_1]\!] \text{ implies } d_2 \in [\![t_2]\!]\ \} \tag{2.1}$$

   or, compactly, $\mathscr{P}(\overline{[\![t_1]\!] \times \overline{[\![t_2]\!]}})$, where the $\overline{S}$ denotes the complement of the set S within the appropriate universe. In other words, these are the sets of pairs in which it is *not* true that the first projection belongs to $[\![t_1]\!]$ and the second does *not* belong to $[\![t_2]\!]$. But then the problem is not circularity but cardinality, since this would require $\mathcal{D}$ to contain $\mathscr{P}(\mathcal{D}^2)$, which is impossible.

The solution to *both* problems is given by the theory of semantic subtyping [FCB08], and relies on the observation that in order to use types in a programming language one does not need to know what types are, but just how they are related (by subtyping). In other terms, the semantic interpretation is not required to map arrow types into the set in (2.1), but just to map them into sets that induce the same subtyping relation as (2.1) does. Roughly speaking, this turns out to require that for all s_1, s_2, t_1, t_2, the function $[\![\_]\!]$ satisfies the property:

$$[\![s_1 \to s_2]\!] \subseteq [\![t_1 \to t_2]\!] \iff \mathscr{P}(\overline{[\![s_1]\!] \times \overline{[\![s_2]\!]}}) \subseteq \mathscr{P}(\overline{[\![t_1]\!] \times \overline{[\![t_2]\!]}}) \tag{2.2}$$

whatever the sets denoted by $s_1 \to s_2$ and $t_1 \to t_2$ are. To put it otherwise, instead of univocally defining the semantics of arrow by setting $[\![s_1 \to s_2]\!]$ as equal to $\mathscr{P}(\overline{[\![s_1]\!] \times \overline{[\![s_2]\!]}})$ and $[\![t_1 \to t_2]\!]$ as equal to $\mathscr{P}(\overline{[\![t_1]\!] \times \overline{[\![t_2]\!]}})$, the much weaker condition, expressed by (2.2), that whatever the interpretation of arrows is it must induce the same containment relation *as if* arrows were interpreted into the set in (2.1) is imposed.

Equation (2.2) above covers only the case in which two single arrow types are compared. But, of course, a similar restriction must be imposed also when arbitrary Boolean combinations of arrows are compared. Formally, this can be enforced through a new mapping $\mathbb{E}_{[\![\ ]\!]}$, which is associated to $[\![\_]\!]$ and called extensional interpretation. Henceforth the subscript $[\![\ ]\!]$ is omitted if it is clear from context.

Definition 2.1.4. *Let $[\![\_]\!] : \mathcal{T} \to \mathscr{P}(\mathcal{D})$ be a set-theoretic interpretation. Its associated* extensional interpretation *is the unique function $\mathbb{E}(\_)$ defined as follows:*

$$\begin{aligned} \mathbb{E}(\mathbb{0}) &= \emptyset & \mathbb{E}(\mathbb{1}) &= \mathcal{D} \\ \mathbb{E}(\neg t) &= \mathcal{D} \setminus \mathbb{E}(t) & \mathbb{E}(b) &= [\![b]\!] \\ \mathbb{E}(t_1 \vee t_2) &= \mathbb{E}(t_1) \cup \mathbb{E}(t_2) & \mathbb{E}(t_1 \times t_2) &= [\![t_1]\!] \times [\![t_2]\!] \\ \mathbb{E}(t_1 \wedge t_2) &= \mathbb{E}(t_1) \cap \mathbb{E}(t_2) & \mathbb{E}(t_1 \to t_2) &= \mathscr{P}(\overline{[\![t_1]\!] \times \overline{[\![t_2]\!]}}) \end{aligned}$$

Then $[\![\_]\!]$ forms a set-theoretic *model* of types if it induces the same subtyping relation as if type constructors were interpreted in an extensional way.

Definition 2.1.5. *A set-theoretic interpretation $[\![\_]\!] : \mathcal{T} \to \mathscr{P}(\mathcal{D})$ is a* model *if it induces the same subtyping relation as its associated extensional interpretation, that is:*

$$\forall t_1, t_2 \in \mathcal{T} \,.\, [\![t_1]\!] \subseteq [\![t_2]\!] \iff \mathbb{E}(t_1) \subseteq \mathbb{E}(t_2)$$

These definitions yield a subtyping relation with all the desired properties: type connectives (*i.e.*, unions, intersections, and negations) have a set-theoretic semantics, type constructors (*i.e.*, products and arrows) behave as set-theoretic products and function spaces, and (with some care in defining the language and its typing relation) a type can be interpreted as the set of values that have that type. All that remains to do is:

1. to show that a model exists (which is easy) and
2. to show how to decide the subtyping relation (which is difficult).

Both points are solved in [FCB08] and the resulting type system is at the core of the programming language ℂDuce [BCF03a].

Theorem 2.1.6 (Lemma 5.6 in [FCB08])**.** *There exists a model.*

Theorem 2.1.7 (Lemma 5.8 in [FCB08])**.** *The subtyping relation induced by models is decidable.*

Here, we do not need to look for a particular model, since all models induce essentially the same subtyping relation. However for the subtyping algorithm we will use the particular interpretation that maps $[\![t_1 \times t_2]\!]$ into $[\![t_1]\!]\times[\![t_2]\!]$ and $[\![t_1 \to t_2]\!]$ into $\mathscr{P}_f(\overline{[\![t_1]\!]\times\overline{[\![t_2]\!]}})$, where $\mathscr{P}_f(X)$ denotes the set of finite subsets of X. It is easy to check that this interpretation is a model as well (see [Fri04] for details).

Based on set-theoretic theory, there are two important decomposition rules used in the subtyping algorithm: one for product types and the other for arrow types.

Lemma 2.1.8 (Lemma 6.5 in [FCB08])**.** *Let P, N be two finite sets of product types. Then:*

$$\bigcap_{t_1\times t_2\in P} \mathbb{E}(t_1 \times t_2) \subseteq \bigcup_{t_1\times t_2\in N} \mathbb{E}(t_1 \times t_2) \iff$$

$$\forall N' \subseteq N.\ \begin{cases} [\![\bigwedge_{t_1\times t_2\in P} t_1 \wedge \bigwedge_{t_1\times t_2\in N'} \neg t_1]\!] = \emptyset \\ \textit{or} \\ [\![\bigwedge_{t_1\times t_2\in P} t_2 \wedge \bigwedge_{t_1\times t_2\in N\setminus N'} \neg t_2]\!] = \emptyset \end{cases}$$

Lemma 2.1.9 (Lemma 6.8 in [FCB08])**.** *Let P, N be two finite sets of arrow types. Then:*

$$\bigcap_{t\to s\in P} \mathbb{E}(t \to s) \subseteq \bigcup_{t\to s\in N} \mathbb{E}(t \to s) \iff$$

$$\exists (t_0 \to s_0) \in N.\ \forall P' \subseteq P.\ \begin{cases} [\![t_0 \wedge \bigwedge_{t\to s\in P'} \neg t]\!] = \emptyset \\ \textit{or} \\ P \neq P' \textit{ and } [\![\bigwedge_{t\to s\in P\setminus P'} s \wedge \neg s_0]\!] = \emptyset \end{cases}$$

2.2 Core calculus

The core of ℂDuce is a λ-calculus with explicitly-typed recursive functions, pairs and a type-case expression, which we dub "Core ℂDuce".

Definition 2.2.1. *An* expression *in Core ℂDuce is inductively produced by the following grammar:*

$$e \ ::= \ c \ \mid \ x \ \mid \ (e, e) \ \mid \ \pi_i(e) \ \mid \ e\ e \ \mid \ \mu^{\wedge_{i\in I} s_i\to t_i} f\ \lambda x.\ e \ \mid \ e\in t\,?\,e : e$$

*where c ranges over constants (*e.g.*, Booleans, integers, characters, and so on), the λ-abstraction comes with an intersection of arrow types $\bigwedge_{i\in I} s_i \to t_i$ such that the whole λ-abstraction is explicitly typed by $\bigwedge_{i\in I} s_i \to t_i$, and $e\in t\,?\,e_1 : e_2$ denotes the type-case expression that evaluates either e_1 or e_2 according to whether the value returned by e (if any) is of type t or not.*

The reason of inclusion of a type-case is twofold. First, a natural application of intersection types is to type overloaded functions, and without a type-case only "coherent overloading" *à la* Forsythe [Rey96] can be defined (which, for instance, precludes the definition of a —non diverging— function of type, say, $(\texttt{Int}\to\texttt{Bool})\wedge(\texttt{Bool}\to\texttt{Int})$). The second motivation derives from the way arrow types are interpreted in (2.1). In particular for every type s_1, s_2, t_1, t_2 the following containment is *strict*:

$$s_1 \vee s_2 \to t_1 \wedge t_2 \quad \leq \quad (s_1 \to t_1) \wedge (s_2 \to t_2) \tag{2.3}$$

so there is a function in the type on the right that is not in the type of the left. Notice that from a typing viewpoint the functions on the left do not distinguish inputs of s_1 and s_2 types, while the ones on the right do. So the interpretation of types naturally induces the definition of functions that can distinguish inputs of two different types s_1 and s_2 whatever s_1 and s_2 are. Actually this second motivation is just a different facet of the full-fledged vs. only "coherent" overloading motivation, since the functions that are in the difference of the two types in (2.3) are also those that make the difference between coherent and non coherent overloading. Both arguments, thus, advocate for "real" overloaded functions, that execute different code according to the type of their input, whence the need of type-case.

The need of explicitly typed functions is a direct consequence of the introduction of the type case, because without explicit typing there could be some paradoxes such as the following recursively defined (constant) function

$$\mu f.\lambda x.f\in(\mathbb{1}\to\texttt{Int})\,?\,\texttt{true}:\texttt{42} \tag{2.4}$$

This function has type $\mathbb{1}\to\texttt{Int}$ if and only if it *does not* have type $\mathbb{1}\to\texttt{Int}$. In order to decide whether the function above is well-typed or not, a type must be explicitly given to it. For instance, the function in (2.4) is well-typed if it is explicitly assigned the type $\mathbb{1}\to(\texttt{Int}\vee\texttt{Bool})$. This shows both that functions must be explicitly typed and that specifying not only the type of parameters but also the type of the result is strictly more expressive, as more expressions can be typed. As a matter of fact, if just the type of the parameter x (not used in the body) is provided, then there is no type (apart from the useless $\mathbb{0}$ type) that makes (2.4) typeable.

More generally, if a Church-style abstraction is adopted and functions are typed as $\lambda x^t.e$, then all the functions would have a principal type of the form $t\to s$. But then it would not be possible to write a terminating function of type $(\texttt{Int}\to\texttt{Bool})\wedge(\texttt{Bool}\to\texttt{Int})$ (the only way to obtain the previous type by instantiating and/or subsuming a type of the form $s\to t$, is to start from $(\texttt{Int}\vee\texttt{Bool})\to\mathbb{0}$, so only non terminating function could be defined). While it is pretty easy to write a function with the type if the whole abstractions rather than their parameters are explicitly typed:

$$\lambda^{(\texttt{Int}\to\texttt{Bool})\wedge(\texttt{Bool}\to\texttt{Int})}x.x\in\texttt{Int}\,?\,\texttt{true}:42$$

Therefore, the whole λ-abstractions are explicitly typed instead.

The typing judgments for expressions are of form $\Gamma\vdash_C e:t$, where e is an expression, t is a type, and Γ is a typing environment (*i.e.* a finite mapping from expression variables to types). The typing rules[1] are present in Figure 2.1. When Γ is empty, we write $\vdash_C e:t$ for short.

[1]For simplicity, we do not consider the negation arrow types in the typing rule for abstractions.

$$\frac{}{\Gamma \vdash_C c : b_c}\ (Cconst) \qquad \frac{}{\Gamma \vdash_C x : \Gamma(x)}\ (Cvar)$$

$$\frac{\Gamma \vdash_C e_1 : t_1 \qquad \Gamma \vdash_C e_2 : t_2}{\Gamma \vdash_C (e_1, e_2) : t_1 \times t_2}\ (Cpair) \qquad \frac{\Gamma \vdash_C e : t_1 \times t_2}{\Gamma \vdash_C \pi_i(e) : t_i}\ (Cproj)$$

$$\frac{\Gamma \vdash_C e_1 : t_1 \to t_2 \qquad \Gamma \vdash_C e_2 : t_1}{\Gamma \vdash_C e_1 e_2 : t_2}\ (Cappl)$$

$$\frac{\forall i \in I.\ \Gamma, (f : \bigwedge_{i\in I} t_i \to s_i), (x : t_i) \vdash_C e : s_i}{\Gamma \vdash_C \mu^{\bigwedge_{i\in I} t_i \to s_i} f\ \lambda x.\ e : \bigwedge_{i\in I} t_i \to s_i}\ (Cabstr)$$

$$\frac{\Gamma \vdash_C e : t' \qquad \begin{cases} t' \not\leq \neg t & \Rightarrow \quad \Gamma \vdash_C e_1 : s \\ t' \not\leq t & \Rightarrow \quad \Gamma \vdash_C e_2 : s \end{cases}}{\Gamma \vdash_C (e \in t\,?\,e_1 : e_2) : s}\ (Ccase)$$

$$\frac{\Gamma \vdash_C e : s \qquad s \leq t}{\Gamma \vdash_C e : t}\ (Csubsum)$$

Figure 2.1: Typing rules of Core$\mathbb{C}$Duce

Definition 2.2.2. *An expression e is a* value *if it is closed, well-typed ($\vdash_C e : t$ for some type t), and produced by the following grammar:*

$$v \ ::= \ c \mid (v, v) \mid \mu^{\bigwedge_{i\in I} s_i \to t_i} f\ \lambda x.\ e$$

These values are enough to encode XML documents. For example in ℂDuce, lists are encoded, *à la* Lisp, as nested pairs, the empty list being represented by the atom `nil`. Thus an XML document[2] is the pair of its tag, represented by an atom and the list of its children (its content).

Finally, the dynamic semantics is given by the three notions of reduction applied by a leftmost-outermost strategy in Figure 2.2.

[2]The attributes are excluded from the formal treatment.

$$
\begin{array}{ll}
(CRproj) & \pi_i(v_1, v_2) \rightsquigarrow_C v_i \\
(CRappl) & (\mu^{\wedge_{i\in I} t_i \to s_i} f\ \lambda x.\ e')v \rightsquigarrow_C e'\{\mu^{\wedge_{i\in I} t_i \to s_i} f\ \lambda x.\ e'/f, v/x\} \\
(CRcase) & (v \in t\ ?\ e_1\ :\ e_2) \rightsquigarrow_C \begin{cases} e_1 & \text{if } \vdash_C v : t \\ e_2 & \text{otherwise} \end{cases}
\end{array}
$$

Figure 2.2: Reduction rules of Core$\mathbb{C}$Duce

Part II

Polymorphic Semantic Subtyping

Chapter 3

The Key Ideas

In this chapter, we focus on the definition of the subtyping relation for XML types in the presence of type variables, and present an informal description of the main ideas and intuitions underlying our approach. More precisely, we first examine why this problem is deemed unfeasible or unpractical and simple solutions do not work (Section 3.1). Then we present the intuition underlying our solution (Section 3.2) and outline, in an informal way, the main properties that make the definition of subtyping possible (Section 3.3) as well as the key technical details of the algorithm that decides it (Section 3.4). We conclude this chapter by giving some examples of the subtyping relation (Section 3.5) and discussing some related work (Section 3.6). The formal development is described in Chapter 4.

3.1 A naive (wrong) solution

The problem we want to solve in this chapter is how to extend the approach of semantic subtyping described in Chapter 2 when we add type variables (in bold):

$$t \ ::= \ \boldsymbol{\alpha} \ \mid \ b \ \mid \ t \times t \ \mid \ t \to t \ \mid \ t \vee t \ \mid \ t \wedge t \ \mid \ \neg t \ \mid \ \mathbb{0} \ \mid \ \mathbb{1}$$

where α ranges over a countable set of type variables $\mathcal{V}$. We use $\mathcal{T}$ to denote the set of all types.

For a simple setting, we did not include any explicit quantification for type variables: in this work (as well as, all works in the domain we are aware of, foremost [HFC09, Vou06]) we focus on prenex parametric polymorphism where type quantification is meta-theoretic.

Once more, the crux of the problem is how to *define* the subtyping relation between two types that contain type variables. Since we know how to subtype ground types (*i.e.*, types without variables), then a naive solution is to reuse this relation by considering all possible ground instances of types with variables. Let θ denote a *ground substitution*, that is a substitution from type variables to ground types. Then, according to our naive definition, two types are in subtyping relation if so are their ground instances:

$$t_1 \leq t_2 \quad \stackrel{\text{def}}{\iff} \quad \forall \theta.\ [\![t_1\theta]\!] \subseteq [\![t_2\theta]\!] \tag{3.1}$$

(provided that the domain of θ contains all the variables occurring in t_1 and t_2) where $[\![\_]\!]$ is thus the interpretation defined for ground types in Chapter 2. This closely matches the syntactic intuition of subtyping for prenex polymorphism according to which the statement $t_1 \leq t_2$ is to be intended as $\forall\alpha_1...\alpha_n(t_1 \leq t_2)$, where $\alpha_1...\alpha_n$ are all the variables occurring in t_1 or t_2. Clearly, the containment on the right hand side of (3.1) is a *necessary* condition for subtyping (this property is proved for our system by Lemma 4.4.3). Unfortunately, considering it also as *sufficient* and, thus, using (3.1) to define subtyping, yields a subtyping relation that suffers too many problems to be useful.

The first obstacle is that, as conjectured by Hosoya in [HFC09], if the subtyping relation defined by (3.1) is decidable (which is an open problem, though we believe more in undecidability), then deciding it is at least as hard as the satisfiability problem for set constraint systems with *negative constraints* [AKW95, GTT99], which is NEXPTIME-complete [Ste94] and for which, so far, no practical algorithm is known.

But even if the subtyping relation defined by (3.1) were decidable and Hosoya's conjecture wrong, definition (3.1) yields a subtyping relation that misses the intuitiveness of the relation on ground types. This can be shown by an example drawn from [HFC09]. For the sake of the example, imagine that our system includes singleton types, that is types that contain just one value, for every value of the language. Then, consider the following subtyping statement:

$$t \times \alpha \quad \leq \quad (t \times \neg t) \vee (\alpha \times t) \tag{3.2}$$

where t is a ground type.
According to (3.1) the statement holds if and only if $t \times s \leq (t \times \neg t) \vee (s \times t)$ holds for every ground type s. It is easy to see that the latter holds if and only if t is a singleton type. This follows from the set theoretic property that if S is a singleton, then for every set X, either $S \subseteq X$ or $X \subseteq \overline{S}$. By using this property on the singleton type t, we deduce that for every ground substitution of α either (the relation obtained by applying the substitution to) $\alpha \leq \neg t$ holds (therefore $t \times \alpha \leq t \times \neg t$ also holds, whence (3.2) follows) or (the relation obtained by applying the substitution to) $t \leq \alpha$ holds (therefore $t \times \alpha = (t \times \alpha \backslash t) \vee (t \times t)$ holds and the latter is contained component-wise in $(t \times \neg t) \vee (\alpha \times t)$, whence (3.2) holds again). Vice versa, if t contains at least two values, then substituting α by any singleton containing just one value of t disproves the containment.

More generally, (3.2) holds if and only if t is an *indivisible* type, that is, a non-empty type whose only proper subtype is the empty type. Singleton types are just an example of indivisible types, but in the absence of singleton types, basic types that are pairwise disjoint are indivisible as well. Therefore, while the case of singleton types is evocative, the same problem also occurs in a language with, say, just the `Int` type.

Equation (3.2) is pivotal in our work. It gives us two reasons to think that the subtyping relation defined by (3.1) is unfit to be used in practice. First, it tells us that in such a system deciding subtyping is at least as difficult as deciding the indivisibility of a type. This is a very hard problem (see [CDV08] for an instance of this problem in a simpler setting) that makes us believe more in the undecidability of the relation, than in its decidability. Second, and much worse, it completely breaks parametricity [Rey83] yielding a completely non-intuitive subtyping relation. Indeed notice that in the two

types in (3.2) the type variable α occurs on the right component of a product in one type and on the left component of a product in the other. The idea of parametricity is that a function cannot explore arguments whose type is a type variable, it can just discard them, pass them to another function or copy them into the result. Now if (3.1) holds it means that by a simple subsumption a function that is parametric in its second argument can be considered parametric in its first argument instead. Understanding the intuition underlying this subtyping relation for type variables (where the same type variable may appear in unrelated positions in two related types) seems out of reach of even theoretically-oriented programmers. This is why a semantic approach for subtyping polymorphic types has been deemed unfeasible and discarded in favor of partial or syntactic solutions (see related works in Section 3.6).

3.2 Ideas for a solution

Although the problems pointed out in [HFC09] are substantial, they do not preclude a semantic approach to parametric polymorphism. Furthermore the shortcomings caused by the absence of this approach make the study well worth of trying. Here we show that—paraphrasing a famous article by John Reynolds [Rey84]—subtyping of polymorphism *is* set-theoretic.

The conjecture that we have been following since we discovered the problem of [HFC09], and that is at the basis of all this work, is that *the loss of parametricity is only due to the behavior of indivisible types*, all the rest works (more or less) smoothly. The crux of the problem is that for an indivisible type t the validity of the formula

$$t \leq \alpha \quad \text{or} \quad \alpha \leq \neg t \tag{3.3}$$

can *stutter* from one subformula to the other (according to the assignment of α) losing in this way the uniformity typical of parametricity. If we can give a *semantic* characterization of models in which *stuttering* is absent, we believed this would have yielded a subtyping relation that is (*i*) semantic, (*ii*) intuitive for the programmer,[1] and (*iii*) decidable. The problem with indivisible types is that they are either completely inside or completely outside any other type. What we need, then, is to make indivisible types "splittable", so that type variables can range over strict subsets of any type, indivisible ones included. Since this is impossible at a syntactic level, we shall do it at a semantic level. First, we replace ground substitutions with semantic (set) assignments of type variables, $\eta : \mathcal{V} \to \mathscr{P}(\mathcal{D})$ (in set-theoretic notation, $\eta \in \mathscr{P}(\mathcal{D})^{\mathcal{V}}$), and add to interpretation functions a semantic assignment as a further parameter (as it is customary in denotational semantics):

$$[\![.]\!] : \mathcal{T} \to \mathscr{P}(\mathcal{D})^{\mathcal{V}} \to \mathscr{P}(\mathcal{D}).$$

Such an interpretation (actually, the pair $([\![.]\!], \mathcal{D})$) is then a *set-theoretic model* if and only if for all assignments η it satisfies the following conditions (in bold the condition

[1] For instance, type variables can only be subsumed to themselves and according to whether they occur in a covariant or contravariant position, to $\mathbb{1}$ and to unions in which they explicitly appear or to $\mathbb{0}$ and intersections in which they explicitly appear, respectively. De Morgan's laws can be used to reduce other cases to one of these.

that shows the role of the assignment parameter η):

$$\begin{array}{cc} [\![\boldsymbol{\alpha}]\!]\boldsymbol{\eta} = \boldsymbol{\eta}(\boldsymbol{\alpha}) & [\![\neg t]\!]\eta = \mathcal{D}\backslash[\![t]\!]\eta \\ [\![\mathbb{0}]\!]\eta = \varnothing & [\![t_1 \vee t_2]\!]\eta = [\![t_1]\!]\eta \cup [\![t_2]\!]\eta \\ [\![\mathbb{1}]\!]\eta = \mathcal{D} & [\![t_1 \wedge t_2]\!]\eta = [\![t_1]\!]\eta \cap [\![t_2]\!]\eta \\ \multicolumn{2}{c}{[\![t_1]\!]\eta \subseteq [\![t_2]\!]\eta \iff \mathbb{E}(t_1)\eta \subseteq \mathbb{E}(t_2)\eta} \end{array}$$

where $\mathbb{E}()$ is extended in the obvious way to cope with semantic assignments, that is,

$$\begin{array}{cc} \mathbb{E}(\neg t)\eta = \mathcal{D} \setminus \mathbb{E}(t)\eta & \mathbb{E}(b)\eta = [\![b]\!]\eta \\ \mathbb{E}(t_1 \vee t_2)\eta = \mathbb{E}(t_1) \cup \mathbb{E}(t_2)\eta & \mathbb{E}(t_1 \times t_2)\eta = [\![t_1]\!]\eta \times [\![t_2]\!]\eta \\ \mathbb{E}(t_1 \wedge t_2)\eta = \mathbb{E}(t_1)\eta \cap \mathbb{E}(t_2)\eta & \mathbb{E}(t_1 \to t_2)\eta = \mathscr{P}(\overline{[\![t_1]\!]\eta \times \overline{[\![t_2]\!]\eta}}) \\ \mathbb{E}(\boldsymbol{\alpha})\boldsymbol{\eta} = \boldsymbol{\eta}(\boldsymbol{\alpha}) & \mathbb{E}(\mathbb{0})\eta = \emptyset \qquad \mathbb{E}(\mathbb{1})\eta = \mathcal{D} \end{array}$$

Then the subtyping relation is defined as follows:

$$t_1 \leq t_2 \stackrel{\text{def}}{\iff} \forall \eta \in \mathscr{P}(\mathcal{D})^{\mathcal{V}} \,.\, [\![t_1]\!]\eta \subseteq [\![t_2]\!]\eta \tag{3.4}$$

In this setting, every type t that denotes a set of at least two elements of $\mathcal{D}$ can be split by an assignment. That is, it is possible to define an assignment for which a type variable α denotes a subset of $\mathcal{D}$ such that is the interpretation of t neither completely inside nor completely outside the set. Therefore for such a type t, neither equation (3.3) nor, *a fortiori*, equation (3.2) hold. It is then clear that the stuttering of (3.3) is absent in every set-theoretic model in which all non-empty types—indivisible types included—denote *infinite* subsets of $\mathcal{D}$. Infinite denotations for non-empty types look as a possible, though specific, solution to the problem of indivisible types. But what we are looking for is not a particular solution. We are looking for a semantic characterization of the "uniformity" that characterizes parametricity, in order to define a subtyping relation that is, we repeat, semantic, intuitive, and decidable.

This characterization is provided by the property of *convexity*.

3.3 Convexity

A set theoretic model $([\![.]\!], \mathcal{D})$ is *convex* if and only if for every finite set of types $t_1, \ldots, t_n$ it satisfies the following property:

$$\forall\eta.([\![t_1]\!]\eta{=}\varnothing \text{ or } \cdots \text{ or } [\![t_n]\!]\eta{=}\varnothing) \iff (\forall\eta.[\![t_1]\!]\eta{=}\varnothing) \text{ or } \cdots \text{ or } (\forall\eta.[\![t_n]\!]\eta{=}\varnothing) \tag{3.5}$$

This property is the cornerstone of our approach. As such it deserves detailed comments. It states that, given any finite set of types, if every assignment makes some of these types empty, then it is so because there exists one particular type that is empty for all possible assignments.[2] Therefore convexity forces the interpretation function to behave uniformly on its zeros (*i.e.*, on types whose interpretation is the empty set).

[2] We dubbed this property *convexity* after convex formulas: a formula is convex if whenever it entails a disjunction of formulas, then it entails one of them. The $\Rightarrow$ direction of (3.5) (the other direction is trivial) states the convexity of assignments with respect to emptiness: $\eta \in \mathscr{P}(\mathcal{D})^{\mathcal{V}} \Rightarrow \bigvee_{i\in I}[\![t_i]\!]\eta = \varnothing$ implies that there exists $h \in I$ such that $\eta \in \mathscr{P}(\mathcal{D})^{\mathcal{V}} \Rightarrow [\![t_h]\!]\eta = \varnothing$.

Now, the zeros of the interpretation function play a crucial role in the theory of semantic subtyping, since they completely characterize the subtyping relation. Indeed

$$s \leq t \iff [\![s]\!] \subseteq [\![t]\!] \iff [\![s]\!] \cap \overline{[\![t]\!]} \subseteq \varnothing \iff [\![s \wedge \neg t]\!] = \varnothing.$$

Consequently, checking whether $s \leq t$ is equivalent to checking whether the type $s \wedge \neg t$ is empty; likewise, the condition of a model is equivalent to that for all t $[\![t]\!] = \varnothing \iff \mathbb{E}(t) = \varnothing$. We deduce that convexity forces the subtyping relation to have a uniform behavior and, ergo, rules out non-intuitive relations such as the one in (3.2). This is so because convexity prevents stuttering, insofar as in every convex model ($[\![t \wedge \neg\alpha]\!]\eta = \varnothing$ or $[\![t \wedge \alpha]\!]\eta = \varnothing$) holds for all assignments η if and only if t is empty.

Convexity is the property we seek. The resulting subtyping relation is semantically defined and preserves the set-theoretic semantics of type connectives (union, intersection, negation) and the containment behavior of set-theoretic interpretations of type constructors (set-theoretic products for product types and set-theoretic function spaces for arrow types). Furthermore, the subtyping relation is not only semantic but also intuitive. First, it excludes non-intuitive relations by imposing a uniform behavior distinctive of the parametricity *à la* Reynolds: we believe that parametricity and convexity are connected, despite the fact that the former is defined in terms of *transformations* of related terms while the latter deals only with (subtyping) relations. Second, it is very easy to explain the intuition of type variables to a programmer:

> For what concerns subtyping, a type variable can be considered as a special new user-defined basic type that is unrelated to any other atom but $\mathbb{0}$, $\mathbb{1}$, and itself.[3] Type variables are special because their intersection with any ground type may be non-empty, whatever this type is.

Of course, neither in the theoretical development nor in the subtyping algorithm type variables are dealt with as basic types. They need very subtle and elaborated techniques that form the core of our work. But this complexity is completely transparent to the programmer who can thus rely on a very simple intuition.

All that remains to do is (*i*) to prove the convexity property is not too restrictive, that is, that there exists at least one convex set-theoretic model and (*ii*) to show an algorithm that decides the subtyping relation. Contrary to the ground case, *both* problems are difficult. While their solutions require a lot of technical results (see Chapter 4), the intuition underlying them is relatively simple. For what concerns the existence of a convex set-theoretic model, the intuition can be grasped by considering just the logical fragment of our types, that is, the types in which $\mathbb{0}$ and $\mathbb{1}$ are the only atoms. This corresponds to the (classical) propositional logic where the two atoms represent, respectively, false and true. Next, consider the instance of the convexity property given for just two types, t_1 and t_2. It is possible to prove that every non-degenerate Boolean algebra (*i.e.*, every Boolean algebra with more than two elements)

[3] This holds true even for languages with bounded quantification which, as it is well known, defines the subtyping relation for type variables. Bounded quantification does not require any modification to our system, since it can be encoded by intersections and union: a type variable α bounded by two types s and t —*i.e.*, with the constraint $s \leq \alpha \leq t$— can be encoded by a fresh (unbounded) variable β by replacing $s \vee (\beta \wedge t)$ for every occurrence of α. In other terms the type schema $\forall s \leq \alpha \leq t.u$ is equivalent to the schema $\forall \beta.u\{s \vee (\beta \wedge t)/\alpha\}$.

satisfies it. Reasoning by induction it is possible to prove that convexity for n types is satisfied by any Boolean algebra containing at least n elements and from there deduce that all infinite Boolean algebras satisfy convexity. It is then possible to extend the proof to the case that includes basic, product, and arrow types and deduce the following result:

> Every set-theoretic model of ground types in which non-empty types denote infinite sets is a convex set-theoretic model for the polymorphic types.

Therefore, not only do we have a large class of convex models, but also we recover our initial intuition that models with infinite denotations was a possible way to go.

All that remains to explain is the subtype checking algorithm. We do it in the next section, but before that we want to address the possible doubts of a reader about what the denotation of a "finite" type like `Bool` is in such models. In particular, since this denotation contains not only (the denotations of) `true` and `false` but infinitely many other elements, then the reader can rightly wonder what these other elements are and whether they carry any intuitive meaning. In order to explain this point, let us first reexamine what convexity does for infinite types. Convexity is a condition that makes the interpretation of the subtyping relation robust with respect to extensions. Imagine that the syntax of types includes just one basic type: `Int`. Then `Int` is an indivisible type and therefore there exist non-convex models in which the following relation (which is an instance of equation (3.2) of Section 3.1) holds:

$$\texttt{Int} \times \alpha \quad \leq \quad (\texttt{Int} \times \neg\texttt{Int}) \vee (\alpha \times \texttt{Int}) \tag{3.6}$$

(*e.g.*, a model where `Int` is interpreted by a singleton set: in a non-convex model nothing prevents such an interpretation). Now, suppose to add the type `Odd`, a subtype of `Int`, to the type system (and to extend the interpretation of basic types accordingly): then in these models equation (3.6) no longer holds (the substitution of α by `Odd` disproves it). Should the presence of `Odd` change the containment relation between `Int` and the other types? Semantically this should not happen. A relation as (3.6) should have the same meaning independently from whether `Odd` is included in the syntax of types or not. In other terms we want the addition of `Odd` to yield a conservative extension of the subtyping relation. Therefore, all models in which (3.6) is valid must be discarded. Convexity does it. More generally, convexity rules out models where the simple addition of a subtype may yield non conservative extensions of the subtyping relation (or, equivalently, it rules out all models that do not have enough points to support any possible extension of the subtyping relation).

The point is that convexity pushes this idea to all types, so that their interpretation is independent from the possible syntactic subtypes they may have. It is as if the interpretation of subtyping assumed that every type has at least one (actually, infinitely many) stricter non empty subtype(s). So what could the denotation of type `Bool` be in such a model, then? A possible choice is to interpret `Bool` into a set containing labeled versions of `true` and `false`, where labels are drawn from an infinite set of labels (a similar interpretation was first introduced by Gesbert *et al.* [GGL11]: see Section 3.6 on related work). Here the singleton type `{true}` is interpreted as an infinite set containing differently labeled versions of the denotation of `true`. So if *tt* is a denotation of the value

`true`, then we can imagine that the singleton type `{true}` denotes a set of the form $\{tt,\ tt^{b_1,b_2},\ tt^{b_2,b_3,b_5}, ...\}$. Does this labeling carry any intuitive meaning? One can think of it as representing name subtyping (*i.e.*, syntactic types with a user-defined subtyping relation): these labels are the names of (potential) subtypes of the singleton type `{true}` for which the subtyping relation is defined by name subtyping: a value belongs to a type defined by name subtype only if it is labeled by it: so tt^{b_1,b_2} belongs to the interpretation of the types b_1 and b_2 but not to the interpretation of type b_3, where all these types are subtypes (defined by name subtyping) of the singleton type `{true}`. As we do not want the subtyping relation for `Int` to change (non conservatively) when adding to the system the type `Odd`, so for the same reason we do not want the subtyping relation for singleton types to change when adding by name subtyping new subtypes, even when these subtypes are subtypes of a singleton type. So convexity makes the subtyping relation insensitive to possible extensions by name subtyping. Or, put differently, it ensures that all extensions by name subtyping of a subtyping relation are conservative extensions of the original relation.

3.4 Subtyping algorithm

The subtyping algorithm for the relation induced by convex models can be decomposed in 6 elementary steps. Let us explain the intuition underlying each of them: all missing details can be found in Chapter 4.

First of all, we already said that deciding $t_1 \leq t_2$—*i.e.*, whether for all η, $[\![t_1]\!]\eta \subseteq [\![t_2]\!]\eta$—is equivalent to decide the emptiness of the type $t_1 \wedge \neg t_2$—*i.e.*, whether for all η, $[\![t_1 \wedge \neg t_2]\!]\eta = \varnothing$—. So the first step of the algorithm is to transform the problem $t_1 \leq t_2$ into the problem $t_1 \wedge \neg t_2 \leq \mathbb{0}$:

Step 1: *transform the subtyping problem into an emptiness decision problem.*

Our types are just a propositional logic whose atoms are type variables, $\mathbb{0}$, $\mathbb{1}$, basic, product, and arrow types. We use $\boldsymbol{a}$ to range over atoms and, following the logic nomenclature, call *literal*, ranged over by ℓ, an atom or its negation:

$$\boldsymbol{a} \ ::= \ b \mid t \times t \mid t \to t \mid \mathbb{0} \mid \mathbb{1} \mid \alpha \qquad \ell \ ::= \ \boldsymbol{a} \mid \neg \boldsymbol{a}$$

By using the laws of propositional logic we can transform every type into a disjunctive normal form, that is, into a union of intersections of literals:

$$\bigvee_{i \in I} \bigwedge_{j \in J_i} \ell_j$$

Since the interpretation function preserves the set-theoretic semantics of type connectives, then every type is empty if and only if its disjunctive normal form is empty. So the second step of our algorithm consists of transforming the type $t_1 \wedge \neg t_2$ whose emptiness was to be checked, into a disjunctive normal form:

Step 2: *put the type whose emptiness is to be decided in a disjunctive normal form.*

Next, we have to decide when a normal form, that is, a union of intersections, is empty. A union is empty if and only if every member of the union is empty. Therefore the problem reduces to deciding emptiness of an intersection of literals: $\bigwedge_{i\in I} \ell_i$. Intersections of literals can be straightforwardly simplified. Every occurrence of the literal $\mathbb{1}$ can be erased since it does not change the result of the intersection. If either any of the literals is $\mathbb{0}$ or two literals are a variable and its negation, then we do not have to perform further checks since the intersection is surely empty. An intersection can be simplified also when two literals with different constructors occur in it: if in the intersections there are two atoms of different constructors, say, $t_1 \times t_2$ and $t_1 \to t_2$, then their intersection is empty and so is the whole intersection; if one of the two atoms is negated, say, $t_1 \times t_2$ and $\neg(t_1 \to t_2)$, then it can be eliminated since it contains the one that is not negated; if both atoms are negated, then the intersection can also be simplified (with some more work: *cf.* the formal development in Chapter 4). Therefore the third step of the algorithm is to perform these simplifications so that the problem is reduced to deciding emptiness of intersections that are formed by literals that are (possible negations of) either type variables or atoms all of the same constructor (all basic, all product, or all arrow types):

Step 3: *simplify mixed intersections.*

At this stage we have to decide emptiness of intersections of the form

$$\bigwedge_{i\in I} \boldsymbol{a}_i \wedge \bigwedge_{j\in J} \neg \boldsymbol{a}'_j \wedge \bigwedge_{h\in H} \alpha_h \wedge \bigwedge_{k\in K} \neg\beta_k$$

where all the $\boldsymbol{a}_i$'s and $\boldsymbol{a}'_j$'s are atoms with the same constructor, and where $\{\alpha_h\}_{h\in H}$ and $\{\beta_k\}_{k\in K}$ are disjoint sets of type variables: we just reordered literals so that negated variables and the other negated atoms are grouped together. In this step we want to get rid of the rightmost group in the intersection, that is, the one with negated type variables. In other terms, we want to reduce our problem to deciding the emptiness of an intersections as the above, but where all top-level occurrences of type variables are positive. This is quite easy, and stems from the observation that if a type with a type variable α is empty for every possible assignment of α, then it will be empty also if one replaces $\neg\alpha$ for α in it: exactly the same set of checks will be performed since the denotation of the first type for $\alpha \mapsto S \subseteq \mathcal{D}$ will be equal to the denotation of the second type for $\alpha \mapsto \overline{S} \subseteq \mathcal{D}$. That is to say, $\forall\eta.[\![t]\!]\eta = \varnothing$ if and only if $\forall\eta.[\![t\{\neg\alpha/\alpha\}]\!]\eta = \varnothing$ (where $t\{t'/\alpha\}$ denotes the substitution of t' for α in t). So all the negations of the group of toplevel negated variables can be eliminated by substituting $\neg\beta_k$ for β_k in the $\boldsymbol{a}_i$'s and $\boldsymbol{a}'_j$'s:

Step 4: *eliminate toplevel negative variables.*

Next comes what probably is the trickiest step of the algorithm. We have to prove emptiness of intersections of atoms $\boldsymbol{a}_i$ and negated atoms $\boldsymbol{a}'_j$ all on the same constructors and of positive variables α_k. To lighten the presentation let us consider just the case in which atoms are all product types (the case for arrow types is similar though trickier, while the case for basic types is trivial since it reduces to the case for basic types

without variables). By using De Morgan's laws we can move negated atoms on the right hand-side of the relation so that we have to check the following containment

$$\bigwedge_{t_1\times t_2\in P} t_1\times t_2 \wedge \bigwedge_{h\in H} \alpha_h \quad \leq \quad \bigvee_{t'_1\times t'_2\in N} t'_1\times t'_2 \tag{3.7}$$

where P and N respectively denote the sets of positive and negative atoms. Our goal is to eliminate all top-level occurrences of variables (the α_h's) so that the problem is reduced to checking emptiness of product literals. To that end observe that each α_h is intersected with other products. Therefore whatever the interpretation of α_h is, the only part of its denotation that matters is the one that intersects $\mathcal{D}^2$. Ergo, it is useless, at least at top-level, to check all possible assignments for α_h, since those contained in $\mathcal{D}^2$ will suffice. These can be checked by replacing $\gamma_h^1\times\gamma_h^2$ for α_h, where $\gamma_h^{1,}$, γ_h^2 are fresh type variables. Of course the above reasoning holds for the top-level variables, but nothing tells us that the non top-level occurrences of α_h will intersect any product. So replacing them with just $\gamma_h^1\times\gamma_h^2$ would yield a sound but incomplete check. We rather replace every non toplevel occurrence of α_h by $(\gamma_h^1\times\gamma_h^2)\vee\alpha_h$. This still is a sound substitution since if (3.7) holds, then it must also hold for the case where $(\gamma_h^1\times\gamma_h^2)\vee\alpha_h$ is substituted for α_h (with the $\vee\alpha_h$ part useless for toplevel occurrences). Rather surprisingly, at least at first sight, this substitution is also complete, that is (3.7) holds if and only if the following holds:

$$\bigwedge_{t_1\times t_2\in P} t_1\sigma\times t_2\sigma \wedge \bigwedge_{h\in H} \gamma_h^1\times\gamma_h^2 \quad \leq \quad \bigvee_{t'_1\times t'_2\in N} t'_1\sigma\times t'_2\sigma$$

where σ is the substitution $\{(\gamma_h^1\times\gamma_h^2)\vee\alpha_h/\alpha_h\}_{h\in H}$.[4] As an aside, we signal that this transformation holds only because α_h's are positive: the application of *Step 4* is thus a necessary precondition to the application of this one. We thus succeeded to eliminate all toplevel occurrences of type variables and, thus, we reduced the initial problem to the problem of deciding emptiness of intersections in which all literals are products or negations of products (and similarly for arrows):

Step 5: *eliminate toplevel variables.*

The final step of our algorithm must decompose the type constructors occurring at toplevel in order to recurse or stop. To that end it will use set-theoretic properties to deconstruct atom types and, above all, the convexity property to decompose the emptiness problem into a set of emptiness subproblems (this is where convexity plays an irreplaceable role: without convexity the definition of an algorithm seems to be out of our reach). Let us continue with our example with products. At this stage all it remains to solve is to decide a containment of the following form (we included the products of fresh variables into P):

$$\bigwedge_{t_1\times t_2\in P} t_1\times t_2 \quad \leq \quad \bigvee_{t'_1\times t'_2\in N} t'_1\times t'_2 \tag{3.8}$$

[4]Note that the result of this substitution is equivalent to using the substitution $\{(\gamma_h^1\times\gamma_h^2)\vee\gamma_h^3/\alpha_h\}_{h\in H}$ where γ_h^3 is also a fresh variable: we just spare a new variable by reusing α_h which would be no longer used (actually this artifice makes proofs much easier).

Using the set-theoretic properties of the interpretation function and our definition of subtyping, we can prove (see Lemma 2.1.8) that (3.8) holds if and only if for all $N'\subseteq N$,

$$\forall\eta.\left(\llbracket \bigwedge_{t_1\times t_2\in P} t_1 \wedge \bigwedge_{t'_1\times t'_2\in N'} \neg t'_1 \rrbracket\eta=\varnothing \;\;\mathsf{or}\;\; \llbracket \bigwedge_{t_1\times t_2\in P} t_2 \wedge \bigwedge_{t'_1\times t'_2\in N\setminus N'} \neg t'_2 \rrbracket\eta=\varnothing\right)$$

We can now apply the *convexity* property and distribute the quantification on η over each subformula of the or. Thus (3.8) holds if and only if for all $N'\subseteq N$,

$$\forall\eta.\left(\llbracket \bigwedge_{t_1\times t_2\in P} t_1 \wedge \bigwedge_{t'_1\times t'_2\in N'} \neg t'_1 \rrbracket\eta=\varnothing\right) \;\;\mathsf{or}\;\; \forall\eta.\left(\llbracket \bigwedge_{t_1\times t_2\in P} t_2 \wedge \bigwedge_{t'_1\times t'_2\in N\setminus N'} \neg t'_2 \rrbracket\eta=\varnothing\right)$$

This is equivalent to state that we have to check the emptiness for each type that occurs as argument of the interpretation function. Playing a little more with De Morgan's laws and applying the definition of subtyping we can thus prove that (3.8) holds if and only if

$$\forall N'\subseteq N.\left(\bigwedge_{t_1\times t_2\in P} t_1 \leq \bigvee_{t'_1\times t'_2\in N'} t'_1\right) \;\;\mathsf{or}\;\; \left(\bigwedge_{t_1\times t_2\in P} t_2 \leq \bigvee_{t'_1\times t'_2\in N\setminus N'} t'_2\right)$$

To understand the rationale of this transformation the reader can consider the case in which both P and N contain just one atom, namely, the case for $t_1\times t_2 \leq t'_1\times t'_2$. There are just two cases to check ($N'=\varnothing$ and $N'=N$) and it is not difficult to see that the condition above becomes: $(t_1\leq\mathbb{0})$ or $(t_2\leq\mathbb{0})$ or $(t_1\leq t'_1$ and $t_2\leq t'_2)$, as expected.

The important point however is that we were able to express the problem of (3.8) in terms of subproblems that rest on strict subterms (there is a similar decomposition rule for arrow types). Remember that our types are possibly infinite trees since they were *coinductively* generated by the grammar in Section 3.1. We do not consider every possible coinductively generated tree, but only those that are regular (*i.e.*, that have a finite number of distinct subtrees) and in which every infinite branch contains infinitely many occurrences of type constructors (*i.e.*, products and arrows). The last condition rules out meaningless terms (such as $t = \neg t$) as well as infinite unions and intersections. It also provides a well-founded order that allows us to use recursion. Therefore, we memoize the relation in (3.8) and recursively call the algorithm from *Step 1* on the subterms we obtained from decomposing the toplevel constructors:

Step 6: *eliminate toplevel constructors, memoize, and recurse.*

The algorithm is sound and complete with respect to the subtyping relation defined by (3.4) and terminates on all types (which implies the decidability of the subtyping relation).

3.5 Examples

The purpose of this subsection is twofold: first, we want to give some examples to convey the idea that the subtyping relation is intuitive; second we present some cases that justify the subtler and more technical aspects of the subtyping algorithm we exposed

in the previous subsection (all the examples below can be tested in our prototype subtype-checker or in the subtype checker of [GGL11] which is available online).

In what follows we will use x, y, z to range over recursion variables and the notation $\mu x.t$ to denote recursive types. This should suffice to avoid confusion with free type variables that are ranged over by α, β, and γ.

As a first example we show how to use type variables to internalize meta-properties. For instance, for all ground types t_1, t_2, and t_3 the relation $(t_1 \to t_3) \wedge (t_2 \to t_3) \leq (t_1 \vee t_2) \to t_3$ and its converse hold. This meta-theoretic property can be expressed in our type system since the following relation holds:

$$(\alpha \to \gamma) \wedge (\beta \to \gamma) \;\simeq\; (\alpha \vee \beta) \to \gamma$$

(where $t \simeq s$ denotes that both $t \leq s$ and $s \leq t$ hold). Of course we can apply this generalization to any relation that holds for generic types. For instance, we can prove common distributive laws such as

$$((\alpha \vee \beta) \times \gamma) \;\simeq\; (\alpha \times \gamma) \vee (\beta \times \gamma) \tag{3.9}$$

and combine it with the previous relation and the covariance of arrow on codomains to deduce

$$(\alpha \times \gamma \to \delta_1) \wedge (\beta \times \gamma \to \delta_2) \;\leq\; ((\alpha \vee \beta) \times \gamma) \to \delta_1 \vee \delta_2$$

Similarly we can prove that $\mu x.(\alpha \times x) \vee \mathsf{nil}$ the type of α-lists —*i.e.*, the set of possibly empty lists whose elements have type α — contains both the α-lists with an even number of elements

$$\mu x.(\alpha \times (\alpha \times x)) \vee \mathsf{nil} \;\leq\; \mu x.(\alpha \times x) \vee \mathsf{nil}$$

(where nil denotes the singleton type containing just the value nil) and the α-lists with an odd number of elements

$$\mu x.(\alpha \times (\alpha \times x)) \vee (\alpha \times \mathsf{nil}) \;\leq\; \mu x.(\alpha \times x) \vee \mathsf{nil}$$

and it is itself contained in the union of the two, that is:

$$\mu x.(\alpha \times x) \vee \mathsf{nil} \;\simeq\; (\mu x.(\alpha \times (\alpha \times x)) \vee \mathsf{nil}) \;\vee\; (\mu x.(\alpha \times (\alpha \times x)) \vee (\alpha \times \mathsf{nil}))$$

We said that the intuition for subtyping type variables is to consider them as basic types. But type variables are *not* basic types. As an example, if t is a non-empty type, then we have that:

$$\alpha \wedge (\alpha \times t) \;\not\leq\; t_1 \to t_2$$

which implies that $\alpha \wedge (\alpha \times t)$ is not empty. This is correct because if for instance we substitute the type $t \vee (t \times t)$ for α, then (by the distributivity law stated in (3.9)) the intersection is equal to $(t \times t)$, which is non-empty. However, note that if α were a basic type, then the intersection $\alpha \wedge (\alpha \times t)$ would be empty, since no basic type intersects a product type. Furthermore, since the following relation holds

$$\alpha \wedge (\alpha \times t) \;\leq\; \alpha$$

then, this last containment is an example of non-trivial containment (in the sense that the left hand-side is not empty) involving type variables. For an example of non-trivial containment involving arrows the reader can check

$$\mathbb{1} \to \mathbb{0} \;\leq\; \alpha \to \beta \;\leq \mathbb{0} \to \mathbb{1}$$

which states that $\mathbb{1} \to \mathbb{0}$, the set of all functions that diverge on all arguments, is contained in all arrow types $\alpha \to \beta$ (whatever types α and β are) and that the latter are contained in $\mathbb{0} \to \mathbb{1}$, which is the set of all function values.

Type connectives implement classic proposition logic. If we use $\alpha \Rightarrow \beta$ to denote $\neg\alpha \vee \beta$, that is logical implication, then the following subtyping relation is a proof of Pierce's law:

$$\mathbb{1} \;\leq\; ((\alpha \Rightarrow \beta) \Rightarrow \alpha) \Rightarrow \alpha$$

since being a supertype of $\mathbb{1}$ logically corresponds to being equivalent to true (note that arrow types do not represent logical implication; for instance, $\mathbb{0}\to\mathbb{1}$ is not empty: it contains all function values). Similarly, the system captures the fundamental property that for all non-empty sets β the set $(\beta\wedge\alpha) \vee (\beta\wedge\neg\alpha)$ is never empty:

$$(\beta \wedge \alpha) \vee (\beta \wedge \neg\alpha) \;\simeq\; \beta$$

from which we can derive

$$\mathbb{1} \;\leq\; (\,((\beta\wedge\alpha) \vee (\beta\wedge\neg\alpha)) \Rightarrow \mathbb{0}\,) \;\Rightarrow\; (\beta \Rightarrow \mathbb{0})$$

This last relation can be read as follows: if $(\beta\wedge\alpha) \vee (\beta\wedge\neg\alpha)$ is empty, then β is empty.

But the property above will never show a stuttering validity since the algorithm returns false when asked to prove

$$\mathsf{nil} \times \alpha \quad\leq\quad (\mathsf{nil} \times \neg\mathsf{nil}) \vee (\alpha \times \mathsf{nil})$$

even for a singleton type as nil.

The subtyping relation has some simple form of introspection since $t_1 \leq t_2$ if and only if $\mathbb{1} \leq t_1 \Rightarrow t_2$ (*i.e.*, by negating both types and reversing the subtyping relation, $t_1 \wedge \neg t_2 \leq \mathbb{0}$). However, the introspection capability is very limited insofar as it is possible to state interesting properties only when atoms are type variables: although we can characterize the subtyping relation $\leq$, we have no idea about how to characterize its negation $\not\leq$. [5]

The necessity for the tricky substitution $\alpha \mapsto (\gamma_1 \times \gamma_2) \vee \alpha$ performed at *Step 5* of the algorithm can be understood by considering the following example where t is any non-empty type:

$$(\alpha \times t) \wedge \alpha \;\leq\; ((\mathbb{1} \times \mathbb{1}) \times t).$$

[5] For instance, it would be nice to prove something like:

$$(\neg\beta_1 \vee ((\beta_1 \Rightarrow \alpha_1) \wedge (\alpha_2 \Rightarrow \beta_2))) \;\simeq\; (\alpha_1 \to \alpha_2 \Rightarrow \beta_1 \to \beta_2)$$

since it seems to provide a complete characterization of the subtyping relation between two arrow types. Unfortunately the equivalence is false since $\beta_1 \not\leq \alpha_1$ does not imply $\beta_1 \wedge \neg\alpha_1 \geq \mathbb{1}$ but just $\beta_1 \wedge \neg\alpha_1 \not\leq \mathbb{0}$. This property can be stated only at meta level, that is: $\alpha_1 \to \alpha_2 \leq \beta_1 \to \beta_2$ if and only if ($\beta_1 \leq \mathbb{0}$ or ($\beta_1 \leq \alpha_1$ and $\alpha_2 \leq \beta_2$)).

If in order to check the relation above we substituted just $\gamma_1 \times \gamma_2$ for α, then this would yield a positive result, which is wrong: if we replace α by $b \vee (b \times t)$, where b is any basic type, then the intersection on the left becomes $(b \times t)$ and b is neither contained in $\mathbb{1} \times \mathbb{1}$ nor empty. Our algorithm correctly disproves the containment, since it checks also the substitution of $(\gamma_1 \times \gamma_2) \vee \alpha$ for the first occurrence of α, which captures the above counterexample.

Finally, the system also proves subtler relations whose meaning is not clear at first sight, such as:

$$\alpha_1 \to \beta_1 \;\leq\; ((\alpha_1 \wedge \alpha_2) \to (\beta_1 \wedge \beta_2)) \vee \neg(\alpha_2 \to (\beta_2 \wedge \neg\beta_1)) \tag{3.10}$$

In order to prove it, the subtyping algorithm first moves the occurrence of $\alpha_2 \to (\beta_2 \wedge \neg\beta_1)$ from the right of the subtyping relation to its left: $(\alpha_1 \to \beta_1) \wedge (\alpha_2 \to (\beta_2 \wedge \neg\beta_1)) \leq ((\alpha_1 \wedge \alpha_2) \to (\beta_1 \wedge \beta_2))$; then following the decomposition rules for arrows the algorithm checks the four following cases (Step 6 of the algorithm), which hold straightforwardly:

$$\begin{cases} \alpha_1 \wedge \alpha_2 \leq \mathbb{0} \text{ or } \beta_1 \wedge (\beta_2 \wedge \neg\beta_1) \leq \beta_1 \wedge \beta_2 \\ \alpha_1 \wedge \alpha_2 \leq \alpha_1 \text{ or } (\beta_2 \wedge \neg\beta_1) \leq \beta_1 \wedge \beta_2 \\ \alpha_1 \wedge \alpha_2 \leq \alpha_2 \text{ or } \beta_1 \leq \beta_1 \wedge \beta_2 \\ \alpha_1 \wedge \alpha_2 \leq \alpha_1 \vee \alpha_2 \end{cases}$$

Notice that relation (3.10) is subtle insofar as neither $\alpha_1 \to \beta_1 \leq (\alpha_1 \wedge \alpha_2) \to (\beta_1 \wedge \beta_2)$ nor $\alpha_1 \to \beta_1 \leq \neg(\alpha_2 \to (\beta_2 \wedge \neg\beta_1))$ hold: the type on left hand-side of (3.10) is contained in the union of the two types on the right hand-side of (3.10) without being completely contained in either of them.

3.6 Related work

3.6.1 Semantic subtyping and polymorphism

The definition of polymorphic semantic subtyping extends the work on semantic subtyping [FCB08], as such these two works share the same approach and common developments. Since convex models of our theory can be derived from the models of [FCB08], then several techniques we used in our subtyping algorithm (in particular the decomposition of toplevel type constructors) are directly issued from the research in [FCB08]. Our work starts precisely from where [FCB08] stopped, that is the monomorphic case, and adds prenex parametric polymorphism to it.

Our subtyping algorithm already has a follow-up. In a paper directly issued from the research presented here [GGL11] Gesbert, Genevès, and Layaïda use the framework we define here to give a different decision procedure for our subtyping relation. More precisely, they take a specific model for the monomorphic type system (*i.e.*, the model defined by Frisch *et al.* [FCB08] and used by the language CDuce), they encode the subtyping relation induced by this model into a tree logic, and use a satisfiability solver to efficiently decide it. Next, they extend the type system with type variables and they obtain a convex model by interpreting non-empty types as infinite sets using a labeling technique similar to the one we outlined at the end of Section 3.3: they label values

by (finite sets of) type variables and every non empty ground type is, thus, interpreted as an infinite set containing the infinitely many labelings of its values. Again the satisfiability solver provides a decision procedure for the subtyping relation. Their technique is interesting in several respects. First it provides a very elegant solution to the problem of deciding our subtyping relation, solution that is completely different from the one given here. Second, their technique shows that the decision problem is EXPTIME, (while here we only prove the decidability of the problem by showing the termination of our algorithm). Finally, their logical encoding paves the way to extending types (and subtyping) with more expressive logical constraints representable by their tree logic. In contrast, our algorithm is interesting for quite different reasons: first, it is defined for generic interpretations rather than for a fixed model; second, it shows how convexity is used in practice (see in particular ***Step 6*** of the algorithm); and, finally, our algorithm is a straightforward modification of the algorithm used in $\mathbb{C}$Duce and, as such, can benefit of the technology and optimizations used there.[6] We expect the integration of this subtyping relation in the $\mathbb{C}$Duce to be available in a not so distant future.

3.6.2 XML processing and polymorphism

The most advanced work on polymorphism for XML types, thus far, is the Hosoya, Frisch, and Castagna's approach introduced in [HFC05] and described in details in [HFC09], whose extended abstract was first presented at POPL '05. Together with [FCB08], the paper by Hosoya, Frisch, and Castagna constitutes the starting point of this work. A starting point that, so far, was rather considered to establish a (negative) final point. As a matter of fact, although the polymorphic system in [HFC09] is the one used to define the polymorphic extension of XDuce [HP03] (incorporated from version 0.5.0 of the language[7]), the three authors of [HFC09] agree that the main interest of their work does not reside in its type system, but rather in the negative results that motivate it. In particular, the pivotal example of our work, equation (3.2), was first presented in [HFC09], and used there to corroborate the idea that a purely semantic approach for polymorphism of regular tree types was an hopeless quest. At that time, this seemed so more hopeless that equation (3.2) did not involve arrow types: a semantically defined polymorphic subtyping looked out of reach even in the restrictive setting of Hosoya and Pierce seminal work [HP03], which did not account for higher-order functions. This is why [HFC09] falls back on a syntactic approach that, even if it retains some flavors of semantic subtyping, cannot be extended to higher-order functions (a lack that nevertheless fits XDuce). Our works shows that the negative results of [HFC09] were not so insurmountable as it had been thought.

Hitherto, the only work that blends polymorphic regular types and arrow types is Jérôme Vouillon's work that was presented at POPL '06 [Vou06]. His approach, however, is very different from ours insofar as it is intrinsically syntactic. Vouillon starts

[6] Alain Frisch's PhD. thesis [Fri04] describes two algorithms that improve over the simple saturation-based strategy described in Section 4.4. They are used both in $\mathbb{C}$Duce compiler and in the prototype we implemented to check the subtyping relation presented in this work.

[7] XDuce is the only language we are aware of that can express the polymorphism of the SOAP functions we described in the introduction. However, since it lacks higher-order functions, it cannot express the type of `register_new_service`.

from a particular language (actually, a peculiar pattern algebra) and coinductively builds up on it the subtyping relation by a set of inference rules. The type algebra includes only the union connective (negation and intersection are missing) and a semantic interpretation of subtyping is given *a posteriori* by showing that a pattern (types are special cases of patterns) can be considered as the set of values that match the pattern. Nevertheless, this interpretation is still syntactic in nature since it relies on the definition of matching and containment, yielding a system tailored for the peculiar language of the paper. This allows Vouillon to state impressive and elegant results such as the translation of the calculus into a non-explicitly-typed one, or the interpretation of open types containment as in our equation (3.1) (according to Vouillon this last result is made possible in his system by the absence of intersection types, although the critical example in (3.2) does not involve any intersection). But the price to pay is a system that lacks naturalness (*e.g.*, the wild-card pattern has different meanings according to whether it occurs in the right or in left type of a subtyping relation) and, even more, it lacks the generality of our approach (we did not state our subtype system for any specific language while Vouillon's system is inherently tied to a particular language whose semantics it completely relies on). The semantics of Vouillon's patterns is so different from ours that typing Vouillon's language with our types seems quite difficult.

Other works less related to ours are those in which XML and polymorphism are loosely coupled. This is the case of OCamlDuce [Fri06] where ML-polymorphism and XML types and patterns are merged together without mixing: the main limitation of this approach is that it does not allow parametric polymorphism for XML types, which is the whole point of our (and Vouillon's) work(s). A similar remark can be done for Xtatic [GLPS05b] that merges C# name subtyping with the XDuce set-theoretic subtyping and for XHaskell [LS04] whose main focus is to implement XML subtyping using Haskell's type-classes. A more thorough comparison of these approaches can be found in [Fri06, HFC09].

Polymorphism can be attained by adopting the so-called data-binding approach which consists in encoding XML types and values into the structures of an existing polymorphic programming language. This is the approach followed by HaXML [WR99]. While the polymorphism is inherited from the target language, the rigid encoding of XML data into fixed structures loses all flexibility of the XML type equivalences so as, for instance, $(t \times s_1) \vee (t \times s_2)$ and $(t \times s_1 \vee s_2)$ are different (and even unrelated) types.

Finally, we signal the work on polymorphic iterators for XML presented in [CN08] which consists of a very simple strongly normalizing calculus fashioned to define tree iterators. These iterators are lightly checked at the moment of their definition: the compiler does not complain unless they are irremediably flawed. This optimistic typing, combined with the relatively limited expressive power of the calculus, makes it possible to type iterator applications in a very precise way (essentially, by performing an abstract execution of the iterator on the types) yielding a kind of polymorphism that is out of reach of parametric or subtype polymorphism (for instance it can precisely type the reverse function applied to *heterogeneous* lists and thus deduce that the application of reverse to a list of type, say, `[ Int Bool* Char+ ]` yields a result of type `[ Char+ Bool* Int ]`). As such it is orthogonal to the kind of polymorphism presented here, and both can and should coexist in a same language.

Chapter 4

Formal Development

In this chapter we describe the technical development that supports the results we exposed in the previous chapter.

4.1 Types

Definition 4.1.1 (Types). *Let $\mathcal{V}$ be a countable set of type variables ranged over by Greek letter α,β,γ,..., and $\mathscr{B}$ a finite set of basic (or constant) types ranged over by b. A* type *is a term co-inductively produced by the following grammar*

$$
\begin{array}{lrcll}
\textbf{\textit{Types}} & t & ::= & \alpha & \textit{type variable} \\
 & & & \mid b & \textit{basic} \\
 & & & \mid t \times t & \textit{product} \\
 & & & \mid t \to t & \textit{arrow} \\
 & & & \mid t \vee t & \textit{union} \\
 & & & \mid \neg t & \textit{negation} \\
 & & & \mid \mathbb{0} & \textit{empty}
\end{array}
$$

that satisfies two additional requirements:

- *(regularity) the term must have a finite number of different sub-terms.*
- *(contractivity) every infinite branch must contain an infinite number of occurrences of atoms (*i.e.*, either a type variable or the immediate application of a type constructor: basic, product, arrow).*

We use $\mathcal{T}$ to denote the set of all types.

We write $t_1 \wedge t_2$, $t_1 \setminus t_2$, and $\mathbb{1}$ respectively as an abbreviation for $\neg(\neg t_1 \vee \neg t_2)$, $t_1 \wedge \neg t_2$, and $\neg \mathbb{0}$.

The condition on infinite branches bars out ill-formed types such as $t = t \vee t$ (which does not carry any information about the set denoted by the type) or $t = \neg t$ (which cannot represent any set). It also ensures that the binary relation $\rhd \subseteq \mathcal{T}^2$ defined by $t_1 \vee t_2 \rhd t_i$, $\neg t \rhd t$ is Noetherian (that is, strongly normalizing). This gives an induction principle on $\mathcal{T}$ that we will use without any further explicit reference to the relation.

Since types are infinite then the accessory definitions on them will be given either by using memoization (*e.g.*, the definition of $\mathsf{var}()$, the variables occurring in a type: Definition 4.1.2), of by co-inductive techniques (*e.g.*, the definition or the application of type-substitution: Definition 4.1.5) or by induction on the relation $\rhd$, but only when induction does not traverse a type constructor (*e.g.*, the definition of $\mathsf{tlv}()$, the variables occurring at top-level of a type: Definition 4.1.3).

Definition 4.1.2 (**Type variables**). *Let var_0 and var_1 be two functions from $\mathcal{T} \times \mathscr{P}(\mathcal{T})$ to $\mathscr{P}(\mathcal{V})$ defined as:*

$$
\begin{aligned}
var_0(t, \beth) &= \begin{cases} \emptyset & \text{if } t \in \beth \\ var_1(t, \beth \cup \{t\}) & \text{otherwise} \end{cases} \\
var_1(\alpha, \beth) &= \{\alpha\} \\
var_1(b, \beth) &= \emptyset \\
var_1(t_1 \times t_2, \beth) &= var_0(t_1, \beth) \cup var_0(t_2, \beth) \\
var_1(t_1 \to t_2, \beth) &= var_0(t_1, \beth) \cup var_0(t_2, \beth) \\
var_1(t_1 \vee t_2, \beth) &= var_1(t_1, \beth) \cup var_1(t_2, \beth) \\
var_1(\neg t_1, \beth) &= var_1(t_1, \beth) \\
var_1(\mathbb{0}, \beth) &= \emptyset
\end{aligned}
$$

where $\beth$ is a set of types. The set of type variables *occurring in a type t, written $\mathsf{var}(t)$, is defined as $var_0(t, \emptyset)$. A type t is said to be* ground *or* closed *if and only if $\mathsf{var}(t)$ is empty. We write $\mathcal{T}_0$ to denote the set of all the ground types.*

Definition 4.1.3 (**Top-level variables**). *Let t be a type. The set $\mathsf{tlv}(t)$ of type variables that occur* at top level *in t, that is, all the variables of t that have at least one occurrence not under a constructor, is defined as:*

$$
\begin{aligned}
\mathsf{tlv}(\alpha) &= \{\alpha\} \\
\mathsf{tlv}(b) &= \emptyset \\
\mathsf{tlv}(t_1 \times t_2) &= \emptyset \\
\mathsf{tlv}(t_1 \to t_2) &= \emptyset \\
\mathsf{tlv}(t_1 \vee t_2) &= \mathsf{tlv}(t_1) \cup \mathsf{tlv}(t_2) \\
\mathsf{tlv}(\neg t_1) &= \mathsf{tlv}(t_1) \\
\mathsf{tlv}(\mathbb{0}) &= \emptyset
\end{aligned}
$$

Definition 4.1.4 (**Type substitution**). *A* type-substitution *σ is a total mapping of type variables to types that is the identity everywhere but on a finite subset of $\mathcal{V}$, which is called the domain of σ and denoted by $\mathsf{dom}(\sigma)$. Given a substitution σ, the* range *of σ is defined as the set of types $\mathsf{ran}(\sigma) = \{\sigma(\alpha) \mid \alpha \in \mathsf{dom}(\sigma)\}$, and the set of type variables occurring in the range is defined as $\mathsf{tvran}(\sigma) = \bigcup_{\alpha \in \mathsf{dom}(\sigma)} \mathsf{var}(\sigma(\alpha))$. We use the notation $\{t_1/\alpha_1, \ldots, t_n/\alpha_n\}$ to denote the type-substitution that maps α_i to t_i for $i = 1..n$.*

Definition 4.1.5. *Given a type $t \in \mathcal{T}$ and a type-substitution σ, the* application *of σ*

to t is co-inductively defined as follows:

$$\begin{aligned}
b\sigma &= b \\
(t_1 \times t_2)\sigma &= (t_1\sigma) \times (t_2\sigma) \\
(t_1 \to t_2)\sigma &= (t_1\sigma) \to (t_2\sigma) \\
(t_1 \vee t_2)\sigma &= (t_1\sigma) \vee (t_2\sigma) \\
(\neg t)\sigma &= \neg(t\sigma) \\
\mathbb{0}\sigma &= \mathbb{0} \\
\alpha\sigma &= \sigma(\alpha) && \text{if } \alpha \in \mathsf{dom}(\sigma) \\
\alpha\sigma &= \alpha && \text{if } \alpha \notin \mathsf{dom}(\sigma)
\end{aligned}$$

4.2 Subtyping

Definition 4.2.1 (Assignment). *Given a set $\mathcal{D}$, a* semantic assignment $\eta : \mathcal{V} \to \mathscr{P}(\mathcal{D})$ *is a finite mapping from type variables to subsets of $\mathcal{D}$. The set of type variables that are defined in η is called the domain of η, denoted as* $\mathsf{dom}(\eta)$.

Definition 4.2.2 (Set-theoretic interpretation). *A* set-theoretic interpretation *of $\mathcal{T}$ is given by a set $\mathcal{D}$ and a function $[\![\_]\!] : \mathcal{T} \to \mathscr{P}(\mathcal{D})^{\mathcal{V}} \to \mathscr{P}(\mathcal{D})$ such that, for all $t_1, t_2, t \in \mathcal{T}$, $\alpha \in \mathcal{V}$ and $\eta \in \mathscr{P}(\mathcal{D})^{\mathcal{V}}$:*

- $[\![t_1 \vee t_2]\!]\eta = [\![t_1]\!]\eta \cup [\![t_2]\!]\eta$,
- $[\![\neg t]\!]\eta = \mathcal{D} \setminus [\![t]\!]\eta$,
- $[\![\alpha]\!]\eta = \eta(\alpha)$,
- $[\![\mathbb{0}]\!]\eta = \emptyset$.

Once such an interpretation is defined, the subtyping relation is naturally defined as the inclusion relation:

Definition 4.2.3 (Subtyping relation). *Let $[\![\_]\!] : \mathcal{T} \to \mathscr{P}(\mathcal{D})^{\mathcal{V}} \to \mathscr{P}(\mathcal{D})$ be a set-theoretic interpretation. We define the* subtyping *relation $\leq_{[\![\,]\!]} \subseteq \mathcal{T}^2$ as follows:*

$$t \leq_{[\![\,]\!]} s \quad \stackrel{\text{def}}{\iff} \quad \forall \eta \in \mathscr{P}(\mathcal{D})^{\mathcal{V}} \,.\, [\![t]\!]\eta \subseteq [\![s]\!]\eta$$

We write $t \leq s$ when the interpretation $[\![\_]\!]$ is clear from the context, and $t \simeq s$ if $t \leq s$ and $s \leq t$.

As argued and stated in Section 3.3, the subtyping problem can be transformed into emptiness decision problem, that is the following lemma.

Lemma 4.2.4. *Let t, s be two types, then:*

$$t \leq s \iff \forall \eta \in \mathscr{P}(\mathcal{D})^{\mathcal{V}} \,.\, [\![t \wedge \neg s]\!]\eta = \emptyset.$$

Proof.

$$\begin{aligned}
t \leq s &\iff \forall \eta \in \mathscr{P}(\mathcal{D})^{\mathcal{V}} \,.\, [\![t]\!]\eta \subseteq [\![s]\!]\eta \\
&\iff \forall \eta \in \mathscr{P}(\mathcal{D})^{\mathcal{V}} \,.\, [\![t]\!]\eta \cap (\mathcal{D} \setminus [\![s]\!]\eta) = \emptyset \\
&\iff \forall \eta \in \mathscr{P}(\mathcal{D})^{\mathcal{V}} \,.\, [\![t \wedge \neg s]\!]\eta = \emptyset.
\end{aligned}$$

□

Let $\mathscr{C}$ be a subset of $\mathcal{D}$ whose elements are called *constants*. For each basic type b, we assume there is a fixed set of constants $\mathbb{B}(b) \subseteq \mathscr{C}$ whose elements are called constants of type b. For two basic types b_1, b_2, the sets $\mathbb{B}(b_i)$ can have a nonempty intersection.

If, as suggested in Section 2.1, we interpret extensionally an arrow $t_1 \to t_2$ as $\mathscr{P}(\overline{[\![t_1]\!] \times \overline{[\![t_2]\!]}})$ (precisely as $\mathscr{P}(\mathcal{D}^2 \setminus ([\![t_1]\!] \times (\mathcal{D} \setminus [\![t_2]\!])))$), then every arrow type is a subtype of $\mathbb{1} \to \mathbb{1}$. We do not want such a property to hold because, otherwise, we could subsume every function to a function that accepts every value and, therefore, every application of a well-typed function to a well-typed argument would be well-typed, independently from the types of the function and of the argument. For example, if, say, $\mathsf{succ}\colon \mathtt{Int} \to \mathtt{Int}$, then we could deduce $\mathsf{succ}\colon \mathbb{1} \to \mathbb{1}$ and then $\mathsf{succ}(\mathsf{true})$ would have type $\mathbb{1}$. To avoid this problem we use a technique introduced in [FCB08] and introduce an explicit type error Ω and use it to define function spaces:

Definition 4.2.5. *If D is a set and X, Y are subsets of D, we write D_Ω for $D + \{\Omega\}$ and define $X \to Y$ as:*

$$X \to Y \stackrel{\text{def}}{=} \{f \subseteq D \times D_\Omega \mid \forall (d, d') \in f.\ d \in X \Rightarrow d' \in Y\}$$

This is used in the definition of the extensional interpretation:

Definition 4.2.6 (Extensional interpretation). *Let $[\![\_]\!] : \mathcal{T} \to \mathscr{P}(\mathcal{D})^{\mathcal{V}} \to \mathscr{P}(\mathcal{D})$ be a set-theoretic interpretation. Its associated* extensional interpretation *is the unique function $\mathbb{E}(\_) : \mathcal{T} \to \mathscr{P}(\mathcal{D})^{\mathcal{V}} \to \mathscr{P}(\mathbb{E}\mathcal{D})$ where $\mathbb{E}\mathcal{D} = \mathcal{D} + \mathcal{D}^2 + \mathscr{P}(\mathcal{D} \times \mathcal{D}_\Omega)$, defined as follows:*

$$\begin{array}{rcl}
\mathbb{E}(\alpha)\eta & = & \eta(\alpha) \subseteq \mathcal{D} \\
\mathbb{E}(b)\eta & = & \mathbb{B}(b) \subseteq \mathcal{D} \\
\mathbb{E}(t_1 \times t_2)\eta & = & [\![t_1]\!]\eta \times [\![t_2]\!]\eta \subseteq \mathcal{D}^2 \\
\mathbb{E}(t_1 \to t_2)\eta & = & [\![t_1]\!]\eta \to [\![t_2]\!]\eta \subseteq \mathscr{P}(\mathcal{D} \times \mathcal{D}_\Omega) \\
\mathbb{E}(t_1 \vee t_2)\eta & = & \mathbb{E}(t_1)\eta \cup \mathbb{E}(t_2)\eta \\
\mathbb{E}(\neg t)\eta & = & \mathbb{E}\mathcal{D} \setminus \mathbb{E}(t)\eta \\
\mathbb{E}(\mathbb{0})\eta & = & \emptyset
\end{array}$$

Since arrow types behave as function spaces under all possible semantics assignments, that is, for all s_1, s_2, t_1, t_2 and for all η, we have:

$$[\![s_1 \to s_2]\!]\eta \subseteq [\![t_1 \to t_2]\!]\eta \iff \mathbb{E}(s_1 \to s_2)\eta \subseteq \mathbb{E}(t_1 \to t_2)\eta$$

the definition of set-theoretic model is:

Definition 4.2.7 (Models). *A set-theoretic interpretation $[\![\_]\!] : \mathcal{T} \to \mathscr{P}(\mathcal{D})^{\mathcal{V}} \to \mathscr{P}(\mathcal{D})$ is a* model *if it induces the same inclusion relation as its associated extensional interpretation, that is, it salsifies:*

$$\forall t \in \mathcal{T}.\, \forall \eta \in \mathscr{P}(\mathcal{D})^{\mathcal{V}}.\, [\![t]\!]\eta = \emptyset \iff \mathbb{E}(t)\eta = \emptyset$$

From the condition of a model, we can deduce that $[\![\_]\!]$ induces the same *subtyping* relation as its associated extensional interpretation, that is,

$$\forall \eta \in \mathscr{P}(\mathcal{D})^{\mathcal{V}}.\, [\![t]\!]\eta = \emptyset \iff \forall \eta \in \mathscr{P}(\mathcal{D})^{\mathcal{V}}.\, \mathbb{E}(t)\eta = \emptyset$$

Definition 4.2.8 (Convexity, foundation). *Let* $[\![\_]\!] : \mathcal{T} \to \mathscr{P}(\mathcal{D})^{\mathcal{V}} \to \mathscr{P}(\mathcal{D})$ *be a set-theoretic interpretation. It is*

1. convex *if for all finite choices of types* $t_1, ..., t_n$*, it satisfies*

$$\forall\eta.([\![t_1]\!]\eta{=}\varnothing \text{ or } \cdots \text{ or } [\![t_n]\!]\eta{=}\varnothing) \iff (\forall\eta.[\![t_1]\!]\eta{=}\varnothing) \text{ or } \cdots \text{ or } (\forall\eta.[\![t_n]\!]\eta{=}\varnothing)$$

2. structural *if* $\mathcal{D}^2 \subseteq \mathcal{D}$, $[\![t_1{\times}t_2]\!]\eta{=}[\![t_1]\!]\eta{\times}[\![t_2]\!]\eta$ *and the relation on* $\mathcal{D}$ *induced by* $(d_1, d_2) \blacktriangleright d_i$ *is Noetherian.*

Definition 4.2.9. *Let* $[\![\_]\!] : \mathcal{T} \to \mathscr{P}(\mathcal{D})^{\mathcal{V}} \to \mathscr{P}(\mathcal{D})$ *be a model. It is*

1. convex *if its set-theoretic interpretation is convex;*
2. well-founded *if it induces the same inclusion relation as a structural set-theoretic interpretation.*

From now on we consider only *well-founded convex* models. We already explained the necessity of the notion of convexity we introduced in Section 3.3. The notion of well-founded model was introduced in Section 4.3 of [FCB08]. Intuitively, well-founded models are models that contain only values that are finite (*e.g.*, in a well-founded model the type $\mu x.(x{\times}x)$—*i.e.*, the type that "should" contain all and only infinite binary trees—is empty). This fits the practical motivations of this work, since XML documents—*i.e.*, values—are *finite* trees.

4.3 Properties of the subtyping relation

Types are essentially a propositional logic (with standard logical connectives: $\wedge, \vee, \neg$) whose atoms are $\mathbb{0}$, $\mathbb{1}$, basic types, product types, arrow types, and type variables. We write $\mathscr{A}_{\mathsf{fun}}$ for atoms of the form $t_1 \to t_2$, $\mathscr{A}_{\mathsf{prod}}$ for atoms of the form $t_1 \times t_2$, $\mathscr{A}_{\mathsf{basic}}$ for basic types, and $\mathscr{A}$ for $\mathscr{A}_{\mathsf{fun}} \cup \mathscr{A}_{\mathsf{prod}} \cup \mathscr{A}_{\mathsf{basic}}$. Therefore $\mathcal{V} \cup \mathscr{A} \cup \{\mathbb{0}, \mathbb{1}\}$ denotes the set of all atoms, which are ranged over by $\boldsymbol{a}$. Henceforth, we will disregard the atoms $\mathbb{0}$ and $\mathbb{1}$ since they can be straightforwardly eliminated during the algorithmic treatment of subtyping.

Definition 4.3.1 (Normal form). *A (disjunctive)* normal form τ *is a finite set of pairs of finite sets of atoms, that is, an element of* $\mathscr{P}_f(\mathscr{P}_f(\mathscr{A}{\cup}\mathcal{V}) \times \mathscr{P}_f(\mathscr{A}{\cup}\mathcal{V}))$*, where* $\mathscr{P}_f(.)$ *denotes the finite powerset. Moreover, we call an element of* $\mathscr{P}_f(\mathscr{A}{\cup}\mathcal{V}){\times}\mathscr{P}_f(\mathscr{A}{\cup}\mathcal{V})$ *a* single normal form*. If* $[\![\_]\!] : \mathcal{T} \to \mathscr{P}(\mathcal{D})^{\mathcal{V}} \to \mathscr{P}(\mathcal{D})$ *is an arbitrary set-theoretic interpretation,* τ *a normal form and* η *an assignment, we define* $[\![\tau]\!]\eta$ *as:*

$$[\![\tau]\!]\eta = \bigcup_{(P,N)\in\tau} \bigcap_{\boldsymbol{a}\in P} [\![\boldsymbol{a}]\!]\eta \cap \bigcap_{\boldsymbol{a}\in N} (\mathcal{D} \setminus [\![\boldsymbol{a}]\!]\eta)$$

(with the convention that an intersection over an empty set is taken to be $\mathcal{D}$*).*

Lemma 4.3.2. *For every type* $t \in \mathcal{T}$*, it is possible to compute a normal form* $\mathsf{dnf}(t)$ *such that for every set-theoretic interpretation* $[\![\_]\!]$ *and assignment* η*,* $[\![t]\!]\eta = [\![\mathsf{dnf}(t)]\!]\eta$*.*

Proof. We can define two functions dnf and dnf_0, both from $\mathcal{T}$ to $\mathscr{P}_f(\mathscr{P}_f(\mathscr{A} \cup \mathcal{V}) \times \mathscr{P}_f(\mathscr{A} \cup \mathcal{V}))$, by mutual induction over types:

$$\begin{aligned}
\mathsf{dnf}(\mathbb{0}) &= \emptyset \\
\mathsf{dnf}(\boldsymbol{a}) &= \{(\{\boldsymbol{a}\}, \emptyset)\} && \text{for } \boldsymbol{a} \in \mathscr{A} \cup \mathcal{V} \\
\mathsf{dnf}(t_1 \vee t_2) &= \mathsf{dnf}(t_1) \cup \mathsf{dnf}(t_2) \\
\mathsf{dnf}(\neg t) &= \mathsf{dnf}_0(t)
\end{aligned}$$

$$\begin{aligned}
\mathsf{dnf}_0(\mathbb{0}) &= \{(\emptyset, \emptyset)\} \\
\mathsf{dnf}_0(\boldsymbol{a}) &= \{(\emptyset, \{\boldsymbol{a}\})\} && \text{for } \boldsymbol{a} \in \mathscr{A} \cup \mathcal{V} \\
\mathsf{dnf}_0(t_1 \vee t_2) &= \{(P_1 \cup P_2, N_1 \cup N_2) \mid (P_i, N_i) \in \mathsf{dnf}_0(t_i), i = 1, 2\} \\
\mathsf{dnf}_0(\neg t) &= \mathsf{dnf}(t)
\end{aligned}$$

Then we check the following property by induction over the type t:

$$[\![t]\!]\eta = [\![\mathsf{dnf}(t)]\!]\eta = \mathcal{D} \setminus [\![\mathsf{dnf}_0(t)]\!]\eta$$

□

For instance, consider the type $t = \boldsymbol{a}_1 \wedge (\boldsymbol{a}_3 \vee \neg \boldsymbol{a}_2)$ where $\boldsymbol{a}_1$, $\boldsymbol{a}_2$, and $\boldsymbol{a}_3$ are any atoms. Then $\mathsf{dnf}(t) = \{(\{\boldsymbol{a}_1, \boldsymbol{a}_3\}, \emptyset), (\{\boldsymbol{a}_1\}, \{\boldsymbol{a}_2\})\}$. This corresponds to the fact that for every set-theoretic interpretation and semantic assignment, $\mathsf{dnf}(t)$, t, and $(\boldsymbol{a}_1 \wedge \boldsymbol{a}_3) \vee (\boldsymbol{a}_1 \wedge \neg \boldsymbol{a}_2)$ have the same denotation.

Note that the converse result is true as well: for any normal form τ, we can find a type t such that for every set-theoretic interpretation $[\![\_]\!]$ and semantic assignment η, $[\![t]\!]\eta = [\![\tau]\!]\eta$. Normal forms are thus simply a different, but handy, syntax for types. In particular, we can rephrase in Definition 4.2.7 the condition for a set-theoretic interpretation to be a model as:

$$\forall \tau \,.\, \forall \eta \in \mathscr{P}(\mathcal{D})^{\mathcal{V}} \,.\, [\![\tau]\!]\eta = \emptyset \iff \mathbb{E}(\tau)\eta = \emptyset.$$

For these reasons henceforth we will often confound the notions of types and normal forms, and often speak of the "type" τ, taking the latter as a canonical representation of all the types in $\mathsf{dnf}^{-1}(\tau)$.

Moreover, the definition of substitution application is naturally extended to normal forms by applying the substitution to each type in the sets that form the normal form.

Let $[\![\_]\!]$ be a set-theoretic interpretation. Given a normal form τ, we are interested in checking the assertion $\forall \eta \in \mathscr{P}(\mathcal{D})^{\mathcal{V}}. [\![\tau]\!]\eta = \emptyset$ or equivalently $\forall \eta \in \mathscr{P}(\mathcal{D})^{\mathcal{V}}. \mathbb{E}(\tau)\eta = \emptyset$. Clearly, the equation $\forall \eta \in \mathscr{P}(\mathcal{D})^{\mathcal{V}}. \mathbb{E}(\tau)\eta = \emptyset$ is equivalent to:

$$\forall \eta \in \mathscr{P}(\mathcal{D})^{\mathcal{V}} \,.\, \forall (P, N) \in \tau \,.\, \bigcap_{\boldsymbol{a} \in P} \mathbb{E}(\boldsymbol{a})\eta \subseteq \bigcup_{\boldsymbol{a} \in N} \mathbb{E}(\boldsymbol{a})\eta \tag{4.1}$$

Let us define $\mathbb{E}^{\mathsf{basic}}\mathcal{D} \stackrel{\text{def}}{=} \mathcal{D}$, $\mathbb{E}^{\mathsf{prod}}\mathcal{D} \stackrel{\text{def}}{=} \mathcal{D}^2$ and $\mathbb{E}^{\mathsf{fun}}\mathcal{D} \stackrel{\text{def}}{=} \mathscr{P}(\mathcal{D} \times \mathcal{D}_\Omega)$. Then we have $\mathbb{E}\mathcal{D} = \bigcup_{u \in U} \mathbb{E}^u\mathcal{D}$ where $U = \{\mathsf{basic}, \mathsf{prod}, \mathsf{fun}\}$. Thus we can rewrite equation (4.1) as:

$$\forall \eta \in \mathscr{P}(\mathcal{D})^{\mathcal{V}} \,.\, \forall u \in U \,.\, \forall (P, N) \in \tau \,.\, \bigcap_{\boldsymbol{a} \in P} (\mathbb{E}(\boldsymbol{a})\eta \cap \mathbb{E}^u\mathcal{D}) \subseteq \bigcup_{\boldsymbol{a} \in N} (\mathbb{E}(\boldsymbol{a})\eta \cap \mathbb{E}^u\mathcal{D}) \tag{4.2}$$

For an atom $\boldsymbol{a} \in \mathscr{A}$, we have $\mathbb{E}(\boldsymbol{a})\eta \cap \mathbb{E}^u\mathcal{D} = \emptyset$ if $\boldsymbol{a} \notin \mathscr{A}_u$ and $\mathbb{E}(\boldsymbol{a})\eta \cap \mathbb{E}^u\mathcal{D} = \mathbb{E}(\boldsymbol{a})\eta$ if $\boldsymbol{a} \in \mathscr{A}_u$. Then we can rewrite equation (4.2) as:

$$\begin{aligned}&\forall \eta \in \mathscr{P}(\mathcal{D})^{\mathcal{V}}.\ \forall u \in U.\ \forall (P,N) \in \tau.\ (P \subseteq \mathscr{A}_u \cup \mathcal{V}) \Rightarrow \\ &\bigcap_{\boldsymbol{a} \in P \cap \mathscr{A}_u} \mathbb{E}(\boldsymbol{a})\eta \cap \bigcap_{\alpha \in P \cap \mathcal{V}} (\eta(\alpha) \cap \mathbb{E}^u\mathcal{D}) \subseteq \bigcup_{\boldsymbol{a} \in N \cap \mathscr{A}_u} \mathbb{E}(\boldsymbol{a})\eta \cup \bigcup_{\alpha \in N \cap \mathcal{V}} (\eta(\alpha) \cap \mathbb{E}^u\mathcal{D})\end{aligned} \tag{4.3}$$

(where the intersection is taken to be $\mathbb{E}\mathcal{D}$ when $P = \emptyset$). Furthermore, if the same variable occurs both in P and in N, then (4.3) is trivially satisfied. So we can suppose that $P \cap N \cap \mathcal{V} = \emptyset$. This justifies the simplifications made in *Step 3* of the subtyping algorithm described in Section 3.4, that is the simplification of mixed single normal forms.

Step 4, the elimination of negated toplevel variables, is justified by the following lemma:

Lemma 4.3.3. *Let P, N be two finite subsets of atoms and α an arbitrary type variable, then we have*

$$\begin{aligned}&\forall \eta \in \mathscr{P}(\mathcal{D})^{\mathcal{V}}\ .\ \textstyle\bigcap_{\boldsymbol{a} \in P} \mathbb{E}(\boldsymbol{a})\eta \subseteq \bigcup_{\boldsymbol{a} \in N} \mathbb{E}(\boldsymbol{a})\eta \cup \eta(\alpha) \iff \\ &\qquad \forall \eta \in \mathscr{P}(\mathcal{D})^{\mathcal{V}}\ .\ \textstyle\bigcap_{\boldsymbol{a} \in P} \mathbb{E}(\sigma_\alpha^{\neg\beta}(\boldsymbol{a}))\eta \cap \eta(\beta) \subseteq \bigcup_{\boldsymbol{a} \in N} \mathbb{E}(\sigma_\alpha^{\neg\beta}(\boldsymbol{a}))\eta\end{aligned}$$

where β is a fresh variable and $\sigma_\alpha^{\neg\beta}(\boldsymbol{a}) = \boldsymbol{a}\{{}^{\neg\beta}/_{\alpha}\}$

Proof. Straightforward application of set theory. □

Note that Lemma 4.3.3 only deals with one type variable, but it is trivial to generalize this lemma to multiple type variables (the same holds for Lemmas 4.3.7 and 4.3.8).

Using Lemma 4.3.3, we can rewrite equation (4.3) as:

$$\begin{aligned}&\forall \eta \in \mathscr{P}(\mathcal{D})^{\mathcal{V}}\ .\ \forall u \in U\ .\ \forall (P,N) \in \tau\ .\ (P \subseteq \mathscr{A}_u \cup \mathcal{V}) \Rightarrow \\ &\qquad \bigcap_{\boldsymbol{a} \in P \cap \mathscr{A}_u} \mathbb{E}(\boldsymbol{a})\eta \cap \bigcap_{\alpha \in P \cap \mathcal{V}} (\eta(\alpha) \cap \mathbb{E}^u\mathcal{D}) \subseteq \bigcup_{\boldsymbol{a} \in N \cap \mathscr{A}_u} \mathbb{E}(\boldsymbol{a})\eta\end{aligned} \tag{4.4}$$

since we can assume $N \cap \mathcal{V} = \emptyset$.

Next, we justify *Step 5* of the algorithm, that is the elimination of toplevel variables. In (4.4) this corresponds to eliminating the variables in $P \cap \mathcal{V}$. When $u = \mathsf{basic}$ this can be easily done since all variables (which can appear only at top-level) can be straightforwardly removed. Indeed, notice that if s and t are closed types then $s \wedge \alpha \leq t$ if and only if $s \leq t$. Since unions and intersections of basic types are closed, then we have the following lemma

Lemma 4.3.4. *Let P, N be two finite subsets of $\mathscr{A}_{\mathsf{basic}}$, X a finite set of variables. Then*

$$\forall \eta \in \mathscr{P}(\mathcal{D})^{\mathcal{V}}\ .\ \bigcap_{b \in P} \mathbb{E}(b)\eta \cap \bigcap_{\alpha \in X} (\eta(\alpha) \cap \mathscr{C}) \subseteq \bigcup_{b \in N} \mathbb{E}(b)\eta \iff \bigcap_{b \in P} \mathbb{B}(b) \subseteq \bigcup_{b \in N} \mathbb{B}(b)$$

(with the convention $\bigcap_{\boldsymbol{a} \in \emptyset} \mathbb{E}(\boldsymbol{a})\eta = \mathscr{C}$)

Proof. Straightforwardly. □

The justification of *Step 5* for $u = \mathsf{prod}$ is given by Lemma 4.3.7. In order to prove it we need to prove a substitution lemma for the extensional interpretation and a substitution lemma for the set-theoretic interpretation.

Lemma 4.3.5. *Let $[\![\_]\!]$ be a model and $\mathbb{E}(\_)$ its associated extensional interpretation. For all types t, t', variable α, and assignment η, if $\mathbb{E}(t')\eta = \eta(\alpha)$, then $\mathbb{E}(t\{t'/\alpha\})\eta = \mathbb{E}(t)\eta$.*

Proof. Let P be a monadic proposition defined as:

$$P(s) \iff^{\text{def}} \mathbb{E}(s\{t'/\alpha\})\eta = \mathbb{E}(s)\eta$$

To prove this lemma, we construct an inference system as follows:

(P-BASIC) $\vdash_P b$	(P-EMPTY) $\vdash_P \mathbb{0}$
(P-ANY) $\vdash_P \mathbb{1}$	(P-VAR) $\vdash_P \beta$
(P-NOT) $s \vdash_P \neg s$	(P-AND) $s_1, s_2 \vdash_P s_1 \wedge s_2$
(P-OR) $s_1, s_2 \vdash_P s_1 \vee s_2$	(P-PROD) $s_1, s_2 \vdash_P s_1 \times s_2$
(P-ARR) $s_1, s_2 \vdash_P s_1 \to s_2$	

where $s_1, \ldots, s_n \vdash_P s$ means that if $P(s_1)$, …, $P(s_n)$ hold, then $P(s)$ holds.
First we prove the inference system is correct by case analysis, that is, that each rule holds.

P-BASIC, P-EMPTY, P-ANY: Straightforward.

P-VAR: If $\beta \neq \alpha$, it is straightforward. Otherwise, from the hypothesis we have $\mathbb{E}(t\{t'/\alpha\})\eta = \mathbb{E}(t')\eta = \eta(\alpha)$. Thus the rule holds.

P-AND: We have $\mathbb{E}(s_i\{t'/\alpha\})\eta = \mathbb{E}(s_i)\eta$ for $i = 1, 2$. Then

$$\begin{aligned}\mathbb{E}((s_1 \wedge s_2)\{t'/\alpha\})\eta &= \mathbb{E}(s_1\{t'/\alpha\})\eta \cap \mathbb{E}(s_2\{t'/\alpha\})\eta \\ &= \mathbb{E}(s_1)\eta \cap \mathbb{E}(s_2)\eta \\ &= \mathbb{E}(s_1 \wedge s_2)\eta\end{aligned}$$

Thus the rule holds.

P-NOT,P-OR: Similar to P-AND.

P-PROD: We have $\mathbb{E}(s_i\{t'/\alpha\})\eta = \mathbb{E}(s_i)\eta$ for $i = 1, 2$, but what we need is that $[\![s_i\{t'/\alpha\}]\!]\eta = [\![s_i]\!]\eta$. Notice however that from $\mathbb{E}(s_i\{t'/\alpha\})\eta = \mathbb{E}(s_i)\eta$ we can deduce both $\mathbb{E}(s_i\{t'/\alpha\})\eta \setminus \mathbb{E}(s_i)\eta \subseteq \emptyset$ and $\mathbb{E}(s_i)\eta \setminus \mathbb{E}(s_i\{t'/\alpha\})\eta \subseteq \emptyset$. That is, by definition of $\mathbb{E}()$:

$$\mathbb{E}(s_i\{t'/\alpha\} \setminus s_i)\eta \subseteq \emptyset \text{ and } \mathbb{E}(s_i \setminus s_i\{t'/\alpha\})\eta \subseteq \emptyset$$

Since $[\![\_]\!]$ is a model, then the zero of the interpretation and of the extensional interpretation coincide, and thus we have

$$[\![s_i\{t'/\alpha\} \setminus s_i]\!]\eta \subseteq \emptyset \text{ and } [\![s_i \setminus s_i\{t'/\alpha\}]\!]\eta \subseteq \emptyset$$

That is $[\![s_i\{^{t'}/_{\alpha}\}]\!]\eta = [\![s_i]\!]\eta$. Therefore,

$$\begin{array}{rcl}\mathbb{E}((s_1 \times s_2)\{^{t'}/_{\alpha}\})\eta = & [\![s_1\{^{t'}/_{\alpha}\}]\!]\eta \times [\![s_2\{^{t'}/_{\alpha}\}]\!]\eta \\ = & [\![s_1]\!]\eta \times [\![s_2]\!]\eta \\ = & \mathbb{E}(s_1 \times s_2)\eta\end{array}$$

Thus the rule holds.

P-ARR: Similar to P-PROD.

Then based on the inference system we define an operator $F_P : \mathscr{P}(\mathcal{T}) \to \mathscr{P}(\mathcal{T})$ as

$$F_P(S) = \{s \in \mathcal{T} \mid \exists s_1, \ldots, s_n \in S \;.\; s_1, \ldots, s_n \vdash_P s\}$$

In other terms, $F_P(S)$ is the set of types that can be inferred to satisfy P in one step from the types satisfying P in S by the inference system above. Clearly, the inference operator F_P is monotone: $F_P(S_1) \subseteq F_P(S_2)$ if $S_1 \subseteq S_2$ and $(\mathscr{P}(\mathcal{T}), \subseteq)$ is cpo. By Tarski's fix point theorem [Tar55], the inference operator possesses both a least fixed point and a greatest fixed point. Clearly the set of all types that satisfy the property P is the greatest fix-point $gfp(F_P)$ of F_P. To prove our result we must prove that the set of types $\mathcal{T}$ is contained in $gfp(F_P)$. It is easy to prove that $\mathcal{T} \subseteq F_P(\mathcal{T})$: simply take any type $t \in \mathcal{T}$ and prove by cases that it belongs to $F_P(\mathcal{T})$. This means that $\mathcal{T}$ is F_P-consistent. By the principle of coinduction we deduce that $\mathcal{T} \subseteq gfp(F_P)$ and we conclude that $gfp(F_P) = \mathcal{T}$, namely, that all types satisfy P. Therefore, the result follows. □

Lemma 4.3.6. *Let $[\![\_]\!]$ be a set-theoretic interpretation. For all type t, substitution σ, and assignments η, η', if $\forall \alpha \in \mathsf{var}(t)$. $\eta'(\alpha) = [\![\sigma(\alpha)]\!]\eta$, then $[\![t]\!]\eta' = [\![t\sigma]\!]\eta$.*

Proof. Similar to the proof of Lemma 4.3.5: first we define a monadic proposition $P(t)$ (*i.e.*, $[\![t]\!]\eta' = [\![t\sigma]\!]\eta$) and an inference system, then we construct an operator F_P, finally we prove the set of types $\mathcal{T}$ is contained in the greatest fix-point $gfp(F_P)$ of F_P.
The correctness of the arrow rule P-ARR is as follows:

P-ARR : We have $[\![s_1]\!]\eta' = [\![s_1\sigma]\!]\eta$ and $[\![s_2]\!]\eta' = [\![s_2\sigma]\!]\eta$. Take any element $d \in [\![s_1 \to s_2]\!]\eta'$. d must be $\{(d_i, d_i') \mid i \in I\}$ and for each $i \in I$ if $d_i \in [\![s_1]\!]\eta'$ then $d_i' \in [\![s_2]\!]\eta'$, where i is a possibly empty or infinite set[1]. Consider each $i \in I$. If $d_i \in [\![s_1\sigma]\!]\eta$, then we have

$$\begin{array}{rll} & d_i \in [\![s_1\sigma]\!]\eta & \\ \Rightarrow & d_i \in [\![s_1]\!]\eta' & ([\![s_1]\!]\eta' = [\![s_1\sigma]\!]\eta) \\ \Rightarrow & d_i' \in [\![s_2]\!]\eta' & ((d_i, d_i') \in d, d \in [\![s_1 \to s_2]\!]\eta') \\ \Rightarrow & d_i' \in [\![s_2\sigma]\!]\eta & ([\![s_2]\!]\eta' = [\![s_2\sigma]\!]\eta)\end{array}$$

Thus, $d \in [\![s_1\sigma \to s_2\sigma]\!]\eta$, that is, $d \in [\![(s_1 \to s_2)\sigma]\!]\eta$. Therefore,

$$[\![s_1 \to s_2]\!]\eta' \subseteq [\![(s_1 \to s_2)\sigma]\!]\eta.$$

Similarly, we can prove that $[\![(s_1 \to s_2)\sigma]\!]\eta \subseteq [\![s_1 \to s_2]\!]\eta'$. Thus the rule holds.

[1]Indeed, we do not give an explicit definition of $[\![s_1 \to s_2]\!]\eta$, but it must be a function on $[\![s_1]\!]\eta$ and $[\![s_2]\!]\eta$, based on which the correctness of P-ARR can be proved as well.

□

Lemma 4.3.7. *Let $(\mathcal{D}, [\![\_]\!])$ be a model, P, N two finite subsets of $\mathscr{A}_{\mathsf{prod}}$ and α an arbitrary type variable.*

$$\forall \eta \in \mathscr{P}(\mathcal{D})^{\mathcal{V}} \,.\, \bigcap_{\boldsymbol{a} \in P} \mathbb{E}(\boldsymbol{a})\eta \cap (\eta(\alpha) \cap \mathbb{E}^{\mathsf{prod}}\mathcal{D}) \subseteq \bigcup_{\boldsymbol{a} \in N} \mathbb{E}(\boldsymbol{a})\eta \iff$$
$$\forall \eta \in \mathscr{P}(\mathcal{D})^{\mathcal{V}} \,.\, \bigcap_{\boldsymbol{a} \in P} \mathbb{E}(\sigma^{\times}_{\alpha}(\boldsymbol{a}))\eta \cap \mathbb{E}(\alpha_1 \times \alpha_2)\eta \subseteq \bigcup_{\boldsymbol{a} \in N} \mathbb{E}(\sigma^{\times}_{\alpha}(\boldsymbol{a}))\eta$$

where $\sigma^{\times}_{\alpha}(\boldsymbol{a}) = \boldsymbol{a}\{(\alpha_1 \times \alpha_2) \vee \alpha/\alpha\}$ and α_1, α_2 are fresh variables.

Proof. "$\Leftarrow$" direction: consider a generic assignment η, and suppose we have $\eta(\alpha) \cap \mathbb{E}^{\mathsf{prod}}\mathcal{D} = \bigcup_{i \in I}(S^i_1 \times S^i_2)$ where S^i_j are subsets of $\mathcal{D}$ and I may be infinite (notice that every intersection with $\mathcal{D}^2$ can be expressed as an infinite union of products: at the limit singleton products can be used). If $|I| = 0$, that is, $\eta(\alpha) \cap \mathbb{E}^{\mathsf{prod}}\mathcal{D} = \emptyset$, clearly, we have

$$\bigcap_{\boldsymbol{a} \in P} \mathbb{E}(\boldsymbol{a})\eta \cap (\eta(\alpha) \cap \mathbb{E}^{\mathsf{prod}}\mathcal{D}) = \emptyset \subseteq \bigcup_{\boldsymbol{a} \in N} \mathbb{E}(\boldsymbol{a})\eta$$

Assume that $|I| > 0$. Then for each $(S^i_1 \times S^i_2)$, we construct another assignment η^i defined as $\eta^i = \eta \oplus \{S^i_1/\alpha_1, S^i_2/\alpha_2\}$, where $\oplus$ denotes both function extension and redefinition. Then, we have

$$\begin{array}{lll}
\mathbb{E}((\alpha_1 \times \alpha_2) \vee \alpha)\eta^i & = (\eta^i(\alpha_1) \times \eta^i(\alpha_2)) \vee \eta^i(\alpha) & \text{by definition of } \mathbb{E}(_) \\
& = (S^i_1 \times S^i_2) \vee \eta(\alpha) & \text{by definition of } \eta^i \\
& = \eta(\alpha) & \text{since } \bigcup_{i \in I}(S^i_1 \times S^i_2) \subseteq \eta(\alpha) \\
& = \eta^i(\alpha) & \text{by definition of } \eta^i
\end{array}$$

We can thus apply Lemma 4.3.5 and for any type t deduce that

$$\mathbb{E}(t\{(\alpha_1 \times \alpha_2) \vee \alpha/\alpha\})\eta^i = \mathbb{E}(t)\eta^i$$

In particular, for all $\boldsymbol{a} \in P \cup N$ we have

$$\mathbb{E}(\sigma^{\times}_{\alpha}(\boldsymbol{a}))\eta^i \stackrel{\text{def}}{=} \mathbb{E}(\boldsymbol{a}\{(\alpha_1 \times \alpha_2 \vee \alpha/\alpha\})\eta^i = \mathbb{E}(\boldsymbol{a})\eta^i$$

Now, notice that η and η^i differ only for the interpretation of α_1 and α_2. Since these variables are fresh, then they do not belong to $\mathsf{var}(\boldsymbol{a})$, and therefore $\mathbb{E}(\boldsymbol{a})\eta^i = \mathbb{E}(\boldsymbol{a})\eta$. This allows us to conclude that

$$\forall \boldsymbol{a} \in (P \cup N) \,.\, \mathbb{E}(\sigma^{\times}_{\alpha}(\boldsymbol{a}))\eta^i = \mathbb{E}(\boldsymbol{a})\eta$$

Therefore,

$$\begin{aligned}
&\forall i{\in}I(\bigcap_{\boldsymbol{a}\in P}\mathbb{E}(\sigma^{\times}_{\alpha}(\boldsymbol{a}))\eta^i\cap\mathbb{E}(\alpha_1{\times}\alpha_2)\eta^i\subseteq\bigcup_{\boldsymbol{a}\in N}\mathbb{E}(\sigma^{\times}_{\alpha}(\boldsymbol{a}))\eta^i)\\
&\quad\Rightarrow\forall i{\in}I(\bigcap_{\boldsymbol{a}\in P}\mathbb{E}(\sigma^{\times}_{\alpha}(\boldsymbol{a}))\eta^i\cap\mathbb{E}(\alpha_1{\times}\alpha_2)\eta^i\cap\bigcap_{\boldsymbol{a}\in N}\overline{\mathbb{E}(\sigma^{\times}_{\alpha}(\boldsymbol{a}))\eta^i}\subseteq\emptyset)\\
&\quad\Rightarrow\bigcup_{i\in I}\left((\bigcap_{\boldsymbol{a}\in P}\mathbb{E}(\sigma^{\times}_{\alpha}(\boldsymbol{a}))\eta^i\cap\mathbb{E}(\alpha_1{\times}\alpha_2)\eta^i\cap\bigcap_{\boldsymbol{a}\in N}\overline{\mathbb{E}(\sigma^{\times}_{\alpha}(\boldsymbol{a}))\eta^i})\right)\subseteq\emptyset\\
&\quad\Rightarrow\bigcup_{i\in I}\left((\bigcap_{\boldsymbol{a}\in P}\mathbb{E}(\boldsymbol{a})\eta\cap\mathbb{E}(\alpha_1{\times}\alpha_2)\eta^i\cap\bigcap_{\boldsymbol{a}\in N}\overline{\mathbb{E}(\boldsymbol{a})\eta})\right)\subseteq\emptyset\\
&\quad\Rightarrow\bigcap_{\boldsymbol{a}\in P}\mathbb{E}(\boldsymbol{a})\eta\cap(\bigcup_{i\in I}\mathbb{E}(\alpha_1{\times}\alpha_2)\eta^i)\cap\bigcap_{\boldsymbol{a}\in N}\overline{\mathbb{E}(\boldsymbol{a})\eta}\subseteq\emptyset\\
&\quad\Rightarrow\bigcap_{\boldsymbol{a}\in P}\mathbb{E}(\boldsymbol{a})\eta\cap(\bigcup_{i\in I}(S^i_1{\times}S^i_2))\cap\bigcap_{\boldsymbol{a}\in N}\overline{\mathbb{E}(\boldsymbol{a})\eta}\subseteq\emptyset\\
&\quad\Rightarrow\bigcap_{\boldsymbol{a}\in P}\mathbb{E}(\boldsymbol{a})\eta\cap(\eta(\alpha)\cap\mathbb{E}^{\mathsf{prod}}\mathcal{D})\cap\bigcap_{\boldsymbol{a}\in N}\overline{\mathbb{E}(\boldsymbol{a})\eta}\subseteq\emptyset\\
&\quad\Rightarrow\bigcap_{\boldsymbol{a}\in P}\mathbb{E}(\boldsymbol{a})\eta\cap(\eta(\alpha)\cap\mathbb{E}^{\mathsf{prod}}\mathcal{D})\subseteq\bigcup_{\boldsymbol{a}\in N}\mathbb{E}(\boldsymbol{a})\eta
\end{aligned}$$

This proves the result.

"⇒" direction: this direction is rather obvious since if a type is empty, then so is every instance of it. In particular, suppose there exists an assignment η such that

$$\bigcap_{\boldsymbol{a}\in P}\mathbb{E}(\sigma^{\times}_{\alpha}(\boldsymbol{a}))\eta\cap\mathbb{E}(\alpha_1{\times}\alpha_2)\eta\subseteq\bigcup_{\boldsymbol{a}\in N}\mathbb{E}(\sigma^{\times}_{\alpha}(\boldsymbol{a}))\eta$$

does not hold. Then, for this assignment η we have

$$\bigcap_{\boldsymbol{a}\in P}\mathbb{E}(\sigma^{\times}_{\alpha}(\boldsymbol{a}))\eta\cap(\mathbb{E}(\alpha_1{\times}\alpha_2)\eta)\not\subseteq\bigcup_{\boldsymbol{a}\in N}\mathbb{E}(\sigma^{\times}_{\alpha}(\boldsymbol{a}))\eta$$

and *a fortiori*

$$\bigcap_{\boldsymbol{a}\in P}\mathbb{E}(\sigma^{\times}_{\alpha}(\boldsymbol{a}))\eta\cap(\mathbb{E}(\alpha_1{\times}\alpha_2)\eta\cup\eta(\alpha))\not\subseteq\bigcup_{\boldsymbol{a}\in N}\mathbb{E}(\sigma^{\times}_{\alpha}(\boldsymbol{a}))\eta$$

That is,

$$\mathbb{E}(\sigma^{\times}_{\alpha}(\bigwedge_{\boldsymbol{a}\in P}\boldsymbol{a}\wedge\alpha\wedge\bigwedge_{\boldsymbol{a}\in N}\neg\boldsymbol{a}))\eta\neq\emptyset$$

Since $[\![\_]\!]$ is a model, then

$$[\![\sigma^{\times}_{\alpha}(\bigwedge_{\boldsymbol{a}\in P}\boldsymbol{a}\wedge\alpha\wedge\bigwedge_{\boldsymbol{a}\in N}\neg\boldsymbol{a})]\!]\eta\neq\emptyset$$

Let $\eta'=\eta\oplus\{[\![(\alpha_1{\times}\alpha_2){\vee}\alpha]\!]\eta/\alpha\}$. Applying Lemma 4.3.6, we have

$$[\![\bigwedge_{\boldsymbol{a}\in P}\boldsymbol{a}\wedge\alpha\wedge\bigwedge_{\boldsymbol{a}\in N}\neg\boldsymbol{a}]\!]\eta'\neq\emptyset$$

Then we have

$$\mathbb{E}(\bigwedge_{\boldsymbol{a}\in P} \boldsymbol{a} \wedge \alpha \wedge \bigwedge_{\boldsymbol{a}\in N} \neg\boldsymbol{a})\eta' \neq \emptyset$$

That is,

$$\bigcap_{\boldsymbol{a}\in P} \mathbb{E}(\boldsymbol{a})\eta' \cap (\eta'(\alpha) \cap \mathbb{E}^{\mathsf{prod}}\mathcal{D}) \subseteq \bigcup_{\boldsymbol{a}\in N} \mathbb{E}(\boldsymbol{a})\eta'$$

does not hold, which contradicts the hypothesis. □

The case for $u = \mathsf{fun}$ is trickier because $\mathbb{1} \to \mathbb{0}$ is contained in every arrow type, and therefore sets of arrows that do not contain it must be explicitly checked. If, analogously to $u = \mathsf{prod}$, we used just $\{(\alpha_1 \to \alpha_2) \vee \alpha/\alpha\}$, then we were considering only instances of α that contain $\mathbb{1} \to \mathbb{0}$, as every possible instance of $\alpha_1 \to \alpha_2$ contains it. Therefore if we just check this family of substitutions the subtypes of $\mathbb{0} \to \mathbb{1}$ that do not contain $\mathbb{1} \to \mathbb{0}$ would never be assigned to α by the algorithm and, thus, never checked. To obviate this problem $\{((\alpha_1 \to \alpha_2) \setminus (\mathbb{1} \to \mathbb{0})) \vee \alpha/\alpha\}$ must be checked, as well.

Lemma 4.3.8. *Let $(\mathcal{D}, [\![\_]\!])$ be a well-founded model, P, N two finite subsets of $\mathscr{A}_{\mathsf{fun}}$ and α an arbitrary type variable. Then*

$$\begin{aligned}
\forall \eta \in \mathscr{P}(\mathcal{D})^{\mathcal{V}}.\ \bigcap_{\boldsymbol{a}\in P} \mathbb{E}(\boldsymbol{a})\eta \cap (\eta(\alpha) \cap \mathbb{E}^{\mathsf{fun}}\mathcal{D}) \subseteq \bigcup_{\boldsymbol{a}\in N} \mathbb{E}(\boldsymbol{a})\eta \iff \\
\forall \eta \in \mathscr{P}(\mathcal{D})^{\mathcal{V}}.\ \bigcap_{\boldsymbol{a}\in P} \mathbb{E}(\sigma^{\rightarrow}_{\alpha}(\boldsymbol{a}))\eta \cap \mathbb{E}(\alpha_1 \to \alpha_2)\eta \subseteq \bigcup_{\boldsymbol{a}\in N} \mathbb{E}(\sigma^{\rightarrow}_{\alpha}(\boldsymbol{a}))\eta \quad \textit{and} \\
\forall \eta \in \mathscr{P}(\mathcal{D})^{\mathcal{V}}.\ \bigcap_{\boldsymbol{a}\in P} \mathbb{E}(\sigma^{\not\rightarrow}_{\alpha}(\boldsymbol{a}))\eta \cap \mathbb{E}((\alpha_1 \to \alpha_2) \setminus (\mathbb{1} \to \mathbb{0}))\eta \subseteq \bigcup_{\boldsymbol{a}\in N} \mathbb{E}(\sigma^{\not\rightarrow}_{\alpha}(\boldsymbol{a}))\eta
\end{aligned}$$

where $\sigma^{\rightarrow}_{\alpha}(\boldsymbol{a}) = \boldsymbol{a}\{(\alpha_1 \to \alpha_2) \vee \alpha/\alpha\}$, $\sigma^{\not\rightarrow}_{\alpha}(\boldsymbol{a}) = \boldsymbol{a}\{((\alpha_1 \to \alpha_2) \setminus (\mathbb{1} \to \mathbb{0})) \vee \alpha/\alpha\}$, and α_1, α_2 are fresh variables.

Proof. "⇐" direction: suppose that there exists an assignment η such that

$$\bigcap_{\boldsymbol{a}\in P} \mathbb{E}(\boldsymbol{a})\eta \cap (\mathbb{E}(\alpha)\eta \cap \mathbb{E}^{\mathsf{fun}}\mathcal{D}) \subseteq \bigcup_{\boldsymbol{a}\in N} \mathbb{E}(\boldsymbol{a})\eta$$

does not hold. Then for this assignment, there exists at least an element d such that

$$d \in (\bigcap_{\boldsymbol{a}\in P} \mathbb{E}(\boldsymbol{a})\eta \setminus \bigcup_{\boldsymbol{a}\in N} \mathbb{E}(\boldsymbol{a})\eta) \cap (\mathbb{E}(\alpha)\eta \cap \mathbb{E}^{\mathsf{fun}}\mathcal{D})$$

If one of these elements d is such that $d \in \mathbb{E}(\mathbb{1} \to \mathbb{0})\eta$, then $|N| = 0$: indeed since $\mathbb{1}\to\mathbb{0}$ is contained in every arrow type, then subtracting any arrow type (*i.e.*, $|N| > 0$) would remove all the elements of $\mathbb{1}\to\mathbb{0}$. Clearly, we have

$$d \in (\bigcap_{\boldsymbol{a}\in P} \mathbb{E}(\sigma^{\rightarrow}_{\alpha}(\boldsymbol{a}))\eta \setminus \bigcup_{\boldsymbol{a}\in N} \mathbb{E}(\sigma^{\rightarrow}_{\alpha}(\boldsymbol{a}))\eta) \cap \mathbb{E}(\alpha_1 \to \alpha_2)\eta$$

Indeed since $|N| = 0$ the set above is an intersection of arrow types, and since they all contain $\mathbb{1}\to\mathbb{0}$, they all contain d as well. This contradicts the premise, therefore, the result follows.

Otherwise, assume that $|N| > 0$ and therefore $d \notin \mathbb{E}(\mathbb{1} \to \mathbb{0})\eta$. Since $d \in (\bigcap_{\boldsymbol{a} \in P} \mathbb{E}(\boldsymbol{a})\eta \setminus \bigcup_{\boldsymbol{a} \in N} \mathbb{E}(\boldsymbol{a})\eta)$ then we have

$$\bigcap_{\boldsymbol{a} \in P} \mathbb{E}(\boldsymbol{a})\eta \not\subseteq \bigcup_{\boldsymbol{a} \in N} \mathbb{E}(\boldsymbol{a})\eta.$$

Let η_1 be an assignment defined as $\eta_1 = \eta \oplus \{\mathcal{D}/\alpha_1, \emptyset/\alpha_2\}$. Then, we have $\mathbb{E}(\alpha_1 \to \alpha_2)\eta_1 = \mathbb{E}(\mathbb{1} \to \mathbb{0})\eta_1$. Therefore:

$$\begin{array}{lll} \mathbb{E}(\sigma^{\rightsquigarrow}_{\alpha}(\alpha))\eta_1 & = (\mathbb{E}(\alpha_1 \to \alpha_2)\eta_1 \setminus \mathbb{E}(\mathbb{1} \to \mathbb{0})\eta_1) \cup \eta_1(\alpha) & \text{(by definition of } \mathbb{E}(\_)) \\ & = \eta_1(\alpha) & \text{(by definition of } \eta_1) \end{array}$$

We thus apply Lemma 4.3.5 and for any $\boldsymbol{a} \in P \cup N$, we deduce that $\mathbb{E}(\sigma^{\rightsquigarrow}_{\alpha}(\boldsymbol{a}))\eta_1 = \mathbb{E}(\boldsymbol{a})\eta_1 = \mathbb{E}(\boldsymbol{a})\eta$. From this, we infer that

$$\bigcap_{\boldsymbol{a} \in P} \mathbb{E}(\sigma^{\rightsquigarrow}_{\alpha}(\boldsymbol{a}))\eta_1 \not\subseteq \bigcup_{\boldsymbol{a} \in N} \mathbb{E}(\sigma^{\rightsquigarrow}_{\alpha}(\boldsymbol{a}))\eta_1$$

By an application of Lemma 2.1.9, we have

$$\forall t_1^0 \to t_2^0 \in N. \exists P' \subseteq P. \begin{cases} [\![\sigma^{\rightsquigarrow}_{\alpha}(t_1^0 \setminus \bigvee_{t_1 \to t_2 \in P'} t_1)]\!]\eta_1 \neq \emptyset \\ \mathsf{and} \\ P' = P \text{ or } [\![\sigma^{\rightsquigarrow}_{\alpha}(\bigwedge_{t_1 \to t_2 \in P \setminus P'} t_2 \setminus t_2^0)]\!]\eta_1 \neq \emptyset \end{cases}$$

Thus there exist at least an element in $[\![\sigma^{\rightsquigarrow}_{\alpha}(t_1^0 \setminus \bigvee_{t_1 \to t_2 \in P'} t_1)]\!]\eta_1$ and, if $P \neq P'$, an element in $[\![\sigma^{\rightsquigarrow}_{\alpha}(\bigwedge_{t_1 \to t_2 \in P \setminus P'} t_2 \setminus t_2^0)]\!]\eta_1$. The next step is to build a new assignment η' such that $[\![\sigma^{\rightsquigarrow}_{\alpha}(t_1^0 \setminus \bigvee_{t_1 \to t_2 \in P'} t_1 \vee \alpha_1)]\!]\eta'$ contains an element and, if $P \neq P'$, $[\![\sigma^{\rightsquigarrow}_{\alpha}((\bigwedge_{t_1 \to t_2 \in P \setminus P'} t_2 \wedge \alpha_2) \setminus t_2^0)]\!]\eta'$ contains an element.
To do so we invoke the procedure explore_pos defined in the proof of Lemma 4.6.5 (this procedure was defined for the proof of that lemma, and it returns also the elements that inhabit the types, which are here useless. We do not repeat its definition here). Let $VS = \bigcup_{\boldsymbol{a} \in P \cup N} \mathsf{var}(\boldsymbol{a}) \cup \{\alpha_1, \alpha_2\}$. For each $\beta \in VS$, we construct a finite set s_β which is initialized as an empty set and appended some elements during the processing of explore_pos. Thanks to the infinite support, to build an element in $\sigma^{\rightsquigarrow}_{\alpha}(t_1^0 \setminus \bigvee_{t_1 \to t_2 \in P'} t_1 \vee \alpha_1)$ is similar to build an element in $\sigma^{\rightsquigarrow}_{\alpha}(t_1^0 \setminus \bigvee_{t_1 \to t_2 \in P'} t_1)$ (it has been proved that such an element exists). And to build an element in $\sigma^{\rightsquigarrow}_{\alpha}((\bigwedge_{t_1 \to t_2 \in P \setminus P'} t_2 \wedge \alpha_2) \setminus t_2^0)$, is to build an element in $\sigma^{\rightsquigarrow}_{\alpha}(\bigwedge_{t_1 \to t_2 \in P \setminus P'} t_2 \setminus t_2^0)$ first and then append this element to s_{α_2}. In the end, we define a new (finite) assignment as $\eta' = \{s_\beta/\beta, \ldots\}$ for $\beta \in VS$. (If any type contains infinite product types it is also possible to construct such an assignment by Lemma 4.6.6). Therefore, under the assignment η', we get

$$\forall t_1^0 \to t_2^0 \in N. \exists P' \subseteq P. \begin{cases} \begin{cases} [\![\sigma^{\rightsquigarrow}_{\alpha}(t_1^0 \setminus (\bigvee_{t_1 \to t_2 \in P'} t_1 \vee \alpha_1))]\!]\eta' \neq \emptyset \\ \mathsf{and} \\ P' = P \text{ or } [\![\sigma^{\rightsquigarrow}_{\alpha}((\bigwedge_{t_1 \to t_2 \in P \setminus P'} t_2) \setminus t_2^0)]\!]\eta' \neq \emptyset \end{cases} \\ \mathsf{or} \\ \begin{cases} [\![\sigma^{\rightsquigarrow}_{\alpha}(t_1^0 \setminus (\bigvee_{t_1 \to t_2 \in P'} t_1))]\!]\eta' \neq \emptyset \\ \mathsf{and} \\ [\![\sigma^{\rightsquigarrow}_{\alpha}((\bigwedge_{t_1 \to t_2 \in P \setminus P'} t_2 \wedge \alpha_2) \setminus t_2^0)]\!]\eta' \neq \emptyset \end{cases} \end{cases}$$

By an application of Lemma 2.1.9 again, we conclude that

$$\bigcap_{\boldsymbol{a}\in P} \mathbb{E}(\sigma_\alpha^{\rightsquigarrow}(\boldsymbol{a}))\eta' \cap \mathbb{E}(\alpha_1 \to \alpha_2)\eta' \not\subseteq \bigcup_{\boldsymbol{a}\in N} \mathbb{E}(\sigma_\alpha^{\rightsquigarrow}(\boldsymbol{a}))\eta'$$

Since $|N| > 0$, then $\mathbb{E}(\mathbb{1}\to\mathbb{0})\eta'$ is contained in $\bigcup_{\boldsymbol{a}\in N} \mathbb{E}(\sigma_\alpha^{\rightsquigarrow}(\boldsymbol{a}))\eta'$. Thus removing it from the the left hand side does not change the result, which allows us to conclude that:

$$\bigcap_{\boldsymbol{a}\in P} \mathbb{E}(\sigma_\alpha^{\rightsquigarrow}(\boldsymbol{a}))\eta' \cap \mathbb{E}((\alpha_1 \to \alpha_2) \setminus (\mathbb{1} \to \mathbb{0}))\eta' \not\subseteq \bigcup_{\boldsymbol{a}\in N} \mathbb{E}(\sigma_\alpha^{\rightsquigarrow}(\boldsymbol{a}))\eta'$$

which again contradicts the hypothesis. Therefore the result follows as well.

"⇒" direction: similar to the "⇒" direction in the proof of Lemma 4.3.7. □

It is necessary to check the substitution $\{((\alpha_1 \to \alpha_2) \setminus (\mathbb{1} \to \mathbb{0})) \vee \alpha/\alpha\}$ for the soundness of the algorithm. To see it consider the following relation (this example is due to Nils Gesbert): $((\alpha \to \beta) \wedge \alpha) \leq ((\mathbb{1} \to \mathbb{0}) \to \beta)$. If the check above were not performed, then the algorithm would return that the relation holds, while it does not. In order to see that it does not hold, consider a type $t_1 = (\mathbb{1} \to \mathbb{0}) \wedge \gamma$, where γ is some variable. Clearly, there exists an assignment η_0 such that $[\![t_1]\!]\eta_0$ is nonempty. Let η be another assignment defined as $\eta = \eta_0 \oplus \{[\![\neg t_1]\!]\eta_0/\alpha, \emptyset/\beta\}$. Next, consider a function $f \in [\![(\alpha \to \beta) \wedge (\neg\alpha \to \mathbb{1})]\!]\eta$, that is, a function that diverges (since $[\![\beta]\!]\eta = \emptyset$) on values in $[\![\alpha]\!]\eta$ (*i.e.*, $[\![\neg t_1]\!]\eta$). Take f so that it converges on the values in $[\![\neg\alpha]\!]\eta$ (*i.e.*, $[\![t_1]\!]\eta$). This implies that that $f \notin [\![\neg\alpha]\!]\eta$: indeed, assume $f \in [\![\neg\alpha]\!]\eta$, that is $f \in [\![t_1]\!]\eta_0$, then $f \in [\![\mathbb{1} \to \mathbb{0}]\!]\eta$ and, thus, f would diverge on all values: contradiction (note that $[\![t_1]\!]\eta_0 \neq \emptyset$). Therefore $f \in [\![\alpha]\!]\eta$, and then $f \in [\![(\alpha \to \beta) \wedge \alpha]\!]\eta$. Instead, by construction f does not diverge on $[\![t_1]\!]\eta_0$, which is a nonempty subset of $[\![\mathbb{1} \to \mathbb{0}]\!]\eta_0$. Therefore $f \notin [\![(\mathbb{1} \to \mathbb{0}) \to \beta]\!]\eta$.

If we remove the check of $\{((\alpha_1 \to \alpha_2) \setminus (\mathbb{1} \to \mathbb{0})) \vee \alpha/\alpha\}$ from the algorithm, then the answer of the algorithm would be positive for the relation above since it would just check that all the following four relations hold:

$$\begin{cases} \mathbb{1} \to \mathbb{0} \leq \mathbb{0} \text{ or } \beta \wedge \alpha_2 \leq \beta & (1) \\ \mathbb{1} \to \mathbb{0} \leq (\alpha\{(\alpha_1 \to \alpha_2) \vee \alpha/\alpha\}) \text{ or } \alpha_2 \leq \beta & (2) \\ \mathbb{1} \to \mathbb{0} \leq \alpha_1 \text{ or } \beta \leq \beta & (3) \\ \mathbb{1} \to \mathbb{0} \leq \alpha_1 \vee (\alpha\{(\alpha_1 \to \alpha_2) \vee \alpha/\alpha\}) & (4) \end{cases}$$

where α_1, α_2 are two fresh variables. Clearly, these four relations hold if (a superset of) $[\![\mathbb{1} \to \mathbb{0}]\!]$ is assigned to α (see the substitution $\{(\alpha_1 \to \alpha_2) \vee \alpha/\alpha\}$). However, if there exists a nonempty set $s \subseteq [\![\mathbb{1} \to \mathbb{0}]\!]\eta$ which is not assigned to α (*e.g.*, the non-empty set $[\![t_1]\!]\eta_0$ defined before), then we should substitute for the occurrences of α that are not at top-level by $((\alpha_1 \to \alpha_2) \setminus ((\mathbb{1} \to \mathbb{0}) \wedge \gamma)) \vee \alpha$ rather than by $(\alpha_1 \to \alpha_2) \vee \alpha$. Thus Case (2) and Case (4) would not hold, and so does not the whole example. Therefore, we need to consider the case that a nonempty subset of $[\![\mathbb{1} \to \mathbb{0}]\!]$ is not assigned to α, namely a strict subset of $[\![\mathbb{1} \to \mathbb{0}]\!]$ is assigned to α.

Since the interpretation $[\![\mathbb{1} \to \mathbb{0}]\!]$ is infinite, there are infinitely many strict subsets to be considered. Assume there exists a strict subset of $[\![\mathbb{1} \to \mathbb{0}]\!]$ that is assigned to α, and

there exists an occurrence of α not at top level such that once it moves up to the top level it occurs in the subtyping relation of the form in Lemma 4.3.8 (otherwise $\eta(\alpha) \cap \mathbb{E}^{\mathsf{fun}}\mathcal{D}$ would be straightforwardly ignored). Since $\mathbb{1} \to \mathbb{0}$ is contained in every arrow type, then either the strict subset of $[\![\mathbb{1} \to \mathbb{0}]\!]$ can be ignored if the occurrence is positive, or the subtyping relation does not hold if the occurrence is negative (see Case (2) above). Note that what the strict subset actually is does not matter. Therefore, we just take the empty set into consideration, that is the case of $\{((\alpha_1 \to \alpha_2) \setminus (\mathbb{1} \to \mathbb{0})) \vee \alpha/\alpha\}$. Thus, only two cases — whether $[\![\mathbb{1} \to \mathbb{0}]\!]$ is assigned to α or not — are enough for Lemma 4.3.8.

As an aside notice that both lemmas above would not hold if the variable α in their statements occurred negated at toplevel, whence the necessity of *Step 4*.

Finally, *Step 6* is justified by the two following lemmas in whose proofs the hypothesis of convexity plays a crucial role:

Lemma 4.3.9. *Let $(\mathcal{D}, [\![\_]\!])$ be a convex set-theoretic interpretation and P, N two finite subsets of $\mathscr{A}_{\mathsf{prod}}$. Then:*

$$\forall\eta.\ \bigcap_{t_1\times t_2\in P} \mathbb{E}(t_1 \times t_2)\eta \subseteq \bigcup_{t_1\times t_2\in N} \mathbb{E}(t_1 \times t_2)\eta \iff$$
$$\forall N' \subseteq N\ .\ \begin{cases} \forall\eta.[\![\bigwedge_{t_1\times t_2\in P} t_1 \wedge \bigwedge_{t_1\times t_2\in N'} \neg t_1]\!]\eta = \emptyset \\ \ \mathit{or} \\ \forall\eta.[\![\bigwedge_{t_1\times t_2\in P} t_2 \wedge \bigwedge_{t_1\times t_2\in N\setminus N'} \neg t_2]\!]\eta = \emptyset \end{cases}$$

(with the convention $\bigcap_{\boldsymbol{a}\in\emptyset} \mathbb{E}(\boldsymbol{a})\eta = \mathbb{E}^{\mathsf{prod}}\mathcal{D}$).

Proof. The Lemma above is a straightforward application of the convexity property and of Lemma 2.1.8. In particular thanks to the latter it is possible to deduce the following equivalence

$$\forall\eta.\ \bigcap_{t_1\times t_2\in P} \mathbb{E}(t_1 \times t_2)\eta \subseteq \bigcup_{t_1\times t_2\in N} \mathbb{E}(t_1 \times t_2)\eta \iff$$
$$\forall\eta.\forall N' \subseteq N\ .\ \begin{cases} [\![\bigwedge_{t_1\times t_2\in P} t_1 \wedge \bigwedge_{t_1\times t_2\in N'} \neg t_1]\!]\eta = \emptyset \\ \ \mathsf{or} \\ [\![\bigwedge_{t_1\times t_2\in P} t_2 \wedge \bigwedge_{t_1\times t_2\in N\setminus N'} \neg t_2]\!]\eta = \emptyset \end{cases}$$

The result is a straightforward application of the convexity property. □

Lemma 4.3.10. *Let $(\mathcal{D}, [\![\_]\!])$ be a convex set-theoretic interpretation and P, N be two finite subsets of $\mathscr{A}_{\mathsf{fun}}$. Then:*

$$\forall\eta.\ \bigcap_{t\to s\in P} \mathbb{E}(t\to s)\eta \subseteq \bigcup_{t\to s\in N} \mathbb{E}(t\to s)\eta \iff$$
$$\exists(t_0\to s_0)\in N\,.\forall P'\subseteq P\,.\ \begin{cases} \forall\eta.[\![t_0 \setminus (\bigvee_{t\to s\in P'} t)]\!]\eta = \emptyset \\ \ \mathit{or} \\ \begin{cases} P \neq P' \\ \ \mathit{and} \\ \forall\eta.[\![(\bigwedge_{t\to s\in P\setminus P'} s) \setminus s_0]\!]\eta = \emptyset \end{cases} \end{cases}$$

(with the convention $\bigcap_{\boldsymbol{a}\in\emptyset} \mathbb{E}(\boldsymbol{a})\eta = \mathbb{E}^{\mathsf{fun}}\mathcal{D}$*)*

Proof. Similar to previous lemma, the proof can be obtained by a straightforward application of Lemma 2.1.9 and the convexity property. □

4.4 Algorithm

The formalization of the subtyping algorithm is done via the notion of *simulation* that we borrow from [FCB08] and extend to account for type variables and type instantiation. First, it is used to define the set of instances of a type.

Definition 4.4.1 (Instances). *Given a type* $t \in \mathcal{T}$*, we define* $[t]_\approx$*, the* set of instances *of* t *as:*

$$[t]_\approx \stackrel{\text{def}}{=} \{s \mid \exists \sigma : \mathcal{V} \to \mathcal{T} \,.\, t\sigma = s\}$$

Definition 4.4.2 (Simulation). *Let* $\mathscr{S}$ *be an arbitrary set of normal forms. We define another set of normal forms* $\mathbb{E}\mathscr{S}$ *as*

$$\{\tau \mid \forall s{\in}[\tau]_\approx.\, \forall u{\in}U.\, \forall (P,N){\in}\mathsf{dnf}(s).\, (P{\subseteq}\mathscr{A}_u \Rightarrow C_u^{P,N\cap\mathscr{A}_u})\}$$

where:

$$C_{\mathsf{basic}}^{P,N} \stackrel{\text{def}}{=} \bigcap_{b\in P} \mathbb{B}(b) \subseteq \bigcup_{b\in N\cap\mathscr{A}_{\mathsf{basic}}} \mathbb{B}(b)$$

$$C_{\mathsf{prod}}^{P,N} \stackrel{\text{def}}{=} \forall N'{\subseteq}N. \begin{cases} \mathsf{dnf}(\bigwedge_{t_1\times t_2\in P} t_1 \wedge \bigwedge_{t_1\times t_2\in N'} \neg t_1) \in \mathscr{S} \\ \textit{or} \\ \mathsf{dnf}(\bigwedge_{t_1\times t_2\in P} t_2 \wedge \bigwedge_{t_1\times t_2\in N\setminus N'} \neg t_2) \in \mathscr{S} \end{cases}$$

$$C_{\mathsf{fun}}^{P,N} \stackrel{\text{def}}{=} \exists t_0 \to s_0 \in N \,.\, \forall P' \subseteq P \,.\, \begin{cases} \mathsf{dnf}(t_0 \wedge \bigwedge_{t\to s\in P'} \neg t) \in \mathscr{S} \\ \textit{or} \\ \begin{cases} P \neq P' \\ \textit{and} \\ \mathsf{dnf}((\neg s_0) \wedge \bigwedge_{t\to s\in P\setminus P'} s) \in \mathscr{S} \end{cases} \end{cases}$$

We say that $\mathscr{S}$ *is a* simulation *if:*

$$\mathscr{S} \subseteq \mathbb{E}\mathscr{S}$$

The notion of simulation is at the basis of our subtyping algorithm. The intuition of the simulation is that if we consider the statements of Lemmas 4.3.9 and 4.3.10 as if they were rewriting rules (from right to left), then $\mathbb{E}\mathscr{S}$ contains all the types that we can deduce to be empty in one step reduction when we suppose that the types in $\mathscr{S}$ are empty. A simulation is thus a set that is already saturated with respect to such a rewriting. In particular, if we consider the statements of Lemmas 4.3.9 and 4.3.10 as inference rules for determining when a type is equal to $\mathbb{0}$, then $\mathbb{E}\mathscr{S}$ is the

set of immediate consequences of $\mathscr{S}$, and a simulation is a *self-justifying* set, that is a co-inductive proof of the fact that all its elements are equal to $\mathbb{0}$.

In what follows we show that simulations soundly and completely characterize the set of empty types of a convex well-founded model. More precisely, we show that every type in a simulation is empty (soundness) and that the set of all empty types is a simulation (completeness), actually, the largest simulation.

Lemma 4.4.3. *Let $[\![\_]\!] : \mathcal{T} \to \mathscr{P}(\mathcal{D})^{\mathcal{V}} \to \mathscr{P}(\mathcal{D})$ be a set-theoretic interpretation and t a type. If $\forall \eta \in \mathscr{P}(\mathcal{D})^{\mathcal{V}} \,.\, [\![t]\!]\eta = \emptyset$, then $\forall \sigma : \mathcal{V} \to \mathcal{T} \,.\, \forall \eta \in \mathscr{P}(\mathcal{D})^{\mathcal{V}} \,.\, [\![t\sigma]\!]\eta = \emptyset$*

Proof. Suppose there exists $\sigma : \mathcal{V} \to \mathcal{T}$ and $\eta \in \mathscr{P}(\mathcal{D})^{\mathcal{V}}$ such that $[\![t\sigma]\!]\eta \neq \emptyset$. Then we consider the semantic assignment η' such that

$$\forall \alpha \in \mathsf{var}(t) \,.\, \eta'(\alpha) = [\![\sigma(\alpha)]\!]\eta$$

Applying Lemma 4.3.6, we have $[\![t]\!]\eta' = [\![t\sigma]\!]\eta \neq \emptyset$, which contradicts that $\forall \eta \in \mathscr{P}(\mathcal{D})^{\mathcal{V}} \,.\, [\![t]\!]\eta = \emptyset$. Therefore, the result follows. □

Lemma 4.4.3 shows that the containment on the right hand side of equation (3.1) in Section 3.1 is a necessary condition, here reformulated as the fact that if a type is empty, then all its syntactic instances are empty. In particular if a type is empty we can rename all its type variables without changing any property. So when working with empty types or, equivalently, with subtyping relations, types can be considered equivalent modulo α-renaming (*i.e.*, the renaming of type variables).

The first result we prove is that every simulation contains only empty types.

Theorem 4.4.4 (Soundness). *Let $[\![\_]\!] : \mathcal{T} \to \mathscr{P}(\mathcal{D})^{\mathcal{V}} \to \mathscr{P}(\mathcal{D})$ be a convex well-founded model and $\mathscr{S}$ a simulation. Then for all $\tau \in \mathscr{S}$, we have $\forall \eta \in \mathscr{P}(\mathcal{D})^{\mathcal{V}} \,.\, [\![\tau]\!]\eta = \emptyset$*

Proof. Consider a type $\tau \in \mathscr{S}$. Then $\forall \eta \in \mathscr{P}(\mathcal{D})^{\mathcal{V}} \,.\, [\![\tau]\!]\eta = \emptyset$ holds, if and only if

$$\forall d \in \mathcal{D} \,.\, \forall \eta \in \mathscr{P}(\mathcal{D})^{\mathcal{V}} \,.\, d \notin [\![\tau]\!]\eta$$

Let us take $d \in \mathcal{D}$ and $\eta \in \mathscr{P}(\mathcal{D})^{\mathcal{V}}$. Since the interpretation is structural we can prove this property by induction on the structure of d. Since $\mathscr{S}$ is a simulation, we also have $\tau \in \mathbb{E}\mathscr{S}$, that is:

$$\forall s \in [\tau]_{\approx} \,.\, \forall u \in U \,.\, \forall (P,N) \in \mathsf{dnf}(s) \,.\, (P \subseteq \mathscr{A}_u \Rightarrow C_u^{P,N \cap \mathscr{A}_u}) \tag{4.5}$$

where the conditions $C_u^{P,N}$ are the same as those in Definition 4.4.2. The result then will follow from proving a statement stronger than the one of the lemma, that is, if $\tau \in \mathbb{E}\mathscr{S}$, then $\forall d \in \mathcal{D} . \forall \eta \in \mathscr{P}(\mathcal{D})^{\mathcal{V}} . d \notin [\![\tau]\!]\eta$. Or, equivalently:

$$\begin{aligned} \forall s \in [\tau]_{\approx} \,.\, \forall u \in U \,.\, \forall (P,N) \in \mathsf{dnf}(s) \,.\, \\ (P \subseteq \mathscr{A}_u \Rightarrow C_u^{P,N \cap \mathscr{A}_u}) \Rightarrow d \notin \bigcap_{\boldsymbol{a} \in P} [\![\boldsymbol{a}]\!]\eta \setminus \bigcup_{\boldsymbol{a} \in N} [\![\boldsymbol{a}]\!]\eta \end{aligned} \tag{4.6}$$

Let us take $s \in [\tau]_{\approx}$, $(P,N) \in \mathsf{dnf}(s)$ and u be the kind of d. Let us consider the possible cases for an atom $\boldsymbol{a} \in P$. Consider first $\boldsymbol{a} \notin \mathcal{V}$: if $\boldsymbol{a} \in \mathscr{A} \setminus \mathscr{A}_u$, then clearly $d \notin [\![\boldsymbol{a}]\!]\eta$.

If $\boldsymbol{a} \in \mathcal{V}$ then we know (by Lemma 4.3.4 if $u = \mathsf{basic}$, Lemma 4.3.7 if $u = \mathsf{prod}$, and Lemma 4.3.8 if $u = \mathsf{fun}$) that (4.6) holds if and only if it holds for another $s' \in [\tau]_{\approx}$ and $(P', N') \in \mathsf{dnf}(s')$ in which the variable $\boldsymbol{a} \notin P'$. As a consequence, if we can prove the result holds when $P \subseteq \mathscr{A}_u$, then (4.6) holds.

So assume that $P \subseteq \mathscr{A}_u$. Applying (4.5), we obtain that $C_u^{P,N\cap\mathscr{A}_u}$ holds. It remains to prove that:

$$d \notin \bigcap_{\boldsymbol{a}\in P} [\![\boldsymbol{a}]\!]\eta \setminus \bigcup_{\boldsymbol{a}\in N} [\![\boldsymbol{a}]\!]\eta$$

$\underline{u = \mathsf{basic}, d = c}$. The condition $C_{\mathsf{basic}}^{P,N\cap\mathscr{A}_{\mathsf{basic}}}$ is:

$$\forall \eta \in \mathscr{P}(\mathcal{D})^{\mathcal{V}} \,.\, \bigcap_{b\in P} \mathbb{B}(b) \subseteq \bigcup_{b\in N\cap\mathscr{A}_{\mathsf{basic}}} \mathbb{B}(b)$$

As a consequence, we get:

$$d \notin \bigcap_{\boldsymbol{a}\in P} [\![\boldsymbol{a}]\!]\eta \setminus \bigcup_{\boldsymbol{a}\in N\cap\mathscr{A}_{\mathsf{basic}}} [\![\boldsymbol{a}]\!]\eta$$

and *a fortiori*:

$$d \notin \bigcap_{\boldsymbol{a}\in P} [\![\boldsymbol{a}]\!]\eta \setminus \Big(\bigcup_{\boldsymbol{a}\in N\cap\mathscr{A}_{\mathsf{basic}}} [\![\boldsymbol{a}]\!]\eta \cup \bigcup_{\boldsymbol{a}\in N\setminus\mathscr{A}_{\mathsf{basic}}} [\![\boldsymbol{a}]\!]\eta \Big)$$

which yields the result.

$\underline{u = \mathsf{prod}, d = (d_1, d_2)}$. The condition $C_u^{P,N\cap\mathscr{A}_u}$ is:

$$\forall N' \subseteq N \cap \mathscr{A}_{\mathsf{prod}} \,.\, \begin{cases} \mathsf{dnf}(\bigwedge_{t_1\times t_2\in P} t_1 \wedge \bigwedge_{t_1\times t_2\in N'} \neg t_1) \in \mathscr{S} \\ \mathsf{or} \\ \mathsf{dnf}(\bigwedge_{t_1\times t_2\in P} t_2 \wedge \bigwedge_{t_1\times t_2\in N\setminus N'} \neg t_2) \in \mathscr{S} \end{cases}$$

For each N', we apply the induction hypothesis to d_1 and to d_2. We get:

$$d_1 \notin [\![\bigwedge_{t_1\times t_2\in P} t_1 \wedge \bigwedge_{t_1\times t_2\in N'} \neg t_1]\!]\eta \;\mathsf{or}\; d_2 \notin [\![\bigwedge_{t_1\times t_2\in P} t_2 \wedge \bigwedge_{t_1\times t_2\in N\setminus N'} \neg t_2]\!]\eta$$

That is:

$$d \notin \Big(\bigcap_{t_1\times t_2\in P} [\![t_1]\!]\eta \setminus \bigcup_{t_1\times t_2\in N'} [\![t_1]\!]\eta \Big) \times \Big(\bigcap_{t_1\times t_2\in P} [\![t_2]\!]\eta \setminus \bigcup_{t_1\times t_2\in N\setminus N'} [\![t_2]\!]\eta \Big)$$

Since to $[\![t_1]\!]\eta \times [\![t_2]\!]\eta = [\![t_1 \times t_2]\!]\eta$, then we get:

$$d \notin \bigcap_{\boldsymbol{a}\in P} [\![\boldsymbol{a}]\!]\eta \setminus \bigcup_{\boldsymbol{a}\in N\cap\mathscr{A}_{\mathsf{prod}}} [\![\boldsymbol{a}]\!]\eta$$

and *a fortiori*:

$$d \notin \bigcap_{\boldsymbol{a}\in P} [\![\boldsymbol{a}]\!]\eta \setminus \bigcup_{\boldsymbol{a}\in N} [\![\boldsymbol{a}]\!]\eta$$

$\underline{u = \mathsf{fun}, d = \{(d_1, d_1'), \ldots, (d_n, d_n')\}}$. The condition $C_u^{P,N\cap\mathscr{A}_u}$ states that there exists

$t_0 \to s_0 \in N$ such that, for all $P' \subseteq P$

$$\mathsf{dnf}(t_0 \wedge \bigwedge_{t \to s \in P'} \neg t) \in \mathscr{S} \text{ or } \begin{cases} P' \neq P \\ \text{and} \\ \mathsf{dnf}((\neg s_0) \wedge \bigwedge_{t \to s \in P \setminus P'} s) \in \mathscr{S} \end{cases}$$

Applying the induction hypothesis to the d_i and d'_i (note that if $d'_i = \Omega$, then $d'_i \notin [\![\tau]\!]\eta$ is trivial for all τ):

$$d_i \notin [\![t_0 \wedge \bigwedge_{t \to s \in P'} \neg t]\!]\eta \text{ or } \begin{cases} P \neq P' \\ \text{and} \\ d'_i \notin [\![(\neg s_0) \wedge \bigwedge_{t \to s \in P \setminus P'} s]\!]\eta \end{cases}$$

Assume that $\forall i \,.\, (d_i \in [\![t_0]\!]\eta \Rightarrow d'_i \in [\![s_0]\!]\eta)$. Then we have $d \in [\![t_0 \to s_0]\!]\eta$.
Otherwise, let us consider i such that $d_i \in [\![t_0]\!]\eta$ and $d'_i \notin [\![s_0]\!]\eta$. The formula above gives for any $P' \subseteq P$:

$$(d_i \in \bigcup_{t \to s \in P'} [\![t]\!]\eta) \text{ or } ((P' \neq P) \text{ and } (d'_i \in \{\Omega\} \cup \bigcup_{t \to s \in P \setminus P'} [\![\neg s]\!]\eta))$$

Let's take $P' = \{t \to s \in P \mid d_i \notin [\![t]\!]\eta\}$. We have $d_i \notin \bigcup_{t \to s \in P'} [\![t]\!]\eta$, and thus $P' \neq P$ and $d'_i \in \{\Omega\} \cup \bigcup_{t \to s \in P \setminus P'} [\![\neg s]\!]\eta$. We can thus find $t \to s \in P \setminus P'$ such that $d'_i \notin [\![s]\!]\eta$,and because $t \to s \notin P'$, we also have $d_i \in [\![t]\!]\eta$. We have thus proved that $d \notin [\![t \to s]\!]\eta$ for some $t \to s \in P$.
In both cases, we get:

$$d \notin \bigcap_{\boldsymbol{a} \in P} [\![\boldsymbol{a}]\!]\eta \setminus \bigcup_{\boldsymbol{a} \in N \cap \mathscr{A}_{\mathsf{fun}}} [\![\boldsymbol{a}]\!]\eta$$

and *a fortiori*

$$d \notin \bigcap_{\boldsymbol{a} \in P} [\![\boldsymbol{a}]\!]\eta \setminus \bigcup_{\boldsymbol{a} \in N} [\![\boldsymbol{a}]\!]\eta$$

□

Completeness derives straightforwardly from the construction of simulation and the lemmas we proved about it.

Theorem 4.4.5. *Let $[\![\_]\!] : \mathcal{T} \to \mathscr{P}(\mathcal{D})^{\mathcal{V}} \to \mathscr{P}(\mathcal{D})$ be a convex set-theoretic interpretation. We define a set of normal forms $\mathscr{S}$ by:*

$$\mathscr{S} \stackrel{\text{def}}{=} \{\tau \mid \forall \eta \in \mathscr{P}(\mathcal{D})^{\mathcal{V}} \,.\, [\![\tau]\!]\eta = \emptyset\}$$

Then:

$$\mathbb{E}\mathscr{S} = \{\tau \mid \forall \eta \in \mathscr{P}(\mathcal{D})^{\mathcal{V}} \,.\, \mathbb{E}(\tau)\eta = \emptyset\}$$

Proof. Immediate consequence of Lemmas 4.3.9, 4.3.10 and 4.4.3. □

Corollary 4.4.6. *Let $[\![\_]\!] : \mathcal{T} \to \mathscr{P}(\mathcal{D})^{\mathcal{V}} \to \mathscr{P}(\mathcal{D})$ be a convex structural interpretation. Define as above $\mathscr{S} = \{\tau \mid \forall \eta \in \mathscr{P}(\mathcal{D})^{\mathcal{V}} \,.\, \mathbb{E}(\tau)\eta = \emptyset\}$. If $[\![\_]\!]$ is a model, then $\mathscr{S} = \mathbb{E}\mathscr{S}$.*

This corollary implies that the simulation that contains all the empty types is the largest simulation[2]. In particular it entails the following corollary.

Corollary 4.4.7 (Completeness). *Let* $[\![\_]\!] : \mathcal{T} \to \mathscr{P}(\mathcal{D})^{\mathcal{V}} \to \mathscr{P}(\mathcal{D})$ *be a convex well-founded model and* s *and* t *two types. Then* $s \leq t$ *if and only if there exists a simulation* $\mathscr{S}$ *such that* $\mathsf{dnf}(s \wedge \neg t) \in \mathscr{S}$.

The corollary states that to check whether $s \leq t$ we have to check whether there exists a simulation that contains the normal form, denoted by τ_0, of $s \wedge \neg t$. Thus a simple subtyping algorithm consists in using Definition 4.4.2 as a set of saturation rules: we start from the set containing just τ_0 and try to saturate it. At each step of saturation we add just the normal forms in which we eliminated top-level variables according to Lemmas 4.3.3, 4.3.4, 4.3.7, and 4.3.8. Because of the presence of "or" in the definition, the algorithm follows different branches until it reaches a simulation (in which case it stops with success) or it adds a non-empty type (in which case the whole branch is abandoned).

All results we stated so far have never used the regularity of types. The theory holds also for non regular types and the algorithm described above is a sound and complete procedure to check their inclusion. The only result that needs regularity is decidability.

4.5 Decidability

The subtyping relation on polymorphic regular types is decidable in EXPTIME.[3] We prove decidability but the result on complexity is due to Gesbert *et al.* [GGL11] who gave a linear encoding of the relation presented here in their variant of the μ-calculus, for which they have an EXPTIME solver [GLS07], thus obtaining a subtyping decision algorithm that is EXPTIME (in doing so they also spotted a subtle error in the original definition of our subtyping algorithm). This is also a lower bound for the complexity since the subtyping problem for ground regular types (without arrows) is known to be EXPTIME-complete.

To prove decidability we just prove that our algorithm terminates. The decision algorithm for which we prove termination is essentially a dull implementation of the one presented in Section 3.4: it tries to check the emptiness of some type by trying to build a simulation containing it. It does so by decomposing the original problem into subproblems and using them to saturate the set obtained up to that point. More precisely, in order to check the emptiness of a type, the algorithm first normalizes it; it simplifies mixed intersections; it eliminates toplevel negative occurrences of variables by applying the substitution $\{\neg\beta/\alpha\}$; it eliminates toplevel positive occurrences of variables by applying $\{(\alpha_1 \times \alpha_2) \vee \alpha_3/\alpha\}$, $\{(\alpha_1 \to \alpha_2) \vee \alpha_3/\alpha\}$ or $\{((\alpha_1 \to \alpha_2) \setminus (\mathbb{1} \to \mathbb{0})) \vee \alpha_3/\alpha\}$ substitutions (in the proof of termination we will not effectively apply these substitutions but keep them symbolical); it eliminates the toplevel constructors by applying the

[2]Moreover, as shown in Section 4.6, the condition for a set-theoretic interpretation to be a model depends only on the subtyping relation for ground types it induces.

[3]Strictly speaking we cannot speak of *the* subtyping relation in general, but just of the subtyping relation induced by a particular model. Here we intend the subtyping relation for the model used in the proof of Corollary 4.6.8, that is, of the universal model of [FCB08] with a set of basic types whose containment relation is decidable in EXPTIME. This coincides with the relation studied in [GGL11].

equations of Definition 4.4.2 as left-to-right rewriting rules; the last point yields a set of different subproblems: the algorithm checks whether all the subproblems are solved (*i.e.*, it reached a simulation), otherwise it recurses on all of them. The implementation we consider is "dull" insofar as it will explore all the subproblems to their end, even in the case when an answer could already be given earlier. So for example to check the emptiness of a disjunctive normal form, the algorithm will check the emptiness of *all* types in the union, even though it could stop as soon as it had found a non-empty type; similarly, to check the emptiness of $t_1 \times t_2$ the algorithm will check the emptiness of t_1 *and* of t_2, even though a positive results for, say, t_1 would make the check for t_2 useless. We do so because we want to prove that the algorithm is strongly normalizing, that is, termination does not depend on the reduction strategy.

The proof of termination is conceptually simple and relies on a well known result in combinatorics (Higman's Lemma [Hig52]) to show that all the branches that do not stop on a non-empty type will eventually stop because they will arrive on an instance of a memoized term. It first requires two simple lemmas:

Lemma 4.5.1. *Let $[\![\_]\!]$ be a set-theoretic interpretation, $\mathbb{E}(\_)$ its associated extensional interpretation, and t, t' be two types. Then*

$$\forall \eta \;.\; \mathbb{E}(t)\eta = \emptyset \Rightarrow \forall \eta \;.\; \mathbb{E}(t \wedge t')\eta = \emptyset.$$

Proof. Straightforward since $\mathbb{E}(t \wedge t')\eta = \mathbb{E}(t)\eta \cap \mathbb{E}(t')\eta$. □

Lemma 4.5.2. *Let $[\![\_]\!]$ be a model, $\mathbb{E}(\_)$ its associated extensional interpretation, and t a type. Then*

$$\forall \eta \;.\; \mathbb{E}(t)\eta = \emptyset \Rightarrow \forall \eta \;.\; \mathbb{E}(t\overline{\sigma})\eta = \emptyset$$

where $\overline{\sigma} = \sigma_1 \dots \sigma_n$ for some $n \geq 0$ and σ_i denotes a substitution of the form $\{\neg\beta/\alpha\}$, $\{(\alpha_1 \times \alpha_2) \vee \alpha_3/\alpha\}$, $\{(\alpha_1 \to \alpha_2) \vee \alpha_3/\alpha\}$ or $\{((\alpha_1 \to \alpha_2) \setminus (\mathbb{1} \to \mathbb{0})) \vee \alpha_3/\alpha\}$.

Proof. An application of Lemma 4.4.3 and by the condition of a model. □

We can now directly prove termination.

Theorem 4.5.3. *The algorithm terminates on all types.*

Proof. Consider a generic type s. Let $\mathsf{dnf}(s)$ be a disjunctive normal form of s. Since it is a union, $\mathsf{dnf}(s)$ is empty if and only if all its single normal forms are empty. Let us just consider one of its single normal forms t.

To check the emptiness of t, the algorithm first checks whether t is memoized (*i.e.*, whether this type was already met modulo α-renaming during the check and is therefore supposed to be empty), or t is an instance of a type t' that is memoized (see Lemma 4.5.2), or t from which some atom intersections are removed is an instance of a memoized type (this step is correct by Lemma 4.5.1). If it is, then the algorithm terminates, otherwise, it memoizes t.[4] Next if $|\mathsf{tlv}(t)| > 0$, then the algorithm performs the appropriate substitution(s) (see Lemmas 4.3.3, 4.3.7, and 4.3.8). Next according to

[4] If t happens to be nonempty, then in the real algorithm t will be removed from the memoized set, but this is just an optimization that reduces the number of required checks and does not affect the final result.

the decomposition rules (see Lemmas 4.3.9 and 4.3.10), the algorithm decomposes t into several types which are the candidate types to be checked on the next iteration. Since t is empty if and only if some of the candidate types (depending on the decomposition rules) are empty, then the algorithm reiterates the process on them. This iteration eventually stops. In order to prove it, consider the form of all types that can be met during the checking of the emptiness of t. For the sake of the proof we will consider only the terms to which we applied the transformation of Lemma 4.3.3 (*i.e.*, without any toplevel negated variable). These are single normal forms (*cf.* Definition 4.3.1), that is, intersections of atoms. Now in any of these intersections we can distinguish two parts: there is a part of the intersection that is formed just by type variables (these are either the variables of the original type or some fresh variables that were introduced to eliminate a toplevel variable), and a second part that intersects basic and/or product and/or arrow types. If the check of memoization fails, then the first part of the type formed by the intersection of variables is eliminated, and the appropriate substitution(s) is (are) applied to the second part. Then the atoms in this second part to which the substitution(s) is (are) applied are decomposed to form the next set of single normal forms. It is then clear that the second part of all the candidate single normal forms met by the algorithm are formed by chunks of the original type to which a succession of substitutions of the same form as those used in Lemmas 4.3.3, 4.3.7, and 4.3.8 is applied. So we can formally characterize all the single normal forms examined by the algorithm when checking the emptiness of a (single normal form) type t.

First, consider the original type t: for the sake of simplicity, in what follows we just consider the case in which $t \leq \mathbb{1} \times \mathbb{1}$ so as we just consider substitutions of the form $\{(\alpha_1 \times \alpha_2) \vee \alpha_3/\alpha\}$ (the proof for the other cases of substition $\{(\alpha_1 \to \alpha_2) \vee \alpha_3/\alpha\}$, and $\{((\alpha_1 \to \alpha_2) \setminus (\mathbb{1} \to \mathbb{0})) \vee \alpha_3/\alpha\}$ is exactly the same, only more cluttered with indexes). Next consider the set of all subtrees of t: since t is regular, then this set is finite. Finally consider the set $\mathcal{C}$ of all Boolean combinations of terms of the previous sets (actually, just single normal forms would suffice) union $\{\mathbb{1}\}$: modulo normalization (or modulo type semantics) there are only finitely many distinct combinations of a finite set of types, therefore $\mathcal{C}$ is finite as well. It is clear from what we said before that all the types that will be considered during the emptiness check for t will be of the form

$$(t' \wedge \beta_1 \wedge ... \wedge \beta_h)\{(\alpha_1^1 \times \alpha_1^2) \vee \alpha_1^3/\alpha_1\}...\{(\alpha_n^1 \times \alpha_n^2) \vee \alpha_n^3/\alpha_n\} \wedge \gamma_1 \wedge \ldots \wedge \gamma_p \qquad (4.7)$$

where $t' \in \mathcal{C}$, $h, n, p \geq 0$, $\alpha_i \in \mathsf{var}(t') \cup \{\alpha_j^1, \alpha_j^2, \alpha_j^3 | 1 \leq j \leq i-1\}$, $\{\beta_i | 1 \leq i \leq h\} \cup \{\gamma_i | 1 \leq i \leq p\} \subseteq \{\alpha_i^1, \alpha_i^2, \alpha_i^3 | 1 \leq i \leq n\}$, $\{\beta_i | 1 \leq i \leq h\} \subseteq \{\alpha_i | 1 \leq i \leq n\}$, and $\{\gamma_i | 1 \leq i \leq p\} \cap \{\alpha_i | 1 \leq i \leq n\} = \emptyset$. In a nutshell, t' is an intersection of possibly negated products, the β_i's are the type variables that will be affected by at least one of the substitutions following them, and the γ_i's are those that are not affected by any substitution (note that: (*i*) the γ_i's could be moved in the scope of the substitution, but we prefer to keep them separated for the time being, and (*ii*) all toplevel variables are not negated since we consider that we already applied Lemma 4.3.3 to get rid of possible negations).

Let us now follow one particular sequence of the check and imagine by contradiction that this sequence is infinite. All the types in the sequence are of the form described in (4.7). Since $\mathcal{C}$ is finite, then there exists a type $t^\circ \in \mathcal{C}$ occurring infinitely many times in the sequence. Let s_1 and s_2 be two single normal forms in the sequence containing

this particular t°, namely:

$$\begin{array}{rcl} s_1 & = & (t^\circ \wedge \beta_1 \wedge \ldots \wedge \beta_h)\{(\alpha_1^1 \times \alpha_1^2) \vee \alpha_1^3/\alpha_1\}\ldots\{(\alpha_n^1 \times \alpha_n^2) \vee \alpha_n^3/\alpha_n\} \wedge \gamma_1 \wedge \ldots \wedge \gamma_p \\ s_2 & = & (t^\circ \wedge \beta_1 \wedge \ldots \wedge \beta_k)\{(\alpha_1^1 \times \alpha_1^2) \vee \alpha_1^3/\alpha_1\}\ldots\{(\alpha_m^1 \times \alpha_m^2) \vee \alpha_m^3/\alpha_m\} \wedge \gamma_1 \wedge \ldots \wedge \gamma_q \end{array}$$

Since we are checking the emptiness of these two types, then the types can be considered modulo α-renaming of their variables. This justifies the fact that, without loss of generality, in the two terms above we can consider the first $\min\{n, m\}$ substitutions, the first $\min\{h, k\}$ β-variables and the first $\min\{p, q\}$ γ variables to be the same in both terms.

Let us consider again the infinite sequence of candidates that are formed by t° and consider the three cardinalities of the β variables, of the substitutions, and of the γ variables, respectively. Since $\mathbb{N}^3$ with a point-wise order is a well-quasi-order, we can apply Higman's Lemma [Hig52] to this sequence and deduce that in the sequence there occur two types as the s_1 and s_2 above such that s_1 occurs before s_2 and $h \leq k$, $n \leq m$, and $p \leq q$.

Let us write σ_i^j for the substitution

$$\{(\alpha_i^1 \times \alpha_i^2) \vee \alpha_i^3/\alpha_i\}\ldots\{(\alpha_j^1 \times \alpha_j^2) \vee \alpha_j^3/\alpha_j\}$$

with $i \leq j$. We have that s_2 is equal to

$$(t^\circ \wedge \beta_1 \wedge \ldots \wedge \beta_h \wedge \ldots \wedge \beta_k)\sigma_1^m \wedge \gamma_1 \wedge \ldots \wedge \gamma_p \wedge \ldots \wedge \gamma_q.$$

We exit the rightmost β's by duplicating the σ_1^m substitutions, yielding:

$$(t^\circ \wedge \beta_1 \wedge \ldots \wedge \beta_h)\sigma_1^m \wedge \gamma_1 \wedge \ldots \wedge \gamma_p \wedge \ldots \wedge \gamma_q \wedge (\beta_{h+1} \wedge \ldots \wedge \beta_k)\sigma_1^m$$

since the γ's are independent from the α's we can move the p leftmost ones inside a part of the substitutions obtaining

$$((t^\circ \wedge \beta_1 \wedge \ldots \wedge \beta_h)\sigma_1^n \wedge \gamma_1 \wedge \ldots \wedge \gamma_p)\sigma_{n+1}^m \wedge \gamma_{p+1} \wedge \ldots \wedge \gamma_q \wedge (\beta_{h+1} \wedge \ldots \wedge \beta_k)\sigma_1^m$$

which by definition of s_1 is equal to

$$s_1\sigma_{n+1}^m \wedge \gamma_{p+1} \wedge \ldots \wedge \gamma_q \wedge (\beta_{h+1} \wedge \ldots \wedge \beta_k)\sigma_1^m$$

In conclusion s_2 has the following form:

$$s_2 = s_1\sigma_{n+1}^m \wedge \gamma_{p+1} \wedge \ldots \wedge \gamma_q \wedge \beta_{h+1}\sigma_1^m \wedge \ldots \wedge \beta_k\sigma_1^m$$

therefore it is an intersection a part of which is an instance of the (memoized) type s_1. Therefore the algorithm (and thus the sequence) should have stopped on the check of s_2, which contradicts the hypothesis that the sequence is infinite. □

In order to understand how the algorithm actually terminates on empty infinite types, consider for instance the following type:

$$\alpha \wedge (\alpha \times x) \wedge \neg(\alpha \times y)$$

where $x = (\alpha \wedge (\alpha \times x)) \vee \mathsf{nil}$ and $y = (\alpha \times y) \vee \mathsf{nil}$. First, the algorithm memoizes it. By an application of Lemma 4.3.7, the algorithm performs the substitution yielding

$$(\alpha_1 \times \alpha_2) \wedge (((\alpha_1 \times \alpha_2) \vee \alpha_3) \times x) \wedge \neg(((\alpha_1 \times \alpha_2) \vee \alpha_3) \times y).$$

Following Lemma 4.3.9, the algorithm checks the candidate types as follows:

$$\begin{cases} \begin{cases} \alpha_1 \wedge ((\alpha_1 \times \alpha_2) \vee \alpha_3) = \mathbb{0} & (1) \\ \mathsf{or} \\ \alpha_2 \wedge x \wedge \neg y = \mathbb{0} & (2) \end{cases} \\ \mathsf{and} \\ \begin{cases} \alpha_1 \wedge ((\alpha_1 \times \alpha_2) \vee \alpha_3) \wedge \neg((\alpha_1 \times \alpha_2) \vee \alpha_3) = \mathbb{0} & (3) \\ \mathsf{or} \\ \alpha_2 \wedge x = \mathbb{0} & (4) \end{cases} \end{cases}$$

Type (1) is finite and nonempty, and type (3) is finite and empty. It is not necessary to check type (4) insofar as one of its expansions, $\alpha_1 \wedge \mathsf{nil}$, is not empty. So the algorithm terminates on type (4) as well. Considering type (2), it is neither memoized nor an instance of a memoized type, so it is memoized as well. Then the algorithm unfolds it and gets

$$\alpha_2 \wedge ((\alpha_1 \times \alpha_2) \vee \alpha_3) \wedge (((\alpha_1 \times \alpha_2) \vee \alpha_3) \times x) \wedge \neg(((\alpha_1 \times \alpha_2) \vee \alpha_3) \times y)$$

The algorithm matches the unfolded type with the memoized ones. It is an instance of

$$\alpha_2 \wedge \alpha \wedge (\alpha \times x) \wedge \neg(\alpha \times y)$$

where the substitution is $\{(\alpha_1 \times \alpha_2) \vee \alpha_3 /_\alpha\}$. Although it is not memoized, it is deduced to be empty from the memoized $\alpha \wedge (\alpha \times x) \wedge \neg(\alpha \times y)$ and Lemma 4.5.1.

4.6 Convex models

The last step of our formal development is to prove that there exists at least one set-theoretical model that is convex. From a practical point of view this step is not necessary since one can always take the work we did so far as a syntactic definition of the subtyping relation. However, the mathematical support makes our theory general and applicable to several different domains (*e.g.*, Gesbert *et al*'s starting point and early attempts relied on this result), so finding a model is not done just for the sake of the theory. As it turns out, there actually exist a lot of convex models since every model for ground types with infinite denotations is convex. So to define a convex model it just suffices to take any model defined in [FCB08] and straightforwardly modify the interpretation of basic and singleton types (more generally, of all indivisible types[5]) so they have infinite denotations.

[5]Our system has a very peculiar indivisible type: $\mathbb{1} \to \mathbb{0}$, the type of the functions that diverge on all arguments. This can be handled by adding a fixed infinite set of fresh elements of the domain to the interpretation of every arrow type (*cf.* the proof of Corollary 4.6.8).

Definition 4.6.1 (Infinite support). *A model* $(\mathcal{D}, [\![\_]\!])$ *is* with infinite support *if for every ground type* t *and assignment* η*, if* $[\![t]\!]\eta \neq \emptyset$*, then* $[\![t]\!]\eta$ *is an infinite set.*

What we want to prove then is the following theorem.

Theorem 4.6.2. *Every well-founded model with infinite support is convex.*

The proof of this theorem is quite technical—it is the proof that required us most effort—and proceeds in three logical steps. First, we prove that the theorem holds when all types at issue do not contain any product or arrow type. In other words, we prove that equation (3.5) holds for $[\![\_]\!]$ with infinite support and where all t_i's are Boolean combinations of type variables and basic types. This is the key step in which we use the hypothesis of infinite support, since in the proof—done by contradiction—we need to pick an unbounded number of elements from our basic types in order to build a counterexample that proves the result. Second, we extend the previous proof to any type t_i that contains finitely many applications of the product constructor. In other terms, we prove the result for any (possibly infinite) type, provided that recursion traverses just arrow types, but not product types. As in the first case, the proof builds some particular elements of the domain. In the presence of type constructors the elements are built inductively. This is possible since products are not recursive, while for arrows it is always possible to pick a fresh appropriate element that resides in that arrow since every arrow type contains the (indivisible) closed type $\mathbb{1} \to \mathbb{0}$. Third, and last, we use the previous result and the well-foundedness of the model to show that the result holds for all types, that is, also for types in which recursion traverses a product type. More precisely, we prove that if we assume that the result does not hold for some infinite product type then it is possible to build a finite product type (actually, a finite expansion of the infinite type) that disproves equation (3.5), contradicting what is stated in our second step. Well-foundedness of the model allows us to build this finite type by induction on the elements denoted by the infinite one.

Although the proof of Theorem 4.6.2 that follows is quite technical, its essence can be grasped by considering the case of propositional logic $t ::= \alpha \mid t \vee t \mid t \wedge t \mid \neg t \mid \mathbb{0} \mid \mathbb{1}$, where α's are then propositional variables (the construction works also in the presence of basic types). It is relatively easy to prove by contrapositive that if all propositions (*i.e.*, types) are interpreted into infinite sets, then for all $n \geq 2$

$$\forall \eta \,.\, ([\![t_1]\!]\eta{=}\emptyset) \text{ or } \ldots \text{ or } ([\![t_n]\!]\eta{=}\emptyset) \;\Rightarrow\; (\forall \eta \,.\, [\![t_1]\!]\eta{=}\emptyset) \text{ or } \ldots \text{ or } (\forall \eta \,.\, [\![t_n]\!]\eta{=}\emptyset) \qquad (4.8)$$

holds. First, put all the t_i's in disjunctive normal form. Without loss of generality we can consider that

$$t_i = \bigwedge_{j \in P_i} \alpha_j \wedge \bigwedge_{j \in N_i} \neg \alpha_j \quad (P_i \cap N_i = \emptyset)$$

since, if t_i is a union, then by exiting the union from the interpretation we obtain a special case of (4.8) with a larger n. Next, proceed by contrapositive: suppose $\exists \eta_1, \ldots, \eta_n$ such that $[\![t_i]\!]\eta_i \neq \emptyset$, then we can build $\bar{\eta}$ such that for all i, $[\![t_i]\!]\bar{\eta} \neq \emptyset$. The construction of $\bar{\eta}$ is done by iterating on the t_i's, picking at each iteration a particular element d of the domain and recording this choice in a set s_0 so as to ensure that at each iteration a different d is chosen. The construction of $\bar{\eta}$ is performed as follows:

Start with $s_0 = \emptyset$ and for $i = 1..n$:
- choose $d \in (\llbracket \mathbb{1} \rrbracket \eta_i \setminus s_0)$ (which is possible since $\llbracket \mathbb{1} \rrbracket \eta_i$ *is infinite* and s_0 is finite)
- add d to s_0 (to record that we used it) and to $\bar{\eta}(\alpha_j)$ for all $j \in P_i$.

Notice that at each step i of the loop we construct $\bar{\eta}$ such that $\llbracket t_i \rrbracket \bar{\eta} \neq \emptyset$ (since it contains the d selected in the i-th cycle) without disrupting the interpretation of the preceding loops. This is possible since a fresh element d of the domain (an element not in s_0) is used at each loop: the hypothesis that denotations are infinite ensures that a fresh element always exists. The construction in the proof also shows that in order to satisfy equation (4.8) for a given n every denotation of a non-empty type must contain at least n distinct elements (see also Theorem 4.6.9).

The technical proof proceeds as follows.

Definition 4.6.3 (**Positive and negative occurrences**). *Let $t \in \mathcal{T}$ and t' be a tree that occurs in t. An occurrence of t' in t is a* negative occurrence *of t if on the path going from the root of the occurrence to the root of t there is an odd number of $\neg$ nodes. It is a* positive occurrence *otherwise.*

For instance if $t = \neg(\alpha \times \neg\beta)$ then β is a positive occurrence of t while α and $\alpha \times \neg\beta$ are negative ones. Similarly no occurrence in $\alpha \to \beta$ is negative (so the notion of positive/negative concerns only negation and not co/contra-variant position).

Definition 4.6.4. *We use $\mathcal{T}^{\mathsf{fp}}$ to denote the set of* types with finite products*, that is, the set of all types in which every infinite branch contains a finite number of occurrences of the $\times$ constructor.*

The first two steps of our proof are proved simultaneously in the following lemma.

Lemma 4.6.5. *Let $(\mathcal{D}, \llbracket \_ \rrbracket)$ be a well-founded model with infinite support, and $t_i \in \mathcal{T}^{\mathsf{fp}}$ for $i \in [1..n]$. Then*

$$\forall \eta \,.\, (\llbracket t_1 \rrbracket \eta = \emptyset) \text{ or } \ldots \text{ or } (\llbracket t_n \rrbracket \eta = \emptyset) \iff (\forall \eta \,.\, \llbracket t_1 \rrbracket \eta = \emptyset) \text{ or } \ldots \text{ or } (\forall \eta \,.\, \llbracket t_n \rrbracket \eta = \emptyset)$$

Proof. The $\Leftarrow$ direction is trivial. For the other direction we proceed as follows. If at most one type is not ground, then the result is trivial. If $\mathsf{var}(t_i) \cap \mathsf{var}(t_j) = \emptyset$ for any two types t_i and t_j, that is, the sets of variables occurring in the types are pairwise disjoint, then the result follows since there is no correlation between the different interpretations. Assume that $\bigcup_{1 \leq i < j \leq n} (\mathsf{var}(t_i) \cap \mathsf{var}(t_j)) \neq \emptyset$. For convenience, we write CV for the above set of common variables. Suppose by contradiction that

$$(\forall \eta \,.\, \llbracket t_1 \rrbracket \eta = \emptyset) \text{ or } \ldots \text{ or } (\forall \eta \,.\, \llbracket t_n \rrbracket \eta = \emptyset) \tag{4.9}$$

does not hold. Then for each $i \in [1..n]$ there exists η_i such that $\llbracket t_i \rrbracket \eta_i \neq \emptyset$. If under the hypothesis that (4.9) does not hold we can find another assignment η' such that $\llbracket t_i \rrbracket \eta' \neq \emptyset$ for every $i \in [1..n]$, then this contradicts the hypothesis of the lemma and the result follows. We proceed by a case analysis on all possible combinations of the t_i's: under the hypothesis of existence of η_i such that $\llbracket t_i \rrbracket \eta_i \neq \emptyset$, we show how to build the η' at issue.

Case 1: Each t_i is a single normal form, that is, $t_i \in \mathscr{P}_f(\mathscr{A} \cup \mathcal{V}) \times \mathscr{P}_f(\mathscr{A} \cup \mathcal{V})$. Recall that we want to show that it is possible to construct an assignment of the common variables such that all the t_i's have non empty denotation. To do that we build the assignment for each variable step by step, by considering one t_i at a time, and adding to the interpretation of the common variables one element after one element. In order to produce for a given type t_i an assignment that does not interfere with the interpretation defined for a different t_j, we keep track in a set s_0 of the elements of the domain $\mathcal{D}$ we use during the construction. Since we start with an empty s_0 and we add to it an element at a time, then at each step s_0 will be finite and, thanks to the property of infinite support, it will always be possible to choose some element of the domain that we need to produce our assignment and that is not in s_0. More precisely, we construct a set s_0 such that $s_0 \cap [\![t'_i]\!]\eta_i \neq \emptyset$ for each $i \in [1..n]$ where t'_i is obtained from t_i by eliminating the top-level variables. Meanwhile, considering each variable $\alpha \in \bigcup_{i\in\{1,\dots,n\}} \mathsf{var}(t_i)$ (clearly including CV), we also construct a set s_α which is initialized as an empty set and that at the end of this construction is used to define $\eta'(\alpha)$.

Subcase 1: $\mathsf{dnf}(t_i) = \bigwedge_{j\in J_1} b^1_j \wedge \bigwedge_{j\in J_2} \neg b^2_j \wedge \bigwedge_{j\in J_3} \alpha^1_j \wedge \bigwedge_{j\in J_4} \neg\alpha^2_j$, where J_i's are finite sets. The construction is as follows:
If $\exists d \,.\, d \in [\![t_i]\!]\eta_i \wedge d \notin s_0$, then set $s_0 = s_0 \cup \{d\}$. For each variable α^1_j with $j \in J_3$, set $s_{\alpha^1_j} = s_{\alpha^1_j} \cup \{d\}$. If such a d does not exist, then t_i is not ground, since $[\![t_i]\!]\eta_i \subseteq s_0$ and s_0 is finite. As $\mathsf{dnf}(t_i) = \bigwedge_{j\in J_1} b^1_j \wedge \bigwedge_{j\in J_2} \neg b^2_j \wedge \bigwedge_{j\in J_3} \alpha^1_j \wedge \bigwedge_{j\in J_4} \neg\alpha^2_j$, then either $J_1 \cup J_2 = \emptyset$ and then we chose any element d' such that $d' \notin s_0$, or $J_1 \cup J_2 \neq \emptyset$ and $[\![\bigwedge_{j\in J_3} \alpha^1_j \wedge \bigwedge_{j\in J_4} \neg\alpha^2_j]\!]\eta_i \subseteq s_0$, then there exists an element d' such that $d' \in [\![\bigwedge_{j\in J_1} b^1_j \wedge \bigwedge_{j\in J_2} \neg b^2_j]\!]\eta_i$ and $d' \notin s_0$, since $[\![\bigwedge_{j\in J_1} b^1_j \wedge \bigwedge_{j\in J_2} \neg b^2_j]\!]\eta_i$ is non empty (because $[\![t_i]\!]\eta_i \neq \emptyset$) and infinite (since the type at issue is closed and $[\![\_]\!]$ is with infinite support). In both cases we set $s_0 = s_0 \cup \{d'\}$, and for each variable α^1_j with $j \in J_3$, set $s_{\alpha^1_j} = s_{\alpha^1_j} \cup \{d'\}$.

Subcase 2: $\mathsf{dnf}(t_i) = \bigwedge_{j\in J_1}(t^1_j \times t^2_j) \wedge \bigwedge_{j\in J_2} \neg(t^3_j \times t^4_j) \wedge \bigwedge_{j\in J_3} \alpha^1_j \wedge \bigwedge_{j\in J_4} \neg\alpha^2_j$, where J_i's are finite sets. If $|J_1| = |J_2| = 0$, then we are in the case of Subcase 1. If $|J_1| = 0$ and $|J_2| \neq 0$, then we can do the construction as Subcase 1 since $\mathscr{C} \subseteq [\![\bigwedge_{j\in J_2} \neg(t^3_j \times t^4_j)]\!]\eta_i$. Suppose then that $|J_1| > 0$: since we have

$$\bigwedge_{j\in J_1}(t^1_j \times t^2_j) = (\bigwedge_{j\in J_1} t^1_j \times \bigwedge_{j\in J_1} t^2_j)$$

then without loss of generality we can assume that $|J_1| = 1$, that is there is a single toplevel non negated product type. So we are considering the specific case for

$$\mathsf{dnf}(t_i) = (t'_1 \times t'_2) \wedge \bigwedge_{j\in J_3} \alpha^1_j \wedge \bigwedge_{j\in J_4} \neg\alpha^2_j.$$

What we do next is to build a particular element of the domain by exploring $(t'_1 \times t'_2)$ in a top-down way and stopping when we arrive to a basic type, or a variable, or *an arrow type*. So even though $(t'_1 \times t'_2)$ may be infinite (since it may contain an arrow type of infinite depth) the exploration will always terminate (unions, negations, *and* products always are finite: in particular products are finite because by hypothesis we are considering only types in $\mathcal{T}^{\mathsf{fp}}$). It can then

be defined recursively in terms of two mutually recursive explorations that return different results according to whether the exploration step has already crossed an even or an odd number of negations. So let t_1 be a type different from $\mathbb{0}$ and t_2 a type different from $\mathbb{1}$, we define the $\mathsf{explore\_pos}(t_1)$ and $\mathsf{explore\_neg}(t_2)$ procedures that, intuitively, explore the syntax tree for positive and negative occurrences, respectively, and which are defined as follows:

$\mathsf{explore\_pos}(t)$ case t of:

1. $t = t_1 \times t_2$. Let d_i be the result of $\mathsf{explore\_pos}(t_i)$ (for $i = 1, 2$): since t is not empty so must t_1 and t_2 be; we add both d_1 and d_2 to s_0 and return $d = (d_1, d_2)$.
2. $t = t_1 \to t_2$: we can choose any element $d \in \mathbb{1} \to \mathbb{0}$ (whatever t_1 and t_2 are) and return it.
3. $t = \mathbb{0}$: impossible.
4. $t = \mathbb{1}$: return any element $d \notin s_0$.
5. $t = b$: we can choose any element d such that $d \in [\![b]\!]\eta_i$ and $d \notin s_0$ and return it.
6. $t = \alpha$: we can choose any element $d \notin s_0$, set $s_\alpha = s_\alpha \cup \{d\}$ and return d.
7. $t = t_1 \vee t_2$: one of the two types is not empty. If it is t_1, then call $\mathsf{explore\_pos}(t_1)$. It yields $d_1 \notin s_0$ and we return it. Otherwise we call $\mathsf{explore\_pos}(t_2)$ and return its result.
8. $t = t_1 \wedge t_2$, then put it in disjunctive normal form. Since it is not empty, then one of its single normal forms must be non empty, as well: repeat for this non empty single normal form the construction of the corresponding subcase in this proof and return the d constructed by it.
9. $t = \neg t'$, then we call $\mathsf{explore\_neg}(t')$ add its result to s_0 and return it.

$\mathsf{explore\_neg}(t)$, case t of:

1. $t = t_1 \times t_2$: we can choose any element $d \in \mathscr{C}$ and $d \notin s_0$ and return it.
2. $t = t_1 \to t_2$: we can choose any element $d \in \mathscr{C}$ and $d \notin s_0$ and return it.
3. $t = \mathbb{0}$: return any element $d \notin s_0$.
4. $t = \mathbb{1}$: impossible.
5. $t = b$: we can choose any element $d \notin [\![b]\!]\eta_i$ and $d \notin s_0$ and return it.
6. $t = \alpha$: we can choose any element $d \notin s_0$ (which clearly implies that $d \notin s_\alpha$), and return it.
7. $t = (t_1 \vee t_2)$: call $\mathsf{explore\_pos}(\neg t_1 \wedge \neg t_2)$ and return it.
8. $t = (t_1 \wedge t_2)$: since this intersection is not $\mathbb{1}$ then one of the two types is not $\mathbb{1}$. If it is t_1 then call $\mathsf{explore\_neg}(t_1)$ and return it else return $\mathsf{explore\_neg}(t_2)$.
9. $t = \neg t'$, then we call $\mathsf{explore\_pos}(t')$ and return it.

Let $d = \mathsf{explore\_pos}(t'_1 \times t'_2)$. Since $[\![t'_1 \times t'_2]\!]\eta_i \neq \emptyset$, then the call is well defined. Then set $s_0 = s_0 \cup \{d\}$ and $s_{\alpha^1_j} = s_{\alpha^1_j} \cup \{d\}$ for all $j \in J_3$.

Finally there is the case in which also $|J_2| > 0$, that is, there exists at least one toplevel negative product type. Since we have

$$(t_1 \times t_2) \wedge \neg(t_3 \times t_4) = (t_1 \backslash t_3 \times t_2) \vee (t_1 \times t_2 \backslash t_4)$$

then we can do the construction as above either for $(t_1 \setminus t_3 \times t_2)$ or $(t_1 \times t_2 \setminus t_4)$: multiple negative product types are treated in the same way.

Subcase 3: $\mathsf{dnf}(t_i) = \bigwedge_{j\in J_1}(t_j^1 \to t_j^2) \wedge \bigwedge_{j\in J_2} \neg(t_j^3 \to t_j^4) \wedge \bigwedge_{j\in J_3} \alpha_j^1 \wedge \bigwedge_{j\in J_4} \neg\alpha_j^2$, where J_i's are finite sets. If $|J_1| = |J_2| = 0$, then we are in the case of Subcase 1. If $|J_1| = 0$ and $|J_2| \neq 0$, then we can do the construction as Subcase 1 since $\mathscr{C} \subseteq [\![\bigwedge_{j\in J_2} \neg(t_j^3 \to t_j^4)]\!]\eta_i$. Therefore let us suppose that $|J_1| \neq 0$. The remaining two cases, that is, $|J_2| = 0$ and $|J_2| \neq 0$, deserve to be treated separately:

$|J_2| = 0$**:** In this case we have at toplevel an intersection of arrows and no negated arrow. Notice that for all $j \in J_1$ we have $\mathbb{1}\to\mathbb{0} \leq t_j^1 \to t_j^2$, therefore we deduce that $\mathbb{1}\to\mathbb{0} \leq \bigwedge_{j\in J_1}(t_j^1 \to t_j^2)$. Since $\mathbb{1}\to\mathbb{0}$ is a closed type (actually, an indivisible one) and $[\![\_]\!]$ is with infinite support, then the denotation of $\mathbb{1}\to\mathbb{0}$ contains infinitely many elements. Since s_0 is finite, then it is possible to choose a d in the denotation of $\mathbb{1}\to\mathbb{0}$ such that $d \notin s_0$. Once we have chosen such a d, we proceed as before, namely, we set $s_0 = s_0 \cup \{d\}$ and similarly add d to s_α for every variable α occurring positively at the top-level of t_i (*i.e.*, for all α_j^1 with $j \in J_3$).

$|J_2| \neq 0$**:** This case cannot be solved as for $|J_2| = 0$, insofar as we can no longer find a closed type that is contained in $\bigwedge_{j\in J_1}(t_j^1 \to t_j^2) \wedge \bigwedge_{j\in J_2} \neg(t_j^3 \to t_j^4)$: since we have at least one negated arrow type, then $\mathbb{1} \to \mathbb{0}$ is no longer contained in the intersection. The only solution is then to build a particular element in this intersection in the same way we did for product types in Subcase 2. Unfortunately, contrary to the case of product types, we cannot work directly on the interpretation function $[\![\_]\!]$ since we do not know its definition on arrow types. However, since we are in a model, we know its behavior with respect to its associated extensional interpretation $\mathbb{E}_{[\![]\!]}$, namely, that for every assignment η and type t it holds $[\![t]\!]\eta = \emptyset \iff \mathbb{E}(t)\eta = \emptyset$. Since we supposed that there exist n assignments η_i such that $[\![t_i]\!]\eta_i \neq \emptyset$ (for $i \in [1..n]$), then the model condition implies that for these same assignments $\mathbb{E}(t_i)\eta_i \neq \emptyset$. If from this we can prove that there exists an assignment η' such that for all $i \in [1..n]$, $\mathbb{E}(t_i)\eta' \neq \emptyset$, then by the model condition again we can deduce that for all $i \in [1..n]$, $[\![t_i]\!]\eta' \neq \emptyset$, that is our thesis.[6]

Consider $\bigwedge_{j\in J_1}(t_j^1 \to t_j^2) \wedge \bigwedge_{j\in J_2} \neg(t_j^3 \to t_j^4)$. By hypothesis we have

$$\mathbb{E}(\bigwedge_{j\in J_1}(t_j^1 \to t_j^2) \wedge \bigwedge_{j\in J_2} \neg(t_j^3 \to t_j^4))\eta_i \neq \emptyset.$$

By definition of $\mathbb{E}$ this is equivalent to

$$\bigcap_{j\in J_1}([\![t_j^1]\!]\eta_i \to [\![t_j^2]\!]\eta_i) \cap \bigcap_{j\in J_2} \neg([\![t_j^3]\!]\eta_i \to [\![t_j^4]\!]\eta_i) \neq \emptyset,$$

[6] As an aside, notice we could have used this technique also in other cases and by the very definition of $\mathbb{E}$ the proof would not have changed (apart from an initial reference to $\mathbb{E}$ at the beginning of each subcase as the one preceding this footnote). Actually, strictly speaking, we already silently used this technique in the case of products since the hypothesis of well-foundedness of model does not state that $[\![t_1 \times t_2]\!]\eta$ is equal to $[\![t_1]\!]\eta \times [\![t_2]\!]\eta$ (an assumption we implicitly did all the proof long) but just that induces the same subtyping relation as a model in which that equality holds. We preferred not to further complicate the presentation of that case.

or equivalently,

$$\bigcap_{j\in J_1} \mathscr{P}(\overline{[\![t_j^1]\!]\eta_i \times \overline{[\![t_j^2]\!]\eta_i}}) \cap \bigcap_{j\in J_2} \neg\mathscr{P}(\overline{[\![t_j^3]\!]\eta_i \times \overline{[\![t_j^4]\!]\eta_i}}) \neq \emptyset$$

We want to construct an assignment η' and a set of pairs such that this set of pairs is included in the intersection above. We then use this set of pairs to define our assignment η'. According to Lemma 2.1.9, the intersection above is not empty if and only if

$$\forall j_2 \in J_2 \,.\, \exists J' \subseteq J_1 \,.\, \begin{cases} [\![t_{j_2}^3 \setminus \bigvee_{j_1\in J'} t_{j_1}^1]\!]\eta_i \neq \emptyset & \text{if } J_1 = J' \\ \begin{cases} [\![t_{j_2}^3 \setminus \bigvee_{j_1\in J'} t_{j_1}^1]\!]\eta_i \neq \emptyset \\ \text{and} \\ [\![\bigwedge_{j_1\in J_1\setminus J'} t_{j_1}^2 \setminus t_{j_2}^4]\!]\eta_i \neq \emptyset \end{cases} & \text{otherwise} \end{cases}$$

Therefore, consider each $j_2 \in J_2$ and let J^{j_2} denote a subset $J' \subseteq J_1$ for which the property above holds. Then we proceed as we did in the Subcase 2 and use explore_pos to build *two* elements $d_{j_2}^1$ and $d_{j_2}^2$. More precisely, if $J^{j_2} \neq J_1$ then we set

$$\begin{cases} d_{j_2}^1 = \mathsf{explore\_pos}(t_{j_2}^3 \setminus \bigvee_{j_1\in J^{j_2}} t_{j_1}^1) \\ d_{j_2}^2 = \mathsf{explore\_pos}(\bigwedge_{j_1\in J_1\setminus J^{j_2}} t_{j_1}^2 \setminus t_{j_2}^4) \end{cases}$$

Otherwise (*i.e.*, $J^{j_2} = J_1$), we set

$$\begin{cases} d_{j_2}^1 = \mathsf{explore\_pos}(t_{j_2}^3 \setminus \bigvee_{j_1\in J_1} t_{j_1}^1) \\ d_{j_2}^2 = \mathsf{explore\_pos}(\neg t_{j_2}^4) \text{ or } \Omega \end{cases}$$

We add $d_{j_2}^1, d_{j_2}^2$, and $(d_{j_2}^1, d_{j_2}^2)$ to s_0. Now consider the various pairs of the form $(d_{j_2}^1, d_{j_2}^2)$ for $j_2 \in J_2$. Since we chose $d_{j_2}^1 \notin [\![\bigvee_{j_1\in J^{j_2}} t_{j_1}^1]\!]\eta_i$, then $(d_{j_2}^1, d_{j_2}^2) \in [\![t_j^1 \to t_j^2]\!]$ for all $j \in J_1$, and therefore it belongs to the intersection $\bigcap_{j\in J_1}[\![t_j^1 \to t_j^2]\!]\eta_i$. Furthermore, by construction each $(d_{j_2}^1, d_{j_2}^2) \notin [\![t_{j_2}^3 \to t_{j_2}^4]\!]\eta_i$. Therefore the set of pairs $\{(d_{j_2}^1, d_{j_2}^2) \mid j_2 \in J_2\}$ is the element we were looking for: we add $\{(d_{j_2}^1, d_{j_2}^2) \mid j_2 \in J_2\}$ to s_0 and to each $s_{\alpha_j^1}$ for $j \in J_3$.

Subcase 4: The previous subcases cover all the cases in which all the literals of the single normal form at issue are on the same constructor (all basic or product or arrow types). So the only remaining subcase is the one in which there are literals with different constructors. This is quite straightforward because it is always possible to reduce the problem to one of the previous cases. More precisely

1. The case in which there are two positive literals with different constructors is impossible since the type would be empty (*e.g.*, the intersection of a basic and a product type is always empty), contradicting our hypothesis.

2. Suppose t_i contains some positive literals all on the same constructor. Then we can erase all the negative literals with a different constructor since they contain all the positive ones, thus reducing the problem to one of the previous subcases.
3. Suppose that t_i contains no positive literal on any constructor, that is it is formed only by negated literals on some constructors. Since the type is not empty, then either the union of all negated basic types does not cover $\mathscr{C}$, or the union of all negated product types does not cover $\mathbb{1} \times \mathbb{1}$, or the union of all negated arrow types does not cover $\mathbb{0} \to \mathbb{1}$. In the first case take as d any element of $\mathscr{C}$ that is neither in s_0 nor in the union of all negated basic types. In the second case, keep just the negated product types, intersect them with $\mathbb{1} \times \mathbb{1}$ and proceed as in Subcase 2. In the third case keep just the negated arrow types, intersect them with $\mathbb{0} \to \mathbb{1}$ and proceed as in Subcase 3.

At the end of this construction we define a new semantic assignment η' as follows $\eta' = \{s_\alpha/\alpha, \ldots\}$ for $\alpha \in \bigcup_{i\in\{1,\ldots,n\}} \mathsf{var}(t_i)$. By construction of η' we have $[\![t_i]\!]\eta' \neq \emptyset$ for each $i \in [1..n]$, which contradicts the premise.

Case 2: There exists $i \in \{1, \ldots, n\}$ such that $t_i = t_i^1 \vee \ldots \vee t_i^m$ while all other t_j's are single normal forms (for all $j \neq i$).
Form Definition 4.2.2, we have

$$[\![t_i]\!]\eta = \emptyset \Longleftrightarrow ([\![t_i^1]\!]\eta = \emptyset) \text{ and } \ldots \text{ and } ([\![t_i^m]\!]\eta = \emptyset)$$

Since

$$\forall \eta \in \mathscr{P}(\mathcal{D})^{\mathcal{V}} \,.\, ([\![t_1]\!]\eta = \emptyset) \text{ or } \ldots \text{ or } ([\![t_i]\!]\eta = \emptyset) \text{ or } \ldots \text{ or } ([\![t_n]\!]\eta = \emptyset)$$

Then consider each t_i^j, we have

$$\forall \eta \in \mathscr{P}(\mathcal{D})^{\mathcal{V}} \,.\, ([\![t_1]\!]\eta = \emptyset) \text{ or } \ldots \text{ or } ([\![t_i^j]\!]\eta = \emptyset) \text{ or } \ldots \text{ or } ([\![t_n]\!]\eta = \emptyset)$$

By Case 1, we have

$$\begin{aligned}&(\forall \eta \in \mathscr{P}(\mathcal{D})^{\mathcal{V}} \,.\, [\![t_1]\!]\eta = \emptyset) \text{ or } \ldots \text{ or } (\forall \eta \in \mathscr{P}(\mathcal{D})^{\mathcal{V}} \,.\, [\![t_i^j]\!]\eta = \emptyset)\\ &\text{ or } \ldots \text{ or } (\forall \eta \in \mathscr{P}(\mathcal{D})^{\mathcal{V}} \,.\, [\![t_n]\!]\eta = \emptyset)\end{aligned}$$

If there exists one type t_k such that $\forall \eta \in \mathscr{P}(\mathcal{D})^{\mathcal{V}} \,.\, [\![t_k]\!]\eta = \emptyset$ holds, where $k \in \{1, \ldots, n\} \setminus \{i\}$, then the result follows. Otherwise, we have

$$\begin{aligned}&(\forall \eta \in \mathscr{P}(\mathcal{D})^{\mathcal{V}} \,.\, [\![t_i^1]\!]\eta = \emptyset) \text{ and } \ldots \text{ and } (\forall \eta \in \mathscr{P}(\mathcal{D})^{\mathcal{V}} \,.\, [\![t_i^m]\!]\eta = \emptyset)\\ \Longleftrightarrow &\forall \eta \in \mathscr{P}(\mathcal{D})^{\mathcal{V}} \,.\, ([\![t_i^1]\!]\eta = \emptyset) \text{ and } \ldots \text{ and } ([\![t_i^m]\!]\eta = \emptyset)\\ \Longleftrightarrow &\forall \eta \in \mathscr{P}(\mathcal{D})^{\mathcal{V}} \,.\, [\![t_i]\!]\eta = \emptyset\end{aligned}$$

Therefore the result follows.

Other cases: If no t_i matches the first two cases, then we proceed similarly to Case 2. We decompose one of the types, say, t_1, then by Case 2 either one of the other types is empty, or all the decompositions of t_1 are empty, then t_1 is empty. □

Finally, it just remains to prove Theorem 4.6.2, that is to say, that Lemma 4.6.5 above holds also for t_i's with recursive products. This result requires the following preliminary lemma.

Lemma 4.6.6. *Let $[\![\_]\!]$ be a well-founded model with infinite support and t a type (which may thus contain infinite product types). If there exists an element d and an assignment $\overline{\eta}$ such that $d \in [\![t]\!]\overline{\eta}$, then there exists a type $t^{\mathsf{fp}} \in \mathcal{T}^{\mathsf{fp}}$ such that $d \in [\![t^{\mathsf{fp}}]\!]\overline{\eta}$ and for all assignment η if $[\![t]\!]\eta = \emptyset$, then $[\![t^{\mathsf{fp}}]\!]\eta = \emptyset$.*

Proof. Since the model is well founded, then by Definition 4.2.8 we can use a well-founded preorder $\blacktriangleright$ on the elements d of the domain $\mathcal{D}$. Furthermore, since our types are regular, then there are just finitely many distinct subtrees of t that are product types. So we proceed by induction on $\blacktriangleright$ and the number n of distinct subtrees of t of the form $t_1 \times t_2$ that do not belong to $\mathcal{T}^{\mathsf{fp}}$. If $n = 0$, then t already belongs to $\mathcal{T}^{\mathsf{fp}}$. Suppose that $t = t_1 \times t_2 \notin \mathcal{T}^{\mathsf{fp}}$. Then the element d of the statement is a pair, that is, $d = (d_1, d_2)$. By induction hypothesis on d_1, d_2, there exist $t_1^{\mathsf{fp}}, t_2^{\mathsf{fp}} \in \mathcal{T}^{\mathsf{fp}}$ such that $d_i \in [\![t_i^{\mathsf{fp}}]\!]\overline{\eta}$ and for all η, $[\![t_i]\!]\eta = \emptyset \Rightarrow [\![t_i^{\mathsf{fp}}]\!]\eta = \emptyset$, for $i = 1, 2$. Then take $t^{\mathsf{fp}} = t_1^{\mathsf{fp}} \times t_2^{\mathsf{fp}}$ and the result follows. Finally, if the product at issue is not at toplevel then we can choose any (recursive) product subtree in t and we have two cases. Either the product does not "participate" to the non emptiness of $[\![t]\!]\overline{\eta}$ (*e.g.*, it occurs in a union addendum that is empty) and then it can be replaced by any type. Or we can decompose d to arrive to a d' that corresponds to the product subtree at issue, and then apply the induction hypothesis as above. In both cases we removed one of the distinct product subtrees that did not belong to $\mathcal{T}^{\mathsf{fp}}$ and the result follows by induction on n. □

While the statement of the previous lemma may, at first sight, seem obscure, its meaning is rather obvious. It states that in a well-founded model (*i.e.*, a model in which all the values are finite) whenever a recursive (product) type contains some value, then we can find a *finite* expansion of this type that contains the same value; furthermore, if the recursive type is empty in a given assignment, then also its finite expansion is empty in that assignment. This immediately yields our final result.

Lemma 4.6.7. *Let $(\mathcal{D}, [\![\_]\!])$ be a well-founded model with infinite support, and t_i for $i \in [1..n]$. Then*

$$\forall \eta \,.\, ([\![t_1]\!]\eta = \emptyset) \text{ or } \ldots \text{ or } ([\![t_n]\!]\eta = \emptyset) \Longleftrightarrow (\forall \eta \,.\, [\![t_1]\!]\eta = \emptyset) \text{ or } \ldots \text{ or } (\forall \eta \,.\, [\![t_n]\!]\eta = \emptyset)$$

Proof. The $\Leftarrow$ direction is trivial. For the other direction, by Lemma 4.6.5 we know that if $t_i \in \mathcal{T}^{\mathsf{fp}}$ for $i \in [1..n]$ then it holds. Suppose by contradiction, that the result does not hold. Then there exists η_i' such that $[\![t_i]\!]\eta_i' \neq \emptyset$ for all $i \in [1..n]$. Let $d_i \in [\![t_i]\!]\eta_i'$, we can apply Lemma 4.6.6 and find $t_i' \in \mathcal{T}^{\mathsf{fp}}$ such that $[\![t_i']\!]\eta_i' \neq \emptyset$. Then by Lemma 4.6.5 we know that there exists an assignment η' such that $[\![t_1']\!]\eta' \neq \emptyset$ and $\ldots$ and $[\![t_n']\!]\eta' \neq \emptyset$. Applying Lemma 4.6.6 again, we deduce that $[\![t_1]\!]\eta' \neq \emptyset$ and $\ldots$ and $[\![t_n]\!]\eta' \neq \emptyset$, which contradicts the premise. □

Corollary 4.6.8 (Convex model). *There exists a convex model.*

Proof. It suffices to take any model for the ground types with an infinite domain (see [Fri04] for examples), interpret indivisible types into infinite sets, and then add a semantic assignment η for type variables to the interpretation function as a further parameter. For instance, imagine we have n basic types b_1, ..., b_n and suppose, for simplicity, that they are pairwise disjoint. If we use the "universal model" of [FCB08] it yields (roughly, without the modifications for Ω) the following model. $\mathcal{D} = \mathscr{C} + \mathcal{D}^2 + \mathscr{P}_f(\mathcal{D}^2)$ where $\mathscr{C} = S_0 \cup S_1 \cup ... \cup S_n$ with S_i are pairwise disjoint infinite sets:

$$\begin{array}{cc} [\![\mathbb{0}]\!]=\emptyset & [\![\mathbb{1}]\!]=\mathcal{D} \\ [\![\neg t]\!]\eta=\mathcal{D}\setminus[\![t]\!] & [\![b_i]\!]=S_i \\ [\![t_1\vee t_2]\!]=[\![t_1]\!]\cup[\![t_2]\!] & [\![t_1\times t_2]\!]=[\![t_1]\!]\times[\![t_2]\!] \\ [\![t_1\wedge t_2]\!]=[\![t_1]\!]\cap[\![t_2]\!] & [\![t_1\to t_2]\!]=\mathscr{P}_f([\![t_1]\!]\times\overline{[\![t_2]\!]})\cup S_0 \end{array}$$

Notice that all denotations of arrow types contain S_0 thus, in particular, the indivisible type $\mathbb{1}\to\mathbb{0}$, too. If the basic types are not pairwise disjoint then it suffice to take for $\mathscr{C}$ a set of S_i whose intersections correspond to those of the corresponding basic types. The only requirement is that all intersections must be infinite sets, as well. For the assignments, one can use a labeling technique similar to the one in Section 5.1: to label elements by (finite sets of) type variables such that $d \in \eta(\alpha)$ if and only if d is labeled by α. □

All the development in this section is generic in the particular type constructors considered: we have seen that the results hold not only for the the full system but also for the propositional logic. However, Christine Paulin noticed that in the presence of product types the above result holds also under weaker hypotheses: thanks to products, requiring that (4.8) holds just for $n = 2$ is enough, since it implies (4.8) for all n. This is is formally stated by the following theorem, whose proof is due to Christine.

Theorem 4.6.9 (Paulin). *If products are included in the system, then*

$$\forall\eta\,.\,([\![t_1]\!]\eta=\emptyset)\ \textit{or}\ ([\![t_2]\!]\eta=\emptyset)\iff(\forall\eta\,.\,[\![t_1]\!]\eta=\emptyset)\ \textit{or}\ (\forall\eta\,.\,[\![t_2]\!]\eta=\emptyset)$$

implies that for all $n \geq 2$

$$\forall\eta\,.\,([\![t_1]\!]\eta=\emptyset)\ \textit{or}\ ...\ \textit{or}\ ([\![t_n]\!]\eta=\emptyset)\iff(\forall\eta\,.\,[\![t_1]\!]\eta=\emptyset)\ \textit{or}\ \ldots\ \textit{or}\ (\forall\eta\,.\,[\![t_n]\!]\eta=\emptyset)$$

Proof. By induction on n. For $n = 2$ the result is straightforward. For $n > 2$ notice that by the extensional interpretation of products, for a given η we have

$$([\![t_1]\!]\eta=\emptyset)\ \text{or}\ ...\ \text{or}\ ([\![t_n]\!]\eta=\emptyset)\iff[\![t_1\times...\times t_n]\!]\eta=\emptyset.$$

Then

$$\begin{array}{lll} & \forall\eta.\ ([\![t_1]\!]\eta=\emptyset)\ \text{or}\ ...\ \text{or}\ ([\![t_n]\!]\eta=\emptyset) & \\ \iff & \forall\eta.\ ([\![t_1\times...\times t_{n-1}]\!]\eta=\emptyset)\ \text{or}\ ([\![t_n]\!]\eta=\emptyset) & \text{products} \\ \iff & (\forall\eta.\ [\![t_1\times...\times t_{n-1}]\!]\eta=\emptyset)\ \text{or}\ (\forall\eta.\ [\![t_n]\!]\eta=\emptyset) & \text{convexity for } n=2 \\ \iff & (\forall\eta.\ ([\![t_1]\!]\eta=\emptyset)\ \text{or}\ ...\ \text{or}\ ([\![t_{n-1}]\!]\eta=\emptyset))\ \text{or}\ (\forall\eta.\ [\![t_n]\!]\eta=\emptyset) & \text{products} \\ \iff & (\forall\eta.\ [\![t_1]\!]\eta=\emptyset)\ \text{or}\ ...\ \text{or}\ (\forall\eta.[\![t_{n-1}]\!]\eta=\emptyset)\ \text{or}\ (\forall\eta.[\![t_n]\!]\eta=\emptyset) & \text{induction} \end{array}$$

□

Chapter 5

A Practical Subtyping Algorithm

This chapter describes a subtyping algorithm that, contrary to the algorithm of Chapter 3, does not perform any type substitution and that generates a counterexample and a counter assignment whenever the subtyping check fails. Thus this algorithm is better fitted to be used in practice.

5.1 An algorithm without substitutions

The subtyping algorithm presented in Chapter 3 is general insofar as it works for any subtyping relation induced by a well-founded convex model. Furthermore it is correct also for types that are not regular. Its definition, however, requires type substitutions, which make the algorithm involved both computationally (efficient implementation of substitutions is challenging) and theoretically (*e.g.*, it is the presence of substitutions that makes the decidability of subtyping hard to prove: see Section 4.5). In this section we show that if we restrict our attention to some particular models (those with infinite support), then it is possible to avoid substitutions. This is inspired by the construction used in the proof of Lemma 4.6.5 and by Gesbert, Genevès, and Layaïda's work [GGL11] which encodes the subtyping relation induced by a convex model with infinite support into a tree logic and uses a satisfiability solver to efficiently decide it without resorting to substitutions.

Let us first recall why the algorithm uses substitutions. Consider a type $\alpha \wedge (t \times s)$. If it is not empty then α must contain some non empty product type. In other words, only product types may make this type non-empty. Thus the idea underlying the algorithm of the previous chapter is that we can just focus on product types and to that end we substitute α by $(\alpha_1 \times \alpha_2)$ (see Lemma 4.3.7 for detail). A similar reasoning applies for arrow types. Hence, the subtyping algorithm works only at type level trying to reduce the emptiness of a type to the emptiness of simpler (or memoized) types.

The algorithm of Gesbert, Genevès, and Layaïda [GGL11] instead works at the level of values by using refutation: to prove that a type is empty it tries to build a value of that type —*ie*, it tries to prove that the type is not empty— and, if it fails to do so, it deduces emptiness. Let us transpose such a reasoning in our setting by considering the single normal form $\alpha \wedge (t \times s)$. From the point of view of values this type is not empty if and only if there exists an element d and an assignment η such

that $d \in [\![\alpha \wedge (t \times s)]\!]\eta$, that is $d \in \eta(\alpha)$ and $d \in [\![(t \times s)]\!]\eta$. This suggests that a possible procedure to "refute" the emptiness of the previous type is to build an element d that belongs to $(t \times s)$ and then assign d to α. Of course, this may not work in general because α may occur in $(t \times s)$ and assigning d to α may make $(t \times s)$ empty. However, in every *well-founded* (convex) model with *infinite support*, this does hold: we can build an element for $(t \times s)$ inductively and during the process we can always pick a fresh appropriate element for each subtree (including type variables) of $(t \times s)$ and avoid the conflict with the assignment for α (see Lemma 4.6.5 for details). In conclusion in a well-founded convex model with infinite support $\alpha \wedge (t \times s)$ is empty if and only if $(t \times s)$ is empty. Therefore in such a setting emptiness can be checked without performing any type substitution.

It is important to stress that well-founded convex models with infinite support not only are the setting used by Gesbert, Genevès, and Layaïda's algorithm (which explains why they can achieve optimal complexity), but also that, so far, they are the only model we know how to use in practice. Therefore it is sensible to define the following version of the algorithm simplified for the case of models with infinite support:

Step 1: *transform the subtyping problem into an emptiness decision problem;*

Step 2: put the type whose emptiness is to be decided in normal form;

Step 3: *simplify mixed intersections;*

Step 4+5: discard all toplevel variables;

Step 6: eliminate toplevel constructors, memoize, and recurse.

This algorithm is the same as the one presented in Section 3.4 except for `Step 4` and `5` which are now merged together to simply discard the toplevel variables independently from their polarities (simply note that after ***Step 3*** a type variable and its negation cannot occur simultaneously at the top level of the type). In detail, at this stage we have to decide the emptiness of intersections of the form

$$\bigwedge_{i \in I} \boldsymbol{a}_i \wedge \bigwedge_{j \in J} \neg \boldsymbol{a}'_j \wedge \bigwedge_{h \in H} \alpha_h \wedge \bigwedge_{k \in K} \neg \beta_k$$

where all the $\boldsymbol{a}_i$'s and $\boldsymbol{a}'_j$'s are atoms with the same constructor, and $\{\alpha_h\}_{h \in H}$ and $\{\beta_k\}_{k \in K}$ are disjoint sets of type variables. In a model with infinite support, to decide the emptiness of the whole type is equivalent to decide the emptiness of $\bigwedge_{i \in I} \boldsymbol{a}_i \wedge \bigwedge_{j \in J} \neg \boldsymbol{a}'_j$.

In order to justify `Step 4+5`, we first prove the following lemma which is the general case of `Step 4+5`.

Lemma 5.1.1. *Let $(\mathcal{D}, [\![\_]\!])$ be a well-founded (convex) model with infinite support, t a type and α a type variable. Then*

$$\forall \eta.\ [\![t \wedge \alpha]\!]\eta = \emptyset \iff (\exists t'.\ t \simeq (\neg\alpha) \wedge t') \text{ or } (\forall \eta.\ [\![t]\!]\eta = \emptyset)$$

Proof. **"$\Leftarrow$":** straightforward.

"⇒": Assume that $\exists \eta_0.\ [\![t]\!]\eta_0 \neq \emptyset$. Consider a normal form of t, that is, $\mathsf{dnf}(t) = \bigvee_{i \in I} \tau_i$ where for all i τ_i is a single normal form. Then there must exist at least one τ_i such that $\exists \eta_i.\ [\![\tau_i]\!]\eta_i \neq \emptyset$. Let $I' = \{i \mid \exists \eta_i.\ [\![\tau_i]\!]\eta_i \neq \emptyset\}$. If there exists $i_0 \in I'$ such that $\tau_{i_0} \not\simeq (\neg\alpha) \wedge \tau'$ for any (single) normal form τ', then we invoke the procedure explore_pos defined in the proof of Lemma 4.6.5 to construct an element d for τ_{i_0} (if τ_{i_0} contains infinite product types, by Lemma 4.6.6, the construction works as well). The procedure explore_pos also generates an assignment η_0 for the type variables in $\mathsf{var}(\tau)$. We define η' such that $\eta' = \eta_0 \oplus \{\eta_0(\alpha) \cup \{d\}/\alpha\}$. Clearly, $[\![\tau_{i_0} \wedge \alpha]\!]\eta' \neq \emptyset$, which deduces that $[\![t \wedge \alpha]\!]\eta' \neq \emptyset$, which conflicts with the premise $\forall \eta.\ [\![t \wedge \alpha]\!]\eta = \emptyset$. Therefore, the result follows.
Otherwise, for each $i \in I'$, we have $\tau_i \simeq (\neg\alpha) \wedge \tau_i'$ for some single normal form τ_i'. Moreover, for each $i \in I \setminus I'$, we have $\forall \eta.\ [\![\tau_i]\!]\eta = \emptyset$, and *a fortiori* $\tau_i \simeq (\neg\alpha) \wedge \tau_i$. Then

$$\begin{array}{rl} t \simeq & \bigvee_{i \in I} \tau_i \\ \simeq & \bigvee_{i \in I'} \tau_i \vee \bigvee_{i \in I \setminus I'} \tau_i \\ \simeq & \bigvee_{i \in I'} ((\neg\alpha) \wedge \tau_i') \vee \bigvee_{i \in I \setminus I'} \tau_i \\ \simeq & \bigvee_{i \in I'} ((\neg\alpha) \wedge \tau_i') \vee \bigvee_{i \in I \setminus I'} ((\neg\alpha) \wedge \tau_i) \\ \simeq & (\neg\alpha) \wedge (\bigvee_{i \in I'} \tau_i' \vee \bigvee_{i \in I \setminus I'} \tau_i) \end{array}$$

Let $t' = (\bigvee_{i \in I'} \tau_i' \vee \bigvee_{i \in I \setminus I'} \tau_i)$. We have $t \simeq (\neg\alpha) \wedge t'$. Therefore the result follows as well.

□

Note that the using $\neg\alpha$ instead of α does not change the validity of Lemma 5.1.1, which explains why in this simpler algorithm we no longer need to eliminate the negative top-level type variables first by replacing them with some fresh positive ones.

The justification of `Step 4+5` is given by Lemmas 5.1.2 and 5.1.3.

Lemma 5.1.2. *Let $\leq$ be the subtyping relation induced by a well-founded (convex) model with infinite support, P, N two finite subsets of $\mathscr{A}_{\mathsf{prod}}$ and α an arbitrary type variable.*

$$\bigwedge_{t_1 \times t_2 \in P} (t_1 \times t_2) \wedge \alpha \leq \bigvee_{s_1 \times s_2 \in N} (s_1 \times s_2) \Longleftrightarrow \bigwedge_{t_1 \times t_2 \in P} (t_1 \times t_2) \leq \bigvee_{s_1 \times s_2 \in N} (s_1 \times s_2)$$

Proof. Consequence of Lemma 5.1.1. □

Lemma 5.1.3. *Let $\leq$ be the subtyping relation induced by a well-founded (convex) model with infinite support, P, N two finite subsets of $\mathscr{A}_{\mathsf{fun}}$ and α an arbitrary type variable.*

$$\bigwedge_{t \to s \in P} (t \to s) \wedge \alpha \leq \bigvee_{t' \to s' \in N} (t' \to s') \Longleftrightarrow \bigwedge_{t \to s \in P} (t \to s) \leq \bigvee_{t' \to s' \in N} (t' \to s')$$

Proof. Consequence of Lemma 5.1.1. □

Note that Lemmas 5.1.2 and 5.1.3 deal with only one type variable, but it is trivial to generalize these lemmas to multiple type variables.

To recap, while the algorithm with substitutions is a universal subtyping algorithm that works for any convex model that satisfies Lemmas 4.3.7 and 4.3.8, the subtyping

algorithm without substitutions works only for (convex) models that are well-founded and with infinite support. Moreover, the algorithm with substitutions is somehow more intuitive since it states the kind of types a type variable should be assigned to (*e.g.*, the type variable α in the type $\alpha \wedge (t \times s)$ should be a product type), while it is not easy to understand why in the simpler algorithm type variables that are at toplevel can be eliminated. To convey this intuition first notice that using an infinite support may yield a different subtyping relation. As an example take a closed type that denotes a singleton: for instance $\mathbb{1}\to\mathbb{0}$ which in the standard $\mathbb{C}$Duce model contains only one element, that is, the function $\bot$ that diverges on every argument. In such a model $(\alpha \wedge (\mathbb{1}\to\mathbb{0}) \times \neg\alpha \wedge (\mathbb{1}\to\mathbb{0}))$ is empty (either $\bot$ is in α or it is in $\neg\alpha$) while in a model with infinite support it is not (it suffices to assign α to a strict non-empty subset of the denotation of $\mathbb{1}\to\mathbb{0}$). Also in the first model it is not true that $\alpha \wedge (t \times s)$ is empty if and only if so $(t \times s)$ is. Take $t = \alpha \wedge (\mathbb{1}\to\mathbb{0})$ and $s = \neg\alpha \wedge (\mu x.(\mathbb{1}\to\mathbb{0}) \times x)$. For the sake of simplicity consider a *non* well-founded model in which $\mathbb{1}\to\mathbb{0}$ denotes $\{\bot\}$. In such a model $(t \times s)$ is not empty and contains exactly one element, namely the element d that satisfies $d = (\bot, d)$ (the infinite list of the diverging function). However if we assign α to a set that contains d, then s denotes the empty set and so does the product. So in this model $\alpha \wedge (t \times s)$ is empty although $(t \times s)$ is not. In a model with infinite support also $\alpha \wedge (t \times s)$ is non empty since it contains, for instance, (d_1, d) where d_1 is any element in the denotation of $\mathbb{1}\to\mathbb{0}$ different from $\bot$.

Finally, remember that infinite denotations for non-empty types is just a possible solution to the convexity property and that the simplified algorithm works only in that case. However, we are not aware of any other different convex model. So while the general algorithm with substitutions works also for models without infinite support, we currently do not know any convex model in which the simpler algorithm would not be enough. Finding such a model would have a purely theoretical interests since well-founded convex models with infinite support (see Section 4.6) are to be used in practice and, ergo, so does the simpler algorithm. The latter has several advantages:

- it is more efficient, since no substitution is performed,
- its decidability is easier to prove, since all the candidate types to be checked are subtrees of the original type,
- in case the subtyping relation does not hold it is easier to give counterexamples and counter assignments as we show in the next section.

5.2 Counterexamples and counter assignments

Given a nonempty type t, there exists an assignment η and an element d such that $d \in [\![t]\!]\eta$. Generally, d is called a counterexample of t and η is called a counter assignment of t. One advantage of semantic subtyping is that once such a subtyping relation does not hold, it is able to yield a counterexample and a counter assignment. A simple way to generate a counterexample is to invoke the procedure $\mathsf{explore\_pos}(s \wedge \neg t)$ defined in Lemma 4.6.5, whenever the check of $s \leq t$ fails. But this does some duplicate work, for example, decomposing terms into sub-terms: one time for the subtyping check and one

time for the procedure. Indeed, we can improve the subtyping algorithms by embedding the procedure explore_pos(t) into it.

Compared with the algorithm without substitutions, to generate counterexamples and counter assignments, the algorithm with substitutions needs (i) to consider not only the type variables contained in types but also the fresh type variables introduced by substitutions and (ii) to follow the tracks of substitutions. Therefore, it is easier to generate counterexamples and counter assignments for the algorithm without substitutions. In the following, we only consider the algorithm without substitutions.

If a type is not empty, then there must exist a non-empty single normal form in its normal form. Hence at `Step 2`, a counterexample for a non-empty single normal form is enough. Next, we simplify mixed intersections (`Step 3`) and discard all the toplevel variables (`Step 4+5`). Then at `Step 6`, the single normal form τ to be check is an intersection of same constructors, which has three possible cases:

- $\tau = \bigwedge_{b\in P} b \wedge \bigwedge_{b\in N} \neg b$. If τ is not empty, then any element d that belongs to τ is a counterexample. Here we require the counterexample to be *fresh* to ensure the existence of a counter assignment, that is, the counterexample is not selected (by other types) before. For example, consider $((b \wedge \alpha) \times (b \wedge \neg\alpha))$. If we take a same element d for these two occurrences of b, then it is impossible to construct a counter assignment, since we have to assign d to the first occurrence of α but we can not assign d to the second occurrence of α. Notice that infinite denotation ensures that a fresh element always exists.

- $\tau = \bigwedge_{t_1\times t_2\in P}(t_1 \times t_2) \wedge \bigwedge_{t_1\times t_2\in N} \neg(t_1 \times t_2)$. Then by Lemma 4.3.9, we must check whether for any subset $N' \subseteq N$,

$$\bigwedge_{t_1\times t_2\in P} t_1 \wedge \bigwedge_{t_1\times t_2\in N'} \neg t_1 \leq \mathbb{0} \text{ or } \bigwedge_{t_1\times t_2\in P} t_2 \wedge \bigwedge_{t_1\times t_2\in N\setminus N'} \neg t_2 \leq \mathbb{0}$$

  If τ is not empty, then there exists a subset $N' \subseteq N$ such that

$$\bigwedge_{t_1\times t_2\in P} t_1 \wedge \bigwedge_{t_1\times t_2\in N'} \neg t_1 \not\simeq \mathbb{0} \text{ and } \bigwedge_{t_1\times t_2\in P} t_2 \wedge \bigwedge_{t_1\times t_2\in N\setminus N'} \neg t_2 \not\simeq \mathbb{0}$$

  So two counterexamples, assuming they are d_1 and d_2 will be generated by the subtyping algorithm for these two types respectively. Then the element (d_1, d_2) belongs to $t_1 \times t_2$ for any $t_1 \times t_2 \in P$, while it does not belong to $t_1 \times t_2$ for any $t_1 \times t_2 \in N$. Therefore, (d_1, d_2) is a counterexample for τ.

- $\tau = \bigwedge_{t_1\to t_2\in P}(t_1 \to t_2) \wedge \bigwedge_{t_1\to t_2\in N} \neg(t_1 \to t_2)$. Then following Lemma 4.3.10, we must check whether there exists $t_1^0 \to t_2^0 \in N$ such that for any subset $P' \subseteq P$,

$$t_1^0 \wedge \bigwedge_{t_1\times t_2\in P'} \neg t_1 \leq \mathbb{0} \text{ or } (\bigwedge_{t_1\times t_2\in P\setminus P'} t_2 \wedge \neg t_2^0 \leq \mathbb{0} \text{ and } P \neq P')$$

  If τ is not empty, then for any $t_1^i \to t_2^i \in N$ there exists a subset $P_i' \subseteq P$ such that

$$t_1^i \wedge \bigwedge_{t_1\times t_2\in P_i'} \neg t_1 \not\simeq \mathbb{0} \text{ and } (\bigwedge_{t_1\times t_2\in P\setminus P_i'} t_2 \wedge \neg t_2^i \not\simeq \mathbb{0} \text{ or } P = P_i')$$

Assume the counterexamples generated for the types $t_1^i \wedge \bigwedge_{t_1 \times t_2 \in P_i'} \neg t_1$ and $\bigwedge_{t_1 \times t_2 \in P \setminus P_i'} t_2 \wedge \neg t_2^i$ respectively by the subtyping algorithm are d_1^i and d_2^i (if $P = P_i'$, d_2^i may be Ω). Then d_1^i belongs to t_1^i while d_2^i does not belong to t_2^i, that is, for any element f that belongs to $t_1^i \to t_2^i$, we have $(d_1^i, d_2^i) \notin f$. Moreover, considering $t_1 \to t_2 \in P$, either d_1^i does not belong to t_1 or d_2^i belongs to t_2. So for any element f, if f belongs to $t_1 \to t_2$, so does $\{(d_1^i, d_2^i)\} \cup f$. Therefore, the element $\{(d_1^i, d_2^i) \mid t_1^i \to t_2^i \in N\}$ is a counterexample for τ.

Since d is generated, we then mark it with the set of top-level type variables $\{\alpha_h\}_{h \in H} \cup \{\neg\beta_k\}_{k \in K}$ to record the assignment information. That is, the element d should be assigned to a type variable if the type variable is positive, while it should not if the type variable is negative[1]. Finally, from these marking information, we can easily construct a counter assignment by collecting all possible elements that could be assigned to each type variable respectively.

For example, consider the following subtyping statement, a variant of (3.2).

$$(b \times \alpha) \leq (b \times \neg b) \vee (\alpha \times b) \tag{5.1}$$

where b is a basic type. According to Lemma 4.3.9, this is equivalent to the following statement:

$$\mathsf{and} \begin{cases} b \leq \mathbb{0} \text{ or } \alpha \wedge b \wedge \neg b \leq \mathbb{0} \\ b \wedge \neg b \leq \mathbb{0} \text{ or } \alpha \wedge \neg b \leq \mathbb{0} \\ b \wedge \neg \alpha \leq \mathbb{0} \text{ or } \alpha \wedge b \leq \mathbb{0} \\ b \wedge \neg b \wedge \neg \alpha \leq \mathbb{0} \text{ or } \alpha \leq \mathbb{0} \end{cases}$$

The first, second and fourth statements hold, while the third one does not. And the algorithm generates the marked counterexamples $d_1^{\{\neg\alpha\}}$ and $d_2^{\{\alpha\}}$ for $b \wedge \neg\alpha$ and $\alpha \wedge b$ respectively, where d_1 and d_2 are two fresh elements belonging to b. So a marked counterexample for (5.1) is $(d_1^{\{\neg\alpha\}}, d_2^{\{\alpha\}})$. Then a counter assignment corresponding to this marked counterexample is $\{\{d_2\}/\alpha\}$. Besides, from the marked counterexample, we deduce that d_2 is different from d_1, as it requires that d_1 should not be assigned to α while d_2 should be. That is, b should contains at least two elements. Hence, a byproduct of the mark information is that it is easy for us to deduce at least how many elements a type should contain to make a subtyping statement invalid (*e.g.*, (5.1)) or an instance of convexity hold (*e.g.*, $\forall\eta.\ (\llbracket b \wedge \neg\alpha \rrbracket\eta = \emptyset \text{ or } \llbracket \alpha \wedge b \rrbracket\eta = \emptyset) \Longleftrightarrow (\forall\eta.\ \llbracket b \wedge \neg\alpha \rrbracket\eta = \emptyset) \text{ or } (\forall\eta.\ \llbracket \alpha \wedge b \rrbracket\eta = \emptyset)$).

Take b as `Int`. Following the construction above, we can generate for the subtyping relation

$$(\texttt{Int} \times \alpha) \leq (\texttt{Int} \times \neg\texttt{Int}) \vee (\alpha \times \texttt{Int})$$

a counterexample $(42, 3)$ and its corresponding counter assignment $\eta_0 = \{\{3\}/\alpha\}$. It is easy to see that $(42, 3) \in \llbracket(\texttt{Int} \times \alpha)\rrbracket\eta_0$ but $(42, 3) \notin \llbracket(\texttt{Int} \times \neg\texttt{Int}) \vee (\alpha \times \texttt{Int})\rrbracket\eta_0$.

[1] Another meaning of labeling (representing name subtyping) was referred to Section 3.3 and a similar labeling is introduced by Gesbert *et al.* [GGL11].

Part III

Polymorphic Calculus

Chapter 6

Overview

In this part, we design and study a higher-order functional language that takes full advantage of the new capabilities of the type system defined in Part II, which can then be easily extended to a full-fledged polymorphic functional language for processing XML documents. Namely, the calculus we want to define is a polymorphic version of Core$\mathbb{C}$Duce.

As explained in Section 2.2, we need to define an explicitly typed λ-calculus with intersection types. This is a notoriously hard problem for which no full-fledged solution exists yet: intersection types systems with explicitly typed terms can be counted on the fingers of one hand and none of them is completely satisfactory (see related works in Section 12.7). The solution introduced by Core$\mathbb{C}$Duce is to explicitly type by an intersection of arrows types the whole λ-abstractions instead of just their parameters. However, it is not trivial to extend this solution to parametric polymorphism.

The expressions in Core$\mathbb{C}$Duce are produced by the following grammar[1]:

$$e \ ::= \ c \ \mid \ x \ \mid \ (e,e) \ \mid \ \pi_i(e) \ \mid \ ee \ \mid \ \lambda^{\wedge_{i\in I} s_i\to t_i} x.\ e \ \mid \ e{\in}t\,?\,e : e \qquad (6.1)$$

Let us recall what we presented in the introduction and expand it with more details. The novelty of a polymorphic extension is to allow type variables (ranged over by lower-case Greek letters: α, β, ...) to occur in the types and, thus, in the types labeling λ-abstractions. It becomes thus possible to define the polymorphic identity function as $\lambda^{\alpha\to\alpha}x.x$, while classic "auto-application" term is written as $\lambda^{((\alpha\to\beta)\wedge\alpha)\to\beta}x.xx$. The intended meaning of using a type variable, such as α, is that a (well-typed) λ-abstraction not only has the type specified in its label (and by subsumption all its super-types) but also all the types obtained by instantiating the type variables occurring in the label. So $\lambda^{\alpha\to\alpha}x.x$ has by subsumption both the type $\mathbb{0}\to\mathbb{1}$ (the type of all functions, which is a super-type of $\alpha\to\alpha$) and the type $\neg$`Int`, and by instantiation the types `Int`$\to$`Int`, `Bool`$\to$`Bool`, etc.

The use of instantiation in combination with intersection types has nasty consequences, for if a term has two distinct types, then it has also their intersection type. In the monomorphic case a term can have distinct types only by subsumption and, thus, intersection types are assigned transparently to terms by the type system via subsumption. But in the polymorphic case this is no longer possible: a term can be

[1]For simplicity, here we do not consider recursive functions, which can be added straightforwardly.

typed by the intersection of two distinct instances of its polymorphic type which, in general, are not in any subtyping relation with the latter: for instance, $\alpha{\to}\alpha$ is neither a subtype of $\texttt{Int}{\to}\texttt{Int}$ nor vice-versa, since the subtyping relation must hold for *all possible* instantiations of α (and there are infinitely many instances of $\alpha{\to}\alpha$ that are neither subtype nor super-type of $\texttt{Int}{\to}\texttt{Int}$).

Concretely, if we want to apply the polymorphic identity $\lambda^{\alpha\to\alpha}x.x$ to, say, $\texttt{42}$, then the particular instance obtained by the type substitution $\{\texttt{Int}/\alpha\}$ (denoting the replacement of every occurrence of α by $\texttt{Int}$) must be used, that is $(\lambda^{\texttt{Int}\to\texttt{Int}}x.x)\texttt{42}$. We have thus to *relabel* the type decorations of λ-abstractions before applying them. In implicitly typed languages, such as ML, the relabeling is meaningless (no type decoration is used) while in their explicitly typed counterparts relabeling can be seen as a logically meaningful but computationally useless operation, insofar as execution takes place on type erasures. In the presence of type-case expressions, however, relabeling is necessary since the label of a λ-abstraction determines its type: testing whether an expression has type, say, $\texttt{Int}{\to}\texttt{Int}$ should succeed for the application of $\lambda^{\alpha\to\alpha\to\alpha}x.\lambda^{\alpha\to\alpha}y.x$ to $\texttt{42}$ and fail for its application to $\texttt{true}$. This means that, in Reynolds' terminology, our terms have an *intrinsic* meaning [Rey03].

If we need to relabel some function, then it may be necessary to relabel also its body as witnessed by the following "daffy" —though well-typed— definition of the identity function:[2]

$$(\lambda^{\alpha\to\alpha}x.(\lambda^{\alpha\to\alpha}y.x)x) \tag{6.2}$$

If we want to apply this function to, say, $\texttt{3}$, then we have first to relabel it by applying the substitution $\{\texttt{Int}/\alpha\}$. However, applying the relabeling only to the outer "λ" does not suffice since the application of (6.2) to $\texttt{3}$ reduces to $(\lambda^{\alpha\to\alpha}y.\texttt{3})\texttt{3}$ which is not well-typed (it is not possible to deduce the type $\alpha{\to}\alpha$ for $\lambda^{\alpha\to\alpha}y.\texttt{3}$, which is the constant function that always returns $\texttt{3}$) although it is the reductum of a well-typed application.

The solution is to apply the relabeling also to the body of the function. Here what "to relabel the body" means is straightforward: apply the same type-substitution $\{\texttt{Int}/\alpha\}$ to the body. This yields a reductum $(\lambda^{\texttt{Int}\to\texttt{Int}}y.\texttt{3})\texttt{3}$ which is well typed. In general, however, the way to perform a relabeling is not so straightforward and clearly defined, since two different problems may arise: (1) it may be necessary to apply more than a single type-substitution and (2) the relabeling of the body may depend on the dynamic type of the actual argument of the function (both problems are better known as —or are instances of— the problem of determining expansions for intersection type systems [CDV80]). We discuss each problem in detail.

First of all notice that we may need to relabel/instantiate functions not only when they are applied but also when they are used as arguments. For instance consider a function that expects arguments of type $\texttt{Int}{\to}\texttt{Int}$. It is clear that we can apply it to the identity function $\lambda^{\alpha\to\alpha}x.x$, since the identity function *has* type $\texttt{Int}{\to}\texttt{Int}$ (feed it by an integer and it will return an integer). Before, though, we have to relabel the latter by the substitution $\{\texttt{Int}/\alpha\}$ yielding $\lambda^{\texttt{Int}\to\texttt{Int}}x.x$. As the identity has type $\texttt{Int}{\to}\texttt{Int}$ so it has type $\texttt{Bool}{\to}\texttt{Bool}$ and, therefore, the intersection of the two: $(\texttt{Int}{\to}\texttt{Int})\wedge(\texttt{Bool}{\to}\texttt{Bool})$.

[2] By convention a type variable is introduced by the outer λ in which it occurs and this λ implicitly binds all inner occurrences of the variable. For instance, all the α's in the term (6.2) are the same while in a term such as $(\lambda^{\alpha\to\alpha}x.x)(\lambda^{\alpha\to\alpha}x.x)$ the variables in the function are distinct from those in its argument and, thus, can be α-converted separately, as $(\lambda^{\gamma\to\gamma}x.x)(\lambda^{\delta\to\delta}x.x)$.

So we can apply a function that expects an argument of this intersection type to our identity function. The problem is now how to relabel $\lambda^{\alpha\to\alpha}x.x$. Intuitively, we have to apply two distinct type-substitutions $\{\mathtt{Int}/\alpha\}$ and $\{\mathtt{Bool}/\alpha\}$ to the label of the λ-abstraction and replace it by the intersection of the two instances. This corresponds to relabel the polymorphic identity from $\lambda^{\alpha\to\alpha}x.x$ into $\lambda^{(\mathtt{Int}\to\mathtt{Int})\wedge(\mathtt{Bool}\to\mathtt{Bool})}x.x$. This is the solution adopted by this work, where we manipulate sets of type-substitutions —delimited by square brackets—. The application of such a set (*eg*, in the previous example $[\{\mathtt{Int}/\alpha\},\{\mathtt{Bool}/\alpha\}]$) to a type t, returns the intersection of all types obtained by applying each substitution in set to t (*eg*, in the example $t\{\mathtt{Int}/\alpha\}\wedge t\{\mathtt{Bool}/\alpha\}$). Thus the first problem has an easy solution.

The second problem is much harder and concerns the relabeling of the body of a function since the naive solution consisting of propagating the application of (sets of) substitutions to the bodies of functions fails in general. This can be seen by considering the relabeling via the set of substitutions $[\{\mathtt{Int}/\alpha\},\{\mathtt{Bool}/\alpha\}]$ of the daffy function in (6.2). If we apply the naive solution this yields

$$(\lambda^{(\mathtt{Int}\to\mathtt{Int})\wedge(\mathtt{Bool}\to\mathtt{Bool})}x.(\lambda^{(\mathtt{Int}\to\mathtt{Int})\wedge(\mathtt{Bool}\to\mathtt{Bool})}y.x)x) \tag{6.3}$$

which is not well typed. That this term is not well typed is clear if we try apply it to, say, `3`: the application of a function of type $(\mathtt{Int}\to\mathtt{Int})\wedge(\mathtt{Bool}\to\mathtt{Bool})$ to an `Int` should have type `Int`, but here it reduces to $(\lambda^{(\mathtt{Int}\to\mathtt{Int})\wedge(\mathtt{Bool}\to\mathtt{Bool})}y.3)3$, and there is no way to deduce the intersection type $(\mathtt{Int}\to\mathtt{Int})\wedge(\mathtt{Bool}\to\mathtt{Bool})$ for the constant function $\lambda y.3$. But we can also directly verify that it is not well typed, by trying to type the function in (6.3). This corresponds to prove that under the hypothesis $x:\mathtt{Int}$ the term $(\lambda^{(\mathtt{Int}\to\mathtt{Int})\wedge(\mathtt{Bool}\to\mathtt{Bool})}y.x)x$ has type `Int`, and that under the hypothesis $x:\mathtt{Bool}$ this same term has type `Bool`. Both checks fail because, in both cases, $\lambda^{(\mathtt{Int}\to\mathtt{Int})\wedge(\mathtt{Bool}\to\mathtt{Bool})}y.x$ is ill-typed (it neither has type $\mathtt{Int}\to\mathtt{Int}$ when x:`Bool`, nor has it type $\mathtt{Bool}\to\mathtt{Bool}$ when x:`Int`). This example shows that in order to ensure that relabeling yields well-typed terms, the relabeling of the body *must change* according to the type the parameter x is bound to. More precisely, $(\lambda^{\alpha\to\alpha}y.x)$ should be relabeled as $\lambda^{\mathtt{Int}\to\mathtt{Int}}y.x$ when x is of type `Int`, and as $\lambda^{\mathtt{Bool}\to\mathtt{Bool}}y.x$ when x is of type `Bool`. An example of this same problem less artificial than our daffy function is given by the classic `apply` function $\lambda f.\lambda x.fx$ which, with our polymorphic type annotations, is written as:

$$\lambda^{(\alpha\to\beta)\to\alpha\to\beta}f.\lambda^{\alpha\to\beta}x.fx \tag{6.4}$$

The "apply" function in (6.4) has the type $(\mathtt{Int}\to\mathtt{Int})\to\mathtt{Int}\to\mathtt{Int}$, obtained by instantiating its type annotation by the substitution $\{\mathtt{Int}/\alpha,\mathtt{Int}/\beta\}$, as well as the type $(\mathtt{Bool}\to\mathtt{Bool})\to\mathtt{Bool}\to\mathtt{Bool}$, obtained by the substitution $\{\mathtt{Bool}/\alpha,\mathtt{Bool}/\beta\}$. If we want to feed this function to another function that expects arguments whose type is the intersections of these two types, then we have to relabel it by using the set of substitutions $[\{\mathtt{Int}/\alpha,\mathtt{Int}/\beta\},\{\mathtt{Bool}/\alpha,\mathtt{Bool}/\beta\}]$. But, once more, it is easy to verify that the naive solution that consists in propagating the application of the set of substitutions down to the body of the function yields an ill-typed expression.

This second problem is the showstopper for the definition of an explicitly typed λ-calculus with intersection types. Most of the solutions found in the literature [Ron02, WDMT02, LR07, BVB08] rely on the duplication of lambda terms and/or typing deriva-

tions, while other calculi such as [WH02] that aim at avoiding such duplication obtain it by adding new expressions and new syntax for types (see related work in Section 12.7).

Here we introduce a new technique that consists in performing a "lazy" relabeling of the bodies. This is obtained by decorating λ-abstractions by (sets of) type-substitutions. For example, in order to pass our daffy identity function (6.2) to a function that expects arguments of type $(\mathtt{Int}\to\mathtt{Int}) \wedge (\mathtt{Bool}\to\mathtt{Bool})$ we first "lazily" relabel it as follows:

$$(\lambda^{\alpha\to\alpha}_{[\{\mathtt{Int}/\alpha\},\{\mathtt{Bool}/\alpha\}]}x.(\lambda^{\alpha\to\alpha}y.x)x). \tag{6.5}$$

The annotation in the outer "λ" indicates that the function must be relabeled and, therefore, that we are using the particular instance whose type is the one in the interface in which we apply the set of type-substitutions. The relabeling will be actually propagated to the body of the function at the moment of the reduction, only if and when the function is applied (relabeling is thus lazy). However, the new annotation is statically used by the type system to check soundness. Notice that, contrary to existing solutions, we preserve the structure of λ-terms (at the expenses of some extra annotation) which is of uttermost importance in a language-oriented study.

From a practical point of view it is important to stress that these annotations will be transparent to the programmer and that all necessary relabeling will be inferred statically and compiled away. In practice, the programmer will program in the language defined by the grammar (6.1), where types in the interfaces of λ's may contain type variables. With the language defined by the grammar (6.1), we can define the `map`[3] and `even` functions mentioned in Section 1.4 as

$$\begin{array}{rcl}\mathtt{map} & \stackrel{\text{def}}{=} & \mu m^{(\alpha\to\beta)\to\mathtt{[}\alpha\mathtt{]}\to\mathtt{[}\beta\mathtt{]}}\ \lambda f\,.\,\lambda^{\mathtt{[}\alpha\mathtt{]}\to\mathtt{[}\beta\mathtt{]}}\ell\,.\,\ell{\in}\mathtt{nil}\,?\,\mathtt{nil}:(f(\pi_1\ell),mf(\pi_2\ell))\\ \mathtt{even} & \stackrel{\text{def}}{=} & \lambda^{(\mathtt{Int}\to\mathtt{Bool})\wedge(\alpha\setminus\mathtt{Int}\to\alpha\setminus\mathtt{Int})}x\,.\,x{\in}\mathtt{Int}\,?\,(x \bmod 2)=0:x\end{array}$$

where the type `nil` tested in the type case denotes the singleton type that contains just the constant `nil`, and `[`α`]` denotes the regular type that is the (least) solution of $X = (\alpha, X) \vee \mathtt{nil}$.

When fed by any expression of this language, the type-checker will infer sets of type-substitutions and insert them into the expression to make it well-typed (if possible, of course). For example, for the application of `map` to `even`, we will see that the inference system of Chapter 9 infers one set of type-substitutions

$$[\{\mathtt{Int}/\alpha, \mathtt{Bool}/\beta\}, \{\alpha\vee\mathtt{Int}/\alpha, (\alpha\setminus\mathtt{Int})\vee\mathtt{Bool}/\beta\}]$$

and inserts it between the two terms. Finally, the compiler will compile the expression with the inserted type-substitutions into a monomorphic expression in which all substitutions are compiled away and only necessary relabelings are hard-coded. The rest of the presentation proceeds then in four logical phases:

1. *Definition of a calculus with explicit (sets of) type-substitutions* (Chapter 7). These explicit type-substitutions are used at reduction time to perform the relabeling of the bodies of the applied function. We define a type systems and prove its soundness.

[3]Strictly speaking, the type case in `map` should be a "binding" one, which is introduced in Section 11.1.

2. *Inference of type-substitutions* (Chapters 8– 10). We want to program with the terms defined by the grammar (6.1), and not in a calculus with explicit type-substitutions. Therefore we must define a system that infers where and whether type-substitutions can be inserted in a term to make it well typed. In order to define this system we proceed in three steps.

   First we define a sound and complete typing algorithm for the type system of the calculus with explicit type-substitutions (this reduces to the elimination of the subsumption) (Chapter 8).

   Second, we study this algorithm to deduce where sets of type-substitutions must be inserted and use it as a guide to define a deduction system that infers type-substitutions in order to give a type to the terms of the grammar (6.1)(Chapter 9). This type inference system is sound and complete with respect to the typing algorithm. This is stated in terms of an *erasure* function which maps a term with explicit type-substitutions into a term of grammar (6.1) by erasing all type-substitutions. If the inference systems assign some type to a term without explicit substitutions, then this term is the erasure of a term that has the same type. Conversely if a term with explicit type-substitutions is well typed, then the inference system infers a (more general) type for it.

   The third and final step is to give an effective procedure to "implement" the sound and complete inference system, whose rules use constraints on the existence of some type-substitutions (Chapter 10). We show that these constraints can in general be reduced to deciding whether for two types s and t there exist two sets of substitutions $[\sigma_i]_{i\in I}$ and $[\sigma'_j]_{j\in J}$ such that $\bigwedge_{i\in I} s\sigma_i \leq \bigvee_{j\in J} t\sigma'_j$. We show how to give a sound and complete set of solutions when the cardinalities of I and J are bounded. This immediately yields a semi-decision procedure (that tries all the cardinalities) for the inference system: the procedure infers a type for an implicit term if and only if it is the erasure of a well-typed explicitly-typed term. If a term is not the erasure of any explicitly-typed term, then the inference procedure either fails or it never stops (decidability is still an open question because of union types).
3. *Compilation into a monomorphic calculus* (Chapter 11). Given a well-typed term of the implicitly typed calculus, the inference system effectively produces a term with explicit type-substitutions that corresponds to it. In the last part we show how to compile this explicitly typed term into a monomorphic term of Core$\mathbb{C}$Duce, thus getting rid of explicit substitutions. The target language is Core$\mathbb{C}$Duce where type variables are considered as new basic types and the subtyping relation is the one defined in Chapter 4. The motivation of this translation is to avoid to perform costly relabeling at run-time and to reuse the efficient and optimized run-time engine of $\mathbb{C}$Duce. We show that all possible relabelings can be statically computed and dynamic relabeling compiled away. We prove that the translation preserve both the static and the dynamic semantics of the source calculus, and show it is sufficient to give an implementation of the polymorphic version.
4. *Extensions* (Chapter 12). Finally we discuss some additional features and design choices, such as recursive functions, type substitution application and/or negation arrows for the typing rule of abstractions, open type cases, and so on.

Chapter 7

A Calculus with Explicit Type-Substitutions

In this chapter, we define our explicitly-typed λ-calculus with sets of type-substitutions.

7.1 Expressions

Definition 7.1.1 (Expressions). *Let $\mathscr{C}$ be a set of constants ranged over by c and $\mathcal{X}$ a countable set of expression variables ranged over by x,y,z,.... An* expression *e is a term inductively generated by the following grammar:*

Expressions	e	$::=$	c	*constant*
			$\mid x$	*expression variable*
			$\mid (e, e)$	*pair*
			$\mid \pi_i(e)$	*projection*$(i \in \{1,2\})$
			$\mid \lambda^{\wedge_{i\in I} t_i \to s_i}_{[\sigma_j]_{j\in J}} x.e$	*abstraction*
			$\mid e\ e$	*application*
			$\mid e{\in}t\,?\,e : e$	*type case*
			$\mid e[\sigma_j]_{j\in J}$	*instantiation*

where t_i, s_i range over types, $t \in \mathcal{T}_0$ is a ground type, and $[\sigma_j]_{j\in J}$ is a set of type-substitutions. We write $\mathscr{E}$ to denote the set of all expressions.

A λ-abstraction comes with a non-empty sequence of arrow types (called its *interface*) and a possibly empty set of type-substitutions (called its *decorations*). When the decoration is an empty set, we write $\lambda^{(\wedge_{i\in I} t_i \to s_i)} x.e$ for short. For simplicity, here we do not consider recursive functions, which can be added straightforwardly (see Chapter 12).

Since expressions are inductively generated, then the accessory definitions on them can be given by induction.

Given a set of type variables Δ and a set of type-substitutions $[\sigma_j]_{j\in J}$, for simplicity, we use the notation $\Delta[\sigma_j]_{j\in J}$ to denote the set of type variables occurring in the applications $\alpha\sigma_j$ for all $\alpha \in \Delta, j \in J$, that is:

$$\Delta[\sigma_j]_{j\in J} \stackrel{\text{def}}{=} \bigcup_{j\in J} \Big(\bigcup_{\alpha\in\Delta} \mathsf{var}(\sigma_j(\alpha)) \Big)$$

Definition 7.1.2. *Let e be an expression. The set $\mathsf{fv}(e)$ of* free variables *of the expression e is defined by induction as:*

$$\begin{aligned}
\mathsf{fv}(x) &= \{x\} \\
\mathsf{fv}(c) &= \emptyset \\
\mathsf{fv}((e_1, e_2)) &= \mathsf{fv}(e_1) \cup \mathsf{fv}(e_2) \\
\mathsf{fv}(\pi_i(e)) &= \mathsf{fv}(e) \\
\mathsf{fv}(\lambda^{\wedge_{i\in I} t_i \to s_i}_{[\sigma_j]_{j\in J}} x.e) &= \mathsf{fv}(e) \setminus \{x\} \\
\mathsf{fv}(e_1\ e_2) &= \mathsf{fv}(e_1) \cup \mathsf{fv}(e_2) \\
\mathsf{fv}(e{\in}t\,?\,e_1 : e_2) &= \mathsf{fv}(e) \cup \mathsf{fv}(e_1) \cup \mathsf{fv}(e_2) \\
\mathsf{fv}(e[\sigma_j]_{j\in J}) &= \mathsf{fv}(e)
\end{aligned}$$

The set $\mathsf{bv}(e)$ of bound variables *of the expression e is defined by induction as:*

$$\begin{aligned}
\mathsf{bv}(x) &= \emptyset \\
\mathsf{bv}(c) &= \emptyset \\
\mathsf{bv}((e_1, e_2)) &= \mathsf{bv}(e_1) \cup \mathsf{bv}(e_2) \\
\mathsf{bv}(\pi_i(e)) &= \mathsf{bv}(e) \\
\mathsf{bv}(\lambda^{\wedge_{i\in I} t_i \to s_i}_{[\sigma_j]_{j\in J}} x.e) &= \mathsf{bv}(e) \cup \{x\} \\
\mathsf{bv}(e_1\ e_2) &= \mathsf{bv}(e_1) \cup \mathsf{bv}(e_2) \\
\mathsf{bv}(e{\in}t\,?\,e_1 : e_2) &= \mathsf{bv}(e) \cup \mathsf{bv}(e_1) \cup \mathsf{bv}(e_2) \\
\mathsf{bv}(e[\sigma_j]_{j\in J}) &= \mathsf{bv}(e)
\end{aligned}$$

The set $\mathsf{tv}(e)$ of type variables *occurring in e is defined by induction as:*

$$\begin{aligned}
\mathsf{tv}(x) &= \emptyset \\
\mathsf{tv}(c) &= \emptyset \\
\mathsf{tv}((e_1, e_2)) &= \mathsf{tv}(e_1) \cup \mathsf{tv}(e_2) \\
\mathsf{tv}(\pi_i(e)) &= \mathsf{tv}(e) \\
\mathsf{tv}(\lambda^{\wedge_{i\in I} t_i \to s_i}_{[\sigma_j]_{j\in J}} x.e) &= \mathsf{tv}(e[\sigma_j]_{j\in J}) \cup \mathsf{var}(\textstyle\bigwedge_{i\in I, j\in J} t_i\sigma_j \to s_i\sigma_j) \\
\mathsf{tv}(e_1\ e_2) &= \mathsf{tv}(e_1) \cup \mathsf{tv}(e_2) \\
\mathsf{tv}(e{\in}t\,?\,e_1 : e_2) &= \mathsf{tv}(e) \cup \mathsf{tv}(e_1) \cup \mathsf{tv}(e_2) \\
\mathsf{tv}(e[\sigma_j]_{j\in J}) &= (\mathsf{tv}(e))[\sigma_j]_{j\in J}
\end{aligned}$$

An expression e is closed *if $\mathsf{fv}(e)$ is empty.*

Note that the set of type variables in $e[\sigma_j]_{j\in J}$ is the set of type variables in the "application" of $[\sigma_j]_{j\in J}$ to the set of type variables $\mathsf{tv}(e)$.

As customary, we assume bound expression variables to be pairwise distinct and distinct from any free expression variable occurring in the expressions under consideration. We equate expressions up to the α-renaming of their bound expression variables. In particular, when substituting an expression e for a variable y in an expression e' (see Definition 7.1.3), we assume that the bound variables of e' are distinct from the bound and free variables of e, to avoid unwanted captures. For example, $(\lambda^{\alpha\to\alpha}x.x)y$ is α-equivalent to $(\lambda^{\alpha\to\alpha}z.z)y$.

The situation is a bit more complex for type variables, as we do not have an explicit binder for them. Intuitively, a type variable can be α-converted if it is a *polymorphic* one, that is, if it can be instantiated. For example, $(\lambda^{\alpha\to\alpha}x.x)y$ is α-equivalent

to $(\lambda^{\beta\to\beta}x.x)y$, and $(\lambda^{\alpha\to\alpha}_{[\mathtt{Int}/\alpha]}x.x)y$ is α-equivalent to $(\lambda^{\beta\to\beta}_{[\mathtt{Int}/\beta]}x.x)y$. Polymorphic variables can be bound by interfaces, but also by decorations or applications between them: for example, in $\lambda^{\mathtt{Int}\to\mathtt{Int}}_{[\mathtt{Int}/\alpha]}x.(\lambda^{\alpha\to\alpha}y.y)x$, the α occurring in the interface of the inner abstraction is "bound" by the decoration $[\mathtt{Int}/\alpha]$, and the whole expression is α-equivalent to $(\lambda^{\mathtt{Int}\to\mathtt{Int}}_{[\mathtt{Int}/\beta]}x.(\lambda^{\beta\to\beta}y.y)x$. Another example is that, $\lambda^{\gamma\to\gamma}_{[\alpha/\gamma]}x.(\lambda^{\alpha\to\alpha}y.y)x$ is α-equivalent to $\lambda^{\gamma\to\gamma}_{[\beta/\gamma]}x.(\lambda^{\beta\to\beta}y.y)x$. If a type variable is bound by an outer abstraction, it cannot be instantiated; such a variable is called *monomorphic*. For example, the expression $\lambda^{(\alpha\to\alpha)\to(\alpha\to\alpha)}y.((\lambda^{\alpha\to\alpha}x.x)[\mathtt{Int}/\alpha]y)$ is not sound (i.e., it cannot be typed), because α is bound at the level of the outer abstraction, not at level of the inner one. Consequently, in this expression, α is monomorphic for the inner abstraction, but polymorphic for the outer one. Monomorphic type variables cannot be α-converted: $\lambda^{(\alpha\to\alpha)\to(\alpha\to\alpha)}y.(\lambda^{\alpha\to\alpha}x.x)y$ is not α-equivalent to $\lambda^{(\alpha\to\alpha)\to(\alpha\to\alpha)}y.(\lambda^{\beta\to\beta}x.x)y$ (but it is α-equivalent to $\lambda^{(\beta\to\beta)\to(\beta\to\beta)}y.(\lambda^{\beta\to\beta}x.x)y$). Note that the scope of polymorphic variables may include some type-substitutions $[\sigma_i]_{i\in I}$: for example, $((\lambda^{\alpha\to\alpha}x.x)y)[\mathtt{Int}/\alpha]$ is α-equivalent to $((\lambda^{\beta\to\beta}x.x)y)[\mathtt{Int}/\beta]$. Finally, we have to be careful when performing expression substitutions and type substitutions to avoid clashes of polymorphic variables namespaces. For example, substituting $\lambda^{\alpha\to\alpha}z.z$ for y in $\lambda^{\alpha\to\alpha}x.x\ y$ would lead to an unwanted capture of α (assuming α is polymorphic, i.e., not bound by a λ-abstraction placed above these two expressions), so we have to α-convert one of them, so that the result of the substitution is, for instance, $\lambda^{\alpha\to\alpha}x.x\ (\lambda^{\beta\to\beta}z.z)$.

To resume, we adopt the following conventions on α-conversion for type variables. We assume that polymorphic variables are pairwise distinct and distinct from any monomorphic variable in the expressions under consideration. We equate expressions up to α-renaming of their polymorphic variables. In particular, when substituting an expression e for a variable y in an expression e' (see Definition 7.1.3), we suppose the polymorphic type variables of e' to be distinct from the monomorphic and polymorphic type variables of e to avoid unwanted capture[1].

Definition 7.1.3 (**Expression substitution**). *An* expression substitution ϱ *is a total mapping of expression variables to expressions that is the identity everywhere but on a finite subset of $\mathcal{X}$, which is called the domain of ϱ and denoted by $\mathsf{dom}(\varrho)$. We use the notation $\{e_1/x_1, \ldots, e_n/x_n\}$ to denote the expression substitution that maps x_i into e_i for $i = 1..n$.*

The definitions of free variables, bound variables and type variables are extended to expression substitutions as follows.

$$\mathsf{fv}(\varrho) = \bigcup_{x\in\mathsf{dom}(\varrho)} \mathsf{fv}(\varrho(x)), \quad \mathsf{bv}(\varrho) = \bigcup_{x\in\mathsf{dom}(\varrho)} \mathsf{bv}(\varrho(x)), \quad \mathsf{tv}(\varrho) = \bigcup_{x\in\mathsf{dom}(\varrho)} \mathsf{tv}(\varrho(x))$$

[1] In this discussion, the definitions of the notions of polymorphic and monomorphic variables remain informal. To make them more formal, we would have to distinguish between the two by carrying around a set of type variables Δ which would contain the monomorphic variables that cannot be α-converted. Then all definitions (such as expression substitutions, for example) would have to be parametrized with Δ, making the definitions and technical developments difficult to read just because of α-conversion. Therefore, for the sake of readability, we decided to keep the distinction between polymorphic and monomorphic variables informal.

Next, we define the application of an expression substitution ϱ to an expression e. To avoid unwanted captures, we remind that we assume that the bound variables of e do not occur in the domain of ϱ and that the polymorphic type variables of e are distinct from the type variables occurring in ϱ (using α-conversion if necessary).

Definition 7.1.4. *Given an expression $e \in \mathscr{E}$ and an expression substitution ϱ, the application of ϱ to e is defined as follows:*

$$\begin{array}{rcll}
c\varrho & = & c & \\
(e_1, e_2)\varrho & = & (e_1\varrho, e_2\varrho) & \\
(\pi_i(e))\varrho & = & \pi_i(e\varrho) & \\
(\lambda^{\wedge_{i\in I} t_i \to s_i}_{[\sigma_j]_{j\in J}} x.e)\varrho & = & \lambda^{\wedge_{i\in I} t_i \to s_i}_{[\sigma_j]_{j\in J}} x.(e\varrho) & \\
(e_1\ e_2)\varrho & = & (e_1\varrho)\ (e_2\varrho) & \\
(e \in t\,?\,e_1 : e_2)\varrho & = & e\varrho \in t\,?\,e_1\varrho : e_2\varrho & \\
x\varrho & = & \varrho(x) & \text{if } x \in \mathsf{dom}(\varrho) \\
x\varrho & = & x & \text{if } x \notin \mathsf{dom}(\varrho) \\
(e[\sigma_j]_{j\in J})\varrho & = & (e\varrho)[\sigma_j]_{j\in J} &
\end{array}$$

In the case for instantiation $(e[\sigma_j]_{j\in J})\varrho$, the σ_j operate on the polymorphic type variables, which we assume distinct from the variables in ϱ (using α-conversion if necessary). As a result, we have $\mathsf{tv}(\varrho) \cap \bigcup_{j\in J} \mathsf{dom}(\sigma_j) = \emptyset$. Similarly, in the abstraction case, we have $x \notin \mathsf{dom}(\varrho)$.

To define the relabeling, we need to define the composition of two sets of type-substitutions.

Definition 7.1.5. *Given two sets of type-substitutions $[\sigma_i]_{i\in I}$ and $[\sigma_j]_{j\in J}$, we define their composition as*

$$[\sigma_i]_{i\in I} \circ [\sigma_j]_{j\in J} \stackrel{\text{def}}{=} [\sigma_i \circ \sigma_j]_{i\in I, j\in J}$$

where

$$\sigma_i \circ \sigma_j(\alpha) = \begin{cases} (\sigma_j(\alpha))\sigma_i & \text{if } \alpha \in \mathsf{dom}(\sigma_j) \\ \sigma_i(\alpha) & \text{if } \alpha \in \mathsf{dom}(\sigma_i) \setminus \mathsf{dom}(\sigma_j) \\ \alpha & \text{otherwise} \end{cases}$$

Next, we define the relabeling of an expression e with a set of type substitutions $[\sigma_j]_{j\in J}$, which consists in propagating the σ_j to the λ-abstractions in e if needed. We suppose that the polymorphic type variables in e are distinct from the type variables in the range of σ_j (this is always possible using α-conversion).

Definition 7.1.6 (Relabeling). *Given an expression $e \in \mathscr{E}$ and a set of type-substitutions $[\sigma_j]_{j\in J}$, we define the* relabeling *of e with $[\sigma_j]_{j\in J}$, written $e@[\sigma_j]_{j\in J}$, as e if $\mathsf{tv}(e) \cap \bigcup_{j\in J} \mathsf{dom}(\sigma_j) = \emptyset$, and otherwise as follows:*

$$\begin{array}{rcl}
(e_1, e_2)@[\sigma_j]_{j\in J} & = & (e_1@[\sigma_j]_{j\in J}, e_2@[\sigma_j]_{j\in J}) \\
(\pi_i(e))@[\sigma_j]_{j\in J} & = & \pi_i(e@[\sigma_j]_{j\in J}) \\
(e_1\ e_2)@[\sigma_j]_{j\in J} & = & (e_1@[\sigma_j]_{j\in J})\ (e_2@[\sigma_j]_{j\in J}) \\
(e \in t\,?\,e_1 : e_2)@[\sigma_j]_{j\in J} & = & e@[\sigma_j]_{j\in J} \in t\,?\,e_1@[\sigma_j]_{j\in J} : e_2@[\sigma_j]_{j\in J} \\
(e[\sigma_i]_{i\in I})@[\sigma_j]_{j\in J} & = & e@([\sigma_j]_{j\in J} \circ [\sigma_i]_{i\in I}) \\
(\lambda^{\wedge_{i\in I} t_i \to s_i}_{[\sigma_k]_{k\in K}} x.e)@[\sigma_j]_{j\in J} & = & \lambda^{\wedge_{i\in I} t_i \to s_i}_{[\sigma_j]_{j\in J} \circ [\sigma_k]_{k\in K}} x.e
\end{array}$$

The substitutions are not propagated if they do not affect the variables of e (i.e., if $\mathsf{tv}(e) \cap \bigcup_{j\in J} \mathsf{dom}(\sigma_j) = \emptyset$). In particular, constants and variables are left unchanged, as they do not contain any type variable. Suppose now that $\mathsf{tv}(e) \cap \bigcup_{j\in J} \mathsf{dom}(\sigma_j) \neq \emptyset$. In the abstraction case, the propagated substitutions are composed with the decorations of the abstraction, without propagating them further down in the body. Propagation in the body occurs, whenever is needed, that is, during either reduction (see (*Rappl*) in Section 7.3) or type-checking (see (*abstr*) in Section 7.2). In the instantiation case $e[\sigma_i]_{i\in I}$, we propagate the result of the composition of $[\sigma_i]_{i\in I}$ with $[\sigma_j]_{j\in J}$ in e. The remaining cases are simple inductive cases. Finally notice that in a type case $e{\in}t\,?\,e_1 : e_2$, we do not apply the $[\sigma_j]_{j\in J}$ to t, simply because t is ground.

7.2 Type system

Because of the type directed nature of our calculus, its dynamic semantics is only defined for well-typed expressions. Therefore we introduce the type system before giving the reduction rules.

Definition 7.2.1 (Typing environment). *A* typing environment Γ *is a finite mapping from expression variables $\mathcal{X}$ to types $\mathcal{T}$, and written as a finite set of pairs $\{(x_1 : t_1), \ldots, (x_n : t_n)\}$. The set of expression variables which is defined in Γ is called the domain of Γ, denoted by $\mathsf{dom}(\Gamma)$. The set of type variables occurring in Γ, that is $\bigcup_{(x:t)\in\Gamma} \mathsf{var}(t)$, is denoted by $\mathsf{var}(\Gamma)$. If Γ is a type environment, then $\Gamma, (x : t)$ is the type environment defined as*

$$(\Gamma, (x : t))(y) = \begin{cases} t & \text{if } y = x \\ \Gamma(y) & \text{otherwise} \end{cases}$$

The definition of type-substitution application can be extended to type environments by applying the type-substitution to each type in the type environment, namely,

$$\Gamma\sigma = \{(x : t\sigma) \mid (x : t) \in \Gamma\}$$

The typing judgment for expressions has the form $\Delta \wr \Gamma \vdash e : t$, which states that under the set Δ of (monomorphic) type variables and the typing environment Γ the expression e has type t. When Δ and Γ are both empty, we write $\vdash e : t$ for short. We assume that there is a basic type b_c for each constant c. We write $\sigma \sharp \Delta$ as abbreviation for $\mathsf{dom}(\sigma) \cap \Delta = \emptyset$. The typing rules are given in Figure 7.1. While most of these rules are standard, some deserve a few comments.

The rule (*abstr*) is a little bit tricky. Consider a λ-abstraction $\lambda^{\wedge_{i\in I} t_i \to s_i}_{[\sigma_j]_{j\in J}} x.e$. Since the type variables introduced in the (relabeled) interface are bound by the abstraction, they cannot be instantiated in the body e, so we add them to the set Δ when type-checking e. We have to verify that the abstraction can be typed with each arrow type $t_i \to s_i$ in the interface to which we apply each decoration σ_j. That is, for each type $t_i\sigma_j \to s_i\sigma_j$, we check the abstraction once: the variable x is assumed to have type $t_i\sigma_j$ and the relabeled body $e@[\sigma_j]$ is checked against the type $s_i\sigma_j$. The type for the

$$\frac{}{\Delta \fatsemi \Gamma \vdash c : b_c}\ (const) \qquad \frac{}{\Delta \fatsemi \Gamma \vdash x : \Gamma(x)}\ (var)$$

$$\frac{\Delta \fatsemi \Gamma \vdash e_1 : t_1 \qquad \Delta \fatsemi \Gamma \vdash e_2 : t_2}{\Delta \fatsemi \Gamma \vdash (e_1, e_2) : t_1 \times t_2}\ (pair) \qquad \frac{\Delta \fatsemi \Gamma \vdash e : t_1 \times t_2}{\Delta \fatsemi \Gamma \vdash \pi_i(e) : t_i}\ (proj)$$

$$\frac{\Delta \fatsemi \Gamma \vdash e_1 : t_1 \to t_2 \qquad \Delta \fatsemi \Gamma \vdash e_2 : t_1}{\Delta \fatsemi \Gamma \vdash e_1 e_2 : t_2}\ (appl)$$

$$\frac{\begin{array}{l}\Delta' = \Delta \cup \mathsf{var}(\bigwedge\limits_{i\in I, j\in J} t_i\sigma_j \to s_i\sigma_j) \\ \forall i \in I, j \in J.\ \Delta' \fatsemi \Gamma, (x : t_i\sigma_j) \vdash e@[\sigma_j] : s_i\sigma_j\end{array}}{\Delta \fatsemi \Gamma \vdash \lambda^{\bigwedge_{i\in I} t_i\to s_i}_{[\sigma_j]_{j\in J}} x.e : \bigwedge\limits_{i\in I, j\in J} t_i\sigma_j \to s_i\sigma_j}\ (abstr)$$

$$\frac{\Delta \fatsemi \Gamma \vdash e : t' \qquad \left\{\begin{array}{lcl} t' \not\leq \neg t & \Rightarrow & \Delta \fatsemi \Gamma \vdash e_1 : s \\ t' \not\leq t & \Rightarrow & \Delta \fatsemi \Gamma \vdash e_2 : s \end{array}\right.}{\Delta \fatsemi \Gamma \vdash (e{\in}t\,?\,e_1 : e_2) : s}\ (case)$$

$$\frac{\Delta \fatsemi \Gamma \vdash e : t \qquad \sigma \sharp \Delta}{\Delta \fatsemi \Gamma \vdash e[\sigma] : t\sigma}\ (inst) \qquad \frac{\forall j \in J.\ \Delta \fatsemi \Gamma \vdash e[\sigma_j] : t_j \qquad |J| > 1}{\Delta \fatsemi \Gamma \vdash e[\sigma_j]_{j\in J} : \bigwedge\limits_{j\in J} t_j}\ (inter)$$

$$\frac{\Delta \fatsemi \Gamma \vdash e : s \qquad s \leq t}{\Delta \fatsemi \Gamma \vdash e : t}\ (subsum)$$

Figure 7.1: Typing rules

abstraction is obtained by taking the intersection of all the types $t_i\sigma_j \to s_i\sigma_j$. For example, consider the daffy identity function

$$(\lambda^{\alpha\to\alpha}_{[\{\mathtt{Int}/\alpha\},\{\mathtt{Bool}/\alpha\}]} x.(\lambda^{\alpha\to\alpha} y.x)x). \tag{7.1}$$

The Δ is always empty and the rule checks whether under the hypothesis $x : \alpha\{\mathtt{Int}/\alpha\}$ (*ie*, $x : \mathtt{Int}$), it is possible to deduce that $((\lambda^{\alpha\to\alpha} y.x)x)@[\{\mathtt{Int}/\alpha\}]$ has type $\alpha\{\mathtt{Int}/\alpha\}$ (*ie*, that $(\lambda^{\mathtt{Int}\to\mathtt{Int}} y.x)x : \mathtt{Int}$), and similarly for the substitution $\{\mathtt{Bool}/\alpha\}$. The relabeling of the body in the premises is a key mechanism of the type system: if we had used $e[\sigma_j]$ instead of $e@[\sigma_j]$ in the premises of the (*abstr*) rule, then the expression (7.1) could not be typed. The reason is that $e[\sigma_j]$ is more demanding on typing than $e@[\sigma_j]$, since the well typing of e is necessary to the former but not to the latter. Indeed while under the hypothesis $x : \mathtt{Int}$ we just showed that $((\lambda^{\alpha\to\alpha} y.x)x)@\{\mathtt{Int}/\alpha\}$ —*ie*, $((\lambda^{\mathtt{Int}\to\mathtt{Int}} y.x)x)$— is well-typed, the term $((\lambda^{\alpha\to\alpha} y.x)x)\{\mathtt{Int}/\alpha\}$ is not, for $(\lambda^{\alpha\to\alpha} y.x)$ has not type $\alpha{\to}\alpha$.

For a type case $e{\in}t\,\mathtt{?}\,e_1 : e_2$ (rule (*case*)), we first infer the type t' of the expression e which is dynamically tested against t. Then we check the type of each branch e_i only if there is a possibility that the branch can be selected. For example, consider the first branch e_1. If t' has a non-empty intersection with t (*i.e.*, $t' \not\leq \neg t$), then e_1 might be selected. In this case, in order for the whole expression to have type s, we need to check that e_1 has also type s. Otherwise (*i.e.*, $t' \leq \neg t$), e_1 cannot be selected, and there is no need to type-check it. The reasoning is the same for the second branch e_2 where $\neg t$ is replaced by t. Note that the ability to ignore the first branch e_1 and/or the second one e_2 when computing the type for a type case $e{\in}t\,\mathtt{?}\,e_1 : e_2$ is important to type-check overloaded functions. For example, consider the abstraction $\lambda^{(\mathtt{Int}\to\mathtt{Int})\wedge(\mathtt{Bool}\to\mathtt{Bool})}x.(x{\in}\mathtt{Int}\,\mathtt{?}\,42 : \mathtt{false})$. According to the rule (*abstr*), the abstraction will be checked against $\mathtt{Int}\to\mathtt{Int}$, that is, the body is checked against type $\mathtt{Int}$ when x is assumed to have type $\mathtt{Int}$. Since $\mathtt{Int}\leq\mathtt{Int}$, then the second branch is not type-checked. Otherwise, the type of the body would contain $\mathtt{Bool}$, which is not a subtype of $\mathtt{Int}$ and thus the function would not be well-typed.

In the rule (*inst*), we require that only the polymorphic type variables (*i.e.*, those not in Δ) can be instantiated. Otherwise, without this requirement, an expression such as $\lambda^{(\alpha\to\beta)}x.x[\{\beta/\alpha\}]$ could be typed as follows:

$$\dfrac{\dfrac{\overline{\{\alpha,\beta\}\,;\,\{(x:\alpha)\}\vdash x:\alpha}}{\{\alpha,\beta\}\,;\,\{(x:\alpha)\}\vdash x[\{\beta/\alpha\}]:\beta}}{\vdash \lambda^{(\alpha\to\beta)}x.x[\{\beta/\alpha\}]:\alpha\to\beta}$$

which is unsound (by applying the above functions it is possible to create polymorphic values of type β, for every β). Moreover, in a judgment $\Delta\,;\,\Gamma\vdash e:t$, the type variables occurring in Γ are to be considered monomorphic, and therefore we require them to be contained in Δ. Without such a requirement it would be possible to have a derivation such as the following one:

$$\dfrac{\overline{\{\alpha\}\,;\,(x:\beta)\vdash x:\beta}\qquad \{\gamma/\beta\}\,\sharp\,\{\alpha\}}{\{\alpha\}\,;\,(x:\beta)\vdash x[\{\gamma/\beta\}]:\gamma}$$

which states that by assuming that x has (any) type β, we can infer that it has also (any other) type γ. This is unsound. We can prove that the condition $\mathsf{var}(\Gamma)\subseteq\Delta$ is preserved by the typing rules (see Lemma 7.4.1). When we type a closed expression, typically in the proof of the progress property, we have $\Delta=\Gamma=\emptyset$, which satisfies the condition. This implies that all the judgments used in the proofs used for soundness satisfy it. Therefore, henceforth, we implicitly assume the condition $\mathsf{var}(\Gamma)\subseteq\Delta$ to hold in all judgments we consider.

The rule (*subsum*) makes the type system depend on the subtyping relation defined inChapter 4. It is important not to confuse the subtyping relation $\leq$ of our system, which denotes semantic subtyping (*ie*, set-theoretic inclusion of denotations), with the ML one, which stands for type variable instantiation. For example, in ML we have $\alpha\to\alpha\leq\mathtt{Int}\to\mathtt{Int}$ (because $\mathtt{Int}\to\mathtt{Int}$ is an instance of $\alpha\to\alpha$). But this is not true in our system, as the relation would have to hold for *every possible instantiation*

of α*,* thus in particular for α equal to `Bool`. Notice that the preorder $\sqsubseteq_\Delta$ defined in Section 9.1 includes the type variable instantiation of the preorder typically used for ML, so any direct comparison with constraint systems for ML types should focus on $\sqsubseteq_\Delta$ rather than $\leq$.

Note that there is no typing rule for intersection elimination, as it can be encoded by subsumption. Indeed, we have $t \wedge s \leq t$ and $t \wedge s \leq s$, so from $\Delta \fatsemi \Gamma \vdash e : t \wedge s$, we can deduce $\Delta \fatsemi \Gamma \vdash e : t$ (or s). The rule (*inter*) introduces intersection only to combine different instances of a same type. It does not restrict the expressiveness of the type system, as we can prove that the usual rule for intersection introduction is admissible in our system (see Lemma 7.4.2).

7.3 Operational semantics

In this section, we define the semantics of the calculus, which depends on the type system given in Section 7.2.

Definition 7.3.1 (**Values**). *An expression* e *is a* value *if it is closed, well-typed (*ie, $\vdash e : t$ *for some type* t*), and produced by the following grammar:*

$$\textbf{\textit{Values}} \qquad v \ ::= \ c \mid (v, v) \mid \lambda^{(\wedge_{i\in I} t_i \to s_i)}_{[\sigma_j]_{j\in J}} x.e$$

We write $\mathscr{V}$ *to denote the set of all values.*

Definition 7.3.2 (**Context**). *Let the symbol* $[\_]$ *denote a hole. A* context $C[\_]$ *is an expression with a hole:*

$$\begin{array}{lrcl} \textbf{\textit{Contexts}} & C[\_] & ::= & [\_] \\ & & \mid & (C[\_], e) \mid (e, C[\_]) \\ & & \mid & C[\_]e \mid eC[\_] \\ & & \mid & C[\_] \in t \,?\, e : e \mid e \in t \,?\, C[\_] : e \mid e \in t \,?\, e : C[\_] \\ & & \mid & \pi_i(C[\_]) \end{array}$$

An evaluation context $E[\_]$ *is a context that implements outermost leftmost reduction:*

$$\begin{array}{lrcl} \textbf{\textit{Evaluation Contexts}} & E[\_] & ::= & [\_] \\ & & \mid & (E[\_], e) \mid (v, E[\_]) \\ & & \mid & E[\_]e \mid vE[\_] \\ & & \mid & E[\_] \in t \,?\, e : e \\ & & \mid & \pi_i(E[\_]) \end{array}$$

We use $C[e]$ *and* $E[e]$ *to denote the expressions obtained by replacing* e *for the hole in* $C[\_]$ *and* $E[\_]$*, respectively.*

We define a small-step call-by-value operational semantics for the calculus. The semantics is given by the relation $\rightsquigarrow$, which is shown in Figure 7.2. There are four notions of reduction: one for projections, one for applications, one for type cases, and one for instantiations, plus context closure. Henceforth we will establish all the properties for the reduction using generic contexts but, of course, these holds also

Notions of reduction:

$$(Rproj) \quad \pi_i(v_1, v_2) \rightsquigarrow v_i$$

$$(Rappl) \quad (\lambda^{\wedge_{i\in I} t_i \to s_i}_{[\sigma_j]_{j\in J}} x.e')v \rightsquigarrow (e'@[\sigma_j]_{j\in P})\{v/x\} \text{ where } P = \{j \in J \mid \exists i \in I.\ \vdash v : t_i\sigma_j\}$$

$$(Rcase) \quad (v \in t\,?\,e_1 : e_2) \rightsquigarrow \begin{cases} e_1 & \text{if } \vdash v : t \\ e_2 & \text{otherwise} \end{cases}$$

$$(Rinst) \quad e[\sigma_j]_{j\in J} \rightsquigarrow e@[\sigma_j]_{j\in J}$$

Context closure:

$$(Rctx) \quad \frac{e \rightsquigarrow e'}{C[e] \rightsquigarrow C[e']}$$

Figure 7.2: Operational semantics of the calculus

when the more restrictive evaluation contexts are used. The latter will only be used in Chapter 11 in order to simplify the setting.

The (*Rproj*) rule is the standard projection rule. The (*Rappl*) rule states the semantics of applications: this is standard call-by-value β-reduction, with the difference that the substitution of the argument for the parameter is performed on the relabeled body of the function. Notice that relabeling depends on the type of the argument and keeps only the substitutions that make the type of the argument v match (at least one of) the input types defined in the interface of the function (formally, we select the set P of substitutions σ_j such that the argument v has type $t_i\sigma_j$ for some i). The (*Rcase*) rule checks whether the value returned by the expression in the type-case matches the specified type and selects the branch accordingly. Finally, the (*Rinst*) rule performs relabeling, that is, it propagates the sets of type-substitutions down into the decorations of the outermost λ-abstractions.

We used a call-by-value semantics to ensure the type soundness property: subject reduction (or type preservation) and progress (closed and well-typed expressions which are not values can be reduced), which are discussed in Section 7.4.2. To understand why, consider each basic reduction rules in turn.

The requirement that the argument of a projection must be a value is imposed to ensure that the property of subject reduction holds. Consider the expression $\pi_1(e_1, e_2)$ where e_1 is an expression of type t_1 (different from $\mathbb{0}$) and e_2 is a (diverging) expression of type $\mathbb{0}$. Clearly, the type system assigns the type $t_1 \times \mathbb{0}$ to (e_1, e_2). In our system, a product type with an empty component is itself empty, and thus (e_1, e_2) has type $\mathbb{0}$. Therefore the type of the projection $\pi_1(e_1, e_2)$ as well has type $\mathbb{0}$ (since $\mathbb{0} \leq \mathbb{0} \times \mathbb{0}$, then by subsumption $(e_1, e_2) : \mathbb{0} \times \mathbb{0}$ and the result follows from the (*proj*) typing rule). If it were possible to reduce a projection when the argument is not a value, then $\pi_1(e_1, e_2)$ could be reduced to e_1, which has type t_1: type preservation would be violated.

Likewise, the reduction rule for applications requires the argument to be a value. Let us consider the application $(\lambda^{(t\to t\times t)\wedge(s\to s\times s)} x.(x, x))(e)$, where $\vdash e : t \vee s$. The type system assigns to the abstraction the type $(t \to t \times t) \wedge (s \to s \times s)$, which is

a subtype of $(t \vee s) \to ((t \times t) \vee (s \times s))$. By subsumption, the abstraction has type $(t \vee s) \to ((t \times t) \vee (s \times s))$, and thus, the application has type $(t \times t) \vee (s \times s)$. If the semantics permits to reduce an application when the argument is not a value, then this application could be reduced to the expression (e, e), which has type $(t \vee s) \times (t \vee s)$ but not $(t \times t) \vee (s \times s)$.

Finally, if we allowed $(e{\in}t\,?\,e_1 : e_2)$ to reduce to e_1 when $\vdash e : t$ but e is not a value, we could break type preservation. For example, assume that $\vdash e : \mathbb{0}$. Then the type system would not check anything about the branches e_1 and e_2 (see the typing rule (*case*) in Figure 7.1) and so e_1 could be ill-typed.

Notice that in all these cases the usage of values ensures subject-reduction but it is not a necessary condition: in some cases weaker constraints could be used. For instance, in order to check whether an expression is a list of integers, in general it is not necessary to fully evaluate the whole list: the head and the type of the tail are all that is needed. Studying weaker conditions for the reduction rules is an interesting topic we leave for future work, in particular in the view of adapting our framework to lazy languages.

7.4 Properties

In this section we present some properties of our type system. First, we study the syntactic meta-theory of our type system, in particular the admissibility of the intersection rule, the generation lemma for values, the property that substitutions preserve typing. These are functional to the proof of *soundness*, the fundamental property of that links every type system of a calculus with its operational counterpart: well-typed expressions do not go wrong [Mil78]. Finally, we prove that the explicitly-typed calculus is able to derive the same typing judgements as the BCD intersection type system defined by Barendregt, Coppo, and Dezani [BCD83]; and that the expressions of the form $e[\sigma_j]_{j\in J}$ are redundant insofar as their presence in the calculus does not increase its expressive power.

7.4.1 Syntactic meta-theory

Lemma 7.4.1. *If $\Delta \fatsemi \Gamma \vdash e : t$ and $\mathsf{var}(\Gamma) \subseteq \Delta$, then $\mathsf{var}(\Gamma') \subseteq \Delta'$ holds for every judgment $\Delta' \fatsemi \Gamma' \vdash e' : t'$ in the derivation of $\Delta \fatsemi \Gamma \vdash e : t$.*

Proof. By induction on the derivation of $\Delta \fatsemi \Gamma \vdash e : t$. □

Lemma 7.4.2 (Admissibility of intersection introduction). *Let e be an expression. If $\Delta \fatsemi \Gamma \vdash e : t$ and $\Delta \fatsemi \Gamma \vdash e : t'$, then $\Delta \fatsemi \Gamma \vdash e : t \wedge t'$.*

Proof. The proof proceeds by induction on the two typing derivations. First, assume that these two derivations end with an instance of the same rule corresponding to the top-level constructor of e.

(*const*)**:** both derivations end with an instance of (*const*):

$$\frac{}{\Delta \fatsemi \Gamma \vdash c : b_c}\ (const) \qquad \frac{}{\Delta \fatsemi \Gamma \vdash c : b_c}\ (const)$$

Trivially, we have $b_c \wedge b_c \simeq b_c$, by subsumption, the result follows.

(*var*)**:** both derivations end with an instance of (*var*):

$$\frac{}{\Delta \wp \Gamma \vdash x : \Gamma(x)}\ (var) \qquad \frac{}{\Delta \wp \Gamma \vdash x : \Gamma(x)}\ (var)$$

Trivially, we have $\Gamma(x) \wedge \Gamma(x) \simeq \Gamma(x)$, by subsumption, the result follows.

(*pair*)**:** both derivations end with an instance of (*pair*):

$$\frac{\dfrac{\ldots}{\Delta \wp \Gamma \vdash e_1 : t_1} \quad \dfrac{\ldots}{\Delta \wp \Gamma \vdash e_2 : t_2}}{\Delta \wp \Gamma \vdash (e_1, e_2) : (t_1 \times t_2)}\ (pair) \qquad \frac{\dfrac{\ldots}{\Delta \wp \Gamma \vdash e_1 : t'_1} \quad \dfrac{\ldots}{\Delta \wp \Gamma \vdash e_2 : t'_2}}{\Delta \wp \Gamma \vdash (e_1, e_2) : (t'_1 \times t'_2)}\ (pair)$$

By induction, we have $\Delta \wp \Gamma \vdash e_i : (t_i \wedge t'_i)$. Then the rule (*pair*) gives us $\Delta \wp \Gamma \vdash (e_1, e_2) : (t_1 \wedge t'_1) \times (t_2 \wedge t'_2)$. Moreover, because intersection distributes over products, we have $(t_1 \wedge t'_1) \times (t_2 \wedge t'_2) \simeq (t_1 \times t_2) \wedge (t'_1 \times t'_2)$. Then by (*subsum*), we have $\Delta \wp \Gamma \vdash (e_1, e_2) : (t_1 \times t_2) \wedge (t'_1 \times t'_2)$.

(*proj*)**:** both derivations end with an instance of (*proj*):

$$\frac{\dfrac{\ldots}{\Delta \wp \Gamma \vdash e' : t_1 \times t_2}}{\Delta \wp \Gamma \vdash \pi_i(e') : t_i}\ (proj) \qquad \frac{\dfrac{\ldots}{\Delta \wp \Gamma \vdash e' : t'_1 \times t'_2}}{\Delta \wp \Gamma \vdash \pi_i(e') : t'_i}\ (proj)$$

By induction, we have $\Delta \wp \Gamma \vdash e' : (t_1 \times t_2) \wedge (t'_1 \times t'_2)$. Since $(t_1 \wedge t'_1) \times (t_2 \wedge t'_2) \simeq (t_1 \times t_2) \wedge (t'_1 \times t'_2)$ (see the case of (*pair*)), by (*subsum*), we have $\Delta \wp \Gamma \vdash e' : (t_1 \wedge t'_1) \times (t_2 \wedge t'_2)$. Then the rule (*proj*) gives us $\Delta \wp \Gamma \vdash \pi_i(e') : t_i \wedge t'_i$.

(*appl*)**:** both derivations end with an instance of (*appl*):

$$\frac{\dfrac{\ldots}{\Delta \wp \Gamma \vdash e_1 : t_1 \to t_2} \quad \dfrac{\ldots}{\Delta \wp \Gamma \vdash e_2 : t_1}}{\Delta \wp \Gamma \vdash e_1\ e_2 : t_2}\ (appl)$$

$$\frac{\dfrac{\ldots}{\Delta \wp \Gamma \vdash e_1 : t'_1 \to t'_2} \quad \dfrac{\ldots}{\Delta \wp \Gamma \vdash e_2 : t'_1}}{\Delta \wp \Gamma \vdash e_1\ e_2 : t'_2}\ (appl)$$

By induction, we have $\Delta \wp \Gamma \vdash e_1 : (t_1 \to t_2) \wedge (t'_1 \to t'_2)$ and $\Delta \wp \Gamma \vdash e_2 : t_1 \wedge t'_1$. Because intersection distributes over arrows, we have $(t_1 \to t_2) \wedge (t'_1 \to t'_2) \leq (t_1 \wedge t'_1) \to (t_2 \wedge t'_2)$. Then by the rule (*subsum*), we get $\Delta \wp \Gamma \vdash e_1 : (t_1 \wedge t'_1) \to (t_2 \wedge t'_2)$. Finally, by applying (*appl*), we get $\Delta \wp \Gamma \vdash e_1\ e_2 : t_2 \wedge t'_2$ as expected.

(*abstr*)**:** both derivations end with an instance of (*abstr*):

$$\frac{\begin{array}{l}\forall i \in I, j \in J.\ \dfrac{\ldots}{\Delta' \wp \Gamma, (x : t_i\sigma_j) \vdash e'@[\sigma_j] : s_i\sigma_j} \\ \Delta' = \Delta \cup \mathsf{var}(\bigwedge_{i \in I, j \in J} t_i\sigma_j \to s_i\sigma_j)\end{array}}{\Delta \wp \Gamma \vdash \lambda^{\wedge_{i \in I} t_i \to s_i}_{[\sigma_j]_{j \in J}} x.e' : \bigwedge_{i \in I, j \in J} t_i\sigma_j \to s_i\sigma_j}$$

$$\frac{\begin{array}{l}\forall i \in I, j \in J.\ \dfrac{\ldots}{\Delta' \wp \Gamma, (x : t_i\sigma_j) \vdash e'@[\sigma_j] : s_i\sigma_j} \\ \Delta' = \Delta \cup \mathsf{var}(\bigwedge_{i \in I, j \in J} t_i\sigma_j \to s_i\sigma_j)\end{array}}{\Delta \wp \Gamma \vdash \lambda^{\wedge_{i \in I} t_i \to s_i}_{[\sigma_j]_{j \in J}} x.e' : \bigwedge_{i \in I, j \in J} t_i\sigma_j \to s_i\sigma_j}$$

It is clear that

$$(\bigwedge_{i\in I,j\in J} t_i\sigma_j \to s_i\sigma_j) \wedge (\bigwedge_{i\in I,j\in J} t_i\sigma_j \to s_i\sigma_j) \simeq \bigwedge_{i\in I,j\in J} t_i\sigma_j \to s_i\sigma_j$$

By subsumption, the result follows.

(*case*)**:** both derivations end with an instance of (*case*):

$$\frac{\dfrac{\ldots}{\Delta \wr \Gamma \vdash e_0 : t_0} \quad \begin{cases} t_0 \not\leq \neg t & \Rightarrow \quad \dfrac{\ldots}{\Delta \wr \Gamma \vdash e_1 : s} \\ t_0 \not\leq t & \Rightarrow \quad \dfrac{\ldots}{\Delta \wr \Gamma \vdash e_2 : s} \end{cases}}{\Delta \wr \Gamma \vdash (e_0 \in t \,?\, e_1 : e_2) : s}\ (case)$$

$$\frac{\dfrac{\ldots}{\Delta \wr \Gamma \vdash e_0 : t_0'} \quad \begin{cases} t_0' \not\leq \neg t & \Rightarrow \quad \dfrac{\ldots}{\Delta \wr \Gamma \vdash e_1 : s'} \\ t_0' \not\leq t & \Rightarrow \quad \dfrac{\ldots}{\Delta \wr \Gamma \vdash e_2 : s'} \end{cases}}{\Delta \wr \Gamma \vdash (e_0 \in t \,?\, e_1 : e_2) : s'}\ (case)$$

By induction, we have $\Delta \wr \Gamma \vdash e_0 : t_0 \wedge t_0'$. Suppose $t_0 \wedge t_0' \not\leq \neg t$; then $t_0 \not\leq \neg t$ and $t_0' \not\leq \neg t$. Consequently, the branch e_1 has been type-checked in both cases, and we have $\Delta \wr \Gamma \vdash e_1 : s \wedge s'$ by the induction hypothesis. Similarly, if $t_0 \wedge t_0' \not\leq t$, then we have $\Delta \wr \Gamma \vdash e_2 : s \wedge s'$. Consequently, we have $\Delta \wr \Gamma \vdash (e_0 \in t \,?\, e_1 : e_2) : s \wedge s'$ by the rule (*case*).

(*inst*)**:** both derivations end with an instance of (*inst*):

$$\frac{\dfrac{\ldots}{\Delta \wr \Gamma \vdash e' : t} \quad \sigma \,\sharp\, \Delta}{\Delta \wr \Gamma \vdash e'[\sigma] : t\sigma}\ (inst) \qquad \frac{\dfrac{\ldots}{\Delta \wr \Gamma \vdash e' : t'} \quad \sigma \,\sharp\, \Delta}{\Delta \wr \Gamma \vdash e'[\sigma] : t'\sigma}\ (inst)$$

By induction, we have $\Delta \wr \Gamma \vdash e' : t \wedge t'$. Since $\sigma \,\sharp\, \Delta$, the rule (*inst*) gives us $\Delta \wr \Gamma \vdash e'[\sigma] : (t \wedge t')\sigma$, that is $\Delta \wr \Gamma \vdash e'[\sigma] : (t\sigma) \wedge (t'\sigma)$.

(*inter*)**:** both derivations end with an instance of (*inter*):

$$\frac{\forall j \in J.\ \dfrac{\ldots}{\Delta \wr \Gamma \vdash e'[\sigma_j] : t_j}}{\Delta \wr \Gamma \vdash e'[\sigma_j]_{j\in J} : \bigwedge_{j\in J} t_j}\ (inter) \qquad \frac{\forall j \in J.\ \dfrac{\ldots}{\Delta \wr \Gamma \vdash e'[\sigma_j] : t_j'}}{\Delta \wr \Gamma \vdash e'[\sigma_j]_{j\in J} : \bigwedge_{j\in J} t_j'}\ (inter)$$

where $|J| > 1$. By induction, we have $\Delta \wr \Gamma \vdash e'[\sigma_j] : t_j \wedge t_j'$ for all $j \in J$. Then the rule (*inter*) gives us $\Delta \wr \Gamma \vdash e'[\sigma_j]_{j\in J} : \bigwedge_{j\in J}(t_j \wedge t_j')$, that is, $\Delta \wr \Gamma \vdash e'[\sigma_j]_{j\in J} : (\bigwedge_{j\in J} t_j) \wedge (\bigwedge_{j\in J} t_j')$.

Otherwise, there exists at least one typing derivation which ends with an instance of (*subsum*), for instance,

$$\frac{\dfrac{\ldots}{\Delta \wr \Gamma \vdash e' : s} \quad s \leq t}{\Delta \wr \Gamma \vdash e' : t}\ (subsum) \qquad \frac{\ldots}{\Delta \wr \Gamma \vdash e' : t'}$$

By induction, we have $\Delta \wr \Gamma \vdash e' : s \wedge t'$. Since $s \leq t$, we have $s \wedge t' \leq t \wedge t'$. Then the rule (*subsum*) gives us $\Delta \wr \Gamma \vdash e' : t \wedge t'$ as expected. □

Lemma 7.4.3 (Generation for values). *Let v be a value. Then*

1. *If $\Delta \fatsemi \Gamma \vdash v : b$, then v is a constant c and $b_c \leq b$.*
2. *If $\Delta \fatsemi \Gamma \vdash v : t_1 \times t_2$, then v has the form of (v_1, v_2) with $\Delta \fatsemi \Gamma \vdash v_i : t_i$.*
3. *If $\Delta \fatsemi \Gamma \vdash v : t \to s$, then v has the form of $\lambda^{\bigwedge_{i\in I} t_i \to s_i}_{[\sigma_j]_{j\in J}} x.e_0$ with $\bigwedge_{i\in I, j\in J}(t_i\sigma_j \to s_i\sigma_j) \leq t \to s$.*

Proof. By a simple examination of the rules it is easy to see that a derivation for $\Delta \fatsemi \Gamma \vdash v : t$ is always formed by an instance of the rule corresponding to the kind of v (*i.e.*, (*const*) for constants, (*pair*) for pairs, and (*abstr*) for abstractions), followed by zero or more instances of (*subsum*). By induction on the depth of the derivation it is then easy to prove that if $\Delta \fatsemi \Gamma \vdash v : t$ is derivable, then $t \not\equiv \mathbb{0}$. The lemma then follows by induction on the number of the instances of the subsubsumption rule that end the derivation of $\Delta \fatsemi \Gamma \vdash v : t$. The base case are straightforward, while the inductive cases are:

$\underline{\Delta \fatsemi \Gamma \vdash v : b}$**:** v is by induction a constant c such that $b_c \leq b$.
$\underline{\Delta \fatsemi \Gamma \vdash v : t_1 \times t_2}$**:** v is by induction a pair (v_1, v_2) and t' is form of $(t'_1 \times t'_2)$ such that $\Delta \fatsemi \Gamma \vdash v_i : t'_i$. Here we use the fact that the type of a value cannot be $\mathbb{0}$: since $\mathbb{0} \not\simeq (t'_1 \times t'_2) \leq (t_1 \times t_2)$, then we have $t'_i \leq t_i$. Finally, by (*subsum*), we have $\Delta \fatsemi \Gamma \vdash v_i : t_i$.
$\underline{\Delta \fatsemi \Gamma \vdash v : t \to s}$**:** v is by induction an abstraction $\lambda^{\bigwedge_{i\in I} t_i \to s_i}_{[\sigma_j]_{j\in J}} x.e_0$ such that $\bigwedge_{i\in I, j\in J}(t_i\sigma_j \to s_i\sigma_j) \leq t \to s$.

□

Lemma 7.4.4. *Let e be an expression and $[\sigma_j]_{j\in J}, [\sigma_k]_{k\in K}$ two sets of type substitutions. Then*

$$(e@[\sigma_j]_{j\in J})@[\sigma_k]_{k\in K} = e@([\sigma_k]_{k\in K} \circ [\sigma_j]_{j\in J})$$

Proof. By induction on the structure of e.

$\underline{e = c}$**:**

$$\begin{aligned}(c@[\sigma_j]_{j\in J})@[\sigma_k]_{k\in K} &= c@[\sigma_k]_{k\in K}\\ &= c\\ &= c@([\sigma_k]_{k\in K} \circ [\sigma_j]_{j\in J})\end{aligned}$$

$\underline{e = x}$**:**

$$\begin{aligned}(x@[\sigma_j]_{j\in J})@[\sigma_k]_{k\in K} &= x@[\sigma_k]_{k\in K}\\ &= x\\ &= x@([\sigma_k]_{k\in K} \circ [\sigma_j]_{j\in J})\end{aligned}$$

$\underline{e = (e_1, e_2)}$**:**

$$\begin{aligned}((e_1, e_2)@[\sigma_j]_{j\in J})@[\sigma_k]_{k\in K} &= (e_1@[\sigma_j]_{j\in J}, e_2@[\sigma_j]_{j\in J})@[\sigma_k]_{k\in K}\\ &= ((e_1@[\sigma_j]_{j\in J})@[\sigma_k]_{k\in K}, (e_2@[\sigma_j]_{j\in J})@[\sigma_k]_{k\in K})\\ &\quad \text{(by induction)}\\ &= (e_1@([\sigma_k]_{k\in K} \circ [\sigma_j]_{j\in J}), e_2@([\sigma_k]_{k\in K} \circ [\sigma_j]_{j\in J}))\\ &= (e_1, e_2)@([\sigma_k]_{k\in K} \circ [\sigma_j]_{j\in J})\end{aligned}$$

$\underline{e = \pi_i(e')}$**:**

$$\begin{array}{rcll}(\pi_i(e')@[\sigma_j]_{j\in J})@[\sigma_k]_{k\in K} &=& (\pi_i(e'@[\sigma_j]_{j\in J}))@[\sigma_k]_{k\in K} & \\ &=& \pi_i((e'@[\sigma_j]_{j\in J})@[\sigma_k]_{k\in K}) & \text{(by induction)} \\ &=& \pi_i(e'@([\sigma_k]_{k\in K}\circ[\sigma_j]_{j\in J})) & \\ &=& \pi(e')@([\sigma_k]_{k\in K}\circ[\sigma_j]_{j\in J}) & \end{array}$$

$\underline{e = e_1e_2}$**:**

$$\begin{array}{rcl}((e_1e_2)@[\sigma_j]_{j\in J})@[\sigma_k]_{k\in K} &=& ((e_1@[\sigma_j]_{j\in J})(e_2@[\sigma_j]_{j\in J}))@[\sigma_k]_{k\in K} \\ &=& ((e_1@[\sigma_j]_{j\in J})@[\sigma_k]_{k\in K})((e_2@[\sigma_j]_{j\in J})@[\sigma_k]_{k\in K}) \\ & & \text{(by induction)} \\ &=& (e_1@([\sigma_k]_{k\in K}\circ[\sigma_j]_{j\in J}))(e_2@([\sigma_k]_{k\in K}\circ[\sigma_j]_{j\in J})) \\ &=& (e_1e_2)@([\sigma_k]_{k\in K}\circ[\sigma_j]_{j\in J})\end{array}$$

$\underline{e = \lambda^{\wedge_{i\in I}t_i\to s_i}_{[\sigma_{j'}]_{j'\in J'}}x.e'}$**:**

$$\begin{array}{rcl}((\lambda^{\wedge_{i\in I}t_i\to s_i}_{[\sigma_{j'}]_{j'\in J'}}x.e')@[\sigma_j]_{j\in J})@[\sigma_k]_{k\in K} &=& (\lambda^{\wedge_{i\in I}t_i\to s_i}_{[\sigma_j]_{j\in J}\circ[\sigma_{j'}]_{j'\in J'}}x.e')@[\sigma_k]_{k\in K} \\ &=& (\lambda^{\wedge_{i\in I}t_i\to s_i}_{[\sigma_k]_{k\in K}\circ[\sigma_j]_{j\in J}\circ[\sigma_{j'}]_{j'\in J'}}x.e') \\ &=& (\lambda^{\wedge_{i\in I}t_i\to s_i}_{[\sigma_{j'}]_{j'\in J'}}x.e')@([\sigma_k]_{k\in K}\circ[\sigma_j]_{j\in J})\end{array}$$

$\underline{e = e_0{\in}t\,?\,e_1 : e_2}$**:** similar to $e = (e_1, e_2)$
$\underline{e = e'[\sigma_i]_{i\in I}}$**:**

$$\begin{array}{rcll}((e'[\sigma_i]_{i\in I})@[\sigma_j]_{j\in J})@[\sigma_k]_{k\in K} &=& (e'@([\sigma_j]_{j\in J}\circ[\sigma_i]_{i\in I}))@[\sigma_k]_{k\in K} & \\ &=& e'@([\sigma_k]_{k\in K}\circ[\sigma_j]_{j\in J}\circ[\sigma_i]_{i\in I}) & \text{(by induction)} \\ &=& (e'[\sigma_i]_{i\in I})@([\sigma_k]_{k\in K}\circ[\sigma_j]_{j\in J}) & \end{array}$$

□

Lemma 7.4.5. *Let e be an expression, ϱ an expression substitution and $[\sigma_j]_{j\in J}$ a set of type substitutions such that $\mathsf{tv}(\varrho)\cap\bigcup_{j\in J}\mathsf{dom}(\sigma_j) = \emptyset$. Then $(e\varrho)@[\sigma_j]_{j\in J} = (e@[\sigma_j]_{j\in J})\varrho$.*

Proof. By induction on the structure of e.

$\underline{e = c}$**:**

$$\begin{array}{rcl}(c\varrho)@[\sigma_j]_{j\in J} &=& c@[\sigma_j]_{j\in J} \\ &=& c \\ &=& c\varrho \\ &=& (c@[\sigma_j]_{j\in J})\varrho\end{array}$$

$\underline{e = x}$**:** if $x\notin\mathsf{dom}(\varrho)$, then

$$\begin{array}{rcl}(x\varrho)@[\sigma_j]_{j\in J} &=& x@[\sigma_j]_{j\in J} \\ &=& x \\ &=& x\varrho \\ &=& (x@[\sigma_j]_{j\in J})\varrho\end{array}$$

Otherwise, let $\varrho(x) = e'$. As $\mathsf{tv}(\varrho) \cap \bigcup_{j\in J} \mathsf{dom}(\sigma_j) = \emptyset$, we have $\mathsf{tv}(e') \cap \bigcup_{j\in J} \mathsf{dom}(\sigma_j) = \emptyset$. Then

$$\begin{array}{rcl}(x\varrho)@[\sigma_j]_{j\in J} &=& e'@[\sigma_j]_{j\in J}\\ &=& e'\\ &=& x\varrho\\ &=& (x@[\sigma_j]_{j\in J})\varrho\end{array}$$

$\underline{e = (e_1, e_2)}$**:**

$$\begin{array}{rcll}((e_1,e_2)\varrho)@[\sigma_j]_{j\in J} &=& (e_1\varrho, e_2\varrho)@[\sigma_j]_{j\in J}\\ &=& ((e_1\varrho)@[\sigma_j]_{j\in J}, (e_2\varrho)@[\sigma_j]_{j\in J})\\ &=& ((e_1@[\sigma_j]_{j\in J})\varrho, (e_2@[\sigma_j]_{j\in J})\varrho) & \text{(by induction)}\\ &=& (e_1@[\sigma_j]_{j\in J}, e_2@[\sigma_j]_{j\in J})\varrho\\ &=& ((e_1,e_2)@[\sigma_j]_{j\in J})\varrho\end{array}$$

$\underline{e = \pi_i(e')}$**:**

$$\begin{array}{rcll}(\pi_i(e')\varrho)@[\sigma_j]_{j\in J} &=& (\pi_i(e'\varrho))@[\sigma_j]_{j\in J}\\ &=& \pi_i((e'\varrho)@[\sigma_j]_{j\in J})\\ &=& \pi_i((e_1@[\sigma_j]_{j\in J})\varrho) & \text{(by induction)}\\ &=& \pi_i(e'@[\sigma_j]_{j\in J})\varrho\\ &=& ((\pi_i(e'))@[\sigma_j]_{j\in J})\varrho\end{array}$$

$\underline{e = e_1e_2}$**:**

$$\begin{array}{rcll}((e_1e_2)\varrho)@[\sigma_j]_{j\in J} &=& ((e_1\varrho)(e_2\varrho))@[\sigma_j]_{j\in J}\\ &=& ((e_1\varrho)@[\sigma_j]_{j\in J})((e_2\varrho)@[\sigma_j]_{j\in J})\\ &=& ((e_1@[\sigma_j]_{j\in J})\varrho)((e_2@[\sigma_j]_{j\in J})\varrho) & \text{(by induction)}\\ &=& ((e_1@[\sigma_j]_{j\in J})(e_2@[\sigma_j]_{j\in J}))\varrho\\ &=& ((e_1e_2)@[\sigma_j]_{j\in J})\varrho\end{array}$$

$\underline{e = \lambda^{\wedge_{i\in I} t_i\to s_i}_{[\sigma_k]_{k\in K}} x.e'}$**:**

$$\begin{array}{rcll}((\lambda^{\wedge_{i\in I} t_i\to s_i}_{[\sigma_k]_{k\in K}} x.e')\varrho)@[\sigma_j]_{j\in J} &=& (\lambda^{\wedge_{i\in I} t_i\to s_i}_{[\sigma_k]_{k\in K}} x.(e'\varrho))@[\sigma_j]_{j\in J}\\ &=& \lambda^{\wedge_{i\in I} t_i\to s_i}_{[\sigma_j]_{j\in J}\circ[\sigma_k]_{k\in K}} x.(e'\varrho)\\ &=& (\lambda^{\wedge_{i\in I} t_i\to s_i}_{[\sigma_j]_{j\in J}\circ[\sigma_k]_{k\in K}} x.e')\varrho\\ & & \quad \text{(because } \mathsf{tv}(\varrho) \cap \bigcup_{j\in J} \mathsf{dom}(\sigma_j) = \emptyset)\\ &=& ((\lambda^{\wedge_{i\in I} t_i\to s_i}_{[\sigma_k]_{k\in K}} x.e')@[\sigma_j]_{j\in J})\varrho\end{array}$$

$\underline{e = e_0{\in}t\,?\,e_1 : e_2}$**:**

$$\begin{array}{rcll}((e_0{\in}t\,?\,e_1 : e_2)\varrho)@[\sigma_j]_{j\in J} &=& ((e_0\varrho){\in}t\,?\,(e_1\varrho) : (e_2\varrho))@[\sigma_j]_{j\in J}\\ &=& ((e_0\varrho)@[\sigma_j]_{j\in J}){\in}t\,?\,((e_1\varrho)@[\sigma_j]_{j\in J}) : ((e_2\varrho)@[\sigma_j]_{j\in J})\\ &=& ((e_0@[\sigma_j]_{j\in J})\varrho){\in}t\,?\,((e_1@[\sigma_j]_{j\in J})\varrho) : ((e_2@[\sigma_j]_{j\in J})\varrho)\\ & & \quad \text{(by induction)}\\ &=& ((e_0@[\sigma_j]_{j\in J}){\in}t\,?\,(e_1@[\sigma_j]_{j\in J}) : (e_2@[\sigma_j]_{j\in J}))\varrho\\ &=& ((e_0{\in}t\,?\,e_1 : e_2)@[\sigma_j]_{j\in J})\varrho\end{array}$$

$\underline{e = e'[\sigma_k]_{k\in K}}$**:** using α-conversion on the polymorphic type variables of e, we can assume $\mathsf{tv}(\varrho) \cap \bigcup_{k\in K} \mathsf{dom}(\sigma_k) = \emptyset$. Consequently we have $\mathsf{tv}(\varrho) \cap \bigcup_{k\in K, j\in J} \mathsf{dom}(\sigma_k \circ \sigma_j) = \emptyset$, and we deduce

$$\begin{array}{rcll}((e'[\sigma_k]_{k\in K})\varrho)@[\sigma_j]_{j\in J} & = & ((e'\varrho)[\sigma_k]_{k\in K})@[\sigma_j]_{j\in J} & \\ & = & (e'\varrho)@([\sigma_j]_{j\in J} \circ [\sigma_k]_{k\in K}) & \\ & = & (e'\varrho)@[\sigma_j \circ \sigma_k]_{j\in J, k\in K} & \\ & = & (e'@[\sigma_j \circ \sigma_k]_{j\in J,k\in K})\varrho & \text{(by induction)} \\ & = & (e'@([\sigma_j]_{j\in J} \circ [\sigma_k]_{k\in K}))\varrho & \\ & = & ((e'[\sigma_k]_{k\in K})@[\sigma_j]_{j\in J})\varrho & \end{array}$$

□

Lemma 7.4.6 ([Expression] substitution lemma). *Let $e, e_1, \ldots, e_n$ be expressions, $x_1, \ldots, x_n$ distinct variables, and $t, t_1, \ldots, t_n$ types. If $\Delta ; \Gamma, (x_1 : t_1), \ldots, (x_n : t_n) \vdash e : t$ and $\Delta ; \Gamma \vdash e_i : t_i$ for all i, then $\Delta ; \Gamma \vdash e\{e_1/x_1, \ldots, e_n/x_n\} : t$.*

Proof. By induction on the typing derivations for $\Delta ; \Gamma, (x_1 : t_1), \ldots, (x_n : t_n) \vdash e : t$. We simply "plug" a copy of the derivation for $\Delta ; \Gamma \vdash e_i : t_i$ wherever the rule (*var*) is used for variable x_i. For simplicity, in what follows, we write Γ' for $\Gamma, (x_1 : t_1), \ldots, (x_n : t_n)$ and ϱ for $\{e_1/x_1, \ldots, e_n/x_n\}$. We procede by a case analysis on the last applied rule.

$\underline{(const)}$**:** straightforward.

$\underline{(var)}$**:** $e = x$ and $\Delta ; \Gamma' \vdash x : \Gamma'(x)$.

If $x = x_i$, then $\Gamma'(x) = t_i$ and $x\varrho = e_i$. From the premise, we have $\Delta ; \Gamma \vdash e_i : t_i$. The result follows.

Otherwise, $\Gamma'(x) = \Gamma(x)$ and $x\varrho = x$. Clearly, we have $\Delta ; \Gamma \vdash x : \Gamma(x)$. Thus the result follows as well.

$\underline{(pair)}$**:** consider the following derivation:

$$\dfrac{\dfrac{\ldots}{\Delta ; \Gamma' \vdash e_1 : t_1} \quad \dfrac{\ldots}{\Delta ; \Gamma' \vdash e_1 : t_1}}{\Delta ; \Gamma' \vdash (e_1, e_2) : (t_1 \times t_2)}\ (pair)$$

By applying the induction hypothesis twice, we have $\Delta ; \Gamma \vdash e_i\varrho : t_i$. By (*pair*), we get $\Delta ; \Gamma \vdash (e_1\varrho, e_2\varrho) : (t_1 \times t_2)$, that is, $\Delta ; \Gamma \vdash (e_1, e_2)\varrho : (t_1 \times t_2)$.

$\underline{(proj)}$**:** consider the following derivation:

$$\dfrac{\dfrac{\ldots}{\Delta ; \Gamma' \vdash e' : t_1 \times t_2}}{\Delta ; \Gamma' \vdash \pi_i(e') : t_i}\ (proj)$$

By induction, we have $\Delta ; \Gamma \vdash e'\varrho : t_1 \times t_2$. Then the rule (*proj*) gives us $\Delta ; \Gamma \vdash \pi_i(e'\varrho) : t_i$, that is $\Delta ; \Gamma \vdash \pi_i(e')\varrho : t_i$.

$\underline{(abstr)}$**:** consider the following derivation:

$$\dfrac{\begin{array}{l}\dfrac{\ldots}{\forall i \in I, j \in J.\ \Delta' ; \Gamma', (x : t_i\sigma_j) \vdash e'@[\sigma_j] : s_i\sigma_j} \\ \Delta' = \Delta \cup \mathsf{var}(\bigwedge_{i\in I, j\in J} t_i\sigma_j \to s_i\sigma_j)\end{array}}{\Delta ; \Gamma' \vdash \lambda^{\wedge_{i\in I} t_i \to s_i}_{[\sigma_j]_{j\in J}} x.e' : \bigwedge_{i\in I, j\in J} t_i\sigma_j \to s_i\sigma_j}\ (abstr)$$

By α-conversion, we can ensure that $\mathsf{tv}(\varrho) \cap \bigcup_{j\in J} \mathsf{dom}(\sigma_j) = \emptyset$. By induction, we have $\Delta' \wr \Gamma, (x : t_i\sigma_j) \vdash (e'@[\sigma_j])\varrho : s_i\sigma_j$ for all $i \in I$ and $j \in J$. Because $\mathsf{tv}(\varrho) \cap \mathsf{dom}(\sigma_j) = \emptyset$, by Lemma 7.4.5, we get $\Delta' \wr \Gamma, (x : t_i\sigma_j) \vdash (e'\varrho)@[\sigma_j] : s_i\sigma_j$. Then by applying (*abstr*), we obtain $\Delta \wr \Gamma \vdash \lambda^{\wedge_{i\in I} t_i\to s_i}_{[\sigma_j]_{j\in J}} x.(e'\varrho) : \bigwedge_{i\in I, j\in J} t_i\sigma_j \to s_i\sigma_j$. That is, $\Delta \wr \Gamma \vdash (\lambda^{\wedge_{i\in I} t_i\to s_i}_{[\sigma_j]_{j\in J}} x.e)\varrho : \bigwedge_{i\in I, j\in J} t_i\sigma_j \to s_i\sigma_j$ (because $\mathsf{tv}(\varrho) \cap \bigcup_{j\in J} \mathsf{dom}(\sigma_j) = \emptyset$).

(*case*)**:** consider the following derivation:

$$\frac{\dfrac{\ldots}{\Delta \wr \Gamma' \vdash e_0 : t'} \quad \begin{cases} t' \not\leq \neg t & \Rightarrow \quad \dfrac{\ldots}{\Delta \wr \Gamma' \vdash e_1 : s} \\ t' \not\leq t & \Rightarrow \quad \dfrac{\ldots}{\Delta \wr \Gamma' \vdash e_2 : s} \end{cases}}{\Delta \wr \Gamma' \vdash (e_0 \in t \,?\, e_1 : e_2) : s}\ (case)$$

By induction, we have $\Delta \wr \Gamma \vdash e_0\varrho : t'$ and $\Delta \wr \Gamma \vdash e_i\varrho : s$ (for i such that $\Delta \wr \Gamma' \vdash e_i : s$ has been type-checked in the original derivation). Then the rule (*case*) gives us $\Delta \wr \Gamma \vdash (e_0\varrho \in t \,?\, e_1\varrho : e_2\varrho) : s$ that is $\Delta \wr \Gamma \vdash (e_0 \in t \,?\, e_1 : e_2)\varrho : s$.

(*inst*)**:**

$$\frac{\dfrac{\ldots}{\Delta \wr \Gamma' \vdash e' : s} \qquad \sigma \not\sharp \Delta}{\Delta \wr \Gamma' \vdash e'[\sigma] : s\sigma}\ (inst)$$

Using α-conversion on the polymorphic type variables of e, we can assume $\mathsf{tv}(\varrho) \cap \mathsf{dom}(\sigma) = \emptyset$. By induction, we have $\Delta \wr \Gamma \vdash e'\varrho : s$. Since $\sigma \not\sharp \Delta$, by applying (*inst*) we obtain $\Delta \wr \Gamma \vdash (e'\varrho)[\sigma] : s\sigma$, that is, $\Delta \wr \Gamma \vdash (e'[\sigma])\varrho : s\sigma$ because $\mathsf{tv}(\varrho) \cap \mathsf{dom}(\sigma) = \emptyset$.

(*inter*)**:**

$$\frac{\forall j \in J.\ \dfrac{\ldots}{\Delta \wr \Gamma' \vdash e'[\sigma_j] : t_j}}{\Delta \wr \Gamma' \vdash e'[\sigma_j]_{j\in J} : \bigwedge_{j\in J} t_j}\ (inter)$$

By induction, for all $j \in J$ we have $\Delta \wr \Gamma \vdash (e'[\sigma_j])\varrho : t_j$, that is $\Delta \wr \Gamma \vdash (e'\varrho)[\sigma_j] : t_j$. Then by applying (*inter*) we get $\Delta \wr \Gamma \vdash (e'\varrho)[\sigma_j]_{j\in J} : \bigwedge_{j\in J} t_j$, that is $\Delta \wr \Gamma \vdash (e'[\sigma_j]_{j\in J})\varrho : \bigwedge_{j\in J} t_j$.

(*subsum*)**:** consider the following derivation:

$$\frac{\dfrac{\ldots}{\Delta \wr \Gamma' \vdash e' : s} \qquad s \leq t}{\Delta \wr \Gamma' \vdash e' : t}\ (subsum)$$

By induction, we have $\Delta \wr \Gamma \vdash e'\varrho : s$. Then the rule (*subsum*) gives us $\Delta \wr \Gamma \vdash e'\varrho : t$.

□

Definition 7.4.7. *Given two typing environments Γ_1, Γ_2, we define their* intersection *as*

$$(\Gamma_1 \wedge \Gamma_2)(x) = \begin{cases} \Gamma_1(x) \wedge \Gamma_2(x) & \text{if } x \in \mathsf{dom}(\Gamma_1) \cap \mathsf{dom}(\Gamma_2) \\ \text{undefined} & \text{otherwise} \end{cases}$$

We define $\Gamma_2 \leq \Gamma_1$ *if* $\Gamma_2(x) \leq \Gamma_1(x)$ *for all* $x \in \mathsf{dom}(\Gamma_1)$*, and* $\Gamma_1 \simeq \Gamma_2$ *if* $\Gamma_1 \leq \Gamma_2$ *and* $\Gamma_2 \leq \Gamma_1$.

Given an expression e and a set Δ of (monomorphic) type variables, we write $e \sharp \Delta$ if $\sigma_j \sharp \Delta$ for all the type substitution σ_j that occur in a subterm of e of the form $e'[\sigma_j]_{j\in J}$ (in other terms, we do not consider the substitutions that occur in the decorations of λ-abstractions).

Lemma 7.4.8 (Weakening). *Let* e *be an expression,* Γ, Γ' *two typing environments and* Δ' *a set of type variables. If* $\Delta \fatsemi \Gamma \vdash e : t$*,* $\Gamma' \leq \Gamma$ *and* $e \sharp \Delta'$*, then* $\Delta \cup \Delta' \fatsemi \Gamma' \vdash e : t$.

Proof. By induction on the derivation of $\Delta \fatsemi \Gamma \vdash e : t$. We perform a case analysis on the last applied rule.

(*const*)**:** straightforward.

(*var*)**:** $\Delta \fatsemi \Gamma \vdash x : \Gamma(x)$. It is clear that $\Delta \cup \Delta' \fatsemi \Gamma' \vdash x : \Gamma'(x)$ by (*var*). Since $\Gamma'(x) \leq \Gamma(x)$, by (*subsum*), we get $\Delta \cup \Delta' \fatsemi \Gamma' \vdash x : \Gamma(x)$.

(*pair*)**:** consider the following derivation:

$$\frac{\dfrac{\cdots}{\Delta \fatsemi \Gamma \vdash e_1 : t_1} \quad \dfrac{\cdots}{\Delta \fatsemi \Gamma \vdash e_2 : t_2}}{\Delta \fatsemi \Gamma \vdash (e_1, e_2) : t_1 \times t_2}\ (pair)$$

By applying the induction hypothesis twice, we have $\Delta \cup \Delta' \fatsemi \Gamma' \vdash e_i : t_i$. Then by (*pair*), we get $\Delta \cup \Delta' \fatsemi \Gamma' \vdash (e_1, e_2) : t_1 \times t_2$.

(*proj*)**:** consider the following derivation:

$$\frac{\dfrac{\cdots}{\Delta \fatsemi \Gamma \vdash e' : t_1 \times t_2}}{\Delta \fatsemi \Gamma \vdash \pi_i(e') : t_i}\ (proj)$$

By the induction hypothesis, we have $\Delta \cup \Delta' \fatsemi \Gamma' \vdash e' : t_1 \times t_2$. Then by (*proj*), we get $\Delta \cup \Delta' \fatsemi \Gamma' \vdash \pi_i(e') : t_i$.

(*abstr*)**:** consider the following derivation:

$$\frac{\begin{array}{l}\forall i \in I, j \in J.\ \dfrac{\cdots}{\Delta'' \fatsemi \Gamma, (x : t_i\sigma_j) \vdash e'@[\sigma_j] : s_i\sigma_j} \\ \Delta'' = \Delta \cup \mathsf{var}(\bigwedge_{i\in I, j\in J} t_i\sigma_j \to s_i\sigma_j)\end{array}}{\Delta \fatsemi \Gamma \vdash \lambda^{\bigwedge_{i\in I} t_i \to s_i}_{[\sigma_j]_{j\in J}} x.e' : \bigwedge_{i\in I, j\in J} t_i\sigma_j \to s_i\sigma_j}\ (abstr)$$

By induction, we have $\Delta'' \cup \Delta' \fatsemi \Gamma', (x : t_i\sigma_j) \vdash e'@[\sigma_j] : s_i\sigma_j$ for all $i \in I$ and $j \in J$. Then by (*abstr*), we get $\Delta \cup \Delta' \fatsemi \Gamma' \vdash \lambda^{\bigwedge_{i\in I} t_i \to s_i}_{[\sigma_j]_{j\in J}} x.e' : \bigwedge_{i\in I, j\in J} t_i\sigma_j \to s_i\sigma_j$.

(*case*)**:** consider the following derivation:

$$\frac{\dfrac{\cdots}{\Delta \fatsemi \Gamma \vdash e_0 : t'} \quad \begin{cases} t' \not\leq \neg t & \Rightarrow \ \dfrac{\cdots}{\Delta \fatsemi \Gamma \vdash e_1 : s} \\ t' \not\leq t & \Rightarrow \ \dfrac{\cdots}{\Delta \fatsemi \Gamma \vdash e_2 : s} \end{cases}}{\Delta \fatsemi \Gamma \vdash (e_0 \in t\,?\,e_1 : e_2) : s}\ (case)$$

By induction, we have $\Delta \cup \Delta' \fatsemi \Gamma' \vdash e_0 : t_0$ and $\Delta \cup \Delta' \fatsemi \Gamma' \vdash e_i : s$ (for i such that e_i has been type-checked in the original derivation). Then by (*case*), we get $\Delta \cup \Delta' \fatsemi \Gamma' \vdash (e_0 \in t\,?\,e_1 : e_2) : s$.

(*inst*)**:** consider the following derivation:

$$\frac{\dfrac{\cdots}{\Delta \wr \Gamma \vdash e' : s} \quad \sigma \sharp \Delta}{\Delta \wr \Gamma \vdash e'[\sigma] : s\sigma}\ (inst)$$

By induction, we have $\Delta \cup \Delta' \wr \Gamma' \vdash e' : s$. Since $e \sharp \Delta'$(*i.e.*, $e'[\sigma] \sharp \Delta'$), we have $\sigma \sharp \Delta'$. Then $\sigma \sharp \Delta \cup \Delta'$. Therefore, by applying (*inst*) we get $\Delta \cup \Delta' \wr \Gamma' \vdash e'[\sigma] : s\sigma$.

(*inter*)**:** consider the following derivation:

$$\frac{\forall j \in J.\ \dfrac{\cdots}{\Delta \wr \Gamma \vdash e'[\sigma_j] : t_j}}{\Delta \wr \Gamma \vdash e'[\sigma_j]_{j \in J} : \bigwedge_{j \in J} t_j}\ (inter)$$

By induction, we have $\Delta \cup \Delta' \wr \Gamma' \vdash e'[\sigma_j] : t_j$ for all $j \in J$. Then the rule (*inst*) gives us $\Delta \cup \Delta' \wr \Gamma' \vdash e'[\sigma_j]_{j \in J} : \bigwedge_{j \in J} t_j$.

(*subsum*)**:** there exists a type s such that

$$\frac{\dfrac{\cdots}{\Delta \wr \Gamma \vdash e' : s} \quad s \leq t}{\Delta \wr \Gamma \vdash e' : t}\ (subsum)$$

By induction, we have $\Delta \cup \Delta' \wr \Gamma' \vdash e' : s$. Then by applying the rule (*subsum*) we get $\Delta \cup \Delta' \wr \Gamma' \vdash e' : t$.

□

Definition 7.4.9. *Let σ_1 and σ_2 be two substitutions such that $\mathsf{dom}(\sigma_1) \cap \mathsf{dom}(\sigma_2) = \emptyset$ ($\sigma_1 \sharp \sigma_2$ for short). Their* union *$\sigma_1 \cup \sigma_2$ is defined as*

$$(\sigma_1 \cup \sigma_2)(\alpha) = \begin{cases} \sigma_1(\alpha) & \alpha \in \mathsf{dom}(\sigma_1) \\ \sigma_2(\alpha) & \alpha \in \mathsf{dom}(\sigma_2) \\ \alpha & \textit{otherwise} \end{cases}$$

The next two lemmas are used to simplify sets of type-substitutions applied to expressions when they are redundant or they work on variables that are not in the expressions.

Lemma 7.4.10 (Useless Substitutions). *Let e be an expression and $[\sigma_k]_{k \in K}$, $[\sigma'_k]_{k \in K}$ two sets of substitutions such that $\sigma'_k \sharp \sigma_k$ and $\mathsf{dom}(\sigma'_k) \cap \mathsf{tv}(e) = \emptyset$ for all $k \in K$. Then*

$$\Delta \wr \Gamma \vdash e@[\sigma_k]_{k \in K} : t \iff \Delta \wr \Gamma \vdash e@[\sigma_k \cup \sigma'_k]_{k \in K} : t$$

Proof. Straightforward. □

Henceforth we use "$\uplus$" to denote the union of multi-sets (*e.g.*, $\{1,2\} \uplus \{1,3\} = \{1,2,1,3\}$).

Lemma 7.4.11 (Redundant Substitutions). *Let $[\sigma_j]_{j \in J}$ and $[\sigma_j]_{j \in J'}$ be two sets of substitutions such that $J' \subseteq J$. Then*

$$\Delta \wr \Gamma \vdash e@[\sigma_j]_{j \in J \uplus J'} : t \iff \Delta \wr \Gamma \vdash e@[\sigma_j]_{j \in J} : t$$

Proof. Similar to Lemma 7.4.10. □

Lemma 7.4.10 states that if a type variable α in the domain of a type substitution σ does not occur in the applied expression e, namely, $\alpha \in \mathsf{dom}(\sigma) \setminus \mathsf{tv}(e)$, then that part of the substitution is useless and can be safely eliminated. Lemma 7.4.11 states that although our $[\sigma_j]_{j\in J}$ are formally multisets of type-substitutions, in practice they behave as sets, since repeated entries of type substitutions can be safely removed. Therefore, to simplify an expression without altering its type (and semantics), we first eliminate the useless type variables, yielding concise type substitutions, and then remove the redundant type substitutions. This is useful especially for the translation from our calculus to the core$\mathbb{C}$Duce, since we need fewer type-case branches to encode abstractions (see Chapter 11 for more details). It also explains why we do not apply relabeling when the domains of the type substitutions do not contain type variables in expressions in Definition 7.1.6.

Moreover, Lemma 7.4.11 also indicates that it is safe to keep only the type substitutions which are different from each other when we merge two sets of substitutions (*e.g.* Lemmas 7.4.14 and 7.4.15). In what follows, without explicit mention, we assume that there are no useless type variables in the domain of any type substitution and no redundant type substitutions in any set of type substitutions.

Lemma 7.4.12 (Relabeling). *Let e be an expression, $[\sigma_j]_{j\in J}$ a set of type substitutions and Δ a set of type variables such that $\sigma_j \sharp \Delta$ for all $j \in J$. If $\Delta \fatsemi \Gamma \vdash e : t$, then*

$$\Delta \fatsemi \Gamma \vdash e@[\sigma_j]_{j\in J} : \bigwedge_{j\in J} t\sigma_j$$

Proof. The proof proceeds by induction and case analysis on the structure of e. For each case we use an auxiliary internal induction on the typing derivation. We label **E** the main (external) induction and **I** the internal induction in what follows.

$\underline{e = c}$**:** the typing derivation $\Delta \fatsemi \Gamma \vdash e : t$ should end with either (*const*) or (*subsum*). Assume that the typing derivation ends with (*const*). Trivially, we have $\Delta \fatsemi \Gamma \vdash c : b_c$. Since $c@[\sigma_j]_{j\in J} = c$ and $b_c \simeq \bigwedge_{j\in J} b_c\sigma_j$, by subsumption, we have $\Delta \fatsemi \Gamma \vdash c@[\sigma_j]_{j\in J} : \bigwedge_{j\in J} b_c\sigma_j$.
Otherwise, the typing derivation ends with an instance of (*subsum*):

$$\frac{\dfrac{\cdots}{\Delta \fatsemi \Gamma \vdash e : s} \qquad s \leq t}{\Delta \fatsemi \Gamma \vdash e : t}\ (subsum)$$

Then by **I**-induction, we have $\Delta \fatsemi \Gamma \vdash e@[\sigma_j]_{j\in J} : \bigwedge_{j\in J} s\sigma_j$. Since $s \leq t$, we get $\bigwedge_{j\in J} s\sigma_j \leq \bigwedge_{j\in J} t\sigma_j$. Then by applying the rule (*subsum*), we have $\Delta \fatsemi \Gamma \vdash e@[\sigma_j]_{j\in J} : \bigwedge_{j\in J} t\sigma_j$.

$\underline{e = x}$**:** the typing derivation $\Delta \fatsemi \Gamma \vdash e : t$ should end with either (*var*) or (*subsum*). Assume that the typing derivation ends with (*var*). Trivially, by (*var*), we get $\Delta \fatsemi \Gamma \vdash x : \Gamma(x)$. Moreover, we have $x@[\sigma_j]_{j\in J} = x$ and $\Gamma(x) = \bigwedge_{j\in J} \Gamma(x)\sigma_j$ (as $\mathsf{var}(\Gamma) \subseteq \Delta$). Therefore, we deduce that $\Delta \fatsemi \Gamma \vdash x@[\sigma_j]_{j\in J} : \bigwedge_{j\in J} \Gamma(x)\sigma_j$.
Otherwise, the typing derivation ends with an instance of (*subsum*), similar to the case of $e = c$, the result follows by **I**-induction.

$\underline{e = (e_1, e_2)}$**:** the typing derivation $\Delta \fatsemi \Gamma \vdash e : t$ should end with either (*pair*) or (*subsum*). Assume that the typing derivation ends with (*pair*):

$$\frac{\overset{\cdots}{\Delta \fatsemi \Gamma \vdash e_1 : t_1} \quad \overset{\cdots}{\Delta \fatsemi \Gamma \vdash e_2 : t_2}}{\Delta \fatsemi \Gamma \vdash (e_1, e_2) : t_1 \times t_2}\ (pair)$$

By **E**-induction, we have $\Delta \fatsemi \Gamma \vdash e_i@[\sigma_j]_{j\in J} : \bigwedge_{j\in J} t_i\sigma_j$. Then by (*pair*), we get $\Delta \fatsemi \Gamma \vdash (e_1@[\sigma_j]_{j\in J}, e_2@[\sigma_j]_{j\in J}) : (\bigwedge_{j\in J} t_1\sigma_j \times \bigwedge_{j\in J} t_2\sigma_j)$, that is, $\Delta \fatsemi \Gamma \vdash (e_1, e_2)@[\sigma_j]_{j\in J} : \bigwedge_{j\in J}(t_1 \times t_2)\sigma_j$.
Otherwise, the typing derivation ends with an instance of (*subsum*), similar to the case of $e = c$, the result follows by **I**-induction.

$\underline{e = \pi_i(e')}$**:** the typing derivation $\Delta \fatsemi \Gamma \vdash e : t$ should end with either (*proj*) or (*subsum*). Assume that the typing derivation ends with (*proj*):

$$\frac{\overset{\cdots}{\Delta \fatsemi \Gamma \vdash e' : t_1 \times t_2}}{\Delta \fatsemi \Gamma \vdash \pi_i(e') : t_i}\ (proj)$$

By **E**-induction, we have $\Delta \fatsemi \Gamma \vdash e'@[\sigma_j]_{j\in J} : \bigwedge_{j\in J}(t_1 \times t_2)\sigma_j$, that is, $\Delta \fatsemi \Gamma \vdash e'@[\sigma_j]_{j\in J} : (\bigwedge_{j\in J} t_1\sigma_j \times \bigwedge_{j\in J} t_2\sigma_j)$. Then the rule (*proj*) gives us that $\Delta \fatsemi \Gamma \vdash \pi_i(e'@[\sigma_j]_{j\in J}) : \bigwedge_{j\in J} t_i\sigma_j$, that is, $\Delta \fatsemi \Gamma \vdash \pi_i(e')@[\sigma_j]_{j\in J} : \bigwedge_{j\in J} t_i\sigma_j$.
Otherwise, the typing derivation ends with an instance of (*subsum*), similar to the case of $e = c$, the result follows by **I**-induction.

$\underline{e = e_1e_2}$**:** the typing derivation $\Delta \fatsemi \Gamma \vdash e : t$ should end with either (*appl*) or (*subsum*). Assume that the typing derivation ends with (*appl*):

$$\frac{\overset{\cdots}{\Delta \fatsemi \Gamma \vdash e_1 : t \to s} \quad \overset{\cdots}{\Delta \fatsemi \Gamma \vdash e_2 : t}}{\Delta \fatsemi \Gamma \vdash e_1e_2 : s}\ (pair)$$

By **E**-induction, we have $\Delta \fatsemi \Gamma \vdash e_1@[\sigma_j]_{j\in J} : \bigwedge_{j\in J}(t \to s)\sigma_j$ and $\Delta \fatsemi \Gamma \vdash e_2@[\sigma_j] : \bigwedge_{j\in J} t\sigma_j$. Since $\bigwedge_{j\in J}(t \to s)\sigma_j \leq (\bigwedge_{j\in J} t\sigma_j) \to (\bigwedge_{j\in J} s\sigma_j)$, by (*subsum*), we have $\Delta \fatsemi \Gamma \vdash e_1@[\sigma_j]_{j\in J} : (\bigwedge_{j\in J} t\sigma_j) \to (\bigwedge_{j\in J} s\sigma_j)$. Then by (*appl*), we get

$$\Delta \fatsemi \Gamma \vdash (e_1@[\sigma_j]_{j\in J})(e_2@[\sigma_j]_{j\in J}) : \bigwedge_{j\in J} s\sigma_j$$

that is, $\Delta \fatsemi \Gamma \vdash (e_1e_2)@[\sigma_j]_{j\in J} : \bigwedge_{j\in J} s\sigma_j$.
Otherwise, the typing derivation ends with an instance of (*subsum*), similar to the case of $e = c$, the result follows by **I**-induction.

$\underline{e = \lambda^{\wedge_{i\in I} t_i \to s_i}_{[\sigma_k]_{k\in K}} x.e'}$**:** the typing derivation $\Delta \fatsemi \Gamma \vdash e : t$ should end with either (*abstr*) or (*subsum*). Assume that the typing derivation ends with (*abstr*):

$$\frac{\begin{array}{l}\forall i \in I, k \in K.\ \overset{\cdots}{\Delta' \fatsemi \Gamma, (x : t_i\sigma_k) \vdash e'@[\sigma_k] : s_i\sigma_k} \\ \Delta' = \Delta \cup \mathsf{var}(\bigwedge_{i\in I,k\in K} t_i\sigma_k \to s_i\sigma_k)\end{array}}{\Delta \fatsemi \Gamma \vdash \lambda^{\wedge_{i\in I} t_i \to s_i}_{[\sigma_k]_{k\in K}} x.e' : \bigwedge_{i\in I,k\in K} t_i\sigma_k \to s_i\sigma_k}\ (abstr)$$

Using α-conversion, we can assume that $\sigma_j \sharp (\mathsf{var}(\bigwedge_{i\in I,k\in K} t_i\sigma_k \to s_i\sigma_k) \setminus \Delta)$ for $j \in J$. Hence $\sigma_j \sharp \Delta'$. By **E**-induction, we have

$$\Delta' \fatsemi \Gamma, (x : (t_i\sigma_k)) \vdash (e'@[\sigma_k])@[\sigma_j] : (s_i\sigma_k)\sigma_j$$

for all $i \in I$, $k \in K$ and $j \in J$. By Lemma 7.4.4, $(e'@[\sigma_k])@[\sigma_j] = e'@([\sigma_j] \circ [\sigma_k])$. So

$$\Delta' \fatsemi \Gamma, (x : (t_i\sigma_k)) \vdash e'@([\sigma_j] \circ [\sigma_k]) : (s_i\sigma_k)\sigma_j$$

Finally, by (*abstr*), we get

$$\Delta \fatsemi \Gamma \vdash \lambda^{\wedge_{i\in I} t_i \to s_i}_{[\sigma_j \circ \sigma_k]_{k\in K, j\in J}} x.e' : \bigwedge_{i\in I, k\in K, j\in J} t_i(\sigma_j \circ \sigma_k) \to s_i(\sigma_k \circ \sigma_j)$$

that is,

$$\Delta \fatsemi \Gamma \vdash (\lambda^{\wedge_{i\in I} t_i \to s_i}_{[\sigma_k]_{k\in K}} x.e')@[\sigma_j]_{j\in J} : \bigwedge_{j\in J} (\bigwedge_{i\in I, k\in K} t_i\sigma_k \to s_i\sigma_k)\sigma_j$$

Otherwise, the typing derivation ends with an instance of (*subsum*), similar to the case of $e = c$, the result follows by **I**-induction.

$\underline{e = e' \in t \,?\, e_1 : e_2}$**:** the typing derivation $\Delta \fatsemi \Gamma \vdash e : t$ should end with either (*case*) or (*subsum*). Assume that the typing derivation ends with (*case*):

$$\dfrac{\dfrac{\ldots}{\Delta \fatsemi \Gamma \vdash e' : t'} \quad \begin{cases} t' \not\leq \neg t & \Rightarrow \quad \dfrac{\ldots}{\Delta \fatsemi \Gamma \vdash e_1 : s} \\ t' \not\leq t & \Rightarrow \quad \dfrac{\ldots}{\Delta \fatsemi \Gamma \vdash e_2 : s} \end{cases}}{\Delta \fatsemi \Gamma \vdash (e' \in t \,?\, e_1 : e_2) : s}\ (case)$$

By **E**-induction, we have $\Delta \fatsemi \Gamma \vdash e'@[\sigma_j]_{j\in J} : \bigwedge_{j\in J} t'\sigma_j$. Suppose $\bigwedge_{j\in J} t'\sigma_j \not\leq \neg t$; then we must have $t' \not\leq \neg t$, and the branch for e_1 has been type-checked. By the **E**-induction hypothesis, we have $\Delta \fatsemi \Gamma \vdash e_1@[\sigma_j]_{j\in J} : \bigwedge_{j\in J} s\sigma_j$. Similarly, if $\bigwedge_{j\in J} t'\sigma_j \not\leq t$, then the second branch e_2 has been type-checked, and we have $\Delta \fatsemi \Gamma \vdash e_2@[\sigma_j]_{j\in J} : \bigwedge_{j\in J} s\sigma_j$ by the **E**-induction hypothesis. By (*case*), we have

$$\Delta \fatsemi \Gamma \vdash (e'@[\sigma_j]_{j\in J} \in t \,?\, e_1@[\sigma_j]_{j\in J} : e_2@[\sigma_j]_{j\in J}) : \bigwedge_{j\in J} s\sigma_j$$

that is $\Delta \fatsemi \Gamma \vdash (e' \in t \,?\, e_1 : e_2)@[\sigma_j]_{j\in J} : \bigwedge_{j\in J} s\sigma_j$.

Otherwise, the typing derivation ends with an instance of (*subsum*), similar to the case of $e = c$, the result follows by **I**-induction.

$\underline{e = e'[\sigma]}$**:** the typing derivation $\Delta \fatsemi \Gamma \vdash e : t$ should end with either (*inst*) or (*subsum*). Assume that the typing derivation ends with (*inst*):

$$\dfrac{\dfrac{\ldots}{\Delta \fatsemi \Gamma \vdash e' : t} \quad \sigma \nsharp \Delta}{\Delta \fatsemi \Gamma \vdash e'[\sigma] : t\sigma}\ (inst)$$

Consider the set of substitutions $[\sigma_j \circ \sigma]_{j\in J}$. It is clear that $\sigma_j \circ \sigma \nsharp \Delta$ for all $j \in J$. By **E**-induction, we have

$$\Delta \fatsemi \Gamma \vdash e'@[\sigma_j \circ \sigma]_{j\in J} : \bigwedge_{j\in J} t(\sigma_j \circ \sigma)$$

that is, $\Delta \fatsemi \Gamma \vdash (e'[\sigma])@[\sigma_j]_{j\in J} : \bigwedge_{j\in J} (t\sigma)\sigma_j$.

Otherwise, the typing derivation ends with an instance of (*subsum*), similar to the case of $e = c$, the result follows by **I**-induction.

$\underline{e = e'[\sigma_k]_{k\in K}}$**:** the typing derivation $\Delta \fatsemi \Gamma \vdash e : t$ should end with either (*inter*) or (*subsum*). Assume that the typing derivation ends with (*inter*):

$$\frac{\forall k \in K.\ \overline{\Delta \fatsemi \Gamma \vdash e'[\sigma_k] : t_k}^{\ \dots}}{\Delta \fatsemi \Gamma \vdash e'[\sigma_k]_{k\in K} : \bigwedge_{k\in K} t_k}\ (inter)$$

As an intermediary result, we first prove that the derivation can be rewritten as

$$\dfrac{\dfrac{\forall k \in K.\ \dfrac{\overline{\Delta \fatsemi \Gamma \vdash e' : s}^{\ \dots} \quad \sigma_k \sharp \Delta}{\Delta \fatsemi \Gamma \vdash e'[\sigma_k] : s\sigma_k}\ (inst)}{\Delta \fatsemi \Gamma \vdash e'[\sigma_k]_{k\in K} : \bigwedge_{k\in K} s\sigma_k}\ (inter) \qquad \bigwedge_{k\in K} s\sigma_k \leq \bigwedge_{k\in K} t_k}{\Delta \fatsemi \Gamma \vdash e'[\sigma_k]_{k\in K} : \bigwedge_{k\in K} t_k}\ (subsum)$$

We proceed by induction on the original derivation. It is clear that each sub-derivation $\Delta \fatsemi \Gamma \vdash e'[\sigma_k] : t_k$ ends with either (*inst*) or (*subsum*). If all the sub-derivations end with an instance of (*inst*), then for all $k \in K$, we have

$$\frac{\overline{\Delta \fatsemi \Gamma \vdash e' : s_k}^{\ \dots} \quad \sigma_k \sharp \Delta}{\Delta \fatsemi \Gamma \vdash e'[\sigma_k] : s_k\sigma_k}\ (inst)$$

By Lemma 7.4.2, we have $\Delta \fatsemi \Gamma \vdash e' : \bigwedge_{k\in K} s_k$. Let $s = \bigwedge_{k\in K} s_k$. Then by (*inst*), we get $\Delta \fatsemi \Gamma \vdash e'[\sigma_k] : s\sigma_k$. Finally, by (*inter*) and (*subsum*), the intermediary result holds. Otherwise, there is at least one of the sub-derivations ends with an instance of (*subsum*), the intermediary result also hold by induction.
Now that the intermediary result is proved, we go back to the proof of the lemma. Consider the set of substitutions $[\sigma_j \circ \sigma_k]_{j\in J,k\in K}$. It is clear that $\sigma_j \circ \sigma_k \sharp \Delta$ for all $j \in J, k \in K$. By **E**-induction on e' (*i.e.*, $\Delta \fatsemi \Gamma \vdash e' : s$), we have

$$\Delta \fatsemi \Gamma \vdash e'@[\sigma_j \circ \sigma_k]_{j\in J,k\in K} : \bigwedge_{j\in J,k\in K} s(\sigma_j \circ \sigma_k)$$

that is, $\Delta \fatsemi \Gamma \vdash (e'[\sigma_k]_{k\in K})@[\sigma_j]_{j\in J} : \bigwedge_{j\in J}(\bigwedge_{k\in K} s\sigma_k)\sigma_j$. As $\bigwedge_{k\in K} s\sigma_k \leq \bigwedge_{k\in K} t_k$, we get $\bigwedge_{j\in J}(\bigwedge_{k\in K} s\sigma_k)\sigma_j \leq \bigwedge_{j\in J}(\bigwedge_{k\in K} t_k)\sigma_j$. Then by (*subsum*), the result follows.
Otherwise, the typing derivation ends with an instance of (*subsum*), similar to the case of $e = c$, the result follows by **I**-induction.

□

Corollary 7.4.13. *If* $\Delta \fatsemi \Gamma \vdash e[\sigma_j]_{j\in J} : t$*, then* $\Delta \fatsemi \Gamma \vdash e@[\sigma_j]_{j\in J} : t$*.*

Proof. Immediate consequence of Lemma 7.4.12. □

Lemma 7.4.14. *If* $\Delta \fatsemi \Gamma \vdash e@[\sigma_j]_{j\in J} : t$ *and* $\Delta' \fatsemi \Gamma' \vdash e@[\sigma_j]_{j\in J'} : t'$*, then* $\Delta \cup \Delta' \fatsemi \Gamma \wedge \Gamma' \vdash e@[\sigma_j]_{j\in J\cup J'} : t \wedge t'$

Proof. The proof proceeds by induction and case analysis on the structure of e. For each case we use an auxiliary internal induction on both typing derivations. We label **E** the main (external) induction and **I** the internal induction in what follows.

$\underline{e = c}$**:** $e@[\sigma_j]_{j\in J} = e@[\sigma_j]_{j\in J'} = c$. Clearly, both typing derivations should end with either (*const*) or (*subsum*). Assume that both derivations end with (*const*):

$$\frac{}{\Delta \wr \Gamma \vdash c : b_c}\ (const) \qquad \frac{}{\Delta' \wr \Gamma' \vdash c : b_c}\ (const)$$

Trivially, by (*const*) we have $\Delta \cup \Delta' \wr \Gamma \wedge \Gamma' \vdash c : b_c$, that is $\Delta \cup \Delta' \wr \Gamma \wedge \Gamma' \vdash e@[\sigma_j]_{j\in J\cup J'} : b_c$. As $b_c \simeq b_c \wedge b_c$, by (*subsum*), the result follows.
Otherwise, there exists at least one typing derivation which ends with an instance of (*subsum*), for instance,

$$\frac{\dfrac{\cdots}{\Delta \wr \Gamma \vdash e@[\sigma_j]_{j\in J} : s} \quad s \leq t}{\Delta \wr \Gamma \vdash e@[\sigma_j]_{j\in J} : t}\ (subsum)$$

Then by **I**-induction on $\Delta \wr \Gamma \vdash e@[\sigma_j]_{j\in J} : s$ and $\Delta' \wr \Gamma' \vdash e@[\sigma_j]_{j\in J'} : t'$, we have

$$\Delta \cup \Delta' \wr \Gamma \wedge \Gamma' \vdash e@[\sigma_j]_{j\in J\cup J'} : s \wedge t'$$

Since $s \leq t$, we have $s \wedge t' \leq t \wedge t'$. By (*subsum*), the result follows as well.

$\underline{e = x}$**:** $e@[\sigma_j]_{j\in J} = e@[\sigma_j]_{j\in J'} = x$. Clearly, both typing derivations should end with either (*var*) or (*subsum*). Assume that both derivations end with an instance of (*var*):

$$\frac{}{\Delta \wr \Gamma \vdash x : \Gamma(x)}\ (var) \qquad \frac{}{\Delta' \wr \Gamma' \vdash x : \Gamma'(x)}\ (var)$$

Since $x \in \mathsf{dom}(\Gamma)$ and $x \in \mathsf{dom}(\Gamma')$, $x \in \mathsf{dom}(\Gamma \wedge \Gamma')$. By (*var*), we have $\Delta \cup \Delta' \wr \Gamma \wedge \Gamma' \vdash x : (\Gamma \wedge \Gamma')(x)$, that is, $\Delta \cup \Delta' \wr \Gamma \wedge \Gamma' \vdash e@[\sigma_j]_{j\in J\cup J'} : \Gamma(x) \wedge \Gamma'(x)$.
Otherwise, there exists at least one typing derivation which ends with an instance of (*subsum*), similar to the case of $e = c$, the result follows by **I**-induction.

$\underline{e = (e_1, e_2)}$**:** $e@[\sigma_j]_{j\in J} = (e_1@[\sigma_j]_{j\in J}, e_2@[\sigma_j]_{j\in J})$ and $e@[\sigma_j]_{j\in J'} = (e_1@[\sigma_j]_{j\in J'}, e_2@[\sigma_j]_{j\in J'})$. Clearly, both typing derivations should end with either (*pair*) or (*subsum*). Assume that both derivations end with an instance of (*pair*):

$$\frac{\dfrac{\cdots}{\Delta \wr \Gamma \vdash e_1@[\sigma_j]_{j\in J} : s_1} \quad \dfrac{\cdots}{\Delta \wr \Gamma \vdash e_2@[\sigma_j]_{j\in J} : s_2}}{\Delta \wr \Gamma \vdash (e_1@[\sigma_j]_{j\in J}, e_2@[\sigma_j]_{j\in J}) : (s_1 \times s_2)}\ (pair)$$

$$\frac{\dfrac{\cdots}{\Delta' \wr \Gamma' \vdash e_1@[\sigma_j]_{j\in J'} : s'_1} \quad \dfrac{\cdots}{\Delta' \wr \Gamma' \vdash e_2@[\sigma_j]_{j\in J'} : s'_2}}{\Delta' \wr \Gamma' \vdash (e_1@[\sigma_j]_{j\in J'}, e_2@[\sigma_j]_{j\in J'}) : (s'_1 \times s'_2)}\ (pair)$$

By **E**-induction, we have $\Delta \cup \Delta' \wr \Gamma \wedge \Gamma' \vdash e_i@[\sigma_j]_{j\in J\cup J'} : s_i \wedge s'_i$. Then by (*pair*), we get $\Delta \cup \Delta' \wr \Gamma \wedge \Gamma' \vdash (e_1@[\sigma_j]_{j\in J\cup J'}, e_2@[\sigma_j]_{j\in J\cup J'}) : (s_1 \wedge s'_1) \times (s_2 \wedge s'_2)$, that is $\Delta \cup \Delta' \wr \Gamma \wedge \Gamma' \vdash (e_1, e_2)@[\sigma_j]_{j\in J\cup J'} : (s_1 \wedge s'_1) \times (s_2 \wedge s'_2)$. Moreover, because intersection distribute over product, we have $(s_1 \wedge s'_1) \times (s_2 \wedge s'_2) \simeq (s_1 \times s_2) \wedge (s'_1 \times s'_2)$. Finally, by applying (*subsum*), we have $\Delta \cup \Delta' \wr \Gamma \wedge \Gamma' \vdash (e_1, e_2)@[\sigma_j]_{j\in J\cup J'} : (s_1 \times s_2) \wedge (s'_1 \times s'_2)$.
Otherwise, there exists at least one typing derivation which ends with an instance of (*subsum*), similar to the case of $e = c$, the result follows by **I**-induction.

$\underline{e = \pi_i(e')}$**:** $e@[\sigma_j]_{j\in J} = \pi_i(e'@[\sigma_j]_{j\in J})$ and $e@[\sigma_j]_{j\in J'} = \pi_i(e'@[\sigma_j]_{j\in J'})$, where $i = 1, 2$. Clearly, both typing derivations should end with either (*proj*) or (*subsum*). Assume that both derivations end with an instance of (*proj*):

$$\frac{\dfrac{\cdots}{\Delta \wp \Gamma \vdash e'@[\sigma_j]_{j\in J} : s_1 \times s_2}}{\Delta \wp \Gamma \vdash \pi_i(e'@[\sigma_j]_{j\in J}) : s_i}\ (proj) \qquad \frac{\dfrac{\cdots}{\Delta' \wp \Gamma' \vdash e'@[\sigma_j]_{j\in J'} : s'_1 \times s'_2}}{\Delta' \wp \Gamma' \vdash \pi_i(e'@[\sigma_j]_{j\in J'}) : s'_i}\ (proj)$$

By **E**-induction, we have $\Delta \cup \Delta' \wp \Gamma \wedge \Gamma' \vdash e'@[\sigma_j]_{j\in J\cup J'} : (s_1 \times s_2) \wedge (s'_1 \times s'_2)$. Since $(s_1 \times s_2) \wedge (s'_1 \times s'_2) \simeq (s_1 \wedge s'_1) \times (s_2 \wedge s'_2)$ (See the case of $e = (e_1, e_2)$), by (*subsum*), we have $\Delta \cup \Delta' \wp \Gamma \wedge \Gamma' \vdash e'@[\sigma_j]_{j\in J\cup J'} : (s_1 \wedge s'_1) \times (s_2 \wedge s'_2)$. Finally, by applying (*proj*), we get $\Delta \cup \Delta' \wp \Gamma \wedge \Gamma' \vdash \pi_i(e@[\sigma_j]_{j\in J\cup J'}) : s_i \wedge s'_i$, that is $\Delta \cup \Delta' \wp \Gamma \wedge \Gamma' \vdash \pi_i(e)@[\sigma_j]_{j\in J\cup J'} : s_i \wedge s'_i$.

Otherwise, there exists at least one typing derivation which ends with an instance of (*subsum*), similar to the case of $e = c$, the result follows by **I**-induction.

$\underline{e = e_1 e_2}$**:** $e@[\sigma_j]_{j\in J} = (e_1@[\sigma_j]_{j\in J})(e_2@[\sigma_j]_{j\in J})$ and $e@[\sigma_j]_{j\in J'} = (e_1@[\sigma_j]_{j\in J'})(e_2@[\sigma_j]_{j\in J'})$. Clearly, both typing derivations should end with either (*appl*) or (*subsum*). Assume that both derivations end with an instance of (*appl*):

$$\frac{\dfrac{\cdots}{\Delta \wp \Gamma \vdash e_1@[\sigma_j]_{j\in J} : s_1 \to s_2} \quad \dfrac{\cdots}{\Delta \wp \Gamma \vdash e_2@[\sigma_j]_{j\in J} : s_1}}{\Delta \wp \Gamma \vdash (e_1@[\sigma_j]_{j\in J})\ (e_2@[\sigma_j]_{j\in J}) : s_2}\ (appl)$$

$$\frac{\dfrac{\cdots}{\Delta' \wp \Gamma' \vdash e_1@[\sigma_j]_{j\in J'} : s'_1 \to s'_2} \quad \dfrac{\cdots}{\Delta' \wp \Gamma' \vdash e_2@[\sigma_j]_{j\in J'} : s'_1}}{\Delta' \wp \Gamma' \vdash (e_1@[\sigma_j]_{j\in J'})\ (e_2@[\sigma_j]_{j\in J'}) : s'_2}\ (appl)$$

By **E**-induction, we have $\Delta \cup \Delta' \wp \Gamma \wedge \Gamma' \vdash e_1@[\sigma_j]_{j\in J\cup J'} : (s_1 \to s_2) \wedge (s'_1 \to s'_2)$ and $\Delta \cup \Delta' \wp \Gamma \wedge \Gamma' \vdash e_2@[\sigma_j]_{j\in J\cup J'} : s_1 \wedge s'_1$. Because intersection distributes over arrows, we have $(s_1 \to s_2) \wedge (s'_1 \to s'_2) \leq (s_1 \wedge s'_1) \to (s_2 \wedge s'_2)$. Then by the rule (*subsum*), we get $\Delta \cup \Delta' \wp \Gamma \wedge \Gamma' \vdash e_1@[\sigma_j]_{j\in J\cup J'} : (s_1 \wedge s'_1) \to (s_2 \wedge s'_2)$. Finally by (*appl*), we have $\Delta \cup \Delta' \wp \Gamma \wedge \Gamma' \vdash (e_1@[\sigma_j]_{j\in J\cup J'})(e_2@[\sigma_j]_{j\in J\cup J'}) : s_2 \wedge s'_2$, that is, $\Delta \cup \Delta' \wp \Gamma \wedge \Gamma' \vdash (e_1 e_2)@[\sigma_j]_{j\in J\cup J'} : s_2 \wedge s'_2$.

Otherwise, there exists at least one typing derivation which ends with an instance of (*subsum*), similar to the case of $e = c$, the result follows by **I**-induction.

$\underline{e = \lambda^{\wedge_{i\in I} t_i \to s_i}_{[\sigma_k]_{k\in K}} x.e'}$**:** $e@[\sigma_j]_{j\in J} = \lambda^{\wedge_{i\in I} t_i \to s_i}_{[\sigma_j]_{j\in J}\circ[\sigma_k]_{k\in K}} x.e'$ and $e@[\sigma_j]_{j\in J'} = \lambda^{\wedge_{i\in I} t_i \to s_i}_{[\sigma_j]_{j\in J'}\circ[\sigma_k]_{k\in K}} x.e'$. Clearly, both typing derivations should end with either (*abstr*) or (*subsum*). Assume that both derivations end with an instance of (*abstr*):

$$\frac{\begin{array}{l}\forall i \in I, j \in J, k \in K.\ \dfrac{\cdots}{\Delta_1 \wp \Gamma, (x : t_i(\sigma_j \circ \sigma_k)) \vdash e'@[\sigma_j \circ \sigma_k] : s_i(\sigma_j \circ \sigma_k)} \\ \Delta_1 = \Delta \cup \mathsf{var}(\bigwedge_{i\in I, j\in J, k\in K} t_i(\sigma_j \circ \sigma_k) \to s_i(\sigma_j \circ \sigma_k))\end{array}}{\Delta \wp \Gamma \vdash \lambda^{\wedge_{i\in I} t_i \to s_i}_{[\sigma_j]_{j\in J}\circ[\sigma_k]_{k\in K}} x.e' : \bigwedge_{i\in I, j\in J, k\in K} t_i(\sigma_j \circ \sigma_k) \to s_i(\sigma_j \circ \sigma_k)}$$

$$\frac{\begin{array}{l}\forall i \in I, j \in J', k \in K.\ \dfrac{\cdots}{\Delta_2 \wp \Gamma', (x : t_i(\sigma_j \circ \sigma_k)) \vdash e'@[\sigma_j \circ \sigma_k] : s_i(\sigma_j \circ \sigma_k)} \\ \Delta_2 = \Delta' \cup \mathsf{var}(\bigwedge_{i\in I, j\in J', k\in K} t_i(\sigma_j \circ \sigma_k) \to s_i(\sigma_j \circ \sigma_k))\end{array}}{\Delta' \wp \Gamma' \vdash \lambda^{\wedge_{i\in I} t_i \to s_i}_{[\sigma_j]_{j\in J'}\circ[\sigma_k]_{k\in K}} x.e' : \bigwedge_{i\in I, j\in J', k\in K} t_i(\sigma_j \circ \sigma_k) \to s_i(\sigma_j \circ \sigma_k)}$$

Consider any expression $e'@([\sigma_j] \circ [\sigma_k])$ and any $e_0[\sigma_{j_0}]_{j_0 \in J_0}$ in $e'@([\sigma_j] \circ [\sigma_k])$, where $j \in J \cup J', k \in K$. (Then $e_0[\sigma_{j_0}]_{j_0 \in J_0}$ must be from e'). All type variables in $\bigcup_{j_0 \in J_0} \mathsf{dom}(\sigma_{j_0})$ must be polymorphic, otherwise, $e'@([\sigma_j] \circ [\sigma_k])$ is not well-typed under Δ_1 or Δ_2. Using α-conversion, we can assume that these polymorphic type variables are different from $\Delta_1 \cup \Delta_2$, that is $(\bigcup_{j_0 \in J_0} \mathsf{dom}(\sigma_{j_0})) \cap (\Delta_1 \cup \Delta_2) = \emptyset$. So we have $e'@([\sigma_j] \circ [\sigma_k]) \sharp \Delta_1 \cup \Delta_2$. According to Lemma 7.4.8, we have

$$\Delta_1 \cup \Delta_2 \wr \Gamma \wedge \Gamma', (x : t_i(\sigma_j \circ \sigma_k)) \vdash e'@[\sigma_j \circ \sigma_k] : s_i(\sigma_j \circ \sigma_k)$$

for all $i \in I$, $j \in J \cup J'$ and $k \in K$. It is clear that

$$\Delta_1 \cup \Delta_2 = \Delta \cup \Delta' \cup \mathsf{var}(\bigwedge_{i \in I, j \in J \cup J', k \in K} t_i(\sigma_j \circ \sigma_k) \to s_i(\sigma_j \circ \sigma_k))$$

By (*abstr*), we have

$$\Delta \cup \Delta' \wr \Gamma \wedge \Gamma' \vdash \lambda^{\wedge_{i \in I} t_i \to s_i}_{[\sigma_j]_{j \in J \cup J'} \circ [\sigma_k]_{k \in K}} x.e' : \bigwedge_{i \in I, j \in J \cup J', k \in K} t_i(\sigma_j \circ \sigma_k) \to s_i(\sigma_j \circ \sigma_k)$$

that is, $\Delta \cup \Delta' \wr \Gamma \wedge \Gamma' \vdash e@[\sigma_j]_{j \in J \cup J'} : t \wedge t'$, where $t = \bigwedge_{i \in I, j \in J, k \in K} t_i(\sigma_j \circ \sigma_k) \to s_i(\sigma_j \circ \sigma_k)$ and $t' = \bigwedge_{i \in I, j \in J', k \in K} t_i(\sigma_j \circ \sigma_k) \to s_i(\sigma_j \circ \sigma_k)$.
Otherwise, there exists at least one typing derivation which ends with an instance of (*subsum*), similar to the case of $e = c$, the result follows by **I**-induction.

$\underline{e = (e_0 \in t \,?\, e_1 : e_2)}$**:** $e@[\sigma_j]_{j \in J} = (e_0@[\sigma_j]_{j \in J} \in t \,?\, e_1@[\sigma_j]_{j \in J} : e_2@[\sigma_j]_{j \in J})$ and $e@[\sigma_j]_{j \in J'} = (e_0@[\sigma_j]_{j \in J'} \in t \,?\, e_1@[\sigma_j]_{j \in J'} : e_2@[\sigma_j]_{j \in J'})$. Clearly, both typing derivations should end with either (*case*) or (*subsum*). Assume that both derivations end with an instance of (*case*):

$$\frac{\dfrac{\dots}{\Delta \wr \Gamma \vdash e_0@[\sigma_j]_{j \in J} : t_0} \quad \begin{cases} t_0 \not\leq \neg t & \Rightarrow \quad \dfrac{\dots}{\Delta \wr \Gamma \vdash e_1@[\sigma_j]_{j \in J} : s} \\ t_0 \not\leq t & \Rightarrow \quad \dfrac{\dots}{\Delta \wr \Gamma \vdash e_2@[\sigma_j]_{j \in J} : s} \end{cases}}{\Delta \wr \Gamma \vdash (e_0@[\sigma_j]_{j \in J} \in t \,?\, e_1@[\sigma_j]_{j \in J} : e_2@[\sigma_j]_{j \in J}) : s} \; (case)$$

$$\frac{\dfrac{\dots}{\Delta' \wr \Gamma' \vdash e_0@[\sigma_j]_{j \in J'} : t'_0} \quad \begin{cases} t'_0 \not\leq \neg t & \Rightarrow \quad \dfrac{\dots}{\Delta' \wr \Gamma' \vdash e_1@[\sigma_j]_{j \in J'} : s'} \\ t'_0 \not\leq t & \Rightarrow \quad \dfrac{\dots}{\Delta' \wr \Gamma' \vdash e_2@[\sigma_j]_{j \in J'} : s'} \end{cases}}{\Delta' \wr \Gamma' \vdash (e_0@[\sigma_j]_{j \in J'} \in t \,?\, e_1@[\sigma_j]_{j \in J'} : e_2@[\sigma_j]_{j \in J'}) : s'} \; (case)$$

By **E**-induction, we have $\Delta \cup \Delta' \wr \Gamma \wedge \Gamma' \vdash e_0@[\sigma_j]_{j \in J \cup J'} : t_0 \wedge t'_0$. Suppose $t_0 \wedge t'_0 \not\leq \neg t$, then we must have $t_0 \not\leq \neg t$ and $t'_0 \not\leq \neg t$, and the first branch has been checked in both derivations. Therefore we have $\Delta \cup \Delta' \wr \Gamma \wedge \Gamma' \vdash e_1@[\sigma_j]_{j \in J \cup J'} : s \wedge s'$ by the induction hypothesis. Similarly, if $t_0 \wedge t'_0 \not\leq t$, we have $\Delta \cup \Delta' \wr \Gamma \wedge \Gamma' \vdash e_2@[\sigma_j]_{j \in J \cup J'} : s \wedge s'$. By applying the rule (*case*), we have

$$\Delta \cup \Delta' \wr \Gamma \wedge \Gamma' \vdash (e_0@[\sigma_j]_{j \in J \cup J'} \in t \,?\, e_1@[\sigma_j]_{j \in J \cup J'} : e_2@[\sigma_j]_{j \in J \cup J'}) : s \wedge s'$$

that is, $\Delta \cup \Delta' \wr \Gamma \wedge \Gamma' \vdash (e_0 \in t \,?\, e_1 : e_2)@[\sigma_j]_{j \in J \cup J'} : s \wedge s'$.
Otherwise, there exists at least one typing derivation which ends with an instance of (*subsum*), similar to the case of $e = c$, the result follows by **I**-induction.

$\underline{e = e'[\sigma_i]_{i\in I}}$**:** $e@[\sigma_j]_{j\in J} = e'@([\sigma_j\circ\sigma_i]_{(j,i)\in(J\times I)})$ and $e@[\sigma_j]_{j\in J'} = e'@([\sigma_j\circ\sigma_i]_{(j,i)\in(J'\times I)})$.
By **E**-induction on e', we have

$$\Delta \cup \Delta' \fatsemi \Gamma \wedge \Gamma' \vdash e'@[\sigma_j \circ \sigma_i]_{(j,i)\in(J\times I)\cup(J'\times I)} : t \wedge t'$$

that is, $\Delta \cup \Delta' \fatsemi \Gamma \wedge \Gamma' \vdash (e'[\sigma_i]_{i\in I})@[\sigma_j]_{j\in J\cup J'} : t \wedge t'$.

□

Corollary 7.4.15. *If* $\Delta \fatsemi \Gamma \vdash e@[\sigma_j]_{j\in J_1} : t_1$ *and* $\Delta \fatsemi \Gamma \vdash e@[\sigma_j]_{j\in J_2} : t_2$*, then* $\Delta \fatsemi \Gamma \vdash e@[\sigma_j]_{j\in J_1\cup J_2} : t_1 \wedge t_2$

Proof. Immediate consequence of Lemmas 7.4.14 and 7.4.8. □

7.4.2 Type soundness

In this section, we prove the soundness of the type system: well-typed expressions do not "go wrong". We proceed in two steps, commonly known as the *subject reduction* and *progress* theorems:

- Subject reduction: a well-typed expression keeps being well-typed during reduction.
- Progress: a well-typed expression can not be "stuck" (*i.e.*, a well-typed expression which is not value can be reduced).

Theorem 7.4.16 (Subject reduction). *Let* e *be an expression and* t *a type. If* $\Delta \fatsemi \Gamma \vdash e : t$ *and* $e \rightsquigarrow e'$*, then* $\Delta \fatsemi \Gamma \vdash e' : t$.

Proof. By induction on the derivation of $\Delta \fatsemi \Gamma \vdash e : t$. We proceed by a case analysis on the last rule used in the derivation of $\Delta \fatsemi \Gamma \vdash e : t$.

$\underline{(const)}$**:** the expression e is a constant. It cannot be reduced. Thus the result follows.
$\underline{(var)}$**:** similar to the (*const*) case.
$\underline{(pair)}$**:** $e = (e_1, e_2)$, $t = t_1 \times t_2$. We have $\Delta \fatsemi \Gamma \vdash e_i : t_i$ for $i = 1..2$. There are two ways to reduce e, that is
(1) $(e_1, e_2) \rightsquigarrow (e_1', e_2)$: by induction, we have $\Delta \fatsemi \Gamma \vdash e_1' : t_1$. Then the rule (*pair*) gives us $\Delta \fatsemi \Gamma \vdash (e_1', e_2) : t_1 \times t_2$.
(2) The case $(e_1, e_2) \rightsquigarrow (e_1, e_2')$ is treated similarly.
$\underline{(proj)}$**:** $e = \pi_i(e_0)$, $t = t_i$, $\Delta \fatsemi \Gamma \vdash e_0 : t_1 \times t_2$.
(1) $e_0 \rightsquigarrow e_0'$: $e' = \pi_i(e_0')$. By induction, we have $\Delta \fatsemi \Gamma \vdash e_0' : t_1 \times t_2$. Then the rule (*proj*) gives us $\Delta \fatsemi \Gamma \vdash e' : t_i$.
(2) $e_0 = (v_1, v_2)$: $e' = v_i$. By Lemma 7.4.3, we get $\Delta \fatsemi \Gamma \vdash e' : t_i$.
$\underline{(appl)}$**:** $e = e_1\ e_2$, $\Delta \fatsemi \Gamma \vdash e_1 : t \to s$ and $\Delta \fatsemi \Gamma \vdash e_2 : t$.
(1) $e_1\ e_2 \rightsquigarrow e_1'\ e_2$ or $e_1\ e_2 \rightsquigarrow e_1\ e_2'$: similar to the case of (*pair*).
(2) $e_1 = \lambda^{\wedge_{i\in I} t_i\to s_i}_{[\sigma_j]_{j\in J}} x.e_0$, $e_2 = v_2$, $e' = (e_0@[\sigma_j]_{j\in P})\{v_2/x\}$ and $P = \{j \in J \mid \exists i \in I.\ \Delta \fatsemi \Gamma \vdash v_2 : t_i\sigma_j\}$: by Lemma 7.4.3, we have $\bigwedge_{i\in I, j\in J} t_i\sigma_j \to s_i\sigma_j \leq t \to s$. From the subtyping for arrow types we deduce that $t \leq \bigvee_{i\in I, j\in J} t_i\sigma_j$ and that for any non-empty set $P \subseteq I \times J$ if $t \not\leq \bigvee_{(i,j)\in I\times J\setminus P} t_i\sigma_j$, then $\bigwedge_{(i,j)\in P} s_i\sigma_j \leq s$.

Let $P_0 = \{(i,j) \mid \Delta \fatsemi \Gamma \vdash v_2 : t_i\sigma_j\}$. Since $\Delta \fatsemi \Gamma \vdash v_2 : t$ and $t \leq \bigvee_{i\in I, j\in J} t_i\sigma_j$, P_0 is non-empty. Also notice that $t \not\leq \bigvee_{(i,j)\in I\times J\setminus P_0} t_i\sigma_j$, since otherwise there would exists some $(i,j) \notin P_0$ such that $\Delta \fatsemi \Gamma \vdash v_2 : t_i\sigma_j$. As a consequence, we get $\bigwedge_{(i,j)\in P_0} s_i\sigma_j \leq s$. Moreover, since e_1 is well-typed under Δ and Γ, there exists an instance of the rule (*abstr*) which infers a type $\bigwedge_{i\in I, j\in J} t_i\sigma_j \to s_i\sigma_j$ for e_1 under Δ and Γ and whose premise is $\Delta' \fatsemi \Gamma, (x : t_i\sigma_j) \vdash e_0@[\sigma_j] : s_i\sigma_j$ for all $i \in I$ and $j \in J$, where $\Delta' = \Delta \cup \mathsf{var}(\bigwedge_{i\in I, j\in J} t_i\sigma_j \to s_i\sigma_j)$. By Lemma 7.4.14, we get $\Delta' \fatsemi \bigwedge_{(i,j)\in P_0}(\Gamma, (x : t_i\sigma_j)) \vdash e_0@[\sigma_j]_{j\in P_0} : \bigwedge_{(i,j)\in P_0} s_i\sigma_j$. Since $\Gamma, (x : \bigwedge_{(i,j)\in P_0} t_i\sigma_j) \simeq \bigwedge_{(i,j)\in P_0}(\Gamma, (x : t_i\sigma_j))$, then from Lemma 7.4.8 we have $\Delta' \fatsemi \Gamma, (x : \bigwedge_{(i,j)\in P_0} t_i\sigma_j) \vdash e_0@[\sigma_j]_{j\in P_0} : \bigwedge_{(i,j)\in P_0} s_i\sigma_j$ and *a fortiori* $\Delta \fatsemi \Gamma, (x : \bigwedge_{(i,j)\in P_0} t_i\sigma_j) \vdash e_0@[\sigma_j]_{j\in P_0} : \bigwedge_{(i,j)\in P_0} s_i\sigma_j$. Furthermore, by definition of P_0 and the admissibility of the intersection introduction (Lemma 7.4.2) we have that $\Delta \fatsemi \Gamma \vdash v_2 : \bigwedge_{(i,j)\in P_0} t_i\sigma_j$. Thus by Lemma 7.4.6, we get $\Delta \fatsemi \Gamma \vdash e' : \bigwedge_{(i,j)\in P_0} s_i\sigma_j$. Finally, by (*subsum*), we obtain $\Delta \fatsemi \Gamma \vdash e' : s$ as expected.

(*abstr*)**:** It cannot be reduced. Thus the result follows.

(*case*)**:** $e = e_0 \in s \,?\, e_1 : e_2$.

(1) $e_0 \rightsquigarrow e_0'$ or $e_1 \rightsquigarrow e_1'$ or $e_2 \rightsquigarrow e_2'$: similar to the case of (*pair*).

(2) $e_0 = v_0$ and $\vdash e_0 : s$: we have $e' = e_1$. The typing rule gives us $\Delta \fatsemi \Gamma \vdash e_1 : t$, thus the result follows.

(3) otherwise ($e_0 = v_0$): we have $e' = e_2$. Similar to the above case.

(*inst*)**:** $e = e_1[\sigma]$, $\Delta \fatsemi \Gamma \vdash e_1 : s$, $\sigma \sharp \Delta$ and $e \rightsquigarrow e_1@[\sigma]$. By applying Lemma 7.4.12, we get $\Delta \fatsemi \Gamma \vdash e_1@[\sigma] : s\sigma$.

(*inter*)**:** $e = e_1[\sigma_j]_{j\in J}$, $\Delta \fatsemi \Gamma \vdash e_1[\sigma_j]_{j\in J} : \bigwedge_{j\in J} t_j$ and $e \rightsquigarrow e_1@[\sigma_j]_{j\in J}$. By applying Corollary 7.4.13, we get $\Delta \fatsemi \Gamma \vdash e_1@[\sigma_j]_{j\in J} : \bigwedge_{j\in J} t_j$.

(*subsum*)**:** there exists a type s such that $\Delta \fatsemi \Gamma \vdash e : s \leq t$ and $e \rightsquigarrow e'$. By induction, we have $\Delta \fatsemi \Gamma \vdash e' : s$, then by subsumption we get $\Delta \fatsemi \Gamma \vdash e' : t$.

□

Theorem 7.4.17 (Progress). *Let e be a well-typed closed expression, that is, $\vdash e : t$ for some t. If e is not a value, then there exists an expression e' such that $e \rightsquigarrow e'$.*

Proof. By induction on the derivation of $\vdash e : t$. We proceed by a case analysis of the last rule used in the derivation of $\vdash e : t$.

(*const*)**:** immediate since a constant is a value.

(*var*)**:** impossible since a variable cannot be well-typed in an empty environment.

(*pair*)**:** $e = (e_1, e_2)$, $t = t_1 \times t_2$, and $\vdash e_i : t_i$ for $i = 1..2$. If one of the e_i can be reduced, then e can also be reduced. Otherwise, by induction, both e_1 and e_2 are values, and so is e.

(*proj*)**:** $e = \pi_i(e_0)$, $t = t_i$, and $\vdash e_0 : t_1 \times t_2$. If e_0 can be reduced to e_0', then $e \rightsquigarrow \pi_i(e_0')$. Otherwise, e_0 is a value. By Lemma 7.4.3, we get $e_0 = (v_1, v_2)$, and thus $e \rightsquigarrow v_i$.

(*appl*)**:** $e = e_1\ e_2$, $\vdash e_1 : t \to s$ and $\vdash e_2 : t$. If one of the e_i can be reduced, then e can also be reduced. Otherwise, by induction, both e_1 and e_2 are values. By Lemma 7.4.3, we get $e_1 = \lambda^{\wedge_{i\in I} t_i \to s_i}_{[\sigma_j]_{j\in J}} x.e_0$ such that $\bigwedge_{i\in I, j\in J} t_i\sigma_j \to s_i\sigma_j \leq t \to s$. By the definition of subtyping for arrow types, we have $t \leq \bigvee_{i\in I, j\in J} t_i\sigma_j$.

Moreover, as $\vdash e_2 : t$, the set $P = \{j \in J \mid \exists i \in I.\ \vdash e_2 : t_i\sigma_j\}$ is non-empty. Then $e \rightsquigarrow (e_0@[\sigma_j]_{j\in P})\{e_2/x\}$.

(abstr): the expression e is an abstraction which is well-typed under the empty environment. It is thus a value.

(case): $e = e_0 \in s\,?\,e_1 : e_2$. If e_0 can be reduced, then e can also be reduced. Otherwise, by induction, e_0 is a value v. If $\vdash v : s$, then we have $e \rightsquigarrow e_1$. Otherwise, $e \rightsquigarrow e_2$.

(inst): $e = e_1[\sigma]$, $t = s\sigma$ and $\vdash e_1 : s$. Then $e \rightsquigarrow e_1@[\sigma]$.

(inter): $e = e_1[\sigma_j]_{j\in J}$, $t = \bigwedge_{j\in J} t_j$ and $\vdash e_1[\sigma_j] : t_j$ for all $j \in J$. It is clear that $e \rightsquigarrow e_1@[\sigma_j]_{j\in J}$.

(subsum): straightforward application of the induction hypothesis.

□

We now conclude that the type system is type sound.

Corollary 7.4.18 (Type soundness). *Let e be a well-typed closed expression, that is, $\vdash e : t$ for some t. Then either e diverges or it returns a value of type t.*

Proof. Consequence of Theorems 7.4.17 and 7.4.16. □

7.4.3 Expressing intersection types [2]

We now prove that the calculus with explicit substitutions is able to derive the same typings as the Barendregt, Coppo, Dezani (BCD) intersection type system [BCD83] without the universal type ω. We remind the BCD types (a strict subset of $\mathcal{T}$), the BCD typing rules (without ω) and subtyping relation in Figure 7.3, where we use m to range over pure λ-calculus expressions. To make the correspondence between the systems easier, we adopt a n-ary version of the intersection typing rule. Henceforth, we use D to range over BCD typing derivations. We first remark that the BCD subtyping relation is included in the one of this work.

Lemma 7.4.19. *If $t_1 \leq_{BCD} t_2$ then $t_1 \leq t_2$.*

Proof. All the BCD subtyping rules are admissible inour type system. □

In this subsection, we restrict the grammar of expressions with explicit substitutions to

$$e \ ::= \ x \ \mid \ e\ e \ \mid \ \lambda^{\wedge_{i\in I} s_i \to t_i}_{[\sigma_j]_{j\in J}} x.e \tag{7.2}$$

and we write $\lceil e \rceil$ for the pure λ-calculus expression obtained by removing all types references (i.e., interfaces and decorations) from e. Given a pure λ-calculus expression m and a derivation of a judgement $\Gamma \vdash_{BCD} m : t$, we build an expression e such that $\lceil e \rceil = m$ and $\Delta ; \Gamma \vdash e : t$ for some set Δ of type variables. With the restricted grammar of (7.2), the intersection typing rule is used only in conjunction with the abstraction typing rule. We prove that it is possible to put a similar restriction on derivations of BCD typing judgements.

[2] The proofs in this section are mainly done by Serguei Lenglet.

Types:

$$t \ ::= \ \alpha \mid t \to t \mid t \wedge t$$

Typing rules:

$$\frac{}{\Gamma \vdash_{BCD} x : \Gamma(x)} \ (BCD\ var) \qquad \frac{\Gamma \vdash_{BCD} m_1 : t_1 \to t_2 \qquad \Gamma \vdash m_2 : t_1}{\Gamma \vdash_{BCD} m_1\ m_2 : t_2} \ (BCD\ app)$$

$$\frac{\Gamma, x : t_1 \vdash_{BCD} m : t_2}{\Gamma \vdash_{BCD} \lambda x.m : t_1 \to t_2} \ (BCD\ abstr) \qquad \frac{\Gamma \vdash_{BCD} m : t_i \quad i \in I}{\Gamma \vdash_{BCD} m : \bigwedge_{i \in I} t_i} \ {\scriptstyle |I| > 1} \ (BCD\ inter)$$

$$\frac{\Gamma \vdash_{BCD} m : t_1 \qquad t_1 \leq_{BCD} t_2}{\Gamma \vdash m : t_2} \ (BCD\ sub)$$

Subtyping relation:

$$\frac{}{t \leq_{BCD} t} \qquad \frac{}{t \leq_{BCD} t \wedge t} \qquad \frac{}{t_1 \wedge t_2 \leq_{BCD} t_1}$$

$$\frac{}{(t_1 \to t_2) \wedge (t_1 \to t_3) \leq_{BCD} t_1 \to (t_2 \wedge t_3)} \qquad \frac{t_1 \leq_{BCD} t_2 \qquad t_2 \leq_{BCD} t_3}{t_1 \leq_{BCD} t_3}$$

$$\frac{t_1 \leq_{BCD} t_3 \qquad t_2 \leq_{BCD} t_4}{t_1 \wedge t_2 \leq_{BCD} t_3 \wedge t_4} \qquad \frac{t_3 \leq_{BCD} t_1 \qquad t_2 \leq_{BCD} t_4}{t_1 \to t_2 \leq_{BCD} t_3 \to t_4}$$

Figure 7.3: The BCD type system

Definition 7.4.20. *Let D be a BCD typing derivation. We say D is in* intersection-abstraction normal form *if (BCD inter) is used only immediately after (BCD abstr) in D, that is to say, all uses of (BCD inter) in D are of the form*

$$\frac{\dfrac{\dfrac{D_i}{\Gamma, x : t_i \vdash_{BCD} m : s_i}}{\Gamma \vdash_{BCD} \lambda x.m : t_i \to s_i} i \in I}{\Gamma \vdash_{BCD} \lambda x.m : \bigwedge_{i \in I} t_i \to s_i}$$

Definition 7.4.21. *Let D be a (BCD) typing derivation. We define the* size *of D, denoted by $|D|$, as the number of rules used in D.*

We prove that any BCD typing judgement can be proved with a derivation in intersection-abstraction normal form.

Lemma 7.4.22. *If $\Gamma \vdash_{BCD} m : t$, then there exists a derivation in intersection-abstraction normal form proving this judgement.*

Proof. Let D be the derivation proving $\Gamma \vdash_{BCD} m : t$. We proceed by induction on the size of D. If $|D| = 1$ then the rule (*BCD var*) has been used, and D is in intersection-abstraction normal form. Otherwise, assume that $|D| > 1$. We proceed by case analysis on the last rule used in D.

(*BCD sub*)**:** D ends with (*BCD sub*):

$$D = \frac{D' \qquad t' \leq t}{\Gamma \vdash_{BCD} m : t}$$

where D' proves a judgement $\Gamma \vdash_{BCD} m : t'$. By the induction hypothesis, there exists a derivation D'' in intersection-abstraction normal form which proves the same judgement as D'. Then

$$\frac{D'' \qquad t' \leq t}{\Gamma \vdash_{BCD} m : t}$$

is in intersection-abstraction normal form and proves the same judgement as D.

(*BCD abstr*)**:** similar to the case of (*BCD sub*).

(*BCD app*)**:** similar to the case of (*BCD sub*).

(*BCD inter*)**:** D ends with (*BCD inter*):

$$D = \frac{D_i}{\Gamma \vdash_{BCD} m : t} i \in I$$

where each D_i proves a judgement $\Gamma \vdash_{BCD} m : t_i$ and $t = \bigwedge_{i \in I} t_i$. We distinguish several cases.

If one of the derivations ends with (*BCD sub*)**,** there exists $i_0 \in I$ such that

$$D_{i_0} = \frac{D'_{i_0} \qquad t'_{i_0} \leq t_{i_0}}{\Gamma \vdash_{BCD} m : t_{i_0}}$$

The derivation

$$D' = \frac{D_i \qquad D'_{i_0}}{\Gamma \vdash_{BCD} m : \bigwedge\limits_{i \in I \setminus \{i_0\}} t_i \wedge t'_{i_0}} i \in I \setminus \{i_0\}$$

is smaller than D, so by the induction hypothesis, there exists D'' in intersection-abstraction normal form which proves the same judgement as D'. Then the derivation

$$\frac{D'' \qquad \bigwedge\limits_{i \in I \setminus \{i_0\}} t_i \wedge t'_{i_0} \leq_{BCD} \bigwedge\limits_{i \in I} t_i}{\Gamma \vdash_{BCD} m : t}$$

is in intersection-abstraction normal form, and proves the same judgement as D.

If one of the derivations ends with (*BCD inter*)**,** there exists $i_0 \in I$ such that

$$D_{i_0} = \frac{D_{j,i_0}}{\Gamma \vdash_{BCD} m : \bigwedge\limits_{j \in J} t_{j,i_0}} j \in J$$

with $t_{i_0} = \bigwedge_{j \in J} t_{j,i_0}$. The derivation

$$D' = \frac{D_i \qquad D_{j,i_0} \quad \begin{array}{c} i \in I \setminus \{i_0\} \\ j \in J \end{array}}{\Gamma \vdash_{BCD} m : t}$$

is smaller than D, so by the induction hypothesis, there exists D'' in intersection-abstraction normal form which proves the same judgement as D', which is the same as the judgement of D.

If all the derivations are uses of (*BCD var*)**,** then for all $i \in I$, we have

$$D_i = \frac{}{\Gamma \vdash_{BCD} x : \Gamma(x)}$$

which implies $t = \bigwedge_{i \in I} \Gamma(x)$ and $m = x$. Then the derivation

$$\frac{\dfrac{}{\Gamma \vdash_{BCD} x : \Gamma(x)} \qquad \Gamma(x) \leq_{BCD} t}{\Gamma \vdash_{BCD} x : t}$$

is in intersection-abstraction normal form and proves the same judgement as D.

If all the derivations end with (*BCD app*)**,** then for all $i \in I$, we have

$$D_i = \frac{D_i^1 \qquad D_i^2}{\Gamma \vdash_{BCD} m_1\ m_2 : t_i}$$

where $m = m_1\ m_2$, D_i^1 proves $\Gamma \vdash_{BCD} m_1 : s_i \to t_i$, and D_i^2 proves $\Gamma \vdash_{BCD} m_2 : s_i$ for some s_i. Let

$$D_1 = \frac{D_i^1}{\Gamma \vdash_{BCD} m_1 : \bigwedge\limits_{i \in I} s_i \to t_i} i \in I \qquad D_2 = \frac{D_i^2}{\Gamma \vdash_{BCD} m_2 : \bigwedge\limits_{i \in I} s_i} i \in I$$

Both D_1 and D_2 are smaller than D, so by the induction hypothesis, there exist D_1', D_2' in intersection-abstraction normal form which prove the same judgements as D_1 and D_2 respectively. Then the derivation

$$\frac{\dfrac{D_1' \qquad \bigwedge\limits_{i \in I} s_i \to t_i \leq_{BCD} (\bigwedge\limits_{i \in I} s_i) \to (\bigwedge\limits_{i \in I} t_i)}{\Gamma \vdash_{BCD} m_1 : (\bigwedge\limits_{i \in I} s_i) \to (\bigwedge\limits_{i \in I} t_i)} \qquad D_2'}{\Gamma \vdash m_1\ m_2 : t}$$

is in intersection-abstraction normal form and proves the same derivation as D.

If all the derivations end with (*BCD abstr*)**,** then for all $i \in I$, we have

$$D_i = \frac{D'_i}{\Gamma \vdash_{BCD} \lambda x.m' : t_i}\ (BCD\ abstr)$$

with $m = \lambda x.m'$. For all $i \in I$, D'_i is smaller than D, so by induction there exists D''_i in intersection-abstraction normal form which proves the same judgement as D'_i. Then the derivation

$$\frac{\dfrac{D''_i}{\Gamma \vdash_{BCD} \lambda x.m' : t_i}\, i \in I}{\Gamma \vdash_{BCD} \lambda x.m' : \bigwedge_{i \in I} t_i}$$

is in intersection-abstraction normal form, and proves the same judgement as D.

□

We now sketch the principles behind the construction of e from D in intersection-abstraction normal form. If D proves a judgement $\Gamma \vdash_{BCD} \lambda x.m : t \to s$, without any top-level intersection, then we simply put $t \to s$ in the interface of the corresponding expression $\lambda^{t \to s} x.e$.

For a judgement $\Gamma \vdash_{BCD} \lambda x.m : \bigwedge_{i \in I} t_i \to s_i$, we build an expression $\lambda^{\alpha \to \beta}_{[\sigma_i]_{i \in I}} x.e$ where each σ_i corresponds to the derivation which types $\lambda x.m$ with $t_i \to s_i$. For example, let $m = \lambda f.\lambda x.f\ x$, and consider the judgement $\vdash_{BCD} m : ((t_1 \to t_2) \to t_1 \to t_2) \wedge ((s_1 \to s_2) \to s_1 \to s_2)$. We first annotate each abstraction in m with types $\alpha_j \to \beta_j$, where α_j, β_j are fresh, distinct variables, giving us $e = \lambda^{\alpha_1 \to \beta_1} f.\lambda^{\alpha_2 \to \beta_2} x.f\ x$. Comparing $\lambda^{\alpha_2 \to \beta_2} x.f\ x$ to the judgement $f : t_1 \to t_2 \vdash_{BCD} \lambda x.f\ x : t_1 \to t_2$ and e to $\vdash_{BCD} m : (t_1 \to t_2) \to t_1 \to t_2$, we compute $\sigma_1 = \{t_1 \to t_2/\alpha_1, t_1 \to t_2/\beta_1, t_1/\alpha_2, t_2/\beta_2\}$. We compute similarly σ_2 from the derivation of $\vdash_{BCD} m : (s_1 \to s_2) \to s_1 \to s_2$, and we obtain finally

$$\vdash \lambda^{\alpha_1 \to \beta_1}_{[\sigma_1, \sigma_2]} f.\lambda^{\alpha_2 \to \beta_2} x.f\ x : ((t_1 \to t_2) \to t_1 \to t_2) \wedge ((s_1 \to s_2) \to s_1 \to s_2)$$

as wished.

The problem becomes more complex when we have nested uses of the intersection typing rule. For example, let $m = \lambda f.\lambda g.g\ (\lambda x.f\ (\lambda y.x\ y))$ and consider the judgement $\vdash_{BCD} m : (t_f \to t_g \to t_4) \wedge (s_f \to s_g \to s_7)$ with

$$\begin{aligned} t_f &= (t_1 \to t_2) \to t_3 \\ t_g &= t_f \to t_4 \\ s_f &= ((s_1 \to s_2) \to s_3) \wedge ((s_4 \to s_5) \to s_6) \\ s_g &= s_f \to s_7 \end{aligned}$$

Notice that, to derive $\vdash_{BCD} m : s_f \to s_g \to s_7$, we have to prove $f : s_f, g : s_g \vdash_{BCD} \lambda x.f\ (\lambda y.x\ y) : s_f$, which requires the (*BCD inter*) rule. As before, we annotate m

with fresh interfaces, obtaining $\lambda^{\alpha_1\to\beta_1}f.\lambda^{\alpha_2\to\beta_2}g.g\ (\lambda^{\alpha_3\to\beta_3}x.f\ (\lambda^{\alpha_4\to\beta_4}y.x\ y))$. Because the intersection typing rule is used twice (once to type m, and once to type $m' = \lambda x.f\ (\lambda y.x\ y)$), we want to compute four substitutions $\sigma_1, \sigma_2, \sigma_3, \sigma_4$ to obtain a decorated expression

$$\lambda^{\alpha_1\to\beta_1}_{[\sigma_1,\sigma_2]}f.\lambda^{\alpha_2\to\beta_2}g.g\ (\lambda^{\alpha_3\to\beta_3}_{[\sigma_3,\sigma_4]}x.f\ (\lambda^{\alpha_4\to\beta_4}y.x\ y)).$$

The difficult part is in computing σ_3 and σ_4: in one case (corresponding to the branch $\vdash_{BCD} m : t_f \to t_g \to t_4$), we want $\sigma_3 = \sigma_4 = \{t_1 \to t_2/\alpha_3, t_3/\beta_3, t_1/\alpha_4, t_2/\beta_4\}$ to obtain $f : t_f, g : t_g \vdash \lambda^{\alpha_3\to\beta_3}_{[\sigma_3,\sigma_4]}x.f\ (\lambda^{\alpha_4\to\beta_4}y.x\ y) : t_f$, and in the other case (corresponding to the derivation $\vdash_{BCD} m : s_f \to s_g \to s_7$), we want $\sigma_3 = \{s_1 \to s_2/\alpha_3, s_3/\beta_3, s_1/\alpha_4, s_2/\beta_4\}$ and $\sigma_4 = \{s_4 \to s_5/\alpha_3, s_6/\beta_3, s_4/\alpha_4, s_5/\beta_4\}$ to obtain $f : s_f, g : s_g \vdash \lambda^{\alpha_3\to\beta_3}_{[\sigma_3,\sigma_4]}x.f\ (\lambda^{\alpha_4\to\beta_4}y.x\ y) : s_f$. To resolve this issue, we use intermediate fresh variables $\alpha'_3, \beta'_3, \alpha'_4, \beta'_4$ and $\alpha''_3, \beta''_3, \alpha''_4, \beta''_4$ in the definition of σ_3 and σ_4. We define

$$\begin{aligned}
\sigma_1 =&\{t_f/\alpha_1, t_g \to t_4/\beta_1, t_g/\alpha_2, t_4/\beta_2, t_1 \to t_2/\alpha'_3, t_3/\beta'_3, t_1/\alpha'_4, t_2/\beta'_4,\\
&\ t_1 \to t_2/\alpha''_3, t_3/\beta''_3, t_1/\alpha''_4, t_2/\beta''_4\}\\
\sigma_2 =&\{s_f/\alpha_1, s_g \to s_7/\beta_1, s_g/\alpha_2, s_7/\beta_2, s_1 \to s_2/\alpha'_3, s_3/\beta'_3, s_1/\alpha'_4, s_2/\beta'_4,\\
&\ s_4 \to s_5/\alpha''_3, s_6/\beta''_3, s_4/\alpha''_4, s_5/\beta''_4\}\\
\sigma_3 =&\{\alpha'_3/\alpha_3, \beta'_3/\beta_3, \alpha'_4/\alpha_4, \beta'_4/\beta_4\}\\
\sigma_4 =&\{\alpha''_3/\alpha_3, \beta''_3/\beta_3, \alpha''_4/\alpha_4, \beta''_4/\beta_4\}
\end{aligned}$$

Because the substitutions compose themselves, we obtain

$$\vdash \lambda^{\alpha_1\to\beta_1}_{[\sigma_1,\sigma_2]}f.\lambda^{\alpha_2\to\beta_2}g.g\ (\lambda^{\alpha_3\to\beta_3}_{[\sigma_3,\sigma_4]}x.f\ (\lambda^{\alpha_4\to\beta_4}y.x\ y)) : (t_f \to t_g \to t_4) \wedge (s_f \to s_g \to s_7)$$

as wished.

In the next lemma, given n derivations $D_1, \ldots, D_n$ in intersection-abstraction normal form for a same expression m, we construct an expression e containing fresh interfaces and decorations with fresh variables if needed (as explained in the above example) and n substitutions $\sigma_1, \ldots, \sigma_n$ corresponding to $D_1, \ldots, D_n$. Let $\mathsf{var}(D_1, \ldots, D_n)$ denote the set of type variables occurring in the types in $D_1, \ldots, D_n$.

Lemma 7.4.23. *Let m be a pure λ-calculus expression, Δ be a set of type variables, and $D_1, \ldots, D_n$ be derivations in intersection-abstraction normal form such that D_i proves $\Gamma_i \vdash_{BCD} m : t_i$ for all i. Let Δ' be a set of type variables such that $\mathsf{var}(D_1, \ldots, D_n) \subseteq \Delta'$. There exist e, $\sigma_1, \ldots, \sigma_n$ such that $\lceil e \rceil = m$, $\mathsf{dom}(\sigma_1) = \ldots = \mathsf{dom}(\sigma_n) \subseteq \mathsf{tv}(e)$, $\mathsf{tv}(e) \cap (\Delta \cup \Delta') = \emptyset$, and $\Delta' \wr \Gamma_i \vdash e@[\sigma_i] : t_i$ for all i.*

Proof. We proceed by induction on the sum of the size of $D_1, \ldots, D_n$. If this sum is equal to n, then each D_i is a use of the (*BCD var*) rule, and we have $m = x$ for some x. Let $e = x$ and σ_i be the identity; we can then easily check that the result holds. Otherwise, assume this sum is strictly greater than n. We proceed by case analysis on $D_1, \ldots, D_n$.

Case 1: If one of the derivations ends with (*BCD sub*), there exists i_0 such that

$$D_{i_0} = \frac{D'_{i_0} \qquad t'_{i_0} \leq_{BCD} t_{i_0}}{\Gamma_{i_0} \vdash_{BCD} m : t_{i_0}}$$

Clearly, the sum of the size of $D_1, \ldots, D'_{i_0}, \ldots, D_n$ is smaller than that of $D_1, \ldots, D_n$, and $\mathsf{var}(D_1, \ldots, D'_{i_0}, \ldots, D_n) \subseteq \Delta'$. So by the induction hypothesis, we have

$$\begin{array}{lll} \exists e, \sigma_1, \ldots, \sigma_n. & \lceil e \rceil = m \\ & \text{and} & \mathsf{dom}(\sigma_1) = \ldots = \mathsf{dom}(\sigma_n) \subseteq \mathsf{tv}(e) \\ & \text{and} & \mathsf{tv}(e) \cap (\Delta \cup \Delta') = \emptyset \\ & \text{and} & \forall i \in \{1, \ldots, n\} \setminus \{i_0\}.\ \Delta' \fatsemi \Gamma_i \vdash e@[\sigma_i] : t_i \\ & \text{and} & \Delta' \fatsemi \Gamma_{i_0} \vdash e@[\sigma_{i_0}] : t'_{i_0}. \end{array}$$

Since $t'_{i_0} \leq_{BCD} t_{i_0}$, by Lemma 7.4.19, we have $t'_{i_0} \leq t_{i_0}$. Therefore $\Delta' \fatsemi \Gamma_{i_0} \vdash e@[\sigma_{i_0}] : t_{i_0}$ holds, and for all i, we have $\Delta' \fatsemi \Gamma_i \vdash e@[\sigma_i] : t_i$ as wished.

Case 2: If all the derivations end with (*BCD app*), then we have $m = m_1\ m_2$, and for all i

$$D_i = \frac{D_i^1 \qquad D_i^2}{\Gamma_i \vdash_{BCD} m_1\ m_2 : t_i}$$

where D_i^1 proves $\Gamma_i \vdash_{BCD} m_1 : s_i \to t_i$ and D_i^2 proves $\Gamma_i \vdash_{BCD} m_2 : s_i$ for some s_i. Applying the induction hypothesis on $D_1^1, \ldots, D_n^1$ (with Δ and Δ'), we have

$$\begin{array}{lll} \exists e_1, \sigma_1^1, \ldots, \sigma_n^1. & \lceil e \rceil_1 = m_1 \\ & \text{and} & \mathsf{dom}(\sigma_1^1) = \ldots = \mathsf{dom}(\sigma_n^1) \subseteq \mathsf{tv}(e_1) \\ & \text{and} & \mathsf{tv}(e_1) \cap (\Delta \cup \Delta') = \emptyset \\ & \text{and} & \forall i \in \{1, \ldots, n\}.\ \Delta' \fatsemi \Gamma_i \vdash e_1@[\sigma_i^1] : s_i \to t_i. \end{array}$$

Similarly, by induction on $D_1^2, \ldots, D_n^2$ (with $\Delta \cup \mathsf{tv}(e_1)$ and Δ'),

$$\begin{array}{lll} \exists e_2, \sigma_1^2, \ldots, \sigma_n^2. & \lceil e \rceil_2 = m_2 \\ & \text{and} & \mathsf{dom}(\sigma_1^2) = \ldots = \mathsf{dom}(\sigma_n^2) \subseteq \mathsf{tv}(e_2) \\ & \text{and} & \mathsf{tv}(e_2) \cap (\Delta \cup \mathsf{tv}(e_1) \cup \Delta') = \emptyset \\ & \text{and} & \forall i \in \{1, \ldots, n\}.\ \Delta' \fatsemi \Gamma_i \vdash e_2@[\sigma_i^2] : s_i. \end{array}$$

From $\mathsf{tv}(e_2) \cap (\Delta \cup \mathsf{tv}(e_1) \cup \Delta') = \emptyset$, we deduce $\mathsf{tv}(e_1) \cap \mathsf{tv}(e_2) = \emptyset$. Let $i \in \{1, \ldots, n\}$. Because $\mathsf{dom}(\sigma_i^1) \subseteq \mathsf{tv}(e_1)$ and $\mathsf{dom}(\sigma_i^2) \subseteq \mathsf{tv}(e_2)$, we have $\mathsf{dom}(\sigma_i^1) \cap \mathsf{dom}(\sigma_i^2) = \emptyset$, $\mathsf{dom}(\sigma_i^1) \cap \mathsf{tv}(e_2) = \emptyset$, and $\mathsf{dom}(\sigma_i^2) \cap \mathsf{tv}(e_1) = \emptyset$. Consequently, by Lemma 7.4.10, we have $\Delta' \fatsemi \Gamma_i \vdash e_1@[\sigma_i^1 \cup \sigma_i^2] : s_i \to t_i$ and $\Delta' \fatsemi \Gamma_i \vdash e_2@[\sigma_i^1 \cup \sigma_i^2] : s_i$. Therefore, we have $\Delta' \fatsemi \Gamma_i \vdash (e_1\ e_2)@[\sigma_i^1 \cup \sigma_i^2] : t_i$. So we have the required result with $e = e_1\ e_2$ and $\sigma_i = \sigma_i^1 \cup \sigma_i^2$.

Case 3: If all the derivations end with (*BCD abstr*), then $m = \lambda x.m_1$, and for all i,

$$D_i = \frac{D'_i}{\Gamma_i \vdash_{BCD} m : t_i}$$

where D'_i proves $\Gamma_i, x : t_i^1 \vdash_{BCD} m_1 : t_i^2$ and $t_i = t_i^1 \to t_i^2$. By the induction hypothesis,

$$\begin{array}{lll} \exists e_1, \sigma'_1, \ldots, \sigma'_n. & \lceil e \rceil_1 = m_1 \\ & \text{and} & \mathsf{dom}(\sigma'_1) = \ldots = \mathsf{dom}(\sigma'_n) \subseteq \mathsf{tv}(e_1) \\ & \text{and} & \mathsf{tv}(e_1) \cap (\Delta \cup \Delta') = \emptyset \\ & \text{and} & \forall i \in \{1, \ldots, n\}.\ \Delta' \fatsemi \Gamma_i, x : t_i^1 \vdash e_1@[\sigma'_i] : t_i^2. \end{array}$$

Let α, β be two fresh type variables. So $\{\alpha,\beta\}\cap(\Delta\cup\Delta') = \emptyset$ and $\{\alpha,\beta\}\cap\mathsf{tv}(e_1) = \emptyset$. Take $i \in \{1,\ldots,n\}$. Let $\sigma_i = \{t_i^1/\alpha, t_i^2/\beta\} \cup \sigma_i'$, and $e = \lambda^{\alpha\to\beta}x.e_1$. We have $\mathsf{dom}(\sigma_i) = \{\alpha,\beta\} \cup \mathsf{dom}(\sigma_i') \subseteq \{\alpha,\beta\} \cup \mathsf{tv}(e_1) = \mathsf{tv}(e)$. Besides, we have $\mathsf{tv}(e) \cap (\Delta \cup \Delta') = \emptyset$. Because $\mathsf{tv}(e_1) \cap \{\alpha,\beta\} = \emptyset$, we have $\mathsf{dom}(\sigma_i') \cap \{\alpha,\beta\} = \emptyset$, and $\Delta' \wr \Gamma_i, x : t_i^1 \vdash e_1@[\sigma_i] : t_i^2$ by Lemma 7.4.10, which is equivalent to $\Delta' \wr \Gamma_i, x : \alpha\sigma_i \vdash e_1@[\sigma_i] : \beta\sigma_i$. Because $\Delta' \cup \mathsf{var}(t_i^1 \to t_i^2) = \Delta'$, by the abstraction rule, we have $\Delta' \wr \Gamma_i \vdash \lambda^{\alpha\to\beta}_{[\sigma_i]}x.e_1 : t_i$, *i.e.*, $\Delta' \wr \Gamma_i \vdash (\lambda^{\alpha\to\beta}.e_1)@[\sigma_i] : t_i$. Therefore, we have the required result.

Case 4: If one of the derivations ends with (*BCD inter*), then $m = \lambda x.m_1$. The derivations end with either (*BCD inter*) or (*BCD abstr*) (we omit the already treated case of (*BCD sub*)). For simplicity, we suppose they all end with (*BCD inter*), as it is the same if some of them end with (*BCD abstr*) (note that Case 3 is a special case of Case 4). For all i, we have

$$D_i = \dfrac{\dfrac{D_i^j}{\Gamma_i \vdash_{BCD} m : s_i^j \to t_i^j}}{\Gamma_i \vdash_{BCD} m : \bigwedge\limits_{j\in J_i} s_i^j \to t_i^j} j \in J_i$$

where D_i^j proves $\Gamma_i, x : s_i^j \vdash_{BCD} m_1 : t_i^j$ for all $j \in J_i$ and $t_i = \bigwedge_{j\in J_i} s_i^j \to t_i^j$ for all i. By the induction hypothesis on the sequence of D_i^j,

$$\begin{array}{lll}\exists e_1, (\sigma_1^j)_{j\in J_1}, \ldots, (\sigma_n^j)_{j\in J_n}. & \lceil e_1\rceil = m_1 \\ & \text{and} & \forall i,i',j,j'.\ \mathsf{dom}(\sigma_i^j) = \mathsf{dom}(\sigma_{i'}^{j'}) \text{ and } \mathsf{dom}(\sigma_i^j) \subseteq \mathsf{tv}(e_1) \\ & \text{and} & \mathsf{tv}(e_1) \cap (\Delta \cup \Delta') = \emptyset \\ & \text{and} & \forall i,j.\ \Delta' \wr \Gamma_i, x : s_i^j \vdash e_1@[\sigma_i^j] : t_i^j.\end{array}$$

Let $p = max_{i\in\{1,\ldots,n\}}\{|J_i|\}$. For all i, we complete the sequence of substitutions (σ_i^j) so that it contains exactly p elements, by repeating the last one $p-|J_i|$ times, and we number them from 1 to p. All the σ_i^j have the same domain (included in $\mathsf{tv}(e_1)$), that we number from 1 to q. Then $\sigma_i^j = \bigcup_{k\in\{1,\ldots,q\}}\{t_{i,j}^k/\alpha_k\}$ for some types $(t_{i,j}^k)$. Let α, β, $(\alpha_{j,k})_{j\in\{1,\ldots,p\},k\in\{1,\ldots,q\}}$, $(\alpha_{j,0},\beta_{j,0})_{j\in\{1,\ldots,p\}}$ be fresh pairwise distinct variables (which do not occur in $\Delta \cup \Delta' \cup \mathsf{tv}(e_1)$). For all $j \in \{1,\ldots,p\}$, $i \in \{1,\ldots,n\}$, we define:

$$\begin{array}{rl}\sigma_j = & \bigcup\limits_{k\in\{1,\ldots,q\}} \{\alpha_{j,k}/\alpha_k\} \cup \{\alpha_{j,0}/\alpha, \beta_{j,0}/\beta\} \\ e = & \lambda^{\alpha\to\beta}_{[\sigma_j]_{j\in\{1,\ldots,p\}}}x.e_1 \\ \sigma_i = & \bigcup\limits_{j\in\{1,\ldots,p\},k\in\{1,\ldots,q\}} \{t_{i,j}^k/\alpha_{j,k}\} \cup \bigcup\limits_{j\in\{1,\ldots,p\}} \{s_i^j/\alpha_{j,0}, t_i^j/\beta_{j,0}\}\end{array}$$

For all i, j, k, we have by construction $(\alpha_k\sigma_j)\sigma_i = \alpha_k\sigma_i^j$, $(\alpha\sigma_j)\sigma_i = s_i^j$, and

$(\beta\sigma_j)\sigma_i = t_i^j$. Moreover, since

$$\begin{array}{rcl}
\mathsf{tv}(e) = & \mathsf{tv}(e_1@[\sigma_j]_{j\in\{1,\dots,p\}}) \cup \bigcup_{j\in\{1,\dots,p\}} \mathsf{var}((\alpha\to\beta)\sigma_j) \\
= & (\mathsf{tv}(e_1))[\sigma_j]_{j\in\{1,\dots,p\}} \cup \{\alpha_{j,0},\beta_{j,0}\}_{j\in\{1,\dots,p\}} \\
\supseteq & (\mathsf{dom}(\sigma_i^j))[\sigma_j]_{j\in\{1,\dots,p\}} \cup \{\alpha_{j,0},\beta_{j,0}\}_{j\in\{1,\dots,p\}} \\
= & (\{\alpha_k\}_{k\in\{1,\dots,q\}})[\sigma_j]_{j\in\{1,\dots,p\}} \cup \{\alpha_{j,0},\beta_{j,0}\}_{j\in\{1,\dots,p\}} \\
= & \{\alpha_{j,k},\beta_{j,k}\}_{j\in\{1,\dots,p\},k\in\{1,\dots,q\}} \cup \{\alpha_{j,0},\beta_{j,0}\}_{j\in\{1,\dots,p\}}
\end{array}$$

and

$$\begin{array}{rcl}
\mathsf{tv}(e) = & (\mathsf{tv}(e_1))[\sigma_j]_{j\in\{1,\dots,p\}} \cup \{\alpha_{j,0},\beta_{j,0}\}_{j\in\{1,\dots,p\}} \\
\subseteq & \mathsf{tv}(e_1) \cup \bigcup_{j\in\{1,\dots,p\}} \mathsf{tvran}(\sigma_j) \cup \{\alpha_{j,0},\beta_{j,0}\}_{j\in\{1,\dots,p\}} \\
= & \mathsf{tv}(e_1) \cup \{\alpha_{j,k},\beta_{j,k}\}_{j\in\{1,\dots,p\},k\in\{1,\dots,q\}} \cup \{\alpha_{j,0},\beta_{j,0}\}_{j\in\{1,\dots,p\}}
\end{array}$$

we have $\mathsf{dom}(\sigma_i) \subseteq \mathsf{tv}(e)$ and $\mathsf{tv}(e) \cap (\Delta \cup \Delta') = \emptyset$. Because $\Delta' \wr \Gamma_i, x : s_i^j \vdash e_1@[\sigma_i^j] : t_i^j$, by Lemma 7.4.10, we have $\Delta' \wr \Gamma_i, x : s_i^j \vdash e_1@[\sigma_i \circ \sigma_j] : t_i^j$, which is equivalent to $\Delta' \wr \Gamma_i, x : \alpha(\sigma_i \circ \sigma_j) \vdash e_1@[\sigma_i \circ \sigma_j] : \beta(\sigma_i \circ \sigma_j)$ for all i, j. Since $\Delta' \cup \bigcup_{j\in\{1,\dots,p\}} \mathsf{var}(s_i^j \to t_i^j) = \Delta'$, for a given i, by the abstraction typing rule we have $\Delta' \wr \Gamma_i \vdash \lambda^{\alpha\to\beta}_{[\sigma_i\circ\sigma_j]_{j\in\{1,\dots,p\}}} x.e_1 : \bigwedge_{j\in\{1,\dots,p\}} s_i^j \to t_i^j \leq \bigwedge_{j\in J_i} s_i^j \to t_i^j = t_i$. This is equivalent to $\Delta' \wr \Gamma_i \vdash \lambda^{\alpha\to\beta}_{[\sigma_j]_{j\in\{1,\dots,p\}}} x.e_1@[\sigma_i] : t_i$, hence $\Delta' \wr \Gamma_i \vdash e@[\sigma_i] : t_i$ holds for all i, as wished.

□

We are now ready to prove the main result of this subsection.

Theorem 7.4.24. *If $\Gamma \vdash_{BCD} m : t$, then there exist e, Δ such that $\Delta \wr \Gamma \vdash e : t$ and $\lceil e \rceil = m$.*

Proof. By Lemma 7.4.22, there exists D in intersection-abstraction normal form such that D proves $\Gamma \vdash_{BCD} m : t$. Let Δ be a set of type variables such that $\mathsf{var}(D) \subseteq \Delta$. We prove by induction on $|D|$ that there exists e such that $\Delta \wr \Gamma \vdash e : t$ and $\lceil e \rceil = m$.

Case (*BCD var*)**:** The expression m is a variable and the result holds with $e = m$.
Case (*BCD sub*)**:** We have

$$D = \frac{D' \qquad t' \leq_{BCD} t}{\Gamma \vdash_{BCD} m : t}$$

where D' in intersection-abstraction normal form and proves $\Gamma \vdash_{BCD} m : t'$. Clearly we have $|D'| < |D|$ and $\mathsf{var}(D') \subseteq \Delta$, so by the induction hypothesis, there exists e such that $\lceil e \rceil = m$ and $\Delta \wr \Gamma \vdash e : t'$. By Lemma 7.4.19, we have $t' \leq t$, therefore we have $\Delta \wr \Gamma \vdash e : t$, as wished.
Case (*BCD app*)**:** We have

$$D = \frac{D_1 \qquad D_2}{\Gamma \vdash_{BCD} m : t}$$

where D_1 proves $\Gamma \vdash_{BCD} m_1 : s \to t$, D_2 proves $\Gamma \vdash_{BCD} m_2 : s$, $m = m_1\ m_2$, and both D_1 and D_2 are in intersection-abstraction normal form. Since $|D_i| < |D|$ and $\mathsf{var}(D_i) \subseteq \Delta$, by the induction hypothesis, there exist e_1 and e_2 such that $\lceil e_1 \rceil = m_1$, $\lceil e_2 \rceil = m_2$, $\Delta \fatsemi \Gamma \vdash e_1 : s \to t$, and $\Delta \fatsemi \Gamma \vdash e_2 : s$. Consequently we have $\Delta \fatsemi \Gamma \vdash e_1\ e_2 : t$, with $\lceil e_1\ e_2 \rceil = m_1\ m_2$, as wished.

Case (*BCD abstr*) **(or** (*BCD inter*)**):** Because D is in intersection-abstraction normal form, we have

$$D = \frac{\dfrac{D_i}{\Gamma \vdash_{BCD} \lambda x.m' : s_i \to t_i}\, i \in I}{\Gamma \vdash_{BCD} m : t}$$

where each D_i is in intersection-abstraction normal form and proves $\Gamma, x : s_i \vdash_{BCD} m' : t_i$, $t = \bigwedge_{i\in I} s_i \to t_i$, and $m = \lambda x.m'$. Since $\bigcup_{i\in I} \mathsf{var}(D_i) \subseteq \Delta$, by Lemma 7.4.23, there exist e', $\sigma_1, \ldots, \sigma_n$ such that $\lceil e' \rceil = m'$, $\mathsf{dom}(\sigma_1) = \ldots = \mathsf{dom}(\sigma_n) \subseteq \mathsf{tv}(e')$, and $\Delta \fatsemi \Gamma, x : s_i \vdash e'@[\sigma_i] : t_i$ for all $i \in I$. Let α, β be two fresh type variables. We define $\sigma'_i = \sigma_i \cup \{s_i/\alpha, t_i/\beta\}$ for all $i \in I$. Because $\mathsf{dom}(\sigma_i) \cap \{\alpha, \beta\} = \emptyset$ and $\mathsf{tv}(e') \cap \{\alpha, \beta\} = \emptyset$, by Lemma 7.4.10 we have $\Delta \fatsemi \Gamma, x : s_i \vdash e'@[\sigma'_i] : t_i$, which is equivalent to $\Delta \fatsemi \Gamma, x : \alpha\sigma'_i \vdash e'@[\sigma'_i] : \beta\sigma'_i$. Note that $\Delta \cup \mathsf{var}(\bigwedge_{i\in I}(\alpha \to \beta)\sigma'_i) = \Delta \cup \mathsf{var}(\bigwedge_{i\in I} s_i \to t_i) = \Delta$ by definition of Δ, so by rule (*abstr*), we have $\Delta \fatsemi \Gamma \vdash \lambda^{\alpha\to\beta}_{[\sigma'_i]_{i\in I}} x.e' : \bigwedge_{i\in I} s_i \to t_i$. Hence we have the required result with $e = \lambda^{\alpha\to\beta}_{[\sigma'_i]_{i\in I}} x.e'$.

□

Therefore, the restricted calculus defined by the grammar (7.2) (even with types without product, union, negation, and recursive types) constitutes an explicitly-typed λ-calculus with intersection types whose expressive power subsumes those of intersection types (without a universal element). Moreover, the intersection type systems are undecidable [CD78, Bak92], which indicates that how difficult type inference of our type system is going to be (see Section 10.2 for more details).

7.4.4 Elimination of sets of type-substitutions

In this section we prove that the expressions of the form $e[\sigma_j]_{j\in J}$ are redundant insofar as their presence in the calculus does not increase its expressive power. For that we consider a subcalculus, called *normalized calculus*, in which sets of type-substitutions appear only in decorations.

Definition 7.4.25. *A* normalized expression e *is an expression without any subterm of the form* $e[\sigma_j]_{j\in J}$*, which is defined:*

$$e \ ::= \ c \ \mid \ x \ \mid \ (e, e) \ \mid \ \pi_i(e) \ \mid \ e\ e \ \mid \ \lambda^{\bigwedge_{i\in I} s_i \to t_i}_{[\sigma_j]_{j\in J}} x.e \ \mid \ e \in t\,?\,e : e$$

The set of all normalized expressions is denoted as $\mathscr{E}_N$*.*

We then define an embedding of the full calculus into this subcalculus as follows:

Definition 7.4.26. *The* embedding $emd(\_)$ *is mapping from* $\mathscr{E}$ *to* $\mathscr{E}_N$ *defined as*

$$\begin{aligned}
emd(c) &= c \\
emd(x) &= x \\
emd((e_1, e_2)) &= (emd(e_1), emd(e_2)) \\
emd(\pi_i(e)) &= \pi_i(emd(e)) \\
emd(\lambda^{\wedge_{i\in I} t_i \to s_i}_{[\sigma_j]_{j\in J}} x.e) &= \lambda^{\wedge_{i\in I} t_i \to s_i}_{[\sigma_j]_{j\in J}} x.emd(e) \\
emd(e_1 e_2) &= emd(e_1)emd(e_2) \\
emd(e{\in}t\,?\,e_1 : e_2) &= emd(e){\in}t\,?\,emd(e_1) : emd(e_2) \\
emd(e[\sigma_j]_{j\in J}) &= emd(e@[\sigma_j]_{j\in J})
\end{aligned}$$

We want to prove that the subcalculus has the same expressive power with the full calculus, namely, given an expression and its embedding, they reduce to the same value. We proceed in several steps, using auxiliary lemmas.

First we show that the embedding preserves values.

Lemma 7.4.27. *Let* $v \in \mathscr{V}$ *be a value. Then* $emd(v) \in \mathscr{V}$.

Proof. Straightforward. □

Then we prove that values and their embeddings have the same types.

Lemma 7.4.28. *Let* $v \in \mathscr{V}$ *be a value. Then* $\vdash v : t \iff \vdash emd(v) : t$.

Proof. By induction and case analysis on v (note that $emd(\_)$ does not change the types in the interfaces). □

We now want to prove the embedding preserves the reduction, that is if an expression e reduces to e' in the full calculus, then its embedding $emd(e)$ reduces to $emd(e')$ in the subcalculus. Before that we show a substitution lemma.

Lemma 7.4.29. *Let* e *be an expression,* x *an expression variable and* v *a value. Then* $emd(e\{v/x\}) = emd(e)\{emd(v)/x\}$.

Proof. By induction and case analysis on e.

$\underline{c}$**:**

$$\begin{aligned}
emd(c\{v/x\}) &= emd(c) \\
&= c \\
&= c\{emd(v)/x\} \\
&= emd(c)\{emd(v)/x\}
\end{aligned}$$

$\underline{y}$**:**

$$\begin{aligned}
emd(y\{v/x\}) &= emd(y) \\
&= y \\
&= y\{emd(v)/x\} \\
&= emd(y)\{emd(v)/x\}
\end{aligned}$$

$\underline{x}$**:**

$$\begin{aligned}
emd(x\{v/x\}) &= emd(v) \\
&= x\{emd(v)/x\} \\
&= emd(x)\{emd(v)/x\}
\end{aligned}$$

$\underline{(e_1, e_2)}$**:**

$$\begin{array}{rcll} emd((e_1,e_2)\{v/x\}) = & emd((e_1\{v/x\}, e_2\{v/x\})) & & \\ = & (emd(e_1\{v/x\}), emd(e_2\{v/x\})) & & \\ = & (emd(e_1)\{^{emd(v)}/_x\}, emd(e_2)\{^{emd(v)}/_x\}) & \text{(by induction)} & \\ = & (emd(e_1), emd(e_2))\{^{emd(v)}/_x\} & & \\ = & emd((e_1,e_2))\{^{emd(v)}/_x\} & & \end{array}$$

$\underline{\pi_i(e')}$**:**

$$\begin{array}{rcll} emd(\pi_i(e')\{v/x\}) = & emd(\pi_i(e'\{v/x\})) & & \\ = & \pi_i(emd(e'\{v/x\})) & & \\ = & \pi_i(emd(e')\{^{emd(v)}/_x\}) & \text{(by induction)} & \\ = & \pi_i(emd(e'))\{^{emd(v)}/_x\} & & \\ = & emd(\pi_i(e'))\{^{emd(v)}/_x\} & & \end{array}$$

$\underline{e_1 e_2}$**:**

$$\begin{array}{rcll} emd((e_1e_2)\{v/x\}) = & emd((e_1\{v/x\})(e_2\{v/x\})) & & \\ = & emd(e_1\{v/x\})emd(e_2\{v/x\}) & & \\ = & (emd(e_1)\{^{emd(v)}/_x\})(emd(e_2)\{^{emd(v)}/_x\}) & \text{(by induction)} & \\ = & (emd(e_1)emd(e_2))\{^{emd(v)}/_x\} & & \\ = & emd(e_1e_2)\{^{emd(v)}/_x\} & & \end{array}$$

$\underline{\lambda^{\wedge_{i\in I} t_i\to s_i}_{[\sigma_j]_{j\in J}} y.e_0}$**:** using α-conversion, we can assume that $\mathsf{tv}(v) \cap \bigcup_{j\in J} \mathsf{dom}(\sigma_j) = \emptyset$.

$$\begin{array}{rcll} emd((\lambda^{\wedge_{i\in I} t_i\to s_i}_{[\sigma_j]_{j\in J}} y.e_0)\{v/x\}) = & emd(\lambda^{\wedge_{i\in I} t_i\to s_i}_{[\sigma_j]_{j\in J}} y.e_0\{v/x\}) & & \\ = & \lambda^{\wedge_{i\in I} t_i\to s_i}_{[\sigma_j]_{j\in J}} y.emd(e_0\{v/x\}) & & \\ = & \lambda^{\wedge_{i\in I} t_i\to s_i}_{[\sigma_j]_{j\in J}} y.(emd(e_0)\{^{emd(v)}/_x\}) & \text{(by induction)} & \\ = & (\lambda^{\wedge_{i\in I} t_i\to s_i}_{[\sigma_j]_{j\in J}} y.emd(e_0))\{^{emd(v)}/_x\} & & \\ = & (emd(\lambda^{\wedge_{i\in I} t_i\to s_i}_{[\sigma_j]_{j\in J}} y.e_0))\{^{emd(v)}/_x\} & & \end{array}$$

$\underline{e_0 \in t\,?\,e_1 : e_2}$**:**

$$\begin{array}{rcl} & emd((e_0\in t\,?\,e_1 : e_2)\{v/x\}) & \\ = & emd((e_0\{v/x\})\in t\,?\,(e_1\{v/x\}) : (e_2\{v/x\})) & \\ = & emd(e_0\{v/x\})\in t\,?\,emd(e_1\{v/x\}) : emd(e_2\{v/x\}) & \\ = & emd(e_0)\{^{emd(v)}/_x\}\in t\,?\,(emd(e_1)\{^{emd(v)}/_x\}) : (emd(e_2)\{^{emd(v)}/_x\}) & \text{(by induction)} \\ = & (emd(e_0)\in t\,?\,emd(e_1) : emd(e_2))\{^{emd(v)}/_x\} & \\ = & emd(e_0\in t\,?\,e_1 : e_2)\{^{emd(v)}/_x\} & \end{array}$$

$\underline{e'[\sigma_j]_{j\in J}}$**:** using α-conversion, we can assume that $\mathsf{tv}(v) \cap \bigcup_{j\in J} \mathsf{dom}(\sigma_j) = \emptyset$

$$\begin{array}{rcll} emd((e'[\sigma_j]_{j\in J})\{v/x\}) = & emd((e'\{v/x\})[\sigma_j]_{j\in J}) & & \\ = & emd((e'\{v/x\})@[\sigma_j]_{j\in J}) & & \\ = & emd((e'@[\sigma_j]_{j\in J})\{v/x\}) & \text{(Lemma 7.4.5)} & \\ = & emd(e'@[\sigma_j]_{j\in J})\{^{emd(v)}/_x\} & \text{(by induction)} & \\ = & emd(e'[\sigma_j]_{j\in J})\{^{emd(v)}/_x\} & & \end{array}$$

□

Lemma 7.4.30. *Let $e \in \mathscr{E}$ be an expression. If $e \rightsquigarrow e'$, then $emd(e) \rightsquigarrow^* emd(e')$.*

Proof. By induction and case analysis on e.

$\underline{c, x}$**:** irreducible.

$\underline{(e_1, e_2)}$**:** there are two ways to reduce e:
(1) $e_1 \rightsquigarrow e_1'$. By induction, we have $emd(e_1) \rightsquigarrow^* emd(e_1')$. So $(emd(e_1), emd(e_2)) \rightsquigarrow^* (emd(e_1'), emd(e_2))$, that is, $emd((e_1, e_2)) \rightsquigarrow^* emd((e_1', e_2))$.
(2) $e_2 \rightsquigarrow e_2'$. Similar to the subcase above.

$\underline{\pi_i(e_0)}$**:** there are two ways to reduce e:
(1) $e_0 \rightsquigarrow e_0'$. By induction, $emd(e_0) \rightsquigarrow^* emd(e_0')$. Then we have $\pi_i(emd(e_0)) \rightsquigarrow^* \pi_i(emd(e_0'))$, that is, $emd(\pi_i(e_0)) \rightsquigarrow^* emd(\pi_i(e_0'))$.
(2) $e_0 = (v_1, v_2)$ and $e \rightsquigarrow v_i$. According to Lemma 7.4.27, $emd((v_1, v_2)) \in \mathscr{V}$. Moreover, $emd((v_1, v_2)) = (emd(v_1), emd(v_2))$. Therefore, $\pi_i(emd(v_1), emd(v_2)) \rightsquigarrow emd(v_i)$, that is, $emd(\pi_i(v_1, v_2)) \rightsquigarrow emd(v_i)$.

$\underline{e_1 e_2}$**:** there are three ways to reduce e:
(1) $e_1 \rightsquigarrow e_1'$. Similar to the case of (e_1, e_2).
(2) $e_2 \rightsquigarrow e_2'$. Similar to the case of (e_1, e_2).
(3) $e_1 = \lambda^{\bigwedge_{i\in I} t_i \to s_i}_{[\sigma_j]_{j\in J}} x.e_0$, $e_2 = v_2$ and $e_1 e_2 \rightsquigarrow (e_0@[\sigma_j]_{j\in P})\{v_2/x\}$, where $P = \{j \in J \mid \exists i \in I.\ \vdash v_2 : t_i\sigma_j\}$. According to Lemma 7.4.28, we have $\vdash v_2 : t_i\sigma_j \iff \vdash emd(v_2) : t_i\sigma_j$, thus we have $\{j \in J \mid \exists i \in I.\ \vdash emd(v_2) : t_i\sigma_j\} = \{j \in J \mid \exists i \in I.\ \vdash v_2 : t_i\sigma_j\}$. Therefore, $emd(e_1)emd(v_2) \rightsquigarrow emd(e_0@[\sigma_j]_{j\in P})\{emd(v_2)/x\}$. Moreover, by Lemma 7.4.29, $emd(e_0@[\sigma_j]_{j\in P})\{emd(v_2)/x\} = emd(e_0@[\sigma_j]_{j\in P}\{v_2/x\})$, which proves this case.

$\underline{\lambda^{\bigwedge_{i\in I} t_i \to s_i}_{[\sigma_j]_{j\in J}} x.e_0}$**:** It cannot be reduced. Thus the result follows.

$\underline{e_0 \in t\,?\,e_1 : e_2}$**:** there are three ways to reduce e:
(1) $e_i \rightsquigarrow e_i'$. Similar to the case of (e_1, e_2).
(2) $e_0 = v_0$, $\vdash v_0 : t$ and $e \rightsquigarrow e_1$. According to Lemmas 7.4.27 and 7.4.28, $emd(v_0) \in \mathscr{V}$ and $\vdash emd(v_0) : t$. So we have $emd(v_0) \in t\,?\,emd(e_1) : emd(e_2) \rightsquigarrow emd(e_1)$.
(3) $e_0 = v_0$, $\nvdash v_0 : t$ and $e \rightsquigarrow e_2$. According to Lemmas 7.4.27 and 7.4.28, $emd(v_0) \in \mathscr{V}$ and $\nvdash emd(v_0) : t$. Therefore, $emd(v_0) \in t\,?\,emd(e_1) : emd(e_2) \rightsquigarrow emd(e_2)$.

$\underline{e_0[\sigma_j]_{j\in J}}$**:** $e \rightsquigarrow e_0@[\sigma_j]_{j\in J}$. By Definition 7.4.26, $emd(e_0[\sigma_j]_{j\in J}) = emd(e_0@[\sigma_j]_{j\in J})$. Therefore, the result follows.

□

Although the embedding preserves the reduction, it does not indicate that an expression and its embedding reduce to the same value. This is because that there may be some subterms of the form $e[\sigma_j]_{j\in J}$ in the body expression of an abstraction value. For example, the expression

$$(\lambda^{\mathtt{Int}\to\mathtt{Int}\to\mathtt{Int}} z.\lambda^{\mathtt{Int}\to\mathtt{Int}} y.((\lambda^{\alpha\to\alpha} x.x)[\{\mathtt{Int}/\alpha\}]42))3$$

reduces to

$$\lambda^{\mathtt{Int}\to\mathtt{Int}}y.((\lambda^{\alpha\to\alpha}x.x)[\{\mathtt{Int}/\alpha\}]42),$$

while its embedding reduces to

$$\lambda^{\mathtt{Int}\to\mathtt{Int}}y.((\lambda^{\alpha\to\alpha}_{[\{\mathtt{Int}/\alpha\}]}x.x)42).$$

However, the embedding of the value returned by an expression is the value returned by the embedding of the expression. For instance, consider the example above again:

$$emd(\lambda^{\mathtt{Int}\to\mathtt{Int}}y.((\lambda^{\alpha\to\alpha}x.x)[\{\mathtt{Int}/\alpha\}]42)) = \lambda^{\mathtt{Int}\to\mathtt{Int}}y.((\lambda^{\alpha\to\alpha}_{[\{\mathtt{Int}/\alpha\}]}x.x)42).$$

Next, we want to prove an inversion of Lemma 7.4.30, that is, if the embedding $emd(e)$ of an expression e reduces to e', then there exists e'' such that its embedding is e' and e reduces to e''. Prior to that we prove two auxiliary lemmas: the inversions for values and for relabeled expressions.

Lemma 7.4.31. *Let $e \in \mathscr{E}$ an expression. If $emd(e) \in \mathscr{V}$, then there exists a value $v \in \mathscr{V}$ such that $e \rightsquigarrow^*_{(Rinst)} v$ and $emd(e) = emd(v)$. More specifically,*

(1) *if $emd(e) = c$, then $e \rightsquigarrow^*_{(Rinst)} c$ and $emd(e) = c$.*

(2) *if $emd(e) = \lambda^{\wedge_{i\in I}t_i\to s_i}_{[\sigma_j]_{j\in J}}x.e_0$, then there exists e'_0 such that $e \rightsquigarrow^*_{(Rinst)} \lambda^{\wedge_{i\in I}t_i\to s_i}_{[\sigma_j]_{j\in J}}x.e'_0$ and $emd(e'_0) = e_0$.*

(3) *if $emd(e) = (v_1, v_2)$, then there exist v_1, v_2 such that $e \rightsquigarrow^*_{(Rinst)} (v'_1, v'_2)$ and $emd(v'_i) = v_i$.*

Proof. By induction and case analysis on $emd(e)$.

$\underline{c}$**:** according to Definition 7.4.26, e should be the form of $c[\sigma_{j_1}]_{j_1\in J_1}\ldots[\sigma_{j_n}]_{j_n\in J_n}$, where $n \geq 0$. Clearly, we have $e \rightsquigarrow^*_{(Rinst)} c$ and $emd(e) = c$.

$\underline{\lambda^{\wedge_{i\in I}t_i\to s_i}_{[\sigma_j]_{j\in J}}x.e_0}$**:** according to Definition 7.4.26, e should be the form of

$$(\lambda^{\wedge_{i\in I}t_i\to s_i}_{[\sigma_{j_0}]_{j_0\in J_0}}x.e'_0)[\sigma_{j_1}]_{j_1\in J_1}\ldots[\sigma_{j_n}]_{j_n\in J_n}$$

where $emd(e'_0) = e_0$, $[\sigma_{j_n}]_{j_n\in J_n}\circ\ldots\circ[\sigma_{j_1}]_{j_1\in J_1}\circ[\sigma_{j_0}]_{j_0\in J_0} = [\sigma_j]_{j\in J}$, and $n \geq 0$. Moreover, it is clear that $e \rightsquigarrow^*_{(Rinst)} \lambda^{\wedge_{i\in I}t_i\to s_i}_{[\sigma_j]_{j\in J}}x.e'_0$. Let $v = \lambda^{\wedge_{i\in I}t_i\to s_i}_{[\sigma_j]_{j\in J}}x.e'_0$ and the result follows.

$\underline{(v_1, v_2)}$**:** according to Definition 7.4.26, e should be the form of

$$(e_1, e_2)[\sigma_{j_1}]_{j_1\in J_1}\ldots[\sigma_{j_n}]_{j_n\in J_n},$$

where $emd(e_i@[\sigma_{j_1}]_{j_1\in J_1}@\ldots@[\sigma_{j_n}]_{j_n\in J_n}) = v_i$ and $n \geq 0$. Moreover, it is easy to get that

$$e \rightsquigarrow^*_{(Rinst)} (e_1@[\sigma_{j_1}]_{j_1\in J_1}@\ldots@[\sigma_{j_n}]_{j_n\in J_n}, e_2@[\sigma_{j_1}]_{j_1\in J_1}@\ldots@[\sigma_{j_n}]_{j_n\in J_n})$$

By induction on v_i, there exists v'_i such that $e_i@[\sigma_{j_1}]_{j_1\in J_1}@\ldots@[\sigma_{j_n}]_{j_n\in J_n} \rightsquigarrow^*_{(Rinst)} v'_i$ and $emd(e_i@[\sigma_{j_1}]_{j_1\in J_1}@\ldots@[\sigma_{j_n}]_{j_n\in J_n}) = emd(v'_i)$. Let $v = (v'_1, v'_2)$. Then we have $e \rightsquigarrow^*_{(Rinst)} (v'_1, v'_2)$ and $emd(v) = (emd(v'_1), emd(v'_2)) = (v_1, v_2) = emd(e)$. Therefore, the result follows.

□

Lemma 7.4.32. *Let $e \in \mathscr{E}$ be an expression and $[\sigma_j]_{j\in J}$ a set of substitutions. If $emd(e@[\sigma_j]_{j\in J}) \rightsquigarrow e'$, then there exists e'' such that $e@[\sigma_j]_{j\in J} \rightsquigarrow^+ e''$ and $emd(e'') = e'$.*

Proof. By induction and case analysis on e.

$\underline{c, x}$**:** straightforward.

$\underline{(e_1, e_2)}$**:** $emd(e@[\sigma_j]_{j\in J}) = (emd(e_1@[\sigma_j]_{j\in J}), emd(e_2@[\sigma_j]_{j\in J}))$. There are two ways to reduce $emd(e@[\sigma_j]_{j\in J})$:
(1) $emd(e_1@[\sigma_j]_{j\in J}) \rightsquigarrow e_1'$. By induction, there exists e_1'' such that $e_1@[\sigma_j]_{j\in J} \rightsquigarrow^+ e_1''$ and $emd(e_1'') = e_1'$. Then we have $(e_1@[\sigma_j]_{j\in J}, e_2@[\sigma_j]_{j\in J}) \rightsquigarrow^+ (e_1'', e_2@[\sigma_j]_{j\in J})$ and $emd((e_1'', e_2@[\sigma_j]_{j\in J})) = (e_1', emd(e_2@[\sigma_j]_{j\in J}))$.
(2) $emd(e_2@[\sigma_j]_{j\in J}) \rightsquigarrow e_2'$. Similar to the subcase above.

$\underline{\pi_i(e_0)}$**:** $emd(e@[\sigma_j]_{j\in J}) = \pi_i(emd(e_0@[\sigma_j]_{j\in J}))$. There are two ways to reduce the embedding $emd(e@[\sigma_j]_{j\in J})$:
(1) $emd(e_0@[\sigma_j]_{j\in J}) \rightsquigarrow e_0'$. By induction, there exists e_0'' such that $e_0@[\sigma_j]_{j\in J} \rightsquigarrow^+ e_0''$ and $emd(e_0'') = e_0'$. Then we have $\pi_i(e_0@[\sigma_j]_{j\in J}) \rightsquigarrow^+ \pi_i(e_0'')$ and $emd(\pi_i(e_0'')) = \pi_i(e_0')$.
(2) $emd(e_0@[\sigma_j]_{j\in J}) = (v_1, v_2)$ and $emd(e@[\sigma_j]_{j\in J}) \rightsquigarrow v_i$. According to Lemma 7.4.31, there exist v_1' and v_2' such that $e_0@[\sigma_j]_{j\in J} \rightsquigarrow^*_{(Rinst)} (v_1', v_2')$ and $emd(v_i') = v_i$. Then $\pi_i(e_0@[\sigma_j]_{j\in J}) \rightsquigarrow^+ v_i'$. The result follows.

$\underline{e_1 e_2}$**:** $emd(e@[\sigma_j]_{j\in J}) = emd(e_1@[\sigma_j]_{j\in J})emd(e_2@[\sigma_j]_{j\in J})$. There are three ways to reduce $emd(e@[\sigma_j]_{j\in J})$:
(1) $emd(e_1@[\sigma_j]_{j\in J}) \rightsquigarrow e_1'$. Similar to the case of (e_1, e_2).
(2) $emd(e_2@[\sigma_j]_{j\in J}) \rightsquigarrow e_2'$. Similar to the case of (e_1, e_2).
(3) $emd(e_1@[\sigma_j]_{j\in J}) = \lambda^{\wedge_{i\in I} t_i\to s_i}_{[\sigma_k]_{k\in K}} x.e_0$, $emd(e_2@[\sigma_j]_{j\in J}) = v_2$, and $emd(e@[\sigma_j]_{j\in J}) \rightsquigarrow (e_0@[\sigma_k]_{k\in P})\{v_2/x\}$, where $P = \{k \in K \mid \exists i \in I.\ \vdash v_2 : t_i\sigma_k\}$. According to Lemma 7.4.31, we have (i) there exists e_0' such that $e_1@[\sigma_j]_{j\in J} \rightsquigarrow^*_{(Rinst)} \lambda^{\wedge_{i\in I} t_i\to s_i}_{[\sigma_k]_{k\in K}} x.e_0'$ and $emd(e_0') = e_0$; and (ii) there exists v_2' such that $e_2@[\sigma_j]_{j\in J} \rightsquigarrow^*_{(Rinst)} v_2'$ and $emd(v_2') = v_2$. Moreover, by Lemma 7.4.28, we get $\vdash v_2 : t_i\sigma_k \iff \vdash v_2' : t_i\sigma_k$, thus

$$\{k \in K \mid \exists i \in I.\ \vdash v_2 : t_i\sigma_k\} = \{k \in K \mid \exists i \in I.\ \vdash v_2' : t_i\sigma_k\}.$$

Therefore, $e@[\sigma_j]_{j\in J} \rightsquigarrow^+ (e_0'@[\sigma_k]_{k\in P})\{v_2'/x\}$. Finally, by lemma 7.4.29, we have

$$emd(e_0'@[\sigma_k]_{k\in P}\{v_2'/x\}) = emd(e_0')@[\sigma_k]_{k\in P}\{emd(v_2')/x\} = e_0@[\sigma_k]_{k\in P}\{v_2/x\}.$$

$\underline{\lambda^{\wedge_{i\in I} t_i\to s_i}_{[\sigma_k]_{k\in K}} x.e_0}$**:** It cannot be reduced. Thus the result follows.

$\underline{e_0 \in t\,?\,e_1 : e_2}$**:** $emd(e@[\sigma_j]_{j\in J}) = emd(e_0@[\sigma_j]_{j\in J}) \in t\,?\,emd(e_1@[\sigma_j]_{j\in J}) : emd(e_2@[\sigma_j]_{j\in J})$. There are three ways to reduce $emd(e@[\sigma_j]_{j\in J})$:
(1) $emd(e_i@[\sigma_j]_{j\in J}) \rightsquigarrow e_i'$. Similar to the case of (e_1, e_2).
(2) $emd(e_0@[\sigma_j]_{j\in J}) = v_0$, $\vdash v_0 : t$ and $emd(e@[\sigma_j]_{j\in J}) \rightsquigarrow emd(e_1@[\sigma_j]_{j\in J})$. According to Lemma 7.4.31, there exists v_0' such that $e_0@[\sigma_j]_{j\in J} \rightsquigarrow^*_{(Rinst)} v_0'$ and $emd(v_0') = v_0$. Moreover, by Lemma 7.4.28, $\vdash v_0' : t$. So $e@[\sigma_j]_{j\in J} \rightsquigarrow e_1@[\sigma_j]_{j\in J}$.
(3) $emd(e_0@[\sigma_j]_{j\in J}) = v_0$, $\nvdash v_0 : t$ and $emd(e@[\sigma_j]_{j\in J}) \rightsquigarrow emd(e_2@[\sigma_j]_{j\in J})$. Similar to the subcase above.

$\underline{e_0[\sigma_k]_{k\in K}}$**:** $emd(e@[\sigma_j]_{j\in J}) = emd(e_0@([\sigma_j]_{j\in J} \circ [\sigma_k]_{k\in K}))$. By induction, the result follows.

□

Lemma 7.4.33. *Let $e \in \mathscr{E}$ be an expression. If $emd(e) \rightsquigarrow e'$, then there exists e'' such that $e \rightsquigarrow^+ e''$ and $emd(e'') = e'$.*

Proof. By induction and case analysis on e.

$\underline{c}$**,** $\underline{x}$**:** straightforward.

$\underline{(e_1, e_2)}$**:** $emd(e) = (emd(e_1), emd(e_2))$. There are two ways to reduce $emd(e)$:
(1) $emd(e_1) \rightsquigarrow e_1'$. By induction, there exists e_1'' such that $e_1 \rightsquigarrow^+ e_1''$ and $emd(e_1'') = e_1'$. Then we have $(e_1, e_2) \rightsquigarrow^+ (e_1'', e_2)$ and $emd((e_1'', e_2)) = (e_1', emd(e_2))$.
(2) $emd(e_2) \rightsquigarrow e_2'$. Similar to the subcase above.

$\underline{\pi_i(e_0)}$**:** $emd(e) = \pi_i(emd(e_0))$. There are two ways to reduce $emd(e)$:
(1) $emd(e_0) \rightsquigarrow e_0'$. By induction, there exists e_0'' such that $e_0 \rightsquigarrow^+ e_0''$ and $emd(e_0'') = e_0'$. Then we have $\pi_i(e_0) \rightsquigarrow^+ \pi_i(e_0'')$ and $emd(\pi_i(e_0'')) = \pi_i(e_0')$.
(2) $emd(e_0) = (v_1, v_2)$ and $emd(e) \rightsquigarrow v_i$. According to Lemma 7.4.31, there exist v_1' and v_2' such that $e_0 \rightsquigarrow^*_{(Rinst)} (v_1', v_2')$ and $emd(v_i') = v_i$. Then $\pi_i(e_0) \rightsquigarrow^+ v_i'$. The result follows.

$\underline{e_1 e_2}$**:** $emd(e) = emd(e_1)emd(e_2)$. There are three ways to reduce $emd(e)$:
(1) $emd(e_1) \rightsquigarrow e_1'$. Similar to the case of (e_1, e_2).
(2) $emd(e_2) \rightsquigarrow e_2'$. Similar to the case of (e_1, e_2).
(3) $emd(e_1) = \lambda^{\wedge_{i\in I} t_i \to s_i}_{[\sigma_j]_{j\in J}} x.e_0$, $emd(e_2) = v_2$ and $emd(e) \rightsquigarrow (e_0@[\sigma_j]_{j\in P})\{v_2/x\}$, where $P = \{j \in J \mid \exists i \in I.\ \vdash v_2 : t_i\sigma_j\}$. According to Lemma 7.4.31, we have (i) there exists e_0' such that $e_1 \rightsquigarrow^*_{(Rinst)} \lambda^{\wedge_{i\in I} t_i \to s_i}_{[\sigma_k]_{k\in K}} x.e_0'$ and $emd(e_0') = e_0$; and (ii) there exists v_2' such that $e_2 \rightsquigarrow^*_{(Rinst)} v_2'$ and $emd(v_2') = v_2$. Moreover, by Lemma 7.4.28, we get $\vdash v_2 : t_i\sigma_j \iff \vdash v_2' : t_i\sigma_j$, thus $\{j \in J \mid \exists i \in I.\ \vdash v_2 : t_i\sigma_j\} = \{j \in J \mid \exists i \in I.\ \vdash v_2' : t_i\sigma_j\}$. Therefore, $e \rightsquigarrow^+ (e_0'@[\sigma_j]_{j\in P})\{v_2'/x\}$. Finally, by lemma 7.4.29, $emd(e_0'@[\sigma_j]_{j\in P}\{v_2'/x\}) = emd(e_0')@[\sigma_j]_{j\in P}\{emd(v_2')/x\} = e_0@[\sigma_j]_{j\in P}\{v_2/x\}$.

$\underline{\lambda^{\wedge_{i\in I} t_i \to s_i}_{[\sigma_j]_{j\in J}} x.e_0}$**:** It cannot be reduced. Thus the result follows.

$\underline{e_0 \in t\,?\,e_1 : e_2}$**:** $emd(e) = emd(e_0) \in t\,?\,emd(e_1) : emd(e_2)$. There are three ways to reduce $emd(e)$:
(1) $emd(e_i) \rightsquigarrow e_i'$. Similar to the case of (e_1, e_2).
(2) $emd(e_0) = v_0$, $\vdash v_0 : t$ and $emd(e) \rightsquigarrow emd(e_1)$. According to Lemma 7.4.31, there exists v_0' such that $e_0 \rightsquigarrow^*_{(Rinst)} v_0'$ and $emd(v_0') = v_0$. Moreover, by Lemma 7.4.28, $\vdash v_0' : t$. So $e \rightsquigarrow e_1$.
(3) $emd(e_0) = v_0$, $\nvdash v_0 : t$ and $emd(e) \rightsquigarrow emd(e_2)$. Similar to the subcase above.

$\underline{e_0[\sigma_j]_{j\in J}}$**:** $emd(e_0[\sigma_j]_{j\in J}) = emd(e_0@[\sigma_j]_{j\in J})$ and $e_0[\sigma_j]_{j\in J} \rightsquigarrow e_0@[\sigma_j]_{j\in J}$. By Lemma 7.4.32, the result follows.

□

Thus we have the following theorem

Theorem 7.4.34. *Let $e \in \mathscr{E}$ be an expression.*

(1) *if $e \rightsquigarrow^* v$, then $emd(e) \rightsquigarrow^* emd(v)$.*

(2) *if $emd(e) \rightsquigarrow^* v$, then there exists $v' \in \mathscr{V}$ such that $e \rightsquigarrow^* v'$ and $emd(v') = v$.*

Proof. (1)**:** By induction on the reduction and by Lemma 7.4.30.
(2)**:** By induction on the reduction and by Lemma 7.4.33.

□

This theorem means that the subcalculus $\mathscr{E}_N$ is equivalent to the full calculus. Although $e[\sigma_j]_{j\in J}$ do not bring any further expressive power, they play a crucial role in local type inference, which is why we included it in our calculus. As we explain in details in Chapter 9, for local type inference we need to reconstruct sets of type-substitutions that are applied to expressions but we *must not* reconstruct sets of type-substitutions that are decorations of λ-expressions. The reason is pragmatic and can be shown by considering the following two terms: $(\lambda^{\alpha\to\alpha}x.x)\mathtt{3}$ and $(\lambda^{\alpha\to\alpha}x.\mathtt{4})\mathtt{3}$. Every functional programmer will agree that the first expression must be considered well-typed while the second must not, for the simple reason that the constant function $(\lambda^{\alpha\to\alpha}x.\mathtt{4})$ does not have type $\alpha \to \alpha$. Indeed in the first case it is possible to apply a set of type substitutions that makes the term well typed, namely $(\lambda^{\alpha\to\alpha}x.x)[\{\mathtt{Int}/\alpha\}]\mathtt{3}$, while no such application exists for the second term. However, if we allowed reconstructions also for decorations, then the second term could be well typed by adding the following decoration $(\lambda^{\alpha\to\alpha}_{[\{\mathtt{Int}/\alpha\}]}x.\mathtt{4})\mathtt{3}$. In conclusion, for type inference it is important to keep the expression $e[\sigma_j]_{j\in J}$, since well-typing of $e@[\sigma_j]_{j\in J}$ does not imply that of $e[\sigma_j]_{j\in J}$.

In addition, it is easy to prove that the subcalculus $\mathscr{E}_N$ is closed under the reduction rules, and we can safely disregard (*Rinst*) since it cannot be applied. Then the normalized calculus also possess, for example, the soundness property.

Chapter 8

Type Checking

The typing rules provided in Section 7.2 are not syntax-directed because of the presence of the subsumption rule. In this chapter we present an equivalent type system with syntax-directed rules, which is used as a guide in the next chapter to define an inference system that infers where and whether type-substitutions can be inserted in an expression to make it well typed. In order to define it we consider the rules of Section 7.2. First, we merge the rules (*inst*) and (*inter*) into one rule (since we prove that intersection is interesting only to merge different instances of a same type), and then we consider where subsumption is used and whether it can be postponed by moving it down the derivation tree.

8.1 Merging intersection and instantiation

Intersection is used to merge different types derived for the same term. In this calculus, we can derive different types for a term because of either subsumption or instantiation. However, the intersection of different supertypes can be obtained by subsumption itself (if $t \leq t_1$ and $t \leq t_2$, then $t \leq t_1 \wedge t_2$), so intersection is really useful only to merge different instances of a same type, as we can see with rule (*inter*) in Figure 7.1. Note that all the subjects in the premise of (*inter*) share the same structure $e[\sigma]$, and the typing derivations of these terms must end with either (*inst*) or (*subsum*). We show that we can in fact postpone the uses of (*subsum*) after (*inter*), and we can therefore merge the rules (*inst*) and (*inter*) into one rule (*instinter*) as follows:

$$\frac{\Delta \fatsemi \Gamma \vdash e : t \qquad \forall j \in J.\ \sigma_j \sharp \Delta \qquad |J| > 0}{\Delta \fatsemi \Gamma \vdash e[\sigma_j]_{j \in J} : \bigwedge_{j \in J} t\sigma_j} \ (\mathit{instinter})$$

Let $\Delta \fatsemi \Gamma \vdash_m e : t$ denote the typing judgments derivable in the type system with the typing rule (*instinter*) but not (*inst*) and (*inter*). The following theorem proves that the type system $\vdash_m$ (m stands for "merged") is equivalent to the original one $\vdash$.

Theorem 8.1.1. *Let e be an expression. Then $\Delta \fatsemi \Gamma \vdash_m e : t \iff \Delta \fatsemi \Gamma \vdash e : t$.*

Proof. $\Rightarrow$**:** It is clear that (*inst*) is a special case of (*instinter*) where $|J| = 1$. We simulate each instance of (*instinter*) where $|J| > 1$ by using several instances

of (*inst*) followed by one instance of (*inter*). In detail, consider the following derivation

$$\dfrac{\dfrac{\dfrac{\dots}{\Delta' \fatsemi \Gamma' \vdash e' : t'} \quad \sigma_j \sharp \Delta'}{\Delta' \fatsemi \Gamma' \vdash e'[\sigma_j]_{j\in J} : \bigwedge_{j\in J} t'\sigma_j}\,(instinter)}{\dots \quad \vdots \quad \dots}{\Delta \fatsemi \Gamma \vdash e : t}$$

We can rewrite this derivation as follows:

$$\dfrac{\dfrac{\dfrac{\dfrac{\dots}{\Delta' \fatsemi \Gamma' \vdash e' : t'} \quad \sigma_1 \sharp \Delta'}{\Delta' \fatsemi \Gamma' \vdash e'[\sigma_1] : t'\sigma_1}\,(inst) \quad \dots \quad \dfrac{\dfrac{\dots}{\Delta' \fatsemi \Gamma' \vdash e' : t'} \quad \sigma_{|J|} \sharp \Delta'}{\Delta' \fatsemi \Gamma' \vdash e'[\sigma_{|J|}] : t'\sigma_{|J|}}\,(inst)}{\Delta' \fatsemi \Gamma' \vdash e'[\sigma_j]_{j\in J} : \bigwedge_{j\in J} t'\sigma_j}\,(inter)}{\dots \quad \vdots \quad \dots}{\Delta \fatsemi \Gamma \vdash e : t}$$

$\Leftarrow$**:** The proof proceeds by induction and case analysis on the structure of e. For each case we use an auxiliary internal induction on the typing derivation. We label **E** the main (external) induction and **I** the internal induction in what follows.

$e = c$**:** the typing derivation $\Delta \fatsemi \Gamma \vdash e : t$ should end with either (*const*) or (*subsum*). If the typing derivation ends with (*const*), the result follows straightforward.
Otherwise, the typing derivation ends with an instance of (*subsum*):

$$\dfrac{\dfrac{\dots}{\Delta \fatsemi \Gamma \vdash e : s} \quad s \leq t}{\Delta \fatsemi \Gamma \vdash e : t}\,(subsum)$$

Then by **I**-induction, we have $\Delta \fatsemi \Gamma \vdash_m e : s$. Since $s \leq t$, by subsumption, we get $\Delta \fatsemi \Gamma \vdash_m e : t$.

$e = x$**:** similar to the case of $e = c$.

$e = (e_1, e_2)$**:** the typing derivation $\Delta \fatsemi \Gamma \vdash e : t$ should end with either (*pair*) or (*subsum*). Assume that the typing derivation ends with (*pair*):

$$\dfrac{\dfrac{\dots}{\Delta \fatsemi \Gamma \vdash e_1 : t_1} \quad \dfrac{\dots}{\Delta \fatsemi \Gamma \vdash e_2 : t_2}}{\Delta \fatsemi \Gamma \vdash (e_1, e_2) : t_1 \times t_2}\,(pair)$$

By **E**-induction, we have $\Delta \fatsemi \Gamma \vdash_m e_i : t_i$. Then the rule (*pair*) gives us $\Delta \fatsemi \Gamma \vdash_m (e_1, e_2) : t_1 \times t_2$.
Otherwise, the typing derivation ends with an instance of (*subsum*), similar to the case of $e = c$, the result follows by **I**-induction.

$e = \pi_i(e')$**:** the typing derivation $\Delta \fatsemi \Gamma \vdash e : t$ should end with either (*proj*) or (*subsum*). Assume that the typing derivation ends with (*proj*):

$$\dfrac{\dfrac{\dots}{\Delta \fatsemi \Gamma \vdash e' : (t_1 \times t_2)}}{\Delta \fatsemi \Gamma \vdash \pi_i(e') : t_i}\,(proj)$$

By **E**-induction, we have $\Delta \fatsemi \Gamma \vdash_m e' : (t_1 \times t_2)$. Then the rule (*proj*) gives us $\Delta \fatsemi \Gamma \vdash_m \pi_i(e') : t_i$.

Otherwise, the typing derivation ends with an instance of (*subsum*), similar to the case of $e = c$, the result follows by **I**-induction.

$\underline{e = e_1 e_2}$**:** the typing derivation $\Delta \fatsemi \Gamma \vdash e : t$ should end with either (*appl*) or (*subsum*). Assume that the typing derivation ends with (*appl*):

$$\frac{\dfrac{\cdots}{\Delta \fatsemi \Gamma \vdash e_1 : t_1 \to t_2} \quad \dfrac{\cdots}{\Delta \fatsemi \Gamma \vdash e_2 : t_1}}{\Delta \fatsemi \Gamma \vdash e_1 e_2 : t_2}\ (appl)$$

By **E**-induction, we have $\Delta \fatsemi \Gamma \vdash_m e_1 : t_1 \to t_2$ and $\Delta \fatsemi \Gamma \vdash_m e_2 : t_1$. Then the rule (*appl*) gives us $\Delta \fatsemi \Gamma \vdash_m e_1 e_2 : t_2$.

Otherwise, the typing derivation ends with an instance of (*subsum*), similar to the case of $e = c$, the result follows by **I**-induction.

$\underline{e = \lambda^{\wedge_{i\in I} t_i \to s_i}_{[\sigma_j]_{j\in J}} x.e'}$**:** the typing derivation $\Delta \fatsemi \Gamma \vdash e : t$ should end with either (*abstr*) or (*subsum*). Assume that the typing derivation ends with (*abstr*):

$$\frac{\begin{array}{l}\forall i \in I, j \in J.\ \dfrac{\cdots}{\Delta' \fatsemi \Gamma, (x : t_i\sigma_j) \vdash e'@[\sigma_j] : s_i\sigma_j} \\ \Delta' = \Delta \cup \mathsf{var}(\bigwedge_{i\in I, j\in J}(t_i\sigma_j \to s_i\sigma_j))\end{array}}{\Delta \fatsemi \Gamma \vdash \lambda^{\wedge_{i\in I} t_i \to s_i}_{[\sigma_j]_{j\in J}} x.e' : \bigwedge_{i\in I, j\in J}(t_i\sigma_j \to s_i\sigma_j)}\ (abstr)$$

By **E**-induction, for all $i \in I$ and $j \in J$, we have $\Delta' \fatsemi \Gamma, (x : t_i\sigma_j) \vdash_m e'@[\sigma_j] : s_i\sigma_j$. Then the rule (*abstr*) gives us $\Delta \fatsemi \Gamma \vdash_m \lambda^{\wedge_{i\in I} t_i \to s_i}_{[\sigma_j]_{j\in J}} x.e' : \bigwedge_{i\in I, j\in J}(t_i\sigma_j \to s_i\sigma_j)$.

Otherwise, the typing derivation ends with an instance of (*subsum*), similar to the case of $e = c$, the result follows by **I**-induction.

$\underline{e = e' \in t\,?\,e_1 : e_2}$**:** the typing derivation $\Delta \fatsemi \Gamma \vdash e : t$ should end with either (*case*) or (*subsum*). Assume that the typing derivation ends with (*case*):

$$\frac{\dfrac{\cdots}{\Delta \fatsemi \Gamma \vdash e' : t'} \quad \begin{cases} t' \not\leq \neg t & \Rightarrow \ \dfrac{\cdots}{\Delta \fatsemi \Gamma \vdash e_1 : s} \\ t' \not\leq t & \Rightarrow \ \dfrac{\cdots}{\Delta \fatsemi \Gamma \vdash e_2 : s} \end{cases}}{\Delta \fatsemi \Gamma \vdash (e' \in t\,?\,e_1 : e_2) : s}\ (case)$$

By **E**-induction, we have $\Delta \fatsemi \Gamma \vdash_m e' : t'$ and $\Delta \fatsemi \Gamma \vdash_m e_i : s$ (for i such that e_i has been effectively type-checked). Then the rule (*case*) gives us $\Delta \fatsemi \Gamma_m \vdash (e' \in t\,?\,e_1 : e_2) : s$.

Otherwise, the typing derivation ends with an instance of (*subsum*), similar to the case of $e = c$, the result follows by **I**-induction.

$\underline{e = e'[\sigma]}$**:** the typing derivation $\Delta \fatsemi \Gamma \vdash e : t$ should end with either (*inst*) or (*subsum*). Assume that the typing derivation ends with (*inst*):

$$\frac{\dfrac{\cdots}{\Delta \fatsemi \Gamma \vdash e' : t} \quad \sigma \sharp \Delta}{\Delta \fatsemi \Gamma \vdash e'[\sigma] : t\sigma}\ (inst)$$

By **E**-induction, we have $\Delta \fatsemi \Gamma \vdash_m e' : t$. Since $\sigma \sharp \Delta$, applying (*instinter*) where $|J| = 1$, we get $\Delta \fatsemi \Gamma \vdash_m e'[\sigma] : t\sigma$.
Otherwise, the typing derivation ends with an instance of (*subsum*), similar to the case of $e = c$, the result follows by **I**-induction.

$\underline{e = e'[\sigma_j]_{j\in J}}$**:** the typing derivation $\Delta \fatsemi \Gamma \vdash e : t$ should end with either (*inter*) or (*subsum*). Assume that the typing derivation ends with (*inter*):

$$\frac{\forall j \in J.\ \dfrac{\cdots}{\Delta \fatsemi \Gamma \vdash e'[\sigma_j] : t_j} \qquad |J| > 1}{\Delta \fatsemi \Gamma \vdash e'[\sigma_j]_{j\in J} : \bigwedge_{j\in J} t_j}\ (\mathit{inter})$$

As an intermediary result, we first prove that the derivation can be rewritten as

$$\dfrac{\forall j \in J.\ \dfrac{\dfrac{\cdots}{\Delta \fatsemi \Gamma \vdash e' : s} \qquad \sigma_j \sharp \Delta}{\Delta \fatsemi \Gamma \vdash e'[\sigma_j] : s\sigma_j}\ (\mathit{inst})}{\dfrac{\Delta \fatsemi \Gamma \vdash e'[\sigma_j]_{j\in J} : \bigwedge_{j\in J} s\sigma_j}{\Delta \fatsemi \Gamma \vdash e'[\sigma_j]_{j\in J} : \bigwedge_{j\in J} t_j}\ (\mathit{subsum})}\ (\mathit{inter})\qquad \bigwedge_{j\in J} s\sigma_j \leq \bigwedge_{j\in J} t_j$$

We proceed by induction on the original derivation. It is clear that each sub-derivation $\Delta \fatsemi \Gamma \vdash e'[\sigma_j] : t_j$ ends with either (*inst*) or (*subsum*). If all the sub-derivations end with an instance of (*inst*), then for all $j \in J$, we have

$$\frac{\dfrac{\cdots}{\Delta \fatsemi \Gamma \vdash e' : s_j} \qquad \sigma_j \sharp \Delta}{\Delta \fatsemi \Gamma \vdash e'[\sigma_j] : s_j\sigma_j}\ (\mathit{inst})$$

By Lemma 7.4.2, we have $\Delta \fatsemi \Gamma \vdash e' : \bigwedge_{j\in J} s_j$. Let $s = \bigwedge_{j\in J} s_j$. Then by (*inst*), we get $\Delta \fatsemi \Gamma \vdash e'[\sigma_j] : s\sigma_j$. Finally, by (*inter*) and (*subsum*), the intermediary result holds. Otherwise, there is at least one of the sub-derivations ends with an instance of (*subsum*), the intermediary result also hold by induction.
Now that the intermediary result is proved, we go back to the proof of the lemma. By **E**-induction on e' (*i.e.*, $\Delta \fatsemi \Gamma \vdash e' : s$), we have $\Delta \fatsemi \Gamma \vdash_m e' : s$. Since $\sigma_j \sharp \Delta$, applying (*instinter*), we get $\Delta \fatsemi \Gamma \vdash_m e'[\sigma_j]_{j\in J} : \bigwedge_{j\in J} s\sigma_j$. Finally, by subsumption, we get $\Delta \fatsemi \Gamma \vdash_m e'[\sigma]_{j\in J} : \bigwedge_{j\in J} t_j$.
Otherwise, the typing derivation ends with an instance of (*subsum*), similar to the case of $e = c$, the result follows by **I**-induction.

□

From now on we will use $\vdash$ to denote $\vdash_m$, that is the system with the merged rule.

8.2 Algorithmic typing rules

In this section, we analyse the typing derivations produced by the rules of Section 7.2 to see where subsumption is needed and where it can be pushed down the derivation tree. We need first some preliminary definitions and decomposition results about "pair types" and "function types". Intuitively a pair type is a type that ensures that every

(terminating) expression with that type will return a pair of values. Clearly a product type is a pair type but the opposite does not hold in general (*eg*, a union of disjoint products —which in general cannot be expressed as a product— is a pair type, too). Similarly, a function type is a type that ensures that every terminating expression of that type will return a function and, likewise, arrow types are functions types but the opposite may not be true. Let us study some properties of these types that will help us to deal with the projection and application rules.

8.2.1 Pair types

A type s is a pair type if $s \leq \mathbb{1} \times \mathbb{1}$. If an expression e is typable with a pair type s, we want to compute from s a valid type for $\pi_i(e)$. In $\mathbb{C}$Duce, a pair type s is a finite union of product types, which can be decomposed into a finite set of pairs of types, denoted as $\boldsymbol{\pi}(s)$. For example, we decompose $s = (t_1 \times t_2) \vee (s_1 \times s_2)$ as $\boldsymbol{\pi}(s) = \{(t_1, t_2), (s_1, s_2)\}$. We can then compute easily a type $\boldsymbol{\pi}_i(s)$ for $\pi_i(e)$ as $\boldsymbol{\pi}_i(s) = t_i \vee s_i$ (we used boldface symbols to distinguish these type operators from the projections used in expressions). In the calculus considered here, the situation becomes more complex because of type variables, especially top level ones. Let s be a pair type that contains a top-level variable α. Since $\alpha \not\leq \mathbb{1} \times \mathbb{1}$ and $s \leq \mathbb{1} \times \mathbb{1}$, then it is not possible that $s \simeq s' \vee \alpha$. In other terms the toplevel variable cannot appear alone in a union: it must occur intersected with some product type so that it does not "overtake" the $\mathbb{1} \times \mathbb{1}$ bound. Consequently, we have $s \simeq s' \wedge \alpha$ for some $s' \leq \mathbb{1} \times \mathbb{1}$. However, in a typing derivation starting from $\Delta \fatsemi \Gamma \vdash e : s$ and ending with $\Delta \fatsemi \Gamma \vdash \pi_i(e) : t$, there exists an intermediary step where e is assigned a type of the form $(t_1 \times t_2)$ (and that verifies $s \leq (t_1 \times t_2)$) before applying the projection rule. So it is necessary to get rid of the top-level variables of s (using subsumption) before computing the projection. The example above shows that α does not play any role since it is the s' component that will be used to subsume s to a product type. To say it otherwise, since e has type s for all possible assignement of α, then the typing derivation must hold also for $\alpha = \mathbb{1}$. In whatever way we look at it, the top-level type variables are useless and can be safely discarded when decomposing s.

Formally, we define the decomposition of a pair type as follows:

Definition 8.2.1. *Let τ be a disjunctive normal form such that $\tau \leq \mathbb{1} \times \mathbb{1}$. We define the* decomposition *of τ as*

$$
\begin{aligned}
&\boldsymbol{\pi}(\textstyle\bigvee_{i\in I} \tau_i) = \bigcup_{i\in I} \boldsymbol{\pi}(\tau_i)\\
&\boldsymbol{\pi}(\textstyle\bigwedge_{j\in P}(t_1^j \times t_2^j) \wedge \bigwedge_{k\in N} \neg(t_1^k \times t_2^k) \wedge \bigwedge_{\alpha\in P_\mathcal{V}} \alpha \wedge \bigwedge_{\alpha'\in N_\mathcal{V}} \neg\alpha') \quad (|P| > 0)\\
&\qquad = \boldsymbol{\pi}(\textstyle\bigvee_{N'\subseteq N}((\bigwedge_{j\in P} t_1^j \wedge \bigwedge_{k\in N'} \neg t_1^k) \times (\bigwedge_{j\in P} t_2^j \wedge \bigwedge_{k\in N\setminus N'} \neg t_2^k)))\\
&\boldsymbol{\pi}((t_1 \times t_2)) = \begin{cases} \{(t_1, t_2)\} & t_1 \not\simeq \mathbb{0} \text{ and } t_2 \not\simeq \mathbb{0} \\ \emptyset & \text{otherwise} \end{cases}
\end{aligned}
$$

and the i-th projection *of τ as $\boldsymbol{\pi}_i(\tau) = \bigvee_{(s_1,s_2)\in\boldsymbol{\pi}(\tau)} s_i$.*
For all type t such that $t \leq \mathbb{1} \times \mathbb{1}$, the decomposition of t is defined as

$$\boldsymbol{\pi}(t) = \boldsymbol{\pi}(\mathsf{dnf}((\mathbb{1} \times \mathbb{1}) \wedge t))$$

and the i-th projection of t as $\boldsymbol{\pi}_i(t) = \bigvee_{(s_1,s_2)\in\boldsymbol{\pi}(\mathsf{dnf}((\mathbb{1}\times\mathbb{1})\wedge t))} s_i$

The decomposition of a union of pair types is the union of each decomposition. When computing the decomposition of an intersection of product types and type variables, we compute all the possible distributions of the intersections over the products and we discard the variables as discussed above. Finally, the decomposition of a product is the pair of its two components, provided that both components are not empty.

The decomposition of pair types defined above has the following properties:

Lemma 8.2.2. *Let $\leq$ be the subtyping relation induced by a well-founded (convex) model with infinite support and t a type such that $t \leq \mathbb{1} \times \mathbb{1}$. Then*

(1) *For all $(t_1, t_2) \in \boldsymbol{\pi}(t)$, we have $t_1 \not\simeq \mathbb{0}$ and $t_2 \not\simeq \mathbb{0}$;*

(2) *For all s_1, s_2, we have $t \leq (s_1 \times s_2) \iff \bigvee_{(t_1,t_2)\in\boldsymbol{\pi}(t)}(t_1 \times t_2) \leq (s_1 \times s_2)$.*

Proof. (1)**:** straightforward.
(2)**:** Since $t \leq \mathbb{1} \times \mathbb{1}$, we have

$$t \simeq \bigvee_{(P,N)\in\mathsf{dnf}(t)} ((\mathbb{1}\times\mathbb{1}) \wedge \bigwedge_{j\in P\setminus\mathcal{V}} (t_1^j \times t_2^j) \wedge \bigwedge_{k\in N\setminus\mathcal{V}} \neg(t_1^k \times t_2^k) \wedge \bigwedge_{\alpha\in P\cap\mathcal{V}} \alpha \wedge \bigwedge_{\alpha'\in N\cap\mathcal{V}} \neg\alpha')$$

If $t \simeq \mathbb{0}$, then $\boldsymbol{\pi}(t) = \emptyset$, and the result holds. Assume that $t \not\simeq \mathbb{0}$, $|P| > 0$ and each summand of $\mathsf{dnf}(t)$ is not equivalent to $\mathbb{0}$ as well. Let $\ulcorner t \urcorner$ denote the type $\bigvee_{(P,N)\in\mathsf{dnf}(t)}(\bigwedge_{j\in P\setminus\mathcal{V}}(t_1^j \times t_2^j) \wedge \bigwedge_{k\in N\setminus\mathcal{V}} \neg(t_1^k \times t_2^k))$. Using the set-theoretic interpretation of types we have that $\ulcorner t \urcorner$ is equivalent to

$$\bigvee_{(P,N)\in\mathsf{dnf}(t)} \Big(\bigvee_{N'\subseteq N\setminus\mathcal{V}} \Big(\Big(\bigwedge_{j\in P\setminus\mathcal{V}} t_1^j \wedge \bigwedge_{k\in N'} \neg t_1^k \Big) \times \Big(\bigwedge_{j\in P\setminus\mathcal{V}} t_2^j \wedge \bigwedge_{k\in(N\setminus\mathcal{V})\setminus N'} \neg t_2^k \Big) \Big) \Big)$$

This means that, $\ulcorner t \urcorner$ is a equivalent to a union of product types. Let us rewrite this union more explicitly, that is, $\ulcorner t \urcorner \simeq \bigvee_{i\in I}(t_1^i \times t_2^i)$ obtained as follows

$$\bigvee_{(P,N)\in\mathsf{dnf}(t)} \Big(\bigvee_{N'\subseteq N\setminus\mathcal{V}} \Big(\overbrace{\Big(\bigwedge_{j\in P\setminus\mathcal{V}} t_1^j \wedge \bigwedge_{k\in N'} \neg t_1^k \Big)}^{t_1^i} \times \overbrace{\Big(\bigwedge_{j\in P\setminus\mathcal{V}} t_2^j \wedge \bigwedge_{k\in(N\setminus\mathcal{V})\setminus N'} \neg t_2^k \Big)}^{t_2^i} \Big) \Big)$$

We have $\boldsymbol{\pi}(t) = \{(t_1^i, t_2^i) \mid i \in I \text{ and } t_1^i \not\simeq \mathbb{0} \text{ and } t_2^i \not\simeq \mathbb{0}\}$. Finally, for all pair of types s_1 and s_2, we have

$$\begin{aligned}
& t \leq (s_1 \times s_2) \\
\iff & \bigvee_{(P,N)\in\mathsf{dnf}(t)} \Big(\bigwedge_{j\in P\setminus\mathcal{V}} (t_1^j \times t_2^j) \wedge \bigwedge_{k\in N\setminus\mathcal{V}} \neg(t_1^k \times t_2^k) \wedge \bigwedge_{\alpha\in P\cap\mathcal{V}} \alpha \wedge \bigwedge_{\alpha'\in N\cap\mathcal{V}} \neg\alpha' \Big) \leq (s_1 \times s_2) \\
\iff & \bigvee_{(P,N)\in\mathsf{dnf}(t)} \Big(\bigwedge_{j\in P\setminus\mathcal{V}} (t_1^j \times t_2^j) \wedge \bigwedge_{k\in N\setminus\mathcal{V}} \neg(t_1^k \times t_2^k) \wedge \bigwedge_{\alpha\in P\cap\mathcal{V}} \alpha \wedge \bigwedge_{\alpha'\in N\cap\mathcal{V}} \neg\alpha' \wedge \neg(s_1 \times s_2) \Big) \leq \mathbb{0} \\
\iff & \bigvee_{(P,N)\in\mathsf{dnf}(t)} \Big(\bigwedge_{j\in P\setminus\mathcal{V}} (t_1^j \times t_2^j) \wedge \bigwedge_{k\in N\setminus\mathcal{V}} \neg(t_1^k \times t_2^k) \wedge \neg(s_1 \times s_2) \Big) \leq \mathbb{0} \quad \text{(Lemma 5.1.2)} \\
\iff & \bigvee_{(P,N)\in\mathsf{dnf}(t)} \Big(\bigwedge_{j\in P\setminus\mathcal{V}} (t_1^j \times t_2^j) \wedge \bigwedge_{k\in N\setminus\mathcal{V}} \neg(t_1^k \times t_2^k) \Big) \leq (s_1 \times s_2) \\
\iff & \ulcorner t \urcorner \leq (s_1 \times s_2) \\
\iff & \bigvee_{(t_1,t_2)\in\boldsymbol{\pi}(t)} (t_1 \times t_2) \leq (s_1 \times s_2)
\end{aligned}$$

□

Lemma 8.2.3. *Let s be a type such that $s \leq (t_1 \times t_2)$. Then*

(1) $s \leq (\boldsymbol{\pi}_1(s) \times \boldsymbol{\pi}_2(s))$;

(2) $\boldsymbol{\pi}_i(s) \leq t_i$.

Proof. (1)**:** according to the proof of Lemma 8.2.2, $\bigvee_{(s_1,s_2)\in\boldsymbol{\pi}(s)}(s_1 \times s_2)$ is equivalent to the type obtained from s by ignoring all the top-level type variables. Then it is trivial that $s \leq \bigvee_{(s_1,s_2)\in\boldsymbol{\pi}(s)}(s_1 \times s_2)$ and then $s \leq (\boldsymbol{\pi}_1(s) \times \boldsymbol{\pi}_2(s))$.

(2)**:** since $s \leq (t_1 \times t_2)$, according to Lemma 8.2.2, we have $\bigvee_{(s_1,s_2)\in\boldsymbol{\pi}(s)}(s_1 \times s_2) \leq (t_1 \times t_2)$. So for all $(s_1, s_2) \in \boldsymbol{\pi}(s)$, we have $(s_1 \times s_2) \leq (t_1 \times t_2)$. Moreover, as s_i is not empty, we have $s_i \leq t_i$. Therefore, $\boldsymbol{\pi}_i(s) \leq t_i$.

□

Lemma 8.2.4. *Let t and s be two types such that $t \leq \mathbb{1} \times \mathbb{1}$ and $s \leq \mathbb{1} \times \mathbb{1}$. Then $\boldsymbol{\pi}_i(t \wedge s) \leq \boldsymbol{\pi}_i(t) \wedge \boldsymbol{\pi}_i(s)$.*

Proof. Let $t = \bigvee_{j_1\in J_1} \tau_{j_1}$ and $s = \bigvee_{j_2\in J_2} \tau_{j_2}$ such that

$$\tau_j = (t_j^1 \times t_j^2) \wedge \bigwedge_{\alpha\in P_j} \alpha \wedge \bigwedge_{\alpha'\in N_j} \neg\alpha'$$

and $\tau_j \not\simeq \mathbb{0}$ for all $j \in J_1 \cup J_2$. Then we have $t \wedge s = \bigvee_{j_1\in J_1, j_2\in J_2} \tau_{j_1} \wedge \tau_{j_2}$. Let $j_1 \in J_1$ and $j_2 \in J_2$. If $\tau_{j_1} \wedge \tau_{j_2} \simeq \mathbb{0}$, we have $\boldsymbol{\pi}_i(\tau_{j_1} \wedge \tau_{j_2}) = \mathbb{0}$. Otherwise, $\boldsymbol{\pi}_i(\tau_{j_1} \wedge \tau_{j_2}) = t_{j_1}^i \wedge t_{j_2}^i = \boldsymbol{\pi}_i(\tau_{j_1}) \wedge \boldsymbol{\pi}_i(\tau_{j_2})$. For both cases, we have $\boldsymbol{\pi}_i(\tau_{j_1} \wedge \tau_{j_2}) \leq \boldsymbol{\pi}_i(\tau_{j_1}) \wedge \boldsymbol{\pi}_i(\tau_{j_2})$. Therefore

$$\begin{array}{rcl} \boldsymbol{\pi}_i(t \wedge s) & \simeq & \bigvee_{j_1\in J_1, j_2\in J_2} \boldsymbol{\pi}_i(\tau_{j_1} \wedge \tau_{j_2}) \\ & \leq & \bigvee_{j_1\in J_1, j_2\in J_2} (\boldsymbol{\pi}_i(\tau_{j_1}) \wedge \boldsymbol{\pi}_i(\tau_{j_2})) \\ & \simeq & (\bigvee_{j_1\in J_1} \boldsymbol{\pi}_i(\tau_{j_1})) \wedge (\bigvee_{j_2\in J_2} \boldsymbol{\pi}_i(\tau_{j_2})) \\ & \simeq & \boldsymbol{\pi}_i(t) \wedge \boldsymbol{\pi}_i(s) \end{array}$$

□

For example, $\boldsymbol{\pi}_1((\mathtt{Int} \times \mathtt{Int}) \wedge (\mathtt{Int} \times \mathtt{Bool})) = \boldsymbol{\pi}_1(\mathbb{0}) = \mathbb{0}$, while $\boldsymbol{\pi}_1((\mathtt{Int} \times \mathtt{Int})) \wedge \boldsymbol{\pi}_1((\mathtt{Int} \times \mathtt{Bool})) = \mathtt{Int} \wedge \mathtt{Int} = \mathtt{Int}$.

Lemma 8.2.5. *Let t be a type and σ be a type substitution such that $t \leq \mathbb{1} \times \mathbb{1}$. Then $\boldsymbol{\pi}_i(t\sigma) \leq \boldsymbol{\pi}_i(t)\sigma$.*

Proof. We put t into its disjunctive normal form $\bigvee_{j\in J} \tau_j$ such that

$$\tau_j = (t_j^1 \times t_j^2) \wedge \bigwedge_{\alpha\in P_j} \alpha \wedge \bigwedge_{\alpha'\in N_j} \neg\alpha'$$

and $\tau_j \not\simeq \mathbb{0}$ for all $j \in J$. Then we have $t\sigma = \bigvee_{j\in J} \tau_j\sigma$. So $\boldsymbol{\pi}_i(t\sigma) = \bigvee_{j\in J} \boldsymbol{\pi}_i(\tau_j\sigma)$. Let $j \in J$. If $\tau_j\sigma \simeq \mathbb{0}$, then $\boldsymbol{\pi}_i(\tau_j\sigma) = \mathbb{0}$ and trivially $\boldsymbol{\pi}_i(\tau_j\sigma) \leq \boldsymbol{\pi}_i(\tau_j)\sigma$. Otherwise, we have $\tau_j\sigma = (t_j^1\sigma \times t_j^2\sigma) \wedge (\bigwedge_{\alpha\in P_j} \alpha \wedge \bigwedge_{\alpha'\in N_j} \neg\alpha')\sigma$. By Lemma 8.2.4, we get $\boldsymbol{\pi}_i(\tau_j\sigma) \leq t_j^i\sigma \wedge \boldsymbol{\pi}_i((\bigwedge_{\alpha\in P_j} \alpha \wedge \bigwedge_{\alpha'\in N_j} \neg\alpha')\sigma) \leq t_j^i\sigma \simeq \boldsymbol{\pi}_i(\tau_j)\sigma$. Therefore, $\bigvee_{j\in J} \boldsymbol{\pi}_i(\tau_j\sigma) \leq \bigvee_{j\in J} \boldsymbol{\pi}_i(\tau_j)\sigma$, that is, $\boldsymbol{\pi}_i(t\sigma) \leq \boldsymbol{\pi}_i(t)\sigma$. □

For example, $\boldsymbol{\pi}_1(((\texttt{Int} \times \texttt{Int}) \wedge \alpha)\{(\texttt{Int} \times \texttt{Bool})/\alpha\}) = \boldsymbol{\pi}_1((\texttt{Int} \times \texttt{Int}) \wedge (\texttt{Int} \times \texttt{Bool})) = \mathbb{0}$, while $(\boldsymbol{\pi}_1((\texttt{Int} \times \texttt{Int}) \wedge \alpha))\{(\texttt{Int} \times \texttt{Bool})/\alpha\} = \texttt{Int}\{(\texttt{Int} \times \texttt{Bool})/\alpha\} = \texttt{Int}$.

Lemma 8.2.6. *Let t be a type such that $t \leq \mathbb{1} \times \mathbb{1}$ and $[\sigma_k]_{k \in K}$ be a set of type substitutions. Then $\boldsymbol{\pi}_i(\bigwedge_{k \in K} t\sigma_k) \leq \bigwedge_{k \in K} \boldsymbol{\pi}_i(t)\sigma_k$.*

Proof. Consequence of Lemmas 8.2.4 and 8.2.5. □

8.2.2 Function types

A type t is a function type if $t \leq \mathbb{0} \to \mathbb{1}$. In order to type the application of a function having a function type t, we need to determine the domain of t, that is, the set of values the function can be safely applied to. This problem has been solved for ground function types in [FCB08]. Again, the problem becomes more complex if t contains top-level type variables. Another issue is to determine what is the result type of an application of a function type t to an argument of type s (where s belongs to the domain of t), knowing that both t and s may contain type variables.

Following the same reasoning as with pair types, if a function type t contains a top-level variable α, then $t \simeq t' \wedge \alpha$ for some function type t'. In a typing derivation for a judgement $\Delta \fatsemi \Gamma \vdash e_1\ e_2 : t$ which contains $\Delta \fatsemi \Gamma \vdash e_1 : t$, there exists an intermediary step where we assign a type $t_1 \to t_2$ to e_1 (with $t \leq t_1 \to t_2$) before using the application rule. It is therefore necessary to eliminate the top-level variables from the function type t before we can type an application. Once more, the top-level variables are useless when computing the domain of t and can be safely discarded.

Formally, we define the domain of a function type as follows:

Definition 8.2.7 (Domain). *Let τ be a disjunctive normal form such that $\tau \leq \mathbb{0} \to \mathbb{1}$. We define* $\mathsf{dom}(\tau)$*, the* domain *of τ, as:*

$$\begin{aligned}
&\mathsf{dom}(\textstyle\bigvee_{i \in I} \tau_i) = \bigwedge_{i \in I} \mathsf{dom}(\tau_i)\\
&\mathsf{dom}(\textstyle\bigwedge_{j \in P}(t_1^j \to t_2^j) \wedge \bigwedge_{k \in N} \neg(t_1^k \to t_2^k) \wedge \bigwedge_{\alpha \in P_\mathcal{V}} \alpha \wedge \bigwedge_{\alpha' \in N_\mathcal{V}} \neg\alpha')\\
&\quad = \begin{cases} \mathbb{1} & \text{if } \bigwedge_{j \in P}(t_1^j \to t_2^j) \wedge \bigwedge_{k \in N} \neg(t_1^k \to t_2^k) \wedge \bigwedge_{\alpha \in P_\mathcal{V}} \alpha \wedge \bigwedge_{\alpha' \in N_\mathcal{V}} \neg\alpha' \simeq \mathbb{0} \\ \bigvee_{j \in P} t_1^j & \text{otherwise} \end{cases}
\end{aligned}$$

For any type t such that $t \leq \mathbb{0} \to \mathbb{1}$, the domain of t is defined as

$$\mathsf{dom}(t) = \mathsf{dom}(\mathsf{dnf}((\mathbb{0} \to \mathbb{1}) \wedge t)).$$

We also define a decomposition operator $\boldsymbol{\phi}$ that —akin to the decomposition operator $\boldsymbol{\pi}$ for product types— decomposes a function type into a finite set of pairs:

Definition 8.2.8. *Let τ be a disjunctive normal form such that $\tau \leq \mathbb{0} \to \mathbb{1}$. We define the* decomposition *of τ as:*

$$\begin{aligned}
&\boldsymbol{\phi}(\textstyle\bigvee_{i \in I} \tau_i) = \bigcup_{i \in I} \boldsymbol{\phi}(\tau_i)\\
&\boldsymbol{\phi}(\textstyle\bigwedge_{j \in P}(t_1^j \to t_2^j) \wedge \bigwedge_{k \in N} \neg(t_1^k \to t_2^k) \wedge \bigwedge_{\alpha \in P_\mathcal{V}} \alpha \wedge \bigwedge_{\alpha' \in N_\mathcal{V}} \neg\alpha')\\
&\quad = \begin{cases} \emptyset & \text{if } \bigwedge_{j \in P}(t_1^j \to t_2^j) \wedge \bigwedge_{k \in N} \neg(t_1^k \to t_2^k) \wedge \bigwedge_{\alpha \in P_\mathcal{V}} \alpha \wedge \bigwedge_{\alpha' \in N_\mathcal{V}} \neg\alpha' \simeq \mathbb{0} \\ \{(\bigvee_{j \in P'} t_1^j, \bigwedge_{j \in P \setminus P'} t_2^j) \mid P' \subsetneq P\} & \text{otherwise} \end{cases}
\end{aligned}$$

For any type t such that $t \leq \mathbb{0} \to \mathbb{1}$, the decomposition of t is defined as

$$\boldsymbol{\phi}(t) = \boldsymbol{\phi}(\mathsf{dnf}((\mathbb{0} \to \mathbb{1}) \wedge t)).$$

The set $\boldsymbol{\phi}(t)$ satisfies the following fundamental property: for every arrow type $s \to s'$, the constraint $t \leq s \to s'$ holds if and only if $s \leq \mathsf{dom}(t)$ holds and for all $(t_1, t_2) \in \boldsymbol{\phi}(t)$, either $s \leq t_1$ or $t_2 \leq s'$ hold (see Lemma 8.2.9). As a result, the minimum type

$$t \cdot s = \min\{s' \mid t \leq s \to s'\}$$

exists, and it is defined as the union of all t_2 such that $s \not\leq t_1$ and $(t_1, t_2) \in \boldsymbol{\phi}(t)$ (see Lemma 8.2.10). The type $t \cdot s$ is used to type the application of an expression of type t to an expression of type s.

Lemma 8.2.9. *Let $\leq$ be the subtyping relation induced by a well-founded (convex) model with infinite support and t a type such that $t \leq \mathbb{0} \to \mathbb{1}$. Then*

$$\forall s_1, s_2 \,.\, (t \leq s_1 \to s_2) \iff \begin{cases} s_1 \leq \mathsf{dom}(t) \\ \forall (t_1, t_2) \in \boldsymbol{\phi}(t) \,.\, (s_1 \leq t_1) \text{ or } (t_2 \leq s_2) \end{cases}$$

Proof. Since $t \leq \mathbb{0} \to \mathbb{1}$, we have

$$t \simeq \bigvee_{(P,N) \in \mathsf{dnf}(t)} ((\mathbb{0} \to \mathbb{1}) \wedge \bigwedge_{j \in P \setminus \mathcal{V}} (t_1^j \to t_2^j) \wedge \bigwedge_{k \in N \setminus \mathcal{V}} \neg(t_1^k \to t_2^k) \wedge \bigwedge_{\alpha \in P \cap \mathcal{V}} \alpha \wedge \bigwedge_{\alpha' \in N \cap \mathcal{V}} \neg\alpha')$$

If $t \simeq \mathbb{0}$, then $\mathsf{dom}(t) = \mathbb{1}$, $\boldsymbol{\phi}(t) = \emptyset$, and the result holds. If $t \simeq t_1 \vee t_2$, then $t_1 \leq \mathbb{0} \to \mathbb{1}$, $t_2 \leq \mathbb{0} \to \mathbb{1}$, $\mathsf{dom}(t) = \mathsf{dom}(t_1) \wedge \mathsf{dom}(t_2)$ and $\boldsymbol{\phi}(t) = \boldsymbol{\phi}(t_1) \cup \boldsymbol{\phi}(t_2)$. So the result follows if it also holds for t_1 and t_2. Thus, without loss of generality, we can assume that t has the following form:

$$t \simeq \bigwedge_{j \in P} (t_1^j \to t_2^j) \wedge \bigwedge_{k \in N} \neg(t_1^k \to t_2^k) \wedge \bigwedge_{\alpha \in P_\mathcal{V}} \alpha \wedge \bigwedge_{\alpha' \in N_\mathcal{V}} \neg\alpha'$$

where $P \neq \emptyset$ and $t \not\simeq \mathbb{0}$. Then $\mathsf{dom}(t) = \bigvee_{j \in P} t_1^j$ and $\boldsymbol{\phi}(t) = \{(\bigvee_{j \in P'} t_1^j, \bigwedge_{j \in P \setminus P'} t_2^j) \mid P' \subsetneq P\}$. For every pair of types s_1 and s_2, we have

$$\begin{array}{ll} & t \leq (s_1 \to s_2) \\ \iff & \bigwedge_{j \in P} (t_1^j \to t_2^j) \wedge \bigwedge_{k \in N} \neg(t_1^k \to t_2^k) \wedge \bigwedge_{\alpha \in P_\mathcal{V}} \alpha \wedge \bigwedge_{\alpha' \in N_\mathcal{V}} \neg\alpha' \leq (s_1 \to s_2) \\ \iff & \bigwedge_{j \in P} (t_1^j \to t_2^j) \wedge \bigwedge_{k \in N} \neg(t_1^k \to t_2^k) \leq (s_1 \to s_2) \quad \text{(Lemma 5.1.3)} \\ \iff & \bigwedge_{j \in P} (t_1^j \to t_2^j) \leq s_1 \to s_2 \quad (\ulcorner t \urcorner \not\simeq \mathbb{0} \text{ and Lemma 4.3.8 }) \\ \iff & \forall P' \subseteq P. \left(s_1 \leq \bigvee_{j \in P'} t_1^j \right) \vee \left(P \neq P' \wedge \bigwedge_{j \in P \setminus P'} t_2^j \leq s_2 \right) \end{array}$$

where $\ulcorner t \urcorner = \bigwedge_{j \in P} (t_1^j \to t_2^j) \wedge \bigwedge_{k \in N} \neg(t_1^k \to t_2^k)$.

□

Lemma 8.2.10. *Let t and s be two types. If $t \leq s \to \mathbb{1}$, then $t \leq s \to s'$ has a smallest solution s', which is denoted as $t \cdot s$.*

Proof. Since $t \leq s \to \mathbb{1}$, by Lemma 8.2.9, we have $s \leq \mathsf{dom}(t)$. Then the assertion $t \leq s \to s'$ is equivalent to:

$$\forall (t_1, t_2) \in \boldsymbol{\phi}(t).\ (s \leq t_1) \mathsf{or} (t_2 \leq s')$$

that is:

$$\left(\bigvee_{(t_1,t_2)\in\boldsymbol{\phi}(t) \text{ s.t. } (s\not\leq t_1)} t_2 \right) \leq s'$$

Thus the type $\bigvee_{(t_1,t_2)\in\boldsymbol{\phi}(t) \text{ s.t. } (s\not\leq t_1)} t_2$ is a lower bound for all the solutions.

By the subtyping relation on arrows it is also a solution, so it is the smallest one. To conclude, it suffices to take it as the definition for $t \cdot s$. □

We now prove some properties of the operators $\mathsf{dom}(\_)$ and "$\_ \cdot \_$".

Lemma 8.2.11. *Let t be a type such that $t \leq \mathbb{0} \to \mathbb{1}$ and t', s, s' be types. Then*

(1) *if $s' \leq s \leq \mathsf{dom}(t)$, then $t \cdot s' \leq t \cdot s$;*

(2) *if $t' \leq t$, $s \leq \mathsf{dom}(t')$ and $s \leq \mathsf{dom}(t)$, then $t' \cdot s \leq t \cdot s$.*

Proof. (1) Since $s' \leq s$, we have $s \to t \cdot s \leq s' \to t \cdot s$. By definition of $t \cdot s$, we have $t \leq s \to t \cdot s$, therefore $t \leq s' \to t \cdot s$ holds. Consequently, we have $t \cdot s' \leq t \cdot s$ by definition of $t \cdot s'$.

(2) By definition, we have $t \leq s \to t \cdot s$, which implies $t' \leq s \to t \cdot s$. Therefore, $t \cdot s$ is a solution to $t' \leq s \to s'$, hence we have $t' \cdot s \leq t \cdot s$.

□

Lemma 8.2.12. *Let t and s be two types such that $t \leq \mathbb{0} \to \mathbb{1}$ and $s \leq \mathbb{0} \to \mathbb{1}$. Then $\mathsf{dom}(t) \vee \mathsf{dom}(s) \leq \mathsf{dom}(t \wedge s)$.*

Proof. Let $t = \bigvee_{i_1 \in I_1} \tau_{i_1}$ and $s = \bigvee_{i_2 \in I_2} \tau_{i_2}$ such that $\tau_i \not\simeq \mathbb{0}$ for all $i \in I_1 \cup I_2$. Then we have $t \wedge s = \bigvee_{i_1 \in I_1, i_2 \in I_2} \tau_{i_1} \wedge \tau_{i_2}$. Let $i_1 \in I_1$ and $i_2 \in I_2$. If $\tau_{i_1} \wedge \tau_{i_2} \simeq \mathbb{0}$, then $\mathsf{dom}(\tau_{i_1} \wedge \tau_{i_2}) = \mathbb{1}$. Otherwise, $\mathsf{dom}(\tau_{i_1} \wedge \tau_{i_2}) = \mathsf{dom}(\tau_{i_1}) \vee \mathsf{dom}(\tau_{i_2})$. In both cases, we have $\mathsf{dom}(\tau_{i_1} \wedge \tau_{i_2}) \geq \mathsf{dom}(\tau_{i_1}) \vee \mathsf{dom}(\tau_{i_2})$. Therefore

$$\begin{array}{rcl} \mathsf{dom}(t \wedge s) & \simeq & \bigwedge_{i_1 \in I_1, i_2 \in I_2} \mathsf{dom}(\tau_{i_1} \wedge \tau_{i_2}) \\ & \geq & \bigwedge_{i_1 \in I_1, i_2 \in I_2} (\mathsf{dom}(\tau_{i_1}) \vee \mathsf{dom}(\tau_{i_2})) \\ & \simeq & \bigwedge_{i_1 \in I_1} (\bigwedge_{i_2 \in I_2} (\mathsf{dom}(\tau_{i_1}) \vee \mathsf{dom}(\tau_{i_2}))) \\ & \simeq & \bigwedge_{i_1 \in I_1} (\mathsf{dom}(\tau_{i_1}) \vee (\bigwedge_{i_2 \in I_2} \mathsf{dom}(\tau_{i_2}))) \\ & \geq & \bigwedge_{i_1 \in I_1} (\mathsf{dom}(\tau_{i_1})) \\ & \simeq & \mathsf{dom}(t) \end{array}$$

Similarly, $\mathsf{dom}(t \wedge s) \geq \mathsf{dom}(s)$. Therefore $\mathsf{dom}(t) \vee \mathsf{dom}(s) \leq \mathsf{dom}(t \wedge s)$. □

For example, $\mathsf{dom}((\mathtt{Int} \to \mathtt{Int}) \wedge \neg(\mathtt{Bool} \to \mathtt{Bool})) \vee \mathsf{dom}(\mathtt{Bool} \to \mathtt{Bool}) = \mathtt{Int} \vee \mathtt{Bool}$, while $\mathsf{dom}(((\mathtt{Int} \to \mathtt{Int}) \wedge \neg(\mathtt{Bool} \to \mathtt{Bool})) \wedge (\mathtt{Bool} \to \mathtt{Bool})) = \mathsf{dom}(\mathbb{0}) = \mathbb{1}$.

Lemma 8.2.13. *Let t be a type and σ be a type substitution such that $t \leq \mathbb{0} \to \mathbb{1}$. Then $\mathsf{dom}(t)\sigma \leq \mathsf{dom}(t\sigma)$.*

Proof. We put t into its disjunctive normal form $\bigvee_{i\in I} \tau_i$ such that $\tau_i \not\simeq \mathbb{0}$ for all $i \in I$. Then we have $t\sigma = \bigvee_{i\in I} \tau_i\sigma$. So $\mathsf{dom}(t\sigma) = \bigwedge_{i\in I} \mathsf{dom}(\tau_i\sigma)$. Let $i \in I$. If $\tau_i\sigma \simeq \mathbb{0}$, then $\mathsf{dom}(\tau_i\sigma) = \mathbb{1}$. Otherwise, let $\tau_i = \bigwedge_{j\in P}(t_1^j \to t_2^j) \wedge \bigwedge_{k\in N} \neg(t_1^k \to t_2^k) \wedge \bigwedge_{\alpha\in P_\mathcal{V}} \alpha \wedge \bigwedge_{\alpha'\in N_\mathcal{V}} \neg\alpha'$. Then $\mathsf{dom}(\tau_i) = \bigvee_{j\in P} t_1^j$ and $\mathsf{dom}(\tau_i\sigma) = \bigvee_{j\in P} t_1^j\sigma \vee \mathsf{dom}((\bigwedge_{\alpha\in P_\mathcal{V}} \alpha \wedge \bigwedge_{\alpha'\in N_\mathcal{V}} \neg\alpha')\sigma \wedge \mathbb{0} \to \mathbb{1})$. In both cases, we have $\mathsf{dom}(\tau_i)\sigma \leq \mathsf{dom}(\tau_i\sigma)$. Therefore, $\bigwedge_{i\in I} \mathsf{dom}(\tau_i)\sigma \leq \bigwedge_{i\in I} \mathsf{dom}(\tau_i\sigma)$, that is, $\mathsf{dom}(t)\sigma \leq \mathsf{dom}(t\sigma)$. □

For example, $\mathsf{dom}((\mathtt{Int} \to \mathtt{Int}) \wedge \neg\alpha)\{(\mathtt{Int} \to \mathtt{Int})/\alpha\} = \mathtt{Int}\{(\mathtt{Int} \to \mathtt{Int})/\alpha\} = \mathtt{Int}$, while $\mathsf{dom}(((\mathtt{Int} \to \mathtt{Int}) \wedge \neg\alpha)\{(\mathtt{Int} \to \mathtt{Int})/\alpha\}) = \mathsf{dom}((\mathtt{Int} \to \mathtt{Int}) \wedge \neg(\mathtt{Int} \to \mathtt{Int})) = \mathbb{1}$.

Lemma 8.2.14. *Let t be a type such that $t \leq \mathbb{0} \to \mathbb{1}$ and $[\sigma_k]_{k\in K}$ be a set of type substitutions. Then $\bigwedge_{k\in K} \mathsf{dom}(t)\sigma_k \leq \mathsf{dom}(\bigwedge_{k\in K} t\sigma_k)$.*

Proof.

$$\begin{array}{rcll} \bigwedge_{k\in K} \mathsf{dom}(t)\sigma_k \leq & \bigwedge_{k\in K} \mathsf{dom}(t\sigma_k) & \text{(by Lemma 8.2.13)} \\ \leq & \bigvee_{k\in K} \mathsf{dom}(t\sigma_k) & \\ \leq & \mathsf{dom}(\bigwedge_{k\in K} t\sigma_k) & \text{(by Lemma 8.2.12)} \end{array}$$

□

Lemma 8.2.15. *Let t_1, s_1, t_2 and s_2 be types such that $t_1 \cdot s_1$ and $t_2 \cdot s_2$ exists. Then $(t_1 \wedge t_2) \cdot (s_1 \wedge s_2)$ exists and $(t_1 \wedge t_2) \cdot (s_1 \wedge s_2) \leq (t_1 \cdot s_1) \wedge (t_2 \cdot s_2)$.*

Proof. According to Lemma 8.2.10, we have $s_i \leq \mathsf{dom}(t_i)$ and $t_i \leq s_i \to (t_i \cdot s_i)$. Then by Lemma 8.2.12, we get $s_1 \wedge s_2 \leq \mathsf{dom}(t_1) \wedge \mathsf{dom}(t_2) \leq \mathsf{dom}(t_1 \wedge t_2)$. Moreover, $t_1 \wedge t_2 \leq (s_1 \to (t_1 \cdot s_1)) \wedge (s_2 \to (t_2 \cdot s_2)) \leq (s_1 \wedge s_2) \to ((t_1 \cdot s_1) \wedge (t_2 \cdot s_2))$. Therefore, $(t_1 \wedge t_2) \cdot (s_1 \wedge s_2)$ exists and $(t_1 \wedge t_2) \cdot (s_1 \wedge s_2) \leq (t_1 \cdot s_1) \wedge (t_2 \cdot s_2)$. □

For example, $((\mathtt{Int} \to \mathtt{Bool}) \wedge (\mathtt{Bool} \to \mathtt{Bool})) \cdot (\mathtt{Int} \wedge \mathtt{Bool}) = \mathbb{0}$, while $((\mathtt{Int} \to \mathtt{Bool}) \cdot \mathtt{Int}) \wedge ((\mathtt{Bool} \to \mathtt{Bool}) \cdot \mathtt{Bool}) = \mathtt{Bool} \wedge \mathtt{Bool} = \mathtt{Bool}$.

Lemma 8.2.16. *Let t and s be two types such that $t \cdot s$ exists and σ be a type substitution. Then $(t\sigma) \cdot (s\sigma)$ exists and $(t\sigma) \cdot (s\sigma) \leq (t \cdot s)\sigma$.*

Proof. Because $t \cdot s$ exists, we have $s \leq \mathsf{dom}(t)$ and $t \leq s \to (t \cdot s)$. Then $s\sigma \leq \mathsf{dom}(t)\sigma$ and $t\sigma \leq s\sigma \to (t \cdot s)\sigma$. By Lemma 8.2.13, we get $\mathsf{dom}(t)\sigma \leq \mathsf{dom}(t\sigma)$. So $s\sigma \leq \mathsf{dom}(t\sigma)$. Therefore, $(t\sigma) \cdot (s\sigma)$ exists. Moreover, since $(t \cdot s)\sigma$ is a solution to $t\sigma \leq s\sigma \to s'$, by definition, we have $(t\sigma) \cdot (s\sigma) \leq (t \cdot s)\sigma$. □

For example, $(((\mathtt{Int} \to \mathtt{Int}) \wedge \neg\alpha)\sigma) \cdot (\mathtt{Int}\sigma) = \mathbb{0} \cdot \mathtt{Int} = \mathbb{0}$, while $(((\mathtt{Int} \to \mathtt{Int}) \wedge \neg\alpha) \cdot \mathtt{Int})\sigma = \mathtt{Int}\sigma = \mathtt{Int}$, where $\sigma = \{(\mathtt{Int} \to \mathtt{Int})/\alpha\}$.

Lemma 8.2.17. *Let t and s be two types and $[\sigma_k]_{k\in K}$ be a set of type substitutions such that $t \cdot s$ exists. Then $(\bigwedge_{k\in K} t\sigma_k) \cdot (\bigwedge_{k\in K} s\sigma_k)$ exists and $(\bigwedge_{k\in K} t\sigma_k) \cdot (\bigwedge_{k\in K} s\sigma_k) \leq \bigwedge_{k\in K}(t \cdot s)\sigma_k$.*

Proof. According to Lemmas 8.2.16 and 8.2.15, $(\bigwedge_{k\in K} t\sigma_k) \cdot (\bigwedge_{k\in K} s\sigma_k)$ exists. Moreover,

$$\begin{array}{rcl} \bigwedge_{k\in K}(t \cdot s)\sigma_k & \geq & \bigwedge_{k\in K}(t\sigma_k \cdot s\sigma_k) \quad \text{(Lemma 8.2.16)} \\ & \geq & (\bigwedge_{k\in K} t\sigma_k) \cdot (\bigwedge_{k\in K} s\sigma_k) \quad \text{(Lemma 8.2.15)} \end{array}$$

□

8.2.3 Syntax-directed rules

Because of subsumption, the typing rules provided in Section 7.2 are not syntax-directed and so they do not yield a type-checking algorithm directly. In simply type λ-calculus, subsumption is used to bridge gaps between the types expected by functions and the actual types of their arguments in applications [Pie02]. In our calculus, we identify four situations where the subsumption is needed, namely, the rules for projections, abstractions, applications, and type cases. To see why, we consider a typing derivation ending with each typing rule whose immediate sub-derivation ends with (*subsum*). For each case, we explain how the use of subsumption can be pushed through the typing rule under consideration, or how the rule should be modified to take subtyping into account.

First we consider the case where a typing derivation ends with (*subsum*) whose immediate sub-derivation also ends with (*subsum*). The two consecutive uses of (*subsum*) can be merged into one, because the subtyping relation is transitive.

Lemma 8.2.18. *If $\Delta \fatsemi \Gamma \vdash e : t$, then there exists a derivation for $\Delta \fatsemi \Gamma \vdash e : t$ where there are no consecutive instances of* (*subsum*)*.*

Proof. Assume that there exist two consecutive instances of (*subsum*) occurring in a derivation of $\Delta \fatsemi \Gamma \vdash e : t$, that is,

$$\dfrac{\ldots \quad \dfrac{\dfrac{\dfrac{\ldots}{\Delta' \fatsemi \Gamma' \vdash e' : s'_2} \quad s'_2 \leq s'_1}{\Delta' \fatsemi \Gamma' \vdash e' : s'_1}\,(subsum) \quad s'_1 \leq t'}{\Delta' \fatsemi \Gamma' \vdash e' : t'}\,(subsum) \quad \ldots}{\Delta \fatsemi \Gamma \vdash e : t}$$

Since $s'_2 \leq s'_1$ and $s'_1 \leq t'$, we have $s'_2 \leq t'$. So we can rewrite this derivation as follows:

$$\dfrac{\ldots \quad \dfrac{\dfrac{\ldots}{\Delta' \fatsemi \Gamma' \vdash e' : s'_2} \quad s'_2 \leq t'}{\Delta' \fatsemi \Gamma' \vdash e' : t'}\,(subsum) \quad \ldots}{\Delta \fatsemi \Gamma \vdash e : t}$$

Therefore, the result follows. □

Next, consider an instance of (*pair*) such that one of its sub-derivations ends with an instance of (*subsum*), for example, the left sub-derivation:

$$\dfrac{\dfrac{\dfrac{\dfrac{\ldots}{\Delta \fatsemi \Gamma \vdash e_1 : s_1} \quad s_1 \leq t_1}{\Delta \fatsemi \Gamma \vdash e_1 : t_1}\,(subsum) \quad \dfrac{\ldots}{\Delta \fatsemi \Gamma \vdash e_2 : t_2}}{\Delta \fatsemi \Gamma \vdash (e_1, e_2) : (t_1 \times t_2)}\,(pair)$$

As $s_1 \leq t_1$, we have $s_1 \times t_2 \leq t_1 \times t_2$. Then we can move subsumption down through the rule (*pair*), giving the following derivation:

$$\dfrac{\dfrac{\dfrac{\dots}{\Delta \fatsemi \Gamma \vdash e_1 : s_1} \quad \dfrac{\dots}{\Delta \fatsemi \Gamma \vdash e_2 : t_2}}{\Delta \fatsemi \Gamma \vdash (e_1, e_2) : (s_1 \times t_2)}\,(pair) \qquad s_1 \times t_2 \leq t_1 \times t_2}{\Delta \fatsemi \Gamma \vdash (e_1, e_2) : (t_1 \times t_2)}\,(subsum)$$

The rule (*proj*) is a little trickier than (*pair*). Consider the following derivation:

$$\dfrac{\dfrac{\dfrac{\dots}{\Delta \fatsemi \Gamma \vdash e : s} \quad s \leq t_1 \times t_2}{\Delta \fatsemi \Gamma \vdash e : (t_1 \times t_2)}\,(subsum)}{\Delta \fatsemi \Gamma \vdash \pi_i(e) : t_i}\,(proj)$$

As $s \leq t_1 \times t_2$, s is a pair type. According to the decomposition of s and Lemma 8.2.3, we can rewrite the previous derivation into the following one:

$$\dfrac{\dfrac{\dfrac{\dots}{\Delta \fatsemi \Gamma \vdash e : s} \quad s \leq \mathbb{1} \times \mathbb{1}}{\Delta \fatsemi \Gamma \vdash \pi_i(e) : \boldsymbol{\pi}_i(s)} \qquad \boldsymbol{\pi}_i(s) \leq t_i}{\Delta \fatsemi \Gamma \vdash \pi_i(e) : t_i}$$

Note that the subtyping check $s \leq \mathbb{1} \times \mathbb{1}$ ensures that s is a pair type.

Next consider an instance of (*abstr*) (where $\Delta' = \Delta \cup \mathsf{var}(\bigwedge_{i\in I, j\in J}(t_i\sigma_j \to s_i\sigma_j))$. All the sub-derivations may end with (*subsum*):

$$\dfrac{\forall i \in I, j \in J.\ \dfrac{\dfrac{\dots}{\Delta' \fatsemi \Gamma, (x : t_i\sigma_j) \vdash e@[\sigma_j] : s'_{ij}} \quad s'_{ij} \leq s_i\sigma_j}{\Delta' \fatsemi \Gamma, (x : t_i\sigma_j) \vdash e@[\sigma_j] : s_i\sigma_j}}{\Delta \fatsemi \Gamma \vdash \lambda^{\bigwedge_{i\in I} t_i \to s_i}_{[\sigma_j]_{j\in J}} x.e : \bigwedge_{i\in I, j\in J}(t_i\sigma_j \to s_i\sigma_j)}\,(abstr)$$

Without subsumption, we would assign the type $\bigwedge_{i\in I,j\in J}(t_i\sigma_j \to s'_{ij})$ to the abstraction, while we want to assign the type $\bigwedge_{i\in I,j\in J}(t_i\sigma_j \to s_i\sigma_j)$ to it because of the type annotations. Consequently, we have to keep the subtyping checks $s'_{ij} \leq s_i\sigma_j$ as side-conditions of an algorithmic typing rule for abstractions.

$$\dfrac{\forall i \in I, j \in J. \quad \Delta' \fatsemi \Gamma, (x : t_i\sigma_j) \vdash e@[\sigma_j] : s'_{ij} \qquad s'_{ij} \leq s_i\sigma_j}{\Delta \fatsemi \Gamma \vdash \lambda^{\bigwedge_{i\in I} t_i \to s_i}_{[\sigma_j]_{j\in J}} x.e : \bigwedge_{i\in I, j\in J}(t_i\sigma_j \to s_i\sigma_j)}$$

In (*appl*) case, suppose that both sub-derivations end with (*subsum*):

$$\dfrac{\dfrac{\dfrac{\dots}{\Delta \fatsemi \Gamma \vdash e_1 : t} \quad t \leq t' \to s'}{\Delta \fatsemi \Gamma \vdash e_1 : t' \to s'} \qquad \dfrac{\dfrac{\dots}{\Delta \fatsemi \Gamma \vdash e_1 : s} \quad s \leq t'}{\Delta \fatsemi \Gamma \vdash e_2 : t'}}{\Delta \fatsemi \Gamma \vdash e_1\ e_2 : s'}\,(appl)$$

Since $s \leq t'$, then by the contravariance of arrow types we have $t' \to s' \leq s \to s'$. Hence, such a derivation can be rewritten as

$$\dfrac{\dfrac{\dfrac{\dfrac{\dots}{\Delta \fatsemi \Gamma \vdash e_1 : t} \quad t \leq t' \to s'}{\Delta \fatsemi \Gamma \vdash e_1 : t' \to s'} \qquad t' \to s' \leq s \to s'}{\Delta \fatsemi \Gamma \vdash e_1 : s \to s'} \qquad \dfrac{\dots}{\Delta \fatsemi \Gamma \vdash e_1 : s}}{\Delta \fatsemi \Gamma \vdash e_1\ e_2 : s'}\,(appl)$$

Applying Lemma 8.2.18, we can merge the two adjacent instances of (*subsum*) into one:

$$\dfrac{\dfrac{\dfrac{\dots}{\Delta \fatsemi \Gamma \vdash e_1 : t} \quad t \leq s \to s'}{\Delta \fatsemi \Gamma \vdash e_1 : s \to s'} \quad \dfrac{\dots}{\Delta \fatsemi \Gamma \vdash e_1 : s}}{\Delta \fatsemi \Gamma \vdash e_1\ e_2 : s'}\ (\mathit{appl})$$

A syntax-directed typing rule for applications can then be written as follows

$$\dfrac{\Delta \fatsemi \Gamma \vdash e_1 : t \quad \Delta \fatsemi \Gamma \vdash e_2 : s \quad t \leq s \to s'}{\Delta \fatsemi \Gamma \vdash e_1 e_2 : s'}$$

where subsumption is used as a side condition to bridge the gap between the function type and the argument type.

This typing rule is not algorithmic yet, because the result type s' can be any type verifying the side condition. Using Lemma 8.2.9, we can equivalently rewrite the side condition as $t \leq \mathbb{0} \to \mathbb{1}$ and $s \leq \mathsf{dom}(t)$ without involving the result type s'. The first condition ensures that t is a function type and the second one that the argument type s can be safely applied by t. Moreover, we assign the type $t \cdot s$ to the application, which is by definition the smallest possible type for it. We obtain then the following algorithmic typing rule.

$$\dfrac{\Delta \fatsemi \Gamma \vdash e_1 : t \quad \Delta \fatsemi \Gamma \vdash e_2 : s \quad t \leq \mathbb{0} \to \mathbb{1} \quad s \leq \mathsf{dom}(t)}{\Delta \fatsemi \Gamma \vdash e_1 e_2 : t \cdot s}$$

Next, let us discuss the rule (*case*):

$$\dfrac{\Delta \fatsemi \Gamma \vdash e : t' \qquad \begin{cases} t' \not\leq \neg t & \Rightarrow \quad \Delta \fatsemi \Gamma \vdash e_1 : s \\ t' \not\leq t & \Rightarrow \quad \Delta \fatsemi \Gamma \vdash e_2 : s \end{cases}}{\Delta \fatsemi \Gamma \vdash (e \in t\,?\,e_1 : e_2) : s}\ (\mathit{case})$$

The rule covers four different situations, depending on which branches of the type-cases are checked: (*i*) no branch is type-checked, (*ii*) the first branch e_1 is type-checked, (*iii*) the second branch e_2 is type-checked, and (*iv*) both branches are type-checked. Each case produces a corresponding algorithmic rule.

In case (*i*), we have simultaneously $t' \leq t$ and $t' \leq \neg t$, which means that $t' = \mathbb{0}$. Consequently, e does not reduce to a value (otherwise, subject reduction would be violated), and neither does the whole type case. Consequently, we can assign type $\mathbb{0}$ to the whole type case.

$$\dfrac{\dfrac{\dots}{\Delta \fatsemi \Gamma \vdash e : \mathbb{0}}}{\Delta \fatsemi \Gamma \vdash (e \in t\,?\,e_1 : e_2) : \mathbb{0}}$$

Suppose we are in case (*ii*) and the sub-derivation for the first branch e_1 ends with (*subsum*):

$$\dfrac{\Delta \fatsemi \Gamma \vdash e : t' \quad t' \leq t \qquad \dfrac{\dfrac{\dots}{\Delta \fatsemi \Gamma \vdash e_1 : s_1} \quad s_1 \leq s}{\Delta \fatsemi \Gamma \vdash e_1 : s}}{\Delta \fatsemi \Gamma \vdash (e \in t\,?\,e_1 : e_2) : s}\ (\mathit{case})$$

Such a derivation can be rearranged as:

$$\dfrac{\dfrac{\Delta \fatsemi \Gamma \vdash e : t' \quad t' \leq t \quad \overset{\cdots}{\overline{\Delta \fatsemi \Gamma \vdash e_1 : s_1}}}{\Delta \fatsemi \Gamma \vdash (e{\in}t\,?\,e_1 : e_2) : s_1} \qquad s_1 \leq s}{\Delta \fatsemi \Gamma \vdash (e{\in}t\,?\,e_1 : e_2) : s}$$

Moreover, (*subsum*) might also be used at the end of the sub-derivation for e:

$$\dfrac{\dfrac{\overset{\cdots}{\overline{\Delta \fatsemi \Gamma \vdash e : t''}} \quad t'' \leq t'}{\Delta \fatsemi \Gamma \vdash e : t'} \qquad t' \leq t \quad \Delta \fatsemi \Gamma \vdash e_1 : s}{\Delta \fatsemi \Gamma \vdash (e{\in}t\,?\,e_1 : e_2) : s}\ (case)$$

From $t'' \leq t'$ and $t' \leq t$, we deduce $t'' \leq t$ by transitivity. Therefore this use of subtyping can be merged with the subtyping check of the type case rule. We then obtain the following algorithmic rule.

$$\dfrac{\overset{\cdots}{\overline{\Delta \fatsemi \Gamma \vdash e : t''}} \quad t'' \leq t \quad \Delta \fatsemi \Gamma \vdash e_1 : s}{\Delta \fatsemi \Gamma \vdash (e{\in}t\,?\,e_1 : e_2) : s}$$

We obtain a similar rule for case (*iii*), except that e_2 is type-checked instead of e_1, and t'' is tested against $\neg t$.

Finally, consider case (*iv*). We have to type-check both branches and each typing derivation may end with (*subsum*):

$$\dfrac{\Delta \fatsemi \Gamma \vdash e : t' \quad \begin{cases} t' \not\leq \neg t \quad \text{and} \quad \dfrac{\overset{\cdots}{\overline{\Delta \fatsemi \Gamma \vdash e_1 : s_1}} \quad s_1 \leq s}{\Delta \fatsemi \Gamma \vdash e_1 : s} \\ t' \not\leq t \quad \text{and} \quad \dfrac{\overset{\cdots}{\overline{\Delta \fatsemi \Gamma \vdash e_2 : s_2}} \quad s_2 \leq s}{\Delta \fatsemi \Gamma \vdash e_2 : s} \end{cases}}{\Delta \fatsemi \Gamma \vdash (e{\in}t\,?\,e_1 : e_2) : s}\ (case)$$

Subsumption is used there just to unify s_1 and s_2 into a common type s, which is used to type the whole type case. Such a common type can also be obtained by taking the least upper-bound of s_1 and s_2, i.e., $s_1 \vee s_2$. Because $s_1 \leq s$ and $s_2 \leq s$, we have $s_1 \vee s_2 \leq s$, and we can rewrite the derivation as follows:

$$\dfrac{\dfrac{\Delta \fatsemi \Gamma \vdash e : t' \quad \begin{cases} t' \not\leq \neg t \quad \text{and} \quad \overset{\cdots}{\overline{\Delta \fatsemi \Gamma \vdash e_1 : s_1}} \\ t' \not\leq t \quad \text{and} \quad \overset{\cdots}{\overline{\Delta \fatsemi \Gamma \vdash e_2 : s_2}} \end{cases}}{\Delta \fatsemi \Gamma \vdash (e{\in}t\,?\,e_1 : e_2) : s_1 \vee s_2}\ (case) \qquad s_1 \vee s_2 \leq s}{\Delta \fatsemi \Gamma \vdash (e{\in}t\,?\,e_1 : e_2) : s}$$

Suppose now that the sub-derivation for e ends with (*subsum*):

$$\dfrac{\dfrac{\overset{\cdots}{\overline{\Delta \fatsemi \Gamma \vdash e : t''}} \quad t'' \leq t'}{\Delta \fatsemi \Gamma \vdash e : t'} \quad \begin{cases} t' \not\leq \neg t \quad \text{and} \quad \Delta \fatsemi \Gamma \vdash e_1 : s_1 \\ t' \not\leq t \quad \text{and} \quad \Delta \fatsemi \Gamma \vdash e_2 : s_2 \end{cases}}{\Delta \fatsemi \Gamma \vdash (e{\in}t\,?\,e_1 : e_2) : s_1 \vee s_2}\ (case)$$

The relations $t'' \leq t'$, $t' \not\leq \neg t$ do not necessarily imply $t'' \not\leq \neg t$, and $t'' \leq t'$, $t' \not\leq t$ do not necessarily imply $t'' \not\leq \neg t$. Therefore, by using the type t'' instead of t' for e, we may type-check less branches. If so, then we would be in one of the cases $(i) - (iii)$, and the result type (*i.e.*, a type among $\mathbb{0}$, s_1 or s_2) for the whole type case would be smaller than $s_1 \vee s_2$. It would then be possible to type the type case with $s_1 \vee s_2$ by subsumption. Otherwise, we type-check as many branches with t'' as with t', and we can modify the rule into

$$\frac{\dfrac{\cdots}{\Delta \fatsemi \Gamma \vdash e : t''} \quad \begin{cases} t'' \not\leq \neg t & \text{and} \quad \Delta \fatsemi \Gamma \vdash e_1 : s_1 \\ t'' \not\leq t & \text{and} \quad \Delta \fatsemi \Gamma \vdash e_2 : s_2 \end{cases}}{\Delta \fatsemi \Gamma \vdash (e \in t \,?\, e_1 : e_2) : s_1 \vee s_2}$$

Finally, consider the case where the last rule in a derivation is (*instinter*) and all its sub-derivations end with (*subsum*):

$$\frac{\dfrac{\dfrac{\cdots}{\Delta \fatsemi \Gamma \vdash e : s} \quad s \leq t}{\Delta \fatsemi \Gamma \vdash e : t}\,(subsum) \quad \forall j \in J.\ \sigma_j \not\sharp \Delta}{\Delta \fatsemi \Gamma \vdash e[\sigma_j]_{j \in J} : \bigwedge_{j \in J} t\sigma_j}\,(instinter)$$

Since $s \leq t$, we have $\bigwedge_{j \in J} s\sigma_j \leq \bigwedge_{j \in J} t\sigma_j$. So such a derivation can be rewritten into

$$\frac{\dfrac{\dfrac{\cdots}{\Delta \fatsemi \Gamma \vdash e : s} \quad \forall j \in J.\ \sigma_j \not\sharp \Delta}{\Delta \fatsemi \Gamma \vdash e[\sigma_j]_{j \in J} : \bigwedge_{j \in J} s\sigma_j}\,(instinter) \quad \bigwedge_{j \in J} s\sigma_j \leq \bigwedge_{j \in J} t\sigma_j}{\Delta \fatsemi \Gamma \vdash e[\sigma_j]_{j \in J} : \bigwedge_{j \in J} t\sigma_j}\,(subsum)$$

In conclusion, by applying the aforementioned transformations repeatedly, we can rewrite an arbitrary typing derivation into a special form where subsumption are used at the end of sub-derivations of projections, abstractions or applications, in the conditions of type cases and at the very end of the whole derivation. Thus, this transformations yields a set of syntax-directed typing rules, which is given in Figure 8.1 and forms a type-checking algorithm directly. Let $\Delta \fatsemi \Gamma \vdash_{\mathcal{A}} e : t$ denote the typing judgments derivable by the set of syntax-directed typing rules.

We now prove that the syntax-directed typing rules are sound and complete with respect to the original typing rules.

Theorem 8.2.19 (Soundness). *Let e be an expression. If $\Gamma \vdash_{\mathcal{A}} e : t$, then $\Gamma \vdash e : t$.*

Proof. By induction on the typing derivation of $\Delta \fatsemi \Gamma \vdash_{\mathcal{A}} e : t$. We proceed by a case analysis on the last rule used in the derivation.

(Alg-const): straightforward.
(Alg-var): straightforward.
(Alg-pair): consider the derivation

$$\frac{\dfrac{\cdots}{\Delta \fatsemi \Gamma \vdash_{\mathcal{A}} e_1 : t_1} \quad \dfrac{\cdots}{\Delta \fatsemi \Gamma \vdash_{\mathcal{A}} e_2 : t_2}}{\Delta \fatsemi \Gamma \vdash_{\mathcal{A}} (e_1, e_2) : t_1 \times t_2}$$

Applying the induction hypothesis twice, we get $\Delta \fatsemi \Gamma \vdash e_i : t_i$. Then by applying the rule (*pair*), we have $\Delta \fatsemi \Gamma \vdash (e_1, e_2) : t_1 \times t_2$.

$$\frac{}{\Delta \fatsemi \Gamma \vdash_{\mathcal{A}} c : b_c} \text{ (Alg-const)} \qquad \frac{}{\Delta \fatsemi \Gamma \vdash_{\mathcal{A}} x : \Gamma(x)} \text{ (Alg-var)}$$

$$\frac{\Delta \fatsemi \Gamma \vdash_{\mathcal{A}} e_1 : t_1 \qquad \Delta \fatsemi \Gamma \vdash_{\mathcal{A}} e_2 : t_2}{\Delta \fatsemi \Gamma \vdash_{\mathcal{A}} (e_1, e_2) : t_1 \times t_2} \text{ (Alg-pair)}$$

$$\frac{\Delta \fatsemi \Gamma \vdash_{\mathcal{A}} e : t \qquad t \leq \mathbb{1} \times \mathbb{1}}{\Delta \fatsemi \Gamma \vdash_{\mathcal{A}} \pi_i(e) : \boldsymbol{\pi}_i(t)} \text{ (Alg-proj)}$$

$$\frac{\Delta \fatsemi \Gamma \vdash_{\mathcal{A}} e_1 : t \qquad \Delta \fatsemi \Gamma \vdash_{\mathcal{A}} e_2 : s \qquad t \leq \mathbb{0} \to \mathbb{1} \qquad s \leq \mathsf{dom}(t)}{\Delta \fatsemi \Gamma \vdash_{\mathcal{A}} e_1 e_2 : t \cdot s} \text{ (Alg-appl)}$$

$$\frac{\begin{array}{l}\Delta' = \Delta \cup \mathsf{var}(\bigwedge\limits_{i \in I, j \in J} (t_i\sigma_j \to s_i\sigma_j)) \\ \forall i \in I, j \in J.\ \Delta' \fatsemi \Gamma, (x : t_i\sigma_j) \vdash_{\mathcal{A}} e@[\sigma_j] : s'_{ij} \qquad s'_{ij} \leq s_i\sigma_j\end{array}}{\Delta \fatsemi \Gamma \vdash_{\mathcal{A}} \lambda^{\bigwedge_{i \in I} t_i \to s_i}_{[\sigma_j]_{j \in J}} x.e : \bigwedge\limits_{i \in I, j \in J} (t_i\sigma_j \to s_i\sigma_j)} \text{ (Alg-abstr)}$$

$$\frac{\Delta \fatsemi \Gamma \vdash_{\mathcal{A}} e : \mathbb{0}}{\Delta \fatsemi \Gamma \vdash_{\mathcal{A}} (e \in t\,?\,e_1 : e_2) : \mathbb{0}} \text{ (Alg-case-none)}$$

$$\frac{\Delta \fatsemi \Gamma \vdash_{\mathcal{A}} e : t' \qquad t' \leq t \qquad t' \not\leq \neg t \qquad \Delta \fatsemi \Gamma \vdash_{\mathcal{A}} e_1 : s_1}{\Delta \fatsemi \Gamma \vdash_{\mathcal{A}} (e \in t\,?\,e_1 : e_2) : s_1} \text{ (Alg-case-fst)}$$

$$\frac{\Delta \fatsemi \Gamma \vdash_{\mathcal{A}} e : t' \qquad t' \leq \neg t \qquad t' \not\leq t \qquad \Delta \fatsemi \Gamma \vdash_{\mathcal{A}} e_2 : s_2}{\Delta \fatsemi \Gamma \vdash_{\mathcal{A}} (e \in t\,?\,e_1 : e_2) : s_2} \text{ (Alg-case-snd)}$$

$$\frac{\Delta \fatsemi \Gamma \vdash_{\mathcal{A}} e : t' \qquad \left\{\begin{array}{lll} t' \not\leq \neg t & \text{and} & \Delta \fatsemi \Gamma \vdash_{\mathcal{A}} e_1 : s_1 \\ t' \not\leq t & \text{and} & \Delta \fatsemi \Gamma \vdash_{\mathcal{A}} e_2 : s_2 \end{array}\right.}{\Delta \fatsemi \Gamma \vdash_{\mathcal{A}} (e \in t\,?\,e_1 : e_2) : s_1 \vee s_2} \text{ (Alg-case-both)}$$

$$\frac{\Delta \fatsemi \Gamma \vdash_{\mathcal{A}} e : t \qquad \forall j \in J.\ \sigma_j \sharp \Delta \qquad |J| > 0}{\Delta \fatsemi \Gamma \vdash_{\mathcal{A}} e[\sigma_j]_{j \in J} : \bigwedge\limits_{j \in J} t\sigma_j} \text{ (Alg-inst)}$$

Figure 8.1: Syntax-directed typing rules

(Alg-proj)**:** consider the derivation

$$\frac{\dfrac{\ldots}{\Delta \fatsemi \Gamma \vdash_{\mathcal{A}} e : t} \qquad t \leq \mathbb{1} \times \mathbb{1}}{\Delta \fatsemi \Gamma \vdash_{\mathcal{A}} \pi_i(e) : \boldsymbol{\pi}_i(t)}$$

By induction, we have $\Delta \fatsemi \Gamma \vdash e : t$. According to Lemma 8.2.3, we have $t \leq (\boldsymbol{\pi}_1(t) \times \boldsymbol{\pi}_2(t))$. Then by (*subsum*), we get $\Delta \fatsemi \Gamma \vdash e : (\boldsymbol{\pi}_1(t) \times \boldsymbol{\pi}_2(t))$. Finally, the rule (*proj*) gives us $\Delta \fatsemi \Gamma \vdash \pi_i(e) : \boldsymbol{\pi}_i(t)$.

(ALG-APPL): consider the derivation

$$\frac{\dots \quad \dots \quad t \leq \mathbb{0} \to \mathbb{1} \quad s \leq \mathsf{dom}(t)}{\Delta \fatsemi \Gamma \vdash_{\mathcal{A}} e_1 e_2 : t \cdot s}$$

By induction, we have $\Delta \fatsemi \Gamma \vdash e_1 : t$ and $\Delta \fatsemi \Gamma \vdash e_2 : s$. According to Lemma 8.2.10, we have

$$t \cdot s = \min\{s' \mid t \leq s \to s'\}$$

Note that the conditions $t \leq \mathbb{0} \to \mathbb{1}$ and $s \leq \mathsf{dom}(t)$ ensure that such a type exists. It is clear $t \leq s \to (t \cdot s)$. Then by (*subsum*), we get $\Delta \fatsemi \Gamma \vdash e_1 : s \to (t \cdot s)$. Finally, the rule (*appl*) gives us $\Delta \fatsemi \Gamma \vdash e_1 e_2 : t \cdot s$.

(ALG-ABSTR): consider the derivation

$$\frac{\forall i \in I, j \in J.\ \dfrac{\dots}{\Delta' \fatsemi \Gamma, (x : t_i\sigma_j) \vdash_{\mathcal{A}} e@[\sigma_j] : s'_{ij}} \qquad s'_{ij} \leq s_i\sigma_j}{\Delta \fatsemi \Gamma \vdash_{\mathcal{A}} \lambda^{\bigwedge_{i\in I} t_i \to s_i}_{[\sigma_j]_{j\in J}} x.e : \bigwedge_{i\in I, j\in J}(t_i\sigma_j \to s_i\sigma_j)}$$

with $\Delta' = \Delta \cup \mathsf{var}(\bigwedge_{i\in I, j\in J}(t_i\sigma_j \to s_i\sigma_j))$. By induction, for all $i \in I$ and $j \in J$, we have $\Delta' \fatsemi \Gamma, (x : t_i\sigma_j) \vdash e@[\sigma_j] : s'_{ij}$. Since $s'_{ij} \leq s_i\sigma_j$, by (*subsum*), we get $\Delta' \fatsemi \Gamma, (x : t_i\sigma_j) \vdash e@[\sigma_j] : s_i\sigma_j$. Finally, the rule (*abstr*) gives us $\Delta \fatsemi \Gamma \vdash \lambda^{\bigwedge_{i\in I} t_i \to s_i}_{[\sigma_j]_{j\in J}} x.e : \bigwedge_{i\in I, j\in J}(t_i\sigma_j \to s_i\sigma_j)$.

(ALG-CASE-NONE): consider the derivation

$$\frac{\dfrac{\dots}{\Delta \fatsemi \Gamma \vdash_{\mathcal{A}} e : \mathbb{0}}}{\Delta \fatsemi \Gamma \vdash_{\mathcal{A}} (e \in t\,?\,e_1 : e_2) : \mathbb{0}}$$

By induction, we have $\Delta \fatsemi \Gamma \vdash e : \mathbb{0}$. No branch is type-checked by the rule (*case*), so any type can be assigned to the type case expression, and in particular we have $\Delta \fatsemi \Gamma \vdash (e \in t\,?\,e_1 : e_2) : \mathbb{0}$

(ALG-CASE-FST): consider the derivation

$$\frac{\dfrac{\dots}{\Delta \fatsemi \Gamma \vdash_{\mathcal{A}} e : t'} \quad t' \leq t \quad \dfrac{\dots}{\Delta \fatsemi \Gamma \vdash_{\mathcal{A}} e_1 : s_1}}{\Delta \fatsemi \Gamma \vdash_{\mathcal{A}} (e \in t\,?\,e_1 : e_2) : s_1}$$

By induction, we have $\Delta \fatsemi \Gamma \vdash e : t'$ and $\Delta \fatsemi \Gamma \vdash e_1 : s_1$. As $t' \leq t$, then we only need to type-check the first branch. Therefore, by the rule (*case*), we have $\Delta \fatsemi \Gamma \vdash (e \in t\,?\,e_1 : e_2) : s_1$.

(ALG-CASE-SND): similar the case of (ALG-CASE-FST).

(ALG-CASE-BOTH): consider the derivation

$$\frac{\dfrac{\dots}{\Delta \fatsemi \Gamma \vdash_{\mathcal{A}} e : t'} \quad \begin{cases} t' \not\leq \neg t \quad \text{and} \quad \dfrac{\dots}{\Delta \fatsemi \Gamma \vdash_{\mathcal{A}} e_1 : s_1} \\ t' \not\leq t \quad \text{and} \quad \dfrac{\dots}{\Delta \fatsemi \Gamma \vdash_{\mathcal{A}} e_2 : s_2} \end{cases}}{\Delta \fatsemi \Gamma \vdash_{\mathcal{A}} (e \in t\,?\,e_1 : e_2) : s_1 \vee s_2}$$

By induction, we have $\Delta \fatsemi \Gamma \vdash e : t'$, $\Delta \fatsemi \Gamma \vdash e_1 : s_1$ and $\Delta \fatsemi \Gamma \vdash e_2 : s_2$. It is clear that $s_1 \leq s_1 \vee s_2$ and $s_2 \leq s_1 \vee s_2$. Then by (*subsum*), we get $\Delta \fatsemi \Gamma \vdash e_1 : s_1 \vee s_2$ and $\Delta \fatsemi \Gamma \vdash e_2 : s_1 \vee s_2$. Moreover, as $t' \not\leq \neg t$ and $t' \not\leq t$, we have to type-check both branches. Finally, by the rule (*case*), we get $\Delta \fatsemi \Gamma \vdash (e \in t \,?\, e_1 : e_2) : s_1 \vee s_2$.

(ALG-INST): consider the derivation

$$\frac{\dfrac{\cdots}{\Delta \fatsemi \Gamma \vdash_{\mathcal{A}} e : t} \qquad \forall j \in J.\ \sigma_j \sharp \Delta}{\Delta \fatsemi \Gamma \vdash_{\mathcal{A}} e[\sigma_j]_{j \in J} : \bigwedge_{j \in J} t\sigma_j}$$

By induction, we have $\Delta \fatsemi \Gamma \vdash e : t$. As $\forall j \in J.\ \sigma_j \sharp \Delta$, by (*instinter*), we get $\Delta \fatsemi \Gamma \vdash e[\sigma_j]_{j \in J} : \bigwedge_{j \in J} t\sigma_j$.

□

The soundness theorem states that every typing statement that can be derived from the syntax-directed rules also follows from the original rules. While the completeness theorem that follows states that the syntax-directed type system can deduce for an expression a type at least as good as the one deduced for that expression from the original type system.

Theorem 8.2.20 (Completeness). *Let $\leq$ be a subtyping relation induced by a well-founded (convex) model with infinite support and e an expression. If $\Delta \fatsemi \Gamma \vdash e : t$, then there exists a type s such that $\Delta \fatsemi \Gamma \vdash_{\mathcal{A}} e : s$ and $s \leq t$.*

Proof. By induction on the typing derivation of $\Delta \fatsemi \Gamma \vdash e : t$. We proceed by case analysis on the last rule used in the derivation.

(*const*): straightforward (take s as b_c).
(*var*): straightforward (take s as $\Gamma(x)$).
(*pair*): consider the derivation

$$\frac{\dfrac{\cdots}{\Delta \fatsemi \Gamma \vdash e_1 : t_1} \qquad \dfrac{\cdots}{\Delta \fatsemi \Gamma \vdash e_2 : t_2}}{\Delta \fatsemi \Gamma \vdash (e_1, e_2) : t_1 \times t_2}\ (pair)$$

Applying the induction hypothesis twice, we have $\Delta \fatsemi \Gamma \vdash_{\mathcal{A}} e_i : s_i$ where $s_i \leq t_i$. Then the rule (ALG-PAIR) gives us $\Delta \fatsemi \Gamma \vdash_{\mathcal{A}} (e_1, e_2) : s_1 \times s_2$. Since $s_i \leq t_i$, we deduce $(s_1 \times s_2) \leq (t_1 \times t_2)$.

(*proj*): consider the derivation

$$\frac{\dfrac{\cdots}{\Delta \fatsemi \Gamma \vdash e : (t_1 \times t_2)}}{\Delta \fatsemi \Gamma \vdash \pi_i(e) : t_i}\ (proj)$$

By induction, there exists s such that $\Delta \fatsemi \Gamma \vdash_{\mathcal{A}} e : s$ and $s \leq (t_1 \times t_2)$. Clearly we have $s \leq \mathbb{1} \times \mathbb{1}$. Applying (ALG-PROJ), we have $\Delta \fatsemi \Gamma \vdash_{\mathcal{A}} \pi_i(e) : \boldsymbol{\pi}_i(s)$. Moreover, as $s \leq (t_1 \times t_2)$, according to Lemma 8.2.3, we have $\boldsymbol{\pi}_i(s) \leq t_i$. Therefore, the result follows.

(*appl*)**:** consider the derivation

$$\frac{\dfrac{\cdots}{\Delta \fatsemi \Gamma \vdash e_1 : t_1 \to t_2} \quad \dfrac{\cdots}{\Delta \fatsemi \Gamma \vdash e_2 : t_1}}{\Delta \fatsemi \Gamma \vdash e_1 e_2 : t_2}\ (appl)$$

Applying the induction hypothesis twice, we have $\Delta \fatsemi \Gamma \vdash_{\mathcal{A}} e_1 : t$ and $\Delta \fatsemi \Gamma \vdash_{\mathcal{A}} e_2 : s$ where $t \leq t_1 \to t_2$ and $s \leq t_1$. Clearly we have $t \leq \mathbb{0} \to \mathbb{1}$ and $t \leq s \to t_2$ (by contravariance of arrows). From Lemma 8.2.9, we get $s \leq \mathsf{dom}(t)$. So, by applying the rule (Alg-appl), we have $\Delta \fatsemi \Gamma \vdash_{\mathcal{A}} e_1 e_2 : t \cdot s$. Moreover, it is clear that t_2 is a solution for $t \leq s \to s'$. Consequently, it is a super type of $t \cdot s$, that is $t \cdot s \leq t_2$.

(*abstr*)**:** consider the derivation

$$\frac{\forall i \in I, j \in J.\ \dfrac{\cdots}{\Delta' \fatsemi \Gamma, (x : t_i\sigma_j) \vdash e@[\sigma_j] : s_i\sigma_j}}{\Delta \fatsemi \Gamma \vdash \lambda^{\wedge_{i\in I} t_i \to s_i}_{[\sigma_j]_{j\in J}} x.e : \bigwedge_{i\in I, j\in J}(t_i\sigma_j \to s_i\sigma_j)}\ (abstr)$$

where $\Delta' = \Delta \cup \mathsf{var}(\bigwedge_{i\in I, j\in J}(t_i\sigma_j \to s_i\sigma_j))$. By induction, for all $i \in I$ and $j \in J$, there exists s'_{ij} such that $\Delta' \fatsemi \Gamma, (x : t_i\sigma_j) \vdash_{\mathcal{A}} e@[\sigma_j] : s'_{ij}$ and $s'_{ij} \leq s_i\sigma_j$. Then the rule (Alg-abstr) gives us $\Delta \fatsemi \Gamma \vdash_{\mathcal{A}} \lambda^{\wedge_{i\in I} t_i \to s_i}_{[\sigma_j]_{j\in J}} x.e : \bigwedge_{i\in I, j\in J}(t_i\sigma_j \to s_i\sigma_j)$

(*case*)**:** consider the derivation

$$\frac{\dfrac{\cdots}{\Delta \fatsemi \Gamma \vdash e : t'} \quad \begin{cases} t' \not\leq \neg t & \Rightarrow \ \dfrac{\cdots}{\Delta \fatsemi \Gamma \vdash e_1 : s} \\ t' \not\leq t & \Rightarrow \ \dfrac{\cdots}{\Delta \fatsemi \Gamma \vdash e_2 : s} \end{cases}}{\Delta \fatsemi \Gamma \vdash (e \in t\,?\,e_1 : e_2) : s}\ (case)$$

By induction hypothesis on $\Delta \fatsemi \Gamma \vdash e : t'$, there exists a type t'' such that $\Delta \fatsemi \Gamma \vdash_{\mathcal{A}} e : t''$ and $t'' \leq t'$. If $t'' \simeq \mathbb{0}$, by (Alg-case-none), we have $\Delta \fatsemi \Gamma \vdash_{\mathcal{A}} (e \in t\,?\,e_1 : e_2) : \mathbb{0}$. The result follows straightforwardly. In what follows, we assume that $t'' \not\simeq \mathbb{0}$.

Assume that $t'' \leq t$. Because $t'' \leq t'$, we have $t' \not\leq \neg t$ (otherwise, $t'' \simeq \mathbb{0}$). Therefore the first branch is type-checked, and by induction, there exists a type s_1 such that $\Delta \fatsemi \Gamma \vdash_{\mathcal{A}} e_1 : s_1$ and $s_1 \leq s$. Then the rule (Alg-case-fst) gives us $\Delta \fatsemi \Gamma \vdash_{\mathcal{A}} (e \in t\,?\,e_1 : e_2) : s_1$.

Otherwise, $t'' \not\leq t$. In this case, we have $t' \not\leq t$ (otherwise, $t'' \leq t$). Then the second branch is type-checked. By induction, there exists a type s_2 such that $\Delta \fatsemi \Gamma \vdash_{\mathcal{A}} e_2 : s_2$ and $s_2 \leq s$. If $t'' \leq \neg t$, then by the rule (Alg-case-snd), we have $\Delta \fatsemi \Gamma \vdash_{\mathcal{A}} (e \in t\,?\,e_1 : e_2) : s_2$. The result follows. Otherwise, we also have $t'' \not\leq \neg t$. Then we also have $t' \not\leq \neg t$ (otherwise, $t'' \leq \neg t$). So the first branch should be type-checked as well. By induction, we have $\Delta \fatsemi \Gamma \vdash_{\mathcal{A}} e_1 : s_1$ where $s_1 \leq s$. By applying (Alg-case-both), we get $\Delta \fatsemi \Gamma \vdash_{\mathcal{A}} (e \in t\,?\,e_1 : e_2) : s_1 \vee s_2$. Since $s_1 \leq s$ and $s_2 \leq s$, we deduce that $s_1 \vee s_2 \leq s$. The result follows as well.

(*instinter*)**:** consider the derivation

$$\frac{\dfrac{\cdots}{\Delta \fatsemi \Gamma \vdash e : t} \quad \forall j \in J.\ \sigma_j \sharp \Delta}{\Delta \fatsemi \Gamma \vdash e[\sigma_j]_{j\in J} : \bigwedge_{j\in J} t\sigma_j}\ (instinter)$$

By induction, there exists a type s such that $\Delta \fatsemi \Gamma \vdash_{\mathcal{A}} e : s$ and $s \leq t$. Then the rule (ALG-INST) gives us that $\Delta \fatsemi \Gamma \vdash_{\mathcal{A}} e[\sigma_j]_{j\in J} : \bigwedge_{j\in J} s\sigma_j$. Since $s \leq t$, we have $\bigwedge_{j\in J} s\sigma_j \leq \bigwedge_{j\in J} t\sigma_j$. Therefore, the result follows.

□

Corollary 8.2.21 (Minimum typing). *Let e be an expression. If $\Delta \fatsemi \Gamma \vdash_{\mathcal{A}} e : t$, then $t = \min\{s \mid \Delta \fatsemi \Gamma \vdash e : s\}$.*

Proof. Consequence of Theorems 8.2.19 and 8.2.20. □

To prove the termination of the type-checking algorithm, we define the *size* of an expression e as follows.

Definition 8.2.22. *Let e be an expression. We define the* size *of e as:*

$$\begin{aligned}
size(c) &= 1\\
size(x) &= 1\\
size((e_1, e_2)) &= size(e_1) + size(e_2) + 1\\
size(\pi_i(e)) &= size(e) + 1\\
size(e_1 e_2) &= size(e_1) + size(e_2) + 1\\
size(\lambda^{\wedge_{i\in I} t_i \to s_i}_{[\sigma_j]_{j\in J}} x.e) &= size(e) + 1\\
size(e \in t\,?\,e_1 : e_2) &= size(e) + size(e_1) + size(e_2) + 1\\
size(e[\sigma_j]_{j\in J}) &= size(e) + 1
\end{aligned}$$

The relabeling does not enlarge the size of the expression.

Lemma 8.2.23. *Let e be an expression and $[\sigma_j]_{j\in J}$ a set of type substitutions. Then*

$$size(e@[\sigma_j]_{j\in J}) \leq size(e).$$

Proof. By induction on the structure of e. □

Theorem 8.2.24 (Termination). *Let e be an expression. Then the type-checking of e terminates.*

Proof. By induction on the sizes of the expressions to be checked.

(ALG-CONST): it terminates immediately.
(ALG-VAR): it terminates immediately.
(ALG-PAIR): $\text{size}(e_1) + \text{size}(e_2) < \text{size}((e_1, e_2))$.
(ALG-PROJ): $\text{size}(e') < \text{size}(\pi_1(e'))$.
(ALG-APPL): $\text{size}(e_1) + \text{size}(e_2) < \text{size}(e_1 e_2)$.
(ALG-ABSTR): by Lemma 8.2.23, we have $\text{size}(e'@[\sigma_j]) \leq \text{size}(e')$. Then we get $\text{size}(e'@[\sigma_j]) < \text{size}(\lambda^{\wedge_{i\in I} t_i \to s_i}_{[\sigma_j]_{j\in J}} x.e')$.
(ALG-CASE): $\text{size}(e') + \text{size}(e_1) + \text{size}(e_2) < \text{size}(e' \in t\,?\,e_1 : e_2)$.
(ALG-INST): $\text{size}(e') < \text{size}(e'[\sigma_j]_{j\in J})$.

□

Chapter 9

Inference of Type-Substitutions

We want sets of type-substitutions to be inferred by the system, not written by the programmer. To this end, we define a calculus without type substitutions (called *implicitly-typed*, in contrast to the calculus of Section 7.1, which we henceforth call *explicitly-typed*), for which we define a type-substitutions inference system. There will be a single exception: we will not try to insert type-substitutions into decorations, since we suppose that all λ-abstractions initially have empty decorations. The reasons for this restriction is that we want to infer that an expression such as $\lambda^{\alpha\to\alpha}x.3$ is ill-typed and if we allowed to infer decorations, then the expression could be typed by inserting a decoration as in $\lambda^{\alpha\to\alpha}_{\{\mathtt{Int}/\alpha\}}x.3$. Generally, if the programmer specifies some interface for a function it seems reasonable to think that he wants the system to check whether the function conforms the interface rather than knowing whether there exists a set of type substitutions that makes it conforming. We therefore look for completeness of the type-substitutions inference system with respect to the expressions written according to the following grammar.

$$e \;::=\; c \mid x \mid (e,e) \mid \pi_i(e) \mid e\ e \mid \lambda^{\wedge_{i\in I}t_i\to s_i}x.e \mid e{\in}t\,?\,e:e \mid e[\sigma_j]_{j\in J}$$

We write $\mathscr{E}_0$ for the set of such expressions. The implicitly-typed calculus defined in this chapter corresponds to the type-substitution erasures of the expressions of $\mathscr{E}_0$. These are the terms generated by the grammar above without using the last production, that is, without the application of sets of type-substitutions. We then define the type-substitutions inference system by determining where the rule (Alg-inst) have to be used in the typing derivations of explicitly-typed expressions. Finally, we propose an incomplete but more tractable restriction of the type-substitutions inference system, which, we believe, is powerful enough to be used in practice.

9.1 Implicitly-typed calculus

Definition 9.1.1. *An* implicitly-typed expression *a is an expression without any type substitutions. It is inductively generated by the following grammar:*

$$a \;::=\; c \mid x \mid (a,a) \mid \pi_i(a) \mid a\ a \mid \lambda^{\wedge_{i\in I}t_i\to s_i}x.a \mid a{\in}t\,?\,a:a$$

where t_i, s_i range over types and $t \in \mathcal{T}_0$ is a ground type. We write $\mathscr{E}_A$ to denote the set of all implicitly-typed expressions.

Clearly, $\mathscr{E}_A$ is a proper subset of $\mathscr{E}_0$.

The erasure of explicitly-typed expressions to implicitly-typed expressions is defined as follows:

Definition 9.1.2. *The* erasure *is the mapping from $\mathscr{E}_0$ to $\mathscr{E}_A$ defined as*

$$\begin{aligned}
\mathit{erase}(c) &= c \\
\mathit{erase}(x) &= x \\
\mathit{erase}((e_1, e_2)) &= (\mathit{erase}(e_1), \mathit{erase}(e_2)) \\
\mathit{erase}(\pi_i(e)) &= \pi_i(\mathit{erase}(e)) \\
\mathit{erase}(\lambda^{\wedge_{i\in I} t_i \to s_i} x.e) &= \lambda^{\wedge_{i\in I} t_i \to s_i} x.\mathit{erase}(e) \\
\mathit{erase}(e_1 e_2) &= \mathit{erase}(e_1)\mathit{erase}(e_2) \\
\mathit{erase}(e{\in}t\,?\,e_1 : e_2) &= \mathit{erase}(e){\in}t\,?\,\mathit{erase}(e_1) : \mathit{erase}(e_2) \\
\mathit{erase}(e[\sigma_j]_{j\in J}) &= \mathit{erase}(e)
\end{aligned}$$

Prior to introducing the type inference rules, we define a preorder on types, which is similar to the type variable instantiation in ML but with respect to *a set* of type substitutions.

Definition 9.1.3. *Let s and t be two types and Δ a set of type variables. We define the following relations:*

$$\begin{aligned}
[\sigma_i]_{i\in I} \Vdash s \sqsubseteq_\Delta t \quad &\stackrel{\text{def}}{\Longleftrightarrow} \quad \bigwedge_{i\in I} s\sigma_i \leq t \text{ and } \forall i{\in}I.\ \sigma_i \sharp \Delta \\
s \sqsubseteq_\Delta t \quad &\stackrel{\text{def}}{\Longleftrightarrow} \quad \exists [\sigma_i]_{i\in I} \text{ such that } [\sigma_i]_{i\in I} \Vdash s \sqsubseteq_\Delta t
\end{aligned}$$

We write $s \not\sqsubseteq_\Delta t$ if it does not exist a set of type substitutions $[\sigma_i]_{i\in I}$ such that $[\sigma_i]_{i\in I} \Vdash s \sqsubseteq_\Delta t$. We now prove some properties of the preorder $\sqsubseteq_\Delta$.

Lemma 9.1.4. *Let t_1 and t_2 be two types and Δ a set of type variables. If $t_1 \sqsubseteq_\Delta s_1$ and $t_2 \sqsubseteq_\Delta s_2$, then $(t_1 \wedge t_2) \sqsubseteq_\Delta (s_1 \wedge s_2)$ and $(t_1 \times t_2) \sqsubseteq_\Delta (s_1 \times s_2)$.*

Proof. Let $[\sigma_{i_1}]_{i_1\in I_1} \Vdash t_1 \sqsubseteq_\Delta s_1$ and $[\sigma_{i_2}]_{i_2\in I_2} \Vdash t_2 \sqsubseteq_\Delta s_2$. Then

$$\begin{aligned}
\textstyle\bigwedge_{i\in I_1\cup I_2}(t_1 \wedge t_2)\sigma_i &\simeq (\textstyle\bigwedge_{i\in I_1\cup I_2} t_1\sigma_i) \wedge (\textstyle\bigwedge_{i\in I_1\cup I_2} t_2\sigma_i) \\
&\leq (\textstyle\bigwedge_{i_1\in I_1} t_1\sigma_{i_1}) \wedge (\textstyle\bigwedge_{i_2\in I_2} t_2\sigma_{i_2}) \\
&\leq s_1 \wedge s_2
\end{aligned}$$

and

$$\begin{aligned}
\textstyle\bigwedge_{i\in I_1\cup I_2}(t_1 \times t_2)\sigma_i &\simeq ((\textstyle\bigwedge_{i\in I_1\cup I_2} t_1\sigma_i) \times (\textstyle\bigwedge_{i\in I_1\cup I_2} t_2\sigma_i)) \\
&\leq ((\textstyle\bigwedge_{i_1\in I_1} t_1\sigma_{i_1}) \times (\textstyle\bigwedge_{i_2\in I_2} t_2\sigma_{i_2})) \\
&\leq (s_1 \times s_2)
\end{aligned}$$

□

Lemma 9.1.5. *Let t_1 and t_2 be two types and Δ a set of type variables such that $(\mathsf{var}(t_1) \setminus \Delta) \cap (\mathsf{var}(t_2) \setminus \Delta) = \emptyset$. If $t_1 \sqsubseteq_\Delta s_1$ and $t_2 \sqsubseteq_\Delta s_2$, then $t_1 \vee t_2 \sqsubseteq_\Delta s_1 \vee s_2$.*

Proof. Let $[\sigma_{i_1}]_{i_1 \in I_1} \Vdash t_1 \sqsubseteq_\Delta s_1$ and $[\sigma_{i_2}]_{i_2 \in I_2} \Vdash t_2 \sqsubseteq_\Delta s_2$. Then we construct another set of type substitutions $[\sigma_{i_1,i_2}]_{i_1 \in I_1, i_2 \in I_2}$ such that

$$\sigma_{i_1,i_2}(\alpha) = \begin{cases} \sigma_{i_1}(\alpha) & \alpha \in (\mathsf{var}(t_1) \setminus \Delta) \\ \sigma_{i_2}(\alpha) & \alpha \in (\mathsf{var}(t_2) \setminus \Delta) \\ \alpha & \text{otherwise} \end{cases}$$

So we have

$$\begin{array}{rcl} \bigwedge_{i_1 \in I_1, i_2 \in I_2} (t_1 \vee t_2)\sigma_{i_1,i_2} & \simeq & \bigwedge_{i_1 \in I_1} (\bigwedge_{i_2 \in I_2} (t_1 \vee t_2)\sigma_{i_1,i_2}) \\ & \simeq & \bigwedge_{i_1 \in I_1} (\bigwedge_{i_2 \in I_2} ((t_1\sigma_{i_1,i_2}) \vee (t_2\sigma_{i_1,i_2}))) \\ & \simeq & \bigwedge_{i_1 \in I_1} (\bigwedge_{i_2 \in I_2} (t_1\sigma_{i_1} \vee t_2\sigma_{i_2})) \\ & \simeq & \bigwedge_{i_1 \in I_1} (t_1\sigma_{i_1} \vee (\bigwedge_{i_2 \in I_2} t_2\sigma_{i_2})) \\ & \simeq & (\bigwedge_{i_1 \in I_1} t_1\sigma_{i_1}) \vee (\bigwedge_{i_2 \in I_2} t_2\sigma_{i_2}) \\ & \leq & s_1 \vee s_2 \end{array}$$

□

Notice that two successive instantiations can be safely merged into one (see Lemma 9.1.6). Henceforth, we assume that there are no successive instantiations in a given derivation tree. In order to guess where to insert sets of type-substitutions in an implicitly-typed expression, in the following we consider each typing rule of the explicitly-typed calculus used in conjunction with the instantiation rule (Alg-inst). If instantiation can be moved through a given typing rule without affecting typability or changing the result type, then it is not necessary to infer type substitutions at the level of this rule.

Lemma 9.1.6. *Let e be an explicitly-typed expression and $[\sigma_i]_{i \in I}$, $[\sigma_j]_{j \in J}$ two sets of type substitutions. Then*

$$\Delta \wp \Gamma \vdash_{\mathcal{A}} (e[\sigma_i]_{i \in I})[\sigma_j]_{j \in J} : t \iff \Delta \wp \Gamma \vdash_{\mathcal{A}} e([\sigma_j]_{j \in J} \circ [\sigma_i]_{i \in I}) : t$$

Proof. $\Longrightarrow$**:** consider the following derivation:

$$\dfrac{\dfrac{\dfrac{\cdots}{\Delta \wp \Gamma \vdash_{\mathcal{A}} e : s} \quad \sigma_i \sharp \Delta}{\Delta \wp \Gamma \vdash_{\mathcal{A}} e[\sigma_i]_{i \in I} : \bigwedge_{i \in I} s\sigma_i} \quad \sigma_j \sharp \Delta}{\Delta \wp \Gamma \vdash_{\mathcal{A}} (e[\sigma_i]_{i \in I})[\sigma_j]_{j \in J} : \bigwedge_{j \in J} (\bigwedge_{i \in I} s\sigma_i)\sigma_j}$$

As $\sigma_i \sharp \Delta$, $\sigma_j \sharp \Delta$ and $\mathsf{dom}(\sigma_j \circ \sigma_i) = \mathsf{dom}(\sigma_j) \cup \mathsf{dom}(\sigma_i)$, we have $\sigma_j \circ \sigma_i \sharp \Delta$. Then by (Alg-inst), we have $\Delta \wp \Gamma \vdash_{\mathcal{A}} e([\sigma_j \circ \sigma_i]_{i \in I, j \in J}) : \bigwedge_{i \in I, j \in J} s(\sigma_j \circ \sigma_i)$, that is $\Delta \wp \Gamma \vdash_{\mathcal{A}} e([\sigma_j]_{j \in J} \circ [\sigma_i]_{i \in I}) : \bigwedge_{j \in J} (\bigwedge_{i \in I} s\sigma_i)\sigma_j$.

$\Longleftarrow$**:** consider the following derivation:

$$\dfrac{\dfrac{\cdots}{\Delta \wp \Gamma \vdash_{\mathcal{A}} e : s} \quad \sigma_j \circ \sigma_i \sharp \Delta}{\Delta \wp \Gamma \vdash_{\mathcal{A}} e([\sigma_j]_{j \in J} \circ [\sigma_i]_{i \in I}) : \bigwedge_{i \in I, j \in J} s(\sigma_j \circ \sigma_i)}$$

As $\sigma_j \circ \sigma_i \sharp \Delta$ and $\mathsf{dom}(\sigma_j \circ \sigma_i) = \mathsf{dom}(\sigma_j) \cup \mathsf{dom}(\sigma_i)$, we have $\sigma_i \sharp \Delta$ and $\sigma_j \sharp \Delta$. Then applying the rule (Alg-inst) twice, we have $\Delta \wp \Gamma \vdash_{\mathcal{A}} (e[\sigma_i]_{i \in I})[\sigma_j]_{j \in J} : \bigwedge_{j \in J} (\bigwedge_{i \in I} s\sigma_i)\sigma_j$, that is $\Delta \wp \Gamma \vdash_{\mathcal{A}} (e[\sigma_i]_{i \in I})[\sigma_j]_{j \in J} : \bigwedge_{i \in I, j \in J} s(\sigma_j \circ \sigma_i)$.

□

First of all, consider a typing derivation ending with (ALG-PAIR) where both of its sub-derivations end with (ALG-INST)[1]:

$$\dfrac{\dfrac{\dfrac{\cdots}{\Delta \fatsemi \Gamma \vdash_{\mathcal{A}} e_1 : t_1} \quad \forall j_1 \in J_1.\ \sigma_{j_1} \sharp \Delta}{\Delta \fatsemi \Gamma \vdash_{\mathcal{A}} e_1[\sigma_{j_1}]_{j_1 \in J_1} : \bigwedge_{j_1 \in J_1} t_1\sigma_{j_1}} \quad \dfrac{\dfrac{\cdots}{\Delta \fatsemi \Gamma \vdash_{\mathcal{A}} e_2 : t_2} \quad \forall j_2 \in J_2.\ \sigma_{j_2} \sharp \Delta}{\Delta \fatsemi \Gamma \vdash_{\mathcal{A}} e_2[\sigma_{j_2}]_{j_2 \in J_2} : \bigwedge_{j_2 \in J_2} t_1\sigma_{j_2}}}{\Delta \fatsemi \Gamma \vdash_{\mathcal{A}} (e_1[\sigma_{j_1}]_{j_1 \in J_1}, e_2[\sigma_{j_2}]_{j_2 \in J_2}) : (\bigwedge_{j_1 \in J_1} t_1\sigma_{j_1}) \times (\bigwedge_{j_2 \in J_2} t_1\sigma_{j_2})}$$

We rewrite such a derivation as follows:

$$\dfrac{\dfrac{\dfrac{\cdots}{\Delta \fatsemi \Gamma \vdash_{\mathcal{A}} e_1 : t_1} \quad \dfrac{\cdots}{\Delta \fatsemi \Gamma \vdash_{\mathcal{A}} e_2 : t_2}}{\Delta \fatsemi \Gamma \vdash_{\mathcal{A}} (e_1, e_2) : t_1 \times t_2} \quad \forall j \in J_1 \cup J_2.\ \sigma_j \sharp \Delta}{\Delta \fatsemi \Gamma \vdash_{\mathcal{A}} (e_1, e_2)[\sigma_j]_{j \in J_1 \cup J_2} : \bigwedge_{j \in J_1 \cup J_2} (t_1 \times t_2)\sigma_j}$$

Clearly, $\bigwedge_{j \in J_1 \cup J_2} (t_1 \times t_2)\sigma_j \leq (\bigwedge_{j_1 \in J_1} t_1\sigma_{j_1}) \times (\bigwedge_{j_2 \in J_2} t_1\sigma_{j_2})$. Then we can deduce that $(e_1, e_2)[\sigma_j]_{j \in J_1 \cup J_2}$ also has the type $(\bigwedge_{j_1 \in J_1} t_1\sigma_{j_1}) \times (\bigwedge_{j_2 \in J_2} t_1\sigma_{j_2})$ by subsumption. Therefore, we can disregard the sets of type substitutions that are applied inside a pair, since inferring them outside the pair is equivalent. Hence, we can use the following inference rule for pairs.

$$\dfrac{\Delta \fatsemi \Gamma \vdash_{\mathcal{I}} a_1 : t_1 \quad \Delta \fatsemi \Gamma \vdash_{\mathcal{I}} a_2 : t_2}{\Delta \fatsemi \Gamma \vdash_{\mathcal{I}} (a_1, a_2) : t_1 \times t_2}$$

Next, consider a derivation ending of (ALG-PROJ) whose premise is derived by (ALG-INST):

$$\dfrac{\dfrac{\dfrac{\cdots}{\Delta \fatsemi \Gamma \vdash_{\mathcal{A}} e : t} \quad \forall j \in J.\ \sigma_j \sharp \Delta}{\Delta \fatsemi \Gamma \vdash_{\mathcal{A}} e[\sigma_j]_{j \in J} : \bigwedge_{j \in J} t\sigma_j} \quad (\bigwedge_{j \in J} t\sigma_j) \leq \mathbb{1} \times \mathbb{1}}{\Delta \fatsemi \Gamma \vdash_{\mathcal{A}} \pi_i(e[\sigma_j]_{j \in J}) : \boldsymbol{\pi}_i(\bigwedge_{j \in J} t\sigma_j)}$$

According to Lemma 8.2.6, we have $\boldsymbol{\pi}_i(\bigwedge_{j \in J} t\sigma_j) \leq \bigwedge_{j \in J} \boldsymbol{\pi}_i(t)\sigma_j$, but the converse does not necessarily hold. For example, $\boldsymbol{\pi}_1(((t_1 \times t_2) \vee (s_1 \times \alpha \setminus s_2))\{s_2/\alpha\}) = t_1\{s_2/\alpha\}$ while $(\boldsymbol{\pi}_1((t_1 \times t_2) \vee (s_1 \times \alpha \setminus s_2)))\{s_2/\alpha\} = (t_1 \vee s_1)\{s_2/\alpha\}$. So we cannot exchange the instantiation and projection rules without losing completeness. However, as $(\bigwedge_{j \in J} t\sigma_j) \leq \mathbb{1} \times \mathbb{1}$ and $\forall j \in J.\ \sigma_j \sharp \Delta$, we have $t \sqsubseteq_\Delta \mathbb{1} \times \mathbb{1}$. This indicates that for an implicitly-typed expression $\pi_i(a)$, if the inferred type for a is t and there exists $[\sigma_j]_{j \in J}$ such that $[\sigma_j]_{j \in J} \Vdash t \sqsubseteq_\Delta \mathbb{1} \times \mathbb{1}$, then we infer the type $\boldsymbol{\pi}_i(\bigwedge_{j \in J} t\sigma_j)$ for $\pi_i(a)$. Let $\Pi^i_\Delta(t)$ denote the set of such result types, that is,

$$\Pi^i_\Delta(t) = \{u \mid [\sigma_j]_{j \in J} \Vdash t \sqsubseteq_\Delta \mathbb{1} \times \mathbb{1}, u = \boldsymbol{\pi}_i(\bigwedge_{j \in J} t\sigma_j)\}$$

Formally, we have the following inference rule for projections

$$\dfrac{\Delta \fatsemi \Gamma \vdash_{\mathcal{I}} a : t \quad u \in \Pi^i_\Delta(t)}{\Delta \fatsemi \Gamma \vdash_{\mathcal{I}} \pi_i(a) : u}$$

[1] If one of the sub-derivations does not end with (ALG-INST), we can apply a trivial instance of (ALG-INST) with an identity substitution σ_{id}.

The following lemma tells us that $\Pi^i_\Delta(t)$ is "morally" closed by intersection, in the sense that if we take two solutions in $\Pi^i_\Delta(t)$, then we can take also their intersection as a solution, since there always exists in $\Pi^i_\Delta(t)$ a solution at least as precise as their intersection.

Lemma 9.1.7. *Let t be a type and Δ a set of type variables. If $u_1 \in \Pi^i_\Delta(t)$ and $u_2 \in \Pi^i_\Delta(t)$, then $\exists u_0 \in \Pi^i_\Delta(t).\ u_0 \leq u_1 \wedge u_2$.*

Proof. Let $[\sigma_{j_k}]_{j_k \in J_k} \Vdash t \sqsubseteq_\Delta \mathbb{1} \times \mathbb{1}$ and $u_k = \boldsymbol{\pi}_i(\bigwedge_{j_k \in J_k} t\sigma_{j_k})$ for $k = 1, 2$. Then $[\sigma_j]_{j \in J_1 \cup J_2} \Vdash t \sqsubseteq_\Delta \mathbb{1} \times \mathbb{1}$. So $\boldsymbol{\pi}_i(\bigwedge_{j \in J_1 \cup J_2} t\sigma_j) \in \Pi^i_\Delta(t)$. Moreover, by Lemma 8.2.4, we have

$$\boldsymbol{\pi}_i(\bigwedge_{j \in J_1 \cup J_2} t\sigma_j) \leq \boldsymbol{\pi}_i(\bigwedge_{j_1 \in J_1} t\sigma_{j_1}) \wedge \boldsymbol{\pi}_i(\bigwedge_{j_2 \in J_2} t\sigma_{j_2}) = u_1 \wedge u_2$$

□

Since we only consider λ-abstractions with empty decorations, we can consider the following simplified version of (Alg-abstr) that does not use relabeling

$$\frac{\forall i \in I.\ \Delta \cup \mathsf{var}(\bigwedge_{i \in I}(t_i \to s_i)) \fatsemi \Gamma, x : t_i \vdash_{\mathcal{A}} e : s'_i \text{ and } s'_i \leq s_i}{\Delta \fatsemi \Gamma \vdash_{\mathcal{A}} \lambda^{\wedge_{i \in I} t_i \to s_i} x.e : \bigwedge_{i \in I}(t_i \to s_i)} (\text{Alg-abstr0})$$

Suppose the last rule used in the sub-derivations is (Alg-inst).

$$\frac{\forall i \in I.\ \left\{ \begin{array}{l} \dfrac{\overset{\cdots}{\Delta' \fatsemi \Gamma, x : t_i \vdash_{\mathcal{A}} e : s'_i} \quad \forall j \in J.\ \sigma_j \not\sharp \Delta'}{\Delta' \fatsemi \Gamma, x : t_i \vdash_{\mathcal{A}} e[\sigma_j]_{j \in J} : \bigwedge_{j \in J} s'_i \sigma_j} \\ \bigwedge_{j \in J} s'_i \sigma_j \leq s_i \end{array} \right. \quad \Delta' = \Delta \cup \mathsf{var}(\bigwedge_{i \in I}(t_i \to s_i))}{\Delta \fatsemi \Gamma \vdash_{\mathcal{A}} \lambda^{\wedge_{i \in I} t_i \to s_i} x.e[\sigma_j]_{j \in J} : \bigwedge_{i \in I}(t_i \to s_i)}$$

From the side conditions, we deduce that $s'_i \sqsubseteq_{\Delta'} s_i$ for all $i \in I$. Instantiation may be necessary to bridge the gap between the computed type s'_i for e and the type s_i required by the interface, so inferring type substitutions at this stage is mandatory. Therefore, we propose the following inference rule for abstractions.

$$\frac{\forall i \in I.\ \left\{ \begin{array}{l} \Delta \cup \mathsf{var}(\bigwedge_{i \in I} t_i \to s_i) \fatsemi \Gamma, (x : t_i) \vdash_{\mathcal{I}} a : s'_i \\ s'_i \sqsubseteq_{\Delta \cup \mathsf{var}(\bigwedge_{i \in I} t_i \to s_i)} s_i \end{array} \right.}{\Delta \fatsemi \Gamma \vdash_{\mathcal{I}} \lambda^{\wedge_{i \in I} t_i \to s_i} x.a : \bigwedge_{i \in I} t_i \to s_i}$$

In the application case, suppose both sub-derivations end with (Alg-inst):

$$\frac{\begin{array}{l} \dfrac{\overset{\cdots}{\Delta \fatsemi \Gamma \vdash_{\mathcal{A}} e_1 : t} \quad \forall j_1 \in J_1.\ \sigma_{j_1} \not\sharp \Delta}{\Delta \fatsemi \Gamma \vdash_{\mathcal{A}} e_1[\sigma_{j_1}]_{j_1 \in J_1} : \bigwedge_{j_1 \in J_1} t\sigma_{j_1}} \\ \bigwedge_{j_1 \in J_1} t\sigma_{j_1} \leq \mathbb{0} \to \mathbb{1} \end{array} \quad \begin{array}{l} \dfrac{\overset{\cdots}{\Delta \fatsemi \Gamma \vdash_{\mathcal{A}} e_2 : s} \quad \forall j_2 \in J_2.\ \sigma_{j_2} \not\sharp \Delta}{\Delta \fatsemi \Gamma \vdash_{\mathcal{A}} e_2[\sigma_{j_2}]_{j_2 \in J_2} : \bigwedge_{j_2 \in J_2} s\sigma_{j_2}} \\ \bigwedge_{j_2 \in J_2} s\sigma_{j_2} \leq \mathsf{dom}(\bigwedge_{j_1 \in J_1} t\sigma_{j_1}) \end{array}}{\Delta \fatsemi \Gamma \vdash_{\mathcal{A}} (e_1[\sigma_{j_1}]_{j_1 \in J_1})(e_2[\sigma_{j_2}]_{j_2 \in J_2}) : (\bigwedge_{j_1 \in J_1} t\sigma_{j_1}) \cdot (\bigwedge_{j_2 \in J_2} s\sigma_{j_2})}$$

Instantiation may be needed to bridge the gap between the (domain of the) function type and its argument (e.g., to apply $\lambda^{\alpha\to\alpha}x.x$ to 42). The side conditions imply that $[\sigma_{j_1}]_{j_1\in J_1} \Vdash t \sqsubseteq_\Delta \mathbb{0}\to\mathbb{1}$ and $[\sigma_{j_2}]_{j_2\in J_2} \Vdash s \sqsubseteq_\Delta \mathsf{dom}(\bigwedge_{j_1\in J_1} t\sigma_{j_1})$. Therefore, given an implicitly-typed application a_1a_2 where a_1 and a_2 are typed with t and s respectively, we have to find two sets of substitutions $[\sigma_{j_1}]_{j_1\in J_1}$ and $[\sigma_{j_2}]_{j_2\in J_2}$ verifying the above preorder relations to be able to type the application. If such sets of substitutions exist, then we can type the application with $(\bigwedge_{j_1\in J_1} t\sigma_{j_1})\cdot(\bigwedge_{j_2\in J_2} s\sigma_{j_2})$. Let $t \bullet_\Delta s$ denote the set of such result types, that is,

$$t \bullet_\Delta s \stackrel{\text{def}}{=} \left\{ u \;\middle|\; \begin{array}{l} [\sigma_i]_{i\in I} \Vdash t \sqsubseteq_\Delta \mathbb{0}\to\mathbb{1} \\ [\sigma_j]_{j\in J} \Vdash s \sqsubseteq_\Delta \mathsf{dom}(\bigwedge_{i\in I} t\sigma_i) \\ u = \bigwedge_{i\in I} t\sigma_i \cdot \bigwedge_{j\in J} s\sigma_j \end{array} \right\}$$

This set is closed under intersection (see Lemma 9.1.8). Formally, we get the following inference rule for applications

$$\frac{\Delta \fatsemi \Gamma \vdash_{\mathcal{I}} a_1 : t \quad \Delta \fatsemi \Gamma \vdash_{\mathcal{I}} a_2 : s \quad u \in t \bullet_\Delta s}{\Delta \fatsemi \Gamma \vdash_{\mathcal{I}} a_1a_2 : u}$$

Lemma 9.1.8. *Let t, s be two types and Δ a set of type variables. If $u_1 \in t \bullet_\Delta s$ and $u_2 \in t \bullet_\Delta s$, then $\exists u_0 \in t \bullet_\Delta s.\ u_0 \le u_1 \wedge u_2$.*

Proof. Let $u_k = (\bigwedge_{i_k\in I_k} t\sigma_{i_k})\cdot(\bigwedge_{j_k\in J_k} s\sigma_{j_k})$ for $k = 1, 2$. According to Lemma 8.2.15, we have $(\bigwedge_{i\in I_1\cup I_2} t\sigma_i)\cdot(\bigwedge_{j\in J_1\cup J_2} s\sigma_j) \in t \bullet_\Delta s$ and $(\bigwedge_{i\in I_1\cup I_2} t\sigma_i)\cdot(\bigwedge_{j\in J_1\cup J_2} s\sigma_j) \le \bigwedge_{k=1,2}(\bigwedge_{i_k\in I_k} t\sigma_{i_k})\cdot(\bigwedge_{j_k\in J_k} s\sigma_{j_k}) = u_1 \wedge u_2$. □

For type cases, we distinguish the four possible behaviours: (*i*) no branch is selected, (*ii*) the first branch is selected, (*iii*) the second branch is selected, and (*iv*) both branches are selected. In all these cases, we assume that the premises end with (Alg-inst). In case (*i*), we have the following derivation:

$$\frac{\dfrac{\dfrac{\cdots}{\Delta \fatsemi \Gamma \vdash_{\mathcal{A}} e : t'} \quad \forall j\in J.\ \sigma_j \not\sharp \Delta}{\Delta \fatsemi \Gamma \vdash_{\mathcal{A}} e[\sigma_j]_{j\in J} : \bigwedge_{j\in J} t'\sigma_j} \qquad \bigwedge_{j\in J} t'\sigma_j \le \mathbb{0}}{\Delta \fatsemi \Gamma \vdash_{\mathcal{A}} (e[\sigma_j]_{j\in J})\in t\,?\,e_1 : e_2 : \mathbb{0}}$$

Clearly, the side conditions implies $t' \sqsubseteq_\Delta \mathbb{0}$. The type inference rule for implicitly-typed expressions corresponding to this case is then

$$\frac{\Delta \fatsemi \Gamma \vdash_{\mathcal{I}} a : t' \quad t' \sqsubseteq_\Delta \mathbb{0}}{\Delta \fatsemi \Gamma \vdash_{\mathcal{I}} (a\in t\,?\,a_1 : a_2) : \mathbb{0}}$$

For case (*ii*), consider the following derivation:

$$\frac{\dfrac{\dfrac{\cdots}{\Delta \fatsemi \Gamma \vdash_{\mathcal{A}} e : t'} \quad \sigma_j \not\sharp \Delta}{\Delta \fatsemi \Gamma \vdash_{\mathcal{A}} e[\sigma_j]_{j\in J} : \bigwedge_{j\in J} t'\sigma_j} \qquad \bigwedge_{j\in J} t'\sigma_j \le t \qquad \dfrac{\dfrac{\cdots}{\Delta \fatsemi \Gamma \vdash_{\mathcal{A}} e_1 : s_1} \quad \sigma_{j_1} \not\sharp \Delta}{\Delta \fatsemi \Gamma \vdash_{\mathcal{A}} e_1[\sigma_{j_1}]_{j_1\in J_1} : \bigwedge_{j_1\in J_1} s_1\sigma_{j_1}}}{\Delta \fatsemi \Gamma \vdash_{\mathcal{A}} (e[\sigma_j]_{j\in J})\in t\,?\,(e_1[\sigma_{j_1}]_{j_1\in J_1}) : e_2 : \bigwedge_{j_1\in J_1} s_1\sigma_{j_1}}$$

First, such a derivation can be rewritten as

$$\dfrac{\dfrac{\dfrac{\dfrac{\dots}{\Delta \fatsemi \Gamma \vdash_{\mathcal{A}} e : t'} \quad \sigma_j \sharp \Delta}{\Delta \fatsemi \Gamma \vdash_{\mathcal{A}} e[\sigma_j]_{j\in J} : \bigwedge_{j\in J} t'\sigma_j} \quad \bigwedge_{j\in J} t'\sigma_j \leq t \quad \dfrac{\dots}{\Delta \fatsemi \Gamma \vdash_{\mathcal{A}} e_1 : s_1}}{\Delta \fatsemi \Gamma \vdash_{\mathcal{A}} (e[\sigma_j]_{j\in J})\in t\,?\,e_1 : e_2 : s_1} \qquad \sigma_{j_1} \sharp \Delta}{\Delta \fatsemi \Gamma \vdash_{\mathcal{A}} ((e[\sigma_j]_{j\in J})\in t\,?\,e_1 : e_2)[\sigma_{j_1}]_{j_1\in J_1}) : \bigwedge_{j_1\in J_1} s_1\sigma_{j_1}}$$

This indicates that it is equivalent to apply the substitutions $[\sigma_{j_1}]_{j_1\in J_1}$ to e_1 or to the whole type case expression. Looking at the derivation for e, for the first branch to be selected we must have $t' \sqsubseteq_\Delta t$. Note that if $t' \sqsubseteq_\Delta \neg t$, we would have $t' \sqsubseteq_\Delta \mathbb{0}$ by Lemma 9.1.4, and no branch would be selected. Consequently, the type inference rule for a type case where the first branch is selected is as follows.

$$\dfrac{\Delta \fatsemi \Gamma \vdash_{\mathcal{I}} a : t' \quad t' \sqsubseteq_\Delta t \quad t' \not\sqsubseteq_\Delta \neg t \quad \Delta \fatsemi \Gamma \vdash_{\mathcal{I}} a_1 : s}{\Delta \fatsemi \Gamma \vdash_{\mathcal{I}} (a\in t\,?\,a_1 : a_2) : s}$$

Case (iii) is similar to case (ii) where t is replaced by $\neg t$.

At last, consider a derivation of Case (iv):

$$\dfrac{\left\{\begin{array}{l} \dfrac{\dfrac{\dots}{\Delta \fatsemi \Gamma \vdash_{\mathcal{A}} e : t'} \quad \forall j \in J.\ \sigma_j \sharp \Delta}{\Delta \fatsemi \Gamma \vdash_{\mathcal{A}} e[\sigma_j]_{j\in J} : \bigwedge_{j\in J} t'\sigma_j} \\ \\ \bigwedge_{j\in J} t'\sigma_j \not\leq \neg t \quad \text{and} \quad \dfrac{\dfrac{\dots}{\Delta \fatsemi \Gamma \vdash_{\mathcal{A}} e_1 : s_1} \quad \forall j_1 \in J_1.\ \sigma_{j_1} \sharp \Delta}{\Delta \fatsemi \Gamma \vdash_{\mathcal{A}} e_1[\sigma_{j_1}]_{j_1\in J_1} : \bigwedge_{j_1\in J_1} s_1\sigma_{j_1}} \\ \\ \bigwedge_{j\in J} t'\sigma_j \not\leq t \quad \text{and} \quad \dfrac{\dfrac{\dots}{\Delta \fatsemi \Gamma \vdash_{\mathcal{A}} e_2 : s_2} \quad \forall j_2 \in J_2.\ \sigma_{j_2} \sharp \Delta}{\Delta \fatsemi \Gamma \vdash_{\mathcal{A}} e_2[\sigma_{j_2}]_{j_2\in J_2} : \bigwedge_{j_2\in J_2} s_2\sigma_{j_2}} \end{array}\right.}{\Delta \fatsemi \Gamma \vdash_{\mathcal{A}} (e[\sigma_j]_{j\in J}\in t\,?\,(e_1[\sigma_{j_1}]_{j_1\in J_1}) : (e_2[\sigma_{j_2}]_{j_2\in J_2})) : \bigwedge_{j_1\in J_1} s_1\sigma_{j_1} \vee \bigwedge_{j_2\in J_2} s_2\sigma_{j_2}}$$

Using α-conversion if necessary, we can assume that the polymorphic type variables of e_1 and e_2 are distinct, and therefore we have $(\mathsf{var}(s_1) \setminus \Delta) \cap (\mathsf{var}(s_2) \setminus \Delta) = \emptyset$. According to Lemma 9.1.5, we get $s_1 \vee s_2 \sqsubseteq_\Delta \bigwedge_{j_1\in J_1} s_1\sigma_{j_1} \vee \bigwedge_{j_2\in J_2} s_2\sigma_{j_2}$. Let $[\sigma_{j_{12}}]_{j_{12}\in J_{12}} \Vdash s_1 \vee s_2 \sqsubseteq_\Delta \bigwedge_{j_1\in J_1} s_1\sigma_{j_1} \vee \bigwedge_{j_2\in J_2} s_2\sigma_{j_2}$. We can rewrite this derivation as

$$\dfrac{\dfrac{\left\{\begin{array}{l} \dfrac{\dfrac{\dots}{\Delta \fatsemi \Gamma \vdash_{\mathcal{A}} e : t'} \quad \forall j \in J.\ \sigma_j \sharp \Delta}{\Delta \fatsemi \Gamma \vdash_{\mathcal{A}} e[\sigma_j]_{j\in J} : \bigwedge_{j\in J} t'\sigma_j} \\ \\ \bigwedge_{j\in J} t'\sigma_j \not\leq \neg t \quad \text{and} \quad \dfrac{\dots}{\Delta \fatsemi \Gamma \vdash_{\mathcal{A}} e_1 : s_1} \\ \\ \bigwedge_{j\in J} t'\sigma_j \not\leq t \quad \text{and} \quad \dfrac{\dots}{\Delta \fatsemi \Gamma \vdash_{\mathcal{A}} e_2 : s_2} \end{array}\right.}{\Delta \fatsemi \Gamma \vdash_{\mathcal{A}} (e[\sigma_j]_{j\in J}\in t\,?\,e_1 : e_2) : s_1 \vee s_2} \qquad \forall j_{12} \in J_{12}.\ \sigma_{j_{12}} \sharp \Delta}{\Delta \fatsemi \Gamma \vdash_{\mathcal{A}} ((e[\sigma_j]_{j\in J}\in t\,?\,e_1 : e_2)[\sigma_{j_{12}}]_{j_{12}\in J_{12}}) : \bigwedge_{j_{12}\in J_{12}} (s_1 \vee s_2)\sigma_{j_{12}}}$$

As $\bigwedge_{j_{12}\in J_{12}} (s_1 \vee s_2)\sigma_{j_{12}} \leq \bigwedge_{j_1\in J_1} s_1\sigma_{j_1} \vee \bigwedge_{j_2\in J_2} s_2\sigma_{j_2}$, by subsumption, we can deduce that $(e[\sigma_j]_{j\in J}\in t\,?\,e_1 : e_2)[\sigma_{j_{12}}]_{j_{12}\in J_{12}}$ has the type $\bigwedge_{j_1\in J_1} s_1\sigma_{j_1} \vee \bigwedge_{j_2\in J_2} s_2\sigma_{j_2}$. Hence, we eliminate the substitutions that are applied to these two branches.

We now consider the part of the derivation tree which concerns e. With the specific set of substitutions $[\sigma_j]_{j\in J}$, we have $\bigwedge_{j\in J} t'\sigma_j \not\leq \neg t$ and $\bigwedge_{j\in J} t'\sigma_j \not\leq t$, but it does not mean that we have $t' \not\sqsubseteq_\Delta t$ and $t' \not\sqsubseteq_\Delta \neg t$ in general. If $t' \sqsubseteq_\Delta t$ and/or $t' \sqsubseteq_\Delta \neg t$ hold, then we are in one of the previous cases $(i) - (iii)$ (i.e., we type-check at most one branch), and the inferred result type for the whole type case belongs to $\mathbb{0}$, s_1 or s_2. We can then use subsumption to type the whole type-case expression with $s_1 \vee s_2$. Otherwise, both branches are type-checked, and we deduce the corresponding inference rule as follows.

$$\frac{\Delta \fatsemi \Gamma \vdash_{\mathcal{I}} a : t' \quad \begin{cases} t' \not\sqsubseteq_\Delta \neg t & \text{and} \quad \Delta \fatsemi \Gamma \vdash_{\mathcal{I}} a_1 : s_1 \\ t' \not\sqsubseteq_\Delta t & \text{and} \quad \Delta \fatsemi \Gamma \vdash_{\mathcal{I}} a_2 : s_2 \end{cases}}{\Delta \fatsemi \Gamma \vdash_{\mathcal{I}} (a{\in}t\,?\,a_1 : a_2) : s_1 \vee s_2}$$

From the study above, we deduce the type-substitution inference rules for implicitly-typed expressions given in Figure 9.1.

9.2 Soundness and completeness

We now prove that the inference rules of the implicitly-typed calculus given in Figure 9.1 are sound and complete with respect to the type system of the explicitly-typed calculus.

To construct an explicitly-typed expression from an implicitly-typed one a, we have to insert sets of substitutions in a each time a preorder check is performed in the rules of Figure 9.1. For an abstraction $\lambda^{\wedge_{i\in I} t_i\to s_i} x.a$, different sets of substitutions may be constructed when type checking the body under the different hypotheses $x : t_i$. For example, let $a = \lambda^{(\mathtt{Int}\to\mathtt{Int})\wedge(\mathtt{Bool}\to\mathtt{Bool})} x.(\lambda^{\alpha\to\alpha} y.y)x$. When a is type-checked against $\mathtt{Int} \to \mathtt{Int}$, that is, x is assumed to have type $\mathtt{Int}$, we infer the type substitution $\{\mathtt{Int}/\alpha\}$ for $(\lambda^{\alpha\to\alpha} y.y)$. Similarly, we infer $\{\mathtt{Bool}/\alpha\}$ for $(\lambda^{\alpha\to\alpha} y.y)$, when a is type-checked against $\mathtt{Bool} \to \mathtt{Bool}$. We have to collect these two different substitutions when constructing the explicitly-typed expression e which corresponds to a. To this end, we introduce an intersection operator $e \sqcap e'$ of expressions which is defined only for pair of expressions that have similar structure but different type substitutions. For example, the intersection of $(\lambda^{\alpha\to\alpha} y.y)[\{\mathtt{Int}/\alpha\}]x$ and $(\lambda^{\alpha\to\alpha} y.y)[\{\mathtt{Bool}/\alpha\}]x$ will be $(\lambda^{\alpha\to\alpha} y.y)[\{\mathtt{Int}/\alpha\}, \{\mathtt{Bool}/\alpha\}]x$.

Definition 9.2.1. *Let $e, e' \in \mathscr{E}_0$ be two expressions. Their* intersection *$e \sqcap e'$ is defined by induction as:*

$$\begin{array}{rcll}
c \sqcap c & = & c & \\
x \sqcap x & = & x & \\
(e_1, e_2) \sqcap (e'_1, e'_2) & = & ((e_1 \sqcap e'_1), (e_2 \sqcap e'_2)) & \\
\pi_i(e) \sqcap \pi_i(e') & = & \pi_i(e \sqcap e') & \\
e_1 e_2 \sqcap e'_1 e'_2 & = & (e_1 \sqcap e'_1)(e_2 \sqcap e'_2) & \\
(\lambda^{\wedge_{i\in I} t_i\to s_i} x.e) \sqcap (\lambda^{\wedge_{i\in I} t_i\to s_i} x.e') & = & \lambda^{\wedge_{i\in I} t_i\to s_i} x.(e \sqcap e') & \\
(e_0{\in}t\,?\,e_1 : e_2) \sqcap (e'_0{\in}t\,?\,e'_1 : e'_2) & = & e_0 \sqcap e'_0{\in}t\,?\,e_1 \sqcap e'_1 : e_2 \sqcap e'_2 & \\
(e_1[\sigma_j]_{j\in J}) \sqcap (e'_1[\sigma_j]_{j\in J'}) & = & (e_1 \sqcap e'_1)[\sigma_j]_{j\in J\cup J'} & \\
e \sqcap (e'_1[\sigma_j]_{j\in J'}) & = & (e[\sigma_{id}]) \sqcap (e'_1[\sigma_j]_{j\in J'}) & \text{if } e \neq e_1[\sigma_j]_{j\in J} \\
(e_1[\sigma_j]_{j\in J}) \sqcap e' & = & (e_1[\sigma_j]_{j\in J}) \sqcap (e'[\sigma_{id}]) & \text{if } e' \neq e'_1[\sigma_j]_{j\in J'}
\end{array}$$

$$\frac{}{\Delta \fatsemi \Gamma \vdash_{\mathcal{I}} c : b_c}(\textsc{Inf-const}) \qquad \frac{}{\Delta \fatsemi \Gamma \vdash_{\mathcal{I}} x : \Gamma(x)}(\textsc{Inf-var})$$

$$\frac{\Delta \fatsemi \Gamma \vdash_{\mathcal{I}} a_1 : t_1 \qquad \Delta \fatsemi \Gamma \vdash_{\mathcal{I}} a_2 : t_2}{\Delta \fatsemi \Gamma \vdash_{\mathcal{I}} (a_1, a_2) : t_1 \times t_2}(\textsc{Inf-pair})$$

$$\frac{\Delta \fatsemi \Gamma \vdash_{\mathcal{I}} a : t \qquad u \in \Pi^i_\Delta(t)}{\Delta \fatsemi \Gamma \vdash_{\mathcal{I}} \pi_i(a) : u}(\textsc{Inf-proj})$$

$$\frac{\Delta \fatsemi \Gamma \vdash_{\mathcal{I}} a_1 : t \qquad \Delta \fatsemi \Gamma \vdash_{\mathcal{I}} a_2 : s \qquad u \in t \bullet_\Delta s}{\Delta \fatsemi \Gamma \vdash_{\mathcal{I}} a_1 a_2 : u}(\textsc{Inf-appl})$$

$$\frac{\forall i \in I.\ \begin{cases} \Delta \cup \mathsf{var}(\bigwedge_{i\in I} t_i \to s_i) \fatsemi \Gamma, (x : t_i) \vdash_{\mathcal{I}} a : s'_i \\ s'_i \sqsubseteq_{\Delta \cup \mathsf{var}(\bigwedge_{i\in I} t_i \to s_i)} s_i \end{cases}}{\Delta \fatsemi \Gamma \vdash_{\mathcal{I}} \lambda^{\bigwedge_{i\in I} t_i \to s_i} x.a : \bigwedge_{i\in I} t_i \to s_i}(\textsc{Inf-abstr})$$

$$\frac{\Delta \fatsemi \Gamma \vdash_{\mathcal{I}} a : t' \qquad t' \sqsubseteq_\Delta \mathbb{0}}{\Delta \fatsemi \Gamma \vdash_{\mathcal{I}} (a{\in}t\,?\,a_1 : a_2) : \mathbb{0}}(\textsc{Inf-case-none})$$

$$\frac{\Delta \fatsemi \Gamma \vdash_{\mathcal{I}} a : t' \qquad t' \sqsubseteq_\Delta t \qquad t' \not\sqsubseteq_\Delta \neg t \qquad \Delta \fatsemi \Gamma \vdash_{\mathcal{I}} a_1 : s}{\Delta \fatsemi \Gamma \vdash_{\mathcal{I}} (a{\in}t\,?\,a_1 : a_2) : s}(\textsc{Inf-case-fst})$$

$$\frac{\Delta \fatsemi \Gamma \vdash_{\mathcal{I}} a : t' \qquad t' \sqsubseteq_\Delta \neg t \qquad t' \not\sqsubseteq_\Delta t \qquad \Delta \fatsemi \Gamma \vdash_{\mathcal{I}} a_2 : s}{\Delta \fatsemi \Gamma \vdash_{\mathcal{I}} (a{\in}t\,?\,a_1 : a_2) : s}(\textsc{Inf-case-snd})$$

$$\frac{\Delta \fatsemi \Gamma \vdash_{\mathcal{I}} a : t' \qquad \begin{cases} t' \not\sqsubseteq_\Delta \neg t & \text{and} \quad \Delta \fatsemi \Gamma \vdash_{\mathcal{I}} a_1 : s_1 \\ t' \not\sqsubseteq_\Delta t & \text{and} \quad \Delta \fatsemi \Gamma \vdash_{\mathcal{I}} a_2 : s_2 \end{cases}}{\Delta \fatsemi \Gamma \vdash_{\mathcal{I}} (a{\in}t\,?\,a_1 : a_2) : s_1 \vee s_2}(\textsc{Inf-case-both})$$

Figure 9.1: Type-substitution inference rules

where σ_{id} is the identity type substitution and is undefined otherwise.

The intersection of a same constant or a same variable is the constant or the variable itself. If e and e' have the same form, then their intersection is defined if their intersections of the corresponding sub-expressions are defined. In particular when e is form of $e_1[\sigma_j]_{j\in J}$ and e' is form of $e'_1[\sigma_j]_{j\in J'}$, we merge the sets of substitutions $[\sigma_j]_{j\in J}$ and $[\sigma_j]_{j\in J'}$ into one set $[\sigma_j]_{j\in J\cup J'}$. Otherwise, e and e' have different forms. The only possible case for their intersection is they have similar structure but

one with instantiations and the other without (*i.e.*, $e = e_1[\sigma_j]_{j\in J}, e' \neq e'_1[\sigma_j]_{j\in J'}$ or $e \neq e_1[\sigma_j]_{j\in J}, e' = e'_1[\sigma_j]_{j\in J'}$). To keep the inferred information and reuse the defined cases above, we add the identity substitution σ_{id} to the one without substitutions (*i.e.*, $e'[\sigma_{id}]$ or $e[\sigma_{id}]$) to make them have the same form. Note that σ_{id} is important so as to keep the information we have inferred. Let us infer the substitutions for the abstraction $\lambda^{(t_1\to s_1)\wedge(t_2\to s_2)}x.e$. Assume that we have inferred some substitutions for the body e under $t_1 \to s_1$ and $t_2 \to s_2$ respectively, yielding two explicitly-typed expressions e_1 and $e_2[\sigma_j]_{j\in J}$. If we did not add the identity substitution σ_{id} for the intersection of e_1 and $e_2[\sigma_j]_{j\in J}$, that is, $e_1 \sqcap (e_2[\sigma_j]_{j\in J})$ were $(e_1 \sqcap e_2)[\sigma_j]_{j\in J}$ rather than $(e_1 \sqcap e_2)([\sigma_{id}] \cup [\sigma_j]_{j\in J})$, then the substitutions we inferred under $t_1 \to s_1$ would be lost since they may be modified by $[\sigma_j]_{j\in J}$.

Lemma 9.2.2. *Let $e, e' \in \mathscr{E}_0$ be two expressions. If $erase(e) = erase(e')$, then $e \sqcap e'$ exists and $erase(e \sqcap e') = erase(e) = erase(e')$.*

Proof. By induction on the structures of e and e'. Because $erase(e) = erase(e')$, the two expressions have the same structure up to their sets of type substitutions.

$\underline{c}$**,** $\underline{c}$**:** straightforward.
$\underline{x}$**,** $\underline{x}$**:** straightforward.
$\underline{(e_1, e_2)}$**,** $\underline{(e'_1, e'_2)}$**:** we have $erase(e_i) = erase(e'_i)$. By induction, $e_i \sqcap e'_i$ exists and $erase(e_i \sqcap e'_i) = erase(e_i) = erase(e'_i)$. Therefore $(e_1, e_2) \sqcap (e'_1, e'_2)$ exists and

$$\begin{aligned} erase((e_1, e_2) \sqcap (e'_1, e'_2)) &= erase(((e_1 \sqcap e'_1), (e_2 \sqcap e'_2))) \\ &= (erase(e_1 \sqcap e'_1), erase(e_2 \sqcap e'_2)) \\ &= (erase(e_1), erase(e_2)) \\ &= erase((e_1, e_2)) \end{aligned}$$

Similarly, we also have $erase((e_1, e_2) \sqcap (e'_1, e'_2)) = erase((e'_1, e'_2))$.

$\underline{\pi_i(e)}$**,** $\underline{\pi_i(e')}$**:** we have $erase(e) = erase(e')$. By induction, $e \sqcap e'$ exists and $erase(e \sqcap e') = erase(e) = erase(e')$. Therefore $\pi_i(e) \sqcap \pi_i(e')$ exists and

$$\begin{aligned} erase(\pi_i(e) \sqcap \pi_i(e')) &= erase(\pi_i(e \sqcap e')) \\ &= \pi_i(erase(e \sqcap e')) \\ &= \pi_i(erase(e)) \\ &= erase(\pi_i(e)) \end{aligned}$$

Similarly, we also have $erase(\pi_i(e) \sqcap \pi_i(e')) = erase(\pi_i(e'))$.

$\underline{e_1e_2}$**,** $\underline{e'_1e'_2}$**:** we have $erase(e_i) = erase(e'_i)$. By induction, $e_i \sqcap e'_i$ exists and $erase(e_i \sqcap e'_i) = erase(e_i) = erase(e'_i)$. Therefore $e_1e_2 \sqcap e'_1e'_2$ exists and

$$\begin{aligned} erase((e_1e_2) \sqcap (e'_1e'_2)) &= erase((e_1 \sqcap e'_1)(e_2 \sqcap e'_2)) \\ &= erase(e_1 \sqcap e'_1)erase(e_2 \sqcap e'_2) \\ &= erase(e_1)erase(e_2) \\ &= erase(e_1e_2) \end{aligned}$$

Similarly, we also have $erase((e_1e_2) \sqcap (e'_1e'_2)) = erase(e'_1e'_2)$.

$\underline{\lambda^{\wedge_{i\in I}t_i\to s_i}x.e,\ \lambda^{\wedge_{i\in I}t_i\to s_i}x.e'}$**:** we have $erase(e) = erase(e')$. By induction, $e \sqcap e'$ exists and $erase(e \sqcap e') = erase(e) = erase(e')$. Therefore $(\lambda^{\wedge_{i\in I}t_i\to s_i}x.e)\sqcap(\lambda^{\wedge_{i\in I}t_i\to s_i}x.e')$ exists and

$$\begin{array}{rcl} erase((\lambda^{\wedge_{i\in I}t_i\to s_i}x.e) \sqcap (\lambda^{\wedge_{i\in I}t_i\to s_i}x.e')) = & erase(\lambda^{\wedge_{i\in I}t_i\to s_i}x.(e \sqcap e')) \\ = & \lambda^{\wedge_{i\in I}t_i\to s_i}x.erase((e \sqcap e')) \\ = & \lambda^{\wedge_{i\in I}t_i\to s_i}x.erase(e) \\ = & erase(\lambda^{\wedge_{i\in I}t_i\to s_i}x.e) \end{array}$$

Similarly, we also have

$$erase((\lambda^{\wedge_{i\in I}t_i\to s_i}x.e) \sqcap (\lambda^{\wedge_{i\in I}t_i\to s_i}x.e')) = erase(\lambda^{\wedge_{i\in I}t_i\to s_i}x.e')$$

$\underline{e_0{\in}t\,?\,e_1 : e_2,\ e_0'{\in}t\,?\,e_1' : e_2'}$**:** we have $erase(e_i) = erase(e_i')$. By induction, $e_i \sqcap e_i'$ exists and $erase(e_i \sqcap e_i') = erase(e_i) = erase(e_i')$. Therefore $(e_0{\in}t\,?\,e_1 : e_2)\sqcap(e_0'{\in}t\,?\,e_1' : e_2')$ exists and

$$\begin{array}{rl} & erase((e_0{\in}t\,?\,e_1 : e_2) \sqcap (e_0'{\in}t\,?\,e_1' : e_2')) \\ = & erase((e_0 \sqcap e_0'){\in}t\,?\,(e_1 \sqcap e_1') : (e_2 \sqcap e_2')) \\ = & erase(e_0 \sqcap e_0'){\in}t\,?\,erase(e_1 \sqcap e_1') : erase(e_2 \sqcap e_2') \\ = & erase(e_0){\in}t\,?\,erase(e_1) : erase(e_2) \\ = & erase(e_0{\in}t\,?\,e_1 : e_2) \end{array}$$

Similarly, we also have

$$erase((e_0{\in}t\,?\,e_1 : e_2) \sqcap (e_0'{\in}t\,?\,e_1' : e_2')) = erase(e_0'{\in}t\,?\,e_1' : e_2')$$

$\underline{e[\sigma_j]_{j\in J},\ e'[\sigma_j]_{j\in J'}}$**:** we have $erase(e) = erase(e')$. By induction, $e \sqcap e'$ exists and $erase(e \sqcap e') = erase(e) = erase(e')$. Therefore $(e[\sigma_j]_{j\in J}) \sqcap (e'[\sigma_j]_{j\in J'})$ exists and

$$\begin{array}{rcl} erase((e[\sigma_j]_{j\in J}) \sqcap (e'[\sigma_j]_{j\in J'})) = & erase((e \sqcap e')[\sigma_j]_{j\in J\cup J'}) \\ = & erase(e \sqcap e') \\ = & erase(e) \\ = & erase(e[\sigma_j]_{j\in J}) \end{array}$$

Similarly, we also have $erase((e[\sigma_j]_{j\in J}) \sqcap (e'[\sigma_j]_{j\in J'})) = erase(e'[\sigma_j]_{j\in J'})$.
$\underline{e,\ e'[\sigma_j]_{j\in J'}}$**:** a special case of $e[\sigma_j]_{j\in J}$ and $e'[\sigma_j]_{j\in J'}$ where $[\sigma_j]_{j\in J} = [\sigma_{id}]$.
$\underline{e[\sigma_j]_{j\in J},\ e'}$**:** a special case of $e[\sigma_j]_{j\in J}$ and $e'[\sigma_j]_{j\in J'}$ where $[\sigma_j]_{j\in J'} = [\sigma_{id}]$.

□

Lemma 9.2.3. *Let $e, e' \in \mathscr{E}_0$ be two expressions. If $erase(e) = erase(e')$, $\Delta \wr \Gamma \vdash e : t$, $\Delta' \wr \Gamma' \vdash e' : t'$, $e \sharp \Delta'$ and $e' \sharp \Delta$, then $\Delta \wr \Gamma \vdash e \sqcap e' : t$ and $\Delta' \wr \Gamma' \vdash e \sqcap e' : t'$.*

Proof. According to Lemma 9.2.2, $e \sqcap e'$ exists and $erase(e \sqcap e') = erase(e) = erase(e')$. We only prove $\Delta \wr \Gamma \vdash e \sqcap e' : t$ as the other case is similar. For simplicity, we just consider one set of type substitutions. For several sets of type substitutions, we can either compose them or apply (*instinter*) several times. The proof proceeds by induction on $\Delta \wr \Gamma \vdash e : t$.

(*const*): $\Delta \wp \Gamma \vdash c : b_c$. As $erase(e') = c$, e' is either c or $c[\sigma_j]_{j\in J}$. If $e' = c$, then $e \sqcap e' = c$, and the result follows straightforwardly. Otherwise, we have $e \sqcap e' = c[\sigma_{id}, \sigma_j]_{j\in J}$. Since $e' \sharp \Delta$, we have $\sigma_j \sharp \Delta$. By (*instinter*), we have $\Delta \wp \Gamma \vdash c[\sigma_{id}, \sigma_j]_{j\in J} : b_c \wedge \bigwedge_{j\in J} b_c\sigma_j$, that is, $\Delta \wp \Gamma \vdash c[\sigma_{id}, \sigma_j]_{j\in J} : b_c$.

(*var*): $\Gamma \vdash x : \Gamma(x)$. As $erase(e') = x$, e' is either x or $x[\sigma_j]_{j\in J}$. If $e' = x$, then $e \sqcap e' = x$, and the result follows straightforwardly. Otherwise, we have $e \sqcap e' = x[\sigma_{id}, \sigma_j]_{j\in J}$. Since $e' \sharp \Delta$, we have $\sigma_j \sharp \Delta$. By (*instinter*), we have $\Delta \wp \Gamma \vdash x[\sigma_{id}, \sigma_j]_{j\in J} : \Gamma(x) \wedge \bigwedge_{j\in J} \Gamma(x)\sigma_j$, that is, $\Delta \wp \Gamma \vdash x[\sigma_{id}, \sigma_j]_{j\in J} : \Gamma(x)$.

(*pair*): consider the following derivation:

$$\frac{\dfrac{\cdots}{\Delta \wp \Gamma \vdash e_1 : t_1} \quad \dfrac{\cdots}{\Delta \wp \Gamma \vdash e_2 : t_2}}{\Delta \wp \Gamma \vdash (e_1, e_2) : t_1 \times t_2}\ (pair)$$

As $erase(e') = (erase(e_1), erase(e_2))$, e' is either (e'_1, e'_2) or $(e'_1, e'_2)[\sigma_j]_{j\in J}$ such that $erase(e'_i) = erase(e_i)$. By induction, we have $\Delta \wp \Gamma \vdash e_i \sqcap e'_i : t_i$. Then by (*pair*), we have $\Delta \wp \Gamma \vdash (e_1 \sqcap e'_1, e_2 \sqcap e'_2) : (t_1 \times t_2)$. If $e' = (e'_1, e'_2)$, then $e \sqcap e' = (e_1 \sqcap e'_1, e_2 \sqcap e'_2)$. So the result follows.
Otherwise, $e \sqcap e' = (e_1 \sqcap e'_1, e_2 \sqcap e'_2)[\sigma_{id}, \sigma_j]_{j\in J}$. Since $e' \sharp \Delta$, we have $\sigma_j \sharp \Delta$. By (*instinter*), we have $\Delta \wp \Gamma \vdash (e_1 \sqcap e'_1, e_2 \sqcap e'_2)[\sigma_{id}, \sigma_j]_{j\in J} : (t_1 \times t_2) \wedge \bigwedge_{j\in J}(t_1 \times t_2)\sigma_j$. Finally, by (*subsum*), we get $\Delta \wp \Gamma \vdash (e_1 \sqcap e'_1, e_2 \sqcap e'_2)[\sigma_{id}, \sigma_j]_{j\in J} : (t_1 \times t_2)$.

(*proj*): consider the following derivation:

$$\frac{\dfrac{\cdots}{\Delta \wp \Gamma \vdash e_0 : t_1 \times t_2}}{\Delta \wp \Gamma \vdash \pi_i(e_0) : t_i}\ (proj)$$

As $erase(e') = \pi_i(erase(e_0))$, e' is either $\pi_i(e'_0)$ or $\pi_i(e'_0)[\sigma_j]_{j\in J}$ such that $erase(e'_0) = erase(e_0)$. By induction, we have $\Delta \wp \Gamma \vdash e_0 \sqcap e'_0 : (t_1 \times t_2)$. Then by (*proj*), we have $\Delta \wp \Gamma \vdash \pi_i(e_0 \sqcap e'_0) : t_i$. If $e' = \pi_i(e'_0)$, then $e \sqcap e' = \pi_i(e_0 \sqcap e'_0)$. So the result follows.
Otherwise, $e \sqcap e' = \pi_i(e_0 \sqcap e'_0)[\sigma_{id}, \sigma_j]_{j\in J}$. Since $e' \sharp \Delta$, we have $\sigma_j \sharp \Delta$. By (*instinter*), we have $\Delta \wp \Gamma \vdash \pi_i(e_0 \sqcap e'_0)[\sigma_{id}, \sigma_j]_{j\in J} : t_i \wedge \bigwedge_{j\in J} t_i\sigma_j$. Finally, by (*subsum*), we get $\Delta \wp \Gamma \vdash \pi_i(e_0 \sqcap e'_0)[\sigma_{id}, \sigma_j]_{j\in J} : t_i$.

(*appl*): consider the following derivation:

$$\frac{\dfrac{\cdots}{\Delta \wp \Gamma \vdash e_1 : t \to s} \quad \dfrac{\cdots}{\Delta \wp \Gamma \vdash e_2 : t}}{\Delta \wp \Gamma \vdash e_1 e_2 : s}\ (pair)$$

As $erase(e') = erase(e_1)erase(e_2)$, e' is either $e'_1 e'_2$ or $(e'_1 e'_2)[\sigma_j]_{j\in J}$ such that $erase(e'_i) = erase(e_i)$. By induction, we have $\Delta \wp \Gamma \vdash e_1 \sqcap e'_1 : t \to s$ and $\Delta \wp \Gamma \vdash e_2 \sqcap e'_2 : t$. Then by (*appl*), we have $\Delta \wp \Gamma \vdash (e_1 \sqcap e'_1)(e_2 \sqcap e'_2) : s$. If $e' = e'_1 e'_2$, then $e \sqcap e' = (e_1 \sqcap e'_1)(e_2 \sqcap e'_2)$. So the result follows.
Otherwise, $e \sqcap e' = ((e_1 \sqcap e'_1)(e_2 \sqcap e'_2))[\sigma_{id}, \sigma_j]_{j\in J}$. Since $e' \sharp \Delta$, we have $\sigma_j \sharp \Delta$. By (*instinter*), we have $\Delta \wp \Gamma \vdash ((e_1 \sqcap e'_1)(e_2 \sqcap e'_2))[\sigma_{id}, \sigma_j]_{j\in J} : s \wedge \bigwedge_{j\in J} s\sigma_j$. Finally, by (*subsum*), we get $\Delta \wp \Gamma \vdash ((e_1 \sqcap e'_1)(e_2 \sqcap e'_2))[\sigma_{id}, \sigma_j]_{j\in J} : s$.

(*abstr*): consider the following derivation:

$$\frac{\begin{array}{l}\forall i \in I.\ \overset{\cdots}{\overline{\Delta'' \wp \Gamma, (x : t_i) \vdash e_0 : s_i}} \\ \Delta'' = \Delta \cup \mathsf{var}(\bigwedge_{i\in I} t_i \to s_i)\end{array}}{\Delta \wp \Gamma \vdash \lambda^{\wedge_{i\in I} t_i \to s_i} x.e_0 : \bigwedge_{i\in I} t_i \to s_i}\ (abstr)$$

As $erase(e') = \lambda^{\wedge_{i\in I} t_i \to s_i} x.erase(e_0)$, e' is either $\lambda^{\wedge_{i\in I} t_i \to s_i} x.e'_0$ or $(\lambda^{\wedge_{i\in I} t_i \to s_i} x.e'_0)[\sigma_j]_{j\in J}$ such that $erase(e'_0) = erase(e_0)$. As $\lambda^{\wedge_{i\in I} t_i \to s_i} x.e'_0$ is well-typed under Δ' and Γ', $e'_0 \not\equiv \Delta' \cup \mathsf{var}(\bigwedge_{i\in I} t_i \to s_i)$. By induction, we have $\Delta'' \wp \Gamma, (x : t_i) \vdash e_0 \sqcap e'_0 : s_i$. Then by (*abstr*), we have $\Delta \wp \Gamma \vdash \lambda^{\wedge_{i\in I} t_i \to s_i} x.e_0 \sqcap e'_0 : \bigwedge_{i\in I} t_i \to s_i$. If $e' = \lambda^{\wedge_{i\in I} t_i \to s_i} x.e'_0$, then $e \sqcap e' = \lambda^{\wedge_{i\in I} t_i \to s_i} x.e_0 \sqcap e'_0$. So the result follows.

Otherwise, $e \sqcap e' = (\lambda^{\wedge_{i\in I} t_i \to s_i} x.e_0 \sqcap e'_0)[\sigma_{id}, \sigma_j]_{j\in J}$. Since $e' \not\equiv \Delta$, we have $\sigma_j \not\equiv \Delta$. By (*instinter*), we have $\Delta \wp \Gamma \vdash (\lambda^{\wedge_{i\in I} t_i \to s_i} x.e_0 \sqcap e'_0)[\sigma_{id}, \sigma_j]_{j\in J} : (\bigwedge_{i\in I} t_i \to s_i) \wedge \bigwedge_{j\in J} (\bigwedge_{i\in I} t_i \to s_i)\sigma_j$. Finally, by (*subsum*), we get $\Delta \wp \Gamma \vdash (\lambda^{\wedge_{i\in I} t_i \to s_i} x.e_0 \sqcap e'_0)[\sigma_{id}, \sigma_j]_{j\in J} : \bigwedge_{i\in I} t_i \to s_i$.

(*case*): consider the following derivation

$$\frac{\overset{\cdots}{\overline{\Delta \wp \Gamma \vdash e_0 : t'}} \quad \begin{cases} t' \not\leq \neg t & \Rightarrow & \overset{\cdots}{\overline{\Delta \wp \Gamma \vdash e_1 : s}} \\ t' \not\leq t & \Rightarrow & \overset{\cdots}{\overline{\Delta \wp \Gamma \vdash e_2 : s}} \end{cases}}{\Delta \wp \Gamma \vdash (e_0 \in t\,?\,e_1 : e_2) : s}\ (case)$$

As $erase(e') = erase(e_0) \in t\,?\,erase(e_1) : erase(e_2)$, e' is either $e'_0 \in t\,?\,e'_1 : e'_2$ or $(e'_0 \in t\,?\,e'_1 : e'_2)[\sigma_j]_{j\in J}$ such that $erase(e'_i) = erase(e_i)$. By induction, we have $\Delta \wp \Gamma \vdash e_0 \sqcap e'_0 : t'$ and $\Delta \wp \Gamma \vdash e_i \sqcap e'_i : s$. Then by (*case*), we have $\Delta \wp \Gamma \vdash ((e_0 \sqcap e'_0) \in t\,?\,(e_1 \sqcap e'_1) : (e_2 \sqcap e'_2)) : s$. If $e' = e'_0 \in t\,?\,e'_1 : e'_2$, then $e \sqcap e' = (e_0 \sqcap e'_0) \in t\,?\,(e_1 \sqcap e'_1) : (e_2 \sqcap e'_2)$. So the result follows.

Otherwise, $e \sqcap e' = ((e_0 \sqcap e'_0) \in t\,?\,(e_1 \sqcap e'_1) : (e_2 \sqcap e'_2))[\sigma_{id}, \sigma_j]_{j\in J}$. Since $e' \not\equiv \Delta$, we have $\sigma_j \not\equiv \Delta$. By (*instinter*), we have $\Delta \wp \Gamma \vdash ((e_0 \sqcap e'_0) \in t\,?\,(e_1 \sqcap e'_1) : (e_2 \sqcap e'_2))[\sigma_{id}, \sigma_j]_{j\in J} : s \wedge \bigwedge_{j\in J} s\sigma_j$. Finally, by (*subsum*), we get $\Delta \wp \Gamma \vdash ((e_0 \sqcap e'_0) \in t\,?\,(e_1 \sqcap e'_1) : (e_2 \sqcap e'_2))[\sigma_{id}, \sigma_j]_{j\in J} : s$.

(*instinter*): consider the following derivation:

$$\frac{\overset{\cdots}{\overline{\Delta \wp \Gamma \vdash e_0 : t}} \qquad \sigma_j \not\equiv \Delta}{\Delta \wp \Gamma \vdash e_0[\sigma_j]_{j\in J} : \bigwedge_{j\in J} t\sigma_j}\ (instinter)$$

As $erase(e') = erase(e_0)$, e' is either e'_0 or $e'_0[\sigma_j]_{j\in J'}$ such that $erase(e'_0) = erase(e_0)$. By induction, we have $\Delta \wp \Gamma \vdash e_0 \sqcap e'_0 : t$. If $e' = e'_0$, then $e \sqcap e' = (e_0 \sqcap e'_0)[\sigma_j, \sigma_{id}]_{j\in J}$. By (*instinter*), we have $\Delta \wp \Gamma \vdash (e_0 \sqcap e'_0)[\sigma_j, \sigma_{id}]_{j\in J} : \bigwedge_{j\in J} t\sigma_j \wedge t$. Finally, by (*subsum*), we get $\Delta \wp \Gamma \vdash (e_0 \sqcap e'_0)[\sigma_j, \sigma_{id}]_{j\in J} : \bigwedge_{j\in J} t\sigma_j$.

Otherwise, $e \sqcap e' = (e_0 \sqcap e'_0)[\sigma_j]_{j\in J\cup J'}$. Since $e' \not\equiv \Delta$, we have $\sigma_j \not\equiv \Delta$ for all $j \in J'$. By (*instinter*), we have $\Delta \wp \Gamma \vdash (e_0 \sqcap e'_0)[\sigma_j]_{j\in J\cup J'} : \bigwedge_{j\in J\cup J'} t\sigma_j$. Finally, by (*subsum*), we get $\Delta \wp \Gamma \vdash (e_0 \sqcap e'_0)[\sigma_j]_{j\in J\cup J'} : \bigwedge_{j\in J} t\sigma_j$.

(*subsum*)**:** there exists a type s such that

$$\frac{\dfrac{\ldots}{\Delta \fatsemi \Gamma \vdash e : s} \quad s \leq t}{\Delta \fatsemi \Gamma \vdash e : t}\ (subsum)$$

By induction, we have $\Delta \fatsemi \Gamma \vdash e \sqcap e' : s$. Then the rule (*subsum*) gives us $\Delta \fatsemi \Gamma \vdash e \sqcap e' : t$.

□

Corollary 9.2.4. *Let $e, e' \in \mathscr{E}_0$ be two expressions. If $erase(e) = erase(e')$, $\Delta \fatsemi \Gamma \vdash_{\mathcal{A}} e : t$, $\Delta' \fatsemi \Gamma' \vdash_{\mathcal{A}} e' : t'$, $e \sharp \Delta'$ and $e' \sharp \Delta$, then*

1. *there exists s such that $\Delta \fatsemi \Gamma \vdash_{\mathcal{A}} e \sqcap e' : s$ and $s \leq t$.*
2. *there exists s' such that $\Delta' \fatsemi \Gamma' \vdash_{\mathcal{A}} e \sqcap e' : s'$ and $s' \leq t'$.*

Proof. Immediate consequence of Lemma 9.2.3 and Theorems 8.2.19 and 8.2.20. □

These type-substitution inference rules are sound and complete with respect to the typing algorithm in Section 8.2, modulo the restriction that all the decorations in the λ-abstractions are empty.

Theorem 9.2.5 (Soundness). *If $\Delta \fatsemi \Gamma \vdash_{\mathcal{I}} a : t$, then there exists an explicitly-typed expression $e \in \mathscr{E}_0$ such that $erase(e) = a$ and $\Delta \fatsemi \Gamma \vdash_{\mathcal{A}} e : t$.*

Proof. By induction on the derivation of $\Delta \fatsemi \Gamma \vdash_{\mathcal{I}} a : t$. We proceed by a case analysis of the last rule used in the derivation.

(Inf-const)**:** straightforward (take e as c).
(Inf-var)**:** straightforward (take e as x).
(Inf-pair)**:** consider the derivation

$$\frac{\dfrac{\ldots}{\Delta \fatsemi \Gamma \vdash_{\mathcal{I}} a_1 : t_1} \quad \dfrac{\ldots}{\Delta \fatsemi \Gamma \vdash_{\mathcal{I}} a_2 : t_2}}{\Delta \fatsemi \Gamma \vdash_{\mathcal{I}} (a_1, a_2) : t_1 \times t_2}$$

Applying the induction hypothesis, there exists an expression e_i such that $erase(e_i) = a_i$ and $\Delta \fatsemi \Gamma \vdash_{\mathcal{A}} e_i : t_i$. Then by (Alg-pair), we have $\Delta \fatsemi \Gamma \vdash_{\mathcal{A}} (e_1, e_2) : t_1 \times t_2$. Moreover, according to Definition 9.1.2, we have

$$erase((e_1, e_2)) = (erase(e_1), erase(e_2)) = (a_1, a_2).$$

(Inf-proj)**:** consider the derivation

$$\frac{\dfrac{\ldots}{\Delta \fatsemi \Gamma \vdash_{\mathcal{I}} a : t} \quad u \in \Pi^i_\Delta(t)}{\Delta \fatsemi \Gamma \vdash_{\mathcal{I}} \pi_i(a) : u}$$

By induction, there exists an expression e such that $erase(e) = a$ and $\Delta \fatsemi \Gamma \vdash_{\mathcal{A}} e : t$. Let $u = \boldsymbol{\pi}_i(\bigwedge_{i \in I} t\sigma_i)$. As $\sigma_i \sharp \Delta$, by (Alg-inst), we have $\Delta \fatsemi \Gamma \vdash_{\mathcal{A}} e[\sigma_i]_{i \in I} : \bigwedge_{i \in I} t\sigma_i$. Moreover, since $\bigwedge_{i \in I} t\sigma_i \leq \mathbb{1} \times \mathbb{1}$, by (Alg-proj), we get $\Delta \fatsemi \Gamma \vdash_{\mathcal{A}} \pi_i(e[\sigma_i]_{i \in I}) : \boldsymbol{\pi}_i(\bigwedge_{i \in I} t\sigma_i)$. Finally, according to Definition 9.1.2, we have

$$erase(\pi_i(e[\sigma_i]_{i \in I})) = \pi_i(erase(e[\sigma_i]_{i \in I})) = \pi_i(erase(e)) = \pi_i(a)$$

(INF-APPL): consider the derivation

$$\frac{\dfrac{\cdots}{\Delta \fatsemi \Gamma \vdash_{\mathcal{I}} a_1 : t} \quad \dfrac{\cdots}{\Delta \fatsemi \Gamma \vdash_{\mathcal{I}} a_2 : s} \quad u \in t \bullet_{\Delta} s}{\Delta \fatsemi \Gamma \vdash_{\mathcal{I}} a_1 a_2 : u}$$

By induction, we have that (i) there exists an expression e_1 such that $\mathit{erase}(e_1) = a_1$ and $\Delta \fatsemi \Gamma \vdash_{\mathcal{A}} e_1 : t$ and (ii) there exists an expression e_2 such that $\mathit{erase}(e_2) = a_2$ and $\Delta \fatsemi \Gamma \vdash_{\mathcal{A}} e_2 : s$. Let $u = (\bigwedge_{i\in I} t\sigma_i) \cdot (\bigwedge_{j\in J} s\sigma_j)$. As $\sigma_h \sharp \Delta$ for $h \in I \cup J$, applying (ALG-INST), we get $\Delta \fatsemi \Gamma \vdash_{\mathcal{A}} e_1[\sigma_i]_{i\in I} : \bigwedge_{i\in I} t\sigma_i$ and $\Delta \fatsemi \Gamma \vdash_{\mathcal{A}} e_2[\sigma_j]_{j\in J} : \bigwedge_{j\in J} s\sigma_j$. Then by (ALG-APPL), we have $\Delta \fatsemi \Gamma \vdash_{\mathcal{A}} (e_1[\sigma_i]_{i\in I})(e_2[\sigma_j]_{j\in J}) : (\bigwedge_{i\in I} t\sigma_i) \cdot (\bigwedge_{j\in J} s\sigma_j)$. Furthermore, according to Definition 9.1.2, we have

$$\mathit{erase}((e_1[\sigma_i]_{i\in I})(e_2[\sigma_j]_{j\in J})) = \mathit{erase}(e_1)\mathit{erase}(e_2) = a_1 a_2$$

(INF-ABSTR): consider the derivation

$$\frac{\forall i \in I. \begin{cases} \dfrac{\cdots}{\Delta \cup \mathsf{var}(\bigwedge_{i\in I} t_i \to s_i) \fatsemi \Gamma, (x : t_i) \vdash_{\mathcal{I}} a : s'_i} \\ s'_i \sqsubseteq_{\Delta\cup\mathsf{var}(\bigwedge_{i\in I} t_i\to s_i)} s_i \end{cases}}{\Delta \fatsemi \Gamma \vdash_{\mathcal{I}} \lambda^{\wedge_{i\in I} t_i\to s_i} x.a : \bigwedge_{i\in I} t_i \to s_i}$$

Let $\Delta' = \Delta \cup \mathsf{var}(\bigwedge_{i\in I} t_i \to s_i)$ and $[\sigma_{j_i}]_{j_i\in J_i} \Vdash s'_i \sqsubseteq_{\Delta'} s_i$. By induction, there exists an expression e_i such that $\mathit{erase}(e_i) = a$ and $\Delta' \fatsemi \Gamma, (x : t_i) \vdash_{\mathcal{A}} e_i : s'_i$ for all $i \in I$. Since $\sigma_{j_i} \sharp \Delta'$, by (ALG-INST), we have $\Delta' \fatsemi \Gamma, (x : t_i) \vdash_{\mathcal{A}} e_i[\sigma_{j_i}]_{j_i\in J_i} : \bigwedge_{j_i\in J_i} s'_i\sigma_{j_i}$. Clearly, $e_i[\sigma_{j_i}]_{j_i\in J_i} \sharp \Delta'$ and $\mathit{erase}(e_i[\sigma_{j_i}]_{j_i\in J_i}) = \mathit{erase}(e_i) = a$. Then by Lemma 9.2.2, the intersection $\bigsqcap_{i\in I}(e_i[\sigma_{j_i}]_{j_i\in J_i})$ exists and we have $\mathit{erase}(\bigsqcap_{i\in I'}(e_i[\sigma_{j_i}]_{j_i\in J_i})) = a$ for any non-empty $I' \subseteq I$. Let $e = \bigsqcap_{i\in I}(e_i[\sigma_{j_i}]_{j_i\in J_i})$. According to Corollary 9.2.4, there exists a type t'_i such that $\Delta' \fatsemi \Gamma, (x : t_i) \vdash_{\mathcal{A}} e : t'_i$ and $t'_i \leq \bigwedge_{j_i\in J_i} s'_i\sigma_{j_i}$ for all $i \in I$. Moreover, since $t'_i \leq \bigwedge_{j_i\in J_i} s'_i\sigma_{j_i} \leq s_i$, by (ALG-ABSTR), we have $\Delta \fatsemi \Gamma \vdash_{\mathcal{A}} \lambda^{\wedge_{i\in I} t_i\to s_i} x.e : \bigwedge_{i\in I}(t_i \to s_i)$. Finally, according to Definition 9.1.2, we have

$$\mathit{erase}(\lambda^{\wedge_{i\in I} t_i\to s_i} x.e) = \lambda^{\wedge_{i\in I} t_i\to s_i} x.\mathit{erase}(e) = \lambda^{\wedge_{i\in I} t_i\to s_i} x.a$$

(INF-CASE-NONE): consider the derivation

$$\frac{\dfrac{\cdots}{\Delta \fatsemi \Gamma \vdash_{\mathcal{I}} a : t'} \quad t' \sqsubseteq_{\Delta} \mathbb{0}}{\Delta \fatsemi \Gamma \vdash_{\mathcal{I}} (a \in t \,?\, a_1 : a_2) : \mathbb{0}}$$

By induction, there exists an expression e such that $\mathit{erase}(e) = a$ and $\Delta \fatsemi \Gamma \vdash_{\mathcal{A}} e : t'$. Let $[\sigma_i]_{i\in I} \Vdash t' \sqsubseteq_{\Delta} \mathbb{0}$. Since $\sigma_i \sharp \Delta$, by (ALG-INST), we have $\Delta \fatsemi \Gamma \vdash_{\mathcal{A}} e[\sigma_i]_{i\in I} : \bigwedge_{i\in I} t'\sigma_i$. Let e_1 and e_2 be two expressions such that $\mathit{erase}(e_1) = a_1$ and $\mathit{erase}(e_2) = a_2$. Then we have

$$\mathit{erase}((e[\sigma_i]_{i\in I}) \in t \,?\, e_1 : e_2) = (a \in t \,?\, a_1 : a_2)$$

Moreover, since $\bigwedge_{i\in I} t'\sigma_i \leq \mathbb{0}$, by (ALG-CASE-NONE), we have

$$\Delta \fatsemi \Gamma \vdash_{\mathcal{A}} ((e[\sigma_i]_{i\in I}) \in t \,?\, e_1 : e_2) : \mathbb{0}$$

(INF-CASE-FST): consider the derivation

$$\frac{\dfrac{\ldots}{\Delta \fatsemi \Gamma \vdash_{\mathcal{I}} a : t'} \quad t' \sqsubseteq_\Delta t \quad t' \not\sqsubseteq_\Delta \neg t \quad \dfrac{\ldots}{\Delta \fatsemi \Gamma \vdash_{\mathcal{I}} a_1 : s}}{\Delta \fatsemi \Gamma \vdash_{\mathcal{I}} (a{\in}t\,?\,a_1 : a_2) : s}$$

By induction, there exist e, e_1 such that $\mathit{erase}(e) = a$, $\mathit{erase}(e_1) = a_1$, $\Delta \fatsemi \Gamma \vdash_{\mathcal{A}} e : t'$, and $\Delta \fatsemi \Gamma \vdash_{\mathcal{A}} e_1 : s$. Let $[\sigma_{i_1}]_{i_1 \in I_1} \Vdash t' \sqsubseteq_\Delta t$. Since $\sigma_{i_1} \sharp \Delta$, applying (ALG-INST), we get $\Delta \fatsemi \Gamma \vdash_{\mathcal{A}} e[\sigma_{i_1}]_{i_1 \in I_1} : \bigwedge_{i_1 \in I_1} t'\sigma_{i_1}$. Let e_2 be an expression such that $\mathit{erase}(e_2) = a_2$. Then we have

$$\mathit{erase}((e[\sigma_{i_1}]_{i_1 \in I_1}){\in}t\,?\,e_1 : e_2) = (a{\in}t\,?\,a_1 : a_2)$$

Finally, since $\bigwedge_{i_1 \in I_1} t'\sigma_{i_1} \leq t$, by (ALG-CASE-FST), we have

$$\Delta \fatsemi \Gamma \vdash_{\mathcal{A}} ((e[\sigma_{i_1}]_{i_1 \in I_1}){\in}t\,?\,e_1 : e_2) : s$$

(INF-CASE-SND): similar to the case of (INF-CASE-FST).

(INF-CASE-BOTH): consider the derivation

$$\frac{\dfrac{\ldots}{\Delta \fatsemi \Gamma \vdash_{\mathcal{I}} a : t'} \quad \begin{cases} t' \not\sqsubseteq_\Delta \neg t \quad \text{and} \quad \dfrac{\ldots}{\Delta \fatsemi \Gamma \vdash_{\mathcal{I}} a_1 : s_1} \\ t' \not\sqsubseteq_\Delta t \quad \text{and} \quad \dfrac{\ldots}{\Delta \fatsemi \Gamma \vdash_{\mathcal{I}} a_2 : s_2} \end{cases}}{\Delta \fatsemi \Gamma \vdash_{\mathcal{I}} (a{\in}t\,?\,a_1 : a_2) : s_1 \vee s_2}$$

By induction, there exist e, e_i such that $\mathit{erase}(e) = a$, $\mathit{erase}(e_i) = a_i$, $\Delta \fatsemi \Gamma \vdash_{\mathcal{A}} e : t'$, and $\Delta \fatsemi \Gamma \vdash_{\mathcal{A}} e_i : s_i$. According to Definition 9.1.2, we have

$$\mathit{erase}((e{\in}t\,?\,e_1 : e_2)) = (a{\in}t\,?\,a_1 : a_2)$$

Clearly $t' \not\simeq \mathbb{0}$. We claim that $t' \not\leq \neg t$. Let σ_{id} be any identity type substitution. If $t' \leq \neg t$, then $t'\sigma_{id} \simeq t' \leq \neg t$, i.e., $t' \sqsubseteq_\Delta \neg t$, which is in contradiction with $t' \not\sqsubseteq_\Delta \neg t$. Similarly we have $t' \not\leq t$. Therefore, by (ALG-CASE-SND), we have $\Delta \fatsemi \Gamma \vdash_{\mathcal{A}} (e{\in}t\,?\,e_1 : e_2) : s_1 \vee s_2$.

□

The proof of the soundness property constructs along the derivation for the implicitly-typed expression a some expression e that satisfies the statement of the theorem. We denote by $\mathit{erase}^{-1}(a)$ the set of expressions e that satisfy the statement. Notice that $\sqsubseteq_\Delta$ gauges the generality of the solutions found by the inference system: the smaller the type found, the more general the solution is. As a matter of facts, adding to the system in Figure 9.1 a subsumption rule that uses the relation $\sqsubseteq_\Delta$, that is:

$$\frac{\Delta \fatsemi \Gamma \vdash_{\mathcal{I}} a : t_1 \quad t_1 \sqsubseteq_\Delta t_2}{\Delta \fatsemi \Gamma \vdash_{\mathcal{I}} a : t_2} \text{ (INF-SUBSUM)}$$

is sound. This means that the set of solutions is upward closed with respect to $\sqsubseteq_\Delta$ and that from smaller solutions it is possible (by this new subsumption rule) to deduce the larger ones. In that respect, the completeness theorem that follows states that the inference system can always deduce for the erasure of an expression a solution that is at least as good as the one deduced for that expression by the type system for the explicitly typed calculus.

Theorem 9.2.6 (Completeness). *Let $e \in \mathscr{E}_0$ be an explicitly-typed expression. If $\Delta \fatsemi \Gamma \vdash_{\mathcal{A}} e : t$, then there exists a type t' such that $\Delta \fatsemi \Gamma \vdash_{\mathcal{I}} \mathit{erase}(e) : t'$ and $t' \sqsubseteq_\Delta t$.*

Proof. By induction on the typing derivation of $\Delta \fatsemi \Gamma \vdash_{\mathcal{A}} e : t$. We proceed by a case analysis on the last rule used in the derivation.

(ALG-CONST): take t' as b_c.
(ALG-VAR): take t' as $\Gamma(x)$.
(ALG-PAIR): consider the derivation

$$\frac{\dfrac{\cdots}{\Delta \fatsemi \Gamma \vdash_{\mathcal{A}} e_1 : t_1} \quad \dfrac{\cdots}{\Delta \fatsemi \Gamma \vdash_{\mathcal{A}} e_2 : t_2}}{\Delta \fatsemi \Gamma \vdash_{\mathcal{A}} (e_1, e_2) : t_1 \times t_2}$$

Applying the induction hypothesis twice, we have

$$\exists t'_1.\ \Delta \fatsemi \Gamma \vdash_{\mathcal{I}} \mathit{erase}(e_1) : t'_1 \text{ and } t'_1 \sqsubseteq_\Delta t_1$$
$$\exists t'_2.\ \Delta \fatsemi \Gamma \vdash_{\mathcal{I}} \mathit{erase}(e_2) : t'_2 \text{ and } t'_2 \sqsubseteq_\Delta t_2$$

Then by (INF-PAIR), we have

$$\Delta \fatsemi \Gamma \vdash_{\mathcal{I}} (\mathit{erase}(e_1), \mathit{erase}(e_2)) : t'_1 \times t'_2,$$

that is, $\Delta \fatsemi \Gamma \vdash_{\mathcal{I}} \mathit{erase}((e_1, e_2)) : t'_1 \times t'_2$. Finally, Applying Lemma 9.1.4, we have $(t'_1 \times t'_2) \sqsubseteq_\Delta (t_1 \times t_2)$.

(ALG-PROJ): consider the derivation

$$\frac{\dfrac{\cdots}{\Delta \fatsemi \Gamma \vdash_{\mathcal{A}} e : t} \quad t \leq \mathbb{1} \times \mathbb{1}}{\Delta \fatsemi \Gamma \vdash_{\mathcal{A}} \pi_i(e) : \boldsymbol{\pi}_i(t)}$$

By induction, we have

$$\exists t', [\sigma_k]_{k\in K}.\ \Delta \fatsemi \Gamma \vdash_{\mathcal{I}} \mathit{erase}(e) : t' \text{ and } [\sigma_k]_{k\in K} \Vdash t' \sqsubseteq_\Delta t$$

It is clear that $\bigwedge_{k\in K} t'\sigma_k \leq \mathbb{1} \times \mathbb{1}$. So $\boldsymbol{\pi}_i(\bigwedge_{k\in K} t'\sigma_k) \in \Pi^i_\Delta(t')$. Then by (INF-PROJ), we have

$$\Delta \fatsemi \Gamma \vdash_{\mathcal{I}} \pi_i(\mathit{erase}(e)) : \boldsymbol{\pi}_i(\bigwedge_{k\in K} t'\sigma_k),$$

that is, $\Delta \fatsemi \Gamma \vdash_{\mathcal{I}} \mathit{erase}(\pi_i(e)) : \boldsymbol{\pi}_i(\bigwedge_{k\in K} t'\sigma_k)$. According to Lemma 8.2.3, $t \leq (\boldsymbol{\pi}_1(t), \boldsymbol{\pi}_2(t))$. Then $\bigwedge_{k\in K} t'\sigma_k \leq (\boldsymbol{\pi}_1(t), \boldsymbol{\pi}_2(t))$. Finally, applying Lemma 8.2.3 again, we get $\boldsymbol{\pi}_i(\bigwedge_{k\in K} t'\sigma_k) \leq \boldsymbol{\pi}_i(t)$ and *a fortiori* $\boldsymbol{\pi}_i(\bigwedge_{k\in K} t'\sigma_k) \sqsubseteq_\Delta \boldsymbol{\pi}_i(t)$.

(ALG-APPL): consider the derivation

$$\frac{\dfrac{\cdots}{\Delta \fatsemi \Gamma \vdash_{\mathcal{A}} e_1 : t} \quad \dfrac{\cdots}{\Delta \fatsemi \Gamma \vdash_{\mathcal{A}} e_2 : s} \quad t \leq \mathbb{0} \to \mathbb{1} \quad s \leq \mathsf{dom}(t)}{\Delta \fatsemi \Gamma \vdash_{\mathcal{A}} e_1 e_2 : t \cdot s}$$

Applying the induction hypothesis twice, we have

$$\exists t'_1, [\sigma^1_k]_{k\in K_1}.\ \Delta \fatsemi \Gamma \vdash_{\mathcal{I}} \mathit{erase}(e_1) : t'_1 \text{ and } [\sigma^1_k]_{k\in K_1} \Vdash t'_1 \sqsubseteq_\Delta t$$
$$\exists t'_2, [\sigma^2_k]_{k\in K_2}.\ \Delta \fatsemi \Gamma \vdash_{\mathcal{I}} \mathit{erase}(e_2) : t'_2 \text{ and } [\sigma^2_k]_{k\in K_2} \Vdash t'_2 \sqsubseteq_\Delta s$$

It is clear that $\bigwedge_{k\in K_1} t'_1\sigma^1_k \leq \mathbb{0} \to \mathbb{1}$, that is, $\bigwedge_{k\in K_1} t'_1\sigma^1_k$ is a function type. So we get $\mathsf{dom}(t) \leq \mathsf{dom}(\bigwedge_{k\in K_1} t'_1\sigma^1_k)$. Then we have

$$\bigwedge_{k\in K_2} t'_2\sigma^2_k \leq s \leq \mathsf{dom}(t) \leq \mathsf{dom}(\bigwedge_{k\in K_1} t'_1\sigma^1_k)$$

Therefore, $(\bigwedge_{k\in K_1} t'_1\sigma^1_k)\cdot(\bigwedge_{k\in K_2} t'_2\sigma^2_k) \in t'_2 \bullet_\Delta t'_1$.
Then applying (Inf-appl), we have

$$\Delta \wr \Gamma \vdash_{\mathcal{I}} \mathit{erase}(e_1)\mathit{erase}(e_2) : (\bigwedge_{k\in K_1} t'_1\sigma^1_k)\cdot(\bigwedge_{k\in K_2} t'_2\sigma^2_k),$$

that is, $\Delta \wr \Gamma \vdash_{\mathcal{I}} \mathit{erase}(e_1e_2) : (\bigwedge_{k\in K_1} t'_1\sigma^1_k)\cdot(\bigwedge_{k\in K_2} t'_2\sigma^2_k)$. Moreover, from $\bigwedge_{k\in K_2} t'_2\sigma^2_k \leq \mathsf{dom}(t)$ we deduce that $t\cdot(\bigwedge_{k\in K_2} t'_2\sigma^2_k)$ exists. According to Lemma 8.2.11, we have

$$(\bigwedge_{k\in K_1} t'_1\sigma^1_k)\cdot(\bigwedge_{k\in K_2} t'_2\sigma^2_k) \leq t\cdot(\bigwedge_{k\in K_2} t'_2\sigma^2_k) \leq t\cdot s$$

Thus, $(\bigwedge_{k\in K_1} t'_1\sigma^1_k)\cdot(\bigwedge_{k\in K_2} t'_2\sigma^2_k) \sqsubseteq_\Delta t\cdot s$.
(Alg-abstr0)**:** consider the derivation

$$\frac{\forall i\in I.\ \dfrac{\cdots}{\Delta\cup\mathsf{var}(\bigwedge_{i\in I} t_i\to s_i) \wr \Gamma, (x:t_i)\vdash_{\mathcal{A}} e : s'_i} \text{ and } s'_i \leq s_i}{\Delta \wr \Gamma \vdash_{\mathcal{A}} \lambda^{\wedge_{i\in I} t_i\to s_i} x.e : \bigwedge_{i\in I} t_i \to s_i}$$

Let $\Delta' = \Delta \cup \mathsf{var}(\bigwedge_{i\in I} t_i \to s_i)$. By induction, for each $i \in I$, we have

$$\exists t'_i.\ \Delta' \wr \Gamma, (x:t_i) \vdash_{\mathcal{I}} \mathit{erase}(e) : t'_i \text{ and } t'_i \sqsubseteq_{\Delta'} s'_i$$

Clearly, we have $t'_i \sqsubseteq_{\Delta'} s_i$. By (Inf-abstr), we have

$$\Delta \wr \Gamma \vdash_{\mathcal{I}} \lambda^{\wedge_{i\in I} t_i\to s_i} x.\mathit{erase}(e) : \bigwedge_{i\in I} t_i \to s_i$$

that is, $\Delta \wr \Gamma \vdash_{\mathcal{I}} \mathit{erase}(\lambda^{\wedge_{i\in I} t_i\to s_i} x.e) : \bigwedge_{i\in I} t_i \to s_i$.
(Alg-case-none)**:** consider the derivation

$$\frac{\dfrac{\cdots}{\Delta \wr \Gamma \vdash_{\mathcal{A}} e : \mathbb{0}}}{\Delta \wr \Gamma \vdash_{\mathcal{A}} (e{\in}t\,?\,e_1 : e_2) : \mathbb{0}}$$

By induction, we have

$$\exists t'_0.\ \Delta \wr \Gamma \vdash_{\mathcal{I}} \mathit{erase}(e) : t'_0 \text{ and } t'_0 \sqsubseteq_\Delta \mathbb{0}$$

By (Inf-case-none), we have $\Delta \wr \Gamma \vdash_{\mathcal{I}} (\mathit{erase}(e){\in}t\,?\,\mathit{erase}(e_1) : \mathit{erase}(e_2)) : \mathbb{0}$, that is, $\Delta \wr \Gamma \vdash_{\mathcal{I}} \mathit{erase}(e{\in}t\,?\,e_1 : e_2) : \mathbb{0}$.

(ALG-CASE-FST): consider the derivation

$$\frac{\dfrac{\cdots}{\Delta \fatsemi \Gamma \vdash_{\mathcal{A}} e : t'} \quad t' \leq t \quad \dfrac{\cdots}{\Delta \fatsemi \Gamma \vdash_{\mathcal{A}} e_1 : s_1}}{\Delta \fatsemi \Gamma \vdash_{\mathcal{A}} (e{\in}t\,?\,e_1 : e_2) : s_1}$$

Applying the induction hypothesis twice, we have

$$\exists t'_0.\ \Delta \fatsemi \Gamma \vdash_{\mathcal{I}} \mathit{erase}(e) : t'_0 \text{ and } t'_0 \sqsubseteq_\Delta t'$$
$$\exists t'_1.\ \Delta \fatsemi \Gamma \vdash_{\mathcal{I}} \mathit{erase}(e_1) : t'_1 \text{ and } t'_1 \sqsubseteq_\Delta s_1$$

Clearly, we have $t'_0 \sqsubseteq_\Delta t$. If $t'_0 \sqsubseteq_\Delta \neg t$, then by Lemma 9.1.4, we have $t'_0 \leq_\Delta \mathbb{0}$. By (INF-CASE-NONE), we get

$$\Delta \fatsemi \Gamma \vdash_{\mathcal{I}} (\mathit{erase}(e){\in}t\,?\,\mathit{erase}(e_1) : \mathit{erase}(e_2)) : \mathbb{0}$$

that is, $\Delta \fatsemi \Gamma \vdash_{\mathcal{I}} \mathit{erase}(e{\in}t\,?\,e_1 : e_2) : \mathbb{0}$. Clearly, we have $\mathbb{0} \sqsubseteq_\Delta s_1$.
Otherwise, by (INF-CASE-FST), we have

$$\Delta \fatsemi \Gamma \vdash_{\mathcal{I}} (\mathit{erase}(e){\in}t\,?\,\mathit{erase}(e_1) : \mathit{erase}(e_2)) : t'_1$$

that is, $\Delta \fatsemi \Gamma \vdash_{\mathcal{I}} \mathit{erase}(e{\in}t\,?\,e_1 : e_2) : t'_1$. The result follows as well.
(ALG-CASE-SND): similar to the case of (ALG-CASE-FST).
(ALG-CASE-BOTH): consider the derivation

$$\frac{\dfrac{\cdots}{\Delta \fatsemi \Gamma \vdash_{\mathcal{A}} e : t'} \quad \begin{cases} t' \not\leq \neg t \quad \text{and} \quad \dfrac{\cdots}{\Delta \fatsemi \Gamma \vdash_{\mathcal{A}} e_1 : s_1} \\ t' \not\leq t \quad \text{and} \quad \dfrac{\cdots}{\Delta \fatsemi \Gamma \vdash_{\mathcal{A}} e_2 : s_2} \end{cases}}{\Delta \fatsemi \Gamma \vdash_{\mathcal{A}} (e{\in}t\,?\,e_1 : e_2) : s_1 \vee s_2}$$

By induction, we have

$$\exists t'_0.\ \Delta \fatsemi \Gamma \vdash_{\mathcal{I}} \mathit{erase}(e) : t'_0 \text{ and } t'_0 \sqsubseteq_\Delta t'$$
$$\exists t'_1.\ \Delta \fatsemi \Gamma \vdash_{\mathcal{I}} \mathit{erase}(e_1) : t'_1 \text{ and } t'_1 \sqsubseteq_\Delta s_1$$
$$\exists t'_2.\ \Delta \fatsemi \Gamma \vdash_{\mathcal{I}} \mathit{erase}(e_2) : t'_2 \text{ and } t'_2 \sqsubseteq_\Delta s_2$$

If $t'_0 \sqsubseteq_\Delta \mathbb{0}$, then by (INF-CASE-NONE), we get

$$\Delta \fatsemi \Gamma \vdash_{\mathcal{I}} (\mathit{erase}(e){\in}t\,?\,\mathit{erase}(e_1) : \mathit{erase}(e_2)) : \mathbb{0}$$

that is, $\Delta \fatsemi \Gamma \vdash_{\mathcal{I}} \mathit{erase}(e{\in}t\,?\,e_1 : e_2) : \mathbb{0}$. Clearly, we have $\mathbb{0} \sqsubseteq_\Delta s_1 \vee s_2$.
If $t'_0 \sqsubseteq_\Delta t$, then by (INF-CASE-FST), we get

$$\Delta \fatsemi \Gamma \vdash_{\mathcal{I}} (\mathit{erase}(e){\in}t\,?\,\mathit{erase}(e_1) : \mathit{erase}(e_2)) : t'_1$$

that is, $\Delta \fatsemi \Gamma \vdash_{\mathcal{I}} \mathit{erase}(e{\in}t\,?\,e_1 : e_2) : t'_1$. Moreover, it is clear that $t'_1 \sqsubseteq_\Delta s_1 \vee s_2$, the result follows as well.
Similarly for $t'_0 \sqsubseteq_\Delta \neg t$.
Otherwise, by (INF-CASE-BOTH), we have

$$\Delta \fatsemi \Gamma \vdash_{\mathcal{I}} (\mathit{erase}(e){\in}t\,?\,\mathit{erase}(e_1) : \mathit{erase}(e_2)) : t'_1 \vee t'_2$$

that is, $\Delta \wp \Gamma \vdash_{\mathcal{I}} \mathit{erase}(e{\in}t\,?\,e_1 : e_2) : t'_1 \vee t'_2$. Using α-conversion, we can assume that the polymorphic type variables of t'_1 and t'_2 (and of e_1 and e_2) are distinct, i.e., $(\mathsf{var}(t'_1) \setminus \Delta) \cap (\mathsf{var}(t'_1) \setminus \Delta) = \emptyset$. Then applying Lemma 9.1.5, we have $t'_1 \vee t'_2 \sqsubseteq_\Delta t_1 \vee t_2$.

(ALG-INST): consider the derivation

$$\frac{\overset{\cdots}{\Delta \wp \Gamma \vdash_{\mathcal{A}} e : t} \qquad \forall j \in J.\ \sigma_j \sharp \Delta \quad |J| > 0}{\Delta \wp \Gamma \vdash_{\mathcal{A}} e[\sigma_j]_{j\in J} : \bigwedge_{j\in J} t\sigma_j}$$

By induction, we have

$$\exists t', [\sigma_k]_{k\in K}.\ \Delta \wp \Gamma \vdash_{\mathcal{I}} \mathit{erase}(e) : t' \text{ and } [\sigma_k]_{k\in K} \Vdash t' \sqsubseteq_\Delta t$$

Since $\mathit{erase}(e[\sigma_j]_{j\in J}) = \mathit{erase}(e)$, we have $\Delta \wp \Gamma \vdash_{\mathcal{I}} \mathit{erase}(e[\sigma_j]_{j\in J}) : t'$. As $\bigwedge_{k\in K} t'\sigma_k \leq t$, we have $\bigwedge_{j\in J}(\bigwedge_{k\in K} t'\sigma_k)\sigma_j \leq \bigwedge_{j\in J} t\sigma_j$, that is $\bigwedge_{j\in J,k\in K} t'(\sigma_j \circ \sigma_k) \leq \bigwedge_{j\in J} t\sigma_j$. Moreover, it is clear that $\sigma_j \circ \sigma_k \sharp \Delta$. Therefore, we get $t' \sqsubseteq_\Delta \bigwedge_{j\in J} t\sigma_j$.

□

The inference system is syntax directed and describes an algorithm that is parametric in the decision procedures for $\sqsubseteq_\Delta$, $\amalg^i_\Delta(t)$ and $t \bullet_\Delta s$. The problem of deciding them is tackled in Chapter 10 .

Finally, notice that we did not give any reduction semantics for the implicitly typed calculus. The reason is that its semantics is defined in terms of the semantics of the explicitly-typed calculus: the relabeling at run-time is an essential feature —independently from the fact that we started from an explicitly typed expression or not— and we cannot avoid it. The (big-step) semantics for a is then given in expressions of $\mathit{erase}^{-1}(a)$: if an expression in $\mathit{erase}^{-1}(a)$ reduces to v, so does a. As we see the result of computing an implicitly-typed expression is a value of the explicitly typed calculus (so λ-abstractions may contain non-empty decorations) and this is unavoidable since it may be the result of a partial application. Also notice that the semantics is not deterministic since different expressions in $\mathit{erase}^{-1}(a)$ may yield different results. However this may happen only in one particular case, namely, when an occurrence of a polymorphic function flows into a type-case and its type is tested. For instance the application $(\lambda^{(\mathtt{Int}\to\mathtt{Int})\to\mathtt{Bool}} f.f{\in}\mathtt{Bool}\to\mathtt{Bool}\,?\,\mathtt{true} : \mathtt{false})(\lambda^{\alpha\to\alpha} x.x)$ results into `true` or `false` according to whether the polymorphic identity at the argument is instantiated by $[\{\mathtt{Int}/\alpha\}]$ or by $[\{\mathtt{Int}/\alpha\}, \{\mathtt{Bool}/\alpha\}]$. Once more this is unavoidable in a calculus that can dynamically test the types of polymorphic functions that admit several sound instantiations. This does not happen in practice since the inference algorithm always choose one particular instantiation (the existence of principal types would made this choice canonical and remove non determinism). So in practice the semantics is deterministic but implementation dependent.

In summary, programming in the implicitly-typed calculus corresponds to programming in the explicitly-typed one with the difference that we delegate to the system the task to write the type-substitutions for us and with the caveat that doing it makes the test of the type of a polymorphic function implementation dependent.

9.3 A more tractable inference system

With the rules of Figure 9.1, when type-checking an implicitly-typed expression, we have to compute sets of type substitutions for projections, applications, abstractions and type cases. Because type substitutions inference is a costly operation, we would like to perform it as seldom as possible. To this end, we give in this section a restricted version of the inference system, which is not complete but still sound and powerful enough to be used in practice.

First, we want to simplify the type inference rule for projections:

$$\frac{\Delta \fatsemi \Gamma \vdash_{\mathcal{I}} a : t \quad u \in \Pi^i_\Delta(t)}{\Delta \fatsemi \Gamma \vdash_{\mathcal{I}} \pi_i(a) : u}$$

where $\Pi^i_\Delta(t) = \{u \mid [\sigma_j]_{j\in J} \Vdash t \sqsubseteq_\Delta \mathbb{1} \times \mathbb{1},\ u = \boldsymbol{\pi}_i(\bigwedge_{j\in J} t\sigma_j)\}$. Instead of picking any type in $\Pi^i_\Delta(t)$, we would like to simply project t, i.e., assign the type $\boldsymbol{\pi}_i(t)$ to $\pi_i(a)$. By doing so, we lose completeness on pair types that contain top-level variables. For example, if $t = (\texttt{Int} \times \texttt{Int}) \wedge \alpha$, then $\texttt{Int} \wedge \texttt{Bool} \in \Pi^i_\Delta(t)$ (because α can be instantiated with $(\texttt{Bool} \times \texttt{Bool})$), but $\boldsymbol{\pi}_t(t) = \texttt{Int}$. We also lose typability if t is not a pair type, but can be instantiated in a pair type. For example, the type of $(\lambda^{\alpha\to(\alpha\vee((\beta\to\beta)\setminus(\texttt{Int}\to\texttt{Int})))}x.x)(42,3)$ is $(\texttt{Int} \times \texttt{Int}) \vee ((\beta \to \beta) \setminus (\texttt{Int} \to \texttt{Int}))$, which is not a pair type, but can be instantiated in $(\texttt{Int} \times \texttt{Int})$ by taking $\beta = \texttt{Int}$. We believe these kinds of types will not be written by programmers, and it is safe to use the following projection rule in practice.

$$\frac{\Delta \fatsemi \Gamma \vdash_{\mathcal{I}} a : t \qquad t \leq \mathbb{1} \times \mathbb{1}}{\Delta \fatsemi \Gamma \vdash_{\mathcal{I}} \pi_i(a) : \boldsymbol{\pi}_i(t)}\,(\textsc{Inf-proj'})$$

We now look at the type inference rules for the type case $a{\in}t\,?\,a_1 : a_2$. The four different rules consider the different possible instantiations that make the type t' inferred for a fit t or not. For the sake of simplicity, we decide not to infer type substitutions for polymorphic arguments of type cases. Indeed, in the expression $(\lambda^{\alpha\to\alpha}x.x) \in \texttt{Int} \to \texttt{Int}\ ?\ \texttt{true} : \texttt{false}$, we assume the programmer wants to do a type case on the polymorphic identity, and not on one of its instance (otherwise, he would have written the instantiated interface directly), so we do not try to instantiate it. And in any case there is no real reason for which the inference system should choose to instantiate the identity by $\texttt{Int}\to\texttt{Int}$ (and thus make the test succeed) rather than $\texttt{Bool}\to\texttt{Bool}$ (and thus make the test fail). If we decide not to infer types for polymorphic arguments of type-case expression, then since $\alpha \to \alpha$ is not a subtype of $\texttt{Int} \to \texttt{Int}$ (we have $\alpha \to \alpha \sqsubseteq_\emptyset \texttt{Int} \to \texttt{Int}$ but $\alpha \to \alpha \not\leq \texttt{Int} \to \texttt{Int}$) the expression evaluates to `false`. With this choice, we can merge the different inference rules into the following one.

$$\frac{\begin{array}{l}\Delta \fatsemi \Gamma \vdash_{\mathcal{I}} a : t' \quad t_1 = t' \wedge t \quad t_2 = t' \wedge \neg t \\ t_i \not\simeq \mathbb{0} \ \Rightarrow\ \Delta \fatsemi \Gamma \vdash_{\mathcal{I}} a_i : s_i\end{array}}{\Delta \fatsemi \Gamma \vdash_{\mathcal{I}} (a{\in}t\,?\,a_1 : a_2) : \bigvee_{t_i\not\simeq\mathbb{0}} s_i}\,(\textsc{Inf-case'})$$

Finally, consider the inference rule for abstractions:

$$\frac{\forall i \in I.\ \begin{cases} \Delta \cup \mathsf{var}(\bigwedge_{i\in I} t_i \to s_i) \fatsemi \Gamma, (x : t_i) \vdash_{\mathcal{I}} a : s'_i \\ s'_i \sqsubseteq_{\Delta \cup \mathsf{var}(\bigwedge_{i\in I} t_i \to s_i)} s_i \end{cases}}{\Delta \fatsemi \Gamma \vdash_{\mathcal{I}} \lambda^{\bigwedge_{i\in I} t_i \to s_i} x.a : \bigwedge_{i\in I} t_i \to s_i}$$

We verify that the abstraction can be typed with each arrow type $t_i \to s_i$ in the interface. Meanwhile, we also infer a set of type substitutions to tally the type s'_i we infer for the body expression with s_i. In practice, similarly, we expect that the abstraction is well-typed only if the type s'_i we infer for the body expression is a subtype of s_i. For example, the expression

$$\lambda^{\mathtt{Bool}\to(\mathtt{Int}\to\mathtt{Int})} x.x{\in}\mathtt{true}\,?\,(\lambda^{\alpha\to\alpha} y.y) : (\lambda^{\alpha\to\alpha} y.(\lambda^{\alpha\to\alpha} z.z)y)$$

is not well-typed while

$$\lambda^{\mathtt{Bool}\to(\alpha\to\alpha)} x.x{\in}\mathtt{true}\,?\,(\lambda^{\alpha\to\alpha} y.y) : (\lambda^{\alpha\to\alpha} y.(\lambda^{\alpha\to\alpha} z.z)y)$$

is well-typed. So we use the following restricted rule for abstractions instead:

$$\frac{\forall i \in I.\ \Delta \cup \mathsf{var}(\bigwedge_{i\in I} t_i \to s_i) \fatsemi \Gamma, (x : t_i) \vdash_{\mathcal{I}} a : s'_i \text{ and } s'_i \leq s_i}{\Delta \fatsemi \Gamma \vdash_{\mathcal{I}} \lambda^{\bigwedge_{i\in I} t_i \to s_i} x.a : \bigwedge_{i\in I} t_i \to s_i}\ (\textsc{Inf-abstr'})$$

In conclusion, we restrict the inference of type substitutions to applications. We give in Figure 9.2 the inference rules of the system which respects the above restrictions. With these new rules, the system remains sound, but it is not complete.

Theorem 9.3.1. *If $\Gamma \vdash_{\mathcal{I}} a : t$, then there exists an expression $e \in \mathscr{E}_0$ such that $\mathit{erase}(e) = a$ and $\Gamma \vdash_{\mathcal{A}} e : t$.*

Proof. Similar to the proof of Theorem 9.2.5. □

$$\frac{}{\Delta \fatsemi \Gamma \vdash_{\mathcal{I}} c : b_c}(\textsc{Inf-const}) \qquad \frac{}{\Delta \fatsemi \Gamma \vdash_{\mathcal{I}} x : \Gamma(x)}(\textsc{Inf-var})$$

$$\frac{\Delta \fatsemi \Gamma \vdash_{\mathcal{I}} a_1 : t_1 \qquad \Delta \fatsemi \Gamma \vdash_{\mathcal{I}} a_2 : t_2}{\Delta \fatsemi \Gamma \vdash_{\mathcal{I}} (a_1, a_2) : t_1 \times t_2}(\textsc{Inf-pair})$$

$$\frac{\Delta \fatsemi \Gamma \vdash_{\mathcal{I}} a : t \qquad t \leq \mathbb{1} \times \mathbb{1}}{\Delta \fatsemi \Gamma \vdash_{\mathcal{I}} \pi_i(a) : \boldsymbol{\pi}_i(t)}(\textsc{Inf-proj'})$$

$$\frac{\Delta \fatsemi \Gamma \vdash_{\mathcal{I}} a_1 : t \qquad \Delta \fatsemi \Gamma \vdash_{\mathcal{I}} a_2 : s \qquad u \in t \bullet_{\Delta} s}{\Delta \fatsemi \Gamma \vdash_{\mathcal{I}} a_1 a_2 : u}(\textsc{Inf-appl})$$

$$\frac{\forall i \in I.\ \Delta \cup \mathsf{var}(\bigwedge_{i \in I} t_i \to s_i) \fatsemi \Gamma, (x : t_i) \vdash_{\mathcal{I}} a : s'_i \text{ and } s'_i \leq s_i}{\Delta \fatsemi \Gamma \vdash_{\mathcal{I}} \lambda^{\wedge_{i \in I} t_i \to s_i} x.a : \bigwedge_{i \in I} t_i \to s_i}(\textsc{Inf-abstr'})$$

$$\frac{\begin{array}{l}\Delta \fatsemi \Gamma \vdash_{\mathcal{I}} a : t' \quad t_1 = t' \wedge t \quad t_2 = t' \wedge \neg t \\ t_i \not\simeq \mathbb{0} \ \Rightarrow\ \Delta \fatsemi \Gamma \vdash_{\mathcal{I}} a_i : s_i\end{array}}{\Delta \fatsemi \Gamma \vdash_{\mathcal{I}} (a \in t\, ?\, a_1 : a_2) : \bigvee_{t_i \not\simeq \mathbb{0}} s_i}(\textsc{Inf-case'})$$

Figure 9.2: Restricted type-substitution inference rules

Chapter 10

Type Tallying

All the decision procedures in the inference system presented in Chapter 9 correspond to solving the problem to check whether, given two types t and s, there exist pairs of sets of type-substitutions $[\sigma_i]_{i\in I}$ and $[\sigma_j]_{j\in J}$ such that $\bigwedge_{j\in J} s\sigma_j \leq \bigvee_{i\in I} t\sigma_i$[1]. The goal of this chapter is to solve this problem. We first define and solve a *type tallying* problem. We then explain how we can reduce the problem with fixed cardinalities of I and J to the type tallying problem, and provide a semi-algorithm for the original problem. Finally, we give some heuristics to establish upper bounds (which depend on t and s) for the cardinalities of I and J.

10.1 The type tallying problem

We first give some definitions about constraints.

Definition 10.1.1 (**Constraints**). *A* constraint (t, c, s) *is a triple belonging to* $\mathcal{T} \times \{\leq, \geq\} \times \mathcal{T}$. *Let* $\mathcal{C}$ *denote the set of all constraints. Given a constraint-set* $C \subseteq \mathcal{C}$, *the set of type variables occurring in* C *is defined as*

$$\mathsf{var}(C) = \bigcup_{(t,c,s)\in C} \mathsf{var}(t) \cup \mathsf{var}(s)$$

Definition 10.1.2 (**Normalized constraint**). *A constraint* (t, c, s) *is said to be* normalized *if* t *is a type variable. A constraint-set* $C \subseteq \mathcal{C}$ *is said to be normalized if every constraint* $(t, c, s) \in C$ *is normalized. Given a normalized constraint-set* C, *its domain is defined as* $\mathsf{dom}(C) = \{\alpha \mid \exists c, s.\ (\alpha, c, s) \in C\}$.

A type tallying problem and its solution are defined as follows:

Definition 10.1.3 (**Type tallying**). *Let* $C \subseteq \mathcal{C}$ *be a constraint-set and* Δ *a set of type variables. A type-substitution* σ *is a solution for the tallying problem of* C *and* Δ, *noted* $\sigma \Vdash_\Delta C$, *if* $\sigma \sharp \Delta$ *and*

$$\forall (t, \leq, s) \in C\ .\ t\sigma \leq s\sigma \text{ holds} \qquad \text{and} \qquad \forall (t, \geq, s) \in C\ .\ s\sigma \leq t\sigma \text{ holds.}$$

[1]For applications the problem should be $\bigwedge_{j\in J} s\sigma_j \leq \bigvee_{i\in I} \mathsf{dom}(t\sigma_i)$. However, we can conservatively consider it as $\bigwedge_{j\in J} s\sigma_j \leq \bigvee_{i\in I} \mathsf{dom}(t)\sigma_i$.

When Δ is empty, we write $\sigma \Vdash C$ for short.

We also define two operations on sets of constraint-sets:

Definition 10.1.4. *Given two sets of constraint-sets $\mathcal{S}_1, \mathcal{S}_2 \subseteq \mathscr{P}(\mathcal{C})$, we define their* union *as*

$$\mathcal{S}_1 \sqcup \mathcal{S}_2 = \mathcal{S}_1 \cup \mathcal{S}_2$$

and their intersection *as*

$$\mathcal{S}_1 \sqcap \mathcal{S}_2 = \{C_1 \cup C_2 \mid C_1 \in \mathcal{S}_1, C_2 \in \mathcal{S}_2\}$$

The tallying solving algorithm can be logically decomposed in 3 steps. Let us examine each step of the algorithm on some examples.

Step 1: normalize each constraint.

Because normalized constraints are easier to solve than regular ones, we first turn each constraint into an equivalent set of normalized constraint-sets according to the decomposition rules used in the subtyping algorithm (*i.e.*, Lemmas 4.3.9 and 4.3.10). For example, the constraint $c_1 = (\alpha \times \alpha) \leq ((\texttt{Int} \times \mathbb{1}) \times (\mathbb{1} \times \texttt{Int}))$ can be normalized into the set $\mathcal{S}_1 = \{\{(\alpha, \leq, \mathbb{0})\}; \{(\alpha, \leq, (\texttt{Int} \times \mathbb{1})), (\alpha, \leq, (\mathbb{1} \times \texttt{Int}))\}\}$. Another example is the constraint $c_2 = ((\beta \times \beta) \to (\texttt{Int} \times \texttt{Int}), \leq, \alpha \to \alpha)$, which is equivalent to the following set of normalized constraint-sets $\mathcal{S}_2 = \{\{(\alpha, \leq, \mathbb{0})\}; \{(\alpha, \leq, (\beta \times \beta)), (\alpha, \geq, (\texttt{Int} \times \texttt{Int}))\}\}$. Then we join all the sets of constraint-sets by (constraint-set) intersections, yielding the normalization of the original constraint-set. For instance, the normalization $\mathcal{S}$ of $\{c_1, c_2\}$ is $\mathcal{S}_1 \sqcap \mathcal{S}_2$. It is easy to see that the constraint-set $C_1 = \{(\alpha, \leq, (\texttt{Int} \times \mathbb{1})), (\alpha, \leq, (\mathbb{1} \times \texttt{Int})), (\alpha, \leq, (\beta \times \beta)), (\alpha, \geq, (\texttt{Int} \times \texttt{Int}))\}$ is in $\mathcal{S}$ (see Definition 10.1.4).

Step 2: saturate each constraint-set.

Step 2.1: merge the constraints with the same type variables.

In each constraint-set of the normalization of the original constraint-set, there may be several constraints of the form $(\alpha, \geq, t_i)$ (resp. $(\alpha, \leq, t_i)$), which give different lower bounds (resp. upper bounds) for α. We merge all these constraints into one using unions (resp. intersections). For example, the constraint-set C_1 of the previous step can be merged as $C_2 = \{(\alpha, \leq, (\texttt{Int} \times \mathbb{1}) \wedge (\mathbb{1} \times \texttt{Int}) \wedge (\beta \times \beta)), (\alpha, \geq, (\texttt{Int} \times \texttt{Int}))\}$, which is equivalent to $\{(\alpha, \leq, (\texttt{Int} \wedge \beta \times \texttt{Int} \wedge \beta)), (\alpha, \geq, (\texttt{Int} \times \texttt{Int}))\}$.

Step 2.2: saturate the lower and upper bounds of a same type variable.

If a type variable has both a lower bound s and an upper bound t in a constraint-set, then the solutions we are looking for must satisfy the constraint $(s, \leq, t)$ as well. Therefore, we have to saturate the constraint-set with $(s, \leq, t)$, which has to be normalized, merged, and saturated itself first. Take C_2 for example. We have to saturate C_2 with $((\texttt{Int} \times \texttt{Int}), \leq, (\texttt{Int} \wedge \beta \times \texttt{Int} \wedge \beta))$, whose normalization is $\{\{(\beta, \geq, \texttt{Int})\}\}$. Thus, the saturation of C_2 is $\{C_2\} \sqcap \{\{(\beta, \geq, \texttt{Int})\}\}$, which contains only one constraint-set $C_3 = \{(\alpha, \leq, (\texttt{Int} \wedge \beta \times \texttt{Int} \wedge \beta)), (\alpha, \geq, (\texttt{Int} \times \texttt{Int})), (\beta, \geq, \texttt{Int})\}$.

Step 3: extract a substitution from each constraint-set.

Step 3.1**:** transform each constraint-set into an equation system.
To transform constraints into equations, we use the property that some set of constraints is satisfied for all assignments of α included between s and t if and only if the same set in which we replace α by $(s \vee \alpha') \wedge t$[2] is satisfied for all possible assignments of α' (with α' fresh). Of course such a transformation works only if $s \leq t$, but remember that we "checked" that this holds at the moment of the saturation. By performing this replacement for each variable we obtain a system of equations. For example, the constraint set C_3 is equivalent to the following equation system E:

$$\begin{aligned} \alpha = &\quad ((\texttt{Int} \times \texttt{Int}) \vee \alpha') \wedge (\texttt{Int} \wedge \beta \times \texttt{Int} \wedge \beta) \\ \beta = &\quad \texttt{Int} \vee \beta' \end{aligned}$$

where α', β' are fresh type variables.

Step 3.2**:** solve each equation system.
Finally, using the Courcelle's work on infinite trees [Cou83], we solve each equation system, which gives us a substitution which is a solution of the original constraint-set. For example, we can solve the equation system E, yielding $\{(\texttt{Int} \times \texttt{Int})/\alpha, \texttt{Int} \vee \beta'/\beta\}$, which is a solution of C_3 and thus of $\{c_1, c_2\}$.

In the following sections we study in details each step of the algorithm.

10.1.1 Constraint normalization

The type tallying problem is quite similar to the subtyping problem presented in Chapter 3 . We therefore reuse most of the technology developed in Chapter 3 such as, for example, the transformation of the subtyping problem into an emptiness decision problem, the elimination of top-level constructors, and so on. One of the main differences is that we do not want to eliminate top-level type variables from constraints, but, rather, we want to isolate them to build sets of normalized constraints (from which we then construct sets of substitutions).

In general, normalizing a constraint generates a set of constraints. For example, $(\alpha \vee \beta, \geq, \mathbb{0})$ holds if and only if $(\alpha, \geq, \mathbb{0})$ or $(\beta, \geq, \mathbb{0})$ holds; therefore the constraint $(\alpha \vee \beta, \geq, \mathbb{0})$ is equivalent to the normalized constraint-set $\{(\alpha, \geq, \mathbb{0}), (\beta, \geq, \mathbb{0})\}$. Consequently, the normalization of a constraint-set C yields a set $\mathcal{S}$ of normalized constraint-sets.

Several normalized sets may be suitable replacements for a given constraint; for example, $\{(\alpha, \leq, \beta \vee t_1), (\beta, \leq, \alpha \vee t_2)\}$ and $\{(\alpha, \leq, \beta \vee t_1), (\alpha, \geq, \beta \setminus t_2)\}$ are clearly equivalent normalized sets. However, the equation systems generated by the algorithm for these two sets are completely different equation systems and yield different substitutions. Concretely, $\{(\alpha, \leq, \beta \vee t_1), (\beta, \leq, \alpha \vee t_2)\}$ generates the equation system $\{\alpha = \alpha' \wedge (\beta \vee t_1), \beta = \beta' \wedge (\alpha \vee t_2)\}$, which in turn gives the substitution σ_1 such that

$$\begin{aligned} \sigma_1(\alpha) &= \mu x.\ ((\alpha' \wedge \beta' \wedge x) \vee (\alpha' \wedge \beta' \wedge t_2) \vee (\alpha' \wedge t_1)) \\ \sigma_1(\beta) &= \mu x.\ ((\beta' \wedge \alpha' \wedge x) \vee (\beta' \wedge \alpha' \wedge t_1) \vee (\beta' \wedge t_2)) \end{aligned}$$

[2]Or by $s \vee (\alpha' \wedge t)$.

where α' and β' are fresh type variables (see Section 10.1.4 for more details). These recursive types are not valid in our calculus, because x does not occur under a type constructor (this means that the unfolding of the type does not satisfy the property that every infinite branch contains infinitely many occurrences of type constructors). In contrast, the equation system built from $\{(\alpha, \leq, \beta \vee t_1), (\alpha, \geq, \beta \setminus t_2)\}$ is $\alpha = ((\beta \setminus t_2) \vee \alpha') \wedge (\beta \vee t_1)$, and the corresponding substitution is $\sigma_2 = \{((\beta \setminus t_2) \vee \alpha') \wedge (\beta \vee t_1)/\alpha\}$, which is valid since it maps the type variable α into a well-formed type. Ill-formed recursive types are generated when there exists a chain $\alpha_0 = \alpha_1\ B_1\ t_1, \ldots, \alpha_i = \alpha_{i+1}\ B_{i+1}\ t_{i+1}, \ldots, \alpha_n = \alpha_0\ B_{n+1}\ t_{n+1}$ (where $B_i \in \{\wedge, \vee\}$ for all i, and $n \geq 0$) in the equation system built from the normalized constraint-set. This chain implies the equation $\alpha_0 = \alpha_0\ B\ t'$ for some $B \in \{\wedge, \vee\}$ and t', and the corresponding solution for α_0 will be an ill-formed recursive type. To avoid this issue, we give an arbitrary ordering on type variables occurring in the constraint-set C such that different type variables have different orders. Then we always select the normalized constraint (α, c, t) such that the order of α is smaller than all the orders of the top-level type variables in t. As a result, the transformed equation system does not contain any problematic chain like the one above.

Definition 10.1.5 (Ordering). *Let V be a set of type variables. An* ordering *O on V is an injective map from V to $\mathbb{N}$.*

We formalize normalization as a judgement $\Sigma \vdash_{\mathcal{N}} C \rightsquigarrow \mathcal{S}$, which states that under the environment Σ (which, informally, contains the types that have already been processed at this point), C is normalized to $\mathcal{S}$. The judgement is derived according the rules of Figure 10.1.

If the constraint-set is empty, then clearly any substitution is a solution, and, the result of the normalization is simply the singleton containing the empty set (rule (NEMPTY)). Otherwise, each constraint is normalized separately, and the normalization of the constraint-set is the intersection of the normalizations of each constraint (rule (NJOIN)). By using rules (NSYM), (NZERO), and (NDNF) repeatedly, we transform any constraint into the constraint of the form $(\tau, \leq, \mathbb{0})$ where τ is disjunctive normal form: the first rule reverses $(t', \geq, t)$ into $(t, \leq, t')$, the second rule moves the type t' from the right of $\leq$ to the left, yielding $(t \wedge \neg t', \leq, \mathbb{0})$, and finally the last rule puts $t \wedge \neg t'$ in disjunctive normal form. Such a type τ is the type to be normalized. If τ is a union of single normal forms, the rule (NUNION) splits the union of single normal forms into constraints featuring each of the single normal forms. Then the results of each constraint normalization are joined by the rule (NJOIN).

The following rules handle constraints of the form $(\tau, \leq, \mathbb{0})$, where τ is a single normal form. If there are some top-level type variables, the rule (NTLV) generates a normalized constraint for the top-level type variable whose order is the smallest. Otherwise, there are no top-level type variables. If τ has already been normalized (i.e., it belongs to Σ), then it is not processed again (rule (NHYP)). Otherwise, we memoize it and then process it using the predicate for single normal forms $\Sigma \vdash^*_{\mathcal{N}} C \rightsquigarrow \mathcal{S}$ (rule (NASSUM)). Note that switching from $\Sigma \vdash_{\mathcal{N}} C \rightsquigarrow \mathcal{S}$ to $\Sigma \vdash^*_{\mathcal{N}} C \rightsquigarrow \mathcal{S}$ prevents the incorrect use of (NHYP) just after (NASSUM), which would wrongly say that any type is normalized without doing any computation.

Finally, the last four rules state how to normalize constraints of the form $(\tau, \leq, \mathbb{0})$

$$\frac{}{\Sigma \vdash_{\mathcal{N}} \emptyset \rightsquigarrow \{\emptyset\}}(\textsc{Nempty}) \qquad \frac{\Sigma \vdash_{\mathcal{N}} \{(t_i\ c_i\ t_i')\} \rightsquigarrow S_i}{\Sigma \vdash_{\mathcal{N}} \{(t_i\ c_i\ t_i') \mid i \in I\} \rightsquigarrow \prod_{i \in I} S_i}(\textsc{Njoin})$$

$$\frac{\Sigma \vdash_{\mathcal{N}} \{(t \leq t')\} \rightsquigarrow \mathcal{S}}{\Sigma \vdash_{\mathcal{N}} \{(t' \geq t)\} \rightsquigarrow \mathcal{S}}(\textsc{Nsym}) \qquad \frac{\Sigma \vdash_{\mathcal{N}} \{(t \wedge \neg t' \leq \mathbb{0})\} \rightsquigarrow \mathcal{S} \qquad t' \neq \mathbb{0}}{\Sigma \vdash_{\mathcal{N}} \{(t \leq t')\} \rightsquigarrow \mathcal{S}}(\textsc{Nzero})$$

$$\frac{\Sigma \vdash_{\mathcal{N}} \{(\mathsf{dnf}(t) \leq \mathbb{0})\} \rightsquigarrow \mathcal{S}}{\Sigma \vdash_{\mathcal{N}} \{(t \leq \mathbb{0})\} \rightsquigarrow \mathcal{S}}(\textsc{Ndnf}) \qquad \frac{\Sigma \vdash_{\mathcal{N}} \{(\tau_i \leq \mathbb{0}) \mid i \in I\} \rightsquigarrow \mathcal{S}}{\Sigma \vdash_{\mathcal{N}} \{(\bigvee_{i \in I} \tau_i \leq \mathbb{0})\} \rightsquigarrow \mathcal{S}}(\textsc{Nunion})$$

$$\frac{\mathsf{tlv}(\tau_0) = \emptyset \qquad \alpha' \not\downarrow_O P \cup N \qquad \mathcal{S} = \begin{cases} \{\{(\alpha', \leq, \neg t_{\alpha'})\}\} & \alpha' \in P \\ \{\{(\alpha', \geq, t_{\alpha'})\}\} & \alpha' \in N \end{cases}}{\Sigma \vdash_{\mathcal{N}} \{(\bigwedge_{\alpha \in P} \alpha \wedge \bigwedge_{\alpha \in N} \neg\alpha \wedge \tau_0 \leq \mathbb{0})\} \rightsquigarrow \mathcal{S}}(\textsc{Ntlv})$$

$$\frac{\tau_0 \in \Sigma \qquad \mathsf{tlv}(\tau_0) = \emptyset}{\Sigma \vdash_{\mathcal{N}} \{(\tau_0 \leq \mathbb{0})\} \rightsquigarrow \{\emptyset\}}(\textsc{Nhyp}) \qquad \frac{\Sigma \cup \{\tau_0\} \vdash^*_{\mathcal{N}} \{(\tau_0 \leq \mathbb{0})\} \rightsquigarrow \mathcal{S} \qquad \tau_0 \notin \Sigma \qquad \mathsf{tlv}(\tau_0) = \emptyset}{\Sigma \vdash_{\mathcal{N}} \{(\tau_0 \leq \mathbb{0})\} \rightsquigarrow \mathcal{S}}(\textsc{Nassum})$$

$$\frac{\bigwedge_{i \in P} b_i \wedge \bigwedge_{j \in N} \neg b_j \leq \mathbb{0}}{\Sigma \vdash^*_{\mathcal{N}} \{(\bigwedge_{i \in P} b_i \wedge \bigwedge_{j \in N} \neg b_j \leq \mathbb{0})\} \rightsquigarrow \{\emptyset\}}(\textsc{Nbasic-T}) \qquad \frac{\bigwedge_{i \in P} b_i \wedge \bigwedge_{j \in N} \neg b_j \not\leq \mathbb{0}}{\Sigma \vdash^*_{\mathcal{N}} \{(\bigwedge_{i \in P} b_i \wedge \bigwedge_{j \in N} \neg b_j \leq \mathbb{0})\} \rightsquigarrow \emptyset}(\textsc{Nbasic-F})$$

$$\frac{\forall N' \subseteq N. \begin{cases} \Sigma \vdash_{\mathcal{N}} \{\bigwedge_{i \in P} t_i^1 \wedge \bigwedge_{j \in N'} \neg t_j^1 \leq \mathbb{0}\} \rightsquigarrow \mathcal{S}^1_{N'} \\ \Sigma \vdash_{\mathcal{N}} \{\bigwedge_{i \in P} t_i^2 \wedge \bigwedge_{j \in N \setminus N'} \neg t_j^2 \leq \mathbb{0}\} \rightsquigarrow \mathcal{S}^2_{N'} \end{cases}}{\Sigma \vdash^*_{\mathcal{N}} \{(\bigwedge_{i \in P} (t_i^1 \times t_i^2) \wedge \bigwedge_{j \in N} \neg(t_j^1 \times t_j^2) \leq \mathbb{0})\} \rightsquigarrow \prod_{N' \subseteq N} (\mathcal{S}^1_{N'} \sqcup \mathcal{S}^2_{N'})}(\textsc{Nprod})$$

$$\frac{\exists j \in N\ \forall P' \subseteq P. \begin{cases} \Sigma \vdash_{\mathcal{N}} \{t_j^1 \wedge \bigwedge_{i \in P'} \neg t_i^1 \leq \mathbb{0}\} \rightsquigarrow \mathcal{S}^1_{P'} \\ \begin{cases} \Sigma \vdash_{\mathcal{N}} \{\bigwedge_{i \in P \setminus P'} t_i^2 \wedge \neg t_j^2 \leq \mathbb{0}\} \rightsquigarrow \mathcal{S}^2_{P'} & P' \neq P \\ \mathcal{S}^2_{N'} = \emptyset & \text{otherwise} \end{cases} \end{cases}}{\Sigma \vdash^*_{\mathcal{N}} \{(\bigwedge_{i \in P} (t_i^1 \to t_i^2) \wedge \bigwedge_{j \in N} \neg(t_j^1 \to t_j^2) \leq \mathbb{0})\} \rightsquigarrow \bigsqcup_{j \in N} \prod_{P' \subseteq P} (\mathcal{S}^1_{P'} \sqcup \mathcal{S}^2_{P'})}(\textsc{Narrow})$$

where t and t with scripts are types, c_i belongs to $\{\leq, \geq\}$, τ_0 and τ_i are single normal forms, $\alpha \not\downarrow_O P \cup N$ denotes α has the smallest order in $P \cup N$ under the ordering O, and $t_{\alpha'}$ is the type obtained from $\bigwedge_{\alpha \in P} \alpha \wedge \bigwedge_{\alpha \in N} \neg\alpha \wedge \tau_0$ by eliminating α'.

Figure 10.1: Normalization rules

where τ is a single normal form and contains no top-level type variables. Thereby τ should be an intersection of atoms with the same constructor. If τ is an intersection of basic types, normalizing is equivalent to checking whether τ is empty or not: if it is (rule (Nbasic-T)), we return the singleton containing the empty set (any substitution is a solution), otherwise there is no solution and we return the empty set (rule (Nbasic-F)). When τ is an intersection of products, the rule (Nprod) decomposes τ into several candidate types (following Lemma 4.3.9), which are to be further normalized. The case

when τ is an intersection of arrows (rule (NARROW)) is treated similarly. Note that, in the last two rules, we switch from $\Sigma \vdash^*_{\mathcal{N}} C \rightsquigarrow \mathcal{S}$ back to $\Sigma \vdash_{\mathcal{N}} C \rightsquigarrow \mathcal{S}$ in the premises to ensure termination.

If $\emptyset \vdash_{\mathcal{N}} C \rightsquigarrow \mathcal{S}$, then $\mathcal{S}$ is the result of the normalization of C. We now prove soundness, completeness, and termination of the constraint normalization algorithm.

To prove soundness, we use a family of subtyping relations $\leq_n$ that layer $\leq$ (*i.e.*, such that $\bigcup_{n\in\mathcal{N}} \leq_n = \leq$) and a family of satisfaction predicates $\Vdash_n$ that layer $\Vdash$ (*i.e.*, such that $\bigcup_{n\in\mathcal{N}} \Vdash_n = \Vdash$), which are defined as follows.

Definition 10.1.6. *Let $\leq$ be the subtyping relation induced by a well-founded convex model with infinite support $(\llbracket \_ \rrbracket, \mathcal{D})$. We define the family $(\leq_n)_{n\in\mathcal{N}}$ of subtyping relations as*

$$t \leq_n s \stackrel{\text{def}}{\iff} \forall \eta.\ \llbracket t \rrbracket_n \eta \subseteq \llbracket s \rrbracket_n \eta$$

where $\llbracket \_ \rrbracket_n$ is the rank n interpretation of a type, defined as

$$\llbracket t \rrbracket_n \eta = \{ d \in \llbracket t \rrbracket \eta \mid \mathsf{height}(d) \leq n \}$$

and $\mathsf{height}(d)$ is the height of an element d in $\mathcal{D}$, defined as

$$\begin{aligned} \mathsf{height}(c) &= 1 \\ \mathsf{height}((d, d')) &= max(\mathsf{height}(d), \mathsf{height}(d')) + 1 \\ \mathsf{height}(\{(d_1, d'_1), \ldots, (d_n, d'_n)\}) &= \begin{cases} 1 & n = 0 \\ max(\mathsf{height}(d_i), \mathsf{height}(d'_i), \ldots) + 1 & n > 0 \end{cases} \end{aligned}$$

Lemma 10.1.7. *Let $\leq$ be the subtyping relation induced by a well-founded convex model with infinite support. Then*

(1) *$t \leq_0 s$ for all $t, s \in \mathcal{T}$.*

(2) *$t \leq s \iff \forall n.\ t \leq_n s$.*

(3)
$$\bigwedge_{i\in I}(t_i \times s_i) \leq_{n+1} \bigvee_{j\in J}(t_j \times s_j) \iff \forall J' \subseteq J\ .\ \begin{cases} \bigwedge_{i\in I} t_i \leq_n \bigvee_{j\in J'} t_j \\ \vee \\ \bigwedge_{i\in I} s_i \leq_n \bigvee_{j\in J\setminus J'} s_j \end{cases}$$

(4)
$$\bigwedge_{i\in I}(t_i \to s_i) \leq_{n+1} \bigvee_{j\in J}(t_j \to s_j) \iff \exists j_0 \in J\,.\,\forall I' \subseteq I\,.\ \begin{cases} t_{j_0} \leq_n \bigvee_{i\in I'} t_i \\ \vee \\ \begin{cases} I \neq I' \\ \wedge \\ \bigwedge_{i\in I\setminus I'} s_i \leq_n s_{j_0} \end{cases} \end{cases}$$

Proof. (1) straightforward.

(2) straightforward.

(3) the result follows by Lemma 4.3.9 and Definition 10.1.6.

(4) the result follows by Lemma 4.3.10 and Definition 10.1.6.

□

Definition 10.1.8. *Given a constraint-set C and a type substitution σ, we define the rank n satisfaction predicate $\Vdash_n$ as*

$$\sigma \Vdash_n C \stackrel{\text{def}}{\iff} \forall (t, \leq, s) \in C.\ t\ \leq_n\ s \text{ and } \forall (t, \geq, s) \in C.\ s\ \leq_n\ t$$

Lemma 10.1.9. *Let $\leq$ be the subtyping relation induced by a well-founded convex model with infinite support. Then*

(1) *$\sigma \Vdash_0 C$ for all σ and C.*

(2) *$\sigma \Vdash C \iff \forall n.\ \sigma \Vdash_n C$.*

Proof. Consequence of Lemma 10.1.7. □

Given a set of types Σ, we write $C(\Sigma)$ for the constraint-set $\{(t, \leq, \mathbb{0}) \mid t \in \Sigma\}$.

Lemma 10.1.10 (Soundness). *Let C be a constraint-set. If $\emptyset \vdash_{\mathcal{N}} C \rightsquigarrow \mathcal{S}$, then for all normalized constraint-set $C' \in \mathcal{S}$ and all substitution σ, we have $\sigma \Vdash C' \Rightarrow \sigma \Vdash C$.*

Proof. We prove the following stronger statements.

(1) Assume $\Sigma \vdash_{\mathcal{N}} C \rightsquigarrow \mathcal{S}$. For all $C' \in \mathcal{S}$, σ and n, if $\sigma \Vdash_n C(\Sigma)$ and $\sigma \Vdash_n C'$, then $\sigma \Vdash_n C$.

(2) Assume $\Sigma \vdash^*_{\mathcal{N}} C \rightsquigarrow \mathcal{S}$. For all $C' \in \mathcal{S}$, σ and n, if $\sigma \Vdash_n C(\Sigma)$ and $\sigma \Vdash_n C'$, then $\sigma \Vdash_{n+1} C$.

Before proving these statements, we explain how the first property implies the lemma. Suppose $\emptyset \vdash_{\mathcal{N}} C \rightsquigarrow \mathcal{S}$, $C' \in \mathcal{S}$ and $\sigma \Vdash C'$. It is easy to check that $\sigma \Vdash_n C(\emptyset)$ holds for all n. From $\sigma \Vdash C'$, we deduce $\sigma \Vdash_n C'$ for all n (by Lemma 10.1.9). By Property (1), we have $\sigma \Vdash_n C$ for all n, and we have then the required result by Lemma 10.1.9.

We prove the two properties simultaneously by induction on the derivations of $\Sigma \vdash_{\mathcal{N}} C \rightsquigarrow \mathcal{S}$ and $\Sigma \vdash^*_{\mathcal{N}} C \rightsquigarrow \mathcal{S}$.

(NEMPTY): straightforward.
(NJOIN): according to Definition 10.1.4, if there exists $C_i \in \mathcal{S}_i$ such that $C_i = \emptyset$, then $\bigsqcap_{i \in I} \mathcal{S}_i = \emptyset$, and the result follows immediately. Otherwise, we have $C' = \bigcup_{i \in I} C_i$, where $C_i \in \mathcal{S}_i$. As $\sigma \Vdash_n C'$, then clearly $\sigma \Vdash_n C_i$. By induction, we have $\sigma \Vdash_n \{(t_i\ c_i\ t'_i)\}$. Therefore, we get $\sigma \Vdash_n \{(t_i\ c_i\ t'_i) \mid i \in I\}$.
(NSYM): by induction, we have $\sigma \Vdash_n \{(t \leq t')\}$. Then clearly $\sigma \Vdash_n \{(t' \geq t)\}$.
(NZERO): by induction, we have $\sigma \Vdash_n \{(t \wedge \neg t' \leq \mathbb{0})\}$. According to set-theory, we have $\sigma \Vdash_n \{(t \leq t')\}$.
(NDNF): similar to the case of (NZERO).
(NUNION): similar to the case of (NZERO).

(NTLV): assume α' has the smallest order in $P \cup N$. If $\alpha' \in P$, then we have $C' = (\alpha', \leq, \neg t_{\alpha'})$. From $\sigma \Vdash_n C'$, we deduce $\sigma(\alpha') \leq_n \neg t_{\alpha'}\sigma$. According to set-theory, we have $\sigma(\alpha') \wedge t_{\alpha'}\sigma \leq_n \mathbb{0}$, that is, $\sigma \Vdash_n \{(\bigwedge_{\alpha \in P} \alpha \wedge \bigwedge_{\alpha \in N} \neg\alpha \wedge \tau_0 \leq \mathbb{0})\}$. Otherwise, we have $\alpha' \in N$ and the result follows as well.

(NHYP): since we have $\tau_0 \in \Sigma$ and $\sigma \Vdash_n C(\Sigma)$, then $\sigma \Vdash_n \{(\tau_0 \leq \mathbb{0})\}$ holds.

(NASSUM): if $n = 0$, then $\sigma \Vdash_0 \{(\tau_0 \leq \mathbb{0})\}$ holds. Suppose $n > 0$. From $\sigma \Vdash_n C(\Sigma)$ and $\sigma \Vdash_k C'$, it is easy to prove that $\sigma \Vdash_k C(\Sigma)$ (*) and $\sigma \Vdash_k C'$ (**) hold for all $0 \leq k \leq n$. We now prove that $\sigma \Vdash_k \{(\tau_0 \leq \mathbb{0})\}$ (***) holds for all $1 \leq k \leq n$. By definition of $\Vdash_0$, we have $\sigma \Vdash_0 C(\Sigma \cup \{\tau_0\})$ and $\sigma \Vdash_0 C'$. Consequently, by the induction hypothesis (item (2)), we have $\sigma \Vdash_1 \{\tau_0 \leq \mathbb{0}\}$. From this and (*), we deduce $\sigma \Vdash_1 C(\Sigma \cup \{\tau_0\})$. Because we also have $\sigma \Vdash_1 C'$ (by (**)), we can use the induction hypothesis (item (2)) again to deduce $\sigma \Vdash_2 \{(\tau_0 \leq \mathbb{0})\}$. Hence, we can prove (***) by induction on $1 \leq k \leq n$. In particular, we have $\sigma \Vdash_n \{(\tau_0 \leq \mathbb{0})\}$, which is the required result.

(NBASIC): straightforward.

(NPROD): If $\bigsqcap_{N' \subseteq N}(\mathcal{S}^1_{N'} \sqcup \mathcal{S}^2_{N'})$ is $\emptyset$, then the result follows straightforwardly. Otherwise, we have $C' = \bigcup_{N' \subseteq N} C_{N'}$, where $C_{N'} \in (\mathcal{S}^1_{N'} \sqcup \mathcal{S}^2_{N'})$. Since $\sigma \Vdash_n C'$, we have $\sigma \Vdash_n C_{N'}$ for all subset $N' \subseteq N$. Moreover, following Definition 10.1.4, either $C_{N'} \in \mathcal{S}^1_{N'}$ or $C_{N'} \in \mathcal{S}^2_{N'}$. By induction, we have either $\sigma \Vdash_n \{\bigwedge_{i \in P} t^1_i \wedge \bigwedge_{j \in N'} \neg t^1_j \leq \mathbb{0}\}$ or $\sigma \Vdash_n \{\bigwedge_{i \in P} t^2_i \wedge \bigwedge_{j \in N \setminus N'} \neg t^2_j \leq \mathbb{0}\}$. That is, for all subset $N' \subseteq N$, we have

$$\bigwedge_{i \in P} t^1_i\sigma \wedge \bigwedge_{j \in N'} \neg t^1_j\sigma \leq_n \mathbb{0} \text{ or } \bigwedge_{i \in P} t^2_i\sigma \wedge \bigwedge_{j \in N \setminus N'} \neg t^2_j\sigma \leq_n \mathbb{0}$$

Applying Lemma 10.1.7, we have

$$\bigwedge_{i \in P} (t^1_i \times t^2_i)\sigma \wedge \bigwedge_{j \in N} \neg(t^1_j \times t^2_j)\sigma \leq_{n+1} \mathbb{0}$$

Thus, $\sigma \Vdash_{n+1} \{(\bigwedge_{i \in P}(t^1_i \times t^2_i) \wedge \bigwedge_{j \in N} \neg(t^1_j \times t^2_j) \leq \mathbb{0})\}$.

(NARROW): similar to the case of (NPROD).

□

Given a normalized constraint-set C and a set of type variables X, we define the restriction $C|_X$ of C by X to be $\{(\alpha, c, t) \in C \mid \alpha \in X\}$.

Lemma 10.1.11. *Let t be a type and $\emptyset \vdash_{\mathcal{N}} \{(t, \leq, \mathbb{0})\} \rightsquigarrow \mathcal{S}$. Then for all normalized constraint-set $C \in \mathcal{S}$, all substitution σ and all n, if $\sigma \Vdash_n C|_{\mathsf{tlv}(t)}$ and $\sigma \Vdash_{n-1} C \setminus C|_{\mathsf{tlv}(t)}$, then $\sigma \Vdash_n \{(t, \leq, \mathbb{0})\}$.*

Proof. By applying the rules (NDNF) and (NUNION), the constraint-set $\{(t, \leq, \mathbb{0})\}$ is normalized into a new constraint-set C', consisting of the constraints of the form $(\tau, \leq, \mathbb{0})$, where τ is a single normal form. That is, $\emptyset \vdash_{\mathcal{N}} \{(t, \leq, \mathbb{0})\} \rightsquigarrow \{C'\}$. Let $C'_1 = \{(\tau, \leq, \mathbb{0}) \in C' \mid \mathsf{tlv}(\tau) \neq \emptyset\}$ and $C'_2 = C' \setminus C'_1$. It is easy to deduce that all the constraints in $C \setminus C|_{\mathsf{tlv}(t)}$ are generated from C'_2 and must pass at least one instance of $\vdash^*_{\mathcal{N}}$ (*i.e.*, being decomposed at least once). Since $\sigma \Vdash_{n-1} C \setminus C|_{\mathsf{tlv}(t)}$, then according to the statement (2) in the proof of Lemma 10.1.10, we have $\sigma \Vdash_n C'_2$. Moreover, from $\sigma \Vdash_n C|_{\mathsf{tlv}(t)}$, we have $\sigma \Vdash_n C'_1$. Thus, $\sigma \Vdash_n C'$ and *a fortiori* $\sigma \Vdash_n \{(t, \leq, \mathbb{0})\}$. □

Lemma 10.1.12 (Completeness). *Let C be a constraint-set such that $\emptyset \vdash_{\mathcal{N}} C \rightsquigarrow \mathcal{S}$. For all substitution σ, if $\sigma \Vdash C$, then there exists $C' \in \mathcal{S}$ such that $\sigma \Vdash C'$.*

Proof. We prove the following stronger statements.

(1) Assume $\Sigma \vdash_{\mathcal{N}} C \rightsquigarrow \mathcal{S}$. For all σ, if $\sigma \Vdash C(\Sigma)$ and $\sigma \Vdash C$, then there exists $C' \in \mathcal{S}$ such that $\sigma \Vdash C'$.

(2) Assume $\Sigma \vdash^*_{\mathcal{N}} C \rightsquigarrow \mathcal{S}$. For all σ, if $\sigma \Vdash C(\Sigma)$ and $\sigma \Vdash C$, then there exists $C' \in \mathcal{S}$ such that $\sigma \Vdash C'$.

The result is then a direct consequence of the first item (indeed, we have $\sigma \Vdash C(\emptyset)$ for all σ). We prove the two items simultaneously by induction on the derivations of $\Sigma \vdash_{\mathcal{N}} C \rightsquigarrow \mathcal{S}$ and $\Sigma \vdash^*_{\mathcal{N}} C \rightsquigarrow \mathcal{S}$.

(NEMPTY): straightforward.

(NJOIN): as $\sigma \Vdash \{(t_i\ c_i\ t'_i) \mid i \in I\}$, we have in particular $\sigma \Vdash \{(t_i\ c_i\ t'_i)\}$ for all i. By induction, there exists $C_i \in \mathcal{S}_i$ such that $\sigma \Vdash C_i$. So $\sigma \Vdash \bigcup_{i\in I} C_i$. Moreover, according to Definition 10.1.4, $\bigcup_{i\in I} C_i$ must be in $\bigcap_{i\in I} \mathcal{S}_i$. Therefore, the result follows.

(NSYM): if $\sigma \Vdash \{(t' \geq t)\}$, then $\sigma \Vdash \{(t \leq t')\}$. By induction, the result follows.

(NZERO): since $\sigma \Vdash \{(t \leq t')\}$, according to set-theory, $\sigma \Vdash \{(t \wedge \neg t' \leq \mathbb{0})\}$. By induction, the result follows.

(NDNF): similar to the case of (NZERO).

(NUNION): similar to the case of (NZERO).

(NTLV): assume α' has the smallest order in $P \cup N$. If $\alpha' \in P$, then according to set-theory, we have $\alpha'\sigma \leq \neg(\bigwedge_{\alpha\in(P\setminus\{\alpha'\})} \alpha \wedge \bigwedge_{\alpha\in N} \neg\alpha \wedge \tau_0)$, that is $\sigma \Vdash \{(\alpha' \leq \neg t_{\alpha'})\}$. Otherwise, we have $\alpha' \in N$ and the result follows as well.

(NHYP): it is clear that $\sigma \Vdash \emptyset$.

(NASSUM): as $\sigma \Vdash C(\Sigma)$ and $\sigma \Vdash \{(\tau_0 \leq \mathbb{0})\}$, we have $\sigma \Vdash C(\Sigma \cup \{\tau_0\})$. By induction, the result follows.

(NBASIC): straightforward.

(NPROD): as

$$\sigma \Vdash \{(\bigwedge_{i\in P} (t_i^1 \times t_i^2) \wedge \bigwedge_{j\in N} \neg(t_j^1 \times t_j^2) \leq \mathbb{0})\}$$

we have

$$\bigwedge_{i\in P} (t_i^1 \times t_i^2)\sigma \wedge \bigwedge_{j\in N} \neg(t_j^1 \times t_j^2)\sigma \leq \mathbb{0}$$

Applying Lemma 4.3.9, for all subset $N' \subseteq N$, we have

$$\bigwedge_{i\in P} t_i^1\sigma \wedge \bigwedge_{j\in N'} \neg t_j^1\sigma \leq \mathbb{0} \text{ or } \bigwedge_{i\in P} t_i^2\sigma \wedge \bigwedge_{j\in N\setminus N'} \neg t_j^2\sigma \leq \mathbb{0}$$

that is,

$$\sigma \Vdash \{(\bigwedge_{i\in P} t_i^1 \wedge \bigwedge_{j\in N'} \neg t_j^1 \leq \mathbb{0})\} \text{ or } \sigma \Vdash \{(\bigwedge_{i\in P} t_i^2 \wedge \bigwedge_{j\in N\setminus N'} \neg t_j^2 \leq \mathbb{0})\}$$

By induction, either there exists $C^1_{N'} \in \mathcal{S}^1_{N'}$ such that $\sigma \Vdash C^1_{N'}$ or there exists $C^2_{N'} \in \mathcal{S}^2_{N'}$ such that $\sigma \Vdash C^2_{N'}$. According to Definition 10.1.4, we have $C^1_{N'}, C^2_{N'} \in \mathcal{S}^1_{N'} \sqcup \mathcal{S}^2_{N'}$. Thus there exists $C'_{N'} \in \mathcal{S}^1_{N'} \sqcup \mathcal{S}^2_{N'}$ such that $\sigma \Vdash C'_{N'}$. Therefore $\sigma \Vdash \bigcup_{N' \subseteq N} C'_{N'}$. Moreover, according to Definition 10.1.4 again, $\bigcup_{N' \subseteq N} C'_{N'} \in \bigsqcap_{N' \subseteq N} (\mathcal{S}^1_{N'} \sqcup \mathcal{S}^2_{N'})$. Hence, the result follows.

(NARROW): similar to the case (NPROD) except we use Lemma 4.3.10.

□

We now prove termination of the algorithm.

Definition 10.1.13 (Plinth). *A* plinth $\beth \subset \mathcal{T}$ *is a set of types with the following properties:*

- $\beth$ *is finite;*
- $\beth$ *contains* $\mathbb{1}$, $\mathbb{0}$ *and is closed under Boolean connectives* $(\wedge, \vee, \neg)$*;*
- *for all types* $(t_1 \times t_2)$ *or* $(t_1 \to t_2)$ *in* $\beth$*, we have* $t_1 \in \beth$ *and* $t_2 \in \beth$*.*

As stated in [Fri04], every finite set of types is included in a plinth. Indeed, we already know that for a regular type t the set of its subtrees S is finite. The definition of the plinth ensures that the closure of S under Boolean connective is also finite. Moreover, if t belongs to a plinth $\beth$, then the set of its subtrees is contained in $\beth$. This is used to show the termination of algorithms working on types.

Lemma 10.1.14 (Decidability). *Let* C *be a finite constraint-set. Then the normalization of* C *terminates.*

Proof. Let T be the set of type occurring in C. As C is finite, T is finite as well. Let $\beth$ be a plinth such that $T \subseteq \beth$. Then when we normalize a constraint $(t \leq \mathbb{0})$ during the process of $\emptyset \vdash_{\mathcal{N}} C$, t would belong to $\beth$. We prove the lemma by induction on $(|\beth \backslash \Sigma|, U, |C|)$ lexicographically ordered, where Σ is the set of types we have normalized, U is the number of unions $\vee$ occurring in the constraint-set C plus the number of constraints $(t, \geq, s)$ and the number of constraint $(t, \leq, s)$ where $s \neq \mathbb{0}$ or t is not in disjunctive normal form, and C is the constraint-set to be normalized.

(NEMPTY): it terminates immediately.
(NJOIN): $|C|$ decreases.
(NSYM): U decreases.
(NZERO): U decreases.
(NDNF): U decreases.
(NUNION): although $|C|$ increases, U decreases.
(NTLV): it terminates immediately.
(NHYP): it terminates immediately.
(NASSUM): as $\tau_0 \in \beth$ and $\tau_0 \notin \Sigma$, the number $|\beth \setminus \Sigma|$ decreases.
(NBASIC): it terminates immediately.

(NPROD): although $(|\sqsupset \setminus \Sigma|, U, |C|)$ may not change, the next rule to apply must be one of (NEMPTY), (NJOIN), (NSYM), (NZERO), (NDNF), (NUNION), (NTLV), (NHYP) or (NASSUM). Therefore, either the normalization terminates or the triple decreases in the next step.

(NARROW): similar to Case (NPROD).

□

Lemma 10.1.15 (Finiteness). *Let C be a constraint-set and $\emptyset \vdash_{\mathcal{N}} C \rightsquigarrow \mathcal{S}$. Then $\mathcal{S}$ is finite.*

Proof. It is easy to prove that each normalizing rule generates a finite set of finite sets of normalized constraints. □

Definition 10.1.16. *Let C be a normalized constraint-set and O an ordering on $\mathsf{var}(C)$. We say C is* well-ordered *if for all normalized constraint $(\alpha, c, t_\alpha) \in C$ and for all $\beta \in \mathsf{tlv}(t_\alpha)$, $O(\alpha) < O(\beta)$ holds.*

Lemma 10.1.17. *Let C be a constraint-set and $\emptyset \vdash_{\mathcal{N}} C \rightsquigarrow \mathcal{S}$. Then for all normalized constraint-set $C' \in \mathcal{S}$, C' is well-ordered.*

Proof. The only way to generate normalized constraints is Rule (NTLV), where we have selected the normalized constraint for the type variable α whose order is minimum as the representative one, that is, $\forall \beta \in \mathsf{tlv}(t_\alpha)\ .\ O(\alpha) < O(\beta)$. Therefore, the result follows. □

Definition 10.1.18. *A general renaming ρ is a special type substitution that maps each type variable to another (fresh) type variable.*

Lemma 10.1.19. *Let t, s be two types and $[\rho_i]_{i\in I}$, $[\rho_j]_{j\in J}$ two sets of general renamings. Then if $\emptyset \vdash_{\mathcal{N}} \{(s \wedge t, \leq, \mathbb{0})\} \rightsquigarrow \emptyset$, then $\emptyset \vdash_{\mathcal{N}} \{((\bigwedge_{j\in J} s\rho_j) \wedge (\bigwedge_{i\in I} t\rho_i), \leq, \mathbb{0})\} \rightsquigarrow \emptyset$.*

Proof. By induction on the number of (NPROD) and (NARROW) used in the derivation of $\emptyset \vdash_{\mathcal{N}} \{(s \wedge \neg t, \leq, \mathbb{0})\}$ and by cases on the disjunctive normal form τ of $s \wedge \neg t$. The failure of the normalization of $(s \wedge t, \leq, \mathbb{0})$ is essentially due to (NBASIC-F), (NPROD) and (NARROW), where there are no top-level type variables to make the type empty.
The case of arrows is a little complicated, as we need to consider more than two types: one type for the negative parts and two types for the positive parts from t and s respectively. Indeed, what we prove is the following stronger statement:

$$\emptyset \vdash_{\mathcal{N}} \{(\bigwedge_{k\in K} t_k, \leq, \mathbb{0})\} \rightsquigarrow \emptyset \Longrightarrow \emptyset \vdash_{\mathcal{N}} \{(\bigwedge_{k\in K} (\bigwedge_{i_k\in I_k} t_k\rho_{i_k}), \leq, \mathbb{0})\} \rightsquigarrow \emptyset$$

where $|K| \geq 2$ and ρ_{i_k}'s are general renamings. For simplicity, we only consider $|K| = 2$, as it is easy to extend to the case of $|K| > 2$.

Case 1: $\tau = \tau_{b_s} \wedge \tau_{b_t}$ and $\tau \not\simeq \mathbb{0}$, where τ_{b_s} (τ_{b_t} resp.) is an intersection of basic types from s (t resp.). Then the expansion of τ is

$$(\bigwedge_{j\in J} \tau_{b_s}\rho_j) \wedge (\bigwedge_{i\in I} \tau_{b_t}\rho_i) \simeq \tau_{b_s} \wedge \tau_{b_t} \not\simeq \mathbb{0}$$

So $\emptyset \vdash_{\mathcal{N}} \{((\bigwedge_{j\in J} \tau_{b_s}\rho_j) \wedge (\bigwedge_{i\in I} \tau_{b_t}\rho_i), \leq, \mathbb{0})\} \rightsquigarrow \emptyset$.

Case 2: $\tau = \bigwedge_{p_s \in P_s}(w_{p_s} \times v_{p_s}) \wedge \bigwedge_{n_s \in N_s} \neg(w_{n_s} \times v_{n_s}) \wedge \bigwedge_{p_t \in P_t}(w_{p_t} \times v_{p_t}) \wedge \bigwedge_{n_t \in N_t} \neg(w_{n_t} \times v_{n_t})$, where P_s, N_s are from s and P_t, N_t are from t. Since $\emptyset \vdash_{\mathcal{N}} \{\tau, \leq, \mathbb{0})\} \rightsquigarrow \emptyset$, by the rule (NPROD), there exist two sets $N'_s \subseteq N_s$ and $N'_t \subseteq N_t$ such that

$$\begin{cases} \emptyset \vdash_{\mathcal{N}} \{ \bigwedge\limits_{p_s \in P_s} w_{p_s} \wedge \bigwedge\limits_{n_s \in N'_s} \neg w_{n_s} \wedge \bigwedge\limits_{p_t \in P_t} w_{p_t} \wedge \bigwedge\limits_{n_t \in N'_t} \neg w_{n_t}, \leq, \mathbb{0})\} \rightsquigarrow \emptyset \\ \emptyset \vdash_{\mathcal{N}} \{ \bigwedge\limits_{p_s \in P_s} v_{p_s} \wedge \bigwedge\limits_{n_s \in N_s \setminus N'_s} \neg v_{n_s} \wedge \bigwedge\limits_{p_t \in P_t} v_{p_t} \wedge \bigwedge\limits_{n_t \in N_t \setminus N'_t} \neg v_{n_t}, \leq, \mathbb{0})\} \rightsquigarrow \emptyset \end{cases}$$

By induction, we have

$$\begin{cases} \emptyset \vdash_{\mathcal{N}} \{\bigwedge\limits_{j \in J}(\bigwedge\limits_{p_s \in P_s} w_{p_s} \wedge \bigwedge\limits_{n_s \in N'_s} \neg w_{n_s})\rho_j \wedge \bigwedge\limits_{i \in I}(\bigwedge\limits_{p_t \in P_t} w_{p_t} \wedge \bigwedge\limits_{n_t \in N'_t} \neg w_{n_t})\rho_i, \leq, \mathbb{0})\} \rightsquigarrow \emptyset \\ \emptyset \vdash_{\mathcal{N}} \{\bigwedge\limits_{j \in J}(\bigwedge\limits_{p_s \in P_s} v_{p_s} \wedge \bigwedge\limits_{n_s \in N_s \setminus N'_s} \neg v_{n_s})\rho_j \wedge \bigwedge\limits_{i \in I}(\bigwedge\limits_{p_t \in P_t} v_{p_t} \wedge \bigwedge\limits_{n_t \in N_t \setminus N'_t} \neg v_{n_t})\rho_i, \leq, \mathbb{0})\} \rightsquigarrow \emptyset \end{cases}$$

Then by the rule (NPROD) again, we get

$$\emptyset \vdash_{\mathcal{N}} \{\bigwedge_{j \in J}(\tau_s)\rho_j \wedge \bigwedge_{i \in I}(\tau_t)\rho_i, \leq, \mathbb{0})\} \rightsquigarrow \emptyset$$

where $\tau_s = \bigwedge_{p_s \in P_s}(w_{p_s} \times v_{p_s}) \wedge \bigwedge_{n_s \in N_s} \neg(w_{n_s} \times v_{n_s})$ and $\tau_t = \bigwedge_{p_t \in P_t}(w_{p_t} \times v_{p_t}) \wedge \bigwedge_{n_t \in N_t} \neg(w_{n_t} \times v_{n_t})$.

Case 3: $\tau = \bigwedge_{p_s \in P_s}(w_{p_s} \to v_{p_s}) \wedge \bigwedge_{n_s \in N_s} \neg(w_{n_s} \to v_{n_s}) \wedge \bigwedge_{p_t \in P_t}(w_{p_t} \to v_{p_t}) \wedge \bigwedge_{n_t \in N_t} \neg(w_{n_t} \to v_{n_t})$, where P_s, N_s are from s and P_t, N_t are from t. Since $\emptyset \vdash_{\mathcal{N}} \{\tau, \leq, \mathbb{0})\} \rightsquigarrow \emptyset$, by the rule (NARROW), for all $w \to v \in N_s \cup N_t$, there exist a set $P'_s \subseteq P_s$ and a set $P'_t \subseteq P_t$ such that

$$\begin{cases} \emptyset \vdash_{\mathcal{N}} \{ \bigwedge\limits_{p_s \in P'_s} \neg w_{p_s} \wedge \bigwedge\limits_{p_t \in P'_t} \neg w_{p_t} \wedge w, \leq, \mathbb{0})\} \rightsquigarrow \emptyset \\ P'_s = P_s \wedge P'_t = P_t \text{ or } \emptyset \vdash_{\mathcal{N}} \{ \bigwedge\limits_{p_s \in P_s \setminus P'_s} v_{p_s} \wedge \bigwedge\limits_{p_t \in P_t \setminus P'_t} v_{p_t} \wedge \neg v, \leq, \mathbb{0})\} \rightsquigarrow \emptyset \end{cases}$$

By induction, for all $\rho \in [\rho_i]_{i \in I} \cup [\rho_j]_{j \in J}$, we have

$$\begin{cases} \emptyset \vdash_{\mathcal{N}} \{\bigwedge\limits_{j \in J}(\bigwedge\limits_{p_s \in P'_s} \neg w_{p_s})\rho_j \wedge \bigwedge\limits_{i \in I}(\bigwedge\limits_{p_t \in P'_t} \neg w_{p_t})\rho_i \wedge w\rho, \leq, \mathbb{0})\} \rightsquigarrow \emptyset \\ \begin{cases} P'_s = P_s \wedge P'_t = P_t \\ \text{or} \\ \emptyset \vdash_{\mathcal{N}} \{\bigwedge\limits_{j \in J}(\bigwedge\limits_{p_s \in P_s \setminus P'_s} v_{p_s})\rho_j \wedge \bigwedge\limits_{i \in I}(\bigwedge\limits_{p_t \in P_t \setminus P'_t} v_{p_t})\rho_i \wedge \neg v\rho, \leq, \mathbb{0})\} \rightsquigarrow \emptyset \end{cases} \end{cases}$$

Then by the rule (NARROW) again, we get

$$\emptyset \vdash_{\mathcal{N}} \{\bigwedge_{j \in J}(\tau_s)\rho_j \wedge \bigwedge_{i \in I}(\tau_t)\rho_i, \leq, \mathbb{0})\} \rightsquigarrow \emptyset$$

where $\tau_s = \bigwedge_{p_s \in P_s}(w_{p_s} \to v_{p_s}) \wedge \bigwedge_{n_s \in N_s} \neg(w_{n_s} \to v_{n_s})$ and $\tau_t = \bigwedge_{p_t \in P_t}(w_{p_t} \to v_{p_t}) \wedge \bigwedge_{n_t \in N_t} \neg(w_{n_t} \to v_{n_t})$.

Case 4: $\tau = (\bigvee_{k_s \in K_s} \tau_{k_s}) \wedge (\bigvee_{k_t \in K_t} \tau_{k_t})$, where τ_{k_s} and τ_{k_t} are single normal forms. As $\emptyset \vdash_{\mathcal{N}} \{(\tau, \leq, \mathbb{0})\} \rightsquigarrow \emptyset$, there must exist at least one $k_s \in K_s$ and at least one $k_t \in K_t$ such that $\emptyset \vdash_{\mathcal{N}} \{(\tau_{k_s} \wedge \tau_{k_t}, \leq, \mathbb{0})\} \rightsquigarrow \emptyset$. By Cases (1) – (3), the result follows.

□

The type tallying problem is parameterized with a set Δ of type variables that cannot be instantiated, but so far, we have not considered these monomorphic variables in the normalization procedure. Taking Δ into account affects only the (NTLV) rule, where a normalized constraint is built by singling out a variable α. Since the type substitution σ we want to construct must not touch the type variables in Δ (*i.e.*, $\sigma \sharp \Delta$), we cannot choose a variable α in Δ. To avoid this, we order the variables in C so that those belonging to Δ are always greater than those not in Δ. If, by choosing the minimum top-level variable α, we obtain $\alpha \in \Delta$, it means that all the top-level type variables are contained in Δ. According to Lemmas 5.1.2 and 5.1.3, we can then safely eliminate these type variables. So taking Δ into account, we amend the (NTLV) rule as follows.

$$\frac{\mathsf{tlv}(\tau_0) = \emptyset \qquad \alpha' \not\downarrow_O P \cup N \qquad \mathcal{S} = \begin{cases} \{\{(\alpha', \leq, \neg t_{\alpha'})\}\} & \alpha' \in P \setminus \Delta \\ \{\{(\alpha', \geq, t_{\alpha'})\}\} & \alpha' \in N \setminus \Delta \\ \Sigma \vdash_{\mathcal{N}} \{(\tau_0 \leq \mathbb{0})\} & \alpha' \in \Delta \end{cases}}{\Sigma \vdash_{\mathcal{N}} \{(\bigwedge_{\alpha \in P} \alpha \wedge \bigwedge_{\alpha \in N} \neg\alpha \wedge \tau_0 \leq \mathbb{0})\} \rightsquigarrow \mathcal{S}} (\text{NTLV})$$

Furthermore, it is easy to prove the soundness, completeness, and termination of the algorithm extended with Δ.

10.1.2 Constraint saturation

A normalized constraint-set may contain several constraints for the same type variable, which can eventually be merged together. For instance, the constraints $\alpha \geq t_1$ and $\alpha \geq t_2$ can be replaced by $\alpha \geq t_1 \vee t_2$, and the constraints $\alpha \leq t_1$ and $\alpha \leq t_2$ can be replaced by $\alpha \leq t_1 \wedge t_2$. That is to say, we can merge all the lower bounds (resp. upper bounds) of a type variable into only one by unions (resp. intersections). The merging rules are given in Figure 10.2.

After repeated uses of the merging rules, a set C contains at most one lower bound constraint and at most one upper bound constraint for each type variable. If both lower and upper bounds exist for a given α, that is, $\alpha \geq t_1$ and $\alpha \leq t_2$ belong to C, then the substitution we want to construct from C must satisfy the constraint $(t_1, \leq, t_2)$ as well. For that, we first normalize the constraint $(t_1, \leq, t_2)$, yielding a set of constraint-sets $\mathcal{S}$, and then saturate C with any normalized constraint-set $C' \in \mathcal{S}$. Formally, we describe the saturation process as a judgement $\Sigma_p, C_\Sigma \vdash_{\mathcal{S}} C \rightsquigarrow \mathcal{S}$, where Σ_p is a set of type pairs (if $(t_1, t_2) \in \Sigma_p$, then the constraint $t_1 \leq t_2$ has already been treated at this point), C_Σ is a normalized constraint-set (which collects the treated original constraints, like $(\alpha, \geq, t_1)$ and $(\alpha, \leq, t_2)$, that generate the additional constraints), C is the normalized constraint-set we want to saturate, and $\mathcal{S}$ is a set of sets of normalized constraints (the

$$\frac{\forall i \in I \;.\; (\alpha \geq t_i) \in C \qquad |I| \geq 2}{\vdash_{\mathcal{M}} C \rightsquigarrow (C \setminus \{(\alpha \geq t_i) \mid i \in I\}) \cup \{(\alpha \geq \bigvee_{i \in I} t_i)\}} (\text{MLB})$$

$$\frac{\forall i \in I \;.\; (\alpha \leq t_i) \in C \qquad |I| \geq 2}{\vdash_{\mathcal{M}} C \rightsquigarrow (C \setminus \{(\alpha \leq t_i) \mid i \in I\}) \cup \{(\alpha \leq \bigwedge_{i \in I} t_i)\}} (\text{MUB})$$

Figure 10.2: Merging rules

result of the saturation of C joined with C_Σ). The saturation rules are given in Figure 10.3.

$$\frac{\Sigma_p, C_\Sigma \cup \{(\alpha \geq t_1), (\alpha \leq t_2)\} \vdash_{\mathcal{S}} C \rightsquigarrow \mathcal{S} \qquad (t_1, t_2) \in \Sigma_p}{\Sigma_p, C_\Sigma \vdash_{\mathcal{S}} \{(\alpha \geq t_1), (\alpha \leq t_2)\} \cup C \rightsquigarrow \mathcal{S}} (\textsc{Shyp})$$

$$\frac{\begin{array}{l}(t_1, t_2) \notin \Sigma_p \quad \emptyset \vdash_{\mathcal{N}} \{(t_1 \leq t_2)\} \rightsquigarrow \mathcal{S} \\ \mathcal{S}' = \{\{(\alpha \geq t_1), (\alpha \leq t_2)\} \cup C \cup C_\Sigma\} \sqcap \mathcal{S} \\ \forall C' \in \mathcal{S}' .\; \Sigma_p \cup \{(t_1, t_2)\}, \emptyset \vdash_{\mathcal{MS}} C' \rightsquigarrow \mathcal{S}_{C'}\end{array}}{\Sigma_p, C_\Sigma \vdash_{\mathcal{S}} \{(\alpha \geq t_1), (\alpha \leq t_2)\} \cup C \rightsquigarrow \bigsqcup_{C' \in \mathcal{S}'} \mathcal{S}_{C'}} (\textsc{Sassum})$$

$$\frac{\forall \alpha, t_1, t_2 \;\nexists \{(\alpha \geq t_1), (\alpha \leq t_2)\} \subseteq C}{\Sigma_p, C_\Sigma \vdash_{\mathcal{S}} C \rightsquigarrow \{C \cup C_\Sigma\}} (\textsc{Sdone})$$

where $\Sigma_p, C_\Sigma \vdash_{\mathcal{MS}} C \rightsquigarrow \mathcal{S}$ means that there exists C' such that $\vdash_{\mathcal{M}} C \rightsquigarrow C'$ and $\Sigma_p, C_\Sigma \vdash_{\mathcal{S}} C' \rightsquigarrow \mathcal{S}$.

Figure 10.3: Saturation rules

If $\alpha \geq t_1$ and $\alpha \leq t_2$ belongs to the constraint-set C that is being saturated, and $t_1 \leq t_2$ has already been processed (i.e., $(t_1, t_2) \in \Sigma_p$), then the rule (Shyp) simply extends C_Σ (the result of the saturation so far) with $\{\alpha \geq t_1, \alpha \leq t_2\}$. Otherwise, the rule (Sassum) first normalizes the fresh constraint $\{t_1 \leq t_2\}$, yielding a set of normalized constraint-sets $\mathcal{S}$. It then saturates (joins) C and C_Σ with each constraint-set $C_\mathcal{S} \in \mathcal{S}$, the union of which gives a new set $\mathcal{S}'$ of normalized constrain-sets. Each C' in $\mathcal{S}'$ may contain several constraints for the same type variable, so they have to be merged and saturated themselves. Finally, if C does not contain any couple $\alpha \geq t_1$ and $\alpha \leq t_2$ for a given α, the process is over and the rule (Sdone) simply returns $C \cup C_\Sigma$.

If $\emptyset, \emptyset \vdash_{\mathcal{MS}} C \rightsquigarrow \mathcal{S}$, then the result of the saturation of C is $\mathcal{S}$.

Lemma 10.1.20 (Soundness). *Let C be a normalized constraint-set. If $\emptyset, \emptyset \vdash_{\mathcal{MS}} C \rightsquigarrow \mathcal{S}$, then for all normalized constraint-set $C' \in \mathcal{S}$ and all substitution σ, we have $\sigma \Vdash C' \Rightarrow \sigma \Vdash C$.*

Proof. We prove the following statements.

- Assume $\vdash_{\mathcal{M}} C \rightsquigarrow C'$. For all σ, if $\sigma \Vdash C'$, then $\sigma \Vdash C$.
- Assume $\Sigma_p, C_\Sigma \vdash_{\mathcal{S}} C \rightsquigarrow \mathcal{S}$. For all σ and $C_0 \in \mathcal{S}$, if $\sigma \Vdash C_0$, then $\sigma \Vdash C_\Sigma \cup C$.

Clearly, these two statements imply the lemma. The first statement is straightforward. The proof of the second statement proceeds by induction of the derivation of $\Sigma_p, C_\Sigma \vdash_{\mathcal{S}} C \rightsquigarrow \mathcal{S}$.

(SHYP)**:** by induction, we have $\sigma \Vdash (C_\Sigma \cup \{(\alpha \geq t_1), (\alpha \leq t_2)\}) \cup C$, that is $\sigma \Vdash C_\Sigma \cup (\{(\alpha \geq t_1), (\alpha \leq t_2)\} \cup C)$.

(SASSUM)**:** according to Definition 10.1.4, $C_0 \in \mathcal{S}_{C'}$ for some $C' \in \mathcal{S}'$. By induction on the premise $\Sigma_p \cup \{(t_1, t_2)\}, \emptyset \vdash_{\mathcal{MS}} C' \rightsquigarrow \mathcal{S}_{C'}$, we have $\sigma \Vdash C'$. Moreover, the equation $\mathcal{S}' = \{\{(\alpha \geq t_1), (\alpha \leq t_2)\} \cup C \cup C_\Sigma\} \sqcap \mathcal{S}$ gives us $\{(\alpha \geq t_1), (\alpha \leq t_2)\} \cup C \cup C_\Sigma \subseteq C'$. Therefore, we have $\sigma \Vdash C_\Sigma \cup (\{(\alpha \geq t_1), (\alpha \leq t_2)\} \cup C)$.

(SDONE)**:** straightforward.

□

Lemma 10.1.21 (Completeness). *Let C be a normalized constraint-set and $\emptyset, \emptyset \vdash_{\mathcal{MS}} C \rightsquigarrow \mathcal{S}$. Then for all substitution σ, if $\sigma \Vdash C$, then there exists $C' \in \mathcal{S}$ such that $\sigma \Vdash C'$.*

Proof. We prove the following statements.

- Assume $\vdash_{\mathcal{M}} C \rightsquigarrow C'$. For all σ, if $\sigma \Vdash C$, then $\sigma \Vdash C'$.
- Assume $\Sigma_p, C_\Sigma \vdash_{\mathcal{S}} C \rightsquigarrow \mathcal{S}$. For all σ, if $\sigma \Vdash C_\Sigma \cup C$, then there exists $C_0 \in \mathcal{S}$ such that $\sigma \Vdash C_0$.

Clearly, these two statements imply the lemma. The first statement is straightforward. The proof of the second statement proceeds by induction of the derivation of $\Sigma_p, C_\Sigma \vdash_{\mathcal{S}} C \rightsquigarrow \mathcal{S}$.

(SHYP)**:** the result follows by induction.

(SASSUM)**:** as $\sigma \Vdash C_\Sigma \cup (\{(\alpha \geq t_1), (\alpha \leq t_2)\} \cup C)$, we have $\sigma \Vdash \{(t_1 \leq t_2)\}$. As $\emptyset \vdash_{\mathcal{N}} \{(t_1 \leq t_2)\} \rightsquigarrow \mathcal{S}$, applying Lemma 10.1.12, there exists $C'_0 \in \mathcal{S}$ such that $\sigma \Vdash C'_0$. Let $C' = C_\Sigma \cup (\{(\alpha \geq t_1), (\alpha \leq t_2)\} \cup C) \cup C'_0$. Clearly we have $\sigma \Vdash C'$ and $C' \in S'$. By induction on the premise $\Sigma_p \cup \{(t_1, t_2)\}, \emptyset \vdash_{\mathcal{MS}} C' \rightsquigarrow \mathcal{S}_{C'}$, there exists $C_0 \in \mathcal{S}_{C'}$ such that $\sigma \Vdash C_0$. Moreover, it is clear that $C_0 \in \bigsqcup_{C' \in S'} \mathcal{S}_{C'}$. Therefore, the result follows.

(SDONE)**:** straightforward.

□

Lemma 10.1.22 (Decidability). *Let C be a finite normalized constraint-set. Then $\emptyset, \emptyset \vdash_{\mathcal{MS}} C$ terminates.*

Proof. Let T be the set of types occurring in C. As C is finite, T is finite as well. Let $\beth$ be a plinth such that $T \subseteq \beth$. Then when we saturate a fresh constraint $(t_1, \leq, t_2)$ during the process of $\emptyset, \emptyset \vdash_{\mathcal{MS}} C$, (t_1, t_2) would belong to $\beth \times \beth$. According to Lemma 10.1.14, we know that $\emptyset \vdash_{\mathcal{N}} \{(t_1, \leq, t_2)\}$ terminates. Moreover,the termination of the merging of the lower bounds or the upper bounds of a same type variable is straightforward. Finally, we have to prove termination of the saturation process. The proof proceeds by induction on $(|(\beth \times \beth)| - |\Sigma_p|, |C|)$ lexicographically ordered:

(*Shyp*)**:** $|C|$ decreases.
(*Sassum*)**:** as $(t_1, t_2) \notin \Sigma_p$ and $t_1, t_2 \in \beth$, $|(\beth \times \beth)| - |\Sigma_p|$ decreases.
(*Sdone*)**:** it terminates immediately.

□

Definition 10.1.23 (Sub-constraint). *Let $C_1, C_2 \in 2^{\mathcal{C}}$ be two normalized constraint-sets. We say C_1 is a* sub-constraint *of C_2, denoted as $C_1 \lessdot C_2$, if for all $(\alpha, c, t) \in C_1$, there exists $(\alpha, c, t') \in C_2$ such that t' c t, where $c \in \{\leq, \geq\}$.*

Lemma 10.1.24. *Let $C_1, C_2 \in 2^{\mathcal{C}}$ be two normalized constraint-sets and $C_1 \lessdot C_2$. Then for all substitution σ, if $\sigma \Vdash C_2$, then $\sigma \Vdash C_1$.*

Proof. Considering any constraint $(\alpha, c, t) \in C_1$, there exists $(\alpha, c, t') \in C_2$ and t' c t, where $c \in \{\leq, \geq\}$. Since $\sigma \Vdash C_2$, then $\sigma(\alpha)$ c $t'\sigma$. Moreover, as t' c t, we have $t'\sigma$ c $t\sigma$. Thus $\sigma(\alpha)$ c $t\sigma$. □

Definition 10.1.25. *Let $C \in 2^{\mathcal{C}}$ be a normalized constraint-set. We say C is* saturated *if for each type variable $\alpha \in \mathsf{dom}(C)$,*

(1) *there exists at most one form $(\alpha \geq t_1) \in C$,*

(2) *there exists at most one form $(\alpha \leq t_2) \in C$,*

(3) *if $(\alpha \geq t_1), (\alpha \leq t_2) \in C$, then $\emptyset \vdash_{\mathcal{N}} \{(t_1 \leq t_2)\} \rightsquigarrow \mathcal{S}$ and there exists $C' \in \mathcal{S}$ such that C' is a sub-constraint of C (*i.e.*, $C' \lessdot C$).*

Lemma 10.1.26. *Let C be a finite normalized constraint-set and $\emptyset, \emptyset \vdash_{\mathcal{MS}} C \rightsquigarrow \mathcal{S}$. Then for all normalized constraint set $C' \in \mathcal{S}$, C' is saturated.*

Proof. We prove a stronger statement: assume $\Sigma_p, C_\Sigma \vdash_{\mathcal{MS}} C \rightsquigarrow \mathcal{S}$. If

(*i*) for all $(t_1, t_2) \in \Sigma_p$ there exists $C' \in (\emptyset \vdash_{\mathcal{N}} \{(t_1 \leq t_2)\})$ such that $C' \lessdot C_\Sigma \cup C$ and

(*ii*) for all $\{(\alpha \geq t_1), (\alpha \leq t_2)\} \subseteq C_\Sigma$ the pair (t_1, t_2) is in Σ_p,

then C_0 is saturated for all $C_0 \in \mathcal{S}$.

The proof of conditions (1) and (2) for a saturated constraint-set is straightforward for all $C_0 \in \mathcal{S}$. The proof of the condition (3) proceeds by induction on the derivation $\Sigma_p, C_\Sigma \vdash_{\mathcal{S}} C \rightsquigarrow \mathcal{S}$ and a case analysis on the last rule used in the derivation.

(SHYP): as $(t_1, t_2) \in \Sigma_p$, the conditions (i) and (ii) hold for the premise. By induction, the result follows.

(SASSUME): take any premise $\Sigma_p \cup \{(t_1, t_2)\}, \emptyset \vdash_{\mathcal{S}} C'' \rightsquigarrow \mathcal{S}_{C'}$, where $C' \in \mathcal{S}'$ and $\vdash_{\mathcal{M}} C' \rightsquigarrow C''$. For any $(s_1, s_2) \in \Sigma_p$, the condition (i) gives us that there exists $C_0 \in (\emptyset \vdash_{\mathcal{N}} \{(s_1 \leq s_2)\})$ such that $C_0 \lessdot C_\Sigma \cup (\{(\alpha \geq t_1), (\alpha \leq t_2)\} \cup C)$. Since $\mathcal{S}' = C_\Sigma \cup (\{(\alpha \geq t_1), (\alpha \leq t_2)\} \cup C) \sqcap \mathcal{S}$, we have $C_0 \lessdot C''$. Moreover, consider (t_1, t_2). As $\emptyset \vdash_{\mathcal{N}} \{(t_1 \leq t_2)\} \rightsquigarrow \mathcal{S}$, there exists $C_0 \in \mathcal{S}$ such that $C_0 \lessdot C''$. Thus the condition (i) holds for the premise. Moreover, the condition (ii) holds straightforwardly for premise. By induction, the result follows.

(SDONE): the result follows by the conditions (i) and (ii).

□

Lemma 10.1.27 (**Finiteness**). *Let C be a constraint-set and $\emptyset, \emptyset \vdash_{\mathcal{MS}} C \rightsquigarrow \mathcal{S}$. Then $\mathcal{S}$ is finite.*

Proof. It follows by Lemma 10.1.15. □

Lemma 10.1.28. *Let C be a well-ordered normalized constraint-set and $\emptyset, \emptyset \vdash_{\mathcal{MS}} C \rightsquigarrow \mathcal{S}$. Then for all normalized constraint-set $C' \in \mathcal{S}$, C' is well-ordered.*

Proof. The merging of the lower bounds (or the upper bounds) of a same type variable preserves the orderings. The result of saturation is well-ordered by Lemma 10.1.17. □

Normalization and saturation may produce redundant constraint-sets. For example, consider the constraint-set $\{(\alpha \times \beta), \leq, (\texttt{Int} \times \texttt{Bool})\}$. Applying the rule (NPROD), the normalization of this set is

$$\{\{(\alpha, \leq, \mathbb{0})\}, \{(\beta, \leq, \mathbb{0})\}, \{(\alpha, \leq, \mathbb{0}), (\beta, \leq, \mathbb{0})\}, \{(\alpha, \leq, \texttt{Int}), (\beta, \leq, \texttt{Bool})\}\}.$$

Clearly each constraint-set is a saturated one. Note that $\{(\alpha, \leq, \mathbb{0}), (\beta, \leq, \mathbb{0})\}$ is redundant, since any solution of this constraint-set is a solution of $\{(\alpha, \leq, \mathbb{0})\}$ and $\{(\beta, \leq, \mathbb{0})\}$. Therefore it is safe to eliminate it. Generally, for any two different normalized constraint sets $C_1, C_2 \in \mathcal{S}$, if $C_1 \lessdot C_2$, then according to Lemma 10.1.24, any solution of C_2 is a solution of C_1. Therefore, C_2 can be eliminated from $\mathcal{S}$.

Definition 10.1.29. *Let $\mathcal{S}$ be a set of normalized constraint-sets. We say that $\mathcal{S}$ is* minimal *if for any two different normalized constraint-sets $C_1, C_2 \in \mathcal{S}$, neither $C_1 \lessdot C_2$ nor $C_2 \lessdot C_1$. Moreover, we say $\mathcal{S} \simeq \mathcal{S}'$ if for all substitution σ such that $\exists C \in \mathcal{S}.\ \sigma \Vdash C \iff \exists C' \in \mathcal{S}'.\ \sigma \Vdash C'$.*

Lemma 10.1.30. *Let C be a well-ordered normalized constraint-set and $\emptyset, \emptyset \vdash_{\mathcal{MS}} C \rightsquigarrow \mathcal{S}$. Then there exists a minimal set $\mathcal{S}_0$ such that $\mathcal{S}_0 \simeq \mathcal{S}$.*

Proof. By eliminating the redundant constraint-sets in $\mathcal{S}$. □

10.1.3 From constraints to equations

In this section, we transform a well-ordered saturated constraint-set into an equivalent equation system. This shows that the type tallying problem is essentially a unification problem [BS01].

Definition 10.1.31 (**Equation system).** *An* equation system E *is a set of equations of the form* $\alpha = t$ *such that there exists at most one equation in* E *for every type variable* α*. We define the domain of* E*, written* $\mathsf{dom}(E)$*, as the set* $\{\alpha \mid \exists t \,.\, \alpha = t \in E\}$*.*

Definition 10.1.32 (**Equation system solution).** *Let* E *be an equation system. A solution to* E *is a substitution* σ *such that*

$$\forall \alpha \;=\; t \in E \,.\, \sigma(\alpha) \;\simeq\; t\sigma \text{ holds}$$

If σ *is a solution to* E*, we write* $\sigma \Vdash E$*.*

From a normalized constraint-set C, we obtain some explicit conditions for the substitution σ we want to construct from C. For instance, from the constraint $\alpha \leq t$ (resp. $\alpha \geq t$), we know that the type substituted for α must be a subtype of t (resp. a super type of t).

We assume that each type variable $\alpha \in \mathsf{dom}(C)$ has a lower bound t_1 and a upper bound t_2 using, if necessary, the fact that $\mathbb{0} \leq \alpha \leq \mathbb{1}$. Formally, we rewrite C as follows.

$$\begin{cases} t_1 \leq \alpha \leq \mathbb{1} & \text{if } \alpha \geq t_1 \in C \text{ and } \nexists t.\ \alpha \leq t \in C \\ \mathbb{0} \leq \alpha \leq t_2 & \text{if } \alpha \leq t_2 \in C \text{ and } \nexists t.\ \alpha \geq t \in C \\ t_1 \leq \alpha \leq t_2 & \text{if } \alpha \geq t_1, \alpha \leq t_2 \in C \end{cases}$$

We then transform each constraint $t_1 \leq \alpha \leq t_2$ in C into an equation $\alpha = (t_1 \vee \alpha') \wedge t_2$[3], where α' is a fresh type variable. The type $(t_1 \vee \alpha') \wedge t_2$ ranges from t_1 to t_2, so the equation $\alpha = (t_1 \vee \alpha') \wedge t_2$ expresses the constraint that $t_1 \leq \alpha \leq t_2$, as wished. We prove the soundness and completeness of this transformation.

To prove soundness, we define the rank n satisfaction predicate $\Vdash_n$ for equation systems, which is similar to the one for constraint-sets.

Lemma 10.1.33 (**Soundness).** *Let* $C \subseteq \mathcal{C}$ *be a well-ordered saturated normalized constraint-set and* E *its transformed equation system. Then for all substitution* σ*, if* $\sigma \Vdash E$ *then* $\sigma \Vdash C$*.*

Proof. Without loss of generality, we assume that each type variable $\alpha \in \mathsf{dom}(C)$ has a lower bound and an upper bound, that is $t_1 \leq \alpha \leq t_2 \in C$. We write $O(C_1) < O(C_2)$ if $O(\alpha) < O(\beta)$ for all $\alpha \in \mathsf{dom}(C_1)$ and all $\beta \in \mathsf{dom}(C_2)$. We first prove a stronger statement:

(*) for all σ, n and $C_\Sigma \subseteq C$, if $\sigma \Vdash_n E$, $\sigma \Vdash_n C_\Sigma$, $\sigma \Vdash_{n-1} C \setminus C_\Sigma$, and $O(C \setminus C_\Sigma) < O(C_\Sigma)$, then $\sigma \Vdash_n C \setminus C_\Sigma$.

[3] Or, equivalently, $\alpha = t_1 \vee (\alpha' \wedge t_2)$. Besides, in practice, if only $\alpha \geq t_1$ ($\alpha \leq t_2$ resp.) and all the occurrences of α in the co-domain of the function type are positive (negative resp.), we can use $\alpha = t_1$ ($\alpha = t_2$ resp.) instead, and the completeness is ensured by subsumption.

Here C_Σ denotes the set of constraints that have been checked. The proof proceeds by induction on $|C \setminus C_\Sigma|$.

$C \setminus C_\Sigma = \emptyset$**:** straightforward.

$C \setminus C_\Sigma \neq \emptyset$**:** take the constraint $(t_1 \leq \alpha \leq t_2) \in C \setminus C_\Sigma$ such that $O(\alpha)$ is the maximum in $\mathsf{dom}(C \setminus C_\Sigma)$. Clearly, there exists a corresponding equation $\alpha = (t_1 \vee \alpha') \wedge t_2 \in E$. As $\sigma \Vdash_n E$, we have $\sigma(\alpha) \simeq_n ((t_1 \vee \alpha') \wedge t_2)\sigma$. Then,

$$\begin{array}{rcl} \sigma(\alpha) \wedge \neg t_2\sigma \simeq_n & ((t_1 \vee \alpha') \wedge t_2)\sigma \wedge \neg t_2\sigma \\ \simeq_n & \mathbb{0} \end{array}$$

Therefore, $\sigma(\alpha) \leq_n t_2\sigma$.
Consider the constraint $(t_1, \leq, \alpha)$. We have

$$\begin{array}{rcl} t_1\sigma \wedge \neg\sigma(\alpha) \simeq_n & t_1\sigma \wedge \neg((t_1 \vee \alpha') \wedge t_2)\sigma \\ \simeq_n & t_1\sigma \wedge \neg t_2\sigma \end{array}$$

What remains to do is to check the subtyping relation $t_1\sigma \wedge \neg t_2\sigma \leq_n \mathbb{0}$, that is, to check that the judgement $\sigma \Vdash_n \{(t_1 \leq t_2)\}$ holds. Since the whole constraint-set C is saturated, according to Definition 10.1.25, we have $\emptyset \vdash_{\mathcal{N}} \{(t_1 \leq t_2)\} \rightsquigarrow \mathcal{S}$ and there exists $C' \in \mathcal{S}$ such that $C' \lessdot C$, that is $C' \lessdot C_\Sigma \cup C \setminus C_\Sigma$. Moreover, as C is well-ordered, $O(\{\alpha\}) < O(\mathsf{tlv}(t_1) \cup \mathsf{tlv}(t_2))$ and thus $O(C \setminus C_\Sigma) < O(\mathsf{tlv}(t_1) \cup \mathsf{tlv}(t_2))$. Therefore, we can deduce that $C'|_{\mathsf{tlv}(t_1)\cup\mathsf{tlv}(t_2)} \lessdot C_\Sigma$ and $C' \setminus C'|_{\mathsf{tlv}(t_1)\cup\mathsf{tlv}(t_2)} \lessdot C \setminus C_\Sigma$. From the premise and Lemma 10.1.24, we have $\sigma \Vdash_n C'|_{\mathsf{tlv}(t_1)\cup\mathsf{tlv}(t_2)}$ and $\sigma \Vdash_{n-1} C' \setminus C'|_{\mathsf{tlv}(t_1)\cup\mathsf{tlv}(t_2)}$. Then, by Lemma 10.1.11, we get $\sigma \Vdash_n \{(t_1 \leq t_2)\}$.
Finally, consider the constraint-set $C \setminus (C_\Sigma \cup \{(t_1 \leq \alpha \leq t_2)\})$. By induction, we have $\sigma \Vdash_n C \setminus (C_\Sigma \cup \{(t_1 \leq \alpha \leq t_2)\})$. Thus the result follows.

Finally, we explain how to prove the lemma with the statement (*). Take $C_\Sigma = \emptyset$. Since $\sigma \Vdash E$, we have $\sigma \Vdash_n E$ for all n. Trivially, we have $\sigma \Vdash_0 C$. This can be used to prove $\sigma \Vdash_1 C$. Since $\sigma \Vdash_1 E$, by (*), we get $\sigma \Vdash_1 C$, which will be used to prove $\sigma \Vdash_2 C$. Consequently, we can get $\sigma \Vdash_n C$ for all n, which clearly implies the lemma. □

Lemma 10.1.34 (Completeness). *Let $C \subseteq \mathcal{C}$ be a saturated normalized constraint-set and E its transformed equation system. Then for all substitution σ, if $\sigma \Vdash C$ then there exists σ' such that $\sigma' \sharp \sigma$ and $\sigma \cup \sigma' \Vdash E$.*

Proof. Let $\sigma' = \{\sigma(\alpha)/\alpha' \mid \alpha \in \mathsf{dom}(C)\}$. Consider each equation $\alpha = (t_1 \vee \alpha') \wedge t_2 \in E$. Correspondingly, there exist $\alpha \geq t_1 \in C$ and $\alpha \leq t_2 \in C$. As $\sigma \Vdash C$, then $t_1\sigma \leq \sigma(\alpha)$ and $\sigma(\alpha) \leq t_2\sigma$. Thus

$$\begin{array}{rcl} ((t_1 \vee \alpha') \wedge t_2)(\sigma \cup \sigma') = & (t_1(\sigma \cup \sigma') \vee \alpha'(\sigma \cup \sigma')) \wedge t_2(\sigma \cup \sigma') \\ = & (t_1\sigma \vee \sigma(\alpha)) \wedge t_2\sigma \\ \simeq & \sigma(\alpha) \wedge t_2\sigma \quad (t_1\sigma \leq \sigma(\alpha)) \\ \simeq & \sigma(\alpha) \quad (\sigma(\alpha) \leq t_2\sigma) \\ = & (\sigma \cup \sigma')(\alpha) \end{array}$$

□

Definition 10.1.35. *Let E be an equation system and O an ordering on* $\mathsf{dom}(E)$*. We say that E is* well-ordered *if for all equation $\alpha = t_\alpha \in E$, we have $O(\alpha) < O(\beta)$ for all $\beta \in \mathit{tlv}(t_\alpha) \cap \mathsf{dom}(E)$.*

Lemma 10.1.36. *Let C be a well-ordered saturated normalized constraint-set and E its transformed equation system. Then E is well-ordered.*

Proof. Clearly, $\mathsf{dom}(E) = \mathsf{dom}(C)$. Consider an equation $\alpha = (t_1 \vee \alpha') \wedge t_2$. Correspondingly, there exist $\alpha \geq t_1 \in C$ and $\alpha \leq t_2 \in C$. By Definition 10.1.16, for all $\beta \in (\mathsf{tlv}(t_1) \cup \mathsf{tlv}(t_2)) \cap \mathsf{dom}(C)$. $O(\alpha) < O(\beta)$. Moreover, α' is a fresh type variable in C, that is $\alpha' \notin \mathsf{dom}(C)$. And then $\alpha' \notin \mathsf{dom}(E)$. Therefore, $\mathsf{tlv}((t_1 \vee \alpha') \wedge t_2) \cap \mathsf{dom}(E) = (\mathsf{tlv}(t_1) \cup \mathsf{tlv}(t_2)) \cap \mathsf{dom}(C)$. Thus the result follows. □

10.1.4 Solution of equation systems

We now extract a solution (i.e., a substitution) from the equation system we build from C. In an equation $\alpha = t_\alpha$, α may also appear in the type t_α; such an equality reminds the definition of a recursive type. As a first step, we introduce a recursion operator μ in all the equations of the system, transforming $\alpha = t_\alpha$ into $\alpha = \mu x_\alpha.\ t_\alpha\{x_\alpha/\alpha\}$. This ensures that type variables do not appear in the right-hand side of the equalities, making the whole solving process easier. If some recursion operators are in fact not needed in the solution (*i.e.*, we have $\alpha = \mu x_\alpha.t_\alpha$ with $x_\alpha \notin \mathsf{fv}(t_\alpha)$), then we can simply eliminate them.

If the equation system contains only one equation, then this equation is immediately a substitution. Otherwise, consider the equation system $\{\alpha = \mu x_\alpha.\ t_\alpha\} \cup E$, where E contains only equations closed with the recursion operator μ as explained above. The next step is to substitute the content expression $\mu x_\alpha.\ t_\alpha$ for all the occurrences of α in equations in E. In detail, let $\beta = \mu x_\beta.\ t_\beta \in E$. Since t_α may contain some occurrences of β and these occurrences are clearly bounded by μx_β, we in fact replace the equation $\beta = \mu x_\beta.\ t_\beta$ with $\beta = \mu x_\beta.\ t_\beta\{\mu x_\alpha.\ t_\alpha/\alpha\}\{x_\beta/\beta\}$, yielding a new equation system E'. Finally, assume that the equation system E' (which has fewer equations) has a solution σ'. Then the substitution $\{t_\alpha\sigma'/\alpha\} \oplus \sigma'$ is a solution to the original equation system $\{\alpha = \mu x_\alpha.\ t_\alpha\} \cup E$. The solving algorithm $\mathsf{Unify}()$ is given in Figure 10.4.

Definition 10.1.37 (General solution). *Let E be an equation system. A* general solution *to E is a substitution σ from* $\mathsf{dom}(E)$ *to $\mathcal{T}$ such that*

$$\forall \alpha \in \mathsf{dom}(\sigma)\ .\ \mathit{var}(\sigma(\alpha)) \cap \mathsf{dom}(\sigma) = \emptyset$$

and

$$\forall \alpha\ =\ t \in E\ .\ \sigma(\alpha)\ \simeq\ t\sigma\ \mathit{holds}$$

Lemma 10.1.38. *Let E be an equation system. If $\sigma = \mathsf{Unify}(E)$, then* $\mathsf{dom}(\sigma) = \mathsf{dom}(E)$ *and* $\forall \alpha \in \mathsf{dom}(\sigma).\ \mathit{var}(\sigma(\alpha)) \cap \mathsf{dom}(\sigma) = \emptyset$.

Proof. The algorithm $\mathsf{Unify}()$ consists of two steps: (*i*) transform types into recursive types and (*ii*) extract the substitution. After the first step, for each equation $(\alpha = t_\alpha) \in E$, we have $\alpha \notin \mathsf{var}(t_\alpha)$.
Consider the second step. Let $\mathsf{var}(E) = \bigcup_{(\alpha = t_\alpha) \in E} \mathsf{var}(t_\alpha)$ and $\overline{S} = \mathcal{V} \setminus S$ where S

```
Require: an equation system E
Ensure: a substitution σ
 1. let e2mu (α, t_α) = (α, μx_α. t_α{x_α/α}) in
 2. let subst (α, t_α) (β, t_β) = (β, t_β{t_α/α}{x_β/β}) in
 3. let rec mu2sub E =
 4.   match E with
 5.   |[ ] → [ ]
 6.   |(α, t_α) :: E' →
 7.     let E'' = List.map (subst (α, t_α)) E' in
 8.     let σ' = mu2sub E'' in {t_α σ'/α} ⊕ σ'
 9. in
10. let e2sub E =
11.   let E' = List.map e2mu E in
12.   mu2sub E'
```

Figure 10.4: Equation system solving algorithm $\mathsf{Unify}()$

is a set of type variables. We prove a stronger statement: $\forall \alpha \in \mathsf{dom}(\sigma).\ \mathsf{var}(\sigma(\alpha)) \cap (\mathsf{dom}(\sigma) \cup \overline{\mathsf{var}(E)}) = \emptyset$ and $\mathsf{dom}(\sigma) = \mathsf{dom}(E)$. The proof proceeds by induction on E:

$E = \emptyset$**:** straightforward.

$E = \{(\alpha = t_\alpha)\} \cup E'$**:** let $E'' = \{(\beta = t_\beta\{t_\alpha/\alpha\}\{x_\beta/\beta\}) \mid (\beta = t_\beta) \in E'\}$. Then there exists a substitution σ'' such that $\sigma'' = \mathsf{Unify}(E'')$ and $\sigma = \{t_\alpha\sigma''/\alpha\} \oplus \sigma''$. By induction, we have $\forall \beta \in \mathsf{dom}(\sigma'').\ \mathsf{var}(\sigma''(\beta)) \cap (\mathsf{dom}(\sigma'') \cup \overline{\mathsf{var}(E'')}) = \emptyset$ and $\mathsf{dom}(\sigma'') = \mathsf{dom}(E'')$. As $\alpha \notin \mathsf{dom}(E'')$, we have $\alpha \notin \mathsf{dom}(\sigma'')$ and then $\mathsf{dom}(\sigma) = \mathsf{dom}(\sigma'') \cup \{\alpha\} = \mathsf{dom}(E)$.

Moreover, $\alpha \notin \mathsf{var}(E'')$, then $\mathsf{dom}(\sigma) \subset \mathsf{dom}(\sigma'') \cup \overline{\mathsf{var}(E'')}$. Thus, for all $\beta \in \mathsf{dom}(\sigma'')$, we have $\mathsf{var}(\sigma''(\beta)) \cap \mathsf{dom}(\sigma) = \emptyset$. Consider $t_\alpha\sigma''$. It is clear that $\mathsf{var}(t_\alpha\sigma'') \cap \mathsf{dom}(\sigma) = \emptyset$. Besides, the algorithm does not introduce any fresh variable, then for all $\beta \in \mathsf{dom}(\sigma)$, we have $\mathsf{var}(t_\beta) \cap \overline{\mathsf{var}(E)} = \emptyset$. Therefore, the result follows.

□

Lemma 10.1.39 (Soundness). *Let E be an equation system. If $\sigma = \mathsf{Unify}(E)$, then $\sigma \Vdash E$.*

Proof. By induction on E.

$E = \emptyset$**:** straightforward.

$E = \{(\alpha = t_\alpha)\} \cup E'$**:** let $E'' = \{(\beta = t_\beta\{t_\alpha/\alpha\}\{x_\beta/\beta\}) \mid (\beta = t_\beta) \in E'\}$. Then there exists a substitution σ'' such that $\sigma'' = \mathsf{Unify}(E'')$ and $\sigma = \{t_\alpha\sigma''/\alpha\} \oplus \sigma''$. By induction, we have $\sigma'' \Vdash E''$. According to Lemma 10.1.38, we have $\mathsf{dom}(\sigma'') = \mathsf{dom}(E'')$. So $\mathsf{dom}(\sigma) = \mathsf{dom}(\sigma'') \cup \{\alpha\}$. Considering any equation $(\beta = t_\beta) \in E$

where $\beta \in \mathsf{dom}(\sigma'')$. Then

$$\begin{array}{rcll}
\sigma(\beta) = & \sigma''(\beta) & & \text{(apply } \sigma\text{)} \\
\simeq & t_\beta\{t_\alpha/\alpha\}\{x_\beta/\beta\}\sigma'' & & \text{(as } \sigma'' \Vdash E''\text{)} \\
= & t_\beta\{t_\alpha\{x_\beta/\beta\}/\alpha, x_\beta/\beta\}\sigma'' & & \\
= & t_\beta\{t_\alpha\{x_\beta/\beta\}\sigma''/\alpha, x_\beta\sigma''/\beta\} \oplus \sigma'' & & \\
= & t_\beta\{t_\alpha(\{x_\beta\sigma''/\beta\} \oplus \sigma'')/\alpha, x_\beta\sigma''/\beta\} \oplus \sigma'' & & \\
\simeq & t_\beta\{t_\alpha(\{t_\beta\sigma''/\beta\} \oplus \sigma'')/\alpha, t_\beta\sigma''/\beta\} \oplus \sigma'' & & \text{(expand } x_\beta\text{)} \\
\simeq & t_\beta\{t_\alpha(\{\beta\sigma''/\beta\} \oplus \sigma'')/\alpha, \beta\sigma''/\beta\} \oplus \sigma'' & & \text{(as } \sigma'' \Vdash E''\text{)} \\
= & t_\beta\{t_\alpha\sigma''/\alpha\} \oplus \sigma'' & & \\
= & t_\beta\sigma & &
\end{array}$$

Finally, consider the equation $(\alpha = t_\alpha)$. As

$$\begin{array}{rcll}
\sigma(\alpha) = & t_\alpha\sigma'' & & \text{(apply } \sigma\text{)} \\
= & t_\alpha\{\beta\sigma''/\beta \mid \beta \in \mathsf{dom}(\sigma'')\} & & \text{(expand } \sigma''\text{)} \\
= & t_\alpha\{\beta\sigma/\beta \mid \beta \in \mathsf{dom}(\sigma'')\} & & \text{(as } \beta\sigma = \beta\sigma''\text{)} \\
= & t_\alpha\{\beta\sigma/\beta \mid \beta \in \mathsf{dom}(\sigma'') \cup \{\alpha\}\} & & \text{(as } \alpha \notin \mathsf{var}(t_\alpha)\text{)} \\
= & t_\alpha\{\beta\sigma/\beta \mid \beta \in \mathsf{dom}(\sigma)\} & & \text{(as } \mathsf{dom}(\sigma) = \mathsf{dom}(\sigma'') \cup \{\alpha\}\text{)} \\
= & t_\alpha\sigma & &
\end{array}$$

Thus, the result follows.

□

Lemma 10.1.40. *Let E be an equation system. If $\sigma = \mathsf{Unify}(E)$, then σ is a general solution to E.*

Proof. Immediate consequence of Lemmas 10.1.38 and 10.1.39. □

Clearly, given an equation system E, the algorithm $\mathsf{Unify}(E)$ terminates with a substitution σ.

Lemma 10.1.41. *Given an equation system E, the algorithm $\mathsf{Unify}(E)$ terminates with a substitution σ.*

Proof. By induction on the number of equations in E. □

Definition 10.1.42. *Let σ, σ' be two substitutions. We say $\sigma \simeq \sigma'$ if and only if $\forall\alpha.\ \sigma(\alpha) \simeq \sigma'(\alpha)$.*

Lemma 10.1.43 (Completeness). *Let E be an equation system. For all substitution σ, if $\sigma \Vdash E$, then there exist σ_0 and σ' such that $\sigma_0 = \mathsf{Unify}(E)$ and $\sigma \simeq \sigma' \circ \sigma_0$.*

Proof. According to Lemma 10.1.41, there exists σ_0 such that $\sigma_0 = \mathsf{Unify}(E)$. For any $\alpha \notin \mathsf{dom}(\sigma_0)$, clearly we have $\alpha\sigma_0\sigma = \alpha\sigma$ and then $\alpha\sigma_0\sigma \simeq \alpha\sigma$. What remains to prove is that if $\sigma \Vdash E$ and $\sigma_0 = \mathsf{Unify}(E)$ then $\forall\alpha \in \mathsf{dom}(\sigma_0).\ \alpha\sigma_0\sigma \simeq \alpha\sigma$. The proof proceeds by induction on E:

$E = \emptyset$: straightforward.

$E = \{(\alpha = t_\alpha)\} \cup E'$**:** let $E'' = \{(\beta = t_\beta\{t_\alpha/\alpha\}\{x_\beta/\beta\}) \mid (\beta = t_\beta) \in E'\}$. Then there exists a substitution σ'' such that $\sigma'' = \mathsf{Unify}(E'')$ and $\sigma_0 = \{t_\alpha\sigma''/\alpha\} \oplus \sigma''$. Considering each equation $(\beta = t_\beta\{t_\alpha/\alpha\}\{x_\beta/\beta\}) \in E''$, we have

$$\begin{array}{rcll} t_\beta\{t_\alpha/\alpha\}\{x_\beta/\beta\}\sigma = & t_\beta\{t_\alpha\{x_\beta/\beta\}/\alpha, x_\beta/x_\beta\}\sigma & \\ = & t_\beta\{t_\alpha\{x_\beta/\beta\}\sigma/\alpha, x_\beta\sigma/\beta\} \oplus \sigma & \\ = & t_\beta\{t_\alpha(\{x_\beta\sigma/\beta\} \oplus \sigma)/\alpha, x_\beta\sigma/\beta\} \oplus \sigma & \\ \simeq & t_\beta\{t_\alpha(\{t_\beta\sigma/\beta\} \oplus \sigma)/\alpha, t_\beta\sigma/\beta\} \oplus \sigma & (\text{expand } x_\beta) \\ \simeq & t_\beta\{t_\alpha(\{\beta\sigma/\beta\} \oplus \sigma)/\alpha, \beta\sigma/\beta\} \oplus \sigma & (\text{as } \sigma \Vdash E) \\ = & t_\beta\{t_\alpha\sigma/\alpha\} \oplus \sigma & \\ \simeq & t_\beta\{\alpha\sigma/\alpha\} \oplus \sigma & \\ = & t_\beta\sigma & \\ \simeq & \beta\sigma & \end{array}$$

Therefore, $\sigma \Vdash E''$. By induction on E'', we have $\forall\beta \in \mathsf{dom}(\sigma'').\ \beta\sigma''\sigma \simeq \beta\sigma$. According to Lemma 10.1.38, $\mathsf{dom}(\sigma'') = \mathsf{dom}(E'')$. As $\alpha \notin \mathsf{dom}(E'')$, then $\mathsf{dom}(\sigma_0) = \mathsf{dom}(\sigma'') \cup \{\alpha\}$. Therefore for any $\beta \in \mathsf{dom}(\sigma'') \cap \mathsf{dom}(\sigma_0)$, $\beta\sigma_0\sigma \simeq \beta\sigma''\sigma \simeq \beta\sigma$. Finally, considering α, we have

$$\begin{array}{rcll} \alpha\sigma_0\sigma = & t_\alpha\sigma''\sigma & (\text{apply } \sigma_0) \\ = & t_\alpha\{\beta\sigma''/\beta \mid \beta \in \mathsf{dom}(\sigma'')\}\sigma & (\text{expand } \sigma'') \\ = & t_\alpha\{\beta\sigma''\sigma/\beta \mid \beta \in \mathsf{dom}(\sigma'')\} \oplus \sigma & \\ \simeq & t_\alpha\{\beta\sigma/\beta \mid \beta \in \mathsf{dom}(\sigma'')\} \oplus \sigma & (\text{as } \forall\beta \in \sigma''.\ \beta\sigma \simeq \beta\sigma''\sigma) \\ = & t_\alpha\sigma & \\ \simeq & \alpha\sigma & (\text{as } \sigma \Vdash E) \end{array}$$

Therefore, the result follows.

□

In our calculus, a type is well-formed if and only if the recursion traverses a constructor. In other words, the recursive variable should not appear at the top level of the recursive content. For example, the type $\mu x.\ x \vee t$ is not well-formed. To make the substitutions usable, we should avoid these substitutions with ill-formed types. Fortunately, this can been done by giving an ordering on the domain of an equation system to make sure that the equation system is well-ordered.

Lemma 10.1.44. *Let E be a well-ordered equation system. If $\sigma = \mathsf{Unify}(E)$, then for all $\alpha \in \mathsf{dom}(\sigma)$, $\sigma(\alpha)$ is well-formed.*

Proof. Assume that there exists an ill-formed $\sigma(\alpha)$. That is, $\sigma(\alpha) = \mu x.\ t$ where x occurs at the top level of t. According to the algorithm $\mathsf{Unify}()$, there exists a sequence of equations $(\alpha =)\alpha_0 = t_{\alpha_0}, \alpha_1 = t_{\alpha_1}, \ldots, \alpha_n = t_{\alpha_n}$ such that $\alpha_i \in \mathsf{tlv}(t_{\alpha_{i-1}})$ and $\alpha_0 \in \mathsf{tlv}(t_{\alpha_n})$ where $i \in \{1, \ldots, n\}$ and $n \geq 0$. According to Definition 10.1.35, $O(\alpha_{i-1}) < O(\alpha_i)$ and $O(\alpha_n) < O(\alpha_0)$. Therefore, we have $O(\alpha_0) < O(\alpha_1) < \ldots < O(\alpha_n) < O(\alpha_0)$, which is impossible. Thus the result follows. □

As mentioned above, there may be some useless recursion constructor μ. They can be eliminated by checking whether the recursive variable appears in the content

expression or not. Moreover, if a recursive type is empty (which can be checked with the subtyping algorithm), then it can be replaced by $\mathbb{0}$.

To conclude, we now describe the solving procedure $\mathsf{Sol}_\Delta(C)$ for the type tallying problem C. We first normalize C into a finite set $\mathcal{S}$ of well-ordered normalized constraint-sets. If $\mathcal{S}$ is empty, then there are no solutions to C. Otherwise, each constraint-set $C_i \in \mathcal{S}$ is merged and saturated into a finite set $\mathcal{S}_{C_i}$ of well-order saturated normalized constraint-sets. Then all these sets are collected into another set $\mathcal{S}'$ (*i.e.*, $\mathcal{S}' = \bigsqcup_{C_i \in \mathcal{S}} \mathcal{S}_{C_i}$). If $\mathcal{S}'$ is empty, then there are no solutions to C. Otherwise, for each constraint-set $C'_i \in \mathcal{S}'$, we transform C'_i into an equation system E_i and then construct a general solution σ_i from E_i. Finally, we collect all the solutions σ_i, yielding a set Θ of solutions to C. We write $\mathsf{Sol}_\Delta(C) \rightsquigarrow \Theta$ if $\mathsf{Sol}_\Delta(C)$ terminates with Θ, and we call Θ the solution of the type tallying problem C.

Theorem 10.1.45 (**Soundness**). *Let C be a constraint-set. If $\mathsf{Sol}_\Delta(C) \rightsquigarrow \Theta$, then for all $\sigma \in \Theta$, $\sigma \Vdash C$.*

Proof. Consequence of Lemmas 10.1.10, 10.1.17, 10.1.20, 10.1.26, 10.1.28, 10.1.33 and 10.1.39. □

Theorem 10.1.46 (**Completeness**). *Let C be a constraint-set and $\mathsf{Sol}_\Delta(C) \rightsquigarrow \Theta$. Then for all substitution σ, if $\sigma \Vdash C$, then there exists $\sigma' \in \Theta$ and σ'' such that $\sigma \approx \sigma'' \circ \sigma'$.*

Proof. Consequence of Lemmas 10.1.12, 10.1.21, 10.1.34 and 10.1.43. □

Theorem 10.1.47 (**Decidability**). *Let C be a constraint-set. Then $\mathsf{Sol}_\Delta(C)$ terminates.*

Proof. Consequence of Lemmas 10.1.14, 10.1.22 and 10.1.41. □

Lemma 10.1.48. *Let C be a constraint-set and $\mathsf{Sol}_\Delta(C) \rightsquigarrow \Theta$. Then*

(1) *Θ is finite.*

(2) *for all $\sigma \in \Theta$ and for all $\alpha \in \mathsf{dom}(\sigma)$, $\sigma(\alpha)$ is well-formed.*

Proof. (1): Consequence of Lemmas 10.1.15 and 10.1.27.

(2): Consequence of Lemmas 10.1.17, 10.1.28, 10.1.36 and 10.1.44.

□

10.2 Type-substitutions inference algorithm

In Chapter 9 , we presented a sound and complete inference system, which is parametric in the decision procedures for $\sqsubseteq_\Delta$, $\amalg^i_\Delta()$, and $\bullet_\Delta$. In this section we tackle the problem of computing these operators. We focus on the application problem $\bullet_\Delta$, since the other

two can be solved similarly. Recall that to compute $t \bullet_\Delta s$, we have to find two sets of substitutions $[\sigma_i]_{i\in I}$ and $[\sigma_j]_{j\in J}$ such that $\forall h \in I \cup J.\ \sigma_h \sharp \Delta$ and

$$\bigwedge_{i\in I} t\sigma_i \leq \mathbb{0} \to \mathbb{1} \tag{10.1}$$

$$\bigwedge_{j\in J} s\sigma_j \leq \mathsf{dom}(\bigwedge_{i\in I} t\sigma_i) \tag{10.2}$$

This problem is more general than the other two problems. If we are able to decide inequation (10.2), it means that we are able to decide $s' \sqsubseteq_\Delta t'$ for any s' and t', just by considering t' ground. Therefore we can decide $\sqsubseteq_\Delta$. We can also decide $[\sigma_i]_{i\in I} \Vdash s \sqsubseteq_\Delta \mathbb{1} \times \mathbb{1}$ for all s, and therefore compute $\Pi^i_\Delta(s)$.

Let the cardinalities of I and J be p and q respectively. We first show that for fixed p and q, we can reduce the application problem to a type tallying problem. Note that if we increase p, the type on the right of Inequality (10.2) is larger, and if we increase q the type on the left is smaller. Namely, the larger p and q are, the higher the chances that the inequality holds. Therefore, we can search for cardinalities that make the inequality hold by starting from $p = q = 1$, and then by increasing p and q in a dove-tail order [Pys76] until we get a solution. This gives us a semi-decision procedure for the general application problem. For the implementation, we give some heuristics based on the shapes of s and t to set upper bounds for p and q (see Section 10.2.3).

10.2.1 Application problem with fixed cardinalities

We explain how to reduce the application problem with fixed cardinalities for I and J to a type tallying problem. Without loss of generality, we can split each substitution σ_k ($k \in I \cup J$) into two substitutions: a renaming substitution ρ_k that maps each variable in the domain of σ_k into a fresh variable and a second substitution σ'_k such that $\sigma_k = \sigma'_k \circ \rho_k$. The two inequalities then can be rewritten as

$$\bigwedge_{i\in I} (t\rho_i)\sigma'_i \leq \mathbb{0} \to \mathbb{1}$$

$$\bigwedge_{j\in J} (s\rho_j)\sigma'_j \leq \mathsf{dom}(\bigwedge_{i\in I} (t\rho_i)\sigma'_i)$$

The domains of the substitutions σ'_k are pairwise distinct, since they are composed by fresh type variables. We can therefore merge the σ'_k into one substitution $\sigma = \bigcup_{k\in I\cup J} \sigma'_k$. We can then further rewrite the two inequalities as

$$(\bigwedge_{i\in I} (t\rho_i))\sigma \leq \mathbb{0} \to \mathbb{1}$$

$$(\bigwedge_{j\in J} (s\rho_j))\sigma \leq \mathsf{dom}((\bigwedge_{i\in I} (t\rho_i))\sigma)$$

which are equivalent to

$$t'\sigma \leq \mathbb{0} \to \mathbb{1}$$

$$s'\sigma \leq \mathsf{dom}(t'\sigma)$$

where $t' = (\bigwedge_{i\in I} t\rho_i)$ and $s' = (\bigwedge_{j\in J} s\rho_j)$. As $t'\sigma \leq \mathbb{0} \to \mathbb{1}$, then $t'\sigma$ must be a function type. Then according to Lemmas 8.2.9 and 8.2.10, we can reduce these two inequalities to the constraint set[4]:

$$C = \{(t', \leq, \mathbb{0} \to \mathbb{1}), (t', \leq, s' \to \gamma)\}$$

where γ is a fresh type variable. We have reduced the original application problem $t \bullet_\Delta s$ to solving C, which can be done as explained in Section 10.1. We write $\mathsf{AppFix}_\Delta(t, s)$ for the algorithm of the application problem (with fixed cardinalities) $t \bullet_\Delta s$ and $\mathsf{AppFix}_\Delta(t, s) \rightsquigarrow \Theta$ if $\mathsf{AppFix}_\Delta(t, s)$ terminates with Θ.

Lemma 10.2.1. *Let t, s be two types and γ a type variable such that $\gamma \notin \mathsf{var}(t) \cup \mathsf{var}(s)$. Then for all substitution σ, if $t\sigma \leq s\sigma \to \gamma\sigma$, then $s\sigma \leq \mathsf{dom}(t\sigma)$ and $\sigma(\gamma) \geq t\sigma \cdot s\sigma$.*

Proof. Consider any substitution σ. As $t\sigma \leq s\sigma \to \gamma\sigma$, by Lemma 8.2.9, we have $s\sigma \leq \mathsf{dom}(t\sigma)$. Then by Lemma 8.2.10, we get $\sigma(\gamma) \geq t\sigma \cdot s\sigma$. □

Lemma 10.2.2. *Let t, s be two types and γ a type variable such that $\gamma \notin \mathsf{var}(t) \cup \mathsf{var}(s)$. Then for all substitution σ, if $s\sigma \leq \mathsf{dom}(t\sigma)$ and $\gamma \notin \mathsf{dom}(\sigma)$, then there exists σ' such that $\sigma' \sharp \sigma$ and $t(\sigma \cup \sigma') \leq (s \to \gamma)(\sigma \cup \sigma')$.*

Proof. Consider any substitution σ. As $s\sigma \leq \mathsf{dom}(t\sigma)$, by Lemma 8.2.10, the type $(t\sigma) \cdot (s\sigma)$ exists and $t\sigma \leq s\sigma \to ((t\sigma) \cdot (s\sigma))$. Let $\sigma' = \{(t\sigma) \cdot (s\sigma)/\gamma\}$. Then

$$\begin{array}{rcl} t(\sigma \cup \sigma') & = & t\sigma \\ & \leq & s\sigma \to ((t\sigma) \cdot (s\sigma)) \\ & = & s\sigma \to \gamma\sigma' \\ & = & (s \to \gamma)(\sigma \cup \sigma') \end{array}$$

□

Note that the solution of the γ introduced in the constraint $(t, \leq, s \to \gamma)$ represents a result type for the application of t to s. In particular, completeness for the tallying problem ensures that each solution will assign to γ (which occurs in a covariant position) the minimum type for that solution. So the minimum solutions for γ are in $t \bullet_\Delta s$ (see the substitution $\sigma'(\gamma) = (t\sigma) \cdot (s\sigma)$ in the proof of Lemma 10.2.2).

Theorem 10.2.3 (Soundness). *Let t and s be two types. If $\mathsf{AppFix}_\Delta(t, s) \rightsquigarrow \Theta$, then for all $\sigma \in \Theta$, we have $t\sigma \leq \mathbb{0} \to \mathbb{1}$ and $s\sigma \leq \mathsf{dom}(t\sigma)$.*

Proof. Consequence of Lemmas 10.2.1 and 10.1.45. □

Theorem 10.2.4 (Completeness). *Let t and s be two types and $\mathsf{AppFix}_\Delta(t, s) \rightsquigarrow \Theta$. For all substitution σ, if $t\sigma \leq \mathbb{0} \to \mathbb{1}$ and $s\sigma \leq \mathsf{dom}(t\sigma)$, then there exists $\sigma' \in \Theta$ and σ'' such that $\sigma \simeq \sigma'' \circ \sigma'$.*

Proof. Consequence of Lemmas 10.2.2 and 10.1.46. □

[4]The first constraint $(t', \leq, \mathbb{0} \to \mathbb{1})$ can be eliminated since it is implied by the second one.

10.2.2 General application problem

Now we take the cardinalities of I and J into account to solve the general application problem. As stated before, we start with I and J both of cardinality 1 and explore all the possible combinations of the cardinalities of I and J by, say, a dove-tail order [Pys76] until we get a solution. More precisely, the algorithm consists of two steps:

Step A**:** we generate a constraint set as explained in Section 10.2.1 and apply the tallying solving algorithm described in Section 10.1, yielding either a solution or a failure.

Step B**:** if all attempts to solve the constraint sets have failed at `Step` 1 of the tallying solving algorithm given at the beginning of Section 10.1.1, then fail (the expression is not typable). If they all failed but at least one did not fail in `Step` 1, then increment the cardinalities I and J to their successor in the dove-tail order and start from `Step A` again. Otherwise all substitutions found by the algorithm are solutions of the application problem.

Notice that the algorithm returns a failure only if the solving of the constraint-set fails at `Step` 1 of the algorithm for the tallying problem. The reason is that up to `Step` 1 all the constraints at issue are on distinct occurrences of type variables: if they fail there is no possible expansion that can make the constraint-set satisfiable (see Lemma 10.2.5). For example, the function `map` can not be applied to any integer, as the normalization of $\{(\texttt{Int}, \leq, \alpha \to \beta)\}$ is empty (and even for any expansion of $\alpha \to \beta$). In `Step` 2 instead constraints of different occurrences of a same variable are merged. Thus even if the constraints fail it may be the case that they will be satisfied by expanding different occurrences of a same variable into different variables. Therefore an expansion is tried. For example, consider the application of a function of type $((\texttt{Int} \to \texttt{Int}) \wedge (\texttt{Bool} \to \texttt{Bool})) \to t$ to an argument of type $\alpha \to \alpha$. We start with the constraint $(\alpha \to \alpha, \leq, (\texttt{Int} \to \texttt{Int}) \wedge (\texttt{Bool} \to \texttt{Bool}))$. The tallying algorithm first normalizes it into the set $\{(\alpha, \leq, \texttt{Int}), (\alpha, \geq, \texttt{Int}), (\alpha, \leq, \texttt{Bool}), (\alpha, \geq, \texttt{Bool})\}$ (*i.e.*, `Step` 1). But it fails at `Step` 2 as neither $\texttt{Int} \leq \texttt{Bool}$ nor $\texttt{Bool} \leq \texttt{Int}$ hold. However, if we expand $\alpha \to \alpha$, the constraint to be solved becomes

$$((\alpha_1 \to \alpha_1) \wedge (\alpha_2 \to \alpha_2), \leq, (\texttt{Int} \to \texttt{Int}) \wedge (\texttt{Bool} \to \texttt{Bool}))$$

and one of the constraint-set of its normalization is

$$\{(\alpha_1, \leq, \texttt{Int}), (\alpha_1, \geq, \texttt{Int}), (\alpha_2, \leq, \texttt{Bool}), (\alpha_2, \geq, \texttt{Bool})\}$$

The conflict between `Int` and `Bool` disappears and we can find a solution to the expanded constraint.

Note that we keep trying expansion without giving any bound on the cardinalities I and J, so the procedure may not terminate, which makes it only a semi-algorithm. The following lemma justifies why we do not try to expand if normalization (i.e., Step 1 of the tallying algorithm) fails.

Lemma 10.2.5. *Let t, s be two types, γ a fresh type variable and $[\rho_i]_{i\in I}$, $[\rho_j]_{j\in J}$ two sets of general renamings. If $\emptyset \vdash_{\mathcal{N}} \{(t, \leq, \mathbb{0} \to \mathbb{1}), (t, \leq, s \to \gamma)\} \rightsquigarrow \emptyset$, then $\emptyset \vdash_{\mathcal{N}} \{(\bigwedge_{i\in I} t\rho_i, \leq, \mathbb{0} \to \mathbb{1}), (\bigwedge_{i\in I} t\rho_i, \leq, (\bigwedge_{j\in J} s\rho_j) \to \gamma)\} \rightsquigarrow \emptyset$.*

Proof. As $\emptyset \vdash_{\mathcal{N}} \{(t, \leq, \mathbb{0} \to \mathbb{1}), (t, \leq, s \to \gamma)\} \rightsquigarrow \emptyset$, then either $\emptyset \vdash_{\mathcal{N}} \{(t, \leq, \mathbb{0} \to \mathbb{1})\} \rightsquigarrow \emptyset$ or $\emptyset \vdash_{\mathcal{N}} \{(t, \leq, s \to \gamma)\} \rightsquigarrow \emptyset$. If the first one holds, then according to Lemma 10.1.19, we have $\emptyset \vdash_{\mathcal{N}} \{(\bigwedge_{i\in I} t\rho_i, \leq, \mathbb{0} \to \mathbb{1})\} \rightsquigarrow \emptyset$, and *a fortiori*

$$\emptyset \vdash_{\mathcal{N}} \{(\bigwedge_{i\in I} t\rho_i, \leq, \mathbb{0} \to \mathbb{1}), (\bigwedge_{i\in I} t\rho_i, \leq, (\bigwedge_{j\in J} s\rho_j) \to \gamma)\} \rightsquigarrow \emptyset$$

Assume that $\emptyset \vdash_{\mathcal{N}} \{(t, \leq, s \to \gamma)\} \rightsquigarrow \emptyset$. Without loss of generality, we consider the disjunctive normal form τ of t:

$$\tau = \bigvee_{k_b\in K_b} \tau_{k_b} \vee \bigvee_{k_p\in K_p} \tau_{k_p} \vee \bigvee_{k_a\in K_a} \tau_{k_a}$$

where τ_{k_b} (τ_{k_p} and τ_{k_a} resp.) is an intersection of basic types (products and arrows resp.) and type variables. Then there must exist $k \in K_b \cup K_p \cup K_a$ such that $\emptyset \vdash_{\mathcal{N}} \{(\tau_k, \leq, \mathbb{0} \to \mathbb{1})\} \rightsquigarrow \emptyset$. If $k \in K_b \cup K_p$, then the constraint $(\tau_k, \leq, s \to \gamma)$ is equivalent to $(\tau_k, \leq, \mathbb{0})$. By Lemma 10.1.19, we get $\emptyset \vdash_{\mathcal{N}} \{(\bigwedge_{i\in I} \tau_k\rho_i, \leq, \mathbb{0})\} \rightsquigarrow \emptyset$, that is, $\emptyset \vdash_{\mathcal{N}} \{(\bigwedge_{i\in I} \tau_k\rho_i, \leq, (\bigwedge_{j\in J} s\rho_j) \to \gamma)\} \rightsquigarrow \emptyset$. So the result follows.
Otherwise, it must be that $k \in K_a$ and $\tau_k = \bigwedge_{p\in P}(w_p \to v_p) \wedge \bigwedge_{n\in N} \neg(w_n \to v_n)$. We claim that $\emptyset \vdash_{\mathcal{N}} \{(\tau_k, \leq, \mathbb{0})\} \rightsquigarrow \emptyset$ (otherwise, $\emptyset \vdash_{\mathcal{N}} \{(\tau_k, \leq, s \to \gamma)\} \rightsquigarrow \emptyset$ does not hold). Applying Lemma 10.1.19 again, we get $\emptyset \vdash_{\mathcal{N}} \{(\bigwedge_{i\in I} \tau_k\rho_i, \leq, \mathbb{0})\} \rightsquigarrow \emptyset$. Moreover, following the rule (Narrow), there exists a set $P' \subseteq P$ such that

$$\begin{cases} \emptyset \vdash_{\mathcal{N}} \{ \bigwedge_{p\in P'} \neg w_p \wedge s, \leq, \mathbb{0})\} \rightsquigarrow \emptyset \\ P' = P \text{ or } \emptyset \vdash_{\mathcal{N}} \{ \bigwedge_{p\in P\setminus P'} v_p \wedge \neg\gamma, \leq, \mathbb{0})\} \rightsquigarrow \emptyset \end{cases}$$

Applying 10.1.19, we get

$$\begin{cases} \emptyset \vdash_{\mathcal{N}} \{\bigwedge_{i\in I}(\bigwedge_{p\in P'} \neg w_p)\rho_i \wedge \bigwedge_{j\in J} s\rho_j, \leq, \mathbb{0})\} \rightsquigarrow \emptyset \\ P' = P \text{ or } \emptyset \vdash_{\mathcal{N}} \{\bigwedge_{i\in I}(\bigwedge_{p\in P\setminus P'} v_p)\rho_i \wedge \neg\gamma, \leq, \mathbb{0})\} \rightsquigarrow \emptyset \end{cases}$$

By the rule (Narrow), we have

$$\emptyset \vdash_{\mathcal{N}} \{(\bigwedge_{i\in I}(\bigwedge_{p\in P} (w_p \to v_p))\rho_i, \leq, (\bigwedge_{j\in J} s\rho_j) \to \gamma)\} \rightsquigarrow \emptyset$$

Therefore, we have $\emptyset \vdash_{\mathcal{N}} \{(\bigwedge_{i\in I} \tau_k\rho_i, \leq, (\bigwedge_{j\in J} s\rho_j) \to \gamma)\} \rightsquigarrow \emptyset$. So the result follows. □

Let $\mathsf{App}_\Delta(t, s)$ denote the algorithm for the general application problem.

Theorem 10.2.6. *Let t, s be two types and γ the special fresh type variable introduced in $(\bigwedge_{i\in I} t\sigma_i, \leq, (\bigwedge_{j\in J} s\sigma_j) \to \gamma)$. If $\mathsf{App}_\Delta(t, s)$ terminates with Θ, then*

(1) **(Soundness)** *if $\Theta \neq \emptyset$, then for each $\sigma \in \Theta$, $\sigma(\gamma) \in t \bullet_\Delta s$.*

(2) **(Weak completeness)** *if* $\Theta = \emptyset$, *then* $t \bullet_\Delta s = \emptyset$.

Proof. (1)**:** consequence of Theorem 10.2.3 and Lemma 10.2.1.
(2)**:** consequence of Lemma 10.2.5.

□

Let us consider the application `map even` in Section 1.4 again. The types of `map` and `even` are

$$\begin{array}{rcl} \texttt{map} & :: & (\alpha \to \beta) \to [\alpha] \to [\beta] \\ \texttt{even} & :: & (\texttt{Int} \to \texttt{Bool}) \wedge ((\alpha \setminus \texttt{Int}) \to (\alpha \setminus \texttt{Int})) \end{array}$$

We start with the constraint-set

$$C_1 = \{(\alpha_1 \to \beta_1) \to [\alpha_1] \to [\beta_1] \leq ((\texttt{Int} \to \texttt{Bool}) \wedge ((\alpha \setminus \texttt{Int}) \to (\alpha \setminus \texttt{Int}))) \to \gamma\}$$

where γ is a fresh type variable (and where we α-converted the type of `map`). Then the algorithm $\mathsf{Sol}_\Delta(C_1)$ generates a set of eight constraint-sets at `Step 2` :

$$\begin{array}{l} \{\gamma \geq [\alpha_1] \to [\beta_1], \alpha_1 \leq \mathbb{0}\} \\ \{\gamma \geq [\alpha_1] \to [\beta_1], \alpha_1 \leq \mathbb{0}, \beta_1 \geq \texttt{Bool}\} \\ \{\gamma \geq [\alpha_1] \to [\beta_1], \alpha_1 \leq \mathbb{0}, \beta_1 \geq \alpha \setminus \texttt{Int}\} \\ \{\gamma \geq [\alpha_1] \to [\beta_1], \alpha_1 \leq \mathbb{0}, \beta_1 \geq \texttt{Bool} \vee (\alpha \setminus \texttt{Int})\} \\ \{\gamma \geq [\alpha_1] \to [\beta_1], \alpha_1 \leq \mathbb{0}, \beta_1 \geq \texttt{Bool} \wedge (\alpha \setminus \texttt{Int})\} \\ \{\gamma \geq [\alpha_1] \to [\beta_1], \alpha_1 \leq \texttt{Int}, \beta_1 \geq \texttt{Bool}\} \\ \{\gamma \geq [\alpha_1] \to [\beta_1], \alpha_1 \leq \alpha \setminus \texttt{Int}, \beta_1 \geq \alpha \setminus \texttt{Int}\} \\ \{\gamma \geq [\alpha_1] \to [\beta_1], \alpha_1 \leq \texttt{Int} \vee \alpha, \beta_1 \geq \texttt{Bool} \vee (\alpha \setminus \texttt{Int})\} \end{array}$$

Clearly, the solutions to the 2nd-5th constraint-sets are included in those to the first constraint-set. For the other four constraint-sets, by minimum instantiation, we can get four solutions for γ (*i.e.*, the result types of `map even`): $[\,] \to [\,]$, or $[\texttt{Int}] \to [\texttt{Bool}]$, or $[\alpha \setminus \texttt{Int}] \to [\alpha \setminus \texttt{Int}]$, or $[\texttt{Int} \vee \alpha] \to [\texttt{Bool} \vee (\alpha \setminus \texttt{Int})]$. Of these solutions only the last two are minimal (the first type is an instance of the third one and the second is an instance of the fourth one) and since both are valid we can take their intersection, yielding the (minimum) solution

$$([\alpha \setminus \texttt{Int}] \to [\alpha \setminus \texttt{Int}]) \wedge ([\texttt{Int} \vee \alpha] \to [\texttt{Bool} \vee (\alpha \setminus \texttt{Int})]) \tag{10.3}$$

Alternatively, we can dully follow the algorithm, perform an iteration, expand the type of the function, yielding the constraint-set

$$\begin{array}{l} \{((\alpha_1 \to \beta_1) \to [\alpha_1] \to [\beta_1]) \wedge ((\alpha_2 \to \beta_2) \to [\alpha_2] \to [\beta_2]) \\ \quad \leq ((\texttt{Int} \to \texttt{Bool}) \wedge ((\alpha \setminus \texttt{Int}) \to (\alpha \setminus \texttt{Int}))) \to \gamma\} \end{array}$$

from which we get the type (10.3) directly.

As stated in Section 10.1, we chose an arbitrary ordering on type variables, which affects the generated substitutions and then the resulting types. Assume that σ_1 and σ_2 are two type substitutions generated by different orderings. Thanks to the completeness of the tallying problem, there exist σ_1' and σ_2' such that $\sigma_2 \simeq \sigma_1' \circ \sigma_1$ and $\sigma_1 \simeq \sigma_2' \circ \sigma_2$. Therefore, the result types corresponding to σ_1 and σ_2 are equivalent under $\sqsubseteq_\Delta$, that is $\sigma_1(\gamma) \sqsubseteq_\Delta \sigma_2(\gamma)$ and $\sigma_2(\gamma) \sqsubseteq_\Delta \sigma_1(\gamma)$. However, this does not imply that $\sigma_1(\gamma) \simeq \sigma_2(\gamma)$.

For example, $\alpha \sqsubseteq_\Delta \mathbb{0}$ and $\mathbb{0} \sqsubseteq_\Delta \alpha$, but $\alpha \not\simeq \mathbb{0}$. Moreover, some result types are easier to understand or more precise than some others. Which one is better is a language design and implementation problem[5]. For example, consider the `map even` again. The type (10.3) is obtained under the ordering $o(\alpha_1) < o(\beta_1) < o(\alpha)$. While under the ordering $o(\alpha) < o(\alpha_1) < o(\beta_1)$, we would instead get

$$([\beta \setminus \texttt{Int}] \to [\beta]) \wedge ([\texttt{Int} \vee \texttt{Bool} \vee \beta] \to [\texttt{Bool} \vee \beta]) \tag{10.4}$$

It is clear that (10.3) $\sqsubseteq_\emptyset$ (10.4) and (10.4) $\sqsubseteq_\emptyset$ (10.3). However, compared with (10.3), (10.4) is less precise and less comprehensible, if we look at the type $[\texttt{Int} \vee \texttt{Bool} \vee \beta] \to [\texttt{Bool} \vee \beta]$: (1) there is a `Bool` in the domain which is useless here and (2) we know that `Int` cannot appear in the returned list, but this is not expressed in the type.

Besides, note that the tallying algorithm will generate some solutions based on (i) the fact that $\mathbb{0} \to t$ contains all the functions, or (ii) the fact that $(\mathbb{0} \times t)$ or $(t \times \mathbb{0})$ is a subtype of any type. Most of these solutions would yield useless types for the application problem. If there exist some other solutions, then any one can be taken as the result type. For example, consider the application of a function f of type $(\alpha \to \alpha) \to (\alpha \to \alpha)$ to an argument of type $\texttt{Int} \to \texttt{Int}$. The constraint-set to be solved is $\{(\texttt{Int} \to \texttt{Int} \leq \alpha \to \alpha)\}$, to which there are two solutions: one is $\{\mathbb{0}/\alpha\}$ and the other is $\{\texttt{Int}/\alpha\}$. The first one is generated from the fact (i) and yields the useless type $\mathbb{0} \to \mathbb{0}$ (as it provides little information). While the second one yields the type $\texttt{Int} \to \texttt{Int}$, which can be taken as the result type for the application. Otherwise, although the application is typable, it could not be used further. For instance, if f applies to an argument e of type $\texttt{Int} \to \texttt{Bool}$, then the constraint-set to be solved is $\{(\texttt{Int} \to \texttt{Bool} \leq \alpha \to \alpha)\}$, which has only one solution $\{\mathbb{0}/\alpha\}$. Thus, the result type would be $\mathbb{0} \to \mathbb{0}$. At present, we focus on whether an expression is typable, while whether and how the useless solutions can be eliminated is left to the implementation.

There is a final word on completeness, which states that for every solution of the application problem, our algorithm finds a solution that is more general. However this solution is not necessarily the first one found by the algorithm: even if we find a solution, continuing with a further expansion may yield a more general solution. We have just seen that, in the case of `map even`, the good solution is the second one, although this solution could have already been deduced by intersecting the first minimal solutions we found. Another simple example is the case of the application of a function of type $(\alpha \times \beta) \to (\beta \times \alpha)$ to an argument of type $(\texttt{Int} \times \texttt{Bool}) \vee (\texttt{Bool} \times \texttt{Int})$. For this application our algorithm returns after one iteration the type $(\texttt{Int} \vee \texttt{Bool}) \times (\texttt{Int} \vee \texttt{Bool})$ (since it unifies α with β) while one further iteration allows the system to deduce the more precise type $(\texttt{Int} \times \texttt{Bool}) \vee (\texttt{Bool} \times \texttt{Int})$. Of course this raises the problem of the existence of principal types: may an infinite sequence of increasingly general solutions exist? This is a problem we did not tackle in this work, but if the answer to the previous question were negative then it would be easy to prove the existence of a principal type: since at each iteration there are only finitely many solutions, then the principal type would be the intersection of the minimal solutions of the last iteration (how to decide that an iteration is the last one is yet another problem).

[5] In the current implementation we assume that the type variables in the function type always have smaller orders than those in the argument type.

10.2.3 Heuristics to stop type-substitutions inference

We only have a semi-algorithm for $t \bullet_{\Delta} s$ because, as long as we do not find a solution, we may increase the cardinalities of I and J (where I and J are defined as in the previous sections) indefinitely. In this section, we propose two heuristic numbers p and q for the cardinalities of I and J that are established according to the form of s and t. These heuristic numbers set the upper limit for the procedure: if no solution is found when the cardinalities of I and J have reached these heuristic numbers, then the procedure stops returning failure. This yields a terminating algorithm for $t \bullet_{\Delta} s$ which is clearly sound but, in our case, not complete. Whether it is possible to define these boundaries so that they ensure termination *and* completeness is still an open issue.

Through some examples, we first analyze the reasons why one needs to expand the function type t and/or the argument type s: the intuition is that type connectives are what makes the expansions necessary. Then based on this analysis, we give some heuristic numbers for the copies of types that are needed by the expansions. These heuristics follow some simple (but, we believe, reasonable) guidelines. First, when the substitutions found for a given p and q yield a useless type (*e.g.*, "$\mathbb{0} \to \mathbb{0}$" the type of a function that cannot be applied to any value), it seems sensible to expand the types (*i.e.*, increase p or q), in order to find more informative substitutions. Second, if iterating the process does not give a more precise type (in the sense of $\sqsubseteq$), then it seems sensible to stop. Last, when the process continuously yields more and more precise types, we choose to stop when the type is "good enough" for the programmer. In particular we choose to avoid to introduce too many new fresh variables that make the type arbitrarily more precise but at the same time less "programmer friendly". We illustrate these behaviours for three strategies: increasing p (that is, expanding the domain of the function), increasing q (that is, expanding the type of the argument) or lastly increasing both p and q at the same time.

Expansion of t

A simple reason to expand t is the presence of (top-level) unions in s. Generally, it is better to have as many copies of t as there are disjunctions in s. Consider the example,

$$\begin{array}{l} t = (\alpha \to \alpha) \to (\alpha \to \alpha) \\ s = (\texttt{Int} \to \texttt{Int}) \vee (\texttt{Bool} \to \texttt{Bool}) \end{array} \tag{10.5}$$

If we do not expand t (*ie*, if p is 1), then the result type computed for the application of t to s is $\mathbb{0} \to \mathbb{0}$. However, this result type cannot be applied hereafter, since its domain is $\mathbb{0}$, and is therefore useless (more precisely, it can be applied only to expressions that are provably diverging). When p is 2, we get an extra result type, $(\texttt{Int} \to \texttt{Int}) \vee (\texttt{Bool} \to \texttt{Bool})$, which is obtained by instantiating t twice, by `Int` and `Bool` respectively. Carrying on expanding t does not give more precise result types, as we always select only two copies of t to match the two summands in s, according to the decomposition rule for arrows (*ie*, Lemma 4.3.10).

A different example that shows that the cardinality of the summands in the union type of the argument is a good heuristic choice for p is the following one:

$$\begin{array}{l} t = (\alpha \times \beta) \to (\beta \times \alpha) \\ s = (\texttt{Int} \times \texttt{Bool}) \vee (\texttt{Bool} \times \texttt{Int}) \end{array} \tag{10.6}$$

Without expansion, the result type is $((\texttt{Int} \vee \texttt{Bool}) \times (\texttt{Bool} \vee \texttt{Int}))$ (α unifies `Int` and `Bool`). If we expand t, there exists a more precise result type $(\texttt{Int} \times \texttt{Bool}) \vee (\texttt{Bool} \times \texttt{Int})$, each summand of which corresponds to a different summand in s. Besides, due to the decomposition rule for product types (*ie*, Lemma 4.3.9), there also exist some other result types which involve type variables, like $((\texttt{Int} \vee \texttt{Bool}) \times \alpha) \vee ((\texttt{Int} \vee \texttt{Bool}) \times (\texttt{Int} \vee \texttt{Bool}) \setminus \alpha)$. Further expanding t makes more product decompositions possible, which may in turn generate new result types. However, the type $(\texttt{Int} \times \texttt{Bool}) \vee (\texttt{Bool} \times \texttt{Int})$ is informative enough, and so we set the heuristic number to 2, that is, the number of summands in s.

We may have to expand t also because of intersection. First, suppose s is an intersection of basic types; it can be viewed as a single basic type. Consider the example

$$t = \alpha \to (\alpha \times \alpha) \text{ and } s = \texttt{Int} \tag{10.7}$$

Without expansion, the result type is $\gamma_1 = (\texttt{Int} \times \texttt{Int})$. With two copies of t, besides γ_1, we get another result type $\gamma_2 = (\beta \times \beta) \vee (\texttt{Int} \setminus \beta \times \texttt{Int} \setminus \beta)$, which is more general than γ_1 (*eg*, $\gamma_1 = \gamma_2\{\mathbb{0}/\beta\}$). Generally, with k copies, we get k result types of the form

$$\gamma_k = (\beta_1 \times \beta_1) \vee \ldots \vee (\beta_{k-1} \times \beta_{k-1}) \vee (\texttt{Int} \setminus (\bigvee_{i=1..k-1} \beta_i) \times \texttt{Int} \setminus (\bigvee_{i=1..k-1} \beta_i))$$

It is clear that $\gamma_{k+1} \sqsubseteq_\emptyset \gamma_k$. Moreover, it is easy to find two substitutions $[\sigma_1, \sigma_2]$ such that $[\sigma_1, \sigma_2] \Vdash \gamma_k \sqsubseteq_\emptyset \gamma_{k+1}$ ($k \geq 2$). Therefore, γ_2 is the minimum (with respect to $\sqsubseteq_\emptyset$) of $\{\gamma_k, k \geq 1\}$, so expanding t more than once is useless (we do not get a type more precise than γ_2). However, we think the programmer expects $(\texttt{Int} \times \texttt{Int})$ as a result type instead of γ_2. So we take the heuristic number here as 1.

An intersection of product types is equivalent to $\bigvee_{i \in I}(s_1^i \times s_2^i)$, so we consider just a single product type (and then use union for the general case). For instance,

$$\begin{array}{l} t = ((\alpha \to \alpha) \times (\beta \to \beta)) \to ((\beta \to \beta) \times (\alpha \to \alpha)) \\ s = (((\texttt{Even} \to \texttt{Even}) \vee (\texttt{Odd} \to \texttt{Odd})) \times (\texttt{Bool} \to \texttt{Bool})) \end{array} \tag{10.8}$$

For the application to succeed, we have a constraint generated for each component of the product type, namely $(\alpha \to \alpha \geq (\texttt{Even} \to \texttt{Even}) \vee (\texttt{Odd} \to \texttt{Odd}))$ and $(\beta \to \beta \geq \texttt{Bool} \to \texttt{Bool})$. As with Example (10.5), it is better to expand $\alpha \to \alpha$ once for the first constraint, while there is no need to expand $\beta \to \beta$ for the second one. As a result, we expand the whole type t once, and get the result type $((\texttt{Bool} \to \texttt{Bool}) \times ((\texttt{Even} \to \texttt{Even}) \vee (\texttt{Odd} \to \texttt{Odd})))$ as expected. Generally, if the heuristic numbers of the components of a product type are respectively p_1 and p_2, we take $p_1 * p_2$ as the heuristic number for the whole product.

Finally, suppose s is an intersection of arrows, like for example `map even`.

$$\begin{array}{l} t = (\alpha \to \beta) \to [\alpha] \to [\beta] \\ s = (\texttt{Int} \to \texttt{Bool}) \wedge ((\gamma \setminus \texttt{Int}) \to (\gamma \setminus \texttt{Int})) \end{array} \tag{10.9}$$

When $p = 1$, the constraint to solve is $(\alpha \to \beta \geq s)$. As stated in Subsection 10.2.2, we get four possible result types: $[\ ] \to [\ ]$, $[\texttt{Int}] \to [\texttt{Bool}]$, $[\alpha \setminus \texttt{Int}] \to [\alpha \setminus \texttt{Int}]$, or $[\texttt{Int} \vee \alpha] \to [\texttt{Bool} \vee (\alpha \setminus \texttt{Int})]$, and we can build the minimum one by taking the

Table 10.1: Heuristic number $H_p(s)$ for the copies of t

Shape of s	**Number** $H_p(s)$
$\bigvee_{i\in I} s_i$	$\Sigma_{i\in I} H_p(s_i)$
$\bigwedge_{i\in P} b_i \wedge \bigwedge_{i\in N} \neg b_i \wedge \bigwedge_{i\in P_1} \alpha_i \wedge \bigwedge_{i\in N_1} \neg\alpha_i$	1
$\bigwedge_{i\in P}(s_i^1 \times s_i^2) \wedge \bigwedge_{i\in N} \neg(s_i^1 \times s_i^2)$	$\Sigma_{N'\subseteq N} H_p(s^1_{N'} \times s^2_{N'})$
$(s_1 \times s_2)$	$H_p(s_1) * H_p(s_2)$
$\bigwedge_{i\in P}(s_i^1 \to s_i^2) \wedge \bigwedge_{i\in N} \neg(s_i^1 \to s_i^2)$	1

where $(s^1_{N'} \times s^2_{N'}) = (\bigwedge_{i\in P} s_i^1 \wedge \bigwedge_{i\in N'} \neg s_i^1 \times \bigwedge_{i\in P} s_i^2 \wedge \bigwedge_{i\in N\setminus N'} \neg s_i^2)$.

intersection of them. If we continue expanding t, any result type we obtain is an intersection of some of the result types we have deduced for $p = 1$. Indeed, assume we expand t so that we get p copies of t. Then we would have to solve either $(\bigvee_{i=1..p} \alpha_i \to \beta_i \geq s)$ or $(\bigwedge_{i=1..p} \alpha_i \to \beta_i \geq s)$. For the first constraint to hold, by the decomposition rule of arrows, there exists i_0 such that $s \leq \alpha_{i_0} \to \beta_{i_0}$, which is the same constraint as for $p = 1$. The second constraint implies $s \leq \alpha_i \to \beta_i$ for all i; we recognize again the same constraint as for $p = 1$ (except that we intersect p copies of it). Consequently, expanding does not give us more information, and it is enough to take $p = 1$ as the heuristic number for this case.

Following the discussion above, we propose in Table 10.1 a heuristic number $H_p(s)$ that, according to the shape of s, sets an upper bound to the number of copies of t. We assume that s is in normal form. This definition can be easily extended to recursive types by memoization.

The next example shows that performing the expansion of t with $H_p(s)$ copies may not be enough to get a result type, confirming that this number is a heuristic that does not ensure completeness. Let

$$\begin{array}{l} t = ((\texttt{true} \times (\texttt{Int} \to \alpha)) \to t_1) \wedge ((\texttt{false} \times (\alpha \to \texttt{Bool})) \to t_2) \\ s = (\texttt{Bool} \times (\texttt{Int} \to \texttt{Bool})) \end{array} \tag{10.10}$$

Here $\mathsf{dom}(t)$ is $(\texttt{true} \times (\texttt{Int} \to \alpha)) \vee (\texttt{false} \times (\alpha \to \texttt{Bool}))$. The type s cannot be completely contained in either summand of $\mathsf{dom}(t)$, but it can be contained in $\mathsf{dom}(t)$. Indeed, the first summand requires the substitution of α to be a supertype of `Bool` while the second one requires it to be a subtype of `Int`. As `Bool` is not a subtype of `Int`, to make the application possible, we have to expand the function type at least once. However, according to Table 10.1, the heuristic number in this case is 1 (*ie*, no expansions).

Expansion of s

For simplicity, we assume that $\mathsf{dom}(\bigwedge_{i\in I} t\sigma_i) = \bigvee_{i\in I} \mathsf{dom}(t)\sigma_i$, so that the tallying problem for the application becomes $\bigwedge_{j\in J} s\sigma'_j \leq \bigvee_{i\in I} \mathsf{dom}(t)\sigma_i$. We now give some heuristic numbers for $|J|$ depending on $\mathsf{dom}(t)$.

First, consider the following example where $\mathsf{dom}(t)$ is a union:

$$\begin{array}{rl} \mathsf{dom}(t) = & (\mathtt{Int} \to ((\mathtt{Bool} \to \mathtt{Bool}) \wedge (\mathtt{Int} \to \mathtt{Int}))) \\ & \vee(\mathtt{Bool} \to ((\mathtt{Bool} \to \mathtt{Bool}) \wedge (\mathtt{Int} \to \mathtt{Int}) \wedge (\mathtt{Real} \to \mathtt{Real}))) \\ s = & (\mathtt{Int} \to (\alpha \to \alpha)) \vee (\mathtt{Bool} \to (\beta \to \beta)) \end{array} \tag{10.11}$$

For the application to succeed, we need to expand $\mathtt{Int} \to (\alpha \to \alpha)$ with two copies (so that we can make two distinct instantiations $\alpha = \mathtt{Bool}$ and $\alpha = \mathtt{Int}$) and $\mathtt{Bool} \to (\beta \to \beta)$ with three copies (for three instantiations $\beta = \mathtt{Bool}$, $\beta = \mathtt{Int}$, and $\beta = \mathtt{Real}$), corresponding to the first and the second summand in $\mathsf{dom}(t)$ respectively. Since the expansion distributes the union over the intersections, we need to get six copies of s. In detail, we need the following six substitutions: $\{\alpha = \mathtt{Bool}, \beta = \mathtt{Bool}\}$, $\{\alpha = \mathtt{Bool}, \beta = \mathtt{Int}\}$, $\{\alpha = \mathtt{Bool}, \beta = \mathtt{Real}\}$, $\{\alpha = \mathtt{Int}, \beta = \mathtt{Bool}\}$, $\{\alpha = \mathtt{Int}, \beta = \mathtt{Int}\}$, and $\{\alpha = \mathtt{Int}, \beta = \mathtt{Real}\}$, which are the Cartesian products of the substitutions for α and β.

If $\mathsf{dom}(t)$ is an intersection of basic types, we use 1 for the heuristic number. If it is an intersection of product types, we can rewrite it as a union of products and we only need to consider the case of just a single product type. For instance,

$$\begin{array}{rl} \mathsf{dom}(t) = & ((\mathtt{Int} \to \mathtt{Int}) \times (\mathtt{Bool} \to \mathtt{Bool})) \\ s = & ((\alpha \to \alpha) \times (\alpha \to \alpha)) \end{array} \tag{10.12}$$

It is easy to infer that the substitution required by the left component needs α to be $\mathtt{Int}$, while the one required by the right component needs α to be $\mathtt{Bool}$. Thus, we need to expand s at least once. Assume that $s = (s_1 \times s_2)$ and we need q_i copies of s_i with the type substitutions: $\sigma^i_1, \ldots, \sigma^i_{q_i}$. Generally, we can expand the whole product type so that we get $s_1 \times s_2$ copies as follows:

$$\begin{array}{rl} & \bigwedge_{j=1..q_1}(s_1 \times s_2)\sigma^1_j \wedge \bigwedge_{j=1..q_2}(s_1 \times s_2)\sigma^2_j \\ = & ((\bigwedge_{j=1..q_1} s_1\sigma^1_j \wedge \bigwedge_{j=1..q_2} s_1\sigma^2_j) \times (\bigwedge_{j=1..q_1} s_2\sigma^1_j \wedge \bigwedge_{j=1..q_2} s_2\sigma^2_j)) \end{array}$$

Clearly, this expansion type is a subtype of $(\bigwedge_{j=1..q_1} s_1\sigma^1_j \times \bigwedge_{j=1..q_2} s_2\sigma^2_j)$ and so the type tallying succeeds.

Next, consider the case where $\mathsf{dom}(t)$ is an intersection of arrows:

$$\begin{array}{rl} \mathsf{dom}(t) = & (\mathtt{Int} \to \mathtt{Int}) \wedge (\mathtt{Bool} \to \mathtt{Bool}) \\ s = & \alpha \to \alpha \end{array} \tag{10.13}$$

Without expansion, we need $(\alpha \to \alpha) \leq (\mathtt{Int} \to \mathtt{Int})$ and $(\alpha \to \alpha) \leq (\mathtt{Bool} \to \mathtt{Bool})$, which reduce to $\alpha = \mathtt{Int}$ and $\alpha = \mathtt{Bool}$; this is impossible. Thus, we have to expand s once, for the two conjunctions in $\mathsf{dom}(t)$.

Note that we may also have to expand s because of unions or intersections occurring under arrows. For example,

$$\begin{array}{rl} \mathsf{dom}(t) = & t' \to ((\mathtt{Int} \to \mathtt{Int}) \wedge (\mathtt{Bool} \to \mathtt{Bool})) \\ s = & t' \to (\alpha \to \alpha) \end{array} \tag{10.14}$$

As in Example (10.13), expanding once the type $\alpha \to \alpha$ (which is under an arrow in s) makes type tallying succeed. Because $(t' \to s_1) \wedge (t' \to s_2) \simeq t' \to (s_1 \wedge s_2)$, we can in

Table 10.2: Heuristic number $H_q(\mathsf{dom}(t))$ for the copies of s

Shape of $\mathsf{dom}(t)$	**Number** $H_q(\mathsf{dom}(t))$
$\bigvee_{i\in I} t_i$	$\prod_{i\in I} H_q(t_i) + 1$
$\bigwedge_{i\in P} b_i \wedge \bigwedge_{i\in N} \neg b_i \wedge \bigwedge_{i\in P_1} \alpha_i \wedge \bigwedge_{i\in N_1} \neg\alpha_i$	1
$\bigwedge_{i\in P}(t_i^1 \times t_i^2) \wedge \bigwedge_{i\in N} \neg(t_i^1 \times t_i^2)$	$\prod_{N'\subseteq N} H_q(t^1_{N'} \times t^2_{N'})$
$(t_1 \times t_2)$	$H_q(t_1) + H_q(t_2)$
$\bigwedge_{i\in P}(t_i^1 \to t_i^2) \wedge \bigwedge_{i\in N} \neg(t_i^1 \to t_i^2)$	$\lvert P\rvert * (H_q(t_i^1) + H_q(t_i^2))$

where $(t^1_{N'} \times t^2_{N'}) = (\bigwedge_{i\in P} t_i^1 \wedge \bigwedge_{i\in N'} \neg t_i^1 \times \bigwedge_{i\in P} t_i^2 \wedge \bigwedge_{i\in N\setminus N'} \neg t_i^2)$,

fact perform the expansion on s and then use subsumption to obtain the desired result. Likewise, we may have to expand s if $\mathsf{dom}(t)$ is an arrow type and contains an union in its domain. Therefore, we have to look into $\mathsf{dom}(t)$ and s deeply if they contain both arrow types.

Following these intuitions, we define in Table 10.2 a heuristic number $H_q(\mathsf{dom}(t))$ that, according to the sharp of $\mathsf{dom}(t)$, sets an upper bound to the number of copies of s.

Together

Up to now, we have considered the expansions of t and s separately. However, it might be the case that the expansions of t and s are interdependent, namely, the expansion of t causes the expansion of s and vice versa. Here we informally discuss the relationship between the two, and hint as why decidability is difficult to prove.

Let $\mathsf{dom}(t) = t_1 \vee t_2$, $s = s_1 \vee s_2$, and suppose the type tallying between $\mathsf{dom}(t)$ and s requires that $t_i\sigma_i \geq s_i$, where σ_1 and σ_2 are two conflicting type substitutions. Then we can simply expand $\mathsf{dom}(t)$ with σ_1 and σ_2, yielding $t_1\sigma_1 \vee t_2\sigma_1 \vee t_1\sigma_2 \vee t_2\sigma_2$. Clearly, this expansion type is a supertype of $t_1\sigma_1 \vee t_2\sigma_2$ and thus a supertype of s. Note that as t is on the bigger side of $\leq$, then the extra chunk of type brought by the expansion (*i.e.*, $t_2\sigma_1 \vee t_1\sigma_2$) does not matter. That is to say, the expansion of t would not cause the expansion of s.

However, the expansion of s could cause the expansion of t, and even a further expansion of s itself. Assume that $s = s_1 \vee s_2$ and s_i requires a different substitution σ_i (*i.e.*, $s_i\sigma_i \leq \mathsf{dom}(t)$ and σ_1 is in conflict with σ_2). If we expand s with σ_1 and σ_2, then we have

$$\begin{array}{rl} & (s_1 \vee s_2)\sigma_1 \wedge (s_1 \vee s_2)\sigma_2 \\ = & (s_1\sigma_1 \wedge s_1\sigma_2) \vee (s_1\sigma_1 \wedge s_2\sigma_2) \vee (s_2\sigma_1 \wedge s_1\sigma_2) \vee (s_2\sigma_1 \wedge s_2\sigma_2) \end{array}$$

It is clear that $s_1\sigma_1 \wedge s_1\sigma_2$, $s_1\sigma_1 \wedge s_2\sigma_2$ and $s_2\sigma_1 \wedge s_2\sigma_2$ are subtypes of $\mathsf{dom}(t)$. Consider the extra type $s_1\sigma_2 \wedge s_2\sigma_1$. If this extra type is empty (e.g., because s_1 and s_2 have different top-level constructors), or if it is a subtype of $\mathsf{dom}(t)$, then the type tallying succeeds. Otherwise, in some sense, we need to solve another type tallying between $s \wedge (s_2\sigma_1 \wedge s_1\sigma_2)$ and $\mathsf{dom}(t)$, which would cause the expansion of t or s. This is the main reason why we fail to prove the decidability of the application problem (that is, deciding $\bullet_\Delta$) so far.

To illustrate this phenomenon, consider the following example:

$$\begin{aligned} \mathsf{dom}(t) = \;& ((\texttt{Bool} \to \texttt{Bool}) \to (\texttt{Int} \to \texttt{Int})) \\ & \vee((\texttt{Bool} \to \texttt{Bool}) \vee (\texttt{Int} \to \texttt{Int})) \to ((\beta \to \beta) \vee (\texttt{Bool} \to \texttt{Bool})) \\ & \vee(\beta \times \beta) \\ s = \;& (\alpha \to (\texttt{Int} \to \texttt{Int})) \vee ((\texttt{Bool} \to \texttt{Bool}) \to \alpha) \vee (\texttt{Bool} \times \texttt{Bool}) \end{aligned} \tag{10.15}$$

Let us consider each summand in s respectively. A solution for the first summand is $\alpha \geq \texttt{Bool} \to \texttt{Bool}$, which corresponds to the first summand in $\mathsf{dom}(t)$. The second one requires $\alpha \leq \texttt{Int} \to \texttt{Int}$ and the third one $\beta \geq \texttt{Bool}$. Since $(\texttt{Bool} \to \texttt{Bool})$ is not subtype of $(\texttt{Int} \to \texttt{Int})$, we need to expand s once, that is,

$$\begin{aligned} s' = \;& s\{^{\texttt{Bool} \to \texttt{Bool}}/_{\alpha}\} \wedge s\{^{\texttt{Int} \to \texttt{Int}}/_{\alpha}\} \\ = \;& ((\texttt{Bool} \to \texttt{Bool}) \to (\texttt{Int} \to \texttt{Int})) \wedge ((\texttt{Int} \to \texttt{Int}) \to (\texttt{Int} \to \texttt{Int})) \\ & \vee((\texttt{Bool} \to \texttt{Bool}) \to (\texttt{Int} \to \texttt{Int})) \wedge ((\texttt{Bool} \to \texttt{Bool}) \to (\texttt{Int} \to \texttt{Int})) \\ & \vee((\texttt{Bool} \to \texttt{Bool}) \to (\texttt{Bool} \to \texttt{Bool})) \wedge ((\texttt{Int} \to \texttt{Int}) \to (\texttt{Int} \to \texttt{Int})) \\ & \vee((\texttt{Bool} \to \texttt{Bool}) \to (\texttt{Bool} \to \texttt{Bool})) \wedge ((\texttt{Bool} \to \texttt{Bool}) \to (\texttt{Int} \to \texttt{Int})) \\ & \vee(\texttt{Bool} \times \texttt{Bool}) \end{aligned}$$

Almost all the summands of s' are contained in $\mathsf{dom}(t)$ except the extra type

$$((\texttt{Bool} \to \texttt{Bool}) \to (\texttt{Bool} \to \texttt{Bool})) \wedge ((\texttt{Int} \to \texttt{Int}) \to (\texttt{Int} \to \texttt{Int}))$$

Therefore, we need to consider another type tallying involving this extra type and $\mathsf{dom}(t)$. By doing so, we obtain $\beta = \texttt{Int}$; however we have inferred before that β should be a supertype of `Bool`. Consequently, we need to expand $\mathsf{dom}(t)$; the expansion of $\mathsf{dom}(t)$ with $\{^{\texttt{Bool}}/_{\beta}\}$ and $\{^{\texttt{Int}}/_{\beta}\}$ makes the type tallying succeed.

In day-to-day examples, the extra type brought by the expansion of s is always a subtype of (the expansion type of) $\mathsf{dom}(t)$, and we do not have to expand $\mathsf{dom}(t)$ or s again. The heuristic numbers we gave seem to be enough in practice.

Chapter 11

Compilation into CoreℂDuce

In this chapter, we want to compile the polymorphic calculus into a variant of the monomorphic calculus (*e.g.*, CoreℂDuce). The aim of this translation is to provide an execution model for our polymorphic calculus that does not depend on dynamic propagation of type substitutions. We first introduce a "binding" type-case, which is needed by the translation (Section 11.1). Then we present the translation from our polymorphic calculus to CoreℂDuce and prove the soundness of the translation (Section 11.2). Lastly we discuss some limitations of this approach and hint at some possible improvements (Section 11.3).

11.1 A "binding" type-case

The translation we wish to define is *type-driven*, and therefore expects an annotated, well-typed expression of the polymorphic calculus. We therefore assume that we have access to the (already computed) type of every expression. The translation relies on an extension of the (monomorphic) type-case expression so as it features binding, which we write $(x{=}e){\in}t\,?\,e_1 : e_2$, and that can be encoded as:

$$(\lambda^{((s\wedge t)\to t_1)\wedge((s\wedge\neg t)\to t_2)} x.x{\in}t\,?\,e_1 : e_2)e$$

where s is the type of e, t_1 the type of e_1 and t_2 the type of e_2. An extremely useful and frequent case (also in practice) is when the expression e in $(x{=}e){\in}t\,?\,e_1 : e_2$ is syntactically equal to the binding variable x, that is $(x{=}x){\in}t\,?\,e_1 : e_2$. For this particular case it is worth introducing specific syntactic sugar (distinguished by a boldface "belongs to" symbol): $x\,\boldsymbol{\in}\,t\,?\,e_1 : e_2$. The reader may wonder what is the interest of binding a variable to itself. Actually, the two occurrences of x in $(x{=}x){\in}t$ denote two distinct variables: the one on the right is recorded in the environment with some type s; this variable does not occur either in e_1 or e_2 because it is hidden by the x on the left; this binds the occurrences of x in e_1 and e_2 but with different types, $s{\wedge}t$ in e_1 and $s{\wedge}\neg t$ in e_2. The advantage of such a notation is to allow the system to use different type assumptions for x in each branch, as stated by the typing rule directly derived from

the encoding:

$$\frac{\left\{\begin{array}{lll} t_1 = \Gamma(x) \wedge t & & t_2 = \Gamma(x) \wedge \neg t \\ t_i \not\simeq \mathbb{0} & \Rightarrow & \Delta \fatsemi \Gamma, (x : t_i) \vdash_C e_i : s_i \end{array}\right.}{\Delta \fatsemi \Gamma \vdash_C (x \in t \,?\, e_1 : e_2) : \bigvee_{t_i \not\simeq \mathbb{0}} s_i} (Ccase\text{-}var)$$

Note that x is defined in Γ but its type $\Gamma(x)$ is overridden in the premises to type the branches. With this construction, `map` can be defined as:

$$\mu^{(\alpha\to\beta)\to[\alpha]\to[\beta]} m\ \lambda f\ .\ \lambda^{[\alpha]\to[\beta]} \ell\ .\ \ell \in \texttt{nil}\,?\,\texttt{nil} : (f(\pi_1\ell), mf(\pi_2\ell))$$

If the unbinding version $\in$ were used instead, then ℓ in the second branch $(f(\pi_1\ell), mf(\pi_2\ell))$ would still have type $[\alpha]$. So $\pi_i\ell$ is not well-typed (remind that $[\alpha] \simeq (\alpha, [\alpha]) \vee \texttt{nil}$) and thus neither is `map`.

In practice any real programming language would implement either $\in$ (and not $\in$) or an even more generic construct (such as `match_with` or `case_of` pattern matching where each pattern may re-bind the x variable to a different type).

11.2 Translation to Core$\mathbb{C}$Duce

We first illustrate how to translate our polymorphic calculus to Core$\mathbb{C}$Duce and then prove that the translation is sound.

The translation we propose creates Core$\mathbb{C}$Duce expressions whose evaluation simulates the run-time type substitutions that may occur during the evaluation of an expression of the polymorphic calculus. As a starting point, let us recall what happens during the evaluation of an expression of the polymorphic calculus (Figure 7.2 of Section 7.3). First, an explicit substitution is propagated using the relabeling operation (Rule (*Rinst*) in Figure 7.2). This propagation stops at the lambda abstractions, which become annotated with the propagated set of substitutions. Second, when a lambda abstraction is applied to some argument the annotations of that lambda abstraction are propagated to the body (Rule (*Rappl*)). However the *type* of the argument determines which substitutions are propagated to the body. The translation from the polymorphic calculus to Core$\mathbb{C}$Duce reproduces these two steps: (1) it first pushes the explicit substitutions into the decorations of the underlying abstractions, and (2) it *encodes* the run-time choice of which substitutions to be propagated by a set of nested type-case expressions.

Consider the following expression

$$(\lambda^{(\alpha\to\alpha)\to\alpha\to\alpha} f.\lambda^{\alpha\to\alpha} x.fx)[\{\texttt{Int}/\alpha\}, \{\texttt{Bool}/\alpha\}] \tag{11.1}$$

Intuitively, the type substitutions take effect only at polymorphic abstractions. So we first push the explicit type substitutions into the decorations of the underlying abstractions. To do so, we perform the relabeling on the expression, namely the application of @, yielding

$$\lambda^{(\alpha\to\alpha)\to\alpha\to\alpha}_{[\{\texttt{Int}/\alpha\},\{\texttt{Bool}/\alpha\}]} f.\lambda^{\alpha\to\alpha} x.fx$$

Second, we show how we encode the dynamic relabeling that occurs at application time. In our example, the type for the whole abstraction is

$$((\texttt{Int} \to \texttt{Int}) \to \texttt{Int} \to \texttt{Int}) \wedge ((\texttt{Bool} \to \texttt{Bool}) \to \texttt{Bool} \to \texttt{Bool})$$

but we cannot simply propagate all the substitutions to the body expression since this leads to an ill-typed expression. The idea is therefore to use the "binding" type case, to simulate different relabeling on the body expression with different type cases. That is, we check which type substitutions are used by the type of the parameter and then propagate them to the body expression. So the encoding of (11.1) is as follows:

$$\begin{array}{l}\lambda^{((\texttt{Int}\to\texttt{Int})\to\texttt{Int}\to\texttt{Int})\wedge((\texttt{Bool}\to\texttt{Bool})\to\texttt{Bool}\to\texttt{Bool})} f. \\ \quad f \,\boldsymbol{\epsilon}\, (\alpha\to\alpha)\{\texttt{Int}/\alpha\} \wedge (\alpha\to\alpha)\{\texttt{Bool}/\alpha\} \,?\, \mathbb{C}[\![\lambda^{\alpha\to\alpha}_{[\{\texttt{Int}/\alpha\},\{\texttt{Bool}/\alpha\}]} x.fx]\!] \;: \\ \quad f \,\boldsymbol{\epsilon}\, (\alpha\to\alpha)\{\texttt{Int}/\alpha\} \,?\, \mathbb{C}[\![\lambda^{\alpha\to\alpha}_{[\{\texttt{Int}/\alpha\}]} x.fx]\!] \;: \\ \quad f \,\boldsymbol{\epsilon}\, (\alpha\to\alpha)\{\texttt{Bool}/\alpha\} \,?\, \mathbb{C}[\![\lambda^{\alpha\to\alpha}_{\{\texttt{Bool}/\alpha\}]} x.fx]\!] \;: \\ \qquad \mathbb{C}[\![\lambda^{\alpha\to\alpha} x.fx]\!]\end{array} \tag{11.2}$$

where $\mathbb{C}[\![e]\!]$ denotes the encoding of e. The first branch simulates the case where both type substitutions are selected and propagated to the body, that is, the parameter f belongs to the intersection of different instances of $\alpha \to \alpha$ with these two type substitutions. The second and third branches simulate the case where exactly one of the substitutions is used. Finally, the last branch denotes the case where no type substitutions are selected. Note that this last case can never happen, since by typing, we know that the application is well-typed and therefore that the argument is of type $\texttt{Bool} \to \texttt{Bool}$ or $\texttt{Int} \to \texttt{Int}$ (or of course, their intersection). This last case is only here to keep the expression syntactically correct (it is the "else" part of the last type-case) and can be replaced with a dummy expression.

Here, we see that the "binding" type-case is essential since it allows us to "refine" the type of the parameter of a lambda abstraction (replacing it by one of its instances). Note also that although the binding type case is encoded using a lambda abstraction, it is a lambda abstraction of CoreCDuce since it is not decorated.

The trickiest aspect of our encoding is that the order of the branches is essential: the more precise the type (in the sense of the subtyping relation) the sooner it must occur in the chain of type-cases. Indeed, consider the previous example but suppose that one tests for $(\texttt{Int} \to \texttt{Int})$ before $(\texttt{Int} \to \texttt{Int}) \wedge (\texttt{Bool} \to \texttt{Bool})$. For an argument of type $(\texttt{Int} \to \texttt{Int}) \wedge (\texttt{Bool} \to \texttt{Bool})$, the branch $\mathbb{C}[\![\lambda^{\alpha\to\alpha}_{[\{\texttt{Int}/\alpha\}]} x.fx]\!]$ would be evaluated instead of the more precise $\mathbb{C}[\![\lambda^{\alpha\to\alpha}_{[\{\texttt{Int}/\alpha\},\{\texttt{Bool}/\alpha\}]} x.fx]\!]$. Worst, if one now tests for $\texttt{Bool} \to \texttt{Bool}$ before $(\texttt{Int} \to \texttt{Int}) \wedge (\texttt{Bool} \to \texttt{Bool})$ and if the argument is again of type $\texttt{Int} \to \texttt{Int}$, then branch $\mathbb{C}[\![\lambda^{\alpha\to\alpha}_{[\{\texttt{Bool}/\alpha\}]} x.fx]\!]$ is taken (since the intersection between $\texttt{Int} \to \texttt{Int}$ and $\texttt{Bool} \to \texttt{Bool}$ is not empty). This clearly yields an unsound expression since now f is bound to a value of type $\texttt{Int} \to \texttt{Int}$ but used in a branch where the substitution $\{\texttt{Bool}/\alpha\}$ is applied.

Definition 11.2.1. *Let S be a set. We say that a sequence $L = S_0, \ldots, S_n$ is a* sequence of ordered subsets of S *if and only if*

- $\mathscr{P}(S) = \{S_0, \ldots, S_n\}$

- $\forall i, j \in \{0, \ldots, n\}, S_i \subset S_j \implies i > j$

Given a set S, there exists several sequences of ordered subsets of S. We consider the sequences equivalent modulo permutation of incomparable subsets and denote by *OrdSet*(S) the representative of this equivalence class. In layman's terms, *OrdSet*(S) is a sequence of all subsets of S such that any subset of S appears in the sequence before its proper subsets. We also introduce the notation:

$$\{x \,\boldsymbol{\epsilon}\, t_i \,?\, e_i\}_{i=1..n}$$

which is short for

$$\left(\begin{array}{l} x \,\boldsymbol{\epsilon}\, t_1 \,?\, e_1 \;\; : \\ \left(\begin{array}{l} \ldots \\ \left(\begin{array}{l} x \,\boldsymbol{\epsilon}\, t_{n-1} \,?\, e_{n-1} \;\; : \\ [x \,\boldsymbol{\epsilon}\, t_n \,?\,]\; e_n \end{array} \right) \end{array} \right) \end{array} \right)$$

We can now formally define our translation to Core$\mathbb{C}$Duce:

Definition 11.2.2. *Let* $\mathbb{C}[\![\_]\!]$ *be a function from* $\mathscr{E}$ *to* $\mathscr{E}_C$*, defined as*

$$\begin{array}{rcl} \mathbb{C}[\![c]\!] & = & c \\ \mathbb{C}[\![x]\!] & = & x \\ \mathbb{C}[\![(e_1, e_2)]\!] & = & (\mathbb{C}[\![e_1]\!], \mathbb{C}[\![e_2]\!]) \\ \mathbb{C}[\![\pi_i(e)]\!] & = & \pi_i(\mathbb{C}[\![e]\!]) \\ \mathbb{C}[\![e_1\ e_2]\!] & = & \mathbb{C}[\![e_1]\!]\ \mathbb{C}[\![e_2]\!] \\ \mathbb{C}[\![e \in t \,?\, e_1 : e_2]\!] & = & \mathbb{C}[\![e]\!] \in t \,?\, \mathbb{C}[\![e_1]\!] : \mathbb{C}[\![e_2]\!] \\ \mathbb{C}[\![e[\sigma_j]_{j\in J}]\!] & = & \mathbb{C}[\![e@[\sigma_j]_{j\in J}]\!] \\ \mathbb{C}[\![\lambda^{\wedge_{i\in I} t_i \to s_i}_{[\sigma_j]_{j\in J}} x.e]\!] & = & \lambda^{\wedge_{i\in I, j\in J} t_i\sigma_j \to s_i\sigma_j} x. \\ & & \{x \,\boldsymbol{\epsilon}\, t_P \,?\, \mathbb{C}[\![e@[\sigma_j]_{j\propto P}]\!]\}_{P\in OrdSet(I\times J)} \end{array}$$

where $t_P = \bigwedge_{(i,j)\in P} t_i\sigma_j$ *and* $j \propto P$ *means that* $j \in \{k \mid \exists i.\ (i,k) \in P\}$.

We must show that this translation is faithful, that is that given an expression and its translation, they reduce to the same value and that both have the same type. We proceed in several steps, using auxiliary lemmas. First we show that the translation preserves types. Then we prove that values and their translations are the same and have the same types. Lastly we show (at the end of the section) that the translation preserves the reduction of well-typed expressions.

We prove that the translation is *type-preserving* (Lemma 11.2.4). We first show an auxiliary lemma that states that the translation of relabeled expression preserves its type:

Lemma 11.2.3. *Let* e *be an expression and* $[\sigma_j]_{j\in J}$ *be a set of type-substitutions. If* $\Delta \,;\, \Gamma \vdash e@[\sigma_j]_{j\in J} : t$*, then* $\Gamma \vdash_C \mathbb{C}[\![e@[\sigma_j]_{j\in J}]\!] : t$.

Proof. The proof proceeds by induction and case analysis on the structure of the typing derivation e.

(*subsum*)**:** the typing derivation has the form:

$$\dfrac{\dfrac{\ldots}{\Delta \wr \Gamma \vdash e@[\sigma_j]_{j\in J} : s} \quad s \leq t}{\Delta \wr \Gamma \vdash e@[\sigma_j]_{j\in J} : t}\ (subsum)$$

By applying the induction hypothesis on the premise, we have $\Gamma \vdash_C \mathbb{C}[\![e@[\sigma_j]_{j\in J}]\!] : s$. Since $s \leq t$, by subsumption, we have $\Gamma \vdash_C \mathbb{C}[\![e@[\sigma_j]_{j\in J}]\!] : t$.

(*const*) $e \equiv c$**:** here, $\mathbb{C}[\![e@[\sigma_j]_{j\in J}]\!] = e@[\sigma_j]_{j\in J} = c$. Trivially, we have

$$\Gamma \vdash_C \mathbb{C}[\![e@[\sigma_j]_{j\in J}]\!] : b_c$$

(*var*) $e \equiv x$**:** similar to the previous case.

(*pair*) $e \equiv (e_1, e_2)$**:** $e@[\sigma_j]_{j\in J} = (e_1@[\sigma_j]_{j\in J}, e_2@[\sigma_j]_{j\in J})$ and $\mathbb{C}[\![e@[\sigma_j]_{j\in J}]\!] = (\mathbb{C}[\![e_1@[\sigma_j]_{j\in J}]\!], \mathbb{C}[\![e_2@[\sigma_j]_{j\in J}]\!])$. The typing derivation ends with:

$$\dfrac{\dfrac{\ldots}{\Delta \wr \Gamma \vdash e_1@[\sigma_j]_{j\in J} : s_1} \quad \dfrac{\ldots}{\Delta \wr \Gamma \vdash e_2@[\sigma_j]_{j\in J} : s_2}}{\Delta \wr \Gamma \vdash (e_1@[\sigma_j]_{j\in J}, e_2@[\sigma_j]_{j\in J}) : (s_1 \times s_2)}\ (pair)$$

Applying the induction hypothesis on each premise, we obtain $\Gamma \vdash_C \mathbb{C}[\![e_i@[\sigma_j]_{j\in J}]\!] : s_i$. We conclude by applying rule (*Cpair*), which gives us

$$\Gamma \vdash_C (\mathbb{C}[\![e_1@[\sigma_j]_{j\in J}]\!], \mathbb{C}[\![e_2@[\sigma_j]_{j\in J}]\!]) : (s_1 \times s_2)$$

that is $\Gamma \vdash_C \mathbb{C}[\![(e_1, e_2)@[\sigma_j]_{j\in J}]\!] : (s_1 \times s_2)$.

(*proj*) $e \equiv \pi_i(e')$**:** here, $e@[\sigma_j]_{j\in J} = \pi_i(e'@[\sigma_j]_{j\in J})$ and $\mathbb{C}[\![e@[\sigma_j]_{j\in J}]\!] = \pi_i(\mathbb{C}[\![e'@[\sigma_j]_{j\in J}]\!])$. The typing derivations ends with:

$$\dfrac{\dfrac{\ldots}{\Delta \wr \Gamma \vdash e'@[\sigma_j]_{j\in J} : t_1 \times t_2}}{\Delta \wr \Gamma \vdash \pi_i(e'@[\sigma_j]_{j\in J}) : t_i}\ (proj)$$

By induction, we have $\Gamma \vdash_C \mathbb{C}[\![e'@[\sigma_j]_{j\in J}]\!] : t_1 \times t_2$. We conclude this case by applying rule (*Cproj*), which gives us $\Gamma \vdash_C \pi_i(\mathbb{C}[\![e'@[\sigma_j]_{j\in J}]\!]) : t_i$, that is $\Gamma \vdash_C \mathbb{C}[\![\pi_i(e')@[\sigma_j]_{j\in J}]\!] : t_i$.

(*appl*) $e \equiv e_1 e_2$**:** here, $e@[\sigma_j]_{j\in J} = (e_1@[\sigma_j]_{j\in J})(e_2@[\sigma_j]_{j\in J})$ and $\mathbb{C}[\![e@[\sigma_j]_{j\in J}]\!] = \mathbb{C}[\![e_1@[\sigma_j]_{j\in J}]\!]\mathbb{C}[\![e_2@[\sigma_j]_{j\in J}]\!]$. The typing derivation ends with:

$$\dfrac{\dfrac{\ldots}{\Delta \wr \Gamma \vdash e_1@[\sigma_j]_{j\in J} : t \to s} \quad \dfrac{\ldots}{\Delta \wr \Gamma \vdash e_2@[\sigma_j]_{j\in J} : t}}{\Delta \wr \Gamma \vdash (e_1@[\sigma_j]_{j\in J})(e_2@[\sigma_j]_{j\in J}) : s}\ (appl)$$

We apply the induction hypothesis on the premises and obtain $\Gamma \vdash_C \mathbb{C}[\![e_1@[\sigma_j]_{j\in J}]\!] : t \to s$ and $\Gamma \vdash_C \mathbb{C}[\![e_2@[\sigma_j]_{j\in J}]\!] : t$. Then the rule (*Cappl*) gives us the result: $\Gamma \vdash_C \mathbb{C}[\![e_1@[\sigma_j]_{j\in J}]\!]\mathbb{C}[\![e_2@[\sigma_j]_{j\in J}]\!] : s$, that is $\Gamma \vdash_C \mathbb{C}[\![(e_1 e_2)@[\sigma_j]_{j\in J}]\!] : s$.

$(abstr)$ $e \equiv \lambda^{\wedge_{i\in I} t_i \to s_i}_{[\sigma_k]_{k\in K}} x.e'$**:** unsurprisingly, this case is the trickiest. We have $e@[\sigma_j]_{j\in J} = \lambda^{\wedge_{i\in I} t_i\to s_i}_{[\sigma_j]_{j\in J}\circ[\sigma_k]_{k\in K}} x.e'$, and the typing derivation ends with:

$$\dfrac{\begin{array}{l}\forall i \in I, j \in J, k \in K.\ \overset{\cdots}{\overline{\Delta' \wp \Gamma, (x : t_i(\sigma_j \circ \sigma_k)) \vdash e'@[\sigma_j \circ \sigma_k] : s_i(\sigma_j \circ \sigma_k)}} \\ \Delta' = \Delta \cup \mathsf{var}(\bigwedge_{i\in I, j\in J, k\in K} t_i(\sigma_j \circ \sigma_k) \to s_i(\sigma_j \circ \sigma_k))\end{array}}{\Delta \wp \Gamma \vdash \lambda^{\wedge_{i\in I} t_i\to s_i}_{[\sigma_j]_{j\in J}\circ[\sigma_k]_{k\in K}} x.e' : \bigwedge_{i\in I, j\in J, k\in K} t_i(\sigma_j \circ \sigma_k) \to s_i(\sigma_j \circ \sigma_k)}\ (abstr)$$

Let us consider Definition 11.2.2 in which we replace σ_j by $\sigma_j \circ \sigma_k$. We obtain:

$$\mathbb{C}[\![\lambda^{\wedge_{i\in I} t_i\to s_i}_{[\sigma_j]_{j\in J}\circ[\sigma_k]_{k\in K}} x.e']\!] = \lambda^{\wedge_{i\in I, j\in J, k\in K} t_i(\sigma_j\circ\sigma_k)\to s_i(\sigma_j\circ\sigma_k)} x. \\ \{x \in t_P\, ?\, \mathbb{C}[\![e'@[\sigma_j \circ \sigma_k]_{(j,k)\propto P}]\!]\}_{P\in OrdSet(I\times J\times K)}$$

where $t_P = \bigwedge_{(i,j,k)\in P} t_i(\sigma_j \circ \sigma_k)$ and $(j,k) \propto P$ means that $\exists i.\ (i,j,k) \in P$. Let us consider the premises of the $(abstr)$ rule. For each arrow type $t_{i_0}(\sigma_{j_0} \circ \sigma_{k_0}) \to s_{i_0}(\sigma_{j_0} \circ \sigma_{k_0})$, we need to prove

$$\Gamma, (x : t_{i_0}(\sigma_{j_0} \circ \sigma_{k_0})) \vdash_C \\ \{x \in t_P\, ?\, \mathbb{C}[\![e'@[\sigma_j \circ \sigma_k]_{(j,k)\propto P}]\!]\}_{P\in OrdSet(I\times J\times K)} : s_{i_0}(\sigma_{j_0} \circ \sigma_{k_0})$$

or, said differently, that each branch of the type case is either *not type-checked* or if it is, that it has type $s_{i_0}(\sigma_{j_0} \circ \sigma_{k_0})$. Let us consider any branch whose condition type is t_P.

First let us remark that if (i_0, j_0, k_0) (the triple that defines the type of x) is not in P, then the branch is *not type-checked*. Indeed, since the branches are ordered according to *OrdSet*$(I \times J \times K)$, if $(i_0, j_0, k_0) \notin P$, then there exists $P' = P \cup \{(i_0, j_0, k_0)\}$. Since $P \subset P'$ the branch whose condition type is $t_{P'}$ is placed before the one with type t_P in the type case. We can therefore deduce that for the case t_P, x has type $t_{i_0}(\sigma_{j_0} \circ \sigma_{k_0}) \wedge t_P \wedge \neg t_{P'} \wedge \neg \ldots \simeq \mathbb{0}$ (the negations coming from the types of the previous branches –including $t_{P'}$– that are removed, see rule (*Ccase-var*)). Since x has type $\mathbb{0}$ the branch is not type-checked.

Therefore we can assume that $(i_0, j_0, k_0) \in P$ and that the branch is taken. In this branch, the type of x is restricted to:

$$t'_P = \bigwedge_{(i,j,k)\in P} t_i(\sigma_j \circ \sigma_k) \wedge \bigwedge_{(i,j,k)\in(I\times J\times K)\setminus P} \neg t_i(\sigma_j \circ \sigma_k)$$

We now only have to prove that:

$$\Gamma, (x : t'_P) \vdash_C \mathbb{C}[\![e'@[\sigma_j \circ \sigma_k]_{(j,k)\propto P}]\!] : s_{i_0}(\sigma_{j_0} \circ \sigma_{k_0})$$

for all t'_P such that $(i_0, j_0, k_0) \in P$ and $t'_P \not\simeq \mathbb{0}$. Based on the premises of the rules $(abstr)$, by Lemma 7.4.14, we get

$$\Delta' \wp (\bigwedge_{(i,j,k)\in P} \Gamma), (x : t_P) \vdash e'@[\sigma_j \circ \sigma_k]_{(j,k)\propto P} : \bigwedge_{(i,j,k)\in P} s_i(\sigma_j \circ \sigma_k)$$

Then following Lemma 7.4.8, we have

$$\Delta' \wp \Gamma, (x : t'_P) \vdash e'@[\sigma_j \circ \sigma_k]_{(j,k)\propto P} : \bigwedge_{(i,j,k)\in P} s_i(\sigma_j \circ \sigma_k)$$

on to which we can apply the induction hypothesis to obtain:

$$\Gamma, (x : t'_P) \vdash_C \mathbb{C}[\![e'@[\sigma_j \circ \sigma_k]_{(j,k)\propto P}]\!] : \bigwedge_{(i,j,k)\in P} s_i(\sigma_j \circ \sigma_k)$$

Since $(i_0, j_0, k_0) \in P$, we have $\bigwedge_{(i,j,k)\in P} s_i(\sigma_j \circ \sigma_k) \leq s_{i_0}(\sigma_{j_0} \circ \sigma_{k_0})$. The result follows by subsumption.

(*case*) $e \equiv (e_0 \in t \,?\, e_1 : e_2)$**:** here, $e@[\sigma_j]_{j\in J} = (e_0@[\sigma_j]_{j\in J} \in t \,?\, e_1@[\sigma_j]_{j\in J} : e_2@[\sigma_j]_{j\in J})$ and $\mathbb{C}[\![e@[\sigma_j]_{j\in J}]\!] = \mathbb{C}[\![e_0@[\sigma_j]_{j\in J}]\!] \in t \,?\, \mathbb{C}[\![e_1@[\sigma_j]_{j\in J}]\!] : \mathbb{C}[\![e_2@[\sigma_j]_{j\in J}]\!]$. The typing derivation ends with:

$$\frac{\dots}{\Delta \fatsemi \Gamma \vdash e_0@[\sigma_j]_{j\in J} : t'} \quad \begin{cases} t' \not\leq \neg t & \Rightarrow & \dfrac{\dots}{\Delta \fatsemi \Gamma \vdash e_1@[\sigma_j]_{j\in J} : s} \\ t' \not\leq t & \Rightarrow & \dfrac{\dots}{\Delta \fatsemi \Gamma \vdash e_2@[\sigma_j]_{j\in J} : s} \end{cases}}{\Delta \fatsemi \Gamma \vdash (e_0@[\sigma_j]_{j\in J} \in t \,?\, e_1@[\sigma_j]_{j\in J} : e_2@[\sigma_j]_{j\in J}) : s} \; (\mathit{case})$$

By induction hypothesis, we have $\Gamma \vdash_C \mathbb{C}[\![e_0@[\sigma_j]_{j\in J}]\!] : t'$ and $\Gamma \vdash_C \mathbb{C}[\![e_i@[\sigma_j]_{j\in J}]\!] : s$. We can apply the typing rule (*Ccase*), which proves this case.

(*instinter*) $e \equiv e'[\sigma_i]_{i\in I}$**:** here, $e@[\sigma_j]_{j\in J} = e'@([\sigma_j \circ \sigma_i]_{(j,i)\in(J\times I)})$ and $\mathbb{C}[\![e@[\sigma_j]_{j\in J}]\!] = \mathbb{C}[\![e'@([\sigma_j \circ \sigma_i]_{(j,i)\in J\times I})]\!]$. By induction on e', we have $\Gamma \vdash_C \mathbb{C}[\![e'@[\sigma_j \circ \sigma_i]_{(j,i)\in(J\times I)}]\!] : t$, that is, $\Gamma \vdash_C \mathbb{C}[\![(e'[\sigma_i]_{i\in I})@[\sigma_j]_{j\in J}]\!] : t$.

□

Lemma 11.2.4. *Let e be an expression. If $\Delta \fatsemi \Gamma \vdash e : t$, then $\Gamma \vdash_C \mathbb{C}[\![e]\!] : t$.*

Proof. We proceed by case on the last typing rule used to derive the judgment $\Delta \fatsemi \Gamma \vdash e : t$ and build a corresponding derivation for the judgment $\Gamma \vdash_C \mathbb{C}[\![e]\!] : t$.

(*const*)**:** $\Delta \fatsemi \Gamma \vdash c : b_c$ and $\mathbb{C}[\![c]\!] = c$. It is clear that $\Gamma \vdash_C c : b_c$.

(*var*)**:** $\Delta \fatsemi \Gamma \vdash x : \Gamma(x)$ and $\mathbb{C}[\![x]\!] = x$. It is clear that $\Gamma \vdash_C x : \Gamma(x)$.

(*pair*)**:** consider the derivation:

$$\frac{\dfrac{\dots}{\Delta \fatsemi \Gamma \vdash e_1 : t_1} \quad \dfrac{\dots}{\Delta \fatsemi \Gamma \vdash e_2 : t_2}}{\Delta \fatsemi \Gamma \vdash (e_1, e_2) : t_1 \times t_2} \; (\mathit{pair})$$

Applying the induction hypothesis on each premise, we get $\Gamma \vdash_C \mathbb{C}[\![e_i]\!] : t_i$ (for $i = 1..2$). Then by applying (*Cpair*), we get $\Gamma \vdash_C (\mathbb{C}[\![e_1]\!], \mathbb{C}[\![e_2]\!]) : (t_1 \times t_2)$, that is, $\Gamma \vdash_C \mathbb{C}[\![(e_1, e_2)]\!] : (t_1 \times t_2)$.

(*proj*)**:** consider the derivation:

$$\frac{\dfrac{\dots}{\Delta \fatsemi \Gamma \vdash e : t_1 \times t_2}}{\Delta \fatsemi \Gamma \vdash \pi_i(e) : t_i} \; (\mathit{proj})$$

By induction, we have $\Gamma \vdash_C \mathbb{C}[\![e]\!] : t_1 \times t_2$. Then by (*Cproj*), we get $\Gamma \vdash_C \pi_i(\mathbb{C}[\![e]\!]) : t_i$, and consequently $\Gamma \vdash_C \mathbb{C}[\![\pi_i(e)]\!] : t_i$.

(*appl*)**:** consider the derivation:

$$\frac{\dfrac{\cdots}{\Delta \fatsemi \Gamma \vdash e_1 : t \to s} \quad \dfrac{\cdots}{\Delta \fatsemi \Gamma \vdash e_2 : t}}{\Delta \fatsemi \Gamma \vdash e_1 e_2 : s}\ (pair)$$

By induction, we have $\Gamma \vdash_C \mathbb{C}[\![e_1]\!] : t \to s$ and $\Gamma \vdash_C \mathbb{C}[\![e_2]\!] : t$. Then by (*Cappl*), we get $\Gamma \vdash_C \mathbb{C}[\![e_1]\!]\mathbb{C}[\![e_2]\!] : s$, that is, $\Gamma \vdash_C \mathbb{C}[\![e_1 e_2]\!] : s$.

(*abstr*)**:** consider the derivation:

$$\frac{\begin{array}{l}\forall i \in I, j \in J.\ \dfrac{\cdots}{\Delta' \fatsemi \Gamma, (x : t_i\sigma_j) \vdash e@[\sigma_j] : s_i\sigma_j} \\ \Delta' = \Delta \cup \mathsf{var}(\bigwedge_{i\in I, j\in J} t_i\sigma_j \to s_i\sigma_j)\end{array}}{\Delta \fatsemi \Gamma \vdash \lambda^{\wedge_{i\in I} t_i \to s_i}_{[\sigma_j]_{j\in J}} x.e : \bigwedge_{i\in I, j\in J} t_i\sigma_j \to s_i\sigma_j}\ (abstr)$$

According to Definition 11.2.2, we have

$$\mathbb{C}[\![\lambda^{\wedge_{i\in I} t_i \to s_i}_{[\sigma_j]_{j\in J}} x.e]\!] = \quad \lambda^{\wedge_{i\in I, j\in J} t_i\sigma_j \to s_i\sigma_j} x.\{x \,\boldsymbol{\epsilon}\, t_P \,?\, \mathbb{C}[\![e@[\sigma_j]_{j\propto P}]\!]\}_{P\in OrdSet(I\times J)}$$

where $t_P = \bigwedge_{(i,j)\in P} t_i\sigma_j$, $j \propto P$ means that $\exists i.\ (i,j) \in P$. Then for each arrow type $t_{i_0}\sigma_{j_0} \to s_{i_0}\sigma_{j_0}$, we need to prove that

$$\Gamma, x : t_{i_0}\sigma_{j_0} \vdash_C \{x \,\boldsymbol{\epsilon}\, t_P \,?\, \mathbb{C}[\![e@[\sigma_j]_{j\propto P}]\!]\}_{P\in OrdSet(I\times J)} : s_{i_0}\sigma_{j_0}$$

Similar to the proof of Lemma 11.2.3, for every branch P such that $(i_0, j_0) \in P$ (since otherwise, the branch is not type-checked) and $t'_P \not\simeq \mathbb{0}$, we need to prove that

$$\Gamma, x : t'_P \vdash_C \mathbb{C}[\![e@[\sigma_j]_{j\propto P}]\!] : s_{i_0}\sigma_{j_0}$$

where $t'_P = \bigwedge_{(i,j)\in P} t_i\sigma_j \wedge \bigwedge_{(i,j)\in (I\times J)\setminus P} \neg t_i\sigma_j$. Using Lemma 7.4.14, we get

$$\Delta' \fatsemi (\bigwedge_{(i,j)\in P} \Gamma), (x : t_P) \vdash e@[\sigma_j]_{j\propto P} : \bigwedge_{(i,j)\in P} s_i\sigma_j$$

Then following Lemma 7.4.8, we have

$$\Delta' \fatsemi \Gamma, (x : t'_P) \vdash e@[\sigma_j]_{j\propto P} : \bigwedge_{(i,j)\in P} s_i\sigma_j$$

By Lemma 11.2.3, we get

$$\Gamma, (x : t'_P) \vdash_C \mathbb{C}[\![e@[\sigma_j]_{j\propto P}]\!] : \bigwedge_{(i,j)\in P} s_i\sigma_j$$

Since $(i_0, j_0) \in P$, we have $\bigwedge_{(i,j)\in P} s_i\sigma_j \leq s_{i_0}\sigma_{j_0}$ and the result follows by subsumption.

(*case*)**:** consider the following derivation

$$\dfrac{\dfrac{\ldots}{\Delta \fatsemi \Gamma \vdash e : t'} \quad \begin{cases} t' \not\leq \neg t & \Rightarrow \quad \dfrac{\ldots}{\Delta \fatsemi \Gamma \vdash e_1 : s} \\ t' \not\leq t & \Rightarrow \quad \dfrac{\ldots}{\Delta \fatsemi \Gamma \vdash e_2 : s} \end{cases}}{\Delta \fatsemi \Gamma \vdash (e \in t \,?\, e_1 : e_2) : s}\ (case)$$

By induction, we have $\Gamma \vdash_C \mathbb{C}[\![e]\!] : t'$ and $\Gamma \vdash_C \mathbb{C}[\![e_i]\!] : s$. Then by (*Ccase*), we get $\Gamma \vdash_C (\mathbb{C}[\![e]\!] \in t \,?\, \mathbb{C}[\![e_1]\!] : \mathbb{C}[\![e_2]\!]) : s$, which gives $\Gamma \vdash_C \mathbb{C}[\![e \in t \,?\, e_1 : e_2]\!] : s$.

(*instinter*)**:** consider the following derivation:

$$\dfrac{\dfrac{\ldots}{\Delta \fatsemi \Gamma \vdash e : t} \quad \sigma_j \sharp \Delta}{\Delta \fatsemi \Gamma \vdash e[\sigma_j]_{j \in J} : \bigwedge_{j \in J} t\sigma_j}\ (instinter)$$

According to Corollary 7.4.13, we get $\Delta \fatsemi \Gamma \vdash e@[\sigma_j]_{j \in J} : \bigwedge_{j \in J} t\sigma_j$. Then by Lemma 11.2.3, we have $\Gamma \vdash_C \mathbb{C}[\![e@[\sigma_j]_{j \in J}]\!] : \bigwedge_{j \in J} t\sigma_j$, that is, $\Gamma \vdash_C \mathbb{C}[\![e[\sigma_j]_{j \in J}]\!] : \bigwedge_{j \in J} t\sigma_j$.

(*subsum*)**:** there exists a type s such that

$$\dfrac{\dfrac{\ldots}{\Delta \fatsemi \Gamma \vdash e : s} \quad s \leq t}{\Delta \fatsemi \Gamma \vdash e : t}\ (subsum)$$

By induction, we have $\Gamma \vdash_C \mathbb{C}[\![e]\!] : s$. Then by subsumption, we get $\Gamma \vdash_C \mathbb{C}[\![e]\!] : t$.

□

Although desirable, preserving the type of an expression is not enough. The translation also preserves values, as stated by the following lemma:

Lemma 11.2.5. *Let $v \in \mathscr{V}$ be a value. Then $\mathbb{C}[\![v]\!] \in \mathscr{V}_C$.*

Proof. By induction on v.

$\underline{c}$**:** it is clear that $\mathbb{C}[\![c]\!] \in \mathscr{V}_C$.
$\underline{\lambda^{\bigwedge_{i \in I} t_i \to s_i}_{[\sigma_k]_{k \in K}} x.e}$**:** clearly, $\mathbb{C}[\![\lambda^{\bigwedge_{i \in I} t_i \to s_i}_{[\sigma_k]_{k \in K}} x.e]\!] \in \mathscr{V}_C$.
$\underline{(v_1, v_2)}$**:** $\mathbb{C}[\![v]\!] = (\mathbb{C}[\![v_1]\!], \mathbb{C}[\![v_2]\!])$. By induction, we have $\mathbb{C}[\![v_i]\!] \in \mathscr{V}_C$. And so does $(\mathbb{C}[\![v_1]\!], \mathbb{C}[\![v_2]\!]) \in \mathscr{V}_C$.

□

Relating arbitrary expressions of the core calculus and their translation in CoreCDuce is tricky. Indeed, since the translation forces the propagation of type substitutions, then expressions of the polymorphic calculus that are *not* well-typed may have a well-typed translation. For instance, the expression $(\lambda^{\alpha \to \alpha} x.3)[\{\mathtt{Int}/\alpha\}]$ is not well-typed (since the inner λ-abstraction is not), but we can deduce that its translation $\mathbb{C}[\![(\lambda^{\alpha \to \alpha} x.3)[\{\mathtt{Int}/\alpha\}]]\!]$ has type $\mathtt{Int} \to \mathtt{Int}$. To circumvent this problem, we restrict ourselves to values for which we can show that a value and its translation have the same minimal type:

Lemma 11.2.6. *Let $v \in \mathscr{V}$ be a value. There exists a type t such that*

1. $\vdash v : t$ *and for all s if* $\vdash v : s$ *then* $t \leq s$;
2. $\vdash_C \mathbb{C}[\![v]\!] : t$ *and for all s if* $\vdash_C \mathbb{C}[\![v]\!] : s$ *then* $t \leq s$.

Proof. By induction on v.

$\underline{c}$**:** $\mathbb{C}[\![c]\!] = c$. It is clear that $t = b_c$.

$\underline{\lambda^{\wedge_{i\in I} t_i \to s_i}_{[\sigma_j]_{j\in J}} x.e}$**:** according to Definition 11.2.2, we have $\mathbb{C}[\![v]\!] = \lambda^{\wedge_{i\in I, j\in J} t_i\sigma_j \to s_i\sigma_j} x.e'$. Note that $\mathbb{C}[\![\_]\!]$ does not change the types in the interface. Therefore, $t = \bigwedge_{i\in I, j\in J} t_i\sigma_j \to s_i\sigma_j$.

$\underline{(v_1, v_2)}$**:** $\mathbb{C}[\![v]\!] = (\mathbb{C}[\![v_1]\!], \mathbb{C}[\![v_2]\!])$. By induction, there exists t_i such that $(a) \vdash v_i : t_i$ and for all s if $\vdash v_i : s$ then $t_i \leq s$, and $(b) \vdash_C \mathbb{C}[\![v_i]\!] : t_i$ and for all s if $\vdash_C \mathbb{C}[\![v_i]\!] : s$ then $t_i \leq s$. Then, let $t = (t_1, t_2)$ and the result follows.

□

We now want to show that the translation preserves the reduction, that is that if an expression e reduces to e' in the polymorphic calculus, then $\mathbb{C}[\![e]\!]$ reduces to $\mathbb{C}[\![e']\!]$ in Core$\mathbb{C}$Duce. Prior to that we show a technical (but straightforward) substitution lemma.

Lemma 11.2.7. *Let e be an expression, x an expression variable and v a value. Then* $\mathbb{C}[\![e\{v/x\}]\!] = \mathbb{C}[\![e]\!]\{\mathbb{C}[\![v]\!]/x\}$.

Proof. By induction on e.

$\underline{c}$**:**

$$\begin{aligned}\mathbb{C}[\![c\{v/x\}]\!] &= \mathbb{C}[\![c]\!] \\ &= c \\ &= c\{\mathbb{C}[\![v]\!]/x\} \\ &= \mathbb{C}[\![c]\!]\{\mathbb{C}[\![v]\!]/x\}\end{aligned}$$

$\underline{y}$**:**

$$\begin{aligned}\mathbb{C}[\![y\{v/x\}]\!] &= \mathbb{C}[\![y]\!] \\ &= y \\ &= y\{\mathbb{C}[\![v]\!]/x\} \\ &= \mathbb{C}[\![y]\!]\{\mathbb{C}[\![v]\!]/x\}\end{aligned}$$

$\underline{x}$**:**

$$\begin{aligned}\mathbb{C}[\![x\{v/x\}]\!] &= \mathbb{C}[\![v]\!] \\ &= x\{\mathbb{C}[\![v]\!]/x\} \\ &= \mathbb{C}[\![x]\!]\{\mathbb{C}[\![v]\!]/x\}\end{aligned}$$

$\underline{(e_1, e_2)}$**:**

$$\begin{aligned}\mathbb{C}[\![(e_1, e_2)\{v/x\}]\!] &= \mathbb{C}[\![(e_1\{v/x\}, e_2\{v/x\})]\!] \\ &= (\mathbb{C}[\![e_1\{v/x\}]\!], \mathbb{C}[\![e_2\{v/x\}]\!]) \\ &= (\mathbb{C}[\![e_1]\!]\{\mathbb{C}[\![v]\!]/x\}, \mathbb{C}[\![e_2]\!]\{\mathbb{C}[\![v]\!]/x\}) \quad \text{(by induction)} \\ &= (\mathbb{C}[\![e_1]\!], \mathbb{C}[\![e_2]\!])\{\mathbb{C}[\![v]\!]/x\} \\ &= \mathbb{C}[\![(e_1, e_2)]\!]\{\mathbb{C}[\![v]\!]/x\}\end{aligned}$$

$\underline{\pi_i(e')}$**:**

$$\begin{aligned}\mathbb{C}[\![\pi_i(e')\{v/x\}]\!] &= \mathbb{C}[\![\pi_i(e'\{v/x\})]\!]\\ &= \pi_i(\mathbb{C}[\![e'\{v/x\}]\!])\\ &= \pi_i(\mathbb{C}[\![e']\!]\{\mathbb{C}[\![v]\!]/x\}) \quad \text{(by induction)}\\ &= \pi_i(\mathbb{C}[\![e']\!])\{\mathbb{C}[\![v]\!]/x\}\\ &= \mathbb{C}[\![\pi_i(e')]\!]\{\mathbb{C}[\![v]\!]/x\}\end{aligned}$$

$\underline{e_1e_2}$**:**

$$\begin{aligned}\mathbb{C}[\![(e_1e_2)\{v/x\}]\!] &= \mathbb{C}[\![(e_1\{v/x\})(e_2\{v/x\})]\!]\\ &= \mathbb{C}[\![e_1\{v/x\}]\!]\mathbb{C}[\![e_2\{v/x\}]\!]\\ &= (\mathbb{C}[\![e_1]\!]\{\mathbb{C}[\![v]\!]/x\})(\mathbb{C}[\![e_2]\!]\{\mathbb{C}[\![v]\!]/x\}) \quad \text{(by induction)}\\ &= (\mathbb{C}[\![e_1]\!]\mathbb{C}[\![e_2]\!])\{\mathbb{C}[\![v]\!]/x\}\\ &= \mathbb{C}[\![e_1e_2]\!]\{\mathbb{C}[\![v]\!]/x\}\end{aligned}$$

$\underline{\lambda^{\wedge_{i\in I}t_i\to s_i}_{[\sigma_j]_{j\in J}}z.e_0}$**:** using α-conversion, we can assume that $\mathsf{tv}(v)\cap\bigcup_{j\in J}\mathsf{dom}(\sigma_j)=\emptyset$

$$\begin{aligned}&\mathbb{C}[\![(\lambda^{\wedge_{i\in I}t_i\to s_i}_{[\sigma_j]_{j\in J}}z.e_0)\{v/x\}]\!]\\ =\ &\mathbb{C}[\![\lambda^{\wedge_{i\in I}t_i\to s_i}_{[\sigma_j]_{j\in J}}z.e_0\{v/x\}]\!]\\ =\ &\lambda^{\wedge_{i\in I,j\in J}t_i\sigma_j\to s_i\sigma_j}z.\{z\in t_P\,?\,\mathbb{C}[\![(e_0\{v/x\})@[\sigma_j]_{j\propto P}]\!]\}_{P\in OrdSet(I\times J)}\\ =\ &\lambda^{\wedge_{i\in I,j\in J}t_i\sigma_j\to s_i\sigma_j}z.\{z\in t_P\,?\,\mathbb{C}[\![(e_0@[\sigma_j]_{j\propto P})\{v/x\}]\!]\}_{P\in OrdSet(I\times J)} \quad \text{(Lemma 7.4.5)}\\ =\ &\lambda^{\wedge_{i\in I,j\in J}t_i\sigma_j\to s_i\sigma_j}z.\{z\in t_P\,?\,\mathbb{C}[\![e_0@[\sigma_j]_{j\propto P}]\!]\{\mathbb{C}[\![v]\!]/x\}\}_{P\in OrdSet(I\times J)} \quad \text{(by induction)}\\ =\ &(\lambda^{\wedge_{i\in I,j\in J}t_i\sigma_j\to s_i\sigma_j}z.\{z\in t_P\,?\,\mathbb{C}[\![e_0@[\sigma_j]_{j\propto P}]\!]\}_{P\in OrdSet(I\times J)})\{\mathbb{C}[\![v]\!]/x\}\\ =\ &\mathbb{C}[\![\lambda^{\wedge_{i\in I}t_i\to s_i}_{[\sigma_j]_{j\in J}}z.e_0]\!]\{\mathbb{C}[\![v]\!]/x\}\end{aligned}$$

$\underline{e_0\in t\,?\,e_1:e_2}$**:**

$$\begin{aligned}&\mathbb{C}[\![(e_0\in t\,?\,e_1:e_2)\{v/x\}]\!]\\ =\ &\mathbb{C}[\![(e_0\{v/x\})\in t\,?\,(e_1\{v/x\}):(e_2\{v/x\})]\!]\\ =\ &\mathbb{C}[\![e_0\{v/x\}]\!]\in t\,?\,\mathbb{C}[\![e_1\{v/x\}]\!]:\mathbb{C}[\![e_2\{v/x\}]\!]\\ =\ &\mathbb{C}[\![e_0]\!]\{\mathbb{C}[\![v]\!]/x\}\in t\,?\,(\mathbb{C}[\![e_1]\!]\{\mathbb{C}[\![v]\!]/x\}):(\mathbb{C}[\![e_2]\!]\{\mathbb{C}[\![v]\!]/x\}) \quad \text{(by induction)}\\ =\ &(\mathbb{C}[\![e_0]\!]\in t\,?\,\mathbb{C}[\![e_1]\!]:\mathbb{C}[\![e_2]\!])\{\mathbb{C}[\![v]\!]/x\}\\ =\ &\mathbb{C}[\![e_0\in t\,?\,e_1:e_2]\!]\{\mathbb{C}[\![v]\!]/x\}\end{aligned}$$

$\underline{e'[\sigma_j]_{j\in J}}$**:** using α-conversion, we can assume that $\mathsf{tv}(v)\cap\bigcup_{j\in J}\mathsf{dom}(\sigma_j)=\emptyset$

$$\begin{aligned}\mathbb{C}[\![(e'[\sigma_j]_{j\in J})\{v/x\}]\!] &= \mathbb{C}[\![(e'\{v/x\})[\sigma_j]_{j\in J}]\!]\\ &= \mathbb{C}[\![(e'\{v/x\})@[\sigma_j]_{j\in J}]\!]\\ &= \mathbb{C}[\![(e'@[\sigma_j]_{j\in J})\{v/x\}]\!] \quad \text{(Lemma 7.4.5)}\\ &= \mathbb{C}[\![e'@[\sigma_j]_{j\in J}]\!]\{\mathbb{C}[\![v]\!]/x\} \quad \text{(by induction)}\\ &= \mathbb{C}[\![e'[\sigma_j]_{j\in J}]\!]\{\mathbb{C}[\![v]\!]/x\}\end{aligned}$$

□

We can now show that our translation preserves the reductions of the polymorphic calculus.

Lemma 11.2.8. *If $\Gamma\vdash e:t$ and $e\rightsquigarrow e'$, then $\mathbb{C}[\![e]\!]\rightsquigarrow^*_C\mathbb{C}[\![e']\!]$. More specifically,*

1. *if $e\rightsquigarrow_{(Rinst)}e'$, then $\mathbb{C}[\![e]\!]=\mathbb{C}[\![e']\!]$;*

2. if $e \rightsquigarrow_{(R)} e'$ *and* $(R) \neq (Rinst)$*, then* $\mathbb{C}[\![e]\!] \rightsquigarrow_C^+ \mathbb{C}[\![e']\!]$.

Proof. By induction and case analysis on e.

c, x or $\lambda^{\wedge_{i\in I} t_i \to s_i}_{[\sigma_j]_{j\in J}} x.e_0$**:** irreducible.

(e_1, e_2)**:** there are two ways to reduce e:
(1) $e_1 \rightsquigarrow e_1'$. By induction, $\mathbb{C}[\![e_1]\!] \rightsquigarrow_C^* \mathbb{C}[\![e_1']\!]$. Then we have $(\mathbb{C}[\![e_1]\!], \mathbb{C}[\![e_2]\!]) \rightsquigarrow_C^* (\mathbb{C}[\![e_1']\!], \mathbb{C}[\![e_2]\!])$, that is, $\mathbb{C}[\![(e_1, e_2)]\!] \rightsquigarrow_C^* \mathbb{C}[\![(e_1', e_2)]\!]$.
(2) $e_1 = v_1$ and $e_2 \rightsquigarrow e_2'$. By induction, $\mathbb{C}[\![e_2]\!] \rightsquigarrow_C^* \mathbb{C}[\![e_2']\!]$. Moreover, according to Lemma 11.2.5, $\mathbb{C}[\![v_1]\!] \in \mathscr{V}_C$. So we have $(\mathbb{C}[\![v_1]\!], \mathbb{C}[\![e_2]\!]) \rightsquigarrow_C^* (\mathbb{C}[\![v_1]\!], \mathbb{C}[\![e_2']\!])$, that is, $\mathbb{C}[\![(v_1, e_2)]\!] \rightsquigarrow_C^* \mathbb{C}[\![(v_1, e_2')]\!]$.

$\pi_i(e_0)$**:** there are two ways to reduce e:
(1) $e_0 \rightsquigarrow e_0'$. By induction, $\mathbb{C}[\![e_0]\!] \rightsquigarrow_C^* \mathbb{C}[\![e_0']\!]$. Then we have $\pi_i(\mathbb{C}[\![e_0]\!]) \rightsquigarrow_C^* \pi_i(\mathbb{C}[\![e_0']\!])$, that is, $\mathbb{C}[\![\pi_i(e_0)]\!] \rightsquigarrow_C^* \mathbb{C}[\![\pi_i(e_0')]\!]$.
(2) $e_0 = (v_1, v_2)$ and $e \rightsquigarrow v_i$. According to Lemma 11.2.5, $\mathbb{C}[\![(v_1, v_2)]\!] \in \mathscr{V}_C$. Moreover, $\mathbb{C}[\![(v_1, v_2)]\!] = (\mathbb{C}[\![v_1]\!], \mathbb{C}[\![v_2]\!])$. Therefore, $\pi_i(\mathbb{C}[\![v_1]\!], \mathbb{C}[\![v_2]\!]) \rightsquigarrow_C \mathbb{C}[\![v_i]\!]$, that is, $\mathbb{C}[\![\pi_i(v_1, v_2)]\!] \rightsquigarrow_C \mathbb{C}[\![v_i]\!]$.

$e_1 e_2$**:** there are three ways to reduce e:
(1) $e_1 \rightsquigarrow e_1'$. By induction, $\mathbb{C}[\![e_1]\!] \rightsquigarrow_C^* \mathbb{C}[\![e_1']\!]$. Then we have $\mathbb{C}[\![e_1]\!]\mathbb{C}[\![e_2]\!] \rightsquigarrow_C^* \mathbb{C}[\![e_1']\!]\mathbb{C}[\![e_2]\!]$, that is, $\mathbb{C}[\![e_1 e_2]\!] \rightsquigarrow_C^* \mathbb{C}[\![e_1' e_2]\!]$.
(2) $e_1 = v_1$ and $e_2 \rightsquigarrow e_2'$. By induction, $\mathbb{C}[\![e_2]\!] \rightsquigarrow_C^* \mathbb{C}[\![e_2']\!]$. Moreover, according to Lemma 11.2.5, $\mathbb{C}[\![v_1]\!] \in \mathscr{V}_C$. So we have $\mathbb{C}[\![v_1]\!]\mathbb{C}[\![e_2]\!] \rightsquigarrow_C^* \mathbb{C}[\![v_1]\!]\mathbb{C}[\![e_2']\!]$, that is, $\mathbb{C}[\![v_1 e_2]\!] \rightsquigarrow_C^* \mathbb{C}[\![v_1 e_2']\!]$.
(3) $e_1 = \lambda^{\wedge_{i\in I} t_i \to s_i}_{[\sigma_j]_{j\in J}} x.e_0$, $e_2 = v_2$ and $e_1 e_2 \rightsquigarrow (e_0@[\sigma_j]_{j\in P})\{v_2/x\}$, where $P = \{j \in J \mid \exists i \in I.\ \vdash v_2 : t_i\sigma_j\}$. According to Definition 11.2.2, we have

$$\mathbb{C}[\![e_1]\!] \quad = \lambda^{\wedge_{i\in I, j\in J} t_i\sigma_j \to s_i\sigma_j} x.\{x \in t_P \,?\, \mathbb{C}[\![e_0@[\sigma_j]_{j\propto P}]\!]\}_{P\in OrdSet(I\times J)}$$

(recall that $t_P = \bigwedge_{(i,j)\in P} t_i\sigma_j$ and $j \propto P$ means that $j \in \{k \mid \exists i.\ (i,k) \in P\}$). By Lemma 11.2.5, $\mathbb{C}[\![v_2]\!] \in \mathscr{V}_C$. Let $P_0 = \{(i,j) \mid \vdash v_2 : t_i\sigma_j\}$. Since $\vdash v_2 : t_i\sigma_j$, by Lemma 11.2.4, we have $\vdash_C \mathbb{C}[\![v_2]\!] : t_i\sigma_j$. Then we get $\vdash_C \mathbb{C}[\![v_2]\!] : t_{P_0}$. Therefore, $\mathbb{C}[\![e_1]\!]\mathbb{C}[\![v_2]\!] \rightsquigarrow_C^+ \mathbb{C}[\![e_0@[\sigma_j]_{j\propto P_0}]\!]\{\mathbb{C}[\![v_2]\!]/x\}$. Moreover, by lemma 11.2.7, $\mathbb{C}[\![e_0@[\sigma_j]_{j\propto P_0}]\!]\{\mathbb{C}[\![v_2]\!]/x\} = \mathbb{C}[\![e_0@[\sigma_j]_{j\propto P_0}\{v_2/x\}]\!]$, that is, $\mathbb{C}[\![(e_0)@[\sigma_j]_{j\in P}\{v_2/x\}]\!]$, which proves this case.

$e_0 \in t \,?\, e_1 : e_2$**:** there are three ways to reduce e:
(1) $e_0 \rightsquigarrow e_0'$. By induction, $\mathbb{C}[\![e_0]\!] \rightsquigarrow_C^* \mathbb{C}[\![e_0']\!]$. Then we have

$$\mathbb{C}[\![e_0]\!] \in t \,?\, \mathbb{C}[\![e_1]\!] : \mathbb{C}[\![e_2]\!] \rightsquigarrow_C^* \mathbb{C}[\![e_0']\!] \in t \,?\, \mathbb{C}[\![e_1]\!] : \mathbb{C}[\![e_2]\!]$$

that is, $\mathbb{C}[\![e_0 \in t \,?\, e_1 : e_2]\!] \rightsquigarrow_C^* \mathbb{C}[\![e_0' \in t \,?\, e_1 : e_2]\!]$.
(2) $e_0 = v_0$, $\vdash v_0 : t$ and $e \rightsquigarrow e_1$. According to Lemmas 11.2.5 and 11.2.4, $\mathbb{C}[\![v_0]\!] \in \mathscr{V}_C$ and $\vdash_C \mathbb{C}[\![v_0]\!] : t$. So we have $\mathbb{C}[\![v_0]\!] \in t \,?\, \mathbb{C}[\![e_1]\!] : \mathbb{C}[\![e_2]\!] \rightsquigarrow_C \mathbb{C}[\![e_1]\!]$.
(3) $e_0 = v_0$, $\not\vdash v_0 : t$ and $e \rightsquigarrow e_2$. By lemma 11.2.5, $\mathbb{C}[\![v_0]\!] \in \mathscr{V}_C$. According to Lemma 11.2.6, there exists a minimum type t_0 such that $\vdash v_0 : t_0$ and

$\vdash_C \mathbb{C}[\![v_0]\!] : t_0$. It is clear that $t_0 \not\leq t$ (otherwise $\vdash v_0 : t$). So we also have $\nvdash_C \mathbb{C}[\![v_0]\!] : t$. Therefore, $\mathbb{C}[\![v_0]\!] \in t \,?\, \mathbb{C}[\![e_1]\!] : \mathbb{C}[\![e_2]\!] \rightsquigarrow_C \mathbb{C}[\![e_2]\!]$.

$e_0[\sigma_j]_{j\in J}$: $e \rightsquigarrow e_0@[\sigma_j]_{j\in J}$. By Definition 11.2.2, $\mathbb{C}[\![e_0[\sigma_j]_{j\in J}]\!] = \mathbb{C}[\![e_0@[\sigma_j]_{j\in J}]\!]$. Therefore, the result follows.

□

We can finally state the soundness of our translation:

Theorem 11.2.9. *If $\vdash e : t$ and $\mathbb{C}[\![e]\!] \rightsquigarrow_C^* v_C$, then*

1. *there exists v such that $e \rightsquigarrow^* v$ and $\mathbb{C}[\![v]\!] = v_C$.*

2. *$\vdash_C v_C : t$.*

Proof. Since $\vdash e : t$, according to Theorem 7.4.17 and 7.4.16, there exists v such that $e \rightsquigarrow^* v$ and $\vdash v : t$. By Lemmas 11.2.8 and 11.2.5, we have $\mathbb{C}[\![e]\!] \rightsquigarrow_C^* \mathbb{C}[\![v]\!]$ and $\mathbb{C}[\![v]\!] \in \mathscr{V}_C$. From the reduction rules in Figure 2.2, the reduction is deterministic[1]. Therefore, $\mathbb{C}[\![v]\!] = v_C$. Finally, following Lemma 11.2.4, we get $\vdash_C v_C : t$. □

In addition, according to Lemma 11.2.8, the translations of an instantiation expression $e[\sigma_j]_{j\in J}$ and of its corresponding relabeling expression $e@[\sigma_j]_{j\in J}$ are the same: this is because the relabeling only propagates type substitutions without "changing" expressions. Therefore, restricting to the normalized calculus presented in Section 7.4.4, the translation still possesses all the properties presented above. Note that we can use $\mathbb{C}[\![e]\!] \rightsquigarrow_C^+ \mathbb{C}[\![e']\!]$ instead of $\mathbb{C}[\![e]\!] \rightsquigarrow_C^* \mathbb{C}[\![e']\!]$ in Lemma 11.2.8.

11.3 Current limitations and improvements

The translation we present allows one to encode the relabeling operation using only basic monomorphic constructs and the polymorphic subtyping relation defined in Chapter 4 . While of theoretical interest, this "blunt" approach has, in our opinion, two important drawbacks, that we discuss now before presenting some possible optimizations and workarounds. First, it is clear that the number of type-cases generated for a function $\lambda^{\wedge_{i\in I} s_i \to t_i}_{[\sigma_j]_{j\in J}} x.e$ is exponential in the product of the size of I (the set of overloaded functions that forms the interface of a lambda abstraction) and the size of J (the size of the set of substitutions applied to a lambda). This not only increases the code-size but also yields, in the worst case, an exponential time overhead (failing all the type-cases but the last one). Second, the translation breaks modularity, since a function is compiled differently according to the argument it is applied to.

Next we hint at some possible improvements that alleviate the issues highlighted above. The key idea is of course to reduce the number of type cases for the translation of abstractions.

[1] Without the restriction of evaluation contexts, we can prove that the reduction satisfies congruence.

Single Substitution. When only one substitution is present in the label of an abstraction, it can straightforwardly be propagated to the body expression without any encoding:

$$\mathbb{C}[\![\lambda^{\wedge_{i\in I} t_i \to s_i}_{[\sigma]} x.e]\!] = \lambda^{\wedge_{i\in I} t_i\sigma \to s_i\sigma} x.\mathbb{C}[\![e@[\sigma]]\!]$$

Global Elimination. As stated in Section 7.4.1, the type variables in the domain of any type substitution that do not occur in the expression can be safely eliminated (Lemma 7.4.10) and so can redundant substitutions (Lemma 7.4.11), since their elimination does not alter the type of the expression. This elimination simplifies the expression itself and consequently also simplifies its translation since fewer type cases are needed to encode the abstraction. For instance, consider the expression

$$\lambda^{\alpha\to\alpha}_{[\{\mathtt{Int}/\alpha,\mathtt{Int}/\beta\},\{\mathtt{Int}/\alpha,\mathtt{Bool}/\beta\}]} x.x \tag{11.3}$$

Since β is useless it can be eliminated, yielding $\lambda^{\alpha\to\alpha}_{[\{\mathtt{Int}/\alpha\},\{\mathtt{Int}/\alpha\}]} x.x$. Now since the two type substitutions are the same (the second one being redundant), we can safely remove one and obtain the simpler expression $\lambda^{\alpha\to\alpha}_{[\{\mathtt{Int}/\alpha\}]} x.x$. Finally since only one substitution remains, the expression can be further simplified to $\lambda^{(\mathtt{Int}\to\mathtt{Int})} x.x$.

Local Elimination. One can also work on the condition type in the generated type cases and eliminate those that are empty. Consider the translation of (11.3):

$$\begin{aligned}\mathbb{C}[\![\lambda^{\alpha\to\alpha}_{[\{\mathtt{Int}/\alpha,\mathtt{Int}/\beta\},\{\mathtt{Int}/\alpha,\mathtt{Bool}/\beta\}]} x.x]\!] = \quad & \lambda^{(\mathtt{Int}\to\mathtt{Int})\wedge(\mathtt{Int}\to\mathtt{Int})} x. \\ & \quad x \,\boldsymbol{\epsilon}\, \mathtt{Int} \wedge \mathtt{Int}\,?\,x \;\; : \\ & \quad x \,\boldsymbol{\epsilon}\, \mathtt{Int}\,?\,x \;\; : \\ & \quad x \,\boldsymbol{\epsilon}\, \mathtt{Int}\,?\,x \;\; : \\ & \quad x \,\boldsymbol{\epsilon}\, \mathbb{1}\,?\,x\end{aligned}$$

Clearly, the second and third branches can never be used since they are excluded by the first branch and the last branch is trivially useless. Thus the translated expression is equivalent to $\lambda^{(\mathtt{Int}\to\mathtt{Int})} x.x$. More generally, consider a branch $x \,\boldsymbol{\epsilon}\, t_P\,?\,e_P$ in the translation of $\mathbb{C}[\![\lambda^{\wedge_{i\in I} t_i\to s_i}_{[\sigma_j]_{j\in J}} x.e]\!]$. If the type $t'_P = t_P \wedge \bigwedge_{(i,j)\in(I\times J)\setminus P} \neg t_i\sigma_j$ is $\mathbb{0}$, then the branch will never be used. So we can eliminate it. Therefore, the translation of an abstraction can be simplified as

$$\lambda^{\wedge_{i\in I, j\in J} t_i\sigma_j \to s_i\sigma_j} x.\{x \,\boldsymbol{\epsilon}\, t_P\,?\,\mathbb{C}[\![e@[\sigma_j]_{j\propto P}]\!]\}_{P\in OrdSet(I\times J)\wedge t'_P\not\simeq\mathbb{0}}$$

No Abstractions. The reason for encoding an abstraction by a set of type cases is to simulate the relabeling. In each branch, a modified version of e (the body of the original function) is created where nested lambda abstractions are relabeled accordingly. Obviously, if e does not contain any lambda abstraction whose type variable need to be instantiated, then the type substitutions can be straightforwardly propagated to e:

$$\mathbb{C}[\![\lambda^{\wedge_{i\in I} t_i\to s_i}_{[\sigma_j]_{j\in J}} x.e]\!] = \lambda^{\wedge_{i\in I, j\in J} t_i\sigma_j \to s_i\sigma_j} x.\mathbb{C}[\![e@[\sigma_j]_{j\in J}]\!]$$

A Union Rule. Consider the following union typing rule

$$\frac{\Delta \fatsemi \Gamma, (x : s_1) \vdash e : t \qquad \Delta \fatsemi \Gamma, (x : s_2) \vdash e : t}{\Delta \fatsemi \Gamma, (x : s_1 \vee s_2) \vdash e : t}(union)$$

(which is sound but not admissible in our system[2]). This rule allows us to simplify the translation of abstractions by combining the branches with the same relabeling

$$\mathbb{C}[\![\lambda^{\wedge_{i\in I} t_i \to s_i}_{[\sigma_j]_{j\in J}} x.e]\!] = \lambda^{\wedge_{i\in I, j\in J} t_i\sigma_j \to s_i\sigma_j} x.\{x \boldsymbol{\epsilon} t_P \ ?\ \mathbb{C}[\![e@[\sigma_j]_{j\in P}]\!]\}_{P\in OrdSet(J)}$$

where $t_P = \bigwedge_{j\in P}(\bigvee_{i\in I} t_i)\sigma_j$. Notice now the t_i's are combined inside each t_P. The size of the encoding now is only exponential in $|J|$.

[2]We can simulate this union rule with an explicit annotation that drives the case disjunction, see Section 7.4 in [FCB08] for more details.

Chapter 12

Extensions, Design Choices, and Related Work

In this chapter, we first present some extensions and design choices, such as recursive functions, type substitution application and/or negation arrows for Rule (*abstr*), open type cases, and so on. Finally, we discuss related work.

12.1 Recursive function

It is straightforward to add recursive functions with minor modifications in our calculus. First, we use the μ notation to denote recursive functions, that is $\mu^{\wedge_{i\in I} t_i \to s_i}_{[\sigma_j]_{j\in J}} f\ \lambda x.\ e$ (without decoration in the implicitly typed calculus), where f denotes the recursion variable which can, thus, occur in e. Second, we add in the type environment the recursion variable f associated with the type obtained by applying the decoration to the interface, that is $\wedge_{i\in I, j\in J} t_i\sigma_j \to s_i\sigma_j$. Formally, the typing rule for recursive functions is amended as

$$\frac{\begin{array}{l} t = \bigwedge_{i\in I, j\in J} t_i\sigma_j \to s_i\sigma_j \\ \forall i \in I, j \in J.\ \Delta \cup \mathsf{var}(t) \fatsemi \Gamma, (f : t), (x : t_i\sigma_j) \vdash e@[\sigma_j] : s_i\sigma_j \end{array}}{\Delta \fatsemi \Gamma \vdash \mu^{\wedge_{i\in I} t_i \to s_i}_{[\sigma_j]_{j\in J}} f\ \lambda x.\ e : t} \ (\textit{abstr-rec})$$

This suffices for our system: the reader can refer to Section 7.5 in [FCB08] for a discussion on how and why recursion is restricted to functions. Finally, we also need to expand the recursive variable f during (application) reduction, that is to replace f by the recursive function itself, which is formalized as

$$(\textit{Rappl-rec}) \quad (\mu^{\wedge_{i\in I} t_i \to s_i}_{[\sigma_j]_{j\in J}} f\ \lambda x.\ e')v \rightsquigarrow (e'@[\sigma_j]_{j\in P})\{\mu^{\wedge_{i\in I} t_i \to s_i}_{[\sigma_j]_{j\in J}} f\ \lambda x.\ e'/f, v/x\} \quad \text{where } P = \{j\in J \mid \exists i\in I.\ \vdash v : t_i\sigma_j\}$$

12.2 Relabeling and type-substitution application

Given a type substitution σ and an expression e, the application of σ to e, denoted as $e\sigma$, is defined as

$$\begin{array}{rl} c\sigma = & c \\ (e_1\ e_2)\sigma = & (e_1\sigma)\ (e_2\sigma) \\ (\lambda^{\wedge_{i\in I} t_i\to s_i}_{[\sigma_j]_{j\in J}} x.e)\sigma = & \lambda^{\wedge_{i\in I} t_i\to s_i}_{[\sigma'_j]_{j\in J}} x.e\sigma' \\ (e[\sigma_j]_{j\in J})\sigma = & (e\sigma')[\sigma'_j]_{j\in J} \end{array}$$

where $\sigma' = \sigma|_{\mathsf{dom}(\sigma)\setminus\bigcup_{j\in J}\mathsf{dom}(\sigma_j)}$, $\sigma'_j = (\sigma\circ\sigma_j)|_{\bigcup_{j\in J}\mathsf{dom}(\sigma_j)}$ and $\sigma|_X$ denote the restriction of σ to X, that is $\{\sigma(\alpha)/\alpha \mid \alpha \in \mathsf{dom}(\sigma)\cap X\}$.

A concession to the sake of concision is the use of the relabeling operation '@' in the premises of both "abstr" rules in Figures 7.1 and 8.1. A slightly different but better formulation is to use as premises $\ldots \vdash_* e\sigma_j : s_i\sigma_j$ instead of $\ldots \vdash_* e@[\sigma_j] : s_i\sigma_j$. Both formulations are sound and the differences are really minimalist, but this second formulation rules out few anomalies of the current system. For instance, with the current formulation $\lambda^{\mathtt{Int}\to\mathtt{Int}}_D y.((\lambda^{\alpha\to\alpha}_{[]} x.\mathtt{42})[\{\mathtt{Int}/\alpha\}]y)$ is well-typed if and only if the decoration D is a non empty set of types-substitutions (whatever they are). Indeed a non empty D triggers the use of '@' in the premises of the "abstr" rule, and this "pushes" the type substitution $[\{\mathtt{Int}/\alpha\}]$ into the decoration of the body, thus making the body well typed (taken in isolation,$\lambda^{\alpha\to\alpha}_{[\{\mathtt{Int}/\alpha\}]} x.\mathtt{42}$ is well typed while $(\lambda^{\alpha\to\alpha}_{[]} x.\mathtt{42})[\{\mathtt{Int}/\alpha\}]$ is not). Although the second formulation rules out such kind of anomalies, we preferred to present the first one since it does not need the introduction of such technically-motivated new definitions.

There exists another solution to rule out such kind of anomalies. It is easy to see that these two formulations differ only in the instantiations with respect to the type-checking. Similar to the relabeling of abstractions, we can also perform a "lazy" relabeling on the instantiations:

$$(e[\sigma_j]_{j\in J})@[\sigma_k]_{k\in K} = (e[\sigma_j]_{j\in J})_{[\sigma_k]_{k\in K}}$$

and the propagation of substitutions occurs, whenever is needed, that is, during either reduction:

$$(e[\sigma_j]_{j\in J})_{[\sigma_k]_{k\in K}} \rightsquigarrow e@([\sigma_k]_{k\in K}\circ[\sigma_j]_{j\in J})$$

or type-checking:

$$\frac{\forall k\in K.\ \left\{\begin{array}{l}\Delta\fatsemi\Gamma\vdash e@[\sigma'_k] : t_k \\ \forall j\in J.\ \sigma^k_j \not\sharp \Delta\end{array}\right.}{\Delta\fatsemi\Gamma\vdash (e[\sigma_j]_{j\in J})_{[\sigma_k]_{k\in K}} : \bigwedge_{k\in K}(\bigwedge_{j\in J} t_k\sigma^k_j)}\ (\mathit{instinter'})$$

where $\sigma'_k = \sigma_k|_{\mathsf{dom}(\sigma_k)\setminus\bigcup_{j\in J}\mathsf{dom}(\sigma_j)}$ and $\sigma^k_j = (\sigma_k\circ\sigma_j)|_{\bigcup_{j\in J}\mathsf{dom}(\sigma_j)}$. Note that we keep the restriction of the composition of the substitution σ_k to be propagated and the substitutions $[\sigma_j]_{j\in J}$ in the instantiation to the domain of $[\sigma_j]_{j\in J}$, but push the other substitutions into the instantiation term e within the rule (*instinter'*). Thus, the expression $\lambda^{\mathtt{Int}\to\mathtt{Int}}_D y.((\lambda^{\alpha\to\alpha}_{[]} x.\mathtt{42})[\{\mathtt{Int}/\alpha\}]y)$ is not well-typed since we will check $\lambda^{\alpha\to\alpha}_{[]} x.\mathtt{42}$. However, this solution would make our calculus more involved.

12.3 Negation arrows and models

In this work we dodged the problem of the negation of arrow types. Notice indeed that a value can have as type the negation of an arrow type just by subsumption. This implies that no λ-abstraction can have a negated arrow type. So while the type $\neg(\texttt{Bool}\to\texttt{Bool})$ can be inferred for, say, $(3, 42)$, it is not possible to infer it for $\lambda^{\texttt{Int}\to\texttt{Int}}x.(x+3)$. This problem was dealt in $\mathbb{C}$Duce by deducing for a λ-abstraction the type in its interface intersected with any negations of arrow types that did not make the type empty [FCB08]. Technically, this was dealt with "type schemas": a function such as $\lambda^{\texttt{Int}\to\texttt{Int}}x.x + 3$ has type schema $\{\{\texttt{Int}\to\texttt{Int}\}\}$, that is, it has every non empty type of the form $(\texttt{Int}\to\texttt{Int}) \wedge \bigwedge_{j\in J}\neg(t'_j\to s'_j)$ (and thus, in particular, $(\texttt{Int}\to\texttt{Int}) \wedge \neg(\texttt{Bool}\to\texttt{Bool})$). Formally, the typing rule for abstractions in $\mathbb{C}$Duce is

$$\frac{t = \bigwedge_{i\in I} t_i \to s_i \wedge \bigwedge_{j\in J} \neg(t'_j \to s'_j) \quad t \not\simeq \mathbb{0} \qquad \forall i \in I.\ \Gamma, (f : t), (x : t_i) \vdash_C e : s_i}{\Gamma \vdash_C \mu^{\wedge_{i\in I} t_i\to s_i} f\ \lambda x.\ e : t}\ \textit{(Cabstr-neg)}$$

In our context, however, the presence of type variables makes a definition such as that of type schemas more difficult. If we take the same definition of type schemas as $\mathbb{C}$Duce, we would assign the type $\mathbb{0}$ for abstractions, which is clearly unsound. For example, as $(\alpha \to \alpha) \wedge \neg(\texttt{Bool} \to \texttt{Bool})$ is not empty, we deduce that $\vdash \lambda^{\alpha\to\alpha}x.x : (\alpha \to \alpha) \wedge \neg(\texttt{Bool} \to \texttt{Bool})$. Then we can further apply the instantiation rule with the set of type substitutions $[\{\texttt{Bool}/\alpha\}]$, yielding the unsound judgment $\vdash \lambda^{\texttt{Bool}\to\texttt{Bool}}x.x : \mathbb{0}$. Therefore, a type schema should probably denote only types that are not empty under every possible set of type substitutions. Thus, considering negation arrows, the typing rule for abstractions in our system is

$$\frac{\begin{array}{l}\Delta' = \Delta \cup \mathsf{var}(\bigwedge_{i\in I, j\in J} t_i\sigma_j \to s_i\sigma_j) \\ t = \bigwedge_{i\in I, j\in J} t_i\sigma_j \to s_i\sigma_j \wedge \bigwedge_{k\in K} \neg(t'_k \to s'_k) \quad t \not\sqsubseteq_\Delta \mathbb{0} \\ \forall i \in I, j \in J.\ \Delta' \fatsemi \Gamma, (x : t_i\sigma_j) \vdash e@[\sigma_j] : s_i\sigma_j\end{array}}{\Delta \fatsemi \Gamma \vdash \lambda^{\wedge_{i\in I} t_i\to s_i}_{[\sigma_j]_{j\in J}} x.e : t}\ (\textit{abstr-neg})$$

As witnessed by $\mathbb{C}$Duce, dealing with this aspect is mainly of theoretical interest: it allows us to interpret types as sets of values and the interpretation forms a model [FCB08]. In our case, as the presence of type variables, the problem becomes more complicated. For example, there is no way to deduce either $\vdash v : \alpha$ holds or $\vdash v : \neg\alpha$ for any value v and any type variable α. Moreover, does the function $\lambda^{\texttt{Int}\to\texttt{Int}}x.x + 3$ has type $\alpha \to \alpha$? If it does, then it can be used as a polymorphic function, which is clearly unsound. If it does not, then we deduce that it has $\neg(\alpha \to \alpha)$ and thus we get $\vdash \lambda^{\texttt{Int}\to\texttt{Int}}x.x + 3 : \texttt{Int} \to \texttt{Int} \wedge \neg(\alpha \to \alpha)$. We can then apply the instantiation rule with $\{\texttt{Int}/\alpha\}$, yielding the unsound judgment $\vdash \lambda^{\texttt{Int}\to\texttt{Int}}x.x+3 : \mathbb{0}$. The problem is that without any meaning (or assignment) of type variables, the meaning of a non-ground type is not clear.

Therefore, we first focus on the ground types $\mathcal{T}_0$ and study a possible interpretation for the ground types[1]:

$$[\![t]\!]_v = \{v \in \mathscr{V} \mid \vdash v : s, s \sqsubseteq_\emptyset t\}.$$

Under this interpretation, the interpretations of ground arrow types contain not only abstractions with ground arrow types but also abstractions with non-ground arrow types. For example, the interpretation of $\texttt{Int} \to \texttt{Int}$ contains not only the value $\lambda^{\texttt{Int}\to\texttt{Int}} x.e$ but also the value $\lambda^{\alpha\to\alpha} x.e$ (since $\lambda^{\alpha\to\alpha} x.e$ can be instantiated as $\lambda^{\texttt{Int}\to\texttt{Int}} x.e@\{\texttt{Int}/\alpha\}$).

Next, we can prove that the interpretation is a model for ground types (and thus $\mathscr{V}$ is a solution to $\mathcal{D} = \mathscr{C} + \mathcal{D}^2 + \mathscr{P}_f(\mathcal{D}^2)$, where $\mathscr{C} = \bigcup_{b\in\mathscr{B}} [\![b]\!]_v \cup [\![\mathbb{1} \to \mathbb{0}]\!]_v$).

Lemma 12.3.1. *Let b be a basic type, and $t_1, t_2, t, s \in \mathcal{T}_0$ ground types. Then*

(1) $[\![b]\!]_v = \{c \mid b_c \leq b\}$;

(2) $[\![t_1 \times t_2]\!]_v = \{(v_1, v_2) \mid \vdash v_1 : s_i,\ s_i \sqsubseteq_\emptyset t_i\}$;

(3) $[\![t \to s]\!]_v = \{\lambda^{\bigwedge_{i\in I} t_i\to s_i}_{[\sigma_j]_{j\in J}} x.e \in \mathscr{V} \mid \bigwedge_{i\in I, j\in J} t_i\sigma_j \to s_i\sigma_j \sqsubseteq_\emptyset t \to s\}$;

(4) $[\![\mathbb{0}]\!]_v = \emptyset$;

(5) $[\![t_1 \wedge t_2]\!]_v = [\![t_1]\!]_v \cap [\![t_2]\!]_v$;

(6) $[\![\neg t]\!]_v = \mathscr{V} \setminus [\![t]\!]_v$;

(7) $[\![t_1 \vee t_2]\!]_v = [\![t_1]\!]_v \cup [\![t_2]\!]_v$;

(8) $[\![t]\!]_v = \emptyset \iff t \simeq \mathbb{0}$.

Proof. (1), (2), (3): similar to the proof of Lemma 7.4.3.
(4): we prove the following statement:

$$\forall v \in \mathscr{V}. \forall s \in \mathcal{T}.\ \vdash v : s \Rightarrow s \not\sqsubseteq_\emptyset \mathbb{0}$$

by induction on the typing derivation:

(*const*): straightforward.
(*pair*): $v = (v_1, v_2)$, $s = (s_1 \times s_2)$ and $\vdash v_i : s_i$. By induction on $\vdash v_i : s_i$, we have $s_i \not\sqsubseteq_\emptyset \mathbb{0}$ and thus $(s_1 \times s_2) \not\sqsubseteq_\emptyset \mathbb{0}$.
(*abstr-neg*): $v = \lambda^{\bigwedge_{i\in I} t_i\to s_i}_{[\sigma_j]_{j\in J}} x.e$ and $s = \bigwedge_{i\in I}(t_i \to s_i) \wedge \bigwedge_{j\in J} \neg(t'_j \to s'_j)$. One of the premises of the rule is $s \not\sqsubseteq_\emptyset \mathbb{0}$.
(*subsum*): $\vdash v : s'$ and $s' \leq s$. By induction on $\vdash v : s'$, we have $s' \not\sqsubseteq_\emptyset \mathbb{0}$ and thus $s \not\sqsubseteq_\emptyset \mathbb{0}$.

Therefore, $\forall v \in \mathscr{V}.\ v \notin [\![\mathbb{0}]\!]_v$, that is, $[\![\mathbb{0}]\!]_v = \emptyset$.

[1] Another solution is that we could parameterize the typing judgment with a semantics assignment η, which gives the interpretation for type variables (and thus non-ground types). We leave it for future work.

(5): we first prove the statement

$$\text{if } t \leq s \text{ then } [\![t]\!]_v \subseteq [\![s]\!]_v \quad (*).$$

Consider any $v \in [\![t]\!]_v$. There exists t' such that $\vdash v : t'$ and $t' \sqsubseteq_\emptyset t$. As $t \leq s$, we have $t' \sqsubseteq_\emptyset s$. So $v \in [\![s]\!]_v$. Therefore, $[\![t]\!]_v \subseteq [\![s]\!]_v$.
Applying (*) on $t_1 \wedge t_2 \leq t_i$, we get $[\![t_1 \wedge t_2]\!]_v \subseteq [\![t_i]\!]_v$, and thus $[\![t_1 \wedge t_2]\!]_v \subseteq [\![t_1]\!]_v \cap [\![t_2]\!]_v$.
Considering any value v in $[\![t_1]\!]_v \cap [\![t_2]\!]_v$, there exists s_i such that $\vdash v : s_i$ and $s_i \sqsubseteq_\emptyset t_i$. Then $\vdash v : s_1 \wedge s_2$ by Lemma 7.4.2 and $s_1 \wedge s_2 \sqsubseteq_\emptyset t_1 \wedge t_2$ by Lemma 9.1.4. So $v \in [\![t_1 \wedge t_2]\!]_v$. Hence $[\![t_1]\!]_v \cap [\![t_2]\!]_v \subseteq [\![t_1 \wedge t_2]\!]_v$.

(6): since $t \wedge \neg t \simeq \mathbb{0}$, we have

$$[\![t]\!]_v \cap [\![\neg t]\!]_v = [\![t \wedge \neg t]\!]_v = [\![\mathbb{0}]\!]_v = \emptyset.$$

It remains to prove that $[\![t]\!]_v \cup [\![\neg t]\!]_v = \mathscr{V}$, that is,

$$\begin{array}{l} \forall v \in \mathscr{V}. \forall t \in \mathcal{T}_0. \quad (\exists s_1. \vdash v : s_1 \text{ and } s_1 \sqsubseteq_\emptyset t) \\ \qquad\qquad\qquad\qquad\quad \text{or } (\exists s_2. \vdash v : s_2 \text{ and } s_2 \sqsubseteq_\emptyset \neg t). \end{array} \tag{12.1}$$

We prove (12.1) by induction on v and a case analysis on the pair (v, t).
First, to simplify the proof, given a value v, we use T to denote the set

$$\left\{ t \in \mathcal{T}_0 \;\middle|\; \begin{array}{l} (\exists s_1.\ \Delta \fatsemi \Gamma \vdash v : s_1 \text{ and } s_1 \sqsubseteq_\Delta t) \\ \text{or } (\exists s_2.\ \Delta \fatsemi \Gamma \vdash v : s_2 \text{ and } s_2 \sqsubseteq_\Delta \neg t) \end{array} \right\}.$$

and prove the following statement

$$T \text{ contains } \mathbb{0} \text{ and is closed under } \vee \text{ and } \neg \text{ (and thus } \wedge) \quad (**).$$

It is clear that T is closed under $\neg$ and invariant under $\simeq$. As $\Delta \fatsemi \Gamma \vdash v : \neg\mathbb{0} (\simeq \mathbb{1})$, then $\mathbb{0} \in T$. Consider $t_1, t_2 \in T$. If $\exists s \in \mathcal{T}.\ \Delta \fatsemi \Gamma \vdash v : s$ and $s \sqsubseteq_\Delta t_1 \vee t_2$, then the statement holds. Otherwise, $\forall s \in \mathcal{T}.\ \Delta \fatsemi \Gamma \vdash v : s \Rightarrow s \not\sqsubseteq_\Delta t_1 \vee t_2$. Then we have $\forall s \in \mathcal{T}.\ \Delta \fatsemi \Gamma \vdash v : s \Rightarrow s \not\sqsubseteq_\Delta t_i$. Since $t_i \in T$, then there exists s_i such that $\Delta \fatsemi \Gamma \vdash v : s_i$ and $s_i \sqsubseteq_\Delta \neg t_i$. By Lemma 7.4.2, $\Delta \fatsemi \Gamma \vdash v : s_1 \wedge s_2$, and by Lemma 9.1.4, $s_1 \wedge s_2 \sqsubseteq_\Delta \neg t_1 \wedge \neg t_2$, that is, $s_1 \wedge s_2 \sqsubseteq_\Delta \neg(t_1 \vee t_2)$. Thus the statement holds as well.
Thanks to the statement (**), we can assume that t is an atom $\boldsymbol{a}$. If v and $\boldsymbol{a}$ belong to different kinds of constructors, then it is clear that $\vdash v : \neg\boldsymbol{a}$. In what follows we assume that v and $\boldsymbol{a}$ have the same kind.

- $v = c$: we have $\vdash c : b_c$. As b_c is a singleton type, we have either $b_c \leq \boldsymbol{a}$ or $b_c \leq \neg\boldsymbol{a}$. Thus, either $b_c \sqsubseteq_\emptyset \boldsymbol{a}$ or $b_c \sqsubseteq_\emptyset \neg\boldsymbol{a}$.
- $v = (v_1, v_2), \boldsymbol{a} = t_1 \times t_2$: if $\exists s_i. \vdash v_i : s_i$ and $s_i \sqsubseteq_\emptyset t_i$, then $\vdash (v_1, v_2) : (s_1 \times s_2)$ by (*pair*) and $(s_1 \times s_2) \sqsubseteq_\emptyset (t_1 \times t_2)$ by Lemma 9.1.4. Thus (12.1) holds.
  Otherwise, assume that $\forall s_1. \vdash v_1 : s_1 \Rightarrow s_1 \not\sqsubseteq_\emptyset t_1$. Then by induction, there exists s_1' such that $\vdash v_1 : s_1'$ and $s_1' \sqsubseteq_\emptyset \neg t_1$. Besides, $\vdash v_2 : \mathbb{1}$ always holds. Thus, we get $\vdash (v_1, v_2) : (s_1' \times \mathbb{1})$ and $(s_1' \times \mathbb{1}) \sqsubseteq_\emptyset (\neg t_1 \times \mathbb{1}) \leq \neg(t_1 \times t_2)$. (12.1) holds as well.

$v = \lambda^{\wedge_{i\in I} t_i\to s_i}_{[\sigma_j]_{j\in J}} x.e, \boldsymbol{a} = t \to s$**:** we have $\vdash v : \bigwedge_{i\in I,j\in J} t_i\sigma_j \to s_i\sigma_j$. If $\bigwedge_{i\in I,j\in J} t_i\sigma_j \to s_i\sigma_j \sqsubseteq_\emptyset t \to s$ holds, thus (12.1) holds. Otherwise, we have $\bigwedge_{i\in I,j\in J} t_i\sigma_j \to s_i\sigma_j \not\sqsubseteq_\emptyset t \to s$. Since $t \to s$ is a ground type, we deduce that $\bigwedge_{i\in I,j\in J} t_i\sigma_j \to s_i\sigma_j \wedge \neg(t \to s) \not\sqsubseteq_\emptyset \mathbb{0}$. By (*abstr-neg*) and (*subsum*), we get $\vdash v : \neg(t \to s)$.

(7)**:** consequence of (5) and (6).

(8)**:** $\Leftarrow$: by (4).

$\Rightarrow$: since t is ground, we have $[\![t]\!]\eta = [\![t]\!]\eta'$ for all η and η'. So we can write $[\![t]\!]$ for $[\![t]\!]\eta$ where η is any semantics assignment. We prove the following statement:

$$\forall t \in \mathcal{T}_0.\ [\![t]\!] \neq \emptyset \Rightarrow [\![t]\!]_v \neq \emptyset$$

which implies that $[\![t]\!]_v = \emptyset \Rightarrow [\![t]\!] = \emptyset$ (*ie*, $\forall\eta.[\![t]\!]\eta = \emptyset$). Thus $t \simeq \mathbb{0}$. Since $[\![t]\!] \neq \emptyset$, there exists at least one element $d \in [\![t]\!]$. We prove the statement by induction on d.

$d = c$**:** we have $c \vdash c : b_c$. Since b_c is a singleton type, we have $[\![b_c]\!] \subseteq [\![t]\!]$ and thus $b_c \leq t$. Hence $b_c \sqsubseteq_\emptyset t$ and *a fortiori* $c \in [\![t]\!]_v$.

$d = (d_1, d_2)$**:** we can find $(P, N) \in \mathsf{dnf}(t)$ such that $d \in \bigcap_{(t_1\times t_2)\in P}[\![(t_1 \times t_2)]\!] \setminus \bigcup_{(t'_1\times t'_2)\in N}[\![(t'_1 \times t'_2)]\!]$. According to set-theory, there exists $N' \subseteq N$ such that $d_1 \in [\![s_1]\!]$ and $d_2 \in [\![s_2]\!]$, where

$$\begin{cases} s_1 = \bigwedge_{(t_1\times t_2)\in P} t_1 \wedge \bigwedge_{(t'_1\times t'_2)\in N'} \neg t'_1 \\ s_2 = \bigwedge_{(t_1\times t_2)\in P} t_2 \wedge \bigwedge_{(t'_1\times t'_2)\in N\setminus N'} \neg t'_2 \end{cases}$$

By induction on d_i, we have $[\![s_i]\!]_v \neq \emptyset$ and thus $[\![s_1 \times s_2]\!]_v \neq \emptyset$. Besides, by Lemma 4.3.9, we get $(s_1 \times s_2) \leq \bigwedge_{(t_1\times t_2)\in P}(t_1 \times t_2) \wedge \bigwedge_{(t'_1\times t'_2)\in N}(t'_1 \times t'_2)$ and thus $(s_1 \times s_2) \leq t$. According to the statement $(*)$ in the proof of (5), we deduce that $[\![t]\!]_v \neq \emptyset$.

$d = \{(d_1, d'_1), \ldots, (d_n, d'_n)\}$**:** we can find $(P, N) \in \mathsf{dnf}(t)$ such that $d \in \bigcap_{(t_1\to t_2)\in P}[\![(t_1 \to t_2)]\!] \setminus \bigcup_{(t'_1\to t'_2)\in N}[\![(t'_1 \to t'_2)]\!]$. We write t' for $\bigwedge_{(t_1\to t_2)\in P}(t_1 \to t_2) \wedge \bigwedge_{(t'_1\to t'_2)\in N}(t'_1 \to t'_2)$. Then we get $t' \not\leq \mathbb{0}$ and thus $t' \not\sqsubseteq_\emptyset \mathbb{0}$. Let us take the value $v = \mu^{\wedge_{(t_1\to t_2)\in P}(t_1\to t_2)} f.\ \lambda x.\ fx$. By (*abstr-neg*), we have $\vdash v : t'$. Moreover, it is clear that $t' \leq t$. Therefore $v \in [\![t]\!]_v$ (*ie*, $[\![t]\!]_v \neq \emptyset$).

□

Finally, similar to the techniques used in the proof of Corollary 4.6.8, we can extend this model to a convex value model for all the types by adding semantics assignment η_v for type variables.

In addition, concerning the circularity problem mentioned in Section 2.1, based on the value model, we define a new subtyping relation as

$$s \leq_v t \stackrel{\text{def}}{\iff} \forall \eta \in \mathscr{P}(\mathscr{V})^{\mathcal{V}}.[\![s]\!]_v\eta \subseteq [\![t]\!]_v\eta$$

Then we prove that these two subtyping relations $\leq$ and $\leq_v$ coincide (Theorem 12.3.2), so we close the circularity.

Theorem 12.3.2. *Let t be a type. Then $t \simeq \mathbb{0} \iff t \simeq_v \mathbb{0}$.*

Proof. $\Rightarrow$: straightforward.

$\Leftarrow$: we proceed by contrapositive: if $\exists\eta.[\![t]\!]\eta \neq \emptyset$ then $\exists\eta_v.[\![t]\!]_v\eta_v \neq \emptyset$. Similar to the proof of Lemma 4.6.5, we use the procedure explore_pos(t) to construct a value v and an assignment η_v such that $v \in [\![t]\!]_v\eta_v$, where the set from which we select the elements is $\mathscr{V}$ instead of $\mathcal{D}$. The existence of v and η_v is ensured by $\exists\eta.[\![t]\!]\eta \neq \emptyset$. □

12.4 Open type cases

As stated in Section 7.1, we restrict the condition types in type-cases to closed types. This was made by practical considerations: using open types in a type-case would have been computationally prohibitive insofar as it demands to solve tallying problems at run-time. But other motivations justify it: the logical meaning of such a choice is not clear and the compilation into monomorphic ℂDuce would not be possible without changing the semantics of the type-case. We leave for future work the study of type-cases on types with monomorphic variables (*ie*, those in Δ). This does not require dynamic type tallying resolution and would allow the programmer to test capabilities of arguments bound to polymorphic type variables.

Moreover, there is at least one case in which we should have been *more* restrictive, that is, when an expression that is tested in a type-case has a polymorphic type. Our inference system may type it (by deducing a set of type-substitutions that makes it closed), even if this seems to go against the intuition: we are testing whether a polymorphic expression has a closed type. Although completeness ensures that in some cases we can do it, in practice it seems reasonable to consider ill-typed any type-case in which the tested expression has an open type (see Section 9.3).

12.5 Value relabeling

We require the semantics for our calculus to be a call-by-value one to ensure subject reduction[2] (see Section 7.3). However, the notion of reduction for instantiations is

$$e[\sigma_j]_{j\in J} \rightsquigarrow e@[\sigma_j]_{j\in J}$$

which does not conform to call-by-value, as e may not be a value. There indeed exists a conforming one[3]:

$$v[\sigma_j]_{j\in J} \rightsquigarrow v@[\sigma_j]_{j\in J}.$$

This conforming reduction yields a different semantics, with respect to which the type system can be proved to satisfy the *safety* property as well. This follows from the fact that the second notation is a restricted version of the first notation. The main difference between these two semantics is the case where the expression e applied by the set of type substitutions is/contains a type case. Take the expression

[2]Note that the use of call-by-name would not hinder the soundness of the type system (expressed in terms of progress).

[3]In system F, the notion of reduction for instantiations is: $(\lambda\alpha.e)[t] \rightsquigarrow e\{{}^{t}/_{\alpha}\}$, where $\lambda\alpha.e$ is a value in system F.

$(\lambda^{\alpha\to\alpha}x.x{\in}\texttt{Int} \to \texttt{Int}\,?\,\texttt{true} : \texttt{false})[\{\texttt{Int}/\alpha\}]$ for example. Under the first notation, the set of type substitutions is applied to $\lambda^{\alpha\to\alpha}x.x{\in}\texttt{Int} \to \texttt{Int}\,?\,\texttt{true} : \texttt{false}$ first, yielding $\lambda^{\texttt{Int}\to\texttt{Int}}x.x{\in}\texttt{Int} \to \texttt{Int}\,?\,\texttt{true} : \texttt{false}$ (note that the set of type substitutions modifies the type of the argument of the type case). Then the type test is checked. As the type test is passed, the expression reduces to `true` at last. While under the second semantics, the type test is checked first. As the test fails, the type case reduces to `false`, on which the set of type substitutions is applied, which does not affect the value `false` (*ie*, the set of type substitutions is useless here). It looks a little strange that the type $\alpha \to \alpha$ is in the scope of $\{\texttt{Int}/\alpha\}$ but not be affected from it. Therefore, we prefer the first semantics. Moreover, it is more difficult to compile our calculus to Core$\mathbb{C}$Duce with respect to the second semantics as we should not propagate the set of type substitutions to the argument of type-cases (which is different from the relabeling).

12.6 Type reconstruction

Finally, to make the programmers' tasks easier, we would like to reconstruct the interfaces of abstractions, namely, to assign types for abstractions. Taking full advantage of the type tallying problem, we can infer the types for expressions, and thus those in the interfaces of abstractions. To this end, we define an *implicit* calculus without interfaces, for which we define a reconstruction system.

Definition 12.6.1. *An* implicit expression m *is an expression without any interfaces (or type substitutions). It is inductively generated by the following grammar:*

$$m \;::=\; c \;\mid\; x \;\mid\; (m,m) \;\mid\; \pi_i(m) \;\mid\; mm \;\mid\; \mu f\ \lambda x.m \;\mid\; m{\in}t\,?\,m : m$$

The type reconstruction for expressions has the form $\Gamma \vdash_{\mathcal{R}} e : t \rightsquigarrow \mathcal{S}$, which states that under the typing environment Γ, e has type t if there exists at least one constraint-set C in the set of constraint-sets $\mathcal{S}$ such that C are satisfied. The type reconstruction rules are given in Figure 12.1.

The rules (Recon-const), (Recon-var), (Recon-pair), (Recon-proj), (Recon-appl), and (Recon-abstr) are standard but differ from most of the type inference of other work in that they generate a set of constraint-sets rather than a single constraint-set. This is due to the type inference for type-cases. There are four possible cases for type-cases ((Recon-case)): (*i*) if no branch is selected, then the type t_0 inferred for the argument m_0 should be $\mathbb{0}$ (and the result type can be any type); (*ii*) if the first branch is selected, then the type t_0 should be a subtype of t and the result type α for the whole type-case should be a super-type of the type t_1 inferred for the first branch m_1; (*iii*) if the second branch is selected, then the type t_0 should be a subtype of $\neg t$ and the result type α should be a super-type of the type t_2 inferred for the second branch m_2; and (*iv*) both branches are selected, then the result type α should be a super-type of the union of t_1 and t_2 (note that the condition for t_0 is the one that does not satisfy (*i*), (*ii*) and (*iii*)). Therefore, there are four possible solutions for type-cases and thus four possible constraint-sets. Finally, the rule (Recon-case-var) deals with the type inference for the special binding type-case introduced in Section 11.1.

$$\frac{}{\Gamma \vdash_{\mathcal{R}} c : b_c \rightsquigarrow \{\emptyset\}}(\textsc{Recon-const}) \qquad \frac{}{\Gamma \vdash_{\mathcal{R}} x : \Gamma(x) \rightsquigarrow \{\emptyset\}}(\textsc{Recon-var})$$

$$\frac{\Gamma \vdash_{\mathcal{R}} m_1 : t_1 \rightsquigarrow \mathcal{S}_1 \qquad \Gamma \vdash_{\mathcal{R}} m_2 : t_2 \rightsquigarrow \mathcal{S}_2}{\Gamma \vdash_{\mathcal{R}} (m_1, m_2) : t_1 \times t_2 \rightsquigarrow \mathcal{S}_1 \sqcap \mathcal{S}_2}(\textsc{Recon-pair})$$

$$\frac{\Gamma \vdash_{\mathcal{R}} m : t \rightsquigarrow \mathcal{S}}{\Gamma \vdash_{\mathcal{R}} \pi_i(m) : \alpha_i \rightsquigarrow \mathcal{S} \sqcap \{\{(t, \leq, \alpha_1 \times \alpha_2)\}\}}(\textsc{Recon-proj})$$

$$\frac{\Gamma \vdash_{\mathcal{R}} m_1 : t_1 \rightsquigarrow \mathcal{S}_1 \qquad \Gamma \vdash_{\mathcal{R}} m_2 : t_2 \rightsquigarrow \mathcal{S}_2}{\Gamma \vdash_{\mathcal{R}} m_1 m_2 : \alpha \rightsquigarrow \mathcal{S}_1 \sqcap \mathcal{S}_2 \sqcap \{\{(t_1, \leq, t_2 \to \alpha)\}\}}(\textsc{Recon-appl})$$

$$\frac{\Gamma, (f : \alpha \to \beta), (x : \alpha) \vdash_{\mathcal{R}} m : t \rightsquigarrow \mathcal{S}}{\Gamma \vdash_{\mathcal{R}} \mu f\ \lambda x.m : \alpha \to \beta \rightsquigarrow \mathcal{S} \sqcap \{\{(t, \leq, \beta)\}\}}(\textsc{Recon-abstr})$$

$$\frac{\begin{array}{l}\Gamma \vdash_{\mathcal{R}} m_0 : t_0 \rightsquigarrow \mathcal{S}_0 \qquad (m_0 \notin \mathcal{X}) \\ \Gamma \vdash_{\mathcal{R}} m_1 : t_1 \rightsquigarrow \mathcal{S}_1 \\ \Gamma \vdash_{\mathcal{R}} m_2 : t_2 \rightsquigarrow \mathcal{S}_2 \\ \mathcal{S} = \quad (\mathcal{S}_0 \sqcap \{\{(t_0, \leq, \mathbb{0}), (\mathbb{0}, \leq, \alpha)\}\}) \\ \qquad \sqcup \quad (\mathcal{S}_0 \sqcap \mathcal{S}_1 \sqcap \{\{(t_0, \leq, t), (t_1, \leq, \alpha)\}\}) \\ \qquad \sqcup \quad (\mathcal{S}_0 \sqcap \mathcal{S}_2 \sqcap \{\{(t_0, \leq, \neg t), (t_2, \leq, \alpha)\}\}) \\ \qquad \sqcup \quad (\mathcal{S}_0 \sqcap \mathcal{S}_1 \sqcap \mathcal{S}_2 \sqcap \{\{(t_0, \leq, \mathbb{1}), (t_1 \vee t_2, \leq, \alpha)\}\})\end{array}}{\Gamma \vdash_{\mathcal{R}} (m_0 \in t\,?\,m_1 : m_2) : \alpha \rightsquigarrow \mathcal{S}}(\textsc{Recon-case})$$

$$\frac{\begin{array}{l}\Gamma, (x : \Gamma(x) \wedge t) \vdash_{\mathcal{R}} m_1 : t_1 \rightsquigarrow \mathcal{S}_1 \\ \Gamma, (x : \Gamma(x) \wedge \neg t) \vdash_{\mathcal{R}} m_2 : t_2 \rightsquigarrow \mathcal{S}_2 \\ \mathcal{S} = \quad (\{\{(\Gamma(x), \leq, \mathbb{0}), (\mathbb{0}, \leq, \alpha)\}\}) \\ \qquad \sqcup \quad (\mathcal{S}_1 \sqcap \{\{(\Gamma(x), \leq, t), (t_1, \leq, \alpha)\}\}) \\ \qquad \sqcup \quad (\mathcal{S}_2 \sqcap \{\{(\Gamma(x), \leq, \neg t), (t_2, \leq, \alpha)\}\}) \\ \qquad \sqcup \quad (\mathcal{S}_1 \sqcap \mathcal{S}_2 \sqcap \{\{(\Gamma(x), \leq, \mathbb{1}), (t_1 \vee t_2, \leq, \alpha)\}\})\end{array}}{\Gamma \vdash_{\mathcal{R}} (x\ \boldsymbol{\epsilon}\ t\,?\,m_1 : m_2) : \alpha \rightsquigarrow \mathcal{S}}(\textsc{Recon-case-var})$$

where α, α_i and β in each rule are fresh type variables.

Figure 12.1: Type reconstruction rules

The idea of the type reconstruction is that, given an implicit expression m and a typing environment Γ (which is always empty), we first collect the set $\mathcal{S}$ of constraint-sets and then infer a possible type t which contains some fresh type variables in $\mathcal{S}$. In particular, functions are initially typed by a generic function type $\alpha \to \beta$, where α, β are fresh type variables. Finally, to find solutions for t, we just look for any substitution σ that satisfies any constraint-set $C \in \mathcal{S}$. If there are no such substitutions, then m is

not typable.

Consider an implicit version of `map`, which can be defined as:

$$\mu m\ \lambda f\ .\ \lambda \ell\ .\ \ell\ \mathbf{\epsilon}\ \mathtt{nil}\ \mathtt{?}\ \mathtt{nil}\ \mathtt{:}\ (f(\pi_1\ell)\mathtt{,}mf(\pi_2\ell)\mathtt{)}$$

The type inferred for `map` by the type reconstruction system is $\alpha_1 \to \alpha_2$ and the generated set $\mathcal{S}$ of constraint-sets is:

$$\begin{array}{ll}\{ & \{\alpha_3 \to \alpha_4 \leq \alpha_2, \alpha_5 \leq \alpha_4, \alpha_3 \leq \mathbb{0}, \mathbb{0} \leq \alpha_5\},\\ & \{\alpha_3 \to \alpha_4 \leq \alpha_2, \alpha_5 \leq \alpha_4, \alpha_3 \leq \mathtt{nil}, \mathtt{nil} \leq \alpha_5\},\\ & \{\alpha_3 \to \alpha_4 \leq \alpha_2, \alpha_5 \leq \alpha_4, \alpha_3 \leq \neg\mathtt{nil}, (\alpha_6 \times \alpha_9) \leq \alpha_5\} \cup C,\\ & \{\alpha_3 \to \alpha_4 \leq \alpha_2, \alpha_5 \leq \alpha_4, \alpha_3 \leq \mathbb{1}, (\alpha_6 \times \alpha_9) \vee \mathtt{nil} \leq \alpha_5\} \cup C \quad \}\end{array}$$

where C is $\{\alpha_1 \leq \alpha_7 \to \alpha_6, \alpha_3 \setminus \mathtt{nil} \leq (\alpha_7 \times \alpha_8), \alpha_1 \to \alpha_2 \leq \alpha_1 \to \alpha_{10}, \alpha_3 \setminus \mathtt{nil} \leq (\alpha_{11} \times \alpha_{12}), \alpha_{10} \leq \alpha_{12} \to \alpha_9\}$. Then applying the tallying algorithm to the sets, we get the following types for `map`:

$$\begin{array}{l}\alpha_1 \to (\mathbb{0} \to \alpha_5)\\ \alpha_1 \to (\mathtt{nil} \to \mathtt{nil})\\ \mathbb{0} \to ((\alpha_7 \wedge \alpha_{11} \times \alpha_8 \wedge \alpha_{12}) \to (\alpha_6 \times \alpha_9))\\ (\alpha_7 \to \alpha_6) \to (\mathbb{0} \to (\alpha_6 \times \alpha_9))\\ (\alpha_7 \to \alpha_6) \to (\mathbb{0} \to [\alpha_6])\\ \mathbb{0} \to ((\mathtt{nil} \vee (\alpha_7 \wedge \alpha_{11} \times \alpha_8 \wedge \alpha_{12})) \to (\mathtt{nil} \vee (\alpha_6 \times \alpha_9)))\\ (\alpha_7 \to \alpha_6) \to (\mathtt{nil} \to (\mathtt{nil} \vee (\alpha_6 \times \alpha_9)))\\ (\alpha_7 \to \alpha_6) \to ((\mu x.\ \mathtt{nil} \vee (\alpha_7 \wedge \alpha_{11} \times \alpha_8 \wedge x)) \to [\alpha_6])\end{array}$$

By replacing type variables that only occur positively (see Definition 4.6.3) by $\mathbb{0}$ and those only occurring negatively by $\mathbb{1}$, we obtain

$$\begin{array}{l}\mathbb{1} \to (\mathbb{0} \to \mathbb{0})\\ \mathbb{1} \to (\mathtt{nil} \to \mathtt{nil})\\ \mathbb{0} \to ((\mathbb{1} \times \mathbb{1}) \to \mathbb{0})\\ (\mathbb{0} \to \mathbb{1}) \to (\mathbb{0} \to \mathbb{0})\\ (\mathbb{1} \to \beta) \to (\mathbb{0} \to [\beta])\\ \mathbb{0} \to ((\mathtt{nil} \vee (\mathbb{1} \times \mathbb{1})) \to \mathtt{nil})\\ (\mathbb{0} \to \mathbb{1}) \to (\mathtt{nil} \to \mathtt{nil})\\ (\alpha \to \beta) \to ([\alpha] \to [\beta])\end{array}$$

All the types, except the last two, are useless[4], as they provide no further information. Thus we deduce the following type for `map`:

$$((\alpha \to \beta) \to ([\alpha] \to [\beta])) \wedge ((\mathbb{0} \to \mathbb{1}) \to (\mathtt{nil} \to \mathtt{nil}))$$

which is more precise than $(\alpha \to \beta) \to ([\alpha] \to [\beta])$ since it states that the application of `map` to any function and the empty list returns the empty list.

[4]Similar to the type-substitutions inference for the application problems (see Section 10.2.2), these useless types are generated from the fact that $\mathbb{0} \to t$ contains all the functions, or the fact that $(\mathbb{0} \times t)$ or $(t \times \mathbb{0})$ is a subtype of any type, or the fact that Case (i) in type-cases is useless in practice.

12.7 Related work

In this section we present some related work only on explicitly-typed calculi for intersection type systems, constraint-based type inference, and inference for intersection type systems.

12.7.1 Explicitly typed λ-calculus with intersection types

This section present related work on explicitly-typed calculi for intersection type systems by discussing how our term

$$(\lambda^{\alpha\to\alpha}_{[\{\mathtt{Int}/\alpha\},\{\mathtt{Bool}/\alpha\}]}x.(\lambda^{\alpha\to\alpha}y.x)x) \tag{12.2}$$

is rendered.

In [Ron02, WDMT02], typing derivations are written as terms: different typed representatives of the same untyped term are joined together with an intersection $\wedge$. In such systems, the function in (12.2) is written

$$(\lambda^{\mathtt{Int}\to\mathtt{Int}}x.(\lambda^{\mathtt{Int}\to\mathtt{Int}}y.x)x) \wedge (\lambda^{\mathtt{Bool}\to\mathtt{Bool}}x.(\lambda^{\mathtt{Bool}\to\mathtt{Bool}}y.x)x).$$

Type checking verifies that both $\lambda^{\mathtt{Int}\to\mathtt{Int}}x.(\lambda^{\mathtt{Int}\to\mathtt{Int}}y.x)x$ and $\lambda^{\mathtt{Bool}\to\mathtt{Bool}}x.(\lambda^{\mathtt{Bool}\to\mathtt{Bool}}y.x)x$ are well typed separately, which generates two very similar typing derivations. The proposal of [LR07] follows the same idea, except that a separation is kept between the computational and the logical contents of terms. A term consists in the association of a *marked term* and a *proof term*. The marked term is just an untyped term where term variables are marked with integers. The proof term encodes the structure of the typing derivation and relates marks to types. The aforementioned example is written as

$$(\lambda x:0.(\lambda y:1.x)x)@((\lambda 0^{\mathtt{Int}}.(\lambda 1^{\mathtt{Int}}.0)0) \wedge (\lambda 0^{\mathtt{Bool}}.(\lambda 1^{\mathtt{Bool}}.0)0))$$

in this system. In general, different occurrences of a same mark can be paired with different types in the proof term. In [BVB08], terms are duplicated (as in [Ron02, WDMT02]), but the type checking of terms does not generate copies of almost identical proofs. The type checking derivation for the term

$$((\lambda^{\mathtt{Int}\to\mathtt{Int}}x.(\lambda^{\mathtt{Int}\to\mathtt{Int}}y.x)x)\|\lambda^{\mathtt{Bool}\to\mathtt{Bool}}x.(\lambda^{\mathtt{Bool}\to\mathtt{Bool}}y.x)x)$$

verifies in parallel that the two copies are well typed. The duplication of terms and proofs makes the definition of beta reduction (and other transformations on terms) more difficult in the calculi presented so far, because it has to be performed in parallel on all the typed instances that correspond to the same untyped term. *Branching types* have been proposed in [WH02] to circumvent this issue. The idea is to represent different typing derivations for a same term into a compact piece of syntax. To this end, the branching type which corresponds to a given intersection type t records the *branching shape* (*ie*, the uses of the intersection introduction typing rule) of the typing derivation corresponding to t. For example, the type $(\mathtt{Int}\to\mathtt{Int}) \wedge (\mathtt{Bool}\to\mathtt{Bool})$ has only two branches, which is represented in [WH02] by the branching shape $\mathsf{join}\{i=*, j=*\}$. Our running example is then written as

$$\Lambda\mathsf{join}\{i=*, j=*\}.\lambda x^{\{i=\mathtt{Int},j=\mathtt{Bool}\}}.(\lambda y^{\{i=\mathtt{Int},j=\mathtt{Bool}\}}.x)x.$$

Note that the lambda term itself is not copied, and no duplication of proofs happens during type checking either: the branches labeled i and j are verified in parallel.

12.7.2 Constraint-based type inference

Type inference in ML has essentially been considered as a constraint solving problem [OSW97, PR05]. We use a similar approach to solve the problem of type unification: finding a proper substitution that make the type of the domain of a function compatible with the type of the argument it is applied to. Our type unification problem is essentially a specific set constraint problem [AW93]. This is applied in a much more complex setting with a complete set of type connectives and a rich set-theoretic subtyping relation. In particular because of the presence of intersection types to solve the problem of application one has to find a set of substitutions rather than a single one. This is reflected by the definition of our $\sqsubseteq$ relation which is much more thorough than the corresponding relation used in ML inference insofar as it encompasses instantiation, expansion, and subtyping. The important novelty of our work comes from our use of set-theoretic connectives, which allows us to turn sets of constraints of the form $s \leq \alpha \leq t$, into sets of equations of the form $\alpha = (\beta \vee s) \wedge t$. This set of equations is then solved using the Courcelle's work on infinite trees [Cou83].

12.7.3 Inference for intersection type systems

Coppo and Giannini presented a decidable type checking algorithm for simple intersection type system [CG95] where intersection is used in the left-hand side of an arrow and only a term variable is allowed to have different types in its different occurrences. They introduced labeled intersections and labeled intersection schemes, which are intended to represent potential intersections. During an application $M\ N$, the labeled intersection schemes of M and N would be unified to make them match successfully, yielding a transformation, a combination of substitutions and expansions. An expansion expands a labeled intersection into an explicit intersection. The intersection here acts like a variable binding similar to a quantifier in logic. Our rule (*instinter*) is similar to the transformation. We instantiate a quantified type into several instances according to different situations (i.e., the argument types), and then combine them as an intersection type. The difference is that we instantiate a parametric polymorphic function into a function with intersection types, while Coppo and Giannini transform a potential intersection into an explicit intersection. Besides, as the general intersection type system is not decidable [CD78], to get a decidable type checking algorithm, Coppo and Giannini used the intersection in a limited way, while we give some explicit type annotations for functions.

Restricting intersection types to finite ranks (using Leivant's notion of rank [Lei83]) also yields decidable systems. Van Bakel [Bak93] gave the first unification-based inference algorithm for a rank 2 intersection type system. Jim [Jim96] studied a decidable rank 2 intersection type system extended with recursion and parametric polymorphism. Kfoury and Wells proved decidability of type inference for intersection type systems of arbitrary finite rank [KW99]. As a future work, we want to investigate decidability of rank restricted versions of our calculus. A type inference algorithm have also been proposed for a polar type system [Jim00], where intersection is allowed only in negative positions and System F-like quantification only in positive ones.

Part IV

Conclusion

Chapter 13

Conclusion and Future Work

13.1 Summary

XML is a current standard format for exchanging structured data, which has been applied to web services, database, research on formal methods, and so on. Many recent XML processing languages are statically typed functional languages. However, parametric polymorphism, an essential feature of such languages is still missing, or when present it is in a limited form. In this dissertation, we have studied the techniques to extend parametric polymorphism into XML processing languages, which consist of two parts: a definition of a polymorphic semantic subtyping relation and a definition of a polymorphic calculus.

In Part II, we have defined and studied a polymorphic semantic subtyping relation for a type system with recursive, product and arrow types and set-theoretic type connectives (*i.e.*, union, intersection and negation). Instead of replacing type variables by ground types, we assign them by subsets of the domain set. The assignments allow type variables to range over subsets of any type. Thus we define the subtyping relation as the inclusion of denoted sets under all the assignments for type variables. The definition is semantic, intuitive (for the programmer) and decidable. As far as we know, our definition is the first solution to the problem of defining a semantic subtyping relation for a polymorphic extension of regular tree types.

In Part III, we have designed and studied a polymorphism calculus, which is a polymorphic extension of Core$\mathbb{C}$Duce and takes advantage of the new capabilities of the subtyping definition of Part II. The novelty of the polymorphic extension is to decorate λ-abstractions with sets of type-substitutions and to lazily propagate type-substitutions at the moment of the reduction (and type-checking). Our calculus is also an explicitly-typed λ-calculus with intersection (and union and negation) types, which contrasts with current solutions in the literature which require the addition of new operators, stores and/or pointers. In practice, the sets of type-substitutions are transparent to the programmer. For that we have proposed an inference system that infers whether and where type-substitutions can be inserted into an expression to make it well-typed. Our inference algorithm is sound, complete, and semi-decision: decidability is an open problem, yet. Finally, to provide an execution model for our calculus, we have studied the translation from our polymorphic calculus into a variant of Core$\mathbb{C}$Duce.

13.2 Future work

The work presented above provides all the theoretical basis and machinery needed to start the design and implementation of polymorphic functional languages for semi-structured data. There are several problems still to be solved or several continuations to be studied, which are listed below.

Extensions of the type system

The first continuation concerns the definition of extensions of the type system itself. Among the possible extensions the most interesting (and difficult) one seems to be the extension of types with explicit second order quantifications. Currently, we consider prenex polymorphism, thus quantification on types is performed only at meta-level. But since this work proved the feasibility of a semantic subtyping approach for polymorphic types, we are eager to check whether it can be further extended to impredicative second order types, by adding explicit type quantification. This would be interesting not only from a programming language perspective, but also from a logic viewpoint since it would remove some of the limitations to the introspection capabilities we pointed out in Section 3.5. This may move our type system closer to being an expressive logic for subtyping. On the model side, it would be interesting to check whether the infinite support property (Definition 4.6.1) is not only a sufficient but also a necessary condition for convexity. This seems likely to the extent that the result holds for the type system restricted to basic type constructors (*ie*, without products and arrows). However, this is just a weak conjecture since the proof of sufficiency heavily relies (in the case for product types) on the well-foundedness property. Therefore, there may even exist non-well-founded models (non-well-founded models exist in the ground case: see [FCB08, Fri04]) that are with infinite support but not convex. Nevertheless, an equivalent characterization of convexity—whether it were infinite support or some other characterization—would provide us a different angle of attack to study the connections between convexity and parametricity.

Convexity and parametricity

In our opinion, the definition of convexity is the most important and promising theoretical contribution of our work especially in view of its potential implications on the study of parametricity. As a matter of fact, there are strong connections between parametricity and convexity. We already saw that convexity removes the stuttering behavior that clashes with parametricity, as equation (3.2) clearly illustrates. More generally, both convexity and parametricity describe or enforce uniformity of behavior. Parametricity imposes functions to behave uniformly behavior on parameters typed by type variables, since the latter cannot be deconstructed. This allows Wadler to deduce "theorems for free": the uniformity imposed by parametricity (actually, imposed by the property of being definable in second order λ-calculus) dramatically restricts the choice of possible behaviors of parametric functions to a point that it is easy to deduce theorems about a function just by considering its type [Wad89]. In a similar way convexity imposes a uniform behavior to the zeros of the semantic interpretation, which is equivalent to imposing uniformity to the subtyping relation. An example of this uniformity is given

by product types: in our framework a product $(t_1 \times ... \times t_n)$ is empty (*ie*, it is empty for every possible assignment of its variables) if and only if there exists a particular t_i that is empty (for all possible assignments). We recover a property typical of closed types.

We conjecture the connection to be deeper than described above. This can be perceived by rereading the original Reynolds paper on parametricity [Rey83] in the light of our results. Reynolds tries to characterize parametricity—or *abstraction* in Reynolds terminology—in a set-theoretic setting since, in Reynolds words, "if types denote specific subsets of a universe then their unions and intersections are well defined", which in Reynolds opinion is the very idea of abstraction. This can be rephrased as the fact that the operations for some types are well defined independently from the representation used for each type (Reynolds speaks of abstraction and representation since he sees the abstraction property as a result about change of representation). The underlying idea of parametricity according to Reynolds is that "meanings of an expression in 'related' environments will be 'related'" [Rey83]. But as he points out few lines later "while the relation for each type variable is arbitrary, the relation for compound type expressions (*ie*, type constructors) must be induced in a specified way. We must specify how an assignment of relations to type variables is extended to type expressions" [Rey83]. Reynolds formalizes this extension by defining a "relation semantics" for type constructors and, as Wadler brilliantly explains [Wad89], this corresponds to regard types as relations. In particular pairs are related if their corresponding components are related and functions are related if they take related arguments into related results: there is a precise correspondence with the extensional interpretation of type constructors we gave in Definition 4.2.6 and, more generally, between the framework used to state parametricity and the one in our work.

Parametricity holds for terms written in the Girard/Reynolds second order typed lambda calculus (also known as pure polymorphic lambda calculus or System F [Gir71, Rey74]). The property of being definable in the second-order typed lambda-calculus is *the* condition that harnesses expressions and forces them to behave uniformly. Convexity, instead, does not require any definability property. It *semantically* harnesses the denotations of expressions and forces them to behave uniformly. This seems to suggest that convexity is a semantic property related to, albeit weaker than, what in Reynolds approach is the definability in the second-order typed lambda-calculus, a syntactic (rather, syntactically-rooted) property.

Although we have this intuition about the connection between convexity and parametricity, we do not know how to express this connection in a formal way, yet. We believe that the answer may come from the study of the calculus associated to our subtyping relation. We do not speak here of some language that can take advantage of our subtyping relation and whose design space we discussed earlier in this thesis. What we are speaking of is every calculus whose model of values (*ie*, the model obtained by associating each type to the set of values that have that type) induces the same subtyping relation as the one devised here. Indeed, as parametricity leaves little freedom to the definition of transformations, so the semantic subtyping framework leaves little freedom to the definition of a language whose model of values induces the same subtyping relation as the relation used to type its values. If we could determine under which conditions every such language satisfied Reynolds abstraction theorem, then we

would have established a formal connection between convexity and parametricity. But this is a long term and ambitious goal that goes well beyond the scope of the present work.

Decidability

The problem of the decidability of inference of type-substitutions is unsolved yet. The property that our calculus can express intersection type systems *à la* Barendregt, Coppo, and Dezani [BCD83] is not of much help: if we know that a term is well-typed, then it is not difficult to extract from its type the interfaces of the functions occurring in it; undecidability for intersection type systems tells us that it is not possible to decide whether these interfaces exist; but here we are in a simpler setting where the interfaces are given and we "just" want to decide whether they are correct. In particular, the problem we cannot solve is when should we stop trying to expand the types of a function and of its argument. The experience with intersection types seems to suggest that this should not be a problem since undecidability for intersection type systems is due to expansions taking place at different depths. As Kfoury and Wells proved [KW99], it suffices to bound the rank of the types (*ie*, the maximum dept —counted in terms of arrow types— at which intersections can appear) to obtain decidability. As a matter of fact, in our case intersections do not seem to be the main problem: unions are the tough problem, for when we expand a union, then it distributes over intersections thus creating new combinations of types that did not exist before. Trying to restrict our approach only to intersection types seems unfeasible since the use of set-theoretic laws involving a complete set of connectives is so pervasive of our approach that we do not see how the use of unions or negations could be avoided. And in any case it would hinder the application of this theory to XML document processing that, we recall, is our initial motivation. The hope here is either to prove decidability (which looks as the most likely outcome) or at least find some reasonable restrictions on types so as to keep under control the generation of union types and thus limit the number of different expansions to be tried by the algorithm.

In addition, as stated in Section 10.2.3, when computing $\mathsf{App}_\Delta(t, s)$, the expansion of the type t of functions would not cause the expansion of the type s of arguments. This suggests us to expand t with a sufficient number of copies (although the number is not clear yet) and thus we only need to consider the expansion of s, which is the case of $\sqsubseteq_\Delta$. Then a problem in mind is: whether the decidable problem of type-substitution inference can be reduced to the decidable problem of the preorder $\sqsubseteq_\Delta$.

Principal type

We did not tackle the problem that whether our inference system has principal types, due to the unknown of the decidability of type-substitution inference. From a set-theoretic view, this problem is clearly equivalent to deciding whether both the sets $\Pi^i_\Delta(t)$ and $t \bullet_\Delta s$ have minimums with respect to $\sqsubseteq_\Delta$. Clearly, it requires the decidability of $\sqsubseteq_\Delta$. Moreover, we have proved that both sets are closed by intersection (Lemmas 9.1.7 and 9.1.8), which implies that if the minimals of these two sets exist and are finitely many, the minimums must exist (by taking the intersection of all the minimals).

Even if the inference of type-substitutions were decidable, it does not ease the proof of the existence of principle types. As the example (10.6) in Section 10.2.3 shows, one further iteration of the algorithm $\mathsf{App}_\Delta(t, s)$ allows the system to deduce a more precise type. This raises the problem: may an infinite sequence of increasingly general solutions exist? If the answer were negative, then it would be easy to prove the existence of a principal type, which would be the intersection of the minimal solutions of the last iteration.

Non-determinism

The non determinism of the implicitly typed calculus has a negligible practical impact, insofar as it is theoretical (in practice, the semantics is deterministic but implementation dependent) and it concerns only the case when the type of (an instance of) a polymorphic function is tested: in our programming experience with ℂDuce we never met a test for a function type. Nevertheless, it may be interesting to study how to remove such a latitude either by defining a canonical choice for the instances deduced by the inference system (a problem related to the existence of principal types), or by imposing reasonable restrictions, or by checking the flow of polymorphic functions by a static analysis.

Efficient compilation

On the practical side the most interesting direction of research is to study efficient compilation of the polymorphic language. A naive compilation technique that would implement the beta reduction of the explicitly-typed calculus as defined by (*Rappl*) is out of question, since it would be too inefficient. The compilation technique described in Chapter 11 coupled with some optimization techniques that would limit the expansion in type-cases to few necessary cases can provide a rapidly available solution for a prototype implementation that reuses the efficient run-time engine of ℂDuce (interestingly, with such a technique the compilation does not need to keep any information about the source: the compiled term has all the information needed). However it could not scale up to a full-fledged language since it is not modular (a function is compiled differently according to the argument it is applied to) and the use of the nested type-cases, even in a limited number of cases, causes an exponential blow-up that hinders any advantage of avoiding dynamic relabeling. We think that the solution to follow is a smart compilation in typed virtual machines using JIT compilation techniques [Ayc03]. Since the types in the interfaces of λ-abstractions are used to determine the semantics of the type-cases, then relabeling cannot be avoided. However relabeling is necessary only in one very particular situation, that is in case of partial application of curried polymorphic functions. Therefore we plan to explore compilation techniques such as those introduced in the ZINC virtual machine [Ler90] to single out partial applications and apply possible relabeling just to them. This can be further coupled with static analysis techniques that would limit relabeling only to partial applications that may end up in a type cases or escape in the result, yielding an efficient run-time on par with the monomorphic version of ℂDuce.

Type reconstruction

We have shown how to reuse the machinery developed for the inference of type-substitutions to perform type reconstruction in Section 12.6, that is, to assign types to functions whose interfaces are not specified. The idea is to type the body of the function under the hypothesis that the function has the generic type $\alpha \to \beta$ and deduce the corresponding constraints. However, the work is rough and there are still many problems to be studied.

The soundness and the termination of the reconstruction system is easy to be proved, because of the soundness of the type tallying problem and the finiteness of expressions respectively. Completeness instead is not clear: given a well-typed implicitly-typed expression a, it is not clear whether all the interfaces in a can be reconstructed (since we do not consider the expansion in the reconstruction rule for applications), and if so, whether the reconstructed types are more general than those in the interfaces. Meanwhile, the study of whether the type-substitution inference system and the type reconstruction system can be merged seamlessly (*e.g.*, to use (Inf-appl) instead of (Reg-appl)) is worth pursuing as well.

Another problem worth studying is the question of how far the type reconstruction system can go away. Conservatively, we believe the reconstruction embraces the type inference of ML, which is our minimum requirement. As a future work, we would like to extend it to a type inference system for λ-calculus with intersection types. In a nutshell, the problem of inference with intersection types is "expansion" [CDCV81, CG92]. The key idea is to locate the possible "expansions" in types, which are the function types inferred for abstractions (corresponding to the sets of type-substitutions in the decorations of abstractions), and to solve the "expansions" when inferring types for applications, where we need to tally the type for the argument with the domain of the type for the function. Of course, general type reconstruction is undecidable, but we reckon it should not be hard to find reasonable restrictions on the complexity of the inferred types to ensure decidability without hindering type inference in practical cases.

Extensions of the language

The overall design space for a programming language that could exploit the advanced features of our types is rather large. Consequently, besides the extensions and design choices presented in Chapter 12, there are a lot of possible variations can be considered (*e.g.*, the use of type variables in pattern matching) and even more features can be encoded (*e.g.*, as explained in Footnote 3, bounded quantification can only be encoded via type connectives). While exploring this design space it will be interesting to check whether our polymorphic union types can encode advanced type features such as polymorphic variants [Gar98] and GADTs [XCC03].

Notice that the usage of values (*i.e.*, the call-by-value semantics) ensures subject-reduction but it is not a necessary condition: in some cases weaker constraints could be used. For instance, in order to check whether an expression is a list of integers, in general it is not necessary to fully evaluate the whole list: the head and the type of the tail are all that is needed. Studying weaker conditions for the reduction rules is

an interesting topic we leave for future work, in particular in the view of adapting our framework to lazy languages.

Implementation issues

Finally, besides the non-determinism of the implicitly-typed calculus and the efficient compilation problem, there exist other issues to be considered for the implementation: the order on type variables (which will affect the result type), the elimination of the useless result types, the trade-off between precision and efficiency (when to stop $\mathsf{App}_\Delta(t, s)$), and so on.

Bibliography

[AKW95] A. Aiken, D. Kozen, and E. Wimmers. Decidability of systems of set constraints with negative constraints. *Information and Computation*, 122(1):30–44, 1995.

[AW93] Alexander Aiken and Edward L. Wimmers. Type inclusion constraints and type inference. In *Proceedings of the Seventh ACM Conference on Functional Programming and Computer Architecture*, pages 31–41, Copenhagen, Denmark, June 93.

[Ayc03] John Aycock. A brief history of just-in-time. *ACM Comput. Surv.*, 35(2):97–113, June 2003.

[Bak92] Steffen Van Bakel. Complete restrictions of the intersection type discipline. *Theoretical Computer Science*, 102:135–163, 1992.

[Bak93] Steffen Van Bakel. *Intersection Type Disciplines in Lambda Calculus and Applicative Term Rewriting Systems*. PhD thesis, Catholic University of Nijmegen, 1993.

[BCD83] H. Barendregt, M. Coppo, and M. Dezani-Ciancaglini. A filter lambda model and the completeness of type assignment. *Journal of Symbolic Logic*, 48(4):931–940, 1983.

[BCF03a] V. Benzaken, G. Castagna, and A. Frisch. CDuce: an XML-friendly general purpose language. In *ICFP '03*. ACM Press, 2003.

[BCF+03b] S. Boag, D. Chamberlin, M. Fernandez, D. Florescu, J. Robie, J. Siméon, and M. Stefanescu. *XQuery 1.0: An XML Query Language*. W3C Working Draft, `http://www.w3.org/TR/xquery/`, May 2003.

[BLFM05] Tim Berners-Lee, Roy T. Fielding, and Larry Masinter. *Uniform Resource Identifier*, January 2005. RFC 3986, STD 66.

[BS01] Franz Baader and Wayne Snyder. Unification theory. In *Handbook of Automated Reasoning*, pages 445–532. Elsevier and MIT Press, 2001.

[BVB08] V. Bono, B. Venneri, and L. Bettini. A typed lambda calculus with intersection types. *Theor. Comput. Sci.*, 398(1-3):95–113, 2008.

[BVY09] V. Balat, J. Vouillon, and B. Yakobowski. Experience report: Ocsigen, a web programming framework. In *ICFP '09*. ACM Press, 2009.

[CD78] M. Coppo and M. Dezani. A new type assignment for lambda-terms. *Archiv für mathematische Logik und Grundlagenforschung*, 19:139–156, 1978.

[CDCV81] M. Coppo, M. Dezani-Ciancaglini, and B. Venneri. Functional characters of solvable terms. *Mathematical Logic Quarterly*, 27:45–58, 1981.

[CDG+97] H. Comon, M. Dauchet, R. Gilleron, F. Jacquemard, D. Lugiez, S. Tison, and M. Tommasi. Tree automata techniques and applications. Available at: `http://www.grappa.univ-lille3.fr/tata`, 1997. Release October, the 1st 2002.

[CDV80] M. Coppo, M. Dezani, and B. Venneri. Principal type schemes and lambda-calculus semantics. In *To H.B. Curry. Essays on Combinatory Logic, Lambda-calculus and Formalism*. Academic Press, 1980.

[CDV08] G. Castagna, R. De Nicola, and D. Varacca. Semantic subtyping for the π-calculus. *Theor. Comput. Sci.*, 398(1-3):217–242, 2008.

[CG92] Mario Coppo and Paola Giannini. A complete type inference algorithm for simple intersection types. In *Proceedings of the 17th Colloquium on Trees in Algebra and Programming*, CAAP '92, pages 102–123, London, UK, UK, 1992. Springer-Verlag.

[CG95] Mario Coppo and Paola Giannini. Principal types and unification for simple intersection type systems. *Inf. Comput.*, 122:70–96, October 1995.

[CM01] J. Clark and M. Murata. Relax-NG, 2001. `www.relaxng.org`.

[CN08] G. Castagna and K. Nguyen. Typed iterators for XML. In *ICFP '08*, pages 15–26. ACM Press, 2008.

[CNXL12] G. Castagna, K. Nguyễn, Z. Xu, and S. Lenglet. Polymorphic functions with set-theoretic types. Unpublished, Dec 2012.

[Cou83] B. Courcelle. Fundamental properties of infinite trees. *Theoretical Computer Science*, 25:95–169, 1983.

[CX11] G. Castagna and Z. Xu. Set-theoretic Foundation of Parametric Polymorphism and Subtyping. In *ICFP '11*, 2011.

[Dam94] F. Damm. Subtyping with union types, intersection types and recursive types II. Research Report 816, IRISA, 1994.

[DFF+07] Denise Draper, Peter Fankhauser, Mary Fernández, Ashok Malhotra, Michael Rys, Kristoffer Rose, Jérôme Siméon, and Philip Wadler. XQuery 1.0 and XPath 2.0 Formal Semantics, 2007. `http://www.w3.org/TR/query-semantics/`.

[DTD06] W3C: DTD specifications. `http://www.w3.org/TR/REC-xml/#dt-doctype`, 2006.

[Env] Envelope Schema in SOAP. `http://schemas.xmlsoap.org/soap/envelope/`.

[FCB08] A. Frisch, G. Castagna, and V. Benzaken. Semantic subtyping: dealing set-theoretically with function, union, intersection, and negation types. *The Journal of ACM*, 55(4):1–64, 2008.

[Fri04] A. Frisch. *Théorie, conception et réalisation d'un langage de programmation fonctionnel adapté à XML*. PhD thesis, Université Paris 7, 2004.

[Fri06] Alain Frisch. OCaml + XDuce. In *ICFP '06, Proceedings of the 11th ACM SIGPLAN International Conference on Functional Programming*, pages 192–200, 2006.

[Gar98] J. Garrigue. Programming with polymorphic variants. In *Proc. of ML Workshop*, 1998.

[GGL11] N. Gesbert, P. Genevès, and N. Layaïda. Parametric Polymorphism and Semantic Subtyping: the Logical Connection. In *ICFP '11*, 2011.

[Gir71] Jean-Yves Girard. Une extension de l'interprétation de Gödel à l'analyse, et son application à l'élimination de coupures dans l'analyse et la théorie des types. In J. E. Fenstad, editor, *Proceedings of the 2nd Scandinavian Logic Symposium*, pages 63–92. North-Holland, 1971.

[GJSB05] James Gosling, Bill Joy, Guy Steele, and Gilad Bracha. *The Java Language Specification (3rd ed.)*. Java series. Addison-Wesley, 2005.

[GLPS05a] V. Gapeyev, M. Y. Levin, B. C. Pierce, and A. Schmitt. The Xtatic experience. In *PLAN-X*, 2005.

[GLPS05b] V. Gapeyev, M.Y. Levin, B.C. Pierce, and A. Schmitt. The Xtatic compiler and runtime system, 2005. `http://www.cis.upenn.edu/~bcpierce/xtatic`.

[GLS07] P. Genevès, N. Layaïda, and A. Schmitt. Efficient static analysis of XML paths and types. In *PLDI '07*. ACM Press, 2007.

[GTT99] Rémi Gilleron, Sophie Tison, and Marc Tommasi. Set constraints and automata. *Information and Computation*, 149(1):1–41, 1999.

[HFC05] H. Hosoya, A. Frisch, and G. Castagna. Parametric polymorphism for XML. In *POPL '05, 32nd ACM Symposium on Principles of Programming Languages*. ACM Press, 2005.

[HFC09] H. Hosoya, A. Frisch, and G. Castagna. Parametric polymorphism for XML. *ACM TOPLAS*, 32(1):1–56, 2009.

[Hig52] Graham Higman. Ordering by divisibility in abstract algebras. *Proceedings of the London Mathematical Society*, 3(2(7)):326–336, 1952.

[Hos01] H. Hosoya. *Regular Expression Types for XML*. PhD thesis, The University of Tokyo, 2001.

[HP01] H. Hosoya and B.C. Pierce. Regular expression pattern matching for XML. In *POPL '01, 25th ACM Symposium on Principles of Programming Languages*, 2001.

[HP03] H. Hosoya and B.C. Pierce. XDuce: A statically typed XML processing language. *ACM Trans. Internet Techn.*, 3(2):117–148, 2003.

[HRS+05] Matthew Harren, Mukund Raghavachari, Oded Shmueli, Michael G. Burke, Rajesh Bordawekar, Igor Pechtchanski, and Vivek Sarkar. XJ: facilitating XML processing in Java. In *Proceedings of the 14th international conference on World Wide Web*, WWW '05, pages 278–287. ACM, 2005.

[HTM99] W3C: HTML 4.01 Specification. `http://www.w3.org/TR/1999/REC-html401-19991224/`, 1999.

[HWG03] Anders Hejlsberg, Scott Wiltamuth, and Peter Golde. *C♯ Language Specification*. Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 2003.

[Jim96] T. Jim. What are principal typings and what are they good for? In Hans-Juergen Boehm and Guy L. Steele Jr., editors, *POPL '96*, pages 42–53. ACM Press, 1996.

[Jim00] T. Jim. A polar type system. In *ICALP Satellite Workshops*, pages 323–338, 2000.

[KM06] C. Kirkegaard and A. Møller. XAct - XML transformations in Java. In *Programming Language Technologies for XML (PLAN-X)*, 2006.

[KMS00] Nils Klarlund, Anders Møller, and Michael I. Schwartzbach. DSD: A Schema Language for XML. In *In ACM SIGSOFT Workshop on Formal Methods in Software Practice*, 2000.

[KW99] A. J. Kfoury and J. B. Wells. Principality and decidable type inference for finite-rank intersection types. In *POPL '99*, pages 161–174. ACM, 1999.

[Lei83] Daniel Leivant. Polymorphic type inference. In *Proceedings of the 10th ACM SIGACT-SIGPLAN symposium on Principles of programming languages*, POPL '83, pages 88–98, 1983.

[Ler90] X. Leroy. The ZINC experiment: an economical implementation of the ML language. Technical report 117, INRIA, 1990.

[LR07] L. Liquori and S. Ronchi Della Rocca. Intersection-types à la Church. *Inf. Comput.*, 205(9):1371–1386, 2007.

[LS04] K. Zhuo Ming Lu and M. Sulzmann. An implementation of subtyping among regular expression types. In *Proc. of APLAS'04*, volume 3302 of *LNCS*, pages 57–73. Springer, 2004.

[Mil78] R. Milner. A theory of type polymorphism in programming. *Journal of Computer and System Sciences*, 17:348–375, 1978.

[MLM01] M. Murata, D. Lee, and M. Mani. Taxonomy of XML schema languages using formal language theory. In *Extreme Markup Languages*, 2001.

[MTHM97] R. Milner, M. Tofte, R. Harper, and D. MacQueen. *The Definition of Standard ML (Revised)*. MIT Press, 1997.

[Oca] Ocaml. `http://caml.inria.fr/ocaml/`.

[OSW97] Martin Odersky, Martin Sulzmann, and Martin Wehr. Type inference with constrained types. In *Fourth International Workshop on Foundations of Object-Oriented Programming (FOOL)*, 1997.

[Pie02] B.C. Pierce. *Types and Programming Languages*. MIT Press, 2002.

[PR05] François Pottier and Didier Rémy. The essence of ML type inference. In Benjamin C. Pierce, editor, *Advanced Topics in Types and Programming Languages*, chapter 10, pages 389–489. MIT Press, 2005.

[PT00] Benjamin C. Pierce and David N. Turner. Local type inference. *ACM Trans. Program. Lang. Syst.*, 22(1):1–44, January 2000.

[Pys76] Arthur Pyster. A language construct for "dovetailing". *SIGACT News*, 8(1):38–40, Jan 1976.

[Rey74] John C. Reynolds. Towards a theory of type structure. In B. Robinet, editor, *Programming Symposium*, volume 19 of *Lecture Notes in Computer Science*, pages 408–425, Berlin, 1974. Springer-Verlag.

[Rey83] J.C. Reynolds. Types, abstraction and parametric polymorphism. In *Information Processing*, pages 513–523. Elsevier, 1983.

[Rey84] J.C. Reynolds. Polymorphism is not set-theoretic. In *Semantics of Data Types*, volume 173 of *LNCS*, pages 145–156. Springer, 1984.

[Rey96] J.C. Reynolds. Design of the programming language Forsythe. Technical Report CMU-CS-96-146, Carnegie Mellon University, 1996.

[Rey03] J.C. Reynolds. What do types mean?: from intrinsic to extrinsic semantics. In *Programming methodology*. Springer, 2003.

[Ron02] S. Ronchi Della Rocca. Intersection typed lambda-calculus. *Electr. Notes Theor. Comput. Sci.*, 70(1):163–181, 2002.

[Sim92] Simon L. Peyton Jones and Cordy Hall and Kevin Hammond and Jones Cordy and Hall Kevin and Will Partain and Phil Wadler. The Glasgow Haskell compiler: a technical overview, 1992.

[SOAP] W3C. *SOAP Version 1.2*. `http://www.w3.org/TR/soap`.

[Ste94] Kjartan Stefánsson. Systems of set constraints with negative constraints are nexptime-complete. In *Proceedings of Ninth Annual IEEE Symposium on Logic in Computer Science*, pages 137–141. IEEE Computer Society, 1994.

[SZL07] Martin Sulzmann, Kenny Zhuo, and Ming Lu. XHaskell - Adding Regular Expression Types to Haskell. In *IFL*, volume 5083 of *Lecture Notes in Computer Science*, pages 75–92. Springer, 2007.

[Tar55] Alfred Tarski. A lattice-theoretical fixpoint theorem and its applications. *Pacific Journal of Mathematics*, 5(2):285–309, 1955.

[Vou06] J. Vouillon. Polymorphic regular tree types and patterns. In *POPL '06*, pages 103–114, 2006.

[Wad89] P. Wadler. Theorems for free! In *Functional Programming Languages and Computer Architecture*, pages 347–359. ACM Press, 1989.

[WDMT02] J.B. Wells, A. Dimock, R. Muller, and F.A. Turbak. A calculus with polymorphic and polyvariant flow types. *J. Funct. Program.*, 12(3):183–227, 2002.

[WH02] J.B. Wells and C. Haack. Branching types. In *ESOP '02*, volume 2305 of *LNCS*, pages 115–132. Springer, 2002.

[WR99] C. Wallace and C. Runciman. Haskell and XML: Generic combinators or type based translation? In *ICFP '99*, pages 148–159, 1999.

[XCC03] H. Xi, C. Chen, and G. Chen. Guarded recursive datatype constructors. In *POPL '03*, pages 224–235. ACM Press, 2003.

[XML] W3C: XML Version 1.0 (Fourth Edition). `http://www.w3.org/TR/REC-xml/`.

[XSc] W3C: XML Schema. `http://www.w3.org/XML/Schema`.