HAL open science

Combining union, intersection and dependent types in an explicitely typed lambda-calculus

Claude Stolze

To cite this version:

Claude Stolze. Combining union, intersection and dependent types in an explicitely typed lambda-calculus. Logic [math.LO]. COMUE Université Côte d'Azur (2015 - 2019), 2019. English. NNT : 2019AZUR4104 . tel-02406953v2

HAL Id: tel-02406953

https://hal.science/tel-02406953v2

Submitted on 26 Jun 2020

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 CÔTE D'AZUR

ÉCOLE DOCTORALE
SCIENCES ET TECHNOLOGIES DE L'INFORMATION ET DE LA COMMUNICATION

THÈSE DE DOCTORAT

Types union, intersection, et dépendants dans le lambda-calcul explicitement typé

Claude STOLZE

INRIA Sophia-Antipolis Méditerranée

Présentée en vue de l'obtention du grade de docteur en Informatique de l'Université Côte d'Azur.

Dirigée par : Luigi Liquori

Soutenue le : 16 décembre 2019

Devant le jury, composé de :
Giuseppe Castagna
Ugo de'Liguoro
Silvia Ghilezan
Furio Honsell
Delia Kesner
Bruno Martin
Jakob Rehof
Luigi Liquori

Types union, intersection, et dépendants dans le lambda-calcul explicitement typé

Jury

Président du jury

Bruno Martin, Professeur, Université Côte d'Azur

Rapporteurs

Silvia Ghilezan, Professeur, University of Novi Sad, Serbie
Delia Kesner, Professeur, Université de Paris

Examinateurs

Giuseppe Castagna, Directeur de recherche, CNRS, Université de Paris
Ugo de'Liguoro, Professeur, Università degli Studi di Torino, Italie
Furio Honsell, Professeur, Università degli Studi di Udine, Italie
Jakob Rehof, Professeur, Technische Universität Dortmund, Allemagne

Encadrant

Luigi Liquori, Directeur de recherche, INRIA Sophia-Antipolis Méditerranée

Types union, intersection, et dépendants dans le lambda-calcul explicitement typé

Résumé :

Le sujet de cette thèse est sur le lambda-calcul décoré avec des types, communément appelé « lambda-calcul typé à la Church ». Nous étudions des versions de ce lambda-calcul muni de types intersections, tels que ceux décrits dans le livre « Lambda-calculus with types » de Barendregt, Dekkers et Statman ; les types unions, qui ont été introduits par Plotkin, MacQueen et Sethi ; et les types dépendants, tels qu'ils ont été décrits par Plotkin, Harper et Honsell lorsqu'ils ont introduit le Logical Framework d'Edinbourgh LF. Les types intersections et unions sont un moyen d'exprimer du polymorphisme ad hoc et sont une alternative au polymorphisme paramétrique de Girard. Les types dépendants ont été introduits pour formaliser la logique intuitionniste avec la correspondance de Curry-Howard. Le système de types obtenu peut être enrichi avec une relation de soutypage décidable. La combinaison de ces trois disciplines de type donne lieu à une famille de calculs qui peuvent être paramétrés et classifiés. Nous appelons le système générique le Delta-calcul. Nous discutons ensuite des décisions de conception qui nous ont amené à la formulation de ces calculs, nous étudions leur métathéorie, et nous présentons divers exemples d'applications avant de présenter une implémentation logicielle du Delta-calcul, avec une description des algorithmes de vérification de type, de raffinement, de soutypage, d'évaluation, ainsi que de l'interface en ligne de commande. Ce travail de recherche peut être vu comme un petit pas franchi dans la direction d'une théorie des types alternative pour définir du polymorphisme dans les langages de programmation et dans les assistants de preuve interactifs.

Mots clés : lambda-calcul, théorie des types, correspondance de Curry-Howard.

Combining union, intersection and dependent types in an explicitely typed lambda-calculus

Abstract:

The subject of this thesis is about lambda-calculus decorated with types, usually called "Church-style typed lambda-calculus". We study this lambda-calculus enhanced with Intersection types, as described by Barendregt, Dekkers and Statman in the book "Lambda-calculus with Types"; Union types, as introduced by Plotkin, MacQueen and Sethi; and Dependent types, as described by Plotkin, Harper and Honsell when they introduced the Edinburgh Logical Framework LF. Intersection and union types are a way to express ad hoc polymorphism and are an alternative to the parametric polymorphism of Girard. Dependent types were introduced as a way to formalize intuitionistic logic using the "proofs-as-lambda-terms / formulae-as-types" Curry-Howard principle. The resulting type system can be enriched with a decidable subtyping relation. Combining these three type disciplines gives rise to a family of calculi that can be parametrized and classified: we call the resulting system the Delta-calculus. We then discuss the design decisions which have led us to the formulation of these calculi, study their metatheory, and provide various examples of applications; and we finally present a software implementation of the Delta-calculus, with a description of the type checker, the refinement algorithm, the subtyping algorithm, the evaluation algorithm and the command-line interface. This work can be understood as a little step toward an alternative type theory to defining polymorphic programming languages and interactive proof assistants.

Keywords: lambda-calculus, type theory, Curry-Howard correspondence.

Remerciements – Acknowledgements

Je ne peux commencer cette thèse sans remercier mon directeur, Luigi Liquori, pour tout le soutien et tous les conseils qu'il m'a prodigués au cours de ces trois années de travail ensemble. Merci en particulier pour la confiance que tu m'as accordé. Tu m'as appris l'exigeant métier de chercheur, notamment qu'un article n'a fini d'être écrit que lorsque la deadline est passée, et que les détails mathématiques et l'esthétique ont une importance cruciale dans la science. J'aimerais aussi remercier ma mère pour son soutien à distance de l'autre bout de la France, et qui me réconfortait quand j'avais le mal du pays. Je voudrais remercier Enrico Tassi qui m'a détaillé les entrailles du code de Coq. Et sans oublier mes camarades Siargey, Owen, Sophie, Cécile, Damien, Carsten, Giovanni, et tant d'autres.

I would also like to thank my jury for accepting to review my work. I am particularly thankful to Delia Kesner and Silvia Ghilezan, who accepted to be my referees. Many thanks to Bruno Martin, Giuseppe Castagna, Ugo de'Liguoro, Furio Honsell, and Jakob Rehof, for showing interest in my work and for accepting to participate in this jury.

Contents

1 Introduction **1**
- 1.1 Prolegomenon . . . 1
- 1.2 Contributions . . . 3
- 1.3 Related works . . . 5

2 A typed calculus with intersection types **9**
- 2.1 Syntax, Reduction and Types . . . 10
  - 2.1.1 The Δ-calculi . . . 11
  - 2.1.2 The Δ-chair . . . 14
- 2.2 Examples . . . 15
  - 2.2.1 On synchronization and subject reduction . . . 18
- 2.3 Metatheory of $\Delta^{\mathcal{T}}_{\mathcal{R}}$ . . . 19
  - 2.3.1 General properties . . . 19
  - 2.3.2 Synchronous reduction . . . 25
  - 2.3.3 Strong normalization of the generic Δ-calculus . . . 29
- 2.4 Church-style *vs.* Curry-style λ-calculus . . . 31
  - 2.4.1 Relation between type assignment systems $\lambda^{\mathcal{T}}_{\cap}$ and typed systems $\Delta^{\mathcal{T}}_{\mathcal{R}}$ 31
  - 2.4.2 Subtyping and explicit coercions . . . 34

3 Adding union types **39**
- 3.1 Syntax and semantics of Δ^{BDdL} and $\lambda@^{\text{BDdL}}$ . . . 40
- 3.2 Metatheory of Δ^{BDdL} and $\lambda@^{\text{BDdL}}$ . . . 43

4 On Mints' realizability **45**
- 4.1 Presentation of $\mathsf{NJ}(\beta)$ . . . 46
- 4.2 Soundness of $\mathsf{NJ}(\beta)$ . . . 47
- 4.3 Completeness of $\mathsf{NJ}(\beta)$ . . . 49
  - 4.3.1 Failure of completeness of $\mathsf{NJ}(\beta)$ without subtyping . . . 49
  - 4.3.2 Counter-example . . . 50

5 Subtyping algorithm **51**
- 5.1 The algorithm, shortly explained . . . 51
  - 5.1.1 Soundness and correctness of the algorithm . . . 54
- 5.2 Coq implementation of the theory Ξ . . . 55
  - 5.2.1 Definition of normal forms . . . 57
  - 5.2.2 Filters and ideals . . . 59

5.2.3 Induction scheme for filters and ideals . . . 60
5.2.4 Properties of filters and ideals . . . 62
5.3 Coq implementation of the subtyping algorithm . . . 63
5.4 Extracting the subtyping algorithm in OCaml . . . 69
5.5 The preorder tactic . . . 70
5.5.1 Denotation . . . 71
5.5.2 Implementation of the heuristic function . . . 72
5.5.3 Reification . . . 74

6 Dependent types **77**
6.1 The Δ-framework: LF with proof-functional operators . . . 80
6.2 Relating LF_Δ to λ^{BDdL} . . . 84
6.2.1 Typed derivation of Pierce's example . . . 85
6.2.2 LF_Δ metatheory . . . 86
6.3 Minimal relevant implications and type inclusion . . . 89
6.4 Pure Type System presentation of LF_Δ . . . 91
6.5 Future Work . . . 92

7 Implementation of the theorem prover Bull **95**
7.1 de Bruijn indices in Bull . . . 96
7.2 Syntax of terms . . . 97
7.2.1 Concrete syntax . . . 98
7.2.2 Implementation of the syntax . . . 99
7.2.3 Environments . . . 100
7.2.4 Suspended substitution . . . 101
7.3 The evaluator of Bull . . . 102
7.3.1 Reduction rules . . . 102
7.3.2 Implementation . . . 102
7.4 The subtyping algorithm of Bull . . . 104
7.5 The unification algorithm of Bull . . . 105
7.6 The refinement algorithm of Bull . . . 105
7.7 The Read-Eval-Print-Loop of Bull . . . 108
7.8 Future work . . . 109

8 Examples in Bull **115**
8.1 Encodings in LF_Δ . . . 115
8.1.1 Pierce's code . . . 116
8.1.2 Hereditary Harrop formulæ . . . 116
8.1.3 Natural deductions in normal form . . . 118
8.2 Encodings in LF . . . 121

List of Figures

1.1 Pierce's code 7

2.1 Minimal type theory $\leqslant_{\min}$, axioms and rule schemes (from Figure 13.2 and 13.3 of [12]) 10
2.2 Generic intersection type assignment system $\lambda_{\cap}^{\mathcal{T}}$ (from Figure 13.8 of [12]) . 10
2.3 Type theories $\lambda_{\cap}^{\mathrm{CD}}$, $\lambda_{\cap}^{\mathrm{CDS}}$, $\lambda_{\cap}^{\mathrm{CDV}}$, and $\lambda_{\cap}^{\mathrm{BCD}}$. The "ref." column refers to the original articles these theories come from 11
2.4 Generic Δ-calculus $\Delta_{\mathcal{R}}^{\mathcal{T}}$ 13
2.5 Generic intersection typed system $\lambda@_{\mathcal{R}}^{\mathcal{T}}$ 14
2.6 The Δ-chair 15
2.7 On the left: source systems. On the right: target systems without the $(\leqslant_{\mathcal{T}})$ rule 37

3.1 Intersection and union type assignment system λ^{BDdL} [7] 40
3.2 Typed calculus $\lambda@^{\mathrm{BDdL}}$ [40] 41
3.3 Δ-calculus Δ^{BDdL} 42

4.1 The logic $\mathsf{NJ}(\beta)$ 46

5.1 The type theory Ξ of [7] 52

6.1 The type assignment system λ^{BDdL} of [7] and the type theory Ξ 79
6.2 The syntax of the Δ-framework 80
6.3 The extended essence function 81
6.4 The reduction semantics 81
6.5 Valid signatures and contexts 82
6.6 Valid kinds and families 83
6.7 The type rules for valid objects 83
6.8 Encoding of Ω 84
6.9 Pierce's one-step reduction counter-example 84
6.10 The forgetful mappings $\|-\|$ and $|-|$ 87
6.11 The coercion function 90
6.12 Pure Type System presentation of the Δ-framework (signature and context) 91
6.13 Pure Type System presentation of the Δ-framework (terms) 92

7.1 Structural rules of the unification algorithm 106
7.2 Rules for $\overset{\Uparrow}{\rightsquigarrow}$ (1st part) 108

7.3 Rules for $\overset{\Uparrow}{\rightsquigarrow}$ (2nd part) 111
7.4 Rules for $\overset{\mathcal{F}}{\rightsquigarrow}$ 112
7.5 Rules for $\overset{\Downarrow}{\rightsquigarrow}$ 112
7.6 Rules for $\overset{\mathcal{E}^{\Uparrow}}{\rightsquigarrow}$ 113
7.7 Rules for $\overset{\mathcal{E}^{\Downarrow}}{\rightsquigarrow}$ 114

8.1 The LF_{Δ} encoding of Hereditary Harrop Formulæ 117
8.2 LF encoding of the Pierce's code (from Figure 1.1) 125

Chapter 1

Introduction

"That logic has advanced in this sure course, even from the earliest times, is apparent from the fact that, since Aristotle, it has been unable to advance a single step and, thus, to all appearance, has reached its completion."

Immanuel Kant, Preface to the second edition of *The Critique of Pure Reason*, 1787

1.1 Prolegomenon

When George Boole wrote *Mathematical Analysis of Logic* in 1847 [19], he modestly aimed at an algebraic clarification of Aristotelian logic, and did not immediately realize his work was the beginning of a deep change in the study of mathematics which would later trigger the foundational crisis of mathematics.

In 1903, Bertrand Russell [85], in *The Principles of Mathematics*[1], opened a Pandora's box when he considered *"predicates which are not predicable of themselves"*[2]. As it is widely known nowadays, Russell's contradiction – a modern version of the liar's paradox – consists of defining a predicate $P(x) \stackrel{def}{=} \neg x(x)$, and deducing both $P(P)$ and $\neg P(P)$. In order to circumvent this contradiction, Russell introduced, in the Appendix B of the same book, the *Doctrine of Types*:

> *"The doctrine of types is here put forward tentatively, as affording a possible solution of the contradiction [...]. Every propositional function $\phi(x)$ – so it is contended – has, in addition to its range of truth, a range of significance, i.e. a range in which x must lie if $\phi(x)$ is to be a proposition at all, whether true or false. This is the first point in the theory of types; the second point is that ranges of significance form types, i.e. if x belongs to the range of significance of $\phi(x)$, then there is a class of objects, the type of x, all of which must also belong to the range of significance of $\phi(x)$."*

This general idea set the foundation of the (many) theories of types, which were widely developed during the course of the twentieth century. In 1934, Haskell Curry was *"concerned with statements [...] of the form "f is a function on X to Y""* [31]. Haskell

[1] Not to be confused with *Principia Mathematica*, which he wrote with Alfred Whitehead from 1910 to 1913.

[2] In Section 78, simply called *"The contradiction"*.

Curry, and later William Howard, discovered that rules determining that a function has some type where very similar to logical rules determining that a proof shows the validity of some proposition. The proofs-as-functions/propositions-as-types principle is now known as the Curry-Howard correspondence [58].

In short, types are a tool used to give a notion of a well-formed expression:

- we can use types to describe well-formed propositions and proofs;
- we can use types to describe well-formed computable functions.

Among the most impactful developments from the previous century of type theory as a foundation for mathematics, we can cite Automath by N. G. de Bruijn [34], the first theorem prover, whose development started in the sixties, the intuitionistic type theory of Per Martin-Löf [69], and finally the Calculus of (Inductive) Constructions by Thierry Coquand, Gérard Huet [30], Frank Pfenning, and Christine Paulin-Mohring [76], which is the theoretical foundation of the Coq theorem prover [36].

The model of computation which is the most associated with type theory is the λ-calculus, this language developed by Alonzo Church in the thirties. The pure λ-calculus has two basic operations:

1. the first one is application, simply noted with a space: the expression MN denotes the function M applied to its argument N;
2. the second one is λ-abstraction, noted with the binder λ: the expression $\lambda x.M$ denotes a function taking x as an argument and returning the expression M (possibly containing occurrences of x).

We can see λ-abstraction as a function constructor, and application as a function eliminator. Combining λ-abstraction and application gives two computational rules:

(β) $(\lambda x.M)\,N$ reduces to $M[N/x]$, *i.e.* all the free occurrences of x in M are substituted by N. This is the elimination of the construction of a function, and it is called a β-reduction;

(η) $\lambda x.M\,x$ reduces to M if x is not free in M. This is the construction of the elimination of a function, and it is called an η-reduction.

We can also assign type to terms. Intuitively:

- in the λ-calculus *à la* Curry: we assign a type to a pure λ-term. For the simply-typed λ-calculus, we get the following rules for λ-abstraction and application:

$$\frac{\Gamma, x{:}\sigma \vdash M : \tau}{\Gamma \vdash \lambda x.M : \sigma \to \tau}\ (\to I) \qquad \frac{\Gamma \vdash M : \sigma \to \tau \quad \Gamma \vdash N : \sigma}{\Gamma \vdash M\,N : \tau}\ (\to E)$$

  As you can see, stating that M has $\sigma \to \tau$ intuitively means that M is a function on σ to τ;

- in the λ-calculus *à la* Church: we decorate a λ-term with types, typically the λ-abstraction becomes $\lambda x{:}\sigma.M$, where we explicitly state that x has type σ. For the simply-typed λ-calculus, we get the following rules for λ-abstraction and application:

$$\frac{\Gamma, x{:}\sigma \vdash M : \tau}{\Gamma \vdash \lambda x{:}\sigma.M : \sigma \to \tau}\ (\to I) \qquad \frac{\Gamma \vdash M : \sigma \to \tau \quad \Gamma \vdash N : \sigma}{\Gamma \vdash M\,N : \tau}\ (\to E)$$

Among the many type systems for the λ-calculi, intersection is an interesting connective (see Henk Barendregt, Wil Dekkers, and Richard Statman [12] : intuitively, if some term M has both type σ and type τ, then we say it is polymorphic and it has type $\sigma \cap \tau$, as the derivation rules show:

$$\frac{\Gamma \vdash M : \sigma \quad \Gamma \vdash M : \tau}{\Gamma \vdash M : \sigma \cap \tau}\ (\cap I) \qquad \frac{\Gamma \vdash M : \sigma \cap \tau}{\Gamma \vdash M : \sigma}\ (\cap E_1) \qquad \frac{\Gamma \vdash M : \sigma \cap \tau}{\Gamma \vdash M : \tau}\ (\cap E_2)$$

The dual connective of intersection, namely union (see David MacQueen, Gordon Plotkin, and Ravi Sethi [68]), states that if M has type σ or τ, then it has type $\sigma \cup \tau$, as the derivation rules show:

$$\frac{\Gamma \vdash M : \sigma}{\Gamma \vdash M : \sigma \cup \tau}\ (\cup I_1) \qquad \frac{\Gamma \vdash M : \tau}{\Gamma \vdash M : \sigma \cup \tau}\ (\cup I_2) \qquad \frac{\begin{array}{c}\Gamma, x{:}\sigma \vdash M : \rho \\ \Gamma, x{:}\tau \vdash M : \rho \quad \Gamma \vdash N : \sigma \cup \tau\end{array}}{\Gamma \vdash M[N/x] : \rho}\ (\cup E)$$

1.2 Contributions

The subject of this thesis is to study intersection and union types in λ-calculi *à la* Church, with two goals:

1. finding a λ-calculus *à la* Church with intersection and union types which is faithful to the corresponding pure λ-calculi *à la* Curry;

2. conceiving and implementing a logical framework based on intersection and union types.

Building a λ-calculus *à la* Church with intersection and union types

Designing a λ-calculus *à la* Church with intersection and union types is challenging. The usual approach of simply adding types to binders does not work, as shown here:

$$\frac{\dfrac{\dfrac{}{x{:}\sigma \vdash x{:}\sigma}\ (Var)}{\vdash \lambda x{:}\sigma.x{:}\sigma \to \sigma}\ (\to I) \qquad \dfrac{\dfrac{}{x{:}\tau \vdash x{:}\tau}\ (Var)}{\vdash \lambda x{:}\tau.x{:}\tau \to \tau}\ (\to I)}{\vdash \lambda x{:}???.x{:}(\sigma \to \sigma) \cap (\tau \to \tau)}\ (\cap I)$$

Same difficulties can be found with union types. In this thesis we propose a solution to this challenge by designing a λ-calculus *à la* Church, called the Δ-calculus (see Chapters 2 and 3). In a nutshell: each term of the Δ-calculus has a corresponding pure λ-term, called its *essence*. Intersection types are constructed with *strong pairs*, a cartesian pair which ensures both its components share the same essence. Dually, union types are constructed with *strong sums*, whose original feature is that it ensures both cases of its elimination share the same essence.

Building a logical framework based on intersection and union types

We have designed a logical framework based on proof-functional logic (see Chapter 6), and using dependent types as in the Edinburgh Logical Framework [52]. In a nutshell:

- *Strong disjunction* is a proof-functional connective that can be interpreted as the union type $\cup$ [39, 88]: it contrasts with the intuitionistic connective $\vee$. As Pottinger [80] did for intersection, we could say that asserting $(A \cup B) \supset C$ is to assert that one has a

reason for $(A \cup B) \supset C$, which is also a reason to assert $A \supset C$ and $B \supset C$. A simple example of a logical theorem involving intuitionistic disjunction which does not hold for strong disjunction is $((A \supset B) \cup B) \supset A \supset B$. Otherwise there would exist a term which behaves both as I and as K.

- *Strong (relevant) implication* is yet another proof-functional connective that was interpreted in [8] as a relevant arrow type $\rightarrow_r$. As explained in [8], it can be viewed as a special case of implication whose related function space is the simplest one, namely the one containing only the identity function. Because the operators $\supset$ and $\rightarrow_r$ differ, $A \rightarrow_r B \rightarrow_r A$ is not derivable.

- *Dependent types*, as introduced in the Edinburgh Logical Framework [52] by Robert Harper, Furio Honsell, and Gordon Plotkin, allows considering proofs as first-class citizens, albeit differently, with respect to proof-functional logics. The interaction of both dependent and proof-functional operators is intriguing. Their combination opens up new possibilities of formal reasoning on proof-theoretic semantics.

We have also implemented a prototype of an interactive theorem prover based on this logical framework, called Bull. For instance, the following code snippet shows the implementation of a polymorphic identity on `A` and `B`, using a strong pair, which ensures that `id1` and `id2` have the same essence.

```
Definition poly_id : (A -> A) & (B -> B) :=
  let id1 x := x in
  let id2 x := x in
  < id1, id2 >.
```

Organization of this thesis

This thesis is organized as follows:

- Chapter 2 presents a generic Δ-calculus, *i.e.* a generic typed λ-calculus with intersection types. We study some of its instances and their properties, as well as their relation with standard λ-calculus with intersection types;

- Chapter 3 extends the previous Δ-calculus with union types, and defines a typed λ-calculus $\lambda@^{\mathrm{BDdL}}$, and recalls the original λ-calculus [7] these new systems are inspired from;

- Chapter 4 sketches the logical interpretation of intersection and union types. More precisely, we define an interpretation of typing judgments $M : \sigma$ into a first-order logical proposition $r_\sigma[M]$ in the logic $\mathsf{NJ}(\beta)$;

- Chapter 5 defines a subtyping algorithm for intersection and union types. This algorithm is then fully implemented and certified in Coq. We detail the Coq implementation;

- Chapter 6 extends the Δ-calculus by adding dependent types, in the style of the Edinburgh Logical Framework (LF). We call the resulting system the Δ-framework LF_Δ, and we prove its metatheoretical properties;

- Chapter 7 presents an OCaml implementation of the Δ-framework into an interactive theorem prover, called Bull. The technical details of the implementation (syntax, semantics, typechecking, and Read-Eval-Print loop) are described;

- Chapter 8 presents some examples of encodings in the Δ-framework, as well as their implementation in Bull.

Publications

During the course of my thesis, I published the following conference papers:

– Luigi Liquori and Claude Stolze. The Delta-calculus: Syntax and Types. In 4th International Conference on Formal Structures for Computation and Deduction, FSCD 2019, pages 28:1–28:20. Schloss Dagstuhl-Leibniz-Zentrum fuer Informatik, June 2019. [66]

– Furio Honsell, Luigi Liquori, Ivan Scagnetto, and Claude Stolze. The Delta-framework. In 38th IARCS Annual Conference on Foundations of Software Technology and Theoretical Computer Science, FSTTCS 2018, pages 37:1–37:21. Schloss Dagstuhl-Leibniz-Zentrum fuer Informatik, December 2018. [57]

– Luigi Liquori and Claude Stolze. A Decidable Subtyping Logic for Intersection and Union Types. In 2nd International Conference on Topics in Theoretical Computer Science, TTCS 2017, pages 74–90. Springer, September 2017. [65]

– Claude Stolze, Luigi Liquori, Furio Honsell, and Ivan Scagnetto. Towards a Logical Framework with Intersection and Union Types. In 11th International Workshop on Logical Frameworks and Meta-languages, LFMTP 2017, pages 1–9. ACM, September 2017. [88]

– Daniel J. Dougherty, Ugo de'Liguoro, Luigi Liquori, and Claude Stolze. A Realizability Interpretation for Intersection and Union Types. In Programming Languages and Systems – 14th Asian Symposium, APLAS 2016, pages 187–205. Springer, October 2016. [39]

Software

During the course of my thesis, I developed a prototype of an interactive theorem prover implementing the Δ-framework, called Bull [87].

1.3 Related works

In order to foster the imagination of the reader about the topic we will study in this thesis, we shortly present works related to our field of interest.

λ-calculi with intersection types *à la* Curry

Intersection type theories $\mathcal{T}$ were first introduced as a form of *ad hoc* polymorphism in (pure) λ-calculi *à la* Curry. The paper by Barendregt, Coppo, and Dezani [11] is a classic reference, while [12] is a definitive reference.

Intersection type assignment systems $\lambda_{\cap}^{\mathcal{T}}$ have been well-known in the literature for almost 40 years for many reasons: among them, characterization of strongly normalizing λ-terms [12], λ-models [4], automatic type inference [60], type inhabitation [92, 82], type unification [43]. As intersection had its classical development for type assignment systems,

many papers tried to find an explicitly typed λ-calculus *à la* Church corresponding to the original intersection type assignment systems *à la* Curry. The programming language Forsythe, by Reynolds [83], is probably the first reference, while Pierce's Ph.D. thesis [77] combines also unions, intersections and bounded polymorphism. Wells *et al.* [95] use intersection types as a foundation for typed intermediate languages for optimizing compilers for higher-order polymorphic programming languages; implementations of typed programming language featuring intersection (and union) types can be found in ℂDuce [49, 50], SML-CIDRE [33], and in StardustML [44, 45].

Intersection and union type disciplines started to be investigated in a explicitly typed programming language settings *à la* Church, much later by Reynolds and Pierce [83, 77], Wells *et al.* [95, 96], Liquori *et al.* [64, 40], Frisch *et al.* [50] and Dunfield [45].

λ-calculi with intersection types *à la* Church

Several calculi *à la* Church appeared in the literature: they capture the power of intersection types; we briefly review them.

The Forsythe programming language by Reynolds [83] annotates a λ-abstraction with types as in $\lambda x{:}\sigma_1|\cdots|\sigma_n.M$. However, we cannot type a typed term, whose type erasure is the combinator $\mathsf{K} \equiv \lambda x.\lambda y.x$, with the type $(\sigma \to \sigma \to \sigma) \cap (\tau \to \tau \to \tau)$.

Pierce [78] improves Forsythe by using a **for** construct to build *ad hoc* polymorphic typing, as in $\mathbf{for}\, \alpha \in \{\sigma, \tau\}.\lambda x{:}\alpha, \lambda y{:}\alpha.x$. However, we cannot type a typed term, whose type erasure is $\lambda x.\lambda y.\lambda z.(x\, y, x\, z)$, with the type [96]:

$$((\sigma \to \rho) \cap (\tau \to \rho') \to \sigma \to \tau \to \rho \times \rho') \cap ((\sigma \to \sigma) \cap (\sigma \to \sigma) \to \sigma \to \sigma \to \sigma \times \sigma)$$

Freeman and Pfenning [48] introduced refinement types, that is types that allow *ad hoc* polymorphism for ML constructors. Intuitively, refinement types can be seen as subtypes of a standard type: the user first defines a type and then the refinement types of this type. The main motivation for these refinement types is to allow non-exhaustive pattern matching, which becomes exhaustive for a given refinement of the type of the argument. As an example, we can define a type `boolexp` for boolean expressions, with constructors `True`, `And`, `Not` and `Var`, and a refinement type `ground` for boolean expressions without variables, with the same constructors except `Var`: then, the constructor `True` has type $\texttt{boolexp} \cap \texttt{ground}$, the constructor `And` has type $(\texttt{boolexp} * \texttt{boolexp} \to \texttt{boolexp}) \cap (\texttt{ground} * \texttt{ground} \to \texttt{ground})$ and so on. However, intersection is meaningful only when using constructors.

Wells *et al.* [95] introduced λ^{CIL}, a typed intermediate λ-calculus for optimizing compilers for higher-order programming languages. The calculus features intersection, union and flow types, the latter being useful to optimize data representation. λ^{CIL} can faithfully encode an intersection type assignment derivation by introducing the concept of virtual tuple, *i.e.* a special kind of pair whose type erasure leads to exactly the same untyped λ-term. A parallel context and parallel substitution, similar to the notion of [63, 64], is defined to reduce expressions in parallel inside a virtual tuple. Subtyping is defined only on flow types and not on intersection types: this system can encode the $\lambda^{\mathrm{CD}}_{\cap}$ type assignment system.

Wells and Haak [96] introduced λ^{B}, a more compact typed calculus encoding of λ^{CIL}: in fact, by comparing Figure 1 and Figure 2 of [96] we can see that the set of typable terms with intersection types of λ^{CIL} and λ^{B} are the same. In that paper, virtual tuples are removed by introducing branching terms, typable with branching types, the latter

representing intersection type schemes. Two operations on types and terms are defined, namely expand, expanding the branching shape of type annotations when a term is substituted into a new context, and select, to choose the correct branch in terms and types. As there are no virtual tuples, reductions do not need to be done in parallel. As in [95], the $\lambda_{\cap}^{\mathrm{CD}}$ type assignment system can be encoded.

Frisch, Castagna, and Benzaken [50] designed a typed system with intersection, union, negation and recursive types. The authors inherit the usual problem of having a domain space $\mathcal{D}$ that contains all the terms and, at the same time, all the functions from $\mathcal{D}$ to $\mathcal{D}$. They prevent this by having an auxiliary domain space which is the disjoint union of $\mathcal{D}^2$ and $\mathcal{P}(\mathcal{D}^2)$. The authors interpret types as sets in a well-suited model where the set-inspired type constructs are interpreted as the corresponding to set-theoretical constructs. Moreover, the model manages higher-order functions in an elegant way. The subtyping relation is defined as a relation on the set-theoretical interpretation $[\![-]\!]$ of the types. For instance, the problem $\sigma \cap \tau \leqslant \sigma$ will be interpreted as $[\![\sigma]\!] \cap [\![\tau]\!] \subseteq [\![\sigma]\!]$, where $\cap$ becomes the set intersection operator, and the decision program actually decides whether $([\![\sigma]\!] \cap [\![\tau]\!]) \cap \overline{[\![\sigma]\!]}$ is the empty set.

Bono *et al.* [18] introduced a relevant and strict parallel term constructor to build inhabitants of intersections and a simple call-by-value parallel reduction strategy. An infinite number of constants $c^{\sigma \Rightarrow \tau}$ is applied to typed variables x^{σ} such that $c^{\sigma \Rightarrow \tau}\, x^{\sigma}$ is upcasted to type τ. It also uses a local renaming typing rule, which changes type decoration in λ-abstractions, as well as coercions. Term synchronicity in the tuples is guaranteed by the typing rules. The calculus uses van Bakel's strict version [93] of the $\mathcal{T}_{\mathrm{CD}}$ intersection type theory.

λ-calculi with intersection and union types

Union types were introduced as a dual of intersection by MacQueen, Plotkin, and Sethi [68]: Barbanera, Dezani, and de'Liguoro [7] is a definitive reference (see Figure 3.1); Frisch, Castagna, and Benzaken [50] designed a type system with intersection, union, negation, and recursive types whose semantics fits the intuitive behaviour of the corresponding set-theoretical constructs.

A classical example of the expressiveness of union types, due to Pierce [77], is shown in Figure 1.1. Without union types, the best information we can get for (Is_0 Test) in this example is a boolean type.

$$\begin{array}{rcl} \text{Test} & \stackrel{def}{=} & \text{if } b \text{ then } 1 \text{ else } -1 : Pos \cup Neg \\ \text{Is_0} & : & (Neg \to F) \cap (Zero \to T) \cap (Pos \to F) \\ (\text{Is_0 Test}) & : & F \end{array}$$

Figure 1.1: Pierce's code

Algorithm for subtyping intersection and union types

Intersection and union types have an intuitive notion of subtyping. For instance, a term M of type $\sigma \cap \tau$ has also type σ and type τ, therefore $\sigma \cap \tau \leqslant \sigma$ and $\sigma \cap \tau \leqslant \tau$. Hindley was the first to give a subtyping algorithm for intersection types [54]. There is a rich literature reducing the subtyping problem in presence of intersection and union to a set

constraint problem: good references are [32, 1, 46, 50]. For instance, Aiken and Wimmers [2] designed an algorithm whose input is a list of set constraints with unification variables, usual arrow types, intersection, complementation, and constructor types. Their algorithm first rewrites types in disjunctive normal form, then simplifies the constraints until it shows the system has no solution, or until it can safely unify the variables. Rewriting in disjunctive normal form makes this algorithm exponential in time and space in the worst case.

Logical interpretation of intersection and union types

Mints [71] presented a logical interpretation of strong conjunction using *realizers*: the logical predicate $r_{A\cap B}[M]$ is true if the pure λ-term M is a realizer (also read as "M is a method to assess $A \cap B$") for both the formulæ $r_A[M]$ and $r_B[M]$. Inspired by this, Barbanera and Martini tried to answer the question of realizing other proof-functional connectives, like strong implication, López-Escobar's strong equivalence, or Bruce, Di Cosmo, and Longo provable type isomorphism [20].

Pfenning work on *refinement types* [74] pioneered an extension of Edinburgh Logical Framework with subtyping and intersection types. Dezani-Ciancaglini, Ghilezan, and Venneri [37] investigated a Curry-Howard interpretation of intersection and union types (for Combinatory Logic): using the well-understood relation between combinatory logic and λ-calculus, they encode type-free λ-terms into suitable combinatory logic formulæ and then type them using intersection and union types.

Strong connectives arise naturally in investigating the propositions-as-types analogy for intersection and union type assignment systems. Proof-functional (or strong) logical connectives, introduced by Pottinger [80], take into account the shape of logical proofs, thus allowing for polymorphic features of proofs to be made explicit in formulæ. This differs from classical or intuitionistic connectives where the meaning of a compound formula is only dependent on the truth value or the provability of its subformulæ.

Pottinger was the first to consider the intersection $\cap$ as a proof-functional connective. He contrasted it to the intuitionistic connective $\wedge$ as follows: *"The intuitive meaning of $\cap$ can be explained by saying that to assert $A \cap B$ is to assert that one has a reason for asserting A which is also a reason for asserting B, while to assert $A \wedge B$ is to assert that one has a pair of reasons, the first of which is a reason for asserting A and the second of which is a reason for asserting B".*

A simple example of a logical theorem involving intuitionistic conjunction which does not hold for proof-functional conjunction is $(A \supset A) \wedge (A \supset B \supset A)$. Otherwise there would exist a term which behaves both as I and as K.

It is not immediate to extend the judgments-as-types Curry-Howard paradigm to logics supporting proof-functional connectives. These connectives need to compare the shapes of derivations and do not just take into account their provability, *i.e.* the inhabitation of the corresponding type. There are many other proposals to find a suitable logic to fit intersection types; among them we cite [94, 84, 72, 22, 18, 79], and previous papers by the author [39, 65, 88]. This is still an open question that I am currently investigating.

Chapter 2

A typed calculus with intersection types

In this chapter, we define and prove the main properties of the generic Δ-calculus, a generic intersection typed system for an explicitly typed λ-calculus *à la* Church enriched with strong pairs, projections, and type coercions.

This chapter is organized as follows: in Section 2.1, we present the system. We also give then instances of the generic Δ-calculus in a diagram called the Δ-chair. In Section 2.2, we show a number of typable examples in the systems presented in the Δ-chair: each example is provided with a corresponding type assignment derivation of its essence. The aims of this section is both to make the reader comfortable with the different intersection typed systems, and to give a first intuition of the correspondence between Church-style and Curry-style calculi. In Section 2.3, we prove the metatheory for all the systems in the Δ-chair: Church-Rosser, unicity of type, subject reduction, strong normalization, decidability of type checking and type reconstruction. In Section 2.4, we study the relations between intersection type assignment systems *à la* Curry and the corresponding intersection typed systems *à la* Church. We also show how to get rid of type coercions Δ^τ defining a translation function, denoted by $\|_\|$, inspired by the one of Tannen *et al.* [91].

The most original feature of the generic Δ-calculus is the concept of *strong pair*. A *strong pair* $\langle \Delta_1, \Delta_2 \rangle$ is a special kind of cartesian product such that the two parts of a pair satisfies a given relation $\mathcal{R}$ on their *essence*, that is $\wr \Delta_1 \wr \; \mathcal{R} \; \wr \Delta_2 \wr$. The *essence* $\wr \Delta \wr$ of a Δ-term is a pure λ-term obtained by erasing type decorations, projections and choosing one of the two elements inside a strong pair. As examples,

$$\begin{aligned}
\wr \langle \lambda x{:}\sigma \cap \tau.\mathsf{pr}_2\, x, \lambda x{:}\sigma \cap \tau.\mathsf{pr}_1\, x \rangle \wr &= \lambda x.x \\
\wr \lambda x{:}(\sigma \to \tau) \cap \sigma.(\mathsf{pr}_1\, x)(\mathsf{pr}_2\, x) \wr &= \lambda x.x\, x \\
\wr \lambda x{:}\sigma \cap (\tau \cap \rho).\langle \langle \mathsf{pr}_1\, x, \mathsf{pr}_2\, \mathsf{pr}_1\, x \rangle, \mathsf{pr}_2\, \mathsf{pr}_2\, x \rangle \wr &= \lambda x.x
\end{aligned}$$

and so on. Therefore, the essence of a Δ-term is its untyped skeleton: a strong pair $\langle \Delta_1, \Delta_2 \rangle$ can be typechecked if and only if $\wr \Delta_1 \wr \; \mathcal{R} \; \wr \Delta_2 \wr$ is verified, otherwise the strong pair will be ill-typed. The essence also gives the exact mapping between a term and its typing *à la* Church and its corresponding term and type assignment *à la* Curry.

The generic Δ-calculus is parametered with the essence relation $\mathcal{R}$, along with a type theory $\mathcal{T}$ (see Definition 2.1). Changing the parameters $\mathcal{T}$ and $\mathcal{R}$ results in defining a totally different intersection typed system. For the purpose of this chapter, we study the four well-known intersection type theories $\mathcal{T}$, namely Coppo-Dezani $\mathcal{T}_{\mathrm{CD}}$ [27], Coppo-Dezani-Sallé $\mathcal{T}_{\mathrm{CDS}}$ [28], Coppo-Dezani-Venneri $\mathcal{T}_{\mathrm{CDV}}$ [29] and Barendregt-Coppo-Dezani

Minimal type theory $\leqslant_{\min}$

(refl) $\sigma \leqslant \sigma$ (incl) $\sigma \cap \tau \leqslant \sigma, \sigma \cap \tau \leqslant \tau$

(glb) $\rho \leqslant \sigma, \rho \leqslant \tau \Rightarrow \rho \leqslant \sigma \cap \tau$ (trans) $\sigma \leqslant \tau, \tau \leqslant \rho \Rightarrow \sigma \leqslant \rho$

Axiom schemes

($\mathtt{U}_{top}$) $\sigma \leqslant \mathtt{U}$ ($\mathtt{U}_{\to}$) $\mathtt{U} \leqslant \sigma \to \mathtt{U}$

($\to\cap$) $(\sigma \to \tau) \cap (\sigma \to \rho) \leqslant \sigma \to (\tau \cap \rho)$

Rule scheme

($\to$) $\sigma_2 \leqslant \sigma_1, \tau_1 \leqslant \tau_2 \Rightarrow \sigma_1 \to \tau_1 \leqslant \sigma_2 \to \tau_2$

Figure 2.1: Minimal type theory $\leqslant_{\min}$, axioms and rule schemes (from Figure 13.2 and 13.3 of [12])

$$\frac{x{:}\sigma \in \Gamma}{\Gamma \vdash^{\mathcal{T}}_{\cap} x : \sigma}\ (ax) \qquad \frac{\Gamma, x{:}\sigma \vdash^{\mathcal{T}}_{\cap} M : \tau}{\Gamma \vdash^{\mathcal{T}}_{\cap} \lambda x.M : \sigma \to \tau}\ (\to I)$$

$$\frac{\Gamma \vdash^{\mathcal{T}}_{\cap} M : \sigma \quad \Gamma \vdash^{\mathcal{T}}_{\cap} M : \tau}{\Gamma \vdash^{\mathcal{T}}_{\cap} M : \sigma \cap \tau}\ (\cap I) \qquad \frac{\Gamma \vdash^{\mathcal{T}}_{\cap} M : \sigma \to \tau \quad \Gamma \vdash^{\mathcal{T}}_{\cap} N : \sigma}{\Gamma \vdash^{\mathcal{T}}_{\cap} M\,N : \tau}\ (\to E)$$

$$\frac{\Gamma \vdash^{\mathcal{T}}_{\cap} M : \sigma \cap \tau}{\Gamma \vdash^{\mathcal{T}}_{\cap} M : \sigma}\ (\cap E_1) \qquad \frac{\Gamma \vdash^{\mathcal{T}}_{\cap} M : \sigma \cap \tau}{\Gamma \vdash^{\mathcal{T}}_{\cap} M : \tau}\ (\cap E_2)$$

$$\frac{\mathtt{U} \in \mathbb{A}}{\Gamma \vdash^{\mathcal{T}}_{\cap} M : \mathtt{U}}\ (top) \qquad \frac{\Gamma \vdash^{\mathcal{T}}_{\cap} M : \sigma \quad \sigma \leqslant_{\mathcal{T}} \tau}{\Gamma \vdash^{\mathcal{T}}_{\cap} M : \tau}\ (\leqslant_{\mathcal{T}})$$

Figure 2.2: Generic intersection type assignment system $\lambda^{\mathcal{T}}_{\cap}$ (from Figure 13.8 of [12])

$\mathcal{T}_{\mathrm{BCD}}$ [11]. We will inspect the above type theories using three equivalence relations $\mathcal{R}$ on pure λ-terms, namely syntactical equality $\equiv$, β-equality $=_\beta$ and $\beta\eta$-equality $=_{\beta\eta}$.

The combination of the above $\mathcal{T}$ and $\mathcal{R}$ allows to define ten meaningful instances of the generic Δ-calculus that can be displayed in the Δ-chair (see Definition 2.9).

2.1 Syntax, Reduction and Types

Definition 2.1 (Type atoms, type syntax, type theories and type assignment systems)**.** We briefly review some basic definition from Subsection 13.1 of [12], in order to define type assignment systems. The set of atoms, intersection types, intersection type theories and intersection type assignment systems are defined as follows:

1. **(Atoms).** Let $\mathbb{A}$ denote a set of symbols which we will call type atoms, and let $\mathtt{U}$ be a special type atom denoting the universal type. In particular, we will use $\mathbb{A}_\infty = \{\mathtt{a}_i \mid i \in \mathbb{N}\}$ with $\mathtt{a}_i$ being different from $\mathtt{U}$ and $\mathbb{A}^{\mathtt{U}}_\infty = \mathbb{A}_\infty \cup \{\mathtt{U}\}$;

2. **(Syntax).** The syntax of intersection types, parametrized by $\mathbb{A}$, is: $\sigma ::= \mathbb{A} \mid \sigma \to \sigma \mid \sigma \cap \sigma$;

$\lambda_\cap^{\mathcal{T}}$	$\mathcal{T}$	$\mathbb{A}$	$\leqslant_{\min}$ plus	ref.
$\lambda_\cap^{\mathrm{CD}}$	$\mathcal{T}_{\mathrm{CD}}$	$\mathbb{A}_\infty$	$-$	[27]
$\lambda_\cap^{\mathrm{CDS}}$	$\mathcal{T}_{\mathrm{CDS}}$	$\mathbb{A}_\infty^{\mathtt{U}}$	$(\mathtt{U}_{top})$	[28]
$\lambda_\cap^{\mathrm{CDV}}$	$\mathcal{T}_{\mathrm{CDV}}$	$\mathbb{A}_\infty$	$(\to), (\to\cap)$	[29]
$\lambda_\cap^{\mathrm{BCD}}$	$\mathcal{T}_{\mathrm{BCD}}$	$\mathbb{A}_\infty^{\mathtt{U}}$	$(\to), (\to\cap), (\mathtt{U}_{top}), (\mathtt{U}\to)$	[11]

Figure 2.3: Type theories $\lambda_\cap^{\mathrm{CD}}$, $\lambda_\cap^{\mathrm{CDS}}$, $\lambda_\cap^{\mathrm{CDV}}$, and $\lambda_\cap^{\mathrm{BCD}}$. The "ref." column refers to the original articles these theories come from

3. **(Intersection type theories $\mathcal{T}$).** An intersection type theory $\mathcal{T}$ is a set of sentences of the form $\sigma \leqslant \tau$ satisfying at least the axioms and rules of the minimal type theory $\leqslant_{\min}$ defined in Figure 2.1. The type theories $\mathcal{T}_{\mathrm{CD}}, \mathcal{T}_{\mathrm{CDV}}, \mathcal{T}_{\mathrm{CDS}}$, and $\mathcal{T}_{\mathrm{BCD}}$ are the smallest type theories over $\mathbb{A}$ satisfying the axioms and rules given in Figure 2.3. We write $\mathcal{T}_1 \sqsubseteq \mathcal{T}_2$ if, for all σ, τ such that $\sigma \leqslant_{\mathcal{T}_1} \tau$, we have that $\sigma \leqslant_{\mathcal{T}_2} \tau$. In particular $\mathcal{T}_{\mathrm{CD}} \sqsubseteq \mathcal{T}_{\mathrm{CDV}} \sqsubseteq \mathcal{T}_{\mathrm{BCD}}$ and $\mathcal{T}_{\mathrm{CD}} \sqsubseteq \mathcal{T}_{\mathrm{CDS}} \sqsubseteq \mathcal{T}_{\mathrm{BCD}}$. We will sometime note, for instance, BCD instead of $\mathcal{T}_{\mathrm{BCD}}$;

4. **(Intersection type assignment systems $\lambda_\cap^{\mathcal{T}}$).** We define in Figure 2.2 an infinite collection of type assignment systems[1] parametrized by a set of atoms $\mathbb{A}$ and a type theory $\mathcal{T}$. We name four particular type assignment systems in the table below, which is an excerpt from Figure 13.4 of [12]. $\Gamma \vdash_\cap^{\mathcal{T}} M : \sigma$ denotes a derivable type assignment judgment in the type assignment system $\lambda_\cap^{\mathcal{T}}$. Type checking is not decidable for $\lambda_\cap^{\mathrm{CD}}$, $\lambda_\cap^{\mathrm{CDV}}$, $\lambda_\cap^{\mathrm{CDS}}$, and $\lambda_\cap^{\mathrm{BCD}}$ (see Theorem 2.24).

2.1.1 The Δ-calculi

Intersection type assignment systems and Δ-calculi have in common their type syntax and intersection type theories. The syntax of the generic Δ-calculus is defined as follows:

Definition 2.2 (Generic Δ-calculus syntax)**.**

$$\Delta \quad ::= \quad u_M \mid x \mid \lambda x{:}\sigma.\Delta \mid \Delta\,\Delta \mid \langle \Delta, \Delta \rangle \mid \mathsf{pr}_i\,\Delta \mid \Delta^\sigma \qquad\qquad i \in \{1,2\}$$

Intuitively, u_M denotes an infinite set of constants, indexed with a particular pure λ-term. Δ^τ denotes an explicit coercion[2] of a term Δ to type τ, where the typing rules will ensure that Δ has a type σ such that $\sigma \leqslant_{\mathcal{T}} \tau$. The expression $\langle \Delta, \Delta \rangle$ denotes a pair that, following the López-Escobar jargon [67], we call *strong pair* with respective projections pr_1 and pr_2. The essence function $\wr_\wr$ is an erasing function mapping typed Δ-terms into pure λ-terms. It is defined as follows:

Definition 2.3 (Essence function)**.**

$$\begin{array}{rcl@{\qquad}rcl@{\qquad}rcl}
\wr x \wr & \stackrel{def}{=} & x & \wr \Delta^\sigma \wr & \stackrel{def}{=} & \wr \Delta \wr & \wr u_M \wr & \stackrel{def}{=} & M \\
\wr \lambda x{:}\sigma.\Delta \wr & \stackrel{def}{=} & \lambda x.\wr \Delta \wr & \wr \Delta_1\,\Delta_2 \wr & \stackrel{def}{=} & \wr \Delta_1 \wr \wr \Delta_2 \wr & & & \\
\wr \langle \Delta_1, \Delta_2 \rangle \wr & \stackrel{def}{=} & \wr \Delta_1 \wr & \wr \mathsf{pr}_i\,\Delta \wr & \stackrel{def}{=} & \wr \Delta \wr \quad i \in \{1,2\} & & &
\end{array}$$

[1] Although rules $(\cap E_i)$ are derivable with $\leqslant_{\min}$, we add them for clarity.
[2] If type coercions are implicit, then we lose the uniqueness of type property.

One could argue that the choice of $\wr\langle\Delta_1,\Delta_2\rangle\wr \stackrel{def}{=} \wr\Delta_1\wr$ is arbitrary and could have been replaced with $\wr\langle\Delta_1,\Delta_2\rangle\wr \stackrel{def}{=} \wr\Delta_2\wr$. However, the typing rules will ensure that, if $\langle\Delta_1,\Delta_2\rangle$ is typable, then, for some suitable equivalence relation $\mathcal{R}$, we have that $\wr\Delta_1\wr\,\mathcal{R}\,\wr\Delta_2\wr$. Thus, strong pairs can be viewed as constrained cartesian products. The reduction semantics reduces terms of the generic Δ-calculus as follows:

Definition 2.4 (Generic reduction semantics)**.** Let syntactical equality by denoted by $\equiv$.

1. **(Substitution)** Substitution on Δ-terms is defined as usual, with the additional rules:
$$\begin{array}{rcl} u_M[\Delta/x] & \stackrel{def}{=} & u_{(M[\wr\Delta\wr/x])} \\ \Delta_1^\sigma[\Delta_2/x] & \stackrel{def}{=} & (\Delta_1[\Delta_2/x])^\sigma \end{array}$$

2. **(One-step reduction).** We define three notions of reduction:
$$\begin{array}{rcll} (\lambda x{:}\sigma.\Delta_1)\,\Delta_2 & \mapsto & \Delta_1[\Delta_2/x] & (\beta) \\ \mathsf{pr}_i\,\langle\Delta_1,\Delta_2\rangle & \mapsto & \Delta_i \quad i\in\{1,2\} & (\mathsf{pr}_i) \\ \lambda x{:}\sigma.\Delta\,x & \mapsto & \Delta \quad x\notin \mathsf{Fv}(\Delta) & (\eta) \end{array}$$
Observe that $(\lambda x{:}\sigma.\Delta_1)^\sigma\,\Delta_2$ is not a redex, because the λ-abstraction is coerced. The contextual closure is defined as usual except for reductions inside the index of u_M that are forbidden (even though substitutions are propagated). We write $\longrightarrow_{\beta\mathsf{pr}_i}$ for the contextual closure of the (β) and (pr_i) notions of reduction, $\longrightarrow_\eta$ for the contextual closure of (η). We also define a synchronous contextual closure, which is like the usual contextual closure except for the strong pairs, as defined in point (3). Synchronous contextual closure of the notions of reduction generates the reduction relations $\longrightarrow^{\|}_{\beta\mathsf{pr}_i}$ and $\longrightarrow^{\|}_\eta$.

3. **(Synchronous closure of $\longrightarrow^{\|}$).** Synchronous closure is defined on the strong pairs with the following constraint:
$$\frac{\Delta_1 \longrightarrow^{\|} \Delta_1' \quad \Delta_2 \longrightarrow^{\|} \Delta_2' \quad \wr\Delta_1'\wr \equiv \wr\Delta_2'\wr}{\langle\Delta_1,\Delta_2\rangle \longrightarrow^{\|} \langle\Delta_1',\Delta_2'\rangle}\ (Clos^{\|})$$
Note that we reduce in the two components of the strong pair. A longer and more detailed definition of synchronous reduction is given in Subsection 2.1.2;

4. **(Multistep reduction)**. We write $\twoheadrightarrow_{\beta\mathsf{pr}_i}$ (resp. $\twoheadrightarrow^{\|}_{\beta\mathsf{pr}_i}$) as the reflexive and transitive closure of $\longrightarrow_{\beta\mathsf{pr}_i}$ (resp. $\longrightarrow^{\|}_{\beta\mathsf{pr}_i}$);

5. **(Congruence)**. We write $=_{\beta\mathsf{pr}_i}$ as the symmetric, reflexive, transitive closure of $\twoheadrightarrow_{\beta\mathsf{pr}_i}$.

We mostly consider $\beta\mathsf{pr}_i$-reductions, thus to ease the notation, we will often omit the subscript in $\beta\mathsf{pr}_i$-reductions.

The next definition introduces a notion of synchronization inside strong pairs.

Definition 2.5 (Synchronization)**.** A Δ-term is synchronous if and only if, for all its subterms of the shape $\langle\Delta_1,\Delta_2\rangle$, we have that $\wr\Delta_1\wr \equiv \wr\Delta_2\wr$.

$$\frac{\Gamma \vdash^{\mathcal{T}}_{\mathcal{R}} \Delta : \sigma \quad \sigma \leqslant_{\mathcal{T}} \tau}{\Gamma \vdash^{\mathcal{T}}_{\mathcal{R}} \Delta^{\tau} : \tau}\ (\leqslant_{\mathcal{T}}) \qquad \frac{\Gamma, x{:}\sigma \vdash^{\mathcal{T}}_{\mathcal{R}} \Delta : \tau}{\Gamma \vdash^{\mathcal{T}}_{\mathcal{R}} \lambda x{:}\sigma.\Delta : \sigma \to \tau}\ (\to I)$$

$$\frac{\Gamma \vdash^{\mathcal{T}}_{\mathcal{R}} \Delta_1 : \sigma \to \tau \quad \Gamma \vdash^{\mathcal{T}}_{\mathcal{R}} \Delta_2 : \sigma}{\Gamma \vdash^{\mathcal{T}}_{\mathcal{R}} \Delta_1\, \Delta_2 : \tau}\ (\to E) \qquad \frac{x{:}\sigma \in \Gamma}{\Gamma \vdash^{\mathcal{T}}_{\mathcal{R}} x : \sigma}\ (ax)$$

$$\frac{\Gamma \vdash^{\mathcal{T}}_{\mathcal{R}} \Delta_1 : \sigma \quad \Gamma \vdash^{\mathcal{T}}_{\mathcal{R}} \Delta_2 : \tau \quad \wr \Delta_1 \wr\, \mathcal{R} \wr \Delta_2 \wr}{\Gamma \vdash^{\mathcal{T}}_{\mathcal{R}} \langle \Delta_1, \Delta_2 \rangle : \sigma \cap \tau}\ (\cap I) \qquad \frac{\mathtt{U} \in \mathbb{A}}{\Gamma \vdash^{\mathcal{T}}_{\mathcal{R}} u_M : \mathtt{U}}\ (top)$$

$$\frac{\Gamma \vdash^{\mathcal{T}}_{\mathcal{R}} \Delta : \sigma \cap \tau}{\Gamma \vdash^{\mathcal{T}}_{\mathcal{R}} \mathsf{pr}_2\, \Delta : \tau}\ (\cap E_2) \qquad \frac{\Gamma \vdash^{\mathcal{T}}_{\mathcal{R}} \Delta : \sigma \cap \tau}{\Gamma \vdash^{\mathcal{T}}_{\mathcal{R}} \mathsf{pr}_1\, \Delta : \sigma}\ (\cap E_1)$$

Figure 2.4: Generic Δ-calculus $\Delta^{\mathcal{T}}_{\mathcal{R}}$

It is easy to verify that $\longrightarrow^{\parallel}$ preserves synchronization, while it is not the case for $\longrightarrow$. The next definition introduces an intersection typed system for the generic Δ-calculus that is parametrizable by $\mathcal{R}$, a suitable equivalence relation on pure λ-terms, and $\mathcal{T}$, a type theory, as follows:

Definition 2.6 (Generic Δ-calculus $\Delta^{\mathcal{T}}_{\mathcal{R}}$)**.** The generic Δ-calculus is defined in Figure 2.4. We denote by $\Delta^{\mathcal{T}}_{\mathcal{R}}$ a particular typed system with the type theory $\mathcal{T}$ and under an equivalence relation $\mathcal{R}$ and by $\Gamma \vdash^{\mathcal{T}}_{\mathcal{R}} \Delta : \sigma$ a corresponding typing judgment.

The typing rules are intuitive for a calculus *à la* Church except rules $(\cap I)$, (top) and $(\leqslant_{\mathcal{T}})$.

The typing rule for a strong pair $(\cap I)$ is similar to the typing rule for a cartesian product, except for the side-condition $\wr \Delta_1 \wr\, \mathcal{R} \wr \Delta_2 \wr$, forcing the two parts of the strong pair to have essences equivalent under $\mathcal{R}$, thus making a strong pair a special case of a cartesian pair. For instance, $\langle \lambda x{:}\sigma.\lambda y{:}\tau.x, \lambda x{:}\sigma.x \rangle$ is not typable in $\Delta^{\mathcal{T}}_{\equiv}$; meanwhile $\langle (\lambda x{:}\sigma.x)\, y, y \rangle$ is not typable in $\Delta^{\mathcal{T}}_{\equiv}$ but it is in $\Delta^{\mathcal{T}}_{=_\beta}$; and $\langle x, \lambda y{:}\sigma.(\lambda z{:}\tau.z)\, x\, y \rangle$ is not typable in $\Delta^{\mathcal{T}}_{\equiv}$ nor $\Delta^{\mathcal{T}}_{=_\beta}$ but it is in $\Delta^{\mathcal{T}}_{=_{\beta\eta}}$. In the typing rule (top), the subscript M in u_M is an arbitrary pure λ-term. The typing rule $(\leqslant_{\mathcal{T}})$ allows to change the type of a Δ-term from σ to τ if $\sigma \leqslant_{\mathcal{T}} \tau$: the term in the conclusion must record this change with an explicit type coercion $_^{\tau}$, producing the new term Δ^{τ}: explicit type coercions are important to keep the unicity of typing derivations.

The next definition introduces the generic intersection typed system.

Definition 2.7 (Generic intersection typed system $\lambda@^{\mathcal{T}}_{\mathcal{R}}$)**.** For historical reasons (see [63, 64, 40]), we used judgments where Δ-terms were decorated by their essence. We thus get judgments of the form $\Gamma \vdash^{\mathcal{T}}_{\mathcal{R}} M@\Delta : \sigma$ for a system called $\lambda@^{\mathcal{T}}_{\mathcal{R}}$. The derivation rules are given in Figure 2.5. The properties of $\lambda@^{\mathcal{T}}_{\mathcal{R}}$ are the same than those of $\Delta^{\mathcal{T}}_{\mathcal{R}}$, because the decorations do nothing but make the terms easier to understand for newcomers.

The next definition introduces a partial order over equivalence relations on pure λ-terms and an inclusion over typed systems as follows:

$$\frac{\Gamma \vdash^{\mathcal{T}}_{\mathcal{R}} M@\Delta : \sigma \quad \sigma \leqslant_{\mathcal{T}} \tau}{\Gamma \vdash^{\mathcal{T}}_{\mathcal{R}} M@\Delta^{\tau} : \tau}\ (\leqslant_{\mathcal{T}}) \qquad \frac{\Gamma, x{:}\sigma \vdash^{\mathcal{T}}_{\mathcal{R}} M@\Delta : \tau}{\Gamma \vdash^{\mathcal{T}}_{\mathcal{R}} \lambda x.M@\lambda x{:}\sigma.\Delta : \sigma \to \tau}\ (\to I)$$

$$\frac{\Gamma \vdash^{\mathcal{T}}_{\mathcal{R}} M@\Delta_1 : \sigma \to \tau \quad \Gamma \vdash^{\mathcal{T}}_{\mathcal{R}} N@\Delta_2 : \sigma}{\Gamma \vdash^{\mathcal{T}}_{\mathcal{R}} M\,N@\Delta_1\,\Delta_2 : \tau}\ (\to E) \qquad \frac{x{:}\sigma \in \Gamma}{\Gamma \vdash^{\mathcal{T}}_{\mathcal{R}} x@x : \sigma}\ (ax)$$

$$\frac{\Gamma \vdash^{\mathcal{T}}_{\mathcal{R}} M@\Delta_1 : \sigma \quad \Gamma \vdash^{\mathcal{T}}_{\mathcal{R}} N@\Delta_2 : \tau \quad M\ \mathcal{R}\ N}{\Gamma \vdash^{\mathcal{T}}_{\mathcal{R}} M@\langle \Delta_1, \Delta_2 \rangle : \sigma \cap \tau}\ (\cap I) \qquad \frac{\mathtt{U} \in \mathbb{A}}{\Gamma \vdash^{\mathcal{T}}_{\mathcal{R}} M@u_M : \mathtt{U}}\ (top)$$

$$\frac{\Gamma \vdash^{\mathcal{T}}_{\mathcal{R}} M@\Delta : \sigma \cap \tau}{\Gamma \vdash^{\mathcal{T}}_{\mathcal{R}} M@\mathsf{pr}_2\,\Delta : \tau}\ (\cap E_2) \qquad \frac{\Gamma \vdash^{\mathcal{T}}_{\mathcal{R}} M@\Delta : \sigma \cap \tau}{\Gamma \vdash^{\mathcal{T}}_{\mathcal{R}} M@\mathsf{pr}_1\,\Delta : \sigma}\ (\cap E_1)$$

Figure 2.5: Generic intersection typed system $\lambda@^{\mathcal{T}}_{\mathcal{R}}$

Definition 2.8 ($\mathcal{R}$ and $\sqsubseteq$)**.**

1. Let $\mathcal{R} \in \{\equiv, =_\beta, =_{\beta\eta}\}$. $\mathcal{R}_1 \sqsubseteq \mathcal{R}_2$ if, for all pure λ-terms M, N such that $M\ \mathcal{R}_1\ N$, we have that $M\ \mathcal{R}_2\ N$;

2. $\Delta^{\mathcal{T}_1}_{\mathcal{R}_1} \sqsubseteq \Delta^{\mathcal{T}_2}_{\mathcal{R}_2}$ if, for any Γ, Δ, σ such that $\Gamma \vdash^{\mathcal{T}_1}_{\mathcal{R}_1} \Delta : \sigma$, we have that $\Gamma \vdash^{\mathcal{T}_2}_{\mathcal{R}_2} \Delta : \sigma$.

Note that $\sqsubseteq$ correspond to the standard inclusion between relation.

Proposition 2.1.

1. $\Delta^{\mathrm{CD}}_{\mathcal{R}} \sqsubseteq \Delta^{\mathrm{CDS}}_{\mathcal{R}} \sqsubseteq \Delta^{\mathrm{BCD}}_{\mathcal{R}}$ and $\Delta^{\mathrm{CD}}_{\mathcal{R}} \sqsubseteq \Delta^{\mathrm{CDV}}_{\mathcal{R}} \sqsubseteq \Delta^{\mathrm{BCD}}_{\mathcal{R}}$;

2. $\Delta^{\mathcal{T}_1}_{\mathcal{R}_1} \sqsubseteq \Delta^{\mathcal{T}_2}_{\mathcal{R}_2}$ if $\mathcal{T}_1 \sqsubseteq \mathcal{T}_2$ and $\mathcal{R}_1 \sqsubseteq \mathcal{R}_2$.

2.1.2 The Δ-chair

The next definition classifies ten instances of the generic Δ-calculus.

Definition 2.9 (Δ-chair)**.** Ten typed systems $\Delta^{\mathcal{T}}_{\mathcal{R}}$ can be draw in a diagram called Δ-chair, as in Figure 2.6, where the arrows represent an inclusion relation. $\Delta^{\mathrm{CD}}_{\equiv}$ corresponds roughly to [63, 64] (in the expression $M@\Delta$, M is the essence of Δ) and in its intersection part to [88]; $\Delta^{\mathrm{CDS}}_{\equiv}$ corresponds roughly in its intersection part to [40], $\Delta^{\mathrm{BCD}}_{\equiv}$ corresponds in its intersection part to [65], $\Delta^{\mathrm{CD}}_{=_{\beta\eta}}$ corresponds in its intersection part to [39]. The other typed systems are basically new. The main properties of these systems are:

1. All the $\Delta^{\mathcal{T}}_{\equiv}$ systems enjoys the synchronous subject reduction property, the other systems also enjoy ordinary subject reduction (Theorem 2.11);

2. All the systems strongly normalize (Theorem 2.21);

3. All the systems correspond to the to original type assignment systems except $\Delta^{\mathrm{CD}}_{=_\beta}$, $\Delta^{\mathrm{CDV}}_{=_\beta}$, $\Delta^{\mathrm{CDV}}_{=_{\beta\eta}}$ and $\Delta^{\mathrm{BCD}}_{=_{\beta\eta}}$ (Theorem 2.22);

4. Type checking and type reconstruction are decidable for all the systems, except $\Delta^{\mathrm{CDS}}_{=_\beta}$, $\Delta^{\mathrm{BCD}}_{=_\beta}$, and $\Delta^{\mathrm{BCD}}_{=_{\beta\eta}}$ (Theorem 2.24).

$$
\begin{tikzcd}
 & \Delta^{\mathrm{CDV}}_{=\beta\eta} \arrow[r] & \Delta^{\mathrm{BCD}}_{=\beta\eta} \\
 & \Delta^{\mathrm{CDV}}_{=\beta} \arrow[r] \arrow[u] & \Delta^{\mathrm{BCD}}_{=\beta} \arrow[u] \\
\Delta^{\mathrm{CD}}_{=\beta} \arrow[ur] \arrow[rr] & & \Delta^{\mathrm{CDS}}_{=\beta} \arrow[ur] \\
 & \Delta^{\mathrm{CDV}}_{\equiv} \arrow[r] \arrow[u] & \Delta^{\mathrm{BCD}}_{\equiv} \arrow[u] \\
\Delta^{\mathrm{CD}}_{\equiv} \arrow[ur] \arrow[uu] \arrow[rr] & & \Delta^{\mathrm{CDS}}_{\equiv} \arrow[ur] \arrow[uu]
\end{tikzcd}
$$

Figure 2.6: The Δ-chair

2.2 Examples

This section shows examples of typed derivations $\Delta^{\mathcal{T}}_{\mathcal{R}}$ and highlights the corresponding type assignment judgment in $\lambda^{\mathcal{T}}_{\cap}$ they correspond to, in the sense that we have a derivation $\Gamma \vdash^{\mathcal{T}}_{\mathcal{R}} \Delta : \sigma$ and another derivation $\Gamma \vdash^{\mathcal{T}}_{\cap} \wr \Delta \wr : \sigma$. The correspondence between intersection typed systems $\Delta^{\mathcal{T}}_{\mathcal{R}}$ and intersection type assignment $\lambda^{\mathcal{T}}_{\cap}$ will be defined in Subsection 2.4.1.

Example 2.1 (Polymorphic identity)**.** In all of the intersection type assignment systems $\lambda^{\mathcal{T}}_{\cap}$ we can derive:

$$\vdash^{\mathcal{T}}_{\cap} \lambda x.x : (\sigma \to \sigma) \cap (\tau \to \tau)$$

A corresponding Δ-term is:

$$\langle \lambda x{:}\sigma.x, \lambda x{:}\tau.x \rangle$$

It can be typed in all of the typed systems of the Δ-chair as follows:

$$
\dfrac{\dfrac{x{:}\sigma \vdash^{\mathcal{T}}_{\mathcal{R}} x : \sigma}{\vdash^{\mathcal{T}}_{\mathcal{R}} \lambda x{:}\sigma.x : \sigma \to \sigma} \quad \dfrac{x{:}\tau \vdash^{\mathcal{T}}_{\mathcal{R}} x : \tau}{\vdash^{\mathcal{T}}_{\mathcal{R}} \lambda x{:}\tau.x : \tau \to \tau} \quad \lambda x.x \; \mathcal{R} \; \lambda x.x}{\vdash^{\mathcal{T}}_{\mathcal{R}} \langle \lambda x{:}\sigma.x, \lambda x{:}\tau.x \rangle : (\sigma \to \sigma) \cap (\tau \to \tau)}
$$

Example 2.2 (Auto application)**.** In all of the intersection type assignment systems we can derive:

$$\vdash^{\mathcal{T}}_{\cap} \lambda x.x\,x : ((\sigma \to \tau) \cap \sigma) \to \tau$$

A corresponding Δ-term is:

$$\lambda x{:}(\sigma \to \tau) \cap \sigma.(\mathsf{pr}_1\, x)(\mathsf{pr}_2\, x)$$

It can be typed in all of the typed systems of the Δ-chair as follows:

$$
\dfrac{\dfrac{\dfrac{x{:}(\sigma \to \tau) \cap \sigma \vdash^{\mathcal{T}}_{\mathcal{R}} x : (\sigma \to \tau) \cap \sigma}{x{:}(\sigma \to \tau) \cap \sigma \vdash^{\mathcal{T}}_{\mathcal{R}} \mathsf{pr}_1\, x : \sigma \to \tau} \quad \dfrac{x{:}(\sigma \to \tau) \cap \sigma \vdash^{\mathcal{T}}_{\mathcal{R}} x : (\sigma \to \tau) \cap \sigma}{x{:}(\sigma \to \tau) \cap \sigma \vdash^{\mathcal{T}}_{\mathcal{R}} \mathsf{pr}_2\, x : \sigma}}{x{:}(\sigma \to \tau) \cap \sigma \vdash^{\mathcal{T}}_{\mathcal{R}} (\mathsf{pr}_1\, x)(\mathsf{pr}_2\, x) : \tau}}{\vdash^{\mathcal{T}}_{\mathcal{R}} \lambda x{:}(\sigma \to \tau) \cap \sigma.(\mathsf{pr}_1\, x)(\mathsf{pr}_2\, x) : (\sigma \to \tau) \cap \sigma \to \tau}
$$

Example 2.3 (Some examples in $\Delta_{\mathcal{R}}^{\text{CDS}}$)**.** In $\lambda_{\cap}^{\text{CDS}}$ we can derive:

$$\vdash_{\cap}^{\mathcal{T}_{\text{CDS}}} (\lambda x.\lambda y.x) : \sigma \to \mathtt{U} \to \sigma$$

Using this type assignment, we can derive:

$$z{:}\sigma \vdash_{\cap}^{\mathcal{T}_{\text{CDS}}} (\lambda x.\lambda y.x)\, z\, z : \sigma$$

A corresponding Δ-term is:

$$(\lambda x{:}\sigma.\lambda y{:}\mathtt{U}.x)\, z\, z^{\mathtt{U}}$$

It can be typed in $\Delta_{\mathcal{R}}^{\text{CDS}}$ as follows:

$$\dfrac{\dfrac{\dfrac{\dfrac{z{:}\sigma, x{:}\sigma, y{:}\mathtt{U} \vdash_{\mathcal{R}}^{\mathcal{T}_{\text{CDS}}} x : \sigma}{z{:}\sigma, x{:}\sigma \vdash_{\mathcal{R}}^{\mathcal{T}_{\text{CDS}}} \lambda y{:}\mathtt{U}.x : \mathtt{U} \to \sigma}}{z{:}\sigma \vdash_{\mathcal{R}}^{\mathcal{T}_{\text{CDS}}} \lambda x{:}\sigma.\lambda y{:}\mathtt{U}.x : \sigma \to \mathtt{U} \to \sigma} \quad z{:}\sigma \vdash_{\mathcal{R}}^{\mathcal{T}_{\text{CDS}}} z : \sigma}{z{:}\sigma \vdash_{\mathcal{R}}^{\mathcal{T}_{\text{CDS}}} (\lambda x{:}\sigma.\lambda y{:}\mathtt{U}.x)\, z : \mathtt{U} \to \sigma} \quad \dfrac{z{:}\sigma \vdash_{\mathcal{R}}^{\mathcal{T}_{\text{CDS}}} z : \sigma \quad \sigma \leqslant_{\mathcal{T}_{\text{CDS}}} \mathtt{U}}{z{:}\sigma \vdash_{\mathcal{R}}^{\mathcal{T}_{\text{CDS}}} z^{\mathtt{U}} : \mathtt{U}}}{z{:}\sigma \vdash_{\mathcal{R}}^{\mathcal{T}_{\text{CDS}}} (\lambda x{:}\sigma.\lambda y{:}\mathtt{U}.x)\, z\, z^{\mathtt{U}} : \sigma}$$

As another example, we can also derive:

$$\vdash_{\cap}^{\mathcal{T}_{\text{CDS}}} \lambda x.x : \sigma \to \sigma \cap \mathtt{U}$$

A corresponding Δ-term is:

$$\lambda x{:}\sigma.\langle x, x^{\mathtt{U}}\rangle$$

It can be typed in $\Delta_{\mathcal{R}}^{\text{CDS}}$ as follows:

$$\dfrac{\dfrac{x{:}\sigma \vdash_{\mathcal{R}}^{\mathcal{T}_{\text{CDS}}} x : \sigma \quad \dfrac{x{:}\sigma \vdash_{\mathcal{R}}^{\mathcal{T}_{\text{CDS}}} x : \sigma \quad \sigma \leqslant_{\mathcal{T}_{\text{CDS}}} \mathtt{U}}{x{:}\sigma \vdash_{\mathcal{R}}^{\mathcal{T}_{\text{CDS}}} x^{\mathtt{U}} : \mathtt{U}} \quad x \mathrel{\mathcal{R}} x}{x{:}\sigma \vdash_{\mathcal{R}}^{\mathcal{T}_{\text{CDS}}} \langle x, x^{\mathtt{U}}\rangle : \sigma \cap \mathtt{U}}}{\vdash_{\mathcal{R}}^{\mathcal{T}_{\text{CDS}}} \lambda x{:}\sigma.\langle x, x^{\mathtt{U}}\rangle : \sigma \to \sigma \cap \mathtt{U}}$$

Example 2.4 (An example in $\Delta_{\mathcal{R}}^{\text{CDV}}$)**.** In $\lambda_{\cap}^{\text{CDV}}$ we can prove the commutativity of intersection:

$$\vdash_{\cap}^{\mathcal{T}_{\text{CDV}}} \lambda x.x : \sigma \cap \tau \to \tau \cap \sigma$$

A corresponding Δ-term is:

$$\langle \lambda x{:}\sigma \cap \tau.\mathsf{pr}_2\, x, \lambda x{:}\sigma \cap \tau.\mathsf{pr}_1\, x\rangle^{(\sigma\cap\tau)\to(\tau\cap\sigma)}$$

It can be typed in $\Delta_{\mathcal{R}}^{\text{CDV}}$ as follows:

$$\dfrac{\dfrac{\dfrac{x{:}\sigma \cap \tau \vdash_{\mathcal{R}}^{\mathcal{T}_{\text{CDS}}} x : \sigma \cap \tau}{x{:}\sigma \cap \tau \vdash_{\mathcal{R}}^{\mathcal{T}_{\text{CDS}}} \mathsf{pr}_2\, x : \tau}}{\vdash_{\mathcal{R}}^{\mathcal{T}_{\text{CDS}}} \lambda x{:}\sigma \cap \tau.\mathsf{pr}_2\, x : (\sigma \cap \tau) \to \tau} \quad \dfrac{\dfrac{x{:}\sigma \cap \tau \vdash_{\mathcal{R}}^{\mathcal{T}_{\text{CDS}}} x : \sigma \cap \tau}{x{:}\sigma \cap \tau \vdash_{\mathcal{R}}^{\mathcal{T}_{\text{CDS}}} \mathsf{pr}_1\, x : \sigma}}{\vdash_{\mathcal{R}}^{\mathcal{T}_{\text{CDS}}} \lambda x{:}\sigma \cap \tau.\mathsf{pr}_1\, x : (\sigma \cap \tau) \to \sigma} \quad \lambda x.x \mathrel{\mathcal{R}} \lambda x.x}{\vdash_{\mathcal{R}}^{\mathcal{T}_{\text{CDS}}} \langle \lambda x{:}\sigma \cap \tau.\mathsf{pr}_2\, x, \lambda x{:}\sigma \cap \tau.\mathsf{pr}_1\, x\rangle : ((\sigma \cap \tau) \to \tau) \cap ((\sigma \cap \tau) \to \sigma)} \quad *}{\vdash_{\mathcal{R}}^{\mathcal{T}_{\text{CDS}}} \langle \lambda x{:}\sigma \cap \tau.\mathsf{pr}_2\, x, \lambda x{:}\sigma \cap \tau.\mathsf{pr}_1\, x\rangle^{(\sigma\cap\tau)\to(\tau\cap\sigma)} : (\sigma \cap \tau) \to (\tau \cap \sigma)}$$

where $*$ is $((\sigma \cap \tau) \to \tau) \cap ((\sigma \cap \tau) \to \sigma) \leqslant_{\mathcal{T}_{\text{CDV}}} (\sigma \cap \tau) \to (\tau \cap \sigma)$.

Example 2.5 (Another polymorphic identity in $\Delta^{\mathcal{T}}_{=_\beta}$)**.** In all the $\Delta^{\mathcal{T}}_{=_\beta}$ you can type this Δ-term:

$$\langle \lambda x{:}\sigma.x, (\lambda x{:}\tau{\to}\tau.x)\,(\lambda x{:}\tau.x)\rangle$$

The typing derivation is the following:

$$\dfrac{\dfrac{x{:}\sigma \vdash^{\mathcal{T}}_{=_\beta} x : \sigma}{\vdash^{\mathcal{T}}_{=_\beta} \lambda x{:}\sigma.x : \sigma \to \sigma} \quad \dfrac{\dfrac{x{:}\tau \to \tau \vdash^{\mathcal{T}}_{=_\beta} x : \tau \to \tau}{\vdash^{\mathcal{T}}_{=_\beta} \lambda x{:}\tau{\to}\tau.x : (\tau \to \tau) \to (\tau \to \tau)} \quad \dfrac{x{:}\tau \vdash^{\mathcal{T}}_{=_\beta} x : \tau}{\vdash^{\mathcal{T}}_{=_\beta} \lambda x{:}\tau.x : \tau \to \tau}}{\vdash^{\mathcal{T}}_{=_\beta} (\lambda x{:}\tau{\to}\tau.x)\,(\lambda x{:}\tau.x) : \tau \to \tau \;\; \lambda x.x =_\beta (\lambda x.x)\,(\lambda x.x)}}{\vdash^{\mathcal{T}}_{=_\beta} \langle \lambda x{:}\sigma.x, (\lambda x{:}\tau{\to}\tau.x)\,(\lambda x{:}\tau.x)\rangle : (\sigma \to \sigma) \cap (\tau \to \tau)}$$

Example 2.6 (Two examples in $\Delta^{\mathrm{BCD}}_{\equiv}$ and $\Delta^{\mathrm{BCD}}_{=_{\beta\eta}}$)**.** In $\lambda^{\mathrm{BCD}}_{\cap}$ we can can type any term, including the following non-terminating term:

$$\Omega \stackrel{def}{=} (\lambda x.x\,x)\,(\lambda x.x\,x)$$

More precisely, we have:

$$\vdash^{\mathcal{T}_{\mathrm{BCD}}}_{\cap} \Omega : \mathtt{U}$$

A corresponding Δ-term whose essence is Ω is:

$$(\lambda x{:}\mathtt{U}.x^{\mathtt{U}\to\mathtt{U}}\,x)\,(\lambda x{:}\mathtt{U}.x^{\mathtt{U}\to\mathtt{U}}\,x)^{\mathtt{U}}$$

It can be typed in $\Delta^{\mathrm{BCD}}_{\mathcal{R}}$ as follows:

$$\dfrac{\dfrac{*}{\vdash^{\mathcal{T}_{\mathrm{BCD}}}_{\mathcal{R}} \lambda x{:}\mathtt{U}.x^{\mathtt{U}\to\mathtt{U}}\,x : \mathtt{U} \to \mathtt{U}} \quad \dfrac{\dfrac{*}{\vdash^{\mathcal{T}_{\mathrm{BCD}}}_{\mathcal{R}} \lambda x{:}\mathtt{U}.x^{\mathtt{U}\to\mathtt{U}}\,x : \mathtt{U} \to \mathtt{U}} \quad \mathtt{U} \to \mathtt{U} \leqslant_{\mathcal{T}_{\mathrm{BCD}}} \mathtt{U}}{\vdash^{\mathcal{T}_{\mathrm{BCD}}}_{\mathcal{R}} (\lambda x{:}\mathtt{U}.x^{\mathtt{U}\to\mathtt{U}}\,x)^{\mathtt{U}} : \mathtt{U}}}{\vdash^{\mathcal{T}_{\mathrm{BCD}}}_{\mathcal{R}} (\lambda x{:}\mathtt{U}.x^{\mathtt{U}\to\mathtt{U}}\,x)\,(\lambda x{:}\mathtt{U}.x^{\mathtt{U}\to\mathtt{U}}\,x)^{\mathtt{U}} : \mathtt{U}}$$

where $*$ is:

$$\dfrac{\dfrac{x{:}\mathtt{U} \vdash^{\mathcal{T}_{\mathrm{BCD}}}_{\mathcal{R}} x : \mathtt{U} \quad \mathtt{U} \leqslant_{\mathcal{T}_{\mathrm{BCD}}} \mathtt{U} \to \mathtt{U}}{x{:}\mathtt{U} \vdash^{\mathcal{T}_{\mathrm{BCD}}}_{\mathcal{R}} x^{\mathtt{U}\to\mathtt{U}} : \mathtt{U} \to \mathtt{U}} \quad x{:}\mathtt{U} \vdash^{\mathcal{T}_{\mathrm{BCD}}}_{\mathcal{R}} x : \mathtt{U}}{x{:}\mathtt{U} \vdash^{\mathcal{T}_{\mathrm{BCD}}}_{\mathcal{R}} x^{\mathtt{U}\to\mathtt{U}}\,x : \mathtt{U}}$$

In $\lambda^{\mathrm{BCD}}_{\cap}$ we can type the following:

$$x{:}\mathtt{U} \to \mathtt{U} \vdash^{\mathcal{T}_{\mathrm{BCD}}}_{\cap} x : (\mathtt{U} \to \mathtt{U}) \cap (\sigma \to \mathtt{U})$$

A corresponding Δ-term (whose essence is x) is:

$$\langle x, \lambda y{:}\sigma.x\,y^{\mathtt{U}}\rangle$$

It can be typed in $\Delta^{\mathrm{BCD}}_{=_{\beta\eta}}$ as follows:

$$\dfrac{x{:}\mathtt{U} \to \mathtt{U} \vdash^{\mathcal{T}_{\mathrm{BCD}}}_{=_{\beta\eta}} x : \mathtt{U} \to \mathtt{U} \quad \dfrac{\dfrac{x{:}\mathtt{U} \to \mathtt{U}, y{:}\sigma \vdash^{\mathcal{T}_{\mathrm{BCD}}}_{=_{\beta\eta}} y : \sigma \quad \sigma \leqslant \mathtt{U}}{x{:}\mathtt{U} \to \mathtt{U}, y{:}\sigma \vdash^{\mathcal{T}_{\mathrm{BCD}}}_{=_{\beta\eta}} y^{\mathtt{U}} : \mathtt{U}}}{x{:}\mathtt{U} \to \mathtt{U}, y{:}\sigma \vdash^{\mathcal{T}_{\mathrm{BCD}}}_{=_{\beta\eta}} x\,y^{\mathtt{U}} : \mathtt{U}}}{x{:}\mathtt{U} \to \mathtt{U} \vdash^{\mathcal{T}_{\mathrm{BCD}}}_{=_{\beta\eta}} \lambda y{:}\sigma.x\,y^{\mathtt{U}} : \sigma \to \mathtt{U}} \quad x =_{\beta\eta} \lambda y.x\,y}{x{:}\mathtt{U} \to \mathtt{U} \vdash^{\mathcal{T}_{\mathrm{BCD}}}_{=_{\beta\eta}} \langle x, \lambda y{:}\sigma.x\,y^{\mathtt{U}}\rangle : (\mathtt{U} \to \mathtt{U}) \cap (\sigma \to \mathtt{U})}$$

Note that the $=_{\beta\eta}$ condition has an interesting loophole, as it is well-known that $\lambda^{\mathrm{BCD}}_{\cap}$ does not enjoy $=_\eta$-conversion property. Theorem 2.17(1) will show that we can construct a Δ-term which does not correspond to any $\lambda^{\mathrm{BCD}}_{\cap}$ derivation.

Example 2.7 (Pottinger [80])**.** The following examples can be typed in all the type theories of the Δ-chair (we also display in square brackets the corresponding pure λ-terms typable in $\lambda^{\mathcal{T}}_{\cap}$). These are encodings from the examples *à la* Curry given by Pottinger in [80].

$[\lambda x.\lambda y.x\,y]$

$\vdash^{\mathcal{T}}_{\mathcal{R}} \lambda x{:}(\sigma \to \tau) \cap (\sigma \to \rho).\lambda y{:}\sigma.\langle(\mathsf{pr}_1\, x)\, y), (\mathsf{pr}_2\, x)\, y\rangle : (\sigma \to \tau) \cap (\sigma \to \rho) \to \sigma \to \tau \cap \rho$

$[\lambda x.\lambda y.x\,y]$

$\vdash^{\mathcal{T}}_{\mathcal{R}} \lambda x{:}\sigma \to \tau \cap \rho.\langle\lambda y{:}\sigma.\mathsf{pr}_1\,(x\,y), \lambda y{:}\sigma.\mathsf{pr}_2\,(x\,y)\rangle : (\sigma \to \tau \cap \rho) \to (\sigma \to \tau) \cap (\sigma \to \rho)$

$[\lambda x.\lambda y.x\,y]$

$\vdash^{\mathcal{T}}_{\mathcal{R}} \lambda x{:}\sigma \to \rho.\lambda y{:}\sigma \cap \tau.x\,(\mathsf{pr}_1\, y) : (\sigma \to \rho) \to \sigma \cap \tau \to \rho$

$[\lambda x.\lambda y.x]$

$\vdash^{\mathcal{T}}_{\mathcal{R}} \lambda x{:}\sigma \cap \tau.\lambda y{:}\sigma.\mathsf{pr}_2\, x : \sigma \cap \tau \to \sigma \to \tau$

$[\lambda x.\lambda y.x\,y\,y]$

$\vdash^{\mathcal{T}}_{\mathcal{R}} \lambda x{:}\sigma \to \tau \to \rho.\lambda y{:}\sigma \cap \tau.x\,(\mathsf{pr}_1\, y)\,(\mathsf{pr}_2\, y) : (\sigma \to \tau \to \rho) \to \sigma \cap \tau \to \rho$

$[\lambda x.x]$

$\vdash^{\mathcal{T}}_{\mathcal{R}} \lambda x{:}\sigma \cap \tau.\mathsf{pr}_1\, x : \sigma \cap \tau \to \sigma$

$[\lambda x.x]$

$\vdash^{\mathcal{T}}_{\mathcal{R}} \lambda x{:}\sigma.\langle x, x\rangle : \sigma \to \sigma \cap \sigma$

$[\lambda x.x]$

$\vdash^{\mathcal{T}}_{\mathcal{R}} \lambda x{:}\sigma \cap (\tau \cap \rho).\langle\langle\mathsf{pr}_1\, x, \mathsf{pr}_1\, \mathsf{pr}_2\, x\rangle, \mathsf{pr}_2\, \mathsf{pr}_2\, x\rangle : \sigma \cap (\tau \cap \rho) \to (\sigma \cap \tau) \cap \rho$

In the same paper, Pottinger lists some types that cannot be inhabited by any intersection type assignment ($\nvdash^{\mathcal{T}}_{\cap}$) in an empty context, namely: $\sigma \to (\sigma \cap \tau)$ and $(\sigma \to \tau) \to (\sigma \to \rho) \to \sigma \to \tau \cap \rho$ and $((\sigma \cap \tau) \to \rho) \to \sigma \to \tau \to \rho$. It is not difficult to verify that the above types cannot be inhabited by any of the type systems of the Δ-chair because of the failure of the essence condition in the strong pair type rule.

Example 2.8 (Intersection is not the conjunction operator [55, 13])**.** This counter-example is from the corresponding counter-example *à la* Curry given by Hindley [55] and Ben-Yelles [13]. The intersection type $(\sigma \to \sigma) \cap ((\sigma \to \tau \to \rho) \to (\sigma \to \tau) \to \sigma \to \rho)$ where the left part of the intersection corresponds to the type for the combinator I and the right part for the combinator S cannot be assigned to a pure λ-term. Analogously, the same intersection type cannot be assigned to any Δ-term belonging to a type system from the Δ-chair, because of the failure of the essence condition.

2.2.1 On synchronization and subject reduction

For the typed systems $\Delta^{\mathcal{T}}_{\equiv}$, strong pairs have an intrinsic notion of synchronization: some redexes need to be reduced in a synchronous fashion unless we want to create meaningless

Δ-terms that cannot be typed. Consider the Δ-term $\langle(\lambda x{:}\sigma.x)\,y, (\lambda x{:}\sigma.x)\,y\rangle$. If we use the $\longrightarrow$ reduction relation, then the following reduction paths are legal:

$$\langle(\lambda x{:}\sigma.x)\,y, (\lambda x{:}\sigma.x)\,y\rangle \begin{array}{c} \nearrow^{\beta} \langle(\lambda x{:}\sigma.x)\,y, y\rangle \searrow_{\beta} \\ \searrow_{\beta} \langle y, (\lambda x{:}\sigma.x)\,y\rangle \nearrow^{\beta} \end{array} \langle y, y\rangle$$

More precisely, the first and second redexes are rewritten asynchronously, therefore they cannot be typed in any typed system $\Delta^{\mathcal{T}}_{\equiv}$, because we fail to check that the left and the right part of the strong pair are syntactically the same: the $\longrightarrow^{\parallel}$ reduction relation prevents this loophole and allows to type all redexes. In summary, $\longrightarrow^{\parallel}$ can be thought of as the natural reduction relation for the typed systems $\Delta^{\mathcal{T}}_{\equiv}$.

2.3 Metatheory of $\Delta^{\mathcal{T}}_{\mathcal{R}}$

2.3.1 General properties

Unless specified, all properties applies to the intersection typed systems $\Delta^{\mathcal{T}}_{\mathcal{R}}$.

The Church-Rosser property is proved using the technique of Takahashi [90]. The parallel reduction semantics extends Definition 2.4 and it is inductively defined as follows:

Definition 2.10 (Parallel reduction semantics)**.**

$$\begin{array}{rcll}
x & \Longrightarrow & x & \\
u_M & \Longrightarrow & u_M & \\
\Delta^{\sigma} & \Longrightarrow & (\Delta')^{\sigma} & \text{if } \Delta \Longrightarrow \Delta' \\
\Delta_1\,\Delta_2 & \Longrightarrow & \Delta'_1\,\Delta'_2 & \text{if } \Delta_1 \Longrightarrow \Delta'_1 \quad \text{and } \Delta_2 \Longrightarrow \Delta'_2 \\
\lambda x{:}\sigma.\Delta & \Longrightarrow & \lambda x{:}\sigma.\Delta' & \text{if } \Delta \Longrightarrow \Delta' \\
(\lambda x{:}\sigma.\Delta_1)\,\Delta_2 & \Longrightarrow & \Delta'_1[\Delta'_2/x] & \text{if } \Delta_1 \Longrightarrow \Delta'_1 \quad \text{and } \Delta_2 \Longrightarrow \Delta'_2 \\
\langle\Delta_1, \Delta_2\rangle & \Longrightarrow & \langle\Delta'_1, \Delta'_2\rangle & \text{if } \Delta_1 \Longrightarrow \Delta'_1 \quad \text{and } \Delta_2 \Longrightarrow \Delta'_2 \\
\mathsf{pr}_i\,\Delta & \Longrightarrow & \mathsf{pr}_i\,\Delta' & \text{if } \Delta \Longrightarrow \Delta' \quad \text{and } i \in \{1,2\} \\
\mathsf{pr}_i\,\langle\Delta_1, \Delta_2\rangle & \Longrightarrow & \Delta'_i & \text{if } \Delta_i \Longrightarrow \Delta'_i \quad \text{and } i \in \{1,2\}
\end{array}$$

Intuitively, $\Delta \Longrightarrow \Delta'$ means that Δ' is obtained from Δ by simultaneous arbitrary contractions of some $\beta\mathsf{pr}_i$-redexes possibly overlapping each other. Church-Rosser can be achieved by proving a stronger statement, namely:

$$\Delta \Longrightarrow \Delta' \quad \text{implies} \quad \Delta' \Longrightarrow \Delta^*$$

where Δ^* is a Δ-term determined by Δ and independent from Δ'. The statement (2.3.1) is satisfied by the term Δ^* which is obtained from Δ by contracting all the redexes existing in Δ simultaneously, as is shown in the following definition.

Definition 2.11 (The map $_^*$)**.**

$$
\begin{array}{rcll}
x^* & \stackrel{def}{=} & x & \\
u_M^* & \stackrel{def}{=} & u_M & \\
(\Delta^\sigma)^* & \stackrel{def}{=} & (\Delta^*)^\sigma & \\
\langle \Delta_1, \Delta_2 \rangle^* & \stackrel{def}{=} & \langle \Delta_1^*, \Delta_2^* \rangle & \\
(\lambda x{:}\sigma.\Delta)^* & \stackrel{def}{=} & \lambda x{:}\sigma.\Delta^* & \\
(\Delta_1\, \Delta_2)^* & \stackrel{def}{=} & \Delta_1^*\, \Delta_2^* & \text{if } \Delta_1\, \Delta_2 \text{ is not a } \beta\text{-redex} \\
((\lambda x{:}\sigma.\Delta_1)\, \Delta_2)^* & \stackrel{def}{=} & \Delta_1^*[\Delta_2^*/x] & \\
(\mathsf{pr}_i\, \Delta)^* & \stackrel{def}{=} & \mathsf{pr}_i\, \Delta^* & \text{if } \Delta \text{ is not a strong pair} \\
(\mathsf{pr}_i\, \langle \Delta_1, \Delta_2 \rangle)^* & \stackrel{def}{=} & \Delta_i^* & i \in \{1, 2\}
\end{array}
$$

The next technical lemma will be useful in showing that Church-Rosser for $\twoheadrightarrow$ can be inherited from Church-Rosser for $\Longrightarrow$.

Lemma 2.2.

1. If $\Delta_1 \longrightarrow \Delta_1'$, then $\Delta_1 \Longrightarrow \Delta_1'$;
2. if $\Delta_1 \Longrightarrow \Delta_1'$, then $\Delta_1 \twoheadrightarrow \Delta_1'$;
3. if $\Delta_1 \Longrightarrow \Delta_1'$ and $\Delta_2 \Longrightarrow \Delta_2'$, then $\Delta_1[\Delta_2/x] \Longrightarrow \Delta_1'[\Delta_2'/x]$;
4. $\Delta_1 \Longrightarrow \Delta_1^*$.

Proof.

1. Let $C[\cdot]$ be an applicative context, Δ either a β-redex or a pr_i-redex, and Δ' its contractum, such that $\Delta_1 \equiv C[\Delta]$ and $\Delta_1' \equiv C[\Delta']$. We can check that $\Delta \Longrightarrow \Delta'$, and, by induction on $C[\cdot]$, we conclude that $\Delta_1 \Longrightarrow \Delta_1'$;

2. 3. 4. By induction on the structure of Δ_1.

□

We can now prove the Church-Rosser property for the parallel reduction:

Lemma 2.3 (Confluence property for $\Longrightarrow$)**.** If $\Delta \Longrightarrow \Delta'$, then $\Delta' \Longrightarrow \Delta^*$.

Proof. By induction on the shape of Δ.

- if $\Delta \equiv x$, then $\Delta' \equiv x \Longrightarrow x \equiv \Delta^*$;
- if $\Delta \equiv u_M$, then $\Delta' \equiv u_M \Longrightarrow u_M \equiv \Delta^*$;
- if $\Delta \equiv \Delta_1^\sigma$, then, for some Δ_1', we have that $\Delta_1 \Longrightarrow \Delta_1'$ and $\Delta' \equiv (\Delta_1')^\sigma$, therefore, by induction hypothesis, $\Delta' \Longrightarrow (\Delta_1^*)^\sigma \equiv \Delta^*$;

- if $\Delta \equiv \langle \Delta_1, \Delta_2 \rangle$, then, for some Δ'_1 and Δ'_2, we have that $\Delta_1 \Longrightarrow \Delta'_1$, $\Delta_2 \Longrightarrow \Delta'_2$ and $\Delta' \equiv \langle \Delta'_1, \Delta'_2 \rangle$. By induction hypothesis, $\Delta' \Longrightarrow \langle \Delta^*_1, \Delta^*_2 \rangle \equiv \Delta^*$;
- if $\Delta \equiv \lambda x{:}\sigma.\Delta_1$, then, for some Δ'_1, we have that $\Delta_1 \Longrightarrow \Delta'_1$ and $\Delta' \equiv \lambda x{:}\sigma.\Delta'_1$. By induction hypothesis, $\lambda x{:}\sigma.\Delta'_1 \Longrightarrow \lambda x{:}\sigma.\Delta^*_1 \equiv \Delta^*$;
- if $\Delta \equiv \Delta_1\, \Delta_2$ and Δ is not a β-redex, then, for some Δ'_1 and Δ'_2, we have that $\Delta_1 \Longrightarrow \Delta'_1$, $\Delta_2 \Longrightarrow \Delta'_2$ and $\Delta' \equiv \Delta'_1\, \Delta'_2$. By induction hypothesis, $\Delta' \Longrightarrow \Delta^*_1\, \Delta^*_2 \equiv \Delta^*$;
- if $\Delta \equiv (\lambda x{:}\sigma.\Delta_1)\, \Delta_2$, then, for some Δ'_1 and Δ'_2, we have that $\Delta_1 \Longrightarrow \Delta'_1$, $\Delta_2 \Longrightarrow \Delta'_2$ and we have 2 subcases:
  - $\Delta' \equiv (\lambda x{:}\sigma.\Delta'_1)\, \Delta'_2$: by induction hypothesis, $\Delta' \Longrightarrow \Delta^*_1[\Delta^*_2/x] \equiv \Delta^*$;
  - $\Delta' \equiv \Delta'_1[\Delta'_2/x]$: we also have $\Delta' \Longrightarrow \Delta^*_1[\Delta^*_2/x]$, thanks to point (3) of Lemma 2.2;
- if $\Delta \equiv \mathsf{pr}_i\, \Delta_1$ and Δ_1 is not a strong pair, then, for some Δ'_1, we have that $\Delta_1 \Longrightarrow \Delta'_1$ and $\Delta' \equiv \mathsf{pr}_i\, \Delta'_1$, therefore, by induction hypothesis, $\Delta' \Longrightarrow \mathsf{pr}_i\, \Delta^*_1 \equiv \Delta^*$;
- if $\Delta \equiv \mathsf{pr}_i\, \langle \Delta_1, \Delta_2 \rangle$, then, for some Δ'_1 and Δ'_2, we have that $\Delta_1 \Longrightarrow \Delta'_1$, $\Delta_2 \Longrightarrow \Delta'_2$ and we have 2 subcases:
  - $\Delta' \equiv \mathsf{pr}_i\, \langle \Delta'_1, \Delta'_2 \rangle$: by induction hypothesis, $\Delta' \Longrightarrow \Delta^*_i \equiv \Delta^*$;
  - $\Delta' \equiv \Delta'_i$: we also have, by induction hypothesis, $\Delta' \Longrightarrow \Delta^*_i \equiv \Delta^*$.

□

The Church-Rosser property follows from Lemma 2.3.

Theorem 2.4 (Confluence)**.**
If $\Delta_1 \longrightarrow\!\!\!\!\rightarrow \Delta_2$ and $\Delta_1 \longrightarrow\!\!\!\!\rightarrow \Delta_3$, then there exists Δ_4 such that $\Delta_2 \longrightarrow\!\!\!\!\rightarrow \Delta_4$ and $\Delta_3 \longrightarrow\!\!\!\!\rightarrow \Delta_4$.

Proof. Thanks to the first two points of Lemma 2.2, we know that $\longrightarrow\!\!\!\!\rightarrow$ is the transitive closure of $\Longrightarrow$, therefore we can deduce the confluence property of $\longrightarrow\!\!\!\!\rightarrow$ with the usual diagram chase, as suggested below.

$$\begin{tikzcd}
\Delta_{0,0} \arrow[r, Rightarrow] \arrow[d, Rightarrow] & \Delta_{1,0} \arrow[r, Rightarrow] \arrow[d, two heads, dashed] & \Delta_{2,0} \arrow[r, Rightarrow] \arrow[d, two heads, dashed] & \Delta_{3,0} \arrow[d, two heads, dashed] \\
\Delta_{0,1} \arrow[r, Rightarrow] \arrow[d, Rightarrow] & \Delta_{1,1} \arrow[r, Rightarrow] \arrow[d, two heads, dashed] & \Delta_{2,1} \arrow[r, Rightarrow] \arrow[d, two heads, dashed] & \Delta_{3,1} \arrow[d, two heads, dashed] \\
\Delta_{0,2} \arrow[r, Rightarrow] & \Delta_{1,2} \arrow[r, Rightarrow] & \Delta_{2,2} \arrow[r, Rightarrow] & \Delta_{3,2}
\end{tikzcd}$$

□

The next lemma says that all type derivations for Δ have an unique type.

Lemma 2.5 (Unicity of typing)**.**
If $\Gamma \vdash^{\mathcal{T}}_{\mathcal{R}} \Delta : \sigma$, then σ is unique.

Proof. By induction on the shape of Δ. □

The next lemma proves inversion properties on typable Δ-terms.

Lemma 2.6 (Inversion)**.**

1. If $\Gamma \vdash^{\mathcal{T}}_{\mathcal{R}} x : \sigma$, then $x{:}\sigma \in \Gamma$;
2. if $\Gamma \vdash^{\mathcal{T}}_{\mathcal{R}} \lambda x{:}\sigma.\Delta : \rho$, then $\rho \equiv \sigma \to \tau$ for some τ and $\Gamma, x{:}\sigma \vdash^{\mathcal{T}}_{\mathcal{R}} \Delta : \tau$;
3. if $\Gamma \vdash^{\mathcal{T}}_{\mathcal{R}} \Delta_1\, \Delta_2 : \tau$, then there is σ such that $\Gamma \vdash^{\mathcal{T}}_{\mathcal{R}} \Delta_1 : \sigma \to \tau$ and $\Gamma \vdash^{\mathcal{T}}_{\mathcal{R}} \Delta_2 : \sigma$;
4. if $\Gamma \vdash^{\mathcal{T}}_{\mathcal{R}} \langle \Delta_1, \Delta_2 \rangle : \rho$, then there is σ, τ such that $\rho \equiv \sigma \cap \tau$ and $\Gamma \vdash^{\mathcal{T}}_{\mathcal{R}} \Delta_1 : \sigma$ and $\Gamma \vdash^{\mathcal{T}}_{\mathcal{R}} \Delta_2 : \tau$ and $\wr\, \Delta_1 \wr\, \mathcal{R} \wr\, \Delta_2 \wr$;
5. if $\Gamma \vdash^{\mathcal{T}}_{\mathcal{R}} \mathsf{pr}_1\, \Delta : \sigma$, then there is τ such that $\Gamma \vdash^{\mathcal{T}}_{\mathcal{R}} \Delta : \sigma \cap \tau$;
6. if $\Gamma \vdash^{\mathcal{T}}_{\mathcal{R}} \mathsf{pr}_2\, \Delta : \tau$, then there is σ such that $\Gamma \vdash^{\mathcal{T}}_{\mathcal{R}} \Delta : \sigma \cap \tau$;
7. if $\Gamma \vdash^{\mathcal{T}}_{\mathcal{R}} u_M : \sigma$, then $\sigma \equiv \mathtt{U}$;
8. if $\Gamma \vdash^{\mathcal{T}}_{\mathcal{R}} \Delta^{\tau} : \rho$, then $\rho \equiv \tau$ and there is σ such that $\sigma \leqslant_{\mathcal{T}} \tau$ and $\Gamma \vdash^{\mathcal{T}}_{\mathcal{R}} \Delta : \sigma$.

Proof. The typing rules are uniquely syntax-directed, therefore we can immediately conclude. □

The next lemma says that all subterms of a typable Δ-term are typable too.

Lemma 2.7 (Subterms typability)**.**
If $\Gamma \vdash^{\mathcal{T}}_{\mathcal{R}} \Delta : \sigma$, and Δ' is a subterm of Δ, then there exists Γ' and τ such that $\Gamma' \supseteq \Gamma$ and $\Gamma' \vdash^{\mathcal{T}}_{\mathcal{R}} \Delta' : \tau$.

Proof. By induction on the derivation of $\Gamma \vdash^{\mathcal{T}}_{\mathcal{R}} \Delta : \sigma$. For instance, let's consider the case where the applied rule is $(\to I)$. The other cases are similar. If the last applied rule is $(\to I)$, then $\Delta \equiv \lambda x{:}\sigma_1.\Delta_1$ and $\sigma \equiv \sigma_1 \to \sigma_2$ for some σ_1, σ_2, and Δ_1. Moreover, Δ' is a subterm of Δ_1, and:

$$\Gamma, x{:}\sigma_1 \vdash \Delta_1 : \sigma_2$$

By induction hypothesis, we know that there is an extension Γ' of $\Gamma, x{:}\sigma_1$ such that $\Gamma' \vdash^{\mathcal{T}}_{\mathcal{R}} \Delta' : \tau$. As Γ' is also an extension of Γ, we can conclude. □

As expected, the weakening and strengthening properties on contexts are verified.

Lemma 2.8 (Free-variable properties)**.**

1. If $\Gamma \vdash^{\mathcal{T}}_{\mathcal{R}} \Delta : \sigma$, and $\Gamma' \supseteq \Gamma$, then $\Gamma' \vdash^{\mathcal{T}}_{\mathcal{R}} \Delta : \sigma$;
2. if $\Gamma \vdash^{\mathcal{T}}_{\mathcal{R}} \Delta : \sigma$, then $\mathsf{Fv}(\Delta) \subseteq \mathsf{Dom}(\Gamma)$;
3. if $\Gamma \vdash^{\mathcal{T}}_{\mathcal{R}} \Delta : \sigma$, $\Gamma' \subseteq \Gamma$ and $\mathsf{Fv}(\Delta) \subseteq \mathsf{Dom}(\Gamma')$, then $\Gamma' \vdash^{\mathcal{T}}_{\mathcal{R}} \Delta : \sigma$.

Proof. By induction on the derivation of $\Gamma \vdash^{\mathcal{T}}_{\mathcal{R}} \Delta : \sigma$. □

The next lemma also says that essence is closed under substitution.

Lemma 2.9 (Substitution)**.**

1. $\wr\, \Delta_1[\Delta_2/x] \wr \equiv \wr\, \Delta_1 \wr[\wr\, \Delta_2 \wr/x]$;

2. If $\Gamma, x{:}\sigma \vdash^{\mathcal{T}}_{\mathcal{R}} \Delta_1 : \tau$ and $\Gamma \vdash^{\mathcal{T}}_{\mathcal{R}} \Delta_2 : \sigma$, then $\Gamma \vdash^{\mathcal{T}}_{\mathcal{R}} \Delta_1[\Delta_2/x] : \tau$.

Proof.

1. by induction on the shape of Δ_1;
2. by induction on the derivation. As an illustration, we show the case when the last applied rule is $(\cap I)$. In this case, we know that:

$$\Gamma, x{:}\sigma \vdash^{\mathcal{T}}_{\mathcal{R}} \langle \Delta_1, \Delta_1' \rangle : \tau \cap \tau' \quad \text{and} \quad \Gamma \vdash^{\mathcal{T}}_{\mathcal{R}} \Delta_2 : \sigma$$

By induction hypothesis, we have:

$$\Gamma \vdash^{\mathcal{T}}_{\mathcal{R}} \Delta_1[\Delta_2/x] : \tau \quad \text{and} \quad \Gamma \vdash^{\mathcal{T}}_{\mathcal{R}} \Delta_1'[\Delta_2/x] : \tau'$$

Moreover, thanks to point (1), we can show that:

$$\wr \Delta_1[\Delta_2/x] \wr \mathcal{R} \wr \Delta_1'[\Delta_2/x] \wr$$

As a consequence:

$$\frac{\Gamma \vdash^{\mathcal{T}}_{\mathcal{R}} \Delta_1[\Delta_2/x] : \tau \quad \Gamma \vdash^{\mathcal{T}}_{\mathcal{R}} \Delta_1'[\Delta_2/x] : \tau' \quad \wr \Delta_1[\Delta_2/x] \wr \mathcal{R} \wr \Delta_1'[\Delta_2/x] \wr}{\Gamma \vdash^{\mathcal{T}}_{\mathcal{R}} \langle \Delta_1, \Delta_1' \rangle[\Delta_2/x] : \tau \cap \tau'} \ (\cap I)$$

□

In order to prove subject reduction, we need to prove that reducing Δ-terms preserve the side-condition $\wr \Delta_1 \wr \mathcal{R} \wr \Delta_2 \wr$ when typing the strong pair $\langle \Delta_1, \Delta_2 \rangle$. We prove this in the following lemma.

Lemma 2.10 (Essence reduction)**.**

1. if $\Gamma \vdash^{\mathcal{T}}_{\equiv} \Delta_1 : \sigma$ and $\Delta_1 \longrightarrow \Delta_2$, then $\wr \Delta_1 \wr =_\beta \wr \Delta_2 \wr$;
2. for $\mathcal{R} \in \{=_\beta, =_{\beta\eta}\}$, if $\Gamma \vdash^{\mathcal{T}}_{\mathcal{R}} \Delta_1 : \sigma$ and $\Delta_1 \longrightarrow \Delta_2$, then $\wr \Delta_1 \wr \mathcal{R} \wr \Delta_2 \wr$;
3. if $\Gamma \vdash^{\mathcal{T}}_{=_{\beta\eta}} \Delta_1 : \sigma$ and $\Delta_1 \longrightarrow_\eta \Delta_2$, then $\wr \Delta_1 \wr =_\eta \wr \Delta_2 \wr$.

Proof. If Δ_1 is a redex, then we have three cases:

- if $\Delta_1 \equiv (\lambda x{:}\sigma.\Delta_1')\,\Delta_1''$ and Δ_2 is $\Delta_1'[\Delta_1''/x]$, then, thanks to Lemma 2.9(1) we have that $\wr \Delta_2 \wr \equiv \wr \Delta_1' \wr[\wr \Delta_1'' \wr/x]$, therefore $\wr \Delta_1 \wr =_\beta \wr \Delta_2 \wr$;
- if $\Delta_1 \equiv \mathsf{pr}_i\,\langle \Delta_1', \Delta_2' \rangle$ and Δ_2 is Δ_i', we know that Δ_1 is typable in $\Delta^{\mathcal{T}}_{\mathcal{R}}$, and thanks to Lemma 2.6(4), we have that $\wr \Delta_1' \wr \mathcal{R} \wr \Delta_2' \wr$. As a consequence, $\wr \Delta_1 \wr \mathcal{R} \wr \Delta_2 \wr$;
- if $\Delta_1 \equiv \lambda x{:}\sigma.\Delta'\,x$ with $x \notin \mathsf{Fv}(\Delta')$, and Δ_2 is Δ', then $\wr \Delta_1 \wr =_\eta \wr \Delta_2 \wr$.

For the contextual closure, we have that $\Delta_1 \equiv \Delta[\Delta'/x]$, where Δ acts as an applicative context and Δ' is a redex, and Δ_2 is $\Delta[\Delta''/x]$ where Δ'' is the contractum of Δ'.

Then, as Δ' is a subterm of Δ_1, by Lemma 2.7 we deduce that Δ' is typable, therefore Δ'' is also typable, and then we infer, using Lemma 2.9(1), that:

$$\wr \Delta_1 \wr \equiv \wr \Delta \wr[\wr \Delta' \wr/x] \quad \text{and} \quad \wr \Delta_2 \wr \equiv \wr \Delta \wr[\wr \Delta'' \wr/x]$$

Then we can conclude. □

The next theorem states that all the typed systems $\Delta^{\mathcal{T}}_{\equiv}$ preserve synchronous $\beta\mathsf{pr}_i$-reduction, and all the typed systems $\Delta^{\mathcal{T}}_{=_\beta}$ and $\Delta^{\mathcal{T}}_{=_{\beta\eta}}$ preserve $\beta\mathsf{pr}_i$-reduction.

Theorem 2.11 (Subject reduction for $\beta\mathsf{pr}_i$).

1. If $\Gamma \vdash^{\mathcal{T}}_{\equiv} \Delta_1 : \sigma$ and $\Delta_1 \longrightarrow^{\parallel} \Delta_2$, then $\Gamma \vdash^{\mathcal{T}}_{\equiv} \Delta_2 : \sigma$;
2. for $\mathcal{R} \in \{=_\beta, =_{\beta\eta}\}$, if $\Gamma \vdash^{\mathcal{T}}_{\mathcal{R}} \Delta_1 : \sigma$ and $\Delta_1 \longrightarrow \Delta_2$, then $\Gamma \vdash^{\mathcal{T}}_{\mathcal{R}} \Delta_2 : \sigma$.

Proof. We proceed by looking at the cases where Δ_1 is a redex and Δ_2 its contractum, then we consider the contextual closure:

- If Δ_1 is a β-redex $(\lambda x{:}\tau.\Delta)\,\Delta'$, and $\Delta_2 \equiv \Delta[\Delta'/x]$, then by Lemma 2.6, the derivation tree of Δ_1 ends with:

$$\dfrac{\dfrac{\Gamma, x{:}\tau \vdash^{\mathcal{T}}_{\mathcal{R}} \Delta : \tau \to \sigma}{\Gamma \vdash^{\mathcal{T}}_{\mathcal{R}} \lambda x{:}\tau.\Delta : \tau \to \sigma} \quad \Gamma \vdash^{\mathcal{T}}_{\mathcal{R}} \Delta' : \tau}{\Gamma \vdash^{\mathcal{T}}_{\mathcal{R}} (\lambda x{:}\tau.\Delta)\,\Delta' : \sigma}$$

  We conclude that $\Gamma \vdash^{\mathcal{T}}_{\mathcal{R}} \Delta_2 : \sigma$ by using Lemma 2.9(2);

- If Δ_1 is a pr_i-redex $\mathsf{pr}_i\,\langle\Delta, \Delta'\rangle$, and Δ_2 is Δ (if $i = 1$) or Δ' (if $i = 2$), then by Lemma 2.6, the derivation tree of Δ_1 ends with:

$$\dfrac{\dfrac{\Gamma \vdash^{\mathcal{T}}_{\mathcal{R}} \Delta : \sigma_1 \quad \Gamma \vdash^{\mathcal{T}}_{\mathcal{R}} \Delta' : \sigma_2}{\Gamma \vdash^{\mathcal{T}}_{\mathcal{R}} \langle\Delta, \Delta'\rangle : \sigma_1 \cap \sigma_2}}{\Gamma \vdash^{\mathcal{T}}_{\mathcal{R}} \mathsf{pr}_i\,\langle\Delta, \Delta'\rangle : \sigma_i}$$

  Then we see immediately that $\Gamma \vdash^{\mathcal{T}}_{\mathcal{R}} \Delta_2 : \sigma_i$;

- For the contextual closure, we proceed by induction on the derivation: we illustrate the most important case, namely $(\cap I)$ where we have to check that the essence condition is preserved. According to $\mathcal{R}$ we distinguish two cases:

  1. (Case where $\mathcal{R}$ is $\equiv$). If $\Gamma \vdash^{\mathcal{T}}_{\equiv} \langle\Delta_1, \Delta_2\rangle : \sigma \cap \tau$ and $\langle\Delta_1, \Delta_2\rangle \longrightarrow^{\parallel} \langle\Delta'_1, \Delta'_2\rangle$, then $\wr\Delta'_1\wr \equiv \wr\Delta'_2\wr$ and, by induction hypothesis, $\Gamma \vdash^{\mathcal{T}}_{\equiv} \Delta'_1 : \sigma$ and $\Gamma \vdash^{\mathcal{T}}_{\equiv} \Delta'_2 : \tau$, therefore $\Gamma \vdash^{\mathcal{T}}_{\equiv} \langle\Delta'_1, \Delta'_2\rangle : \sigma \cap \tau$;
  2. (Case where $\mathcal{R} \in \{=_\beta, =_{\beta\eta}\}$). If $\Gamma \vdash^{\mathcal{T}}_{\mathcal{R}} \langle\Delta_1, \Delta_2\rangle : \sigma \cap \tau$ and $\langle\Delta_1, \Delta_2\rangle \longrightarrow \langle\Delta'_1, \Delta'_2\rangle$, then:
     - $\wr\Delta_1\wr\,\mathcal{R}\,\wr\Delta_2\wr$;
     - by Lemma 2.10 we have that $\wr\Delta'_1\wr\,\mathcal{R}\,\wr\Delta_1\wr$ and $\wr\Delta_2\wr\,\mathcal{R}\,\wr\Delta'_2\wr$;
     - by induction hypothesis we have that $\Gamma \vdash^{\mathcal{T}}_{\mathcal{R}} \Delta'_1 : \sigma$ and $\Gamma \vdash^{\mathcal{T}}_{\mathcal{R}} \Delta'_2 : \tau$;

     therefore $\wr\Delta'_1\wr\,\mathcal{R}\,\wr\Delta'_2\wr$ and $\Gamma \vdash^{\mathcal{T}}_{\mathcal{R}} \langle\Delta'_1, \Delta'_2\rangle : \sigma \cap \tau$.

□

The next theorem states that some of the typed systems on the back of the Δ-chair preserve η-reduction.

Theorem 2.12 (Subject reduction for η for $\mathcal{T}_{\mathrm{CDV}}, \mathcal{T}_{\mathrm{BCD}}$).
Let $\mathcal{T} \in \{\mathcal{T}_{\mathrm{CDS}}, \mathcal{T}_{\mathrm{BCD}}\}$. If $\Gamma \vdash^{\mathcal{T}}_{=_{\beta\eta}} \Delta_1 : \sigma$ and $\Delta_1 \longrightarrow_\eta \Delta_2$, then $\Gamma \vdash^{\mathcal{T}}_{=_{\beta\eta}} \Delta_2 : \sigma$.

Proof. If Δ_1 is a η-redex, then we proceed as usual using Lemmas 2.6 and 2.8. For the contextual closure the proof proceeds exactly as in Theorem 2.11. □

Remark 2.1 (About subject expansion)**.**
We know that some of the intersection type assignment systems *à la* Curry (*viz.* $\lambda^{\text{BCD}}_{\cap}$ and $\lambda^{\text{CDS}}_{\cap}$) satisfy the subject β-expansion property: one may ask whether this property can also be meaningful in typed systems *à la* Church. It is not surprising to see that the answer is negative because type-decorations of bound variables are hard-coded in the λ-abstraction and cannot be forgotten. As a trivial example of the failure of the subject-expansion in all the typed systems, consider the following reduction:

$$(\lambda x{:}\sigma.x)\,(\lambda x{:}\sigma.x) \longrightarrow (\lambda x{:}\sigma.x)$$

Obviously we can type $\vdash^{\mathcal{T}}_{\mathcal{R}} (\lambda x{:}\sigma.x) : \sigma \to \sigma$ but $\nvdash^{\mathcal{T}}_{\mathcal{R}} (\lambda x{:}\sigma.x)\,(\lambda x{:}\sigma.x) : \sigma \to \sigma$.

2.3.2 Synchronous reduction

We want to define synchronous β-reduction $\longrightarrow^{\parallel}_{\beta}$ such that, whenever $\Delta_1 \longrightarrow^{\parallel}_{\beta} \Delta_2$, we have $\wr\,\Delta_1\,\wr \longrightarrow_{\beta} \wr\,\Delta_2\,\wr$. In order to do that, we extend the generic Δ-calculus and the λ-calculus with an underlining that book-keeps all the reductions that have to be done synchronously. This technique is adapted from the one used in Section 2.3 of [10].

We define the syntax of the $\underline{\lambda}$-calculus and $\underline{\Delta}$-calculus as follows:

$$\begin{array}{rcl}
\underline{M} & ::= & x \mid \lambda x.\underline{M} \mid (\underline{\lambda} x.\underline{M}_1)\,\underline{M}_2 \mid \underline{M}_1\,\underline{M}_2 \\
\underline{\Delta} & ::= & x \mid \lambda x{:}\sigma.\underline{\Delta} \mid (\underline{\lambda} x{:}\sigma.\underline{\Delta}_1)\,\underline{\Delta}_2 \mid \underline{\Delta}_1\,\underline{\Delta}_2 \mid \\
& & \langle \underline{\Delta}_1, \underline{\Delta}_2 \rangle \mid \mathsf{pr}_1\,\underline{\Delta} \mid \mathsf{pr}_2\,\underline{\Delta} \mid \mathsf{in}^{\sigma}_1\,\underline{\Delta} \mid \mathsf{in}^{\sigma}_2\,\underline{\Delta} \mid u_M \mid \underline{\Delta}^{\sigma}
\end{array}$$

Note that there is no underlining in u_M, because there is no reduction inside the index of u_M. The essence function for the $\underline{\Delta}$-calculus is the same as for the Δ-calculus, with the following extra rule:

$$\wr\,\underline{\lambda} x{:}\sigma.\underline{\Delta}\,\wr \stackrel{def}{=} \underline{\lambda} x.\wr\,\underline{\Delta}\,\wr$$

We define the notion of $\underline{\beta}$-reduction for $\underline{\lambda}$-calculus and $\underline{\Delta}$-calculus:

$$\begin{array}{rcl}
(\underline{\lambda} x.\underline{M})\,\underline{N} & \mapsto_{\underline{\beta}} & \underline{M}[\underline{N}/x] \\
(\underline{\lambda} x{:}\sigma.\underline{\Delta}_1)\,\underline{\Delta}_2 & \mapsto_{\underline{\beta}} & \underline{\Delta}_1[\underline{\Delta}_2/x]
\end{array}$$

Then the syntactical closure of $\mapsto_{\underline{\beta}}$ is noted $\longrightarrow_{\underline{\beta}}$, and the transitive reflexive closure of $\longrightarrow_{\underline{\beta}}$ is noted $\twoheadrightarrow_{\underline{\beta}}$.

Definition 2.12. If $\underline{M}$ (resp. $\underline{\Delta}$) is a $\underline{\lambda}$-term (resp. $\underline{\Delta}$-term), then $|\underline{M}|$ (resp. $|\underline{\Delta}|$) is obtained by leaving out all the underlinings.

We define a partial function $\text{Sync}(\Delta, \underline{M})$ which, given a Δ-term Δ (without underlinings) and a $\underline{\lambda}$-term $\underline{M}$, either fails or return a $\underline{\Delta}$-term whose underlinings correspond to the underlinings of $\underline{M}$.

The most important rule is:

$$\text{Sync}((\lambda x{:}\sigma.\Delta_1)\,\Delta_2, (\underline{\lambda} x.\underline{M}_1)\,\underline{M}_2) \stackrel{def}{=} (\underline{\lambda} x{:}\sigma.\text{Sync}(\Delta_1, \underline{M}_1))\,\text{Sync}(\Delta_2, \underline{M}_2)$$

The other rules are structural:

$$
\begin{array}{lcll}
\mathrm{Sync}(\Delta, \underline{M}) & \stackrel{def}{=} & \Delta & \text{if } \underline{M} \text{ has no underlining and } \wr\Delta\wr \equiv \underline{M} \\
\mathrm{Sync}(\Delta_1\,\Delta_2, \underline{M}_1\,\underline{M}_2) & \stackrel{def}{=} & \mathrm{Sync}(\Delta_1, \underline{M}_1)\,\mathrm{Sync}(\Delta_2, \underline{M}_2) & \text{if } \underline{M}_1\,\underline{M}_2 \text{ is not a } \underline{\beta}\text{-redex} \\
\mathrm{Sync}(\lambda x{:}\sigma.\Delta, \lambda x.\underline{M}) & \stackrel{def}{=} & \lambda x{:}\sigma.\mathrm{Sync}(\Delta, \underline{M}) & \\
\mathrm{Sync}(\langle\Delta_1, \Delta_2\rangle, \underline{M}) & \stackrel{def}{=} & \langle\mathrm{Sync}(\Delta_1, \underline{M}), \mathrm{Sync}(\Delta_2, \underline{M})\rangle & \\
\mathrm{Sync}(\mathsf{pr}_i\,\Delta, \underline{M}) & \stackrel{def}{=} & \mathsf{pr}_i\,\mathrm{Sync}(\Delta, \underline{M}) & \\
\mathrm{Sync}(\mathsf{in}_i^\sigma\,\Delta, \underline{M}) & \stackrel{def}{=} & \mathsf{in}_i^\sigma\,\mathrm{Sync}(\Delta, \underline{M}) & \\
\mathrm{Sync}(\Delta^\sigma, \underline{M}) & \stackrel{def}{=} & (\mathrm{Sync}(\Delta, \underline{M}))^\sigma & \\
\mathrm{Sync}(\Delta, \underline{M}) & \stackrel{def}{=} & \mathrm{fail} & \text{otherwise}
\end{array}
$$

We define the reduction mapping φ from $\underline{\Delta}$-terms to Δ-terms, which reduces all underlined redexes. It is an extension of Definition 2.3.11 of [10].

Definition 2.13 (Reduction mapping)**.**

$$
\begin{array}{rcll}
\varphi(x) & \stackrel{def}{=} & x & \\
\varphi(\lambda x{:}\sigma.\underline{\Delta}) & \stackrel{def}{=} & \lambda x{:}\sigma.\varphi(\underline{\Delta}) & \\
\varphi((\underline{\lambda} x{:}\sigma.\underline{\Delta}_1)\,\underline{\Delta}_2) & \stackrel{def}{=} & \varphi(\underline{\Delta}_1)[\varphi(\underline{\Delta}_2)/x] & \\
\varphi(\underline{\Delta}_1\,\underline{\Delta}_2) & \stackrel{def}{=} & \varphi(\underline{\Delta}_1)\,\varphi(\underline{\Delta}_2) & \text{if } \underline{\Delta}_1\,\underline{\Delta}_2 \text{ is not a } \underline{\beta}\text{-redex} \\
\varphi(\langle\underline{\Delta}_1, \underline{\Delta}_2\rangle) & \stackrel{def}{=} & \langle\varphi(\underline{\Delta}_1), \varphi(\underline{\Delta}_2)\rangle & \\
\varphi(\mathsf{pr}_i\,\underline{\Delta}) & \stackrel{def}{=} & \mathsf{pr}_i\,\varphi(\underline{\Delta}) & \\
\varphi(\mathsf{in}_i^\sigma\,\underline{\Delta}) & \stackrel{def}{=} & \mathsf{in}_i^\sigma\,\varphi(\underline{\Delta}) & \\
\varphi(u_M) & \stackrel{def}{=} & u_M & \\
\varphi(\underline{\Delta}^\sigma) & \stackrel{def}{=} & \varphi(\underline{\Delta})^\sigma &
\end{array}
$$

We now define synchronous β-reduction, which, as we will prove in Theorem 2.15, keeps a synchronicity with the β-reduction in the essence.

Definition 2.14 (Synchronous β-reduction)**.**
Let Δ and $\underline{M}$ with exactly one underlining such that $|\underline{M}| \equiv \wr\Delta\wr$ and $\mathrm{Sync}(\Delta, \underline{M})$ is defined. We define synchronous β-reduction as:

$$\Delta \longrightarrow_\beta^{\parallel} \varphi(\mathrm{Sync}(\Delta, \underline{M}))$$

The reflexive and transitive closure of $\longrightarrow_\beta^{\parallel}$ is noted $\twoheadrightarrow_\beta^{\parallel}$.

The following two lemmas establish the basic properties of synchronizations and $\underline{\beta}$-reduction.

Lemma 2.13.

1. $\wr \mathrm{Sync}(\Delta, \underline{M}) \wr \equiv \underline{M}$;
2. $|\mathrm{Sync}(\Delta, \underline{M})| \equiv \Delta$.

Proof. By induction on Δ. □

Lemma 2.14.

1. If $\underline{M} \longrightarrow_{\underline{\beta}} \underline{N}$, then $|\underline{M}| \longrightarrow_{\beta} |\underline{N}|$;
2. If $\underline{\Delta}_1 \longrightarrow_{\underline{\beta}} \underline{\Delta}_2$, then $|\underline{\Delta}_1| \longrightarrow_{\beta} |\underline{\Delta}_2|$;
3. $\underline{\Delta} \twoheadrightarrow_{\underline{\beta}} \varphi(\underline{\Delta})$;
4. If $\underline{\Delta}_1 \longrightarrow_{\underline{\beta}} \underline{\Delta}_2$, then either $\wr \underline{\Delta}_1 \wr \longrightarrow_{\underline{\beta}} \wr \underline{\Delta}_2 \wr$, or $\wr \underline{\Delta}_1 \wr \equiv \wr \underline{\Delta}_2 \wr$.

Proof.

1. 2. 3. Easy;

4. By induction on $\underline{\Delta}_1$.

□

The next theorem shows that synchronous relation behave correctly in typed and untyped reductions.

Theorem 2.15.
If $\Delta_1 \longrightarrow^{\|}_{\beta} \Delta_2$, then:

1. $\Delta_1 \twoheadrightarrow_{\beta} \Delta_2$;
2. $\wr \Delta_1 \wr \longrightarrow_{\beta} \wr \Delta_2 \wr$.

Proof. We know that there is some $\underline{\lambda}$-term $\underline{M}$ with only one underlining such that $|\underline{M}| \equiv \wr \Delta_1 \wr$ and $\varphi(\mathrm{Sync}(\Delta_1, \underline{M})) \equiv \Delta_2$.

1. We know, by Lemma 2.13 that $\Delta_1 \equiv |\mathrm{Sync}(\Delta_1, \underline{M})|$. Moreover, by Lemma 2.14, we know that $\mathrm{Sync}(\Delta_1, \underline{M}) \longrightarrow_{\underline{\beta}} \varphi(\mathrm{Sync}(\Delta_1, \underline{M}))$, therefore $\Delta_1 \twoheadrightarrow_{\beta} |\varphi(\mathrm{Sync}(\Delta_1, \underline{M}))|$. However, the function φ returns a Δ-term without underlining, therefore:

$$|\varphi(\mathrm{Sync}(\Delta_1, \underline{M}))| \equiv \varphi(\mathrm{Sync}(\Delta_1, \underline{M}))$$

   As a conclusion, we have that $\Delta_1 \twoheadrightarrow_{\beta} \Delta_2$;

2. We know that $\mathrm{Sync}(\Delta_1, \underline{M}) \twoheadrightarrow_{\underline{\beta}} \Delta_2$, therefore, as $\wr \mathrm{Sync}(\Delta_1, \underline{M}) \wr \equiv \underline{M}$, we have that $\underline{M} \twoheadrightarrow_{\underline{\beta}} \wr \Delta_2 \wr$. However, $\underline{M}$ has only one underlining, and $\wr \Delta_2 \wr$ has none, therefore this is the single-step reduction $\underline{M} \longrightarrow_{\underline{\beta}} \wr \Delta_2 \wr$. Moreover, $|\underline{M}| \equiv \wr \Delta_1 \wr$, so we conclude, using Lemma 2.14, that $\wr \Delta_1 \wr \longrightarrow_{\beta} \wr \Delta_2 \wr$.

□

The following is a technical lemma used to prove confluence.

Lemma 2.16.
For any Δ and $\underline{M}$, if $\mathrm{Sync}(\Delta, \underline{M})$ is defined, then $\Delta \longrightarrow\!\!\!\!\rightarrow^{\|}_{\beta} \varphi(\mathrm{Sync}(\Delta, \underline{M}))$.

Proof. Reduce the redexes in Δ in the same order as is done by the function φ in $\mathrm{Sync}(\Delta, \underline{M})$. □

The above lemma justifies the following definition:

Definition 2.15 (Labelled synchronous reductions)**.**

1. We note $\Delta_1 \xrightarrow{\underline{M}} \Delta_2$ if $\Delta \longrightarrow\!\!\!\!\rightarrow^{\|}_{\beta} \varphi(\mathrm{Sync}(\Delta, \underline{M}))$;

2. We note $\underline{M} \sqsubseteq \underline{N}$ if $|\underline{M}| \equiv |\underline{N}|$ and the underlinings of M are also found in N;

3. If $|\underline{M}| \equiv |\underline{N}|$, we note $\underline{M} \sqcup \underline{N}$ for the λ-term whose underlinings are the union of those from M and N. Thus $\underline{M} \sqcup \underline{N}$ is the least upper-bound for the $\sqsubseteq$ relation.

We can prove that labeled synchronous reduction is confluent, more precisely, that if $\Delta_1 \xrightarrow{\underline{M}} \Delta_2$, and $\Delta_1 \xrightarrow{\underline{N}} \Delta_3$, then if we note Δ_4 such that $\Delta_1 \xrightarrow{\underline{M} \sqcup \underline{N}} \Delta_4$, then Δ_2 and Δ_3 reduce to Δ_4 under a labeled synchronous reduction. First, we need a technical lemma.

Lemma 2.17.
If $\Delta \xrightarrow{\underline{M}_1} \Delta_1$ and $\Delta \xrightarrow{\underline{M}_2} \Delta_2$, with $\underline{M}_1 \sqsubseteq \underline{M}_2$, then there exists some N such that $\Delta_1 \xrightarrow{\underline{N}} \Delta_2$. More precisely, we obtain N by applying in M_2 the $\underline{\beta}$-reductions that appear in M_1.

Proof. By induction on the shape of Δ, we can see that $\Delta_1 \xrightarrow{\underline{N}} \Delta_2$. The most interesting case is when Δ is some redex $(\lambda x{:}\sigma.\Delta')\,\Delta''$, and $\underline{M}_2$ is an underlined redex $(\underline{\lambda} x.\underline{M}'_2)\,\underline{M}''_2$. We pose Δ'_2 and Δ''_2 such that $\Delta' \xrightarrow{\underline{M}'_2} \Delta'_2$ and $\Delta'' \xrightarrow{\underline{M}''_2} \Delta''_2$. Moreover, $\Delta_2 \stackrel{def}{=} \Delta'_2[\Delta''_2/x]$. We have to subcases:

1. if $\underline{M}_1$ is also an underlined redex $(\underline{\lambda} x.\underline{M}'_1)\,\underline{M}''_1$, then $\Delta_1 \equiv \Delta'_1[\Delta''_1/x]$, where $\Delta' \xrightarrow{\underline{M}'_1} \Delta'_1$ and $\Delta'' \xrightarrow{\underline{M}''_1} \Delta''_1$. By induction hypothesis, we have that there is some $\underline{N}'$ and $\underline{N}''$ such that $\Delta'_1 \xrightarrow{\underline{N}'} \Delta'_2$ and $\Delta''_1 \xrightarrow{\underline{N}''} \Delta''_2$, and we pose $N \stackrel{def}{=} N'[N''/x]$;

2. if $\underline{M}_1$ is a redex $(\lambda x.\underline{M}'_1)\,\underline{M}''_1$, then $\Delta_1 \equiv (\lambda x{:}\sigma.\Delta'_1)\,\Delta'_2$, where $\Delta' \xrightarrow{\underline{M}'_1} \Delta'_1$ and $\Delta'' \xrightarrow{\underline{M}''_1} \Delta''_1$. By induction hypothesis, we have that there is some $\underline{N}'$ and $\underline{N}''$ such that $\Delta'_1 \xrightarrow{\underline{N}'} \Delta'_2$ and $\Delta''_1 \xrightarrow{\underline{N}''} \Delta''_2$, and we pose $N \stackrel{def}{=} (\underline{\lambda} x.N')\,N''$.

□

We now prove the confluence property for label-led synchronous reductions.

Lemma 2.18 (Confluence for labelled synchronous reduction)**.**
If $\Delta_1 \xrightarrow{\underline{M}_1} \Delta_2$ and $\Delta_1 \xrightarrow{\underline{M}_2} \Delta_3$, then we pose Δ_4 such that $\Delta_1 \xrightarrow{\underline{M}_1 \sqcup \underline{M}_2} \Delta_4$. There is some M_3 and M_4 such that $\Delta_2 \xrightarrow{\underline{M}_3} \Delta_4$ and $\Delta_3 \xrightarrow{\underline{M}_4} \Delta_4$.

$$\begin{tikzcd}
\Delta_1 \arrow[r, "\underline{M}_1"] \arrow[d, "\underline{M}_2"'] & \Delta_2 \arrow[d, dashed, "\underline{M}_3"'] \\
\Delta_3 \arrow[r, dashed, "\underline{M}_4"] & \Delta_4
\end{tikzcd}$$

Proof.
By Lemma 2.17. □

We can finally prove confluence.

Theorem 2.19 (Confluence for $\longrightarrow^{\parallel}_{\beta}$)**.**
If $\Delta_1 \twoheadrightarrow^{\parallel}_{\beta} \Delta_2$ and $\Delta_1 \twoheadrightarrow^{\parallel}_{\beta} \Delta_3$, then there is some Δ_4, such that $\Delta_2 \twoheadrightarrow^{\parallel}_{\beta} \Delta_4$ and $\Delta_3 \twoheadrightarrow^{\parallel}_{\beta} \Delta_4$.

Proof.
By Definition 2.14, one-step synchronous reduction corresponds to a labeled synchronous reduction, and by Lemma 2.16, labeled synchronous reduction corresponds to multistep synchronous reduction, therefore we can conclude with the usual diagram chase using Lemma 2.18, as suggested below:

$$\begin{tikzcd}
\Delta_1 \arrow[r, "\underline{M}_3"] \arrow[dd, "\underline{M}_1"'] & \cdot \arrow[r, "\underline{M}_4"] \arrow[d, dashed, "\underline{M}_7"'] & \cdot \arrow[r, "\underline{M}_5"] \arrow[d, dashed, "\underline{M}_9"'] & \Delta_2 \arrow[d, dashed, "\underline{M}_{11}"'] \\
\cdot \arrow[r, dashed, "\underline{M}_6"] \arrow[dd, "\underline{M}_2"'] & \cdot \arrow[r, dashed, "\underline{M}_8"] \arrow[d, dashed, "\underline{M}_{13}"'] & \cdot \arrow[r, dashed, "\underline{M}_{10}"] \arrow[d, dashed, "\underline{M}_{15}"'] & \cdot \arrow[d, dashed, "\underline{M}_{17}"'] \\
\Delta_3 \arrow[r, dashed, "\underline{M}_{12}"] & \cdot \arrow[r, dashed, "\underline{M}_{14}"] & \cdot \arrow[r, dashed, "\underline{M}_{16}"] & \Delta_4
\end{tikzcd}$$

□

2.3.3 Strong normalization of the generic Δ-calculus

The idea of the strong normalization proof of the generic Δ-calculus is to embed typable terms of the generic Δ-calculus into Church-style terms of a target system, which is the simply-typed λ-calculus with pairs, in a structure-preserving way (and forgetting all the essence side-conditions). The translation is sufficiently faithful so as to preserve the number of reductions, and so strong normalization for the generic Δ-calculus follows from strong normalization for simply-typed λ-calculus with pairs. A similar technique has been used in [52] to prove the strong normalization property of LF and in [21] to prove the strong normalization property of a subset of $\lambda^{\mathrm{CD}}_{\cap}$.

The target system has one atomic type called $\circ$, a special constant term $u_\circ$ of type $\circ$ and an infinite number of constants c_σ of type σ for any type of the target system. We denote by $\Gamma \vdash_{\times} M : \sigma$ a typing judgment in the target system.

Definition 2.16 (Forgetful mapping)**.**

– On intersection types.

$$
\begin{aligned}
|\mathsf{a}_i| &\stackrel{def}{=} \circ && \forall \mathsf{a}_i \in \mathbb{A} \\
|\sigma \cap \tau| &\stackrel{def}{=} |\sigma| \times |\tau| \\
|\sigma \to \tau| &\stackrel{def}{=} |\sigma| \to |\tau|
\end{aligned}
$$

– On Δ-terms.

$$
\begin{aligned}
|x|_\Gamma &\stackrel{def}{=} x \\
|u_M|_\Gamma &\stackrel{def}{=} u_\circ \\
|\lambda x{:}\sigma.\Delta|_\Gamma &\stackrel{def}{=} \lambda x.|\Delta|_{\Gamma,x:\sigma} \\
|\Delta_1\, \Delta_2|_\Gamma &\stackrel{def}{=} |\Delta_1|_\Gamma\, |\Delta_2|_\Gamma \\
|\langle \Delta_1, \Delta_2 \rangle|_\Gamma &\stackrel{def}{=} (|\Delta_1|_\Gamma, |\Delta_2|_\Gamma) \\
|\mathsf{pr}_i\, \Delta|_\Gamma &\stackrel{def}{=} \mathsf{pr}_i\, |\Delta|_\Gamma \\
|\Delta^\tau|_\Gamma &\stackrel{def}{=} c_{|\sigma|\to|\tau|}\, |\Delta|_\Gamma && \text{if } \Gamma \vdash^{\mathcal{T}}_{\mathcal{R}} \Delta : \sigma
\end{aligned}
$$

– The map can be easily extended to basis Γ.

The following technical lemma states some properties of the forgetful function.

Lemma 2.20.

1. If $\Gamma \vdash^{\mathcal{T}}_{\mathcal{R}} \Delta : \sigma$, then $|\Delta|_\Gamma$ is defined, and, for all $\Gamma' \supseteq \Gamma$, $|\Delta|_\Gamma \equiv |\Delta|_{\Gamma'}$;
2. $|\Delta_1[\Delta_2/x]|_\Gamma \equiv |\Delta_1|_\Gamma[|\Delta_2|_\Gamma/x]$;
3. If $\Delta_1 \longrightarrow \Delta_2$, then $|\Delta_1|_\Gamma \longrightarrow |\Delta_2|_\Gamma$;
4. If $\Gamma \vdash^{\mathcal{T}}_{\mathcal{R}} \Delta : \sigma$ then $|\Gamma| \vdash_\times |\Delta|_\Gamma : |\sigma|$.

Proof.

1. by induction on the derivation;
2. by induction on Δ_1. The only interesting part is $\Delta_1 \equiv \lambda y{:}\sigma.\Delta_1'$: by induction hypothesis, we have:

   $$|\Delta_1'[\Delta_2/x]|_{\Gamma,x:\sigma} \equiv |\Delta_1'|_{\Gamma,x:\sigma}[|\Delta_2|_{\Gamma,x:\sigma}/x]$$

   Therefore, we see that:

   $$|(\lambda y{:}\sigma.\Delta_1')[\Delta_2/x]|_\Gamma \equiv \lambda y{:}\sigma.|\Delta_1'[\Delta_2/x]|_{\Gamma,x:\sigma} \equiv \lambda y{:}\sigma.|\Delta_1'|_{\Gamma,x:\sigma}[|\Delta_2|_{\Gamma,x:\sigma}/x]$$

   However, from point (1), we know that:

   $$|\Delta_2|_{\Gamma,x:\sigma} \equiv |\Delta_2|_\Gamma$$

   and we conclude;

3. by induction on the context of the redex;

4. by induction on the derivation.

□

s Strong normalization follows easily from the above lemmas.

Theorem 2.21 (Strong normalization)**.**
If $\Gamma \vdash^{\mathcal{T}}_{\mathcal{R}} \Delta : \sigma$, then Δ is strongly normalizing.

Proof. Using Lemma 2.20 and the strong normalization of the simply typed λ-calculus with cartesian pairs. □

2.4 Church-style *vs.* Curry-style λ-calculus

In this section, we will study the intriguing relation between the λ-calculi $\lambda^{\mathcal{T}}_{\cap}$ and their corresponding Δ-calculi $\Delta^{\mathcal{T}}_{\mathcal{R}}$.

2.4.1 Relation between type assignment systems $\lambda^{\mathcal{T}}_{\cap}$ and typed systems $\Delta^{\mathcal{T}}_{\mathcal{R}}$

It is interesting to state some relations between type assignment systems *à la* Church and typed systems *à la* Curry. An interesting property is the one of isomorphism, namely the fact that whenever we assign a type σ to a pure λ-term M, the same type can be assigned to a Δ-term such that the essence of Δ is M. Conversely, for every assignment of σ to a Δ-term, a valid type assignment judgment of the same type for the essence of Δ can be derived.

Soundness, completeness and isomorphism between instances of the generic Δ-calculus and the corresponding intersection type assignment systems for the λ-calculus are defined as follows:

Definition 2.17 (Soundness, completeness and isomorphism)**.** Let $\Delta^{\mathcal{T}}_{\mathcal{R}}$ and $\lambda^{\mathcal{T}}_{\cap}$.

1. (Soundness, $\Delta^{\mathcal{T}}_{\mathcal{R}} \lhd \lambda^{\mathcal{T}}_{\cap}$). $\Gamma \vdash^{\mathcal{T}}_{\mathcal{R}} \Delta : \sigma$ implies $\Gamma \vdash^{\mathcal{T}}_{\cap} \wr \Delta \wr : \sigma$;

2. (Completeness, $\Delta^{\mathcal{T}}_{\mathcal{R}} \rhd \lambda^{\mathcal{T}}_{\cap}$). $\Gamma \vdash^{\mathcal{T}}_{\cap} M : \sigma$ implies there exists Δ such that $M \equiv \wr \Delta \wr$ and $\Gamma \vdash^{\mathcal{T}}_{\mathcal{R}} \Delta : \sigma$;

3. (Isomorphism, $\Delta^{\mathcal{T}}_{\mathcal{R}} \sim \lambda^{\mathcal{T}}_{\cap}$). $\Delta^{\mathcal{T}}_{\mathcal{R}} \rhd \lambda^{\mathcal{T}}_{\cap}$ and $\Delta^{\mathcal{T}}_{\mathcal{R}} \lhd \lambda^{\mathcal{T}}_{\cap}$.

The following properties and relations between typed and type assignment systems can be verified.

Theorem 2.22 (Soundness, completeness and isomorphism)**.** The following properties between Δ-calculi and type assignment systems $\lambda_\cap^{\mathcal{T}}$ can be verified.

$\Delta_{\mathcal{R}}^{\mathcal{T}}$	$\Delta_{\mathcal{R}}^{\mathcal{T}} \lhd \lambda_\cap^{\mathcal{T}}$	$\Delta_{\mathcal{R}}^{\mathcal{T}} \rhd \lambda_\cap^{\mathcal{T}}$
$\Delta_{\equiv}^{\mathrm{CD}}$	$\surd$	$\surd$
$\Delta_{\equiv}^{\mathrm{CDV}}$	$\surd$	$\surd$
$\Delta_{\equiv}^{\mathrm{CDS}}$	$\surd$	$\surd$
$\Delta_{\equiv}^{\mathrm{BCD}}$	$\surd$	$\surd$
$\Delta_{=_\beta}^{\mathrm{CD}}$	$\times$	$\surd$
$\Delta_{=_\beta}^{\mathrm{CDV}}$	$\times$	$\surd$
$\Delta_{=_\beta}^{\mathrm{CDS}}$	$\surd$	$\surd$
$\Delta_{=_\beta}^{\mathrm{BCD}}$	$\surd$	$\surd$
$\Delta_{=_{\beta\eta}}^{\mathrm{CDV}}$	$\times$	$\surd$
$\Delta_{=_{\beta\eta}}^{\mathrm{BCD}}$	$\times$	$\surd$

Proof.

($\lhd$) (a) Soundness for $\Delta_{\equiv}^{\mathcal{T}}$. Let Δ be such that $\Gamma \vdash_{\equiv}^{\mathcal{T}} \Delta : \sigma$. We proceed by induction on the derivation. All cases proceed straightforwardly since all rules of the type and subtype system $\vdash_{\equiv}^{\mathcal{T}}$ correspond exactly to the rules of the same name in the corresponding type assignment system $\vdash_\cap^{\mathcal{T}}$ and in the same type theory $\mathcal{T}$. Therefore $M \equiv \wr\Delta\wr$ can be easily be defined and derived with type σ.

(b) Soundness for $\Delta_{=_\beta}^{\{\mathrm{CDS},\mathrm{BCD}\}}$. Let $\mathcal{T} \in \{\mathcal{T}_{\mathrm{CDS}}, \mathcal{T}_{\mathrm{BCD}}\}$. We know, thanks to [12] (Figure 14.2), that the following rule is admissible for $\lambda_\cap^{\mathcal{T}}$:

$$\frac{\Gamma \vdash_\cap^{\mathcal{T}} M : \sigma \quad \Gamma \vdash_\cap^{\mathcal{T}} N : \tau \quad M =_\beta N}{\Gamma \vdash_\cap^{\mathcal{T}} M : \sigma \cap \tau}\ (\cap I)_{\mathrm{adm}}$$

Then the proof proceeds by induction on the derivation of $\Gamma \vdash_{=_\beta}^{\mathcal{T}} \Delta : \sigma$. The most important case is when the last used rule is $(\cap I)$: by induction we get $\Gamma \vdash_\cap^{\mathcal{T}} \wr\Delta_1\wr : \sigma$, and $\Gamma \vdash_\cap^{\mathcal{T}} \wr\Delta_2\wr : \tau$, and $\wr\Delta_1\wr =_\beta \wr\Delta_2\wr$, and, by the essence definition, $\wr\langle\Delta_1, \Delta_2\rangle\wr =_\beta \wr\Delta_1\wr$. Apply rule $(\cap I)_{\mathrm{adm}}$ and conclude with $\Gamma \vdash_\cap^{\mathcal{T}} \wr\Delta_1\wr : \sigma \cap \tau$.

($\not\lhd$) Loss of soundness for $\Delta_{=_\beta}^{\mathrm{CD}}$ and $\Delta_{=_\beta}^{\mathrm{CDV}}$.

Let $\mathcal{T} \in \{\mathcal{T}_{\mathrm{CD}}, \mathcal{T}_{\mathrm{CDV}}\}$. Let $\mathsf{S} \stackrel{def}{=} \lambda x.\lambda y.\lambda z.x\,z\,(y\,z)$ and $\mathsf{K} \stackrel{def}{=} \lambda x.\lambda y.x$. Let $\Delta \stackrel{def}{=} (\lambda x{:}(\sigma \to \tau \to \rho) \to ((\sigma \to \tau) \to \sigma \to \rho) \to \sigma \to \tau \to \rho.\lambda y{:}(\sigma \to \tau \to \rho) \to (\sigma \to \tau) \to \sigma \to \rho.\lambda z{:}\sigma \to \tau \to \rho.x\,z\,(y\,z))(\lambda x{:}\sigma \to \tau \to \rho.\lambda y{:}(\sigma \to \tau) \to \sigma \to \rho.x)(\lambda x{:}\sigma \to \tau \to \rho.\lambda y{:}\sigma \to \tau.\lambda z{:}\sigma.x\,z\,(y\,z))$. Δ is a simply-typed term of type $(\sigma \to \tau \to \rho) \to (\sigma \to \tau \to \rho)$, and its essence is $\wr\Delta\wr \equiv \mathsf{SKS}$. Consider the following counter-example:

$$\dfrac{\dfrac{\dfrac{\vdots}{\vdash_{=_\beta}^{\mathcal{T}} \Delta : (\sigma \to \tau \to \rho) \to (\sigma \to \tau \to \rho)} \quad \dfrac{x{:}\sigma \vdash_{=_\beta}^{\mathcal{T}} x : \sigma}{\vdash_{=_\beta}^{\mathcal{T}} \lambda x{:}\sigma.x : \sigma \to \sigma} \quad \mathsf{SKS} =_\beta \lambda x.x}{\vdash_{=_\beta}^{\mathcal{T}} \langle\Delta, \lambda x{:}\sigma.x\rangle : ((\sigma \to \tau \to \rho) \to (\sigma \to \tau \to \rho)) \cap (\sigma \to \sigma)}}{\vdash_{=_\beta}^{\mathcal{T}} \mathsf{pr}_2\,\langle\Delta, \lambda x{:}\sigma.x\rangle : \sigma \to \sigma}$$

The essence of the above term is $\mathsf{S}\,\mathsf{K}\,\mathsf{S}$, but, if σ is an atomic type:

$$\not\vdash^{\mathcal{T}}_{\cap} \mathsf{S}\,\mathsf{K}\,\mathsf{S} : \sigma \to \sigma$$

Loss of soundness in $\Delta^{\mathrm{CDV}}_{=\beta\eta}$ is proved via the following counterexample, where $\Gamma \stackrel{def}{=} \{x{:}(\sigma \to \tau) \cap \rho\}$.

$$\dfrac{\dfrac{\dfrac{\dfrac{\dfrac{\Gamma, y{:}\sigma \vdash^{\mathcal{T}_{\mathrm{CDV}}}_{=\beta\eta} x : (\sigma \to \tau) \cap \rho}{\Gamma, y{:}\sigma \vdash^{\mathcal{T}_{\mathrm{CDV}}}_{=\beta\eta} \mathsf{pr}_1\, x : \sigma \to \tau} \quad \Gamma, y{:}\sigma \vdash^{\mathcal{T}_{\mathrm{CDV}}}_{=\beta\eta} y : \sigma}{\Gamma, y{:}\sigma \vdash^{\mathcal{T}_{\mathrm{CDV}}}_{=\beta\eta} (\mathsf{pr}_1\, x)\, y : \tau}}{\Gamma \vdash^{\mathcal{T}_{\mathrm{CDV}}}_{=\beta\eta} \lambda y{:}\sigma.(\mathsf{pr}_1\, x)\, y : \sigma \to \tau} \quad \dfrac{\Gamma \vdash^{\mathcal{T}_{\mathrm{CDV}}}_{=\beta\eta} x : (\sigma \to \tau) \cap \rho}{\Gamma \vdash^{\mathcal{T}_{\mathrm{CDV}}}_{=\beta\eta} \mathsf{pr}_2\, x : \rho} \quad \lambda y.x\, y =_{\beta\eta} x}{\Gamma \vdash^{\mathcal{T}_{\mathrm{CDV}}}_{=\beta\eta} \langle \lambda y{:}\sigma.(\mathsf{pr}_1\, x)\, y, \mathsf{pr}_2\, x\rangle : (\sigma \to \tau) \cap \rho}}{\Gamma \vdash^{\mathcal{T}_{\mathrm{CDV}}}_{=\beta\eta} \mathsf{pr}_2\, \langle \lambda y{:}\sigma.(\mathsf{pr}_1\, x)\, y, \mathsf{pr}_2\, x\rangle : \rho}$$

The essence of $\mathsf{pr}_2\, \langle \lambda y{:}\sigma.(\mathsf{pr}_1\, x)\, y, \mathsf{pr}_2\, x\rangle$ is $\lambda y.x\, y$, but, if ρ is an atomic type:

$$x{:}(\sigma \to \tau) \cap \rho \not\vdash^{\mathcal{T}_{\mathrm{CDV}}}_{\cap} \lambda y.x\, y : \rho$$

Loss of soundness in $\Delta^{\mathrm{BCD}}_{=\beta\eta}$ is proved via the following counterexample:

$$\dfrac{\dfrac{\dfrac{\dfrac{\dfrac{x{:}\sigma, y{:}\mathtt{U} \vdash^{\mathcal{T}_{\mathrm{BCD}}}_{=\beta\eta} x : \sigma \quad \sigma \leqslant_{\mathcal{T}} \mathtt{U} \to \mathtt{U}}{x{:}\sigma, y{:}\mathtt{U} \vdash^{\mathcal{T}_{\mathrm{BCD}}}_{=\beta\eta} x^{\mathtt{U}\to\mathtt{U}} : \mathtt{U} \to \mathtt{U}} \quad x{:}\sigma, y{:}\mathtt{U} \vdash^{\mathcal{T}_{\mathrm{BCD}}}_{=\beta\eta} y : \mathtt{U}}{x{:}\sigma, y{:}\mathtt{U} \vdash^{\mathcal{T}_{\mathrm{BCD}}}_{=\beta\eta} x^{\mathtt{U}\to\mathtt{U}}\, y : \mathtt{U}}}{x{:}\sigma \vdash^{\mathcal{T}_{\mathrm{BCD}}}_{=\beta\eta} \lambda y{:}\mathtt{U}.x^{\mathtt{U}\to\mathtt{U}}\, y : \mathtt{U} \to \mathtt{U}} \quad x{:}\sigma \vdash^{\mathcal{T}_{\mathrm{BCD}}}_{=\beta\eta} x : \sigma \quad \lambda y.x\, y =_{\beta\eta} x}{x{:}\sigma \vdash^{\mathcal{T}_{\mathrm{BCD}}}_{=\beta\eta} \langle \lambda y{:}\mathtt{U}.x^{\mathtt{U}\to\mathtt{U}}\, y, x\rangle : (\mathtt{U} \to \mathtt{U}) \cap \sigma}}{x{:}\sigma \vdash^{\mathcal{T}_{\mathrm{BCD}}}_{=\beta\eta} \mathsf{pr}_2\, \langle \lambda y{:}\mathtt{U}.x^{\mathtt{U}\to\mathtt{U}}\, y, x\rangle : \sigma}$$

The essence of $\mathsf{pr}_2\, \langle \lambda y{:}\mathtt{U}.x^{\mathtt{U}\to\mathtt{U}}\, y, x\rangle$ is $\lambda y.x\, y$, but, if σ is an atomic type (different than $\mathtt{U}$):

$$x{:}\sigma \not\vdash^{\mathcal{T}_{\mathrm{BCD}}}_{\cap} \lambda y.x\, y : \sigma$$

($\triangleright$) Let M be such that $\Gamma \vdash^{\mathcal{T}}_{\cap} M : \sigma$ for a given Γ. We proceed by induction on the derivation. All cases proceed straightforwardly since all rules of the type and subtype assignment system $\vdash^{\mathcal{T}}_{\cap}$ correspond exactly to the rules of the same name in the corresponding typed system $\vdash^{\mathcal{T}}_{\mathcal{R}}$ and in the same type theory $\mathcal{T}$. Therefore a Δ-term can be easily be constructed and derived with type σ.

□

The last theorem characterizes the class of strongly normalizing Δ-terms.

Theorem 2.23 (Characterization)**.** Every strongly normalizing λ-term can be type-annotated so as to be the essence of a typable Δ-term.

Proof. We know that every strongly normalizing λ-term M is typable in $\lambda^{\mathcal{T}}_{\cap}$. By Theorem 2.22 we have that $\Delta^{\mathcal{T}}_{\mathcal{R}} \triangleright \lambda^{\mathcal{T}}_{\cap}$, therefore there exists some typable Δ, such that $M \equiv \wr\, \Delta\, \wr$. □

We can finally state decidability of type checking (TC) and type reconstruction (TR).

Theorem 2.24 (Decidability of type checking and type reconstruction)**.**

$\Delta^{\mathcal{T}}_{\mathcal{R}}$	TC/TR
$\Delta^{\mathrm{CD}}_{\equiv}$	$\surd$
$\Delta^{\mathrm{CDV}}_{\equiv}$	$\surd$
$\Delta^{\mathrm{CDS}}_{\equiv}$	$\surd$
$\Delta^{\mathrm{BCD}}_{\equiv}$	$\surd$
$\Delta^{\mathrm{CD}}_{=_\beta}$	$\surd$
$\Delta^{\mathrm{CDV}}_{=_\beta}$	$\surd$
$\Delta^{\mathrm{CDS}}_{=_\beta}$	$\times$
$\Delta^{\mathrm{BCD}}_{=_\beta}$	$\times$
$\Delta^{\mathrm{CDV}}_{=_{\beta\eta}}$	$\surd$
$\Delta^{\mathrm{BCD}}_{=_{\beta\eta}}$	$\times$

Proof. Both type checking and type reconstruction can be proved by induction on the structure of Δ, using the decidability of $\mathcal{T}_{\mathrm{BCD}}$ proved by Hindley [54] (see also [65]). By Theorem 2.22, the essences of all the Δ-terms, which are typable in $\Delta^{\mathrm{CD}}_{=_\beta}$, $\Delta^{\mathrm{CDV}}_{=_\beta}$, or $\Delta^{\mathrm{CDV}}_{=_{\beta\eta}}$, are typable in $\lambda^{\mathrm{CD}}_{\cap}$ or $\lambda^{\mathrm{CDV}}_{\cap}$, therefore they are strongly normalizing. As a consequence, the side-condition $\wr\,\Delta_1\,\wr\ \mathcal{R}\ \wr\,\Delta_2\,\wr$ is decidable for $\Delta^{\mathrm{CD}}_{=_\beta}$, $\Delta^{\mathrm{CDV}}_{=_\beta}$, and $\Delta^{\mathrm{CDV}}_{=_{\beta\eta}}$ and so type reconstruction and type checking are decidable too.

Type reconstruction and type checking are not decidable in $\Delta^{\mathrm{CDS}}_{=_\beta}$, $\Delta^{\mathrm{BCD}}_{=_\beta}$, and $\Delta^{\mathrm{BCD}}_{=_{\beta\eta}}$, because $\langle u_M, u_N\rangle$ is typable if and only if $M =_\beta N$ (resp. $M =_{\beta\eta} N$). However, M and N are arbitrary pure λ-terms, and both β-equality and $\beta\eta$-equality are undecidable. $\square$

2.4.2 Subtyping and explicit coercions

The typing rule $(\leqslant_{\mathcal{T}})$ in the general typed system introduces type coercions: once a type coercion is introduced, it cannot be eliminated, so *de facto* freezing a Δ-term inside an explicit coercion. Tannen *et al.* [91] showed a translation of a judgment derivation from a "Source" system with subtyping (Cardelli's Fun [24]) into an equivalent judgment derivation in a "Target" system without subtyping (Girard system F with records and recursion). In the same spirit, we present a translation that removes all explicit coercions. Intuitively, the translation proceeds as follows: every derivation ending with rule:

$$\frac{\Gamma \vdash^{\mathcal{T}}_{\mathcal{R}} \Delta : \sigma \quad \sigma \leqslant_{\mathcal{T}} \tau}{\Gamma \vdash^{\mathcal{T}}_{\mathcal{R}} \Delta^{\tau} : \tau}\ (\leqslant_{\mathcal{T}})$$

is translated into the following (coercion-free) derivation

$$\frac{\Gamma \vdash^{\mathcal{T}}_{\mathcal{R}'} \|\sigma \leqslant_{\mathcal{T}} \tau\| : \sigma \to \tau \quad \Gamma \vdash^{\mathcal{T}}_{\mathcal{R}'} \|\Delta\|_{\Gamma} : \sigma}{\Gamma \vdash^{\mathcal{T}}_{\mathcal{R}'} \|\sigma \leqslant_{\mathcal{T}} \tau\|\ \|\Delta\|_{\Gamma} : \tau}\ (\to E)$$

where $\mathcal{R}'$ is a suitable relation such that $\mathcal{R} \sqsubseteq \mathcal{R}'$. Note that changing of the type theory is necessary to guarantee well-typedness in the translation of strong pairs. Summarizing, we provide a type preserving translation of a Δ-term into a coercion-free Δ-term such that $\wr\,\Delta\,\wr =_{\beta\eta} \wr\,\Delta'\,\wr$.

The following example illustrates some trivial compilations of axioms and rule schemes of Figure 2.1.

Example 2.9 (Translation of axioms and rule schemes of Figure 2.1)**.**

(refl) the judgment $x{:}\sigma \vdash^{\mathcal{T}}_{\mathcal{R}} \langle x, x^{\sigma}\rangle : \sigma \cap \sigma$ is translated to a coercion-free judgment

$$x{:}\sigma \vdash^{\mathcal{T}}_{=_\beta} \langle x, (\lambda y{:}\sigma.y)\, x\rangle : \sigma \cap \sigma$$

(incl) the judgment $x{:}\sigma\cap\tau \vdash^{\mathcal{T}}_{\mathcal{R}} \langle x, x^{\tau}\rangle : (\sigma\cap\tau)\cap\tau$ is translated to a coercion-free judgment

$$x{:}\sigma \cap \tau \vdash^{\mathcal{T}}_{=_\beta} \langle x, (\lambda y{:}\sigma \cap \tau.\mathsf{pr}_2\, y)\, x\rangle : (\sigma \cap \tau) \cap \tau$$

(glb) the judgment $x{:}\sigma \vdash^{\mathcal{T}}_{\mathcal{R}} \langle x, x^{\sigma\cap\sigma}\rangle : \sigma \cap (\sigma \cap \sigma)$ is translated to a coercion-free judgment

$$x{:}\sigma \vdash^{\mathcal{T}}_{=_\beta} \langle x, (\lambda y{:}\sigma.\langle y, y\rangle)\, x\rangle : \sigma \cap (\sigma \cap \sigma)$$

($\mathtt{U}_{top}$) the judgment $x{:}\sigma \vdash^{\mathcal{T}}_{\mathcal{R}} \langle x, x^{\mathtt{U}}\rangle : \sigma \cap \mathtt{U}$ is translated to a coercion-free judgment

$$x{:}\sigma \vdash^{\mathcal{T}}_{=_\beta} \langle x, (\lambda y{:}\sigma.u_y)\, x\rangle : \sigma \cap \mathtt{U}$$

($\mathtt{U}_{\to}$) the judgment $x{:}\mathtt{U} \vdash^{\mathcal{T}}_{\mathcal{R}} \langle x, x^{\sigma\to\mathtt{U}}\rangle : \mathtt{U}\cap(\sigma \to \mathtt{U})$ is translated to a coercion-free judgment

$$x{:}\mathtt{U} \vdash^{\mathcal{T}}_{=_{\beta\eta}} \langle x, (\lambda f{:}\mathtt{U}.\lambda y{:}\sigma.u_{(f\,y)})\, x\rangle : \mathtt{U} \cap (\sigma \to \mathtt{U})$$

($\to\cap$) the judgment $x{:}(\sigma \to \tau) \cap (\sigma \to \rho) \vdash^{\mathcal{T}}_{\mathcal{R}} x^{\sigma\to\tau\cap\rho} : \sigma \to \tau \cap \rho$ is translated to a coercion-free judgment

$$x{:}\sigma' \vdash^{\mathcal{T}}_{=_{\beta\eta}} (\lambda f{:}\sigma'.\lambda y{:}\sigma.\langle(\mathsf{pr}_1\, f)\, y, (\mathsf{pr}_2\, f)\, y\rangle)\, x : \sigma \to \tau \cap \rho$$

where $\sigma' = (\sigma \to \tau) \cap (\sigma \to \rho)$

($\to$) the judgment $x{:}\sigma \to \tau \cap \rho \vdash^{\mathcal{T}}_{\mathcal{R}} \langle x, x^{\sigma\cap\rho\to\tau}\rangle : (\sigma \to \tau \cap \rho) \cap (\sigma \cap \rho \to \tau)$ is translated to a coercion-free judgment

$$x{:}\sigma \to \tau \cap \rho \vdash^{\mathcal{T}}_{=_{\beta\eta}} \langle x, (\lambda f{:}\sigma \to \tau \cap \rho.\lambda y{:}\sigma \cap \rho.\mathsf{pr}_1\, (f\, (\mathsf{pr}_1\, y)))\, x\rangle : \sigma''$$

where $\sigma'' = (\sigma \to \tau \cap \rho) \cap (\sigma \cap \rho \to \tau)$

(trans) the judgment $x{:}\sigma \vdash^{\mathcal{T}}_{\mathcal{R}} \langle x, (x^{\mathtt{U}})^{\sigma\to\mathtt{U}}\rangle : \sigma \cap (\sigma \to \mathtt{U})$ is translated to a coercion-free judgment

$$x{:}\sigma \vdash^{\mathcal{T}}_{=_{\beta\eta}} \langle x, (\lambda f{:}\mathtt{U}.\lambda y{:}\sigma.u_{(f\,y)})\, ((\lambda y{:}\sigma.u_y)\, x)\rangle : \sigma \cap (\sigma \to \mathtt{U})$$

The next definition introduces two maps translating subtype judgments into explicit coercions functions and Δ-terms into coercion-free Δ-terms.

Definition 2.18 (Translations $\|-\|$ and $\|-\|_\Gamma$)**.**

1. The minimal type theory $\leqslant_{\min}$ and the extra axioms and schemes are translated as follows:

$$
\begin{array}{lrcl}
\text{(refl)} & \|\sigma \leqslant_{\mathcal{T}} \sigma\| & \stackrel{def}{=} & \vdash^{\mathcal{T}}_{=_\beta} \lambda x{:}\sigma.x : \sigma \to \sigma \\
\text{(incl}_1\text{)} & \|\sigma \cap \tau \leqslant_{\mathcal{T}} \sigma\| & \stackrel{def}{=} & \vdash^{\mathcal{T}}_{=_\beta} \lambda x{:}\sigma \cap \tau.\mathsf{pr}_1\, x : \sigma \cap \tau \to \sigma \\
\text{(incl}_2\text{)} & \|\sigma \cap \tau \leqslant_{\mathcal{T}} \tau\| & \stackrel{def}{=} & \vdash^{\mathcal{T}}_{=_\beta} \lambda x{:}\sigma \cap \tau.\mathsf{pr}_2\, x : \sigma \cap \tau \to \tau \\
\text{(glb)} & \left\| \dfrac{\rho \leqslant_{\mathcal{T}} \sigma \quad \rho \leqslant_{\mathcal{T}} \tau}{\rho \leqslant_{\mathcal{T}} \sigma \cap \tau} \right\| & \stackrel{def}{=} & \vdash^{\mathcal{T}}_{=_\beta} \lambda x{:}\rho.\langle \|\rho \leqslant_{\mathcal{T}} \sigma\|\, x, \|\rho \leqslant_{\mathcal{T}} \tau\|\, x\rangle : \rho \to \sigma \cap \tau \\
\text{(trans)} & \left\| \dfrac{\sigma \leqslant_{\mathcal{T}} \tau \quad \tau \leqslant_{\mathcal{T}} \rho}{\sigma \leqslant_{\mathcal{T}} \rho} \right\| & \stackrel{def}{=} & \vdash^{\mathcal{T}}_{=_\beta} \lambda x{:}\sigma.\, \|\tau \leqslant_{\mathcal{T}} \rho\|\ (\|\sigma \leqslant_{\mathcal{T}} \tau\|\ x) : \sigma \to \rho \\
(\mathtt{U}_{top}) & \|\sigma \leqslant_{\mathcal{T}} \mathtt{U}\| & \stackrel{def}{=} & \vdash^{\mathcal{T}}_{=_\beta} \lambda x{:}\sigma.u_x : \sigma \to \mathtt{U} \\
(\mathtt{U}_{\to}) & \|\mathtt{U} \leqslant_{\mathcal{T}} \sigma \to \mathtt{U}\| & \stackrel{def}{=} & \vdash^{\mathcal{T}}_{=_{\beta\eta}} \lambda f{:}\mathtt{U}.\lambda x{:}\sigma.u_{(f\,x)} : \mathtt{U} \to (\sigma \to \mathtt{U})
\end{array}
$$

Let $\xi_1 \stackrel{def}{=} (\sigma \to \tau) \cap (\sigma \to \rho)$ and $\xi_2 \stackrel{def}{=} \sigma \to \tau \cap \rho$

$$
(\to\cap) \qquad \|\xi_1 \leqslant_{\mathcal{T}} \xi_2\| \stackrel{def}{=} \vdash^{\mathcal{T}}_{=_{\beta\eta}} \lambda f{:}\xi_1.\lambda x{:}\sigma.\langle (\mathsf{pr}_1\, f)\, x, (\mathsf{pr}_2\, f)\, x\rangle : \xi_1 \to \xi_2
$$

Let $\xi_1 \stackrel{def}{=} \sigma_1 \to \tau_1$ and $\xi_2 \stackrel{def}{=} \sigma_2 \to \tau_2$

$$
(\to) \qquad \left\| \frac{\sigma_2 \leqslant_{\mathcal{T}} \sigma_1 \quad \tau_1 \leqslant_{\mathcal{T}} \tau_2}{\sigma_1 \to \tau_1 \leqslant_{\mathcal{T}} \sigma_2 \to \tau_2} \right\| \stackrel{def}{=} \vdash^{\mathcal{T}}_{=_{\beta\eta}} \lambda f{:}\xi_1.\lambda x{:}\sigma_2.\, \|\tau_1 \leqslant_{\mathcal{T}} \tau_2\|\ (f\,(\|\sigma_2 \leqslant_{\mathcal{T}} \sigma_1\|\, x)) : \xi_1 \to \xi_2
$$

2. The translation $\|-\|_\Gamma$ is defined on Δ as follows:

$$
\begin{array}{rcll}
\|u_M\|_\Gamma & \stackrel{def}{=} & u_M & \\
\|x\|_\Gamma & \stackrel{def}{=} & x & \\
\|\lambda x{:}\sigma.\Delta\|_\Gamma & \stackrel{def}{=} & \lambda x{:}\sigma.\, \|\Delta\|_{\Gamma, x:\sigma} & \\
\|\Delta_1\, \Delta_2\|_\Gamma & \stackrel{def}{=} & \|\Delta_1\|_\Gamma\ \|\Delta_2\|_\Gamma & \\
\|\langle \Delta_1, \Delta_2\rangle\|_\Gamma & \stackrel{def}{=} & \langle \|\Delta_1\|_\Gamma, \|\Delta_2\|_\Gamma\rangle & \\
\|\mathsf{pr}_i\, \Delta\|_\Gamma & \stackrel{def}{=} & \mathsf{pr}_i\ \|\Delta\|_\Gamma & i \in \{1,2\} \\
\|\Delta^\tau\|_\Gamma & \stackrel{def}{=} & \|\sigma \leqslant_{\mathcal{T}} \tau\|\ \|\Delta\|_\Gamma & \text{if } \Gamma \vdash^{\mathcal{T}}_{\mathcal{R}} \Delta : \sigma.
\end{array}
$$

By looking at the above translation functions we can see that if $\Gamma \vdash^{\mathcal{T}}_{\mathcal{R}} \Delta : \sigma$, then $\|\Delta\|_\Gamma$ is defined and it is coercion-free.

The following lemma states that a coercion function is always typable in $\Delta^{\mathcal{T}}_{=_{\beta\eta}}$, that it is essentially the identity and that, without using the rule schemes $(\to\cap)$, $(\mathtt{U}_{\to})$, and $(\to)$ the translation can even be derivable in $\Delta^{\mathcal{T}}_{=_\beta}$.

Lemma 2.25 (Essence of a coercion is an identity)**.**

1. If $\sigma \leqslant_{\mathcal{T}} \tau$, then $\vdash^{\mathcal{T}}_{=_{\beta\eta}} \|\sigma \leqslant_{\mathcal{T}} \tau\| : \sigma \to \tau$ and $\wr\, \|\sigma \leqslant_{\mathcal{T}} \tau\|\, \wr =_{\beta\eta} \lambda x.x$;

Source	Target
$\Delta^{\mathrm{CD}}_{\equiv}$	$\Delta^{\mathrm{CD}}_{=_\beta}$
$\Delta^{\mathrm{CDV}}_{\equiv}$	$\Delta^{\mathrm{CDV}}_{=_{\beta\eta}}$
$\Delta^{\mathrm{CDS}}_{\equiv}$	$\Delta^{\mathrm{CDS}}_{=_\beta}$
$\Delta^{\mathrm{BCD}}_{\equiv}$	$\Delta^{\mathrm{BCD}}_{=_{\beta\eta}}$
$\Delta^{\mathrm{CD}}_{=_\beta}$	$\Delta^{\mathrm{CD}}_{=_\beta}$
$\Delta^{\mathrm{CDV}}_{=_\beta}$	$\Delta^{\mathrm{CDV}}_{=_{\beta\eta}}$
$\Delta^{\mathrm{CDS}}_{=_\beta}$	$\Delta^{\mathrm{CDS}}_{=_\beta}$
$\Delta^{\mathrm{BCD}}_{=_\beta}$	$\Delta^{\mathrm{BCD}}_{=_{\beta\eta}}$
$\Delta^{\mathrm{CDV}}_{=_{\beta\eta}}$	$\Delta^{\mathrm{CDV}}_{=_{\beta\eta}}$
$\Delta^{\mathrm{BCD}}_{=_{\beta\eta}}$	$\Delta^{\mathrm{BCD}}_{=_{\beta\eta}}$

Figure 2.7: On the left: source systems. On the right: target systems without the $(\leqslant_{\mathcal{T}})$ rule

2. If $\sigma \leqslant_{\mathcal{T}} \tau$ without using the rule schemes $(\to\cap)$, $(\mathtt{U}_{\to})$, and $(\to)$, then $\vdash^{\mathcal{T}}_{=_\beta} \|\sigma \leqslant_{\mathcal{T}} \tau\| : \sigma \to \tau$ and $\wr \|\sigma \leqslant_{\mathcal{T}} \tau\| \wr =_\beta \lambda x.x$.

Proof. The proofs proceed in both parts by induction on the derivation of $\sigma \leqslant_{\mathcal{T}} \tau$. For instance, in case of (glb), we can verify that $\vdash^{\mathcal{T}}_{=_\beta} \lambda x{:}\rho.\langle \|\rho \leqslant_{\mathcal{T}} \sigma\|\, x, \|\rho \leqslant_{\mathcal{T}} \tau\|\, x\rangle : \rho \to \sigma\cap\tau$ using the induction hypotheses that $\|\rho \leqslant_{\mathcal{T}} \sigma\|$ (resp. $\|\rho \leqslant_{\mathcal{T}} \tau\|$) has type $\rho \to \sigma$ (resp. $\rho \to \tau$) and has an essence convertible to $\lambda x.x$. □

We can now prove the coherence of the translation as follows:

Theorem 2.26 (Coherence)**.** If $\Gamma \vdash^{\mathcal{T}}_{\mathcal{R}} \Delta : \sigma$, then $\Gamma \vdash^{\mathcal{T}}_{\mathcal{R}'} \|\Delta\|_\Gamma : \sigma$ and $\wr \|\Delta\|_\Gamma \wr\, \mathcal{R}' \wr \Delta \wr$, where $\Delta^{\mathcal{T}}_{\mathcal{R}}$ and $\Delta^{\mathcal{T}}_{\mathcal{R}'}$ are respectively the source and target intersection typed systems given in Figure 2.7.

Proof. By induction on the derivation. We illustrate the most important case, namely when the last type rule is $(\leqslant_{\mathcal{T}})$. In this case $\|\Delta^\tau\|_\Gamma$ is translated to $\|\sigma \leqslant_{\mathcal{T}} \tau\|\ \|\Delta\|_\Gamma$. By induction hypothesis we have that $\Gamma \vdash^{\mathcal{T}}_{\mathcal{R}} \Delta : \sigma$, and by Lemma 2.25 we have that $\Gamma \vdash^{\mathcal{T}}_{\mathcal{R}'} \|\sigma \leqslant_{\mathcal{T}} \tau\| : \sigma \to \tau$; therefore $\Gamma \vdash^{\mathcal{T}}_{\mathcal{R}'} \|\Delta^\tau\|_\Gamma : \tau$. Moreover, we know that $\wr \|\sigma \leqslant_{\mathcal{T}} \tau\| \wr\, \mathcal{R}'$ $\lambda x.x$, and this gives $\wr \|\Delta^\tau\|_\Gamma \wr\, \mathcal{R}' \wr \|\Delta\|_\Gamma \wr$. Again by induction hypothesis we have that $\wr \|\Delta\|_\Gamma \wr\, \mathcal{R}' \wr \Delta \wr$, and this gives the thesis $\wr \|\Delta^\tau\|_\Gamma \wr\, \mathcal{R}' \wr \Delta^\tau \wr$. □

Chapter 3

Adding union types

This chapter is a contribution to the study of typed λ-calculi *à la* Church in presence of intersection and union types. We inspect the relationship between pure λ-calculus and its corresponding Δ-calculus. We present and explore the relationships between the following three formal systems:

- λ^{BDdL}, the type assignment system with intersection and union types for pure λ-calculus with subtyping with the type theory Ξ, as defined in [7]: type assignment judgments have the shape $\Gamma \vdash M : \sigma$;

- $\lambda@^{\text{BDdL}}$, an extension of the typed λ-calculus with strong pairs and strong sums, as defined in [40], with subtyping and explicit coercions: type judgments have the shape $\Gamma \vdash M@\Delta : \sigma$, where Δ is a typed λ-term enriched with strong pairs and strong sums;

- Δ^{BDdL}, an extension of the $\Delta^{\text{BCD}}_{\equiv}$ of Chapter 2 with *ad hoc* formulæ and inference rules for subtyping and explicit coercions: type judgments have the shape $\Gamma \vdash \Delta : \sigma$.

Intuitively, Δ denotes a proof for a type assignment derivation for M; from an operational point of view, reductions in pure M and typed Δ must be synchronized by suitable synchronous reduction rules in order to preserve the reduction of subjects. From a typing point of view, the type rules of $\lambda@^{\text{BDdL}}$ should encode the proof-functional nature of strong intersection and strong union, *i.e.* the fact that in a strong pair (or strong sums) the two Δ relate to the same M. Thanks to an erasing function $\wr - \wr$ translating typed Δ into pure M, we could reason only on Δ^{BDdL} assigning types to Δ.

In summary, this chapter extends Chapter 2 with union types. The important points of this chapter are as follows:

- we define Δ^{BDdL} obtained by extending the generic Δ-calculus with union types and by fixing a single type theory $\mathcal{T}$ and equivalence relation $\mathcal{R}$, while keeping decidability of type checking, and showing the isomorphism with the type assignment system λ^{BDdL} of [7]. As such, Δ-terms are typed λ-terms enriched with both strong pairs and and strong sums;

- we define the typed λ-calculus $\lambda@^{\text{BDdL}}$ which is a decorated version of Δ^{BDdL}. Terms of $\lambda@^{\text{BDdL}}$ have the form $M@\Delta$ where M is a pure λ-term;

$$\frac{}{\Gamma \vdash M : \mathtt{U}}\ (\mathtt{U}) \qquad \frac{\Gamma \vdash M : \sigma \quad \sigma \leqslant \tau}{\Gamma \vdash M : \tau}\ (\leqslant) \qquad \frac{x{:}\sigma \in \Gamma}{\Gamma \vdash x : \sigma}\ (Var)$$

$$\frac{\Gamma, x{:}\sigma_1 \vdash M : \sigma_2}{\Gamma \vdash \lambda x.M : \sigma_1 \to \sigma_2}\ (\to I) \qquad \frac{\Gamma \vdash M : \sigma_1 \to \sigma_2 \quad \Gamma \vdash N : \sigma_1}{\Gamma \vdash M\,N : \sigma_2}\ (\to E)$$

$$\frac{\Gamma \vdash M : \sigma_1 \quad \Gamma \vdash M : \sigma_2}{\Gamma \vdash M : \sigma_1 \cap \sigma_2}\ (\cap I) \qquad \frac{\Gamma \vdash M : \sigma_1 \cap \sigma_2 \quad i = 1,2}{\Gamma \vdash M : \sigma_i}\ (\cap E_i)$$

$$\frac{\Gamma \vdash M : \sigma_i \quad i = 1,2}{\Gamma \vdash M : \sigma_1 \cup \sigma_2}\ (\cup I_i) \qquad \frac{\begin{array}{c}\Gamma, x{:}\sigma_1 \vdash M : \sigma_3 \\ \Gamma, x{:}\sigma_2 \vdash M : \sigma_3 \quad \Gamma \vdash N : \sigma_1 \cup \sigma_2\end{array}}{\Gamma \vdash M[N/x] : \sigma_3}\ (\cup E)$$

Figure 3.1: Intersection and union type assignment system λ^{BDdL} [7]

– we prove the isomorphism property between Δ^{BDdL} and $\lambda@^{\text{BDdL}}$ (Theorem 3.1), as well as the other usual properties, such as subject reduction, Church-Rosser, strong normalization, unicity of typing, and decidability of type reconstruction and of type checking (Theorem 3.2).

This chapter is organized as follows: in Section 3.1, we present the syntax and semantics of Δ^{BDdL} and $\lambda@^{\text{BDdL}}$. In Section 3.2, we study the metatheory of Δ^{BDdL} and $\lambda@^{\text{BDdL}}$.

3.1 Syntax and semantics of Δ^{BDdL} and $\lambda@^{\text{BDdL}}$

The syntax of σ, M, Δ, and the derived $M@\Delta$ are defined using a set of atomic types $\mathbb{A}_\infty$ and the following three syntactic categories:

$$\begin{array}{rcl} \sigma & ::= & \mathtt{U} \mid \phi \mid \sigma \to \sigma \mid \sigma \cap \sigma \mid \sigma \cup \sigma \\ M & ::= & x \mid \lambda x.M \mid M\,M \\ \Delta & ::= & u_M \mid x \mid \lambda x{:}\sigma.\Delta \mid \Delta\,\Delta \mid \langle \Delta, \Delta \rangle \mid [\Delta, \Delta] \mid \mathsf{pr}_1\,\Delta \mid \mathsf{pr}_2\,\Delta \mid \mathsf{in}_1^\sigma\,\Delta \mid \mathsf{in}_2^\sigma\,\Delta \mid \Delta^\sigma \end{array}$$

where ϕ denotes atomic types belonging in $\mathbb{A}_\infty$, and $\mathtt{U}$ denotes a special type that is inhabited by all pure λ-terms and all constants u_M. The Δ-expression $\langle \Delta, \Delta \rangle$ denotes the strong pair, while $[\Delta, \Delta]$ denotes the strong sum, with the respective projections and injections, respectively. Finally, Δ^σ denotes the explicit coercion of Δ with the type σ.

The untyped reduction semantics for the calculus *à la* Curry λ^{BDdL} corresponds to ordinary β-reduction, even if subject reduction holds only in presence of the "Gross-Knuth" parallel reduction (see Definition 13.2.7 in [9]), where all redexes in M are contracted simultaneously. Reduction for the calculus *à la* Church $\lambda@^{\text{BDdL}}$ is delicate because it must keep *synchronized* the untyped reduction of M with the typed reduction of Δ: it is defined in Section 5 of [40]. Reductions in Δ^{BDdL} are defined from these three notions of reductions:

$$\begin{array}{rcll} (\lambda x{:}\sigma.\Delta_1)\,\Delta_2 & \mapsto_\beta & \Delta_1[\Delta_2/x] & & (\beta) \\ \mathsf{pr}_i\,\langle \Delta_1, \Delta_2 \rangle & \mapsto_{\mathsf{pr}_i} & \Delta_i & & (\mathsf{pr}_i) \\ [\lambda x{:}\sigma_1.\Delta_1, \lambda x{:}\sigma_2.\Delta_2]\,\mathsf{in}_i^\tau\,\Delta_3 & \mapsto_{\mathsf{in}_i} & \Delta_i[\Delta_3/x] & i \in \{1,2\} & (\mathsf{in}_i) \end{array}$$

$$\frac{}{\Gamma \vdash M@u_M : \mathtt{U}}\ (\mathtt{U}) \qquad \frac{x{:}\sigma \in \Gamma}{\Gamma \vdash x : \sigma}\ (Var)$$

$$\frac{\Gamma, x{:}\sigma_1 \vdash M@\Delta : \sigma_2}{\Gamma \vdash \lambda x.M@\lambda x{:}\sigma_1.\Delta : \sigma_1 \to \sigma_2}\ (\to I) \qquad \frac{\Gamma \vdash M@\Delta_1 : \sigma_1 \to \sigma_2 \quad \Gamma \vdash N@\Delta_2 : \sigma_1}{\Gamma \vdash M\,N@\Delta_1\,\Delta_2 : \sigma_2}\ (\to E)$$

$$\frac{\Gamma \vdash M@\Delta_1 : \sigma_1 \quad \Gamma \vdash M@\Delta_2 : \sigma_2}{\Gamma \vdash M@\langle \Delta_1, \Delta_2 \rangle : \sigma_1 \cap \sigma_2}\ (\cap I) \qquad \frac{\Gamma \vdash M@\Delta : \sigma_1 \cap \sigma_2 \quad i \in \{1,2\}}{\Gamma \vdash M@\mathsf{pr}_i\,\Delta : \sigma_i}\ (\cap E_i)$$

$$\frac{\Gamma \vdash M@\Delta : \sigma_i \quad \{i,j\} = \{1,2\}}{\Gamma \vdash M@\mathsf{in}_i^{\sigma_j}\,\Delta : \sigma_1 \cup \sigma_2}\ (\cup I_i) \qquad \frac{\Gamma \vdash M@\Delta : \sigma \quad \sigma \leqslant \tau}{\Gamma \vdash M@\Delta^{\tau} : \tau}\ (\leqslant)$$

$$\frac{\Gamma, x{:}\sigma_1 \vdash M@\Delta_1 : \sigma_3 \quad \Gamma, x{:}\sigma_2 \vdash M@\Delta_2 : \sigma_3 \quad \Gamma \vdash N@\Delta_3 : \sigma_1 \cup \sigma_2}{\Gamma \vdash M[N/x]@[\lambda x{:}\sigma_1.\Delta_1, \lambda x{:}\sigma_2.\Delta_2]\,\Delta_3 : \sigma_3}\ (\cup E)$$

Figure 3.2: Typed calculus $\lambda@^{\text{BDdL}}$ [40]

We write $\longrightarrow_{\beta\mathsf{pr}_i\mathsf{in}_i}$ for the contextual closure of the (β), (pr_i) and (in_i) notions of reduction. We write $\twoheadrightarrow_{\beta\mathsf{pr}_i\mathsf{in}_i}$ as the reflexive and transitive closure of $\longrightarrow_{\beta\mathsf{pr}_i\mathsf{in}_i}$. We mostly consider $\beta\mathsf{pr}_i\mathsf{in}_i$-reductions, thus to ease the notation, we will often omit the subscript in $\beta\mathsf{pr}_i\mathsf{in}_i$-reductions. We note $\twoheadrightarrow^{\parallel}$ for the synchronous reduction, *i.e.* the reduction relation where the transitive closure for strong pairs and strong sums are the following rules:

$$\frac{\Delta_1 \to_\Delta \Delta_1' \quad \Delta_2 \to_\Delta \Delta_2' \quad \wr\Delta_1'\wr \equiv \wr\Delta_2'\wr}{\langle \Delta_1, \Delta_2 \rangle \to_\Delta \langle \Delta_1', \Delta_2' \rangle}\ (Congr_\cap)$$

$$\frac{\Delta_1 \to_\Delta \Delta_1' \quad \Delta_2 \to_\Delta \Delta_2' \quad \wr\Delta_1'\wr \equiv \wr\Delta_2'\wr}{[\Delta_1, \Delta_2] \to_\Delta [\Delta_1', \Delta_2']}\ (Congr_\cup)$$

Figure 3.1 presents the main rules of the type assignment system of [7]: note that type inference is not syntax-directed, and undecidable. Figure 3.2 presents the main rules of the typed calculus $\lambda@^{\text{BDdL}}$ of [40]; note that this type system is completely syntax directed and decidable.

The next definition clarifies what we intend with isomorphism between an untyped M and a typed Δ: the essence function shows the syntactic relation between pure λ-terms and Δ-terms. Essence maps typed Δ-terms into untyped λ-terms: intuitively, two typed Δ-terms proves the same formula if they have the same essence.
The essence function between pure and typed λ-terms is defined as follows:

Definition 3.1 (Proof Essence)**.**

$$\wr u_M \wr \stackrel{def}{=} M$$

$$\wr x \wr \stackrel{def}{=} x \qquad \wr \lambda x{:}\sigma.\Delta \wr \stackrel{def}{=} \lambda x.\wr\Delta\wr$$

$$\wr \Delta_1\,\Delta_2 \wr \stackrel{def}{=} \wr\Delta_1\wr\wr\Delta_2\wr \qquad \wr \Delta^{\sigma} \wr \stackrel{def}{=} \wr\Delta\wr$$

$$\wr \mathsf{pr}_i\,\Delta \wr \stackrel{def}{=} \wr\Delta\wr \qquad \wr \mathsf{in}_i^{\sigma}\,\Delta \wr \stackrel{def}{=} \wr\Delta\wr$$

$$\wr \langle \Delta_1, \Delta_2 \rangle \wr \stackrel{def}{=} \wr\Delta_1\wr \qquad \wr [\lambda x{:}\sigma_1.\Delta_1, \lambda x{:}\sigma_2.\Delta_2)]\,\Delta_3 \wr \stackrel{def}{=} \wr\Delta_1\wr[\wr\Delta_3\wr/x]$$

$$\frac{}{\Gamma \vdash u_M : \mathtt{U}}\ (\mathtt{U}) \qquad \frac{\Gamma \vdash \Delta : \sigma \quad \sigma \leqslant \tau}{\Gamma \vdash \Delta^{\tau} : \tau}\ (\leqslant) \qquad \frac{x{:}\sigma \in \Gamma}{\Gamma \vdash x : \sigma}\ (Var)$$

$$\frac{\Gamma, x{:}\sigma_1 \vdash \Delta : \sigma_2}{\Gamma \vdash \lambda x{:}\sigma_1.\Delta : \sigma_1 \to \sigma_2}\ (\to I) \qquad \frac{\Gamma \vdash \Delta_1 : \sigma_1 \to \sigma_2 \quad \Gamma \vdash \Delta_2 : \sigma_1}{\Gamma \vdash \Delta_1\, \Delta_2 : \sigma_2}\ (\to E)$$

$$\frac{\begin{array}{l}\Gamma \vdash \Delta_1 : \sigma_1 \\ \Gamma \vdash \Delta_2 : \sigma_2 \quad \wr \Delta_1 \wr \equiv \wr \Delta_2 \wr\end{array}}{\Gamma \vdash \langle \Delta_1, \Delta_2 \rangle : \sigma_1 \cap \sigma_2}\ (\cap I) \qquad \frac{\Gamma \vdash \Delta : \sigma_1 \cap \sigma_2 \quad i \in \{1,2\}}{\Gamma \vdash \mathsf{pr}_i\, \Delta : \sigma_i}\ (\cap E_i)$$

$$\frac{\Gamma \vdash \Delta : \sigma_i \quad \{i,j\} = \{1,2\}}{\Gamma \vdash \mathsf{in}_i^{\sigma_j}\, \Delta : \sigma_1 \cup \sigma_2}\ (\cup I_i) \qquad \frac{\begin{array}{l}\Gamma, x{:}\sigma_1 \vdash \Delta_1 : \sigma_3 \quad \wr \Delta_1 \wr =_\beta \wr \Delta_2 \wr \\ \Gamma, x{:}\sigma_2 \vdash \Delta_2 : \sigma_3 \quad \Gamma \vdash \Delta_3 : \sigma_1 \cup \sigma_2\end{array}}{\Gamma \vdash [\lambda x{:}\sigma_1.\Delta_1, \lambda x{:}\sigma_2.\Delta_2]\, \Delta_3 : \sigma_3}\ (\cup E)$$

Figure 3.3: Δ-calculus Δ^{BDdL}

The essence function is basically an erasing function that forgets all typing information and the second component of strong pairs and strong sums.

Figure 3.3 presents the main rules of Δ^{BDdL} of [39]: this system can be seen as a *proof-functional* logic, in the sense of Pottinger [80] and López-Escobar [67]: formulæ encode, using the Curry-Howard isomorphism, derivations $\mathcal{D} : \Gamma \vdash M : \sigma$ in the type assignment system λ^{BDdL} which are, in turn, isomorphic to typed judgments $\Gamma \vdash M@\Delta : \sigma$ of $\lambda@^{\mathrm{BDdL}}$. It is worth noticing that if we drop the restriction concerning the essence in rules $(\cap I)$ and $(\cup E)$ in the system Δ^{BDdL}, replace $\sigma \cap \tau$ by $\sigma \times \tau$, and $\sigma \cup \tau$ by $\sigma + \tau$, then we get a simply typed λ-calculus with product and sums, namely the usual intuitionistic propositional NJ logic with implication, conjunction, and disjunction in disguise: the resulting system loses its proof-functionality.

All the introduced typed systems also use a subtyping relation, written $\sigma \leqslant \tau$. Subtyping is defined from a type theory, which is a collection of inequalities between types satisfying natural closure conditions.

Definition 3.2 (Type theory Ξ)**.**
The type theory Ξ (see Definition 3.6 of [7]), is an extension of the type theory $\mathcal{T}_{\mathrm{BCD}}$ of Figure 2.3, and is defined by the following subtyping axioms and inference rules:

(1) $\sigma \leqslant \sigma \cap \sigma$

(2) $\sigma \cup \sigma \leqslant \sigma$

(3) $\sigma \cap \tau \leqslant \sigma, \sigma \cap \tau \leqslant \tau$

(4) $\sigma \leqslant \sigma \cup \tau, \tau \leqslant \sigma \cup \tau$

(5) $\sigma \leqslant \mathtt{U}$

(6) $\sigma \leqslant \sigma$

(7) $\sigma_1 \leqslant \sigma_2, \tau_1 \leqslant \tau_2 \Rightarrow \sigma_1 \cap \tau_1 \leqslant \sigma_2 \cap \tau_2$

(8) $\sigma_1 \leqslant \sigma_2, \tau_1 \leqslant \tau_2 \Rightarrow \sigma_1 \cup \tau_1 \leqslant \sigma_2 \cup \tau_2$

(9) $\sigma \leqslant \tau, \tau \leqslant \rho \Rightarrow \sigma \leqslant \rho$

(10) $\sigma \cap (\tau \cup \rho) \leqslant (\sigma \cap \tau) \cup (\sigma \cap \rho)$

(11) $(\sigma \to \tau) \cap (\sigma \to \rho) \leqslant \sigma \to (\tau \cap \rho)$

(12) $(\sigma \to \rho) \cap (\tau \to \rho) \leqslant (\sigma \cup \tau) \to \rho$

(13) $\mathtt{U} \leqslant \mathtt{U} \to \mathtt{U}$

(14) $\sigma_2 \leqslant \sigma_1, \tau_1 \leqslant \tau_2 \Rightarrow \sigma_1 \to \tau_1 \leqslant \sigma_2 \to \tau_2$

The theory Ξ suggests the interpretation of $\mathtt{U}$ as the set universe, of $\cap$ as the set intersection, of $\cup$ as the set union, and of $\leqslant$ as a subset relation, respectively, in the spirit of

[50].

In the following, we write $\sigma \sim \tau$ if, and only if, $\sigma \leqslant \tau$ and $\tau \leqslant \sigma$. We note that distributivity of union over intersection and intersection over union, *i.e.* $\sigma \cup (\tau \cap \rho) \sim (\sigma \cup \tau) \cap (\sigma \cup \rho)$, and $\sigma \cap (\tau \cup \rho) \sim (\sigma \cap \tau) \cup (\sigma \cap \rho)$ are derivable (see, *e.g.* derivation in [7], page 9).

Once the subtyping preorder has been defined, a classical subsumption rule and two explicit coercion rule can be defined as follows:

$$\frac{\Gamma \vdash M : \sigma \quad \sigma \leqslant \tau}{\Gamma \vdash M : \tau}\ (\leqslant) \qquad \frac{\Gamma \vdash M@\Delta : \sigma \quad \sigma \leqslant \tau}{\Gamma \vdash M@\Delta^{\tau} : \tau}\ (\leqslant) \qquad \frac{\Gamma \vdash \Delta : \sigma \quad \sigma \leqslant \tau}{\Gamma \vdash \Delta^{\tau} : \tau}\ (\leqslant)$$

In a nutshell, the first rule is a subsumption, while the two others are explicit coercions, because of the type decoration.

3.2 Metatheory of Δ^{BDdL} and $\lambda@^{\text{BDdL}}$

The next theorem relates the three systems: the key concept is the essence function $\wr - \wr$ that allows to interpret union, intersection, and explicit coercions as proof-functional connectives.

Theorem 3.1 (Isomorphism)**.**
Let M, Δ, Γ, and σ. Then:

1. $\Gamma \vdash M : \sigma$ iff $\Gamma \vdash \Delta : \sigma$ and $\wr \Delta \wr \equiv M$;
2. $\Gamma \vdash M@\Delta : \sigma$ iff $\Gamma \vdash \Delta : \sigma$;
3. $\Gamma \vdash M : \sigma$ iff $\Gamma \vdash M@\Delta : \sigma$.

Proof.

1. by adding union types to the proof of Theorem 2.22;
2. by induction on the structure of derivations;
3. by parts 1 and 2.

□

The next theorem states that adding union types does not break the properties of the new typed systems.

Theorem 3.2 (Conservativity)**.**
The typed systems $\lambda@^{\text{BDdL}}$ and Δ^{BDdL} preserve subject reduction (for the synchronous β-reduction), Church-Rosser, strong normalization, unicity of typing, and decidability of type reconstruction and of type checking.

Proof. For proving properties of $\lambda@^{\text{BDdL}}$ we proceeds by upgrading results of Theorems 2.4, 2.11, 2.21, 2.19, and Lemma 2.5. Properties of $\lambda@^{\text{BDdL}}$ are mostly inherited by Δ^{BDdL} using Theorem 3.1 or, as for case of subject reduction for $\beta\mathsf{pr}_i\,\mathsf{in}_i$-reductions, is proved by induction on the structure of the derivation. □

Chapter 4

On Mints' realizability

This chapter deals with finding a realizability interpretation of union and intersection types using *Mints' realizers* [71]. Intuitively, a Mints' realizer $r_\sigma[M]$ is a logical proposition stating that M has type σ.

Similarly to the system of intersection types, the type assignment system λ^{BDdL} has no trivial set-theoretic interpretation (see [7]). It is noteworthy that some similar systems have a clear set-theoretic interpretation (see *e.g.* Frisch, Castagna, and Benzaken [50]).

This chapter provides both a intuitive semantics for union types and a logical foundation for Δ^{BDdL}. We do this by interpreting the union type assignment system into the intuitionistic first order logic $\mathsf{NJ}(\beta)$ with Mints' provable realizability of intersection types extended with union. Then, we prove that the terms of Δ^{BDdL} correspond to logical derivations in $\mathsf{NJ}(\beta)$.

From Theorem 3.1, we know that if $\Gamma \vdash M@\Delta : \sigma$, then there is a tight relation among Δ and M, namely $\wr\Delta\wr \equiv M$. In Δ^{BDdL}, Δ can also be seen as a simply-typed term: if we drop the restriction concerning the essence in rules $(\cap I)$ and $(\cup E)$ in Δ^{BDdL} replacing $\sigma \cap \tau$ by $\sigma \times \tau$ and $\sigma \cup \tau$ by $\sigma + \tau$, then we get a simply typed λ-calculus with product and sums, namely the intuitionistic propositional logic with implication, conjunction, and disjunction in disguise.

We could provide a logical foundation for Δ^{BDdL} by interpreting it into an extension of Mints' provable realizability. However, when proving a formula $r_\sigma[M]$, we have two kinds of realizers: the former is the pure λ-term M, while the latter is the Δ-term that turn out to be realizers in the ordinary sense of intuitionistic logic.

Therefore, we prove in Theorem 5.4 that, if $\Gamma \vdash \Delta : \sigma$ is derivable in Δ^{BDdL}, then Δ realizes the $\mathsf{NJ}(\beta)$ judgment $G_\Gamma \vdash_{\mathsf{NJ}(\beta)} r_\sigma[\wr\Delta\wr]$. However, the converse is not true, as is shown in Section 4.3.

For this aim, we use and extend Mints' approach of *provable realizability* [71, 3, 8]. We interpret the statement $\vdash M@\Delta : \sigma$ as "Δ is a construction of $M : \sigma$"; on the other hand $M : \sigma$ is the meaning of the formula $r_\sigma[M]$, provided that we extend the notion to cope with union types; the latter formula reads as "M is a method to assess σ", following the terminology of [67, 8]; now, the meaning of Δ is that it is a constructive proof of $r_\sigma[M]$, and hence it is a realizer of this formula. In short, we have two kinds of realizers on two levels: the realizer M, which is a Mints' realizer of σ, and the realizer Δ which is an ordinary realizer, in the sense of standard Brouwer-Heyting-Kolmogorov interpretation of intuitionistic logic, of the statement $r_\sigma[M]$: in simpler words, Δ represents an encoding of the logical derivation of $r_\sigma[M]$.

This chapter is organized as follows: in Section 4.1, we present the logic $\mathsf{NJ}(\beta)$ along

$$\frac{A \in G}{G \vdash_{\mathsf{NJ}(\beta)} A}\ (Ax) \qquad \frac{G_\Gamma \vdash_{\mathsf{NJ}(\beta)} \mathbf{P}_\phi(M) \quad M =_\beta N}{G_\Gamma \vdash_{\mathsf{NJ}(\beta)} \mathbf{P}_\phi(N)}\ (\beta)$$

$$\frac{}{G \vdash_{\mathsf{NJ}(\beta)} \top}\ (\top I) \qquad \frac{G \vdash_{\mathsf{NJ}(\beta)} \bot}{G \vdash_{\mathsf{NJ}(\beta)} A}\ (\bot E)$$

$$\frac{G \vdash_{\mathsf{NJ}(\beta)} A \quad G \vdash_{\mathsf{NJ}(\beta)} B}{G \vdash_{\mathsf{NJ}(\beta)} A \wedge B}\ (\wedge I) \qquad \frac{G \vdash_{\mathsf{NJ}(\beta)} A_1 \wedge A_2 \quad i = 1,2}{G \vdash_{\mathsf{NJ}(\beta)} A_i}\ (\wedge E_i)$$

$$\frac{G \vdash_{\mathsf{NJ}(\beta)} A_i \quad i = 1,2}{G \vdash_{\mathsf{NJ}(\beta)} A_1 \vee A_2}\ (\vee I_i) \qquad \frac{G, A \vdash_{\mathsf{NJ}(\beta)} C \quad G, B \vdash_{\mathsf{NJ}(\beta)} C \quad G \vdash_{\mathsf{NJ}(\beta)} A \vee B}{G \vdash_{\mathsf{NJ}(\beta)} C}\ (\vee E)$$

$$\frac{G, A \vdash_{\mathsf{NJ}(\beta)} B}{G \vdash_{\mathsf{NJ}(\beta)} A \supset B}\ (\supset I) \qquad \frac{G \vdash_{\mathsf{NJ}(\beta)} A \supset B \quad G \vdash_{\mathsf{NJ}(\beta)} A}{G \vdash_{\mathsf{NJ}(\beta)} B}\ (\supset E)$$

$$\frac{G \vdash_{\mathsf{NJ}(\beta)} A \quad x \notin \mathsf{Fv}(G)}{G \vdash_{\mathsf{NJ}(\beta)} \forall x.A}\ (\forall I) \qquad \frac{G \vdash_{\mathsf{NJ}(\beta)} \forall x.A}{G \vdash_{\mathsf{NJ}(\beta)} A[t/x]}\ (\forall E)$$

$$\frac{G \vdash_{\mathsf{NJ}(\beta)} A[t/x]}{G \vdash_{\mathsf{NJ}(\beta)} \exists x.A}\ (\exists I) \qquad \frac{G \vdash_{\mathsf{NJ}(\beta)} \exists x.A \quad G, A[c/x] \vdash_{\mathsf{NJ}(\beta)} C \quad c \notin \mathsf{Fv}(G)}{G \vdash_{\mathsf{NJ}(\beta)} C}\ (\exists E)$$

Figure 4.1: The logic $\mathsf{NJ}(\beta)$

with Mints' realizers, in Section 4.2, we prove that Δ^{BDdL} is sound *w.r.t.* Mints' realizers in $\mathsf{NJ}(\beta)$. In Section 4.3, we discuss the completeness of Δ^{BDdL} is sound *w.r.t.* Mints' realizers in $\mathsf{NJ}(\beta)$.

4.1 Presentation of $\mathsf{NJ}(\beta)$

The next definition introduces formally the $\mathsf{NJ}(\beta)$ logic[1].

Definition 4.1. (Logic $\mathsf{NJ}(\beta)$).

1. let $\mathbf{P}_\phi(x)$ be a unary predicate for each atomic type ϕ: the natural deduction system for first-order intuitionistic logic $\mathsf{NJ}(\beta)$ extends NJ with untyped λ-terms and predicates $\mathbf{P}_\phi(x)$, and the rule (β). The full description of $\mathsf{NJ}(\beta)$ can be found in Figure 4.1. Note that $\supset$ denotes logical implication, $\top$ and $\bot$ denote truth and falsehood, while $\wedge$ and $\vee$ are the logical connectives for conjunction and disjunction respectively, that must be kept distinct from $\cap$ and $\cup$;

2. for a given context $\Gamma \stackrel{def}{=} \{x_1{:}\sigma_1, \ldots, x_n{:}\sigma_n\}$ of Δ^{BDdL}, we define a logical context $G_\Gamma \stackrel{def}{=} r_{\sigma_1}[x_1], \ldots, r_{\sigma_n}[x_n]$ of $\mathsf{NJ}(\beta)$.

It is clear that $G_{\Gamma,x:\sigma} \equiv G_\Gamma, r_\sigma[x]$ and $x \notin \mathsf{Fv}(G_\Gamma)$, since $x \notin \mathsf{Dom}(\Gamma)$, by context definition. In the rest of this chapter, we will define Mints' realizability for Δ^{BDdL}.

We now give a precise definition of the notion of realizer, as first introduced for intersection types by Mints [71], and extended in [39].

[1] the logic NJ has been named by Gentzen [51] as an abbreviation for "Kalkül des natürlichen intuitionistischen Schließens", *i.e.* "calculus of the natural intuitionistic deduction". The letters I and J were often considered the same by the Germans, way back when they used Fraktur letters.

Definition 4.2 (Mints' realizers in $\mathsf{NJ}(\beta)$ [39]).
Let $\mathbf{P}_\phi(-)$ be a unary predicate for each atomic type ϕ. We define the predicates $r_\sigma[-]$ for each type σ by induction over σ, as follows:

$$r_\phi[M] \stackrel{def}{=} \mathbf{P}_\phi(M) \qquad r_{\sigma_1\to\sigma_2}[M] \stackrel{def}{=} \forall y.r_{\sigma_1}[y] \supset r_{\sigma_2}[M\,y]$$

$$r_{\mathtt{U}}[M] \stackrel{def}{=} \top \qquad r_{\sigma_1\cup\sigma_2}[M] \stackrel{def}{=} r_{\sigma_1}[M] \vee r_{\sigma_2}[M]$$

$$r_{\sigma_1\cap\sigma_2}[M] \stackrel{def}{=} r_{\sigma_1}[M] \wedge r_{\sigma_2}[M]$$

Formulæ have the shape $r_\sigma[M]$, whose intended meaning is that M has type σ in the intersection-union type discipline with subtyping.

Intuitively, we write $r_\sigma[M]$ to denote a formula in $\mathsf{NJ}(\beta)$, realized by the pure λ-term M of type σ in λ^{BDdL}.

4.2 Soundness of $\mathsf{NJ}(\beta)$

This section states that Δ^{BDdL} is sound *w.r.t.* Mints' realizers in $\mathsf{NJ}(\beta)$. We first start with a few technical lemmas.

Lemma 4.1 (Admissibility of $(Eq\beta)$ in $\mathsf{NJ}(\beta)$).
The following rule is admissible in $\mathsf{NJ}(\beta)$:

$$\frac{G_\Gamma \vdash_{\mathsf{NJ}(\beta)} r_\sigma[N] \quad M =_\beta N}{G_\Gamma \vdash_{\mathsf{NJ}(\beta)} r_\sigma[M]}\ (Eq\beta)$$

Proof. By induction over σ. □

Lemma 4.2 (Admissibility in $\mathsf{NJ}(\beta)$).
The following rules are admissible in $\mathsf{NJ}(\beta)$:

$$\frac{G_\Gamma, r_{\sigma_1}[x] \vdash_{\mathsf{NJ}(\beta)} r_{\sigma_2}[M]}{G_\Gamma \vdash_{\mathsf{NJ}(\beta)} r_{\sigma_1\to\sigma_2}[\lambda x.M]}\ (\to I) \qquad \frac{G_\Gamma \vdash_{\mathsf{NJ}(\beta)} r_{\sigma_1\to\sigma_2}[M] \quad G_\Gamma \vdash_{\mathsf{NJ}(\beta)} r_{\sigma_1}[N]}{G_\Gamma \vdash_{\mathsf{NJ}(\beta)} r_{\sigma_2}[M\,N]}\ (\to E)$$

$$\frac{G_\Gamma \vdash_{\mathsf{NJ}(\beta)} r_{\sigma_1}[M] \quad G_\Gamma \vdash_{\mathsf{NJ}(\beta)} r_{\sigma_2}[M]}{G_\Gamma \vdash_{\mathsf{NJ}(\beta)} r_{\sigma_1\cap\sigma_2}[M]}\ (\cap I) \qquad \frac{G_\Gamma \vdash_{\mathsf{NJ}(\beta)} r_{\sigma_1\cap\sigma_2}[M] \quad i \in \{1,2\}}{G_\Gamma \vdash_{\mathsf{NJ}(\beta)} r_{\sigma_i}[M]}\ (\cap E)$$

$$\frac{G_\Gamma \vdash_{\mathsf{NJ}(\beta)} r_{\sigma_i}[M] \quad i \in \{1,2\}}{G_\Gamma \vdash_{\mathsf{NJ}(\beta)} r_{\sigma_1\cup\sigma_2}[M]}\ (\cup I) \qquad \frac{G_\Gamma \vdash_{\mathsf{NJ}(\beta)} r_\sigma[M] \quad \sigma \leqslant \tau}{G_\Gamma \vdash_{\mathsf{NJ}(\beta)} r_\tau[M]}\ (\leqslant)$$

$$\frac{G_\Gamma, r_{\sigma_1}[x] \vdash_{\mathsf{NJ}(\beta)} r_{\sigma_3}[M] \quad G_\Gamma, r_{\sigma_2}[x] \vdash_{\mathsf{NJ}(\beta)} r_{\sigma_3}[M] \quad G_\Gamma \vdash_{\mathsf{NJ}(\beta)} r_{\sigma_1\cup\sigma_2}[N]}{G_\Gamma \vdash_{\mathsf{NJ}(\beta)} r_{\sigma_3}[M[N/x]]}\ (\cup E)$$

Proof. We can see that:

- rules $(\cap I)$, $(\cap E)$, and $(\cup I)$ correspond respectively to rules $(\wedge I)$, $(\wedge E)$, and $(\vee I)$;
- rule $(\cup E)$ is derivable from rule $(\vee E)$ and a classical substitution lemma;

– rule $(\to I)$ and $(\to E)$ are derivable:

$$\dfrac{\dfrac{\dfrac{G_\Gamma, r_{\sigma_1}[x] \vdash_{\mathsf{NJ}(\beta)} r_{\sigma_2}[M]}{G_\Gamma, r_{\sigma_1}[x] \vdash_{\mathsf{NJ}(\beta)} r_{\sigma_2}[(\lambda x.M)\,x]}\,(Eq\beta)}{G_\Gamma \vdash_{\mathsf{NJ}(\beta)} r_{\sigma_1}[x] \supset r_{\sigma_2}[(\lambda x.M)\,x]}\,(\supset I)}{G_\Gamma \vdash_{\mathsf{NJ}(\beta)} \forall x.r_{\sigma_1}[x] \supset r_{\sigma_2}[(\lambda x.M)\,x]}\,(\forall I)$$

and:

$$\dfrac{\dfrac{G_\Gamma \vdash_{\mathsf{NJ}(\beta)} r_{\sigma\to\tau}[M]}{G_\Gamma \vdash_{\mathsf{NJ}(\beta)} r_\sigma[N] \supset r_\tau[MN]}\,(\forall E) \qquad G_\Gamma \vdash_{\mathsf{NJ}(\beta)} r_\sigma[N]}{G_\Gamma \vdash_{\mathsf{NJ}(\beta)} r_\tau[MN]}\,(\supset E)$$

– it can be showed that all the subtyping rules are derivable in $\mathsf{NJ}(\beta)$, therefore $(\leqslant)$ is derivable.

□

We can now prove relations between λ^{BDdL} (and $\lambda@^{\text{BDdL}}$, and Δ^{BDdL}) and $\mathsf{NJ}(\beta)$.

Lemma 4.3 (λ^{BDdL} *vs.* $\mathsf{NJ}(\beta)$)**.**
If $\Gamma \vdash M : \sigma$ then $G_\Gamma \vdash_{\mathsf{NJ}(\beta)} r_\sigma[M]$.

Proof. By structural induction on the derivation tree of $\Gamma \vdash M : \sigma$: all the rules can be replaced with rules of the same name (using Lemma 4.2), except rule (Var) which is replaced by rule (Ax). □

Lemma 4.4. If $\Gamma \vdash M@\Delta : \sigma$ in $\lambda@^{\text{BDdL}}$ then $G_\Gamma \vdash_{\mathsf{NJ}(\beta)} r_\sigma[M]$.

Proof. Because of Lemma 4.3 and because $\lambda@^{\text{BDdL}}$ and λ^{BDdL} are equivalent by Theorem 3.1. □

Theorem 4.5 (Soundness)**.**
If $\Gamma \vdash \Delta : \sigma$ is derivable in Δ^{BDdL}, then $G_\Gamma \vdash_{\mathsf{NJ}(\beta)} r_\sigma[\wr\,\Delta\,\wr]$.

Proof. By Theorem 3.1 if $\Gamma \vdash \Delta : \sigma$ is derivable then $\Gamma \vdash \wr\,\Delta\,\wr@\Delta : \sigma$. The thesis follows by Lemma 4.4. □

Remark 4.1.
The type assignment system λ^{BDdL} of [7] was based on the type theory Ξ (see Definition 3.6 of [7]): the paper also introduced a stronger type theory, called Π, by adding the extra axiom:

$$(15) \qquad \mathbf{P}(\sigma) \Rightarrow \sigma \to \tau \cup \rho \leqslant (\sigma \to \tau) \cup (\sigma \to \rho),$$

where $\mathbf{P}(\sigma)$ is true if σ syntactically corresponds to an Harrop formula. However, in $\mathsf{NJ}(\beta)$, the judgment $r_{\sigma\to(\tau\cup\rho)}[x] \vdash_{\mathsf{NJ}(\beta)} r_{(\sigma\to\tau)\cup(\sigma\to\rho)}[x]$ is not derivable because the judgment $A \supset (B \vee C) \vdash_{\mathsf{NJ}(\beta)} (A \supset B) \vee (A \supset C)$ is not derivable in NJ. As such, the type theory Π cannot be overlapped with an interpretation of types as sets, as the following example show. The identity function $\lambda x.x$ inhabits the function set $\{a, b\} \to \{a\} \cup \{b\}$ but, by axiom (15), it should also inhabit $\{a, b\} \to \{a\}$ or $\{a, b\} \to \{b\}$, which is clearly not the case.

4.3 Completeness of $\mathsf{NJ}(\beta)$

One may wonder why there is no theorem stating Δ^{BDdL} can realize every derivation of $\mathsf{NJ}(\beta)$. The conjecture that we would like to prove is the following:

Conjecture 4.1 (Completeness)**.**
If $G_\Gamma \vdash_{\mathsf{NJ}(\beta)} r_\sigma[M]$ then there exists $N =_\beta M$ and Δ such that $\Gamma \vdash N@\Delta : \sigma$ in $\lambda@^{\text{BDdL}}$ and therefore $\Gamma \vdash \Delta : \sigma$ in Δ^{BDdL}.

Note that the condition $N =_\beta M$ is necessary because λ^{BDdL} does not enjoy subject conversion, while Lemma 4.1 enforces subject conversion. However, this conjecture has still not been proven (or disproven) yet.

4.3.1 Failure of completeness of $\mathsf{NJ}(\beta)$ without subtyping

It seems that subtyping is an essential component of the conjectured completeness result. Intuitively, if we naively interpret types as sets, it is clear that the set $(\sigma \cup \tau) \cap (\sigma \cup \rho)$ is a subset of $\sigma \cup (\tau \cap \rho)$, and it appears that we can prove in $\mathsf{NJ}(\beta)$ that the identity function has type $(\sigma \cup \tau) \cap (\sigma \cup \rho) \to \sigma \cup (\tau \cap \rho)$, as is shown in Subsection 4.3.2. However, in λ^{BDdL}, the λ-term $\lambda x.x$ cannot have type $(\sigma \cup \tau) \cap (\sigma \cup \rho) \to \sigma \cup (\tau \cap \rho)$ if we don't consider subtyping.

4.3.2 Counter-example

We show that $\vdash_{\mathsf{NJ}(\beta)} r_{(\sigma\cup\tau)\cap(\sigma\cup\rho)\to\sigma\cup(\tau\cap\rho)}[\lambda x.x]$ is derivable in $\mathsf{NJ}(\beta)$.

$$
\begin{prooftree}
\AxiomC{$\mathcal{D}_1$}
\RightLabel{}
\UnaryInfC{$(r_\sigma[y] \vee r_\tau[y]) \wedge (r_\sigma[y] \vee r_\rho[y]), r_\sigma[y] \vdash_{\mathsf{NJ}(\beta)} r_\sigma[(\lambda x.x)y] \vee (r_\tau[(\lambda x.x)y] \wedge r_\rho[(\lambda x.x)y])$}
\AxiomC{$\mathcal{D}_2$}
\RightLabel{}
\UnaryInfC{$(r_\sigma[y] \vee r_\tau[y]) \wedge (r_\sigma[y] \vee r_\rho[y]), r_\tau[y] \vdash_{\mathsf{NJ}(\beta)} r_\sigma[(\lambda x.x)y] \vee (r_\tau[(\lambda x.x)y] \wedge r_\rho[(\lambda x.x)y])$}
\AxiomC{$\mathcal{D}_6$}
\RightLabel{}
\UnaryInfC{$(r_\sigma[y] \vee r_\tau[y]) \wedge (r_\sigma[y] \vee r_\rho[y]) \vdash_{\mathsf{NJ}(\beta)} r_\sigma[y] \vee r_\tau[y]$}
\RightLabel{$(\vee E)$}
\BinaryInfC{$(r_\sigma[y] \vee r_\tau[y]) \wedge (r_\sigma[y] \vee r_\rho[y]) \vdash_{\mathsf{NJ}(\beta)} r_\sigma[(\lambda x.x)y] \vee (r_\tau[(\lambda x.x)y] \wedge r_\rho[(\lambda x.x)y])$}
\RightLabel{$(\supset I)$}
\UnaryInfC{$\vdash_{\mathsf{NJ}(\beta)} (r_\sigma[y] \vee r_\tau[y]) \wedge (r_\sigma[y] \vee r_\rho[y]) \supset r_\sigma[(\lambda x.x)y] \vee (r_\tau[(\lambda x.x)y] \wedge r_\rho[(\lambda x.x)y])$}
\RightLabel{$(\equiv)$}
\UnaryInfC{$\vdash_{\mathsf{NJ}(\beta)} r_{(\sigma\cup\tau)\cap(\sigma\cup\rho)}[y] \supset r_{\sigma\cup(\tau\cap\rho)}[(\lambda x.x)y]$}
\RightLabel{$(\forall I)$}
\UnaryInfC{$\vdash_{\mathsf{NJ}(\beta)} \forall y.r_{(\sigma\cup\tau)\cap(\sigma\cup\rho)}[y] \supset r_{\sigma\cup(\tau\cap\rho)}[(\lambda x.x)y]$}
\RightLabel{$(\equiv)$}
\UnaryInfC{$\vdash_{\mathsf{NJ}(\beta)} r_{(\sigma\cup\tau)\cap(\sigma\cup\rho)\to\sigma\cup(\tau\cap\rho)}[\lambda x.x]$}
\end{prooftree}
$$

where $\mathcal{D}_1$ is:

$$
\begin{prooftree}
\AxiomC{}
\RightLabel{(Ax)}
\UnaryInfC{$(r_\sigma[y] \vee r_\tau[y]) \wedge (r_\sigma[y] \vee r_\rho[y]), r_\sigma[y] \vdash_{\mathsf{NJ}(\beta)} r_\sigma[y]$}
\AxiomC{$y =_{\beta\eta} (\lambda x.x)y$}
\RightLabel{$(Ax\beta\eta)$}
\BinaryInfC{$(r_\sigma[y] \vee r_\tau[y]) \wedge (r_\sigma[y] \vee r_\rho[y]), r_\sigma[y] \vdash_{\mathsf{NJ}(\beta)} r_\sigma[(\lambda x.x)y]$}
\RightLabel{$(\vee I)$}
\UnaryInfC{$(r_\sigma[y] \vee r_\tau[y]) \wedge (r_\sigma[y] \vee r_\rho[y]), r_\sigma[y] \vdash_{\mathsf{NJ}(\beta)} r_\sigma[(\lambda x.x)y] \vee (r_\tau[(\lambda x.x)y] \wedge r_\rho[(\lambda x.x)y])$}
\end{prooftree}
$$

and $\mathcal{D}_2$ is:

$$
\begin{prooftree}
\AxiomC{$\mathcal{D}_3$}
\RightLabel{}
\UnaryInfC{$(r_\sigma[y] \vee r_\tau[y]) \wedge (r_\sigma[y] \vee r_\rho[y]), r_\tau[y], r_\sigma[y] \vdash_{\mathsf{NJ}(\beta)} r_\sigma[(\lambda x.x)y] \vee (r_\tau[(\lambda x.x)y] \wedge r_\rho[(\lambda x.x)y])$}
\AxiomC{$\mathcal{D}_4$}
\RightLabel{}
\UnaryInfC{$(r_\sigma[y] \vee r_\tau[y]) \wedge (r_\sigma[y] \vee r_\rho[y]), r_\tau[y], r_\rho[y] \vdash_{\mathsf{NJ}(\beta)} r_\sigma[(\lambda x.x)y] \vee (r_\tau[(\lambda x.x)y] \wedge r_\rho[(\lambda x.x)y])$}
\AxiomC{$\mathcal{D}_5$}
\RightLabel{}
\UnaryInfC{$(r_\sigma[y] \vee r_\tau[y]) \wedge (r_\sigma[y] \vee r_\rho[y]), r_\tau[y] \vdash_{\mathsf{NJ}(\beta)} r_\sigma[y] \vee r_\rho[y]$}
\RightLabel{$(\vee E)$}
\BinaryInfC{$(r_\sigma[y] \vee r_\tau[y]) \wedge (r_\sigma[y] \vee r_\rho[y]), r_\tau[y] \vdash_{\mathsf{NJ}(\beta)} r_\sigma[(\lambda x.x)y] \vee (r_\tau[(\lambda x.x)y] \wedge r_\rho[(\lambda x.x)y])$}
\end{prooftree}
$$

and $\mathcal{D}_3$ is:

$$
\begin{prooftree}
\AxiomC{}
\RightLabel{(Ax)}
\UnaryInfC{$(r_\sigma[y] \vee r_\tau[y]) \wedge (r_\sigma[y] \vee r_\rho[y]), r_\tau[y], r_\sigma[y] \vdash_{\mathsf{NJ}(\beta)} r_\sigma[y]$}
\AxiomC{$y =_{\beta\eta} (\lambda x.x)y$}
\RightLabel{$(Ax\beta\eta)$}
\BinaryInfC{$(r_\sigma[y] \vee r_\tau[y]) \wedge (r_\sigma[y] \vee r_\rho[y]), r_\tau[y], r_\sigma[y] \vdash_{\mathsf{NJ}(\beta)} r_\sigma[(\lambda x.x)y]$}
\UnaryInfC{$(r_\sigma[y] \vee r_\tau[y]) \wedge (r_\sigma[y] \vee r_\rho[y]), r_\tau[y], r_\sigma[y] \vdash_{\mathsf{NJ}(\beta)} r_\sigma[(\lambda x.x)y] \vee (r_\tau[(\lambda x.x)y] \wedge r_\rho[(\lambda x.x)y])$}
\end{prooftree}
$$

and $\mathcal{D}_4$ is:

$$
\begin{prooftree}
\AxiomC{}
\RightLabel{(Ax)}
\UnaryInfC{$(r_\sigma[y] \vee r_\tau[y]) \wedge (r_\sigma[y] \vee r_\rho[y]), r_\tau[y], r_\rho[y] \vdash_{\mathsf{NJ}(\beta)} r_\tau[y]$}
\AxiomC{$y =_{\beta\eta} (\lambda x.x)y$}
\RightLabel{$(Ax\beta\eta)$}
\BinaryInfC{$(r_\sigma[y] \vee r_\tau[y]) \wedge (r_\sigma[y] \vee r_\rho[y]), r_\tau[y], r_\rho[y] \vdash_{\mathsf{NJ}(\beta)} r_\tau[(\lambda x.x)y]$}
\AxiomC{}
\RightLabel{(Ax)}
\UnaryInfC{$(r_\sigma[y] \vee r_\tau[y]) \wedge (r_\sigma[y] \vee r_\rho[y]), r_\tau[y], r_\rho[y] \vdash_{\mathsf{NJ}(\beta)} r_\rho[y]$}
\AxiomC{$y =_{\beta\eta} (\lambda x.x)y$}
\RightLabel{$(Ax\beta\eta)$}
\BinaryInfC{$(r_\sigma[y] \vee r_\tau[y]) \wedge (r_\sigma[y] \vee r_\rho[y]), r_\tau[y], r_\rho[y] \vdash_{\mathsf{NJ}(\beta)} r_\rho[(\lambda x.x)y]$}
\RightLabel{$(\wedge I)$}
\BinaryInfC{$(r_\sigma[y] \vee r_\tau[y]) \wedge (r_\sigma[y] \vee r_\rho[y]), r_\tau[y], r_\rho[y] \vdash_{\mathsf{NJ}(\beta)} r_\tau[(\lambda x.x)y] \wedge r_\rho[(\lambda x.x)y]$}
\RightLabel{$(\vee I)$}
\UnaryInfC{$(r_\sigma[y] \vee r_\tau[y]) \wedge (r_\sigma[y] \vee r_\rho[y]), r_\tau[y], r_\rho[y] \vdash_{\mathsf{NJ}(\beta)} r_\sigma[(\lambda x.x)y] \vee (r_\tau[(\lambda x.x)y] \wedge r_\rho[(\lambda x.x)y])$}
\end{prooftree}
$$

and $\mathcal{D}_5$ is:

$$
\begin{prooftree}
\AxiomC{}
\RightLabel{(Ax)}
\UnaryInfC{$(r_\sigma[y] \vee r_\tau[y]) \wedge (r_\sigma[y] \vee r_\rho[y]), r_\tau[y] \vdash_{\mathsf{NJ}(\beta)} (r_\sigma[y] \vee r_\tau[y]) \wedge (r_\sigma[y] \vee r_\rho[y])$}
\RightLabel{$(\wedge E)$}
\UnaryInfC{$(r_\sigma[y] \vee r_\tau[y]) \wedge (r_\sigma[y] \vee r_\rho[y]), r_\tau[y] \vdash_{\mathsf{NJ}(\beta)} r_\sigma[y] \vee r_\rho[y]$}
\end{prooftree}
$$

and $\mathcal{D}_6$ is:

$$
\begin{prooftree}
\AxiomC{}
\RightLabel{(Ax)}
\UnaryInfC{$(r_\sigma[y] \vee r_\tau[y]) \wedge (r_\sigma[y] \vee r_\rho[y]) \vdash_{\mathsf{NJ}(\beta)} (r_\sigma[y] \vee r_\tau[y]) \wedge (r_\sigma[y] \vee r_\rho[y])$}
\RightLabel{$(\wedge E)$}
\UnaryInfC{$(r_\sigma[y] \vee r_\tau[y]) \wedge (r_\sigma[y] \vee r_\rho[y]) \vdash_{\mathsf{NJ}(\beta)} r_\sigma[y] \vee r_\tau[y]$}
\end{prooftree}
$$

Chapter 5

Subtyping algorithm

This chapter presents a subtyping algorithm for a type system with intersection, union and the universal type `U`. In the literature, there is already a subtyping algorithm for intersection types with `U` [54], but without union types. The correction and completeness of such an algorithm is not trivial, even though it is a crucial part of the proof of the decidability of type reconstruction and type checking. Modern theorem provers such as Coq allow us to design and certify such algorithms[1]. We have designed and certified on paper an algorithm that I have thereafter certified in Coq, in the spirit of Bessai's Coq implementation of the subtyping algorithm without unions [16]. The full source code of the Coq implementation can be found at `https://github.com/cstolze/Bull-Subtyping`. The certification of the algorithm occurs in two steps:

1. first, we define the subtyping relation and prove some basic properties;

2. then we implement the subtyping algorithm and show it is sound and complete *w.r.t.* the subtyping relation.

This chapter is organized as follows: in Section 5.1, we shortly present the subtyping algorithm. In Section 5.2, we explain the details of the Coq implementation of the theory. In Section 5.3, we show the implementation of the subtyping algorithm in Coq and show the intricacies of certified programming. In Section 5.4, we show how to extract the Coq code into valid OCaml code. In Section 5.5, we detail the `preorder` tactic we have developed in order to ease the certification of the subtyping algorithm.

5.1 The algorithm, shortly explained

The types have the following BNF syntax:

$$\sigma, \tau, \rho \quad ::= \quad \alpha \mid \sigma \cap \sigma \mid \sigma \cup \sigma \mid \sigma \to \sigma \mid \mathtt{U}$$

Subtyping is defined as the theory Ξ from [7], as recalled in Figure 5.1. The subtyping algorithm rewrites the types in some normal form, then proceeds on the syntactical structure of these normal forms.

We thus define the Arrow Normal Form (ANF) as follows :

[1] As long as we trust the metatheory and the implementation of the theorem prover.

$$\frac{}{\sigma \leqslant \sigma \cap \sigma} \quad (1)$$

$$\frac{}{\sigma \cup \sigma \leqslant \sigma} \quad (2)$$

$$\frac{i \in \{1,2\}}{\sigma_1 \cap \sigma_2 \leqslant \sigma_i} \quad (3)$$

$$\frac{i \in \{1,2\}}{\sigma_i \leqslant \sigma_1 \cup \sigma_2} \quad (4)$$

$$\frac{}{\sigma \leqslant \mathtt{U}} \quad (5)$$

$$\frac{}{\sigma \leqslant \sigma} \quad (6)$$

$$\frac{\sigma_1 \leqslant \sigma_2 \quad \tau_1 \leqslant \tau_2}{\sigma_1 \cap \tau_1 \leqslant \sigma_2 \cap \tau_2} \quad (7)$$

$$\frac{\sigma_1 \leqslant \sigma_2 \quad \tau_1 \leqslant \tau_2}{\sigma_1 \cup \tau_1 \leqslant \sigma_2 \cup \tau_2} \quad (8)$$

$$\frac{\sigma \leqslant \tau \quad \tau \leqslant \rho}{\sigma \leqslant \rho} \quad (9)$$

$$\frac{}{\sigma \cap (\tau \cup \rho) \leqslant (\sigma \cap \tau) \cup (\sigma \cap \rho)} \quad (10)$$

$$\frac{}{(\sigma \to \tau) \cap (\sigma \to \rho) \leqslant \sigma \to (\tau \cap \rho)} \quad (11)$$

$$\frac{}{(\sigma \to \rho) \cap (\tau \to \rho) \leqslant (\sigma \cup \tau) \to \rho} \quad (12)$$

$$\frac{}{\mathtt{U} \leqslant \mathtt{U} \to \mathtt{U}} \quad (13)$$

$$\frac{\sigma_2 \leqslant \sigma_1 \quad \tau_1 \leqslant \tau_2}{\sigma_1 \to \tau_1 \leqslant \sigma_2 \to \tau_2} \quad (14)$$

Figure 5.1: The type theory Ξ of [7]

Definition 5.1 (ANF)**.**
A type is in *Arrow Normal Form* (ANF) if:

- it is a type variable;
- it is a type $\sigma \to \tau$, where σ is an intersection of ANFs (or $\mathtt{U}$) and τ is an union of ANFs.

Note that $\mathtt{U}$ is not an ANF.

Definition 5.2 (CANF and DANF)**.**
These normal forms are similar to the usual *Conjunctive* and *Disjunctive Normal Forms* (CNF and DNF) found in boolean algebras.

- An intersection of unions of ANFs is called a *Conjunctive Arrow Normal Form* (CANF);
- An union of intersections of ANFs is called a *Disjunctive Arrow Normal Form* (DANF).

The type $\mathtt{U}$ is considered to be a CANF and a DANF.

We use four rewriting subroutines, $\mathcal{R}_1$, $\mathcal{R}_2$, $\mathcal{R}_3$, and $\mathcal{R}_4$, in order to rewrite types in normal form.
The first routine $\mathcal{R}_1$ removes all useless occurrences of $\mathtt{U}$.

Definition 5.3. (Subroutine $\mathcal{R}_1$)
The term rewriting system $\mathcal{R}_1$ (called `deleteOmega` in the Coq code) is defined as follows:

- $\mathtt{U} \cap \sigma$ and $\sigma \cap \mathtt{U}$ rewrite to σ;
- $\mathtt{U} \cup \sigma$ and $\sigma \cup \mathtt{U}$ rewrite to $\mathtt{U}$;
- $\sigma \to \mathtt{U}$ rewrites to $\mathtt{U}$.

It is easy to verify that $\mathcal{R}_1$ terminates and his complexity is linear.

The subroutines $\mathcal{R}_2$ and $\mathcal{R}_3$ rewrite a term in conjunctive and disjunctive normal form, respectively.

Definition 5.4. (Subroutines $\mathcal{R}_2$ and $\mathcal{R}_3$)

- The term rewriting system $\mathcal{R}_2$ rewrites a type in its CNF; it is defined as follows:
  - $\sigma \cup (\tau \cap \rho)$ rewrites to $(\sigma \cup \tau) \cap (\sigma \cup \rho)$;
  - $(\sigma \cap \tau) \cup \rho$ rewrites to $(\sigma \cup \rho) \cap (\tau \cup \rho)$;
- The term rewriting system $\mathcal{R}_3$ rewrites a type in its DNF; it is defined as follows:
  - $\sigma \cap (\tau \cup \rho)$ rewrites to $(\sigma \cap \tau) \cup (\sigma \cap \rho)$;
  - $(\sigma \cup \tau) \cap \rho$ rewrites to $(\sigma \cap \rho) \cup (\tau \cup \rho)$.

It is well-known that $\mathcal{R}_2$ and $\mathcal{R}_3$ terminate and that the complexity of those algorithms is exponential.

Subroutine $\mathcal{R}_4$ rewrites a type as an intersection of ANFs.

Definition 5.5. (Subroutine $\mathcal{R}_4$)
The term rewriting system $\mathcal{R}_4$ rewrites an arrow type into an intersection of ANFs, it is defined as follows:

- $\sigma \to \tau$ rewrites to $\mathcal{R}_3(\sigma) \to \mathcal{R}_2(\tau)$;
- $\cup_i \sigma_i \to \cap_h \tau_h$ rewrites to $\cap_i(\cap_h(\sigma_i \to \tau_h))$.

Since $\mathcal{R}_2$ and $\mathcal{R}_3$ terminate, it follows that $\mathcal{R}_4$ terminates and its complexity is exponential. Note that in the Coq implementation, we found it easier to directly write functions that rewrite terms in CANF and DANF (these are respectively called `_CANF` and `_DANF`). We could finally introduce the main algorithm $\mathcal{A}$ as follows:

Definition 5.6. (Algorithm $\mathcal{A}$)
The main algorithm $\mathcal{A}$ takes as inputs two types σ in DANF and τ in CANF, and decides whether $\sigma \leqslant \tau$ by structural induction as follows:

- if σ and τ are two type variables α and β, then $\sigma \leqslant \tau$ if, and only if, $\alpha \equiv \beta$;
- if $\tau \equiv \mathtt{U}$, then $\sigma \leqslant \tau$;
- if $\sigma \equiv \mathtt{U}$ and $\tau \not\equiv \mathtt{U}$, then $\sigma \not\leqslant \tau$;
- if $\sigma \equiv \sigma_1 \cup \sigma_2$, then $\sigma \leqslant \tau$ if, and only if, $\sigma_1 \leqslant \tau$ and $\sigma_2 \leqslant \tau$;
- if $\tau \equiv \tau_1 \cap \tau_2$, then $\sigma \leqslant \tau$ if, and only if, $\sigma \leqslant \tau_1$ and $\sigma \leqslant \tau_2$;
- if $\sigma \equiv \sigma_1 \cap \sigma_2$, then $\sigma \leqslant \tau$ if, and only if, $\sigma_1 \leqslant \tau$ or $\sigma_2 \leqslant \tau$;
- if $\tau \equiv \tau_1 \cup \tau_2$, then $\sigma \leqslant \tau$ if, and only if, $\sigma \leqslant \tau_1$ or $\sigma \leqslant \tau_2$;
- if $\sigma \equiv \sigma_1 \to \sigma_2$ and $\tau \equiv \tau_1 \to \tau_2$, then $\sigma \leqslant \tau$ if, and only if, $\tau_1 \leqslant \sigma_1$ and $\sigma_2 \leqslant \tau_2$;
- for all other cases, $\sigma \not\leqslant \tau$.

The next section gives a short explanation of the proof of soundness and correctness of the algorithm, which has been formally implemented in Coq.

5.1.1 Soundness and correctness of the algorithm

Lemma 5.1. For all the term rewriting systems $\mathcal{R}_{1,2,3,4}$ we have that $\mathcal{R}(\sigma) \sim \sigma$.

Proof. Each rewriting rule rewrites a term into an equivalent ($\sim$) term. □

The previous lemma has been also proved in Coq using a *strong specification*. A strong specification is a type that fully specify the desired behavior of a function. For instance, the function `deleteOmega` will have type $\forall\ \sigma : \texttt{term}, \{\tau \mid \tau \sim \sigma \wedge (\texttt{Omega_free}\ \tau \vee \tau = \texttt{U})\}$, which means that `deleteOmega` (*i.e.* $\mathcal{R}_1$) takes as input an expression σ, and return an expression τ such that $\tau \sim \sigma$ and τ has the property $\texttt{Omega_free}\ \tau \vee \tau = \texttt{U}$, which means there are no useless occurrences of `U` inside τ.

Lemma 5.2.

1. $\sigma \cup \tau \leqslant \rho$ iff $\sigma \leqslant \rho$ and $\tau \leqslant \rho$;
2. $\sigma \leqslant \tau \cap \rho$ iff $\sigma \leqslant \tau$ and $\sigma \leqslant \rho$.

Proof. The two parts can be proved by examining the subtyping rules of the type theory Ξ. □

Lemma 5.3.
If all the σ_i and τ_j are ANFs, then:

1. If $\exists j, \cap_i \sigma_i \leqslant \tau_j$, then $\cap_i \sigma_i \leqslant \cup_j \tau_j$;
2. If $\exists i, \sigma_i \leqslant \cup \tau_j$, then $\cap_i \sigma_i \leqslant \cup_j \tau_j$.

Proof. The two parts can be proved by induction on the subtyping rules of the type theory Ξ using the ANF definition. □

Lemma 5.3 is quite complicated to encode in Coq: we have chosen to define filters (*i.e.* a set of all the types *bigger* than a given type) for intersection of ANFs, and ideals (*i.e.* a set of all the types *smaller* than a given type) for union of ANFs, then we prove that the definition of filters and ideals are sound and complete *w.r.t.* subtyping. You can see the previous lemma as the interesting part of the completeness proof of ideals and filters.
The soundness and completeness proofs are then straightforward.

Theorem 5.4 (Soundness of $\mathcal{A}$)**.**
Let σ (resp. τ) be in DANF (resp. CANF). If $\mathcal{A}(\sigma, \tau)$, then $\sigma \leqslant \tau$.

Proof. The proof follows the algorithm, therefore it proceeds by structural induction, using Lemmas 5.2 and 5.3. □

Theorem 5.5 (Completeness of $\mathcal{A}$)**.**
Let σ (resp. τ) be in DANF (resp. CANF), such that $\sigma \leqslant \tau$. We have that $\mathcal{A}(\sigma, \tau)$.

Proof. The proof proceeds by mutual induction, using Lemmas 5.2 and 5.3. □

Soundness and completess of $\mathcal{A}$ (called `main_algo` in the Coq source) is mechanically proven through strong specification. The function `main_algo` takes as input two types σ and τ, a proof that σ is in DANF and τ in CANF, and returns either a proof that $\sigma \leqslant \tau$, or a proof that $\sigma \nleqslant \tau$.

5.2 Coq implementation of the theory Ξ

This section documents the Coq implementation of the theory Ξ, as well as the the important lemmas used to certify the subtyping algorithm. We start with the parameter module, which contains the alphabet of variables, is called 𝕍, and has two important fields:

- 𝕍.t, which is the type of variables;
- eq_dec, which is a proof that equality is decidable.

The types for which we define a subtyping algorithm only have atoms, arrows, intersections, union, and the universal type U. The syntax of types is defined using a simple inductive type:

```
Inductive term : Set :=
| Var : 𝕍.t -> term
| Arrow : term -> term -> term
| Inter : term -> term -> term
| Union : term -> term -> term
| Omega : term.
Infix "→ " := (Arrow) (at level 60, right associativity).
Notation "(→ )" := Arrow (only parsing).
Infix "∩ " := (Inter) (at level 35, right associativity).
Notation "(∩ )" := (Inter) (only parsing).
Infix "∪ " := (Union) (at level 30, right associativity).
Notation "(∪ )" := (Union) (only parsing).
Notation "'U'" := (Omega).
```

The subtyping relation is the theory Ξ defined in [7], and is defined inductively in Coq as follows:

```
Inductive Subtype : term -> term -> Prop :=
| R_InterMeetLeft : ∀ σ τ, σ ∩ τ ⩽ σ
| R_InterMeetRight : ∀ σ τ, σ ∩ τ ⩽ τ
| R_InterIdem : ∀ τ, τ ⩽ τ ∩ τ
| R_UnionMeetLeft : ∀ σ τ, σ ⩽ σ ∪ τ
| R_UnionMeetRight : ∀ σ τ, τ ⩽ σ ∪ τ
| R_UnionIdem : ∀ τ, τ ∪ τ ⩽ τ
| R_InterDistrib : ∀ σ τ ρ, (σ → ρ) ∩ (σ → τ) ⩽ σ → ρ ∩ τ
| R_UnionDistrib : ∀ σ τ ρ, (σ → ρ) ∩ (τ → ρ) ⩽ σ ∪ τ → ρ
| R_InterSubtyDistrib: ∀ σ σ' τ τ', σ ⩽ σ' -> τ ⩽ τ' -> σ ∩ τ ⩽ σ' ∩ τ'
| R_UnionSubtyDistrib: ∀ σ σ' τ τ', σ ⩽ σ' -> τ ⩽ τ' -> σ ∪ τ ⩽ σ' ∪ τ'
| R_InterUnionDistrib: ∀ σ τ ρ, σ ∩ (τ ∪ ρ) ⩽ (σ ∩ τ) ∪ (σ ∩ ρ)
| R_CoContra : ∀ σ σ' τ τ', σ ⩽ σ' -> τ ⩽ τ' -> σ' → τ ⩽ σ → τ'
| R_OmegaTop : ∀ σ, σ ⩽ U
| R_OmegaArrow : U ⩽ U → U
| R_Reflexive : ∀ σ, σ ⩽ σ
| R_Transitive : ∀ σ τ ρ, σ ⩽ τ -> τ ⩽ ρ -> σ ⩽ ρ
where "σ ⩽ τ" := (Subtype σ τ).
Notation "(⩽ )" := (Subtype) (only parsing).
```

We say that two types σ and τ are equivalent if they are subtype of one another:

```
Definition equiv (σ τ : term) : Prop := (σ ⩽ τ) ∧ (τ ⩽ σ).
Notation "σ ∼ τ" := (equiv σ τ).
```

The subtyping relation is obviously a preorder relation, thanks to the `R_Reflexive` and `R_Transitive` rules. There are special Coq tactics, such as the `reflexivity` tactic, for reflexive relations, or `transitivity` for transitive relations, and we have defined a tactic for dealing with preorders (see Section 5.5). In order to use these tactics, Coq has to know our subtyping relation has the corresponding properties, and to do so we use Coq typeclasses:

```
Instance Subtypes_Reflexive : Reflexive (⩽ ) := R_Reflexive.
Instance Subtypes_Transitive : Transitive (⩽ ) := R_Transitive.
Instance Subtypes_Preorder : PreOrder (⩽ ) :=
  {| PreOrder_Reflexive  := Subtypes_Reflexive;
     PreOrder_Transitive := Subtypes_Transitive |}.
```

Coq has an automatic proof search engine. We can declare which theorems, or constructors the search engine can use on its own, the art is to guide it so the proofs are concise, and the engine does not take too much time. These declarations are called *hints*, and are stored in a *hint database*. For instance, the following code creates a hint database SubtypeHints, which contains all the subtyping rules, and also allows the unfolding of the definition of the equivalence relation:

```
Create HintDb SubtypeHints.
Hint Constructors Subtype : SubtypeHints.
Hint Unfold equiv : SubtypeHints.
```

The proof that the equivalence relation is reflexive is then trivial, and can be added to the hint database:

```
Instance equiv_Reflexive: Reflexive (∼ ).
Proof.
  auto with SubtypeHints.
Qed.
Hint Immediate equiv_Reflexive : SubtypeHints.
```

Then we can prove simple facts on the subtyping relation, then add these facts to the hint database when it is clear they make any proof progress:

```
Fact Inter_inf : ∀ σ τ ρ, σ ⩽ τ -> σ ⩽ ρ -> σ ⩽ τ ∩ ρ.
Proof with auto with SubtypeHints.
  intros.
  transitivity (σ ∩ σ)...
Qed.
Hint Resolve Inter_inf : SubtypeHints.
```

However, the converse of `Inter_inf`, called `Inter_inf'`, should not be added to the hint database, or else the search engine would uselessly loop by applying `Inter_inf` and then `Inter_inf'`:

```
Fact Inter_inf' : ∀ σ τ ρ, σ ⩽ τ ∩ ρ -> (σ ⩽ τ) ∧ (σ ⩽ ρ).
Proof with auto with SubtypeHints.
  intros; split;
    etransitivity;
    try eassumption...
Qed.
```

The search engine can automatically simplify hypotheses if we told it so. For instance, if we know that $\sigma \leqslant \tau \cap \rho$, then we know automatically that $\sigma \leqslant \tau$ and $\sigma \leqslant \rho$. We can tell that to the search engine like this:

```
Hint Extern 1 (_ ⩽ _) ⇒
lazymatch goal with
| H : ?σ ⩽ ?τ ∩ ?ρ |- _ ⇒ apply Inter_inf' in H; destruct H
end : SubtypeHints.
```

The search engine gives a weight to each action in its search, so `Hint Extern 1 (_ ⩽ _)` states that this hint weighs 1 and should be used when we have to prove a subtyping relation.

5.2.1 Definition of normal forms

In order to talk about arbitrary intersection (or unions) of ANFs in Coq, we define the type `Generalize`:

```
Inductive Generalize (c : term -> term -> term) (P : term -> Prop) : term -> Prop :=
| G_nil : ∀ σ, P σ -> Generalize c P σ
| G_cons : ∀ σ τ, Generalize c P σ -> Generalize c P τ -> Generalize c P (c σ τ).
Hint Constructors Generalize : SubtypeHints.
Notation "[ ⋂ P ]" := (Generalize (∩ ) P).
Notation "[ ⋃ P ]" := (Generalize (∪ ) P).
```

Hence, the concept of ANF can easily be declared in Coq:

```
Inductive ANF : term -> Prop :=
| VarisANF : ∀ α, ANF (Var α)
| ArrowisANF : ∀ σ τ, [⋂ ANF] σ -> [⋃ ANF] τ -> ANF (σ → τ)
| ArrowisANF' : ∀ τ, [⋃ ANF] τ -> ANF (U → τ).
Hint Constructors ANF : SubtypeHints.
```

We define Conjunctive Arrow Normal Forms (CANF) and Disjunctive Arrow Normal Forms (DANF):

```
Definition CANF (σ : term) : Prop := [⋂ [⋃ ANF]] σ ∨ σ = U.
Definition DANF (σ : term) : Prop := [⋃ [⋂ ANF]] σ ∨ σ = U.
Hint Unfold CANF : SubtypeHints.
Hint Unfold DANF : SubtypeHints.
```

It is clear from the Coq definition that a term is in CANF if it is the intersection of unions of ANFs (or the type `U`), and that a term is in DANF if it is the union of intersection of ANFs (or the type `U`). We also define quasi-`U`-free types (or `Omega_free` is the Coq source), which are types where there is no useless occurrence of `U`, *i.e.* the only accepted occurrences are lone occurrences of `U` on the left-hand side of an arrow.

```
Inductive Omega_free : term -> Prop :=
| Of_Var : ∀ α, Omega_free (Var α)
| Of_Union : ∀ σ τ, Omega_free σ -> Omega_free τ -> Omega_free (σ ∪ τ)
| Of_Inter : ∀ σ τ, Omega_free σ -> Omega_free τ -> Omega_free (σ ∩ τ)
| Of_Arrow1 : ∀ σ, Omega_free σ -> Omega_free (U → σ)
| Of_Arrow2 : ∀ σ τ, Omega_free σ -> Omega_free τ -> Omega_free (σ → τ).
Hint Constructors Omega_free : SubtypeHints.
```

Of course, we can automatically decompose hypotheses stating that some non-atomic type is quasi-`U`-free, and the hypothesis that `U` is quasi-`U`-free is automatically absurd:

```
Hint Extern 1 ⇒
match goal with
| H : Omega_free U |- _ ⇒ inversion H
| H : Omega_free (_ _ _) |- _ ⇒ inv H
end : SubtypeHints.
```

It is quite tedious to prove that a term is in ANF (or CANF, or DANF), because we usually have to decompose the hypotheses and the goal as much as possible before applying trivial tactics. That is why we have defined a *ad hoc* tactic that tries as much as possible to prove that a term is in normal form:

```
Ltac decide_nf :=
  try (lazymatch goal with
       | H : ANF _ |- _ ⇒ idtac
       | H : Generalize _ _ _ |- _ ⇒ idtac
       | H : CANF _ |- _ ⇒ idtac
       | H : DANF _ |- _ ⇒ idtac
       | _ ⇒ fail
       end;
       repeat lazymatch goal with
              | H : DANF (_ _ _) |- _ ⇒
                inversion H as [?|H']; [| inversion H']; subst; clear H
              | H : CANF (_ _ _) |- _ ⇒
                inversion H as [?|H']; [| inversion H']; subst; clear H
              | H : [⋃ ANF] (_ ∪ _) |- _ ⇒
                inversion H as [? H'|]; [inversion H'|]; subst; clear H
              | H : [⋃ [⋂ ANF]] (_ ∪ _) |- _ ⇒
                inversion H as [? H'|];
                [inversion H' as [? H''|]; inversion H''|];
                subst; clear H
              | H : [⋃ _] (_ ∩ _) |- _ ⇒ inv H
              | H : [⋃ _] (_ → _) |- _ ⇒ inv H
              | H : [⋃ _] (Var _) |- _ ⇒ inv H
              | H : [⋃ _] U |- _ ⇒ inv H
              | H : [⋂ ANF] (_ ∩ _) |- _ ⇒
                inversion H as [? H'|]; [inversion H'|]; subst; clear H
              | H : [⋂ [⋃ ANF]] (_ ∩ _) |- _ ⇒
                inversion H as [? H'|];
                [inversion H' as [? H''|]; inversion H''|];
                subst; clear H
              | H : [⋂ _] (_ ∪ _) |- _ ⇒ inv H
              | H : [⋂ _] (_ → _) |- _ ⇒ inv H
              | H : [⋂ _] (Var _) |- _ ⇒ inv H
              | H : [⋂ _] U |- _ ⇒ inv H
              | H : ANF (U → _) |- _ ⇒
                inversion H as [|? ? H'|];
                [inversion H' as [? H''|]; inversion H''|];
                subst; clear H
              | H : ANF (_ → _) |- _ ⇒ inv H
              | H : ANF (_ ∩ _) |- _ ⇒ inversion H
              | H : ANF (_ ∪ _) |- _ ⇒ inversion H
              | H : ANF U |- _ ⇒ inversion H
```

```
            | H : Omega_free (_ _ _) |- _ ⇒ inv H
            end);
  repeat lazymatch goal with
        | H : ?x |- ?x ⇒ assumption
        | |- [⋃ _] (_ ∪ _) ⇒ apply G_cons
        | |- [⋃ _] _ ⇒ apply G_cons
        | |- [⋂ _] (_ ∩ _) ⇒ apply G_cons
        | |- [⋂ _] _ ⇒ apply G_nil
        | |- ANF (Var _) ⇒ constructor
        | |- ANF (U → _) ⇒ apply ArrowisANF'
        | |- ANF (_ → _) ⇒ constructor
        | |- CANF U ⇒ right; reflexivity
        | |- DANF U ⇒ right; reflexivity
        | |- CANF (Var _) ⇒ left; repeat constructor
        | |- DANF (Var _) ⇒ left; repeat constructor
        | |- CANF (_ _ _) ⇒ left
        | |- DANF (_ _ _) ⇒ left
        end.
Hint Extern 1 (CANF _) ⇒ decide_nf : SubtypeHints.
Hint Extern 1 (DANF _) ⇒ decide_nf : SubtypeHints.
Hint Extern 1 (ANF _) ⇒ decide_nf : SubtypeHints.
Hint Extern 1 (Generalize _ _ _) ⇒ decide_nf : SubtypeHints.
```

This tactic operates in three steps:

- for performance reasons, we first check that there is some hypothesis we can work with;
- we then simplify such hypotheses as much as possible;
- finally, we simplify the goal as much as possible, and we try to conclude.

5.2.2 Filters and ideals

The filter generated by a type σ is the set of all types greater than σ. The ideal generated by a type σ is the set of all types greater than σ. We define filters and ideals syntactically, in order to get a decision procedure. However, we cannot give a syntactical definition of filters or ideals for every type. Therefore we only define ideals for unions of ANFs, and we define filters for terms which have the syntax defined by the following predicate:

```
Unset Elimination Schemes.
Inductive isFilter : term -> Prop :=
| OmegaisFilter : isFilter U
| VarisFilter : ∀ α, isFilter (Var α)
| ArrowisFilter : ∀ σ τ, isFilter (σ → τ)
| InterisFilter : ∀ σ τ, isFilter σ -> isFilter τ -> isFilter (σ ∩ τ).
Set Elimination Schemes.
Hint Constructors isFilter : SubtypeHints.
```

We can now define filters and ideals:

```
Reserved Notation "↑[ σ ] τ" (at level 65).
Reserved Notation "↓[ σ ] τ" (at level 65).
Inductive Filter : term -> term -> Prop :=
```

```
| F_Refl : ∀ σ : term, isFilter σ -> ↑[σ] σ
| F_Inter : ∀ σ τ ρ : term, ↑[σ] τ -> ↑[σ] ρ -> ↑[σ] τ ∩ ρ
| F_Union1 : ∀ σ τ ρ : term, ↑[σ] τ -> ↑[σ] τ ∪ ρ
| F_Union2 : ∀ σ τ ρ : term, ↑[σ] ρ -> ↑[σ] τ ∪ ρ
| F_Arrow1 : ∀ σ1 σ2 τ1 τ2 : term, σ2 ⩽ σ1 -> τ1 ⩽ τ2 -> ↑[σ1 → τ1] σ2 → τ2
| F_Arrow2 : ∀ σ1 σ2 τ1 τ2 ρ1 ρ2 : term, ↑[σ1 ∩ σ2] τ1 → ρ1 -> τ2 ⩽ τ1 ->
                                          ρ1 ⩽ ρ2 -> ↑[σ1 ∩ σ2] τ2 → ρ2
| F_OmegaTopV : ∀ (α : 𝕍.t) (τ : term), ↑[U] τ -> ↑[Var α] τ
| F_OmegaTopA : ∀ σ1 σ2 τ : term, ↑[U] τ -> ↑[σ1 → σ2] τ
| F_OmegaTopI : ∀ σ1 σ2 τ : term, isFilter (σ1 ∩ σ2) -> ↑[U] τ -> ↑[σ1 ∩ σ2] τ
| F_Omega : ∀ σ τ : term, ↑[U] τ -> ↑[U] σ → τ
| F_Inter1 : ∀ σ1 σ2 τ : term, isFilter σ2 -> ↑[σ1] τ -> ↑[σ1 ∩ σ2] τ
| F_Inter2 : ∀ σ1 σ2 τ : term, isFilter σ1 -> ↑[σ2] τ -> ↑[σ1 ∩ σ2] τ
| F_ArrowInter : ∀ σ1 σ2 τ ρ1 ρ2 : term, ↑[σ1 ∩ σ2] (τ → ρ1) ∩ (τ → ρ2) ->
                                          ↑[σ1 ∩ σ2] τ → ρ1 ∩ ρ2
| F_ArrowUnion : ∀ σ1 σ2 τ1 τ2 ρ : term, ↑[σ1 ∩ σ2] (τ1 → ρ) ∩ (τ2 → ρ) ->
                                          ↑[σ1 ∩ σ2] τ1 ∪ τ2 → ρ
where "↑[σ ] τ" := (Filter σ τ).
Hint Constructors Filter : SubtypeHints.

Inductive Ideal : term -> term -> Prop :=
| I_Refl : ∀ σ : term, [⋃ ANF] σ -> ↓[σ] σ
| I_Inter1 : ∀ σ τ ρ : term, ↓[σ] τ -> ↓[σ] τ ∩ ρ
| I_Inter2 : ∀ σ τ ρ : term, ↓[σ] ρ -> ↓[σ] τ ∩ ρ
| I_Union : ∀ σ τ ρ : term, ↓[σ] τ -> ↓[σ] ρ -> ↓[σ] τ ∪ ρ
| I_Arrow1 : ∀ σ1 σ2 τ1 τ2 : term, [⋂ ANF] σ1 -> ↑[σ1] σ2 -> ↓[τ1] τ2 ->
                                    ↓[σ1 → τ1] σ2 → τ2
| I_Arrow2 : ∀ σ τ1 τ2 : term, ↑[U] σ -> ↓[τ1] τ2 -> ↓[U → τ1] σ → τ2
| I_Union1 : ∀ σ1 σ2 τ : term, [⋃ ANF] σ2 -> ↓[σ1] τ -> ↓[σ1 ∪ σ2] τ
| I_Union2 : ∀ σ1 σ2 τ : term, [⋃ ANF] σ1 -> ↓[σ2] τ -> ↓[σ1 ∪ σ2] τ
where "↓[σ ] τ" := (Ideal σ τ).
Hint Constructors Ideal : SubtypeHints.
```

5.2.3 Induction scheme for filters and ideals

When we reason about a filter ↑[σ], we usually do an induction on the predicate `isFilter` σ, so that we only consider the case where σ is syntactically a type for which we have defined filters. Coq usually automatically generates an induction scheme for declared inductive types. For `isFilter`, the induction scheme would be:

```
isFilter_ind : ∀ P : term -> Prop,
        P U ->
        (∀  α : 𝕍.t, P (Var α)) ->
        (∀  σ τ : term, P (σ → τ)) ->
        (∀  σ τ : term, isFilter σ -> P σ -> isFilter τ -> P τ -> P (σ ∩ τ)) ->
         ∀ σ : term, isFilter σ -> P σ.
```

However, this induction scheme has a small problem. If we want to prove `P` σ with the hypothesis `H1` : ↑[σ] τ, we can deduce that σ verify the property `isFilter` σ, therefore we do an induction on this predicate, as there are four constructors, we get four subcases. The problem arises in the second subcase:

- the first case is `OmegaisFilter`: we have to prove `P U` under the hypothesis `H1 : ↑[U] τ`;

- the second case is `VarisFilter`: we have to prove `P (Var α` under the hypothesis `H1 : ↑[Var α] τ`. Here we can apply the tactic `inversion H1`, which generate as many subcases as there are possible constructors for `H1`. Among these constructors, there is `F_OmegaTopV : ∀ (α : 𝕍.t) (τ : term), ↑[U] τ -> ↑[Var α] τ`, in which case the hypothesis `H1` becomes `H1 : ↑[U] τ`. From the case `OmegaisFilter`, we know that `P U`, but Coq has not remembered it. In order not to rewrite the proof of `P U`, It would be natural to consider that `P U` is a kind of induction hypothesis.

Coq allows us to define our own induction scheme for `isFilter`, by preventing it from automatically generate the induction scheme[2], and by proving a lemma having the name `isFilter_ind`.

```
Lemma isFilter_ind : ∀ P : term -> Prop,
      P U ->
      (∀ α : 𝕍.t, P U -> P (Var α)) ->
      (∀ σ τ : term, P U -> P (σ → τ)) ->
      (∀ σ τ : term, isFilter σ -> P σ -> isFilter τ -> P τ -> P U -> P (σ ∩ τ)) ->
       ∀ σ : term, isFilter σ -> P σ.
```

Ideals are defined for unions of ANFs, which is, *per se*, not an inductive type, but we may want nonetheless to have an induction scheme for it:

```
Lemma Uanf_ind : ∀ P : term -> Prop,
   (∀ α, P (Var α)) ->
   (∀ σ τ, P σ -> P τ -> P (σ ∪ τ)) ->
   (∀ σ τ, P τ -> P (σ → τ)) ->
   (∀ σ, [⋃ ANF] σ -> P σ).
```

However, the `induction` tactic will not call the `Uanf_ind` lemma, because the type `Uanf` does not exist. We bypass this issue with a handmade tactic which has approximately the same behavior:

```
Ltac uanf_ind σ :=
  let foo HH :=
      repeat match goal with
             | H : context[σ] |- _ ⇒ lazymatch H with
                                      | HH ⇒ fail
                                      | _ ⇒ revert H
                                      end
             end;
      revert HH; revert σ;
      refine (Uanf_ind _ _ _ _); intros
  in
  lazymatch goal with
  | HH : [⋃ ANF] σ |- _ ⇒ foo HH
  | _ ⇒
    assert (HH : [⋃ ANF] σ) by (auto with SubtypeHints);
    foo HH
  end.
```

[2]Thanks to the command `Unset Elimination Schemes`.

The tactic `uanf_ind` σ either finds the hypothesis `HH` : [$\bigcup$ `ANF`] σ or creates it, and then reverts all the hypotheses containing σ back into the goal (including `HH`), in such a way that we get a goal of the form $\forall$ σ : `term`, [$\bigcup$ `ANF`] σ -> `A` (where `A` is some Coq expression). By then applying `refine` (`Uanf_ind _ _ _ _`), Coq applies Lemma `Uanf_ind` and tries to fill the four wildcards. The first wildcard is a term `P` : `term` -> `Prop` such that `P` σ is convertible with `A`.

Of course, the Coq refiner does not create the constant function `fun x : term` $\Rightarrow$ `A`, but captures all the free occurrences of σ in `A` and creates the most general function possible `fun` σ : `term` $\Rightarrow$ `A` (where the free σ in `A` are captured by the abstraction). The three remaining wildcards are a term of type $\forall$ α, `P` (`Var` α), another term of type $\forall$ σ τ, `P` σ -> `P` τ -> `P` ($\sigma \cup \tau$), and a third term of type $\forall$ σ τ, `P` τ -> `P` ($\sigma \to \tau$), and the Coq refiner usually cannot create them automatically. These missing terms become new goals for the user. We then introduce as much hypotheses as possible, using the `intros` tactic. This refinement process is what Coq does normally whenever the user applies the `induction` tactic.

5.2.4 Properties of filters and ideals

We first prove that filters and ideals correspond to subtyping.

```
Theorem Filter_correct : ∀ σ τ, ↑[σ] τ -> σ ⩽ τ.
Theorem Ideal_correct : ∀ σ τ, ↓[σ] τ -> τ ⩽ σ.
```

The proofs are done either by induction on the rules of filters, or by induction on the rules of ideals. We then prove that filters are only defined for types verifying the predicate `isFilter`, and ideals are only defined for unions of ANFs. The proofs are also done either by induction on the rules of filters, or by induction on the rules of ideals:

```
Lemma Filter_isFilter: ∀ σ τ, ↑[σ] τ -> isFilter σ.
Lemma Ideal_isDANF: ∀ σ τ, ↓[σ] τ -> [⋃ ANF] σ.
```

Then we can prove that intersection and unions can be nicely decomposed inside filters and ideals:

```
Lemma FilterInter : ∀ σ τ ρ, ↑[σ] τ ∩ ρ -> ↑[σ] τ ∧ ↑[σ] ρ.
Lemma IdealInter : ∀ σ τ ρ, ↓[σ] τ ∩ ρ -> ↓[σ] τ ∨ ↓[σ] ρ.
Lemma FilterUnion : ∀ σ τ ρ, ↑[σ] τ ∪ ρ -> ↑[σ] τ ∨ ↑[σ] ρ.
Lemma IdealUnion : ∀ σ τ ρ, ↓[σ] τ ∪ ρ -> ↓[σ] τ ∧ ↓[σ] ρ.
```

The proofs are done either by induction on the predicate `isFilter` (for filters), or by induction on the predicate [$\bigcup$ `ANF`] (for ideals).

The following lemma is a trivial simplification of the construction rules of `Filter` for the case of arrows:

```
Lemma FilterArrow : ∀ σ σ' τ τ', ↑[σ → σ'] τ → τ' -> (↑[U] τ → τ' ∨ (τ ⩽ σ ∧ σ' ⩽ τ')).
```

We can prove that ↑[σ] $\tau \to \rho$' by contravariance of the domain and covariance of the codomain:

```
Lemma FilterArrow' : ∀ σ τ' ρ, ↑[σ] τ' → ρ -> ∀ τ ρ', τ ⩽ τ' -> ρ ⩽ ρ' -> ↑[σ] τ → ρ'.
```

The proof is done by induction on the predicate `isFilter`.

Now we prove that every filter contains at least the filter ↑[`U`], and ideals never contain `U`, nor `U` $\to$ `U`.

Lemma FilterOmega : ∀ σ τ, isFilter σ -> ↑[U] τ -> ↑[σ] τ.
Lemma IdealnoOmega : ∀ σ, ¬ ↓[σ] U.
Lemma IdealnoOmegaArrow : ∀ σ, ¬ ↓[σ] U → U.

Lemma FilterOmega is proved by induction on the predicate isFilter, while Lemma IdealnoOmega is simply proved by induction on σ. Lemma IdealnoOmegaArrow is also proved by induction on σ, with one non-trivial case: if σ is some type $\sigma_1 \to \sigma_2$ (corresponding to the rules I_Arrow1 or I_Arrow2), then, by induction hypothesis, we have that $\downarrow[\sigma_2]$ U, which is absurd because of the IdealnoOmegaArrow lemma.
We prove that filters are upward-closed:

Lemma Filter_closed : ∀ σ τ1 τ2, ↑[σ] τ1 -> τ1 ⩽ τ2 -> ↑[σ] τ2.

The proof is done by induction on the subtyping rules. The interesting cases are the following:

- Rule R_CoContra : ∀ σ σ' τ τ', σ ⩽ σ' -> τ ⩽ τ' -> σ' → τ ⩽ σ → τ': we know, by hypothesis, that $\uparrow[\sigma]\ \sigma_2 \to \tau_1$, $\sigma_1 \leqslant \sigma_2$ and $\tau_1 \leqslant \tau_2$ and we need to show that $\uparrow[\sigma]\ \sigma_1 \to \tau_2$. Moreover, by induction hypothesis, we know that $\uparrow[\sigma]\ \sigma_1$ -> $\uparrow[\sigma]\ \sigma_2$ and that $\uparrow[\sigma]\ \tau_1$ -> $\uparrow[\sigma]\ \tau_2$. We conclude by contravariance of the domain and covariance of the codomain, which is given by Lemma FilterArrow';

- Rules R_OmegaTop : ∀ σ, σ ⩽ U and R_OmegaArrow : U ⩽ U → U: we need to show that $\uparrow[\sigma]$ U and $\uparrow[\sigma]$ U → U. Using Lemma FilterOmega, we only have to show that $\uparrow$[U] U and $\uparrow$[U] U → U.

For the other cases, we reason by induction on isFilter σ.

We can then deduce the completeness of filters, *i.e.* for any type σ verifying isFilter σ, we have that $\uparrow[\sigma]\ \tau$ if $\sigma \leqslant \tau$:

Theorem Filter_complete : ∀ σ, isFilter σ -> ∀ τ, σ ⩽ τ -> ↑[σ] τ.

The proof follows trivially by applying Lemma Filter_closed. Similarly, we prove that ideals are downward-closed:

Lemma Ideal_closed : ∀ σ, [⋃ ANF] σ -> ∀ τ1, ↓[σ] τ1 -> ∀ τ2, τ2 ⩽ τ1 -> ↓[σ] τ2.

The proof is done by induction on the fact that σ is an union of ANFs (using the uanf_ind tactic). We can then deduce that ideals are complete:

Theorem Ideal_complete : ∀ σ, [⋃ ANF] σ -> ∀ τ, τ ⩽ σ -> ↓[σ] τ.

The proof follows trivially by applying Lemma Ideal_closed.

5.3 Coq implementation of the subtyping algorithm

We implement the subtyping algorithm by mixing executable code and proofs: each function takes as input some data and possibly some proof on the data, then returns some data along with the proof the data verify some specification. The code alone is then extracted in OCaml. This technique is called *strong specification* in [15].

For instance, the function deleteOmega takes a type σ and returns a type τ along with a proof that τ is equivalent to σ while being either quasi-U-free or syntactically equal to U. The implementation of the function is done in two parts:

- first, the computational part of the algorithm is given inside the refine tactic. Most of the proofs are not given, instead we put a wildcard (_);

– finally, the Coq refiner tries to either fill the wildcards or to generate the corresponding goal, therefore all the proofs are postponed to the second part of the implementation. The proofs are then written the usual way.

```
Fixpoint deleteOmega (σ : term) : {τ | τ ∼ σ ∧ (Omega_free τ ∨ τ = U)}.
(* algorithmic part *)
refine(match σ with
| σ → τ ⇒ let (σ,pfσ) := deleteOmega σ in
let (τ,pfτ) := deleteOmega τ in
match τ as x return τ = x -> _ with
| U ⇒ λ _, exist _ U _
| _ ⇒ λ _, exist _ (σ → τ) _
end eq_refl
| σ ∩ τ ⇒ let (σ,pfσ) := deleteOmega σ in
let (τ,pfτ) := deleteOmega τ in
match σ as x return σ = x -> _ with
| U ⇒ λ _, exist _ τ _
| _ ⇒ λ _, match τ as x return τ = x -> _ with
| U ⇒ λ _, exist _ σ _
| _ ⇒ λ _, exist _ (σ ∩ τ) _
end eq_refl
end eq_refl
| σ ∪ τ ⇒ let (σ,pfσ) := deleteOmega σ in
let (τ,pfτ) := deleteOmega τ in
match σ as x return σ = x -> _ with
| U ⇒ λ _, exist _ U _
| _ ⇒ λ _, match τ as x return τ = x -> _ with
| U ⇒ λ _, exist _ U _
| _ ⇒ λ _, exist _ (σ ∪ τ) _
end eq_refl
end eq_refl
| Var α ⇒ exist _ (Var α) _
| U ⇒ exist _ U _
end);
(* logical part *)
clear deleteOmega; subst; simpl in *;
first[destruct pfσ as [? [||]];
  destruct pfτ as [? [||]]; subst|
  auto with SubtypeHints];
first[match goal with | H : Omega_free U |- _ ⇒ inversion H end|
  discriminate|
  split; auto with SubtypeHints].
Defined.
```

The algorithms that rewrite types in CANF and DANF are quite heavy, so we define helper functions for the arrow, union, and intersection cases.

```
Fixpoint distrArrow (σ τ : term) (pfσ : [⋃ [⋂ ANF]] σ ∨ σ = U) (pfτ : [⋂ [⋃ ANF]] τ) :
  {σ' | σ' ∼ σ → τ ∧ [⋂ ANF] σ'}.
  refine(match σ as x return σ = x -> _ with
        | σ1 ∪ σ2 ⇒
          λ _, let (σ1,pfσ1) := distrArrow σ1 τ _ _ in
```

```
                    let (σ2,pfσ2) := distrArrow σ2 τ _ _ in
                    exist _ (σ1 ∩ σ2) _
        | _ ⇒
          λ _,
          (fix distrArrow' σ τ (pfσ:[⋂ ANF] σ ∨ σ = U) (pfτ:[⋂ [⋃ ANF]] τ) :
             {σ' | σ' ∼ σ → τ ∧ [⋂ ANF] σ'} :=
             match τ as x return τ = x -> _ with
             | τ1 ∩ τ2 ⇒
               λ _, let (τ1,pfτ1) := distrArrow' σ τ1 _ _ in
                         let (τ2,pfτ2) := distrArrow' σ τ2 _ _ in
                         exist _ (τ1 ∩ τ2) _
             | _ ⇒ λ _, exist _ (σ → τ) _
             end eq_refl) σ τ _ pfτ
        end eq_refl); subst; (destruct pfσ; [|try discriminate]); simpl in *;
    auto with SubtypeHints.
Defined.

Fixpoint distrUnion (σ τ : term) (pfσ : [⋂ [⋃ ANF]] σ) (pfτ : [⋂ [⋃ ANF]] τ) :
  {σ' | σ' ∼ σ ∪ τ ∧ [⋂ [⋃ ANF]] σ'}.
  refine(match σ as x return σ = x -> _ with
        | σ1 ∩ σ2 ⇒
          λ _, let (σ1,pfσ1) := distrUnion σ1 τ _ _ in
                    let (σ2,pfσ2) := distrUnion σ2 τ _ _ in
                    exist _ (σ1 ∩ σ2) _
        | _ ⇒
          λ _,
          (fix distrUnion' σ τ (pfσ:[⋃ ANF] σ) (pfτ:[⋂ [⋃ ANF]] τ) :
             {σ' | σ' ∼ σ ∪ τ ∧ [⋂ [⋃ ANF]] σ'} :=
             match τ as x return τ = x -> _ with
             | τ1 ∩ τ2 ⇒
               λ _, let (τ1,pfτ1) := distrUnion' σ τ1 _ _ in
                         let (τ2,pfτ2) := distrUnion' σ τ2 _ _ in
                         exist _ (τ1 ∩ τ2) _
             | _ ⇒ λ _, exist _ (σ ∪ τ) _
             end eq_refl) σ τ _ pfτ
        end eq_refl); subst; simpl in *;
    auto with SubtypeHints.
Defined.

Fixpoint distrInter (σ τ : term) (pfσ : [⋃ [⋂ ANF]] σ) (pfτ : [⋃ [⋂ ANF]] τ) :
  {σ' | σ' ∼ σ ∩ τ ∧ [⋃ [⋂ ANF]] σ'}.
  refine(match σ as x return σ = x -> _ with
        | σ1 ∪ σ2 ⇒
          λ _, let (σ1,pfσ1) := distrInter σ1 τ _ _ in
                    let (σ2,pfσ2) := distrInter σ2 τ _ _ in
                    exist _ (σ1 ∪ σ2) _
        | _ ⇒
          λ _,
          (fix distrInter' σ τ (pfσ:[⋂ ANF] σ) (pfτ:[⋃ [⋂ ANF]] τ) :
             {σ' | σ' ∼ σ ∩ τ ∧ [⋃ [⋂ ANF]] σ'} :=
             match τ as x return τ = x -> _ with
```

```
            | τ1 ∪ τ2 ⇒
              λ _, let (τ1,pfτ1) := distrInter' σ τ1 _ _ in
                       let (τ2,pfτ2) := distrInter' σ τ2 _ _ in
                       exist _ (τ1 ∪ τ2) _
            | _ ⇒
              λ _, exist _ (σ ∩ τ) _
            end eq_refl) σ τ _ pfτ
        end eq_refl); subst; simpl in *;
    auto with SubtypeHints.
Defined.
```

Now, we can implement the functions that rewrite a term in CANF or DANF. These functions are mutually recursive:

```
Fixpoint _CANF (σ : term) : (Omega_free σ ∨ σ = U) -> {τ | τ ~ σ ∧ CANF τ}
with _DANF (σ : term) : (Omega_free σ ∨ σ = U) -> {τ | τ ~ σ ∧ DANF τ}.
Proof.
  - refine(match σ with
          | Var α ⇒ λ _, exist _ (Var α) _
          | σ → τ ⇒ λ pf,
                      let (σ,pfσ) := _DANF σ _ in
                      let (τ,pfτ) := _CANF τ _ in
                      let (σ',pfσ') := distrArrow σ τ _ _ in
                      exist _ σ' _
          | σ ∩ τ ⇒ λ pf,
                      let (σ,pfσ) := _CANF σ _ in
                      let (τ,pfτ) := _CANF τ _ in
                      exist _ (σ ∩ τ) _
          | σ ∪ τ ⇒ λ pf, let (σ,pfσ) := _CANF σ _ in
                          let (τ,pfτ) := _CANF τ _ in
                          let (σ',pfσ') := distrUnion σ τ _ _ in
                          exist _ σ' _
          | U ⇒ λ _, exist _ U _
          end); try (destruct pf; [|discriminate]); simpl in *;
      match goal with
      | |- _ ∨ _ ⇒ auto with SubtypeHints
      | |- _ ∧ _ ⇒ split; [trivial|]
      | _ ⇒ idtac
      end;
      try (destruct pfσ as [Hσ [?|?]];
          [| subst; exfalso; match type of Hσ with
                              | U ~ ?σ' ⇒ apply (Omega_free_Omega σ')
                              end; auto 2 with SubtypeHints; fail]);
      try (destruct pfτ as [Hτ [?|?]];
          [| subst; exfalso; match type of Hτ with
                              | U ~ ?τ' ⇒ apply (Omega_free_Omega τ')
                              end; auto 2 with SubtypeHints; fail]);
      auto with SubtypeHints.
  - refine(match σ with
          | Var α ⇒ λ _, exist _ (Var α) _
          | σ → τ ⇒ λ pf,
                      let (σ,pfσ) := _DANF σ _ in
                      let (τ,pfτ) := _CANF τ _ in
```

```
                    let (σ',pfσ') := distrArrow σ τ _ _ in
                    exist _ σ' _
          | σ ∪ τ ⇒ λ pf,
                    let (σ,pfσ) := _DANF σ _ in
                    let (τ,pfτ) := _DANF τ _ in
                    exist _ (σ ∪ τ) _
          | σ ∩ τ ⇒ λ pf,
                    let (σ,pfσ) := _DANF σ _ in
                    let (τ,pfτ) := _DANF τ _ in
                    let (σ',pfσ') := distrInter σ τ _ _ in
                    exist _ σ' _
          | U ⇒ λ _, exist _ U _
          end); try (destruct pf; [|discriminate]); simpl in *;
      match goal with
      | |- _ ∨ _ ⇒ auto with SubtypeHints
      | |- _ ∧ _ ⇒ split; [trivial|]
      | _ ⇒ idtac
      end;
      try (destruct pfσ as [Hσ [?|?]];
          [| subst; exfalso; match type of Hσ with
                          | U ~ ?σ' ⇒ apply (Omega_free_Omega σ')
                          end; auto 2 with SubtypeHints; fail]);
      try (destruct pfτ as [Hτ [?|?]];
          [| subst; exfalso; match type of Hτ with
                          | U ~ ?τ' ⇒ apply (Omega_free_Omega τ')
                          end; auto 2 with SubtypeHints; fail]);
      auto with SubtypeHints.
Defined.
```

The algorithm $\mathcal{A}$ (called `main_algo` in the Coq source) is difficult to implement: recursive functions in Coq may call themselves recursively only if some argument is structurally decreasing, but this has to be the same argument for every recursive call. However, in this algorithm either the first or the second argument decreases during a recursive call.

The usual workaround is to add an extra argument to the function that will structurally decrease. Hopefully, the Coq standard library has some functions already implemented to help us. We define a measure on the types, and prove this measure cannot infinitely decrease:

```
(* measure on the types *)
Fixpoint size (σ : term) : nat :=
  match σ with
  | Var α ⇒ 0
  | σ → τ ⇒ S((size σ) + (size τ))
  | σ ∩ τ ⇒ S((size σ) + (size τ))
  | σ ∪ τ ⇒ S((size σ) + (size τ))
  | U ⇒ 0
  end.
Definition pair_size (x : term * term) : nat :=
  let (s,t) := x in size s + size t.

(* Well-founded principle for the main algorithm *)
Definition main_algo_order : relation (term * term) :=
```

```
  λ x y, pair_size x < pair_size y.
Definition wf_main_algo : well_founded main_algo_order := well_founded_ltof _ _.
```

Now we can implement the algorithm, thanks to the `Fix` function. Among the generated goals, we have to prove that the measure on the argument indeed decreases (it is the `|- main_algo_order _ _` case).

```
Definition main_algo : ∀ pair : term * term,
    DANF (fst pair) -> CANF (snd pair) ->
    {fst pair ⩽ snd pair} + {¬ fst pair ⩽ snd pair}.
  refine (Fix wf_main_algo _ _). intros [σ τ] rec.
  refine (match (σ,τ) as x return x = (σ,τ) -> _ with
          | (_, U) ⇒ λ eq _ _, left _
          | (U, _) ⇒ λ eq _ Cτ, right _
          | (σ1 ∪ σ2, _) ⇒ λ eq _ _, match rec (σ1,τ) _ _ _ with
                                  | left _ ⇒ match rec (σ2,τ) _ _ _ with
                                              | left _ ⇒ left _
                                              | right _ ⇒ right _
                                              end
                                  | right _ ⇒ right _
                                  end
          | (_, τ1 ∩ τ2) ⇒ λ eq _ _, match rec (σ,τ1) _ _ _ with
                                  | left _ ⇒ match rec (σ,τ2) _ _ _ with
                                              | left _ ⇒ left _
                                              | right _ ⇒ right _
                                              end
                                  | right _ ⇒ right _
                                  end
          | (σ1 → σ2, τ1 → τ2) ⇒ λ eq Dσ Cτ, match rec (τ1,σ1) _ _ _ with
                                        | left _ ⇒ match rec (σ2,τ2) _ _ _ with
                                                    | left _ ⇒ left _
                                                    | right HAA ⇒ right _
                                                    end
                                        | right HAA ⇒ right _
                                        end
          | (σ1 ∩ σ2, _) ⇒ λ eq Dσ Cτ, match rec (σ1,τ) _ _ _ with
                                  | left _ ⇒ left _
                                  | right _ ⇒ match rec (σ2,τ) _ _ _ with
                                              | left _ ⇒ left _
                                              | right _ ⇒ right _
                                              end
                                  end
          | (_, τ1 ∪ τ2) ⇒ λ eq Dσ Cτ, match rec (σ,τ1) _ _ _ with
                                  | left _ ⇒ left _
                                  | right _ ⇒ match rec (σ,τ2) _ _ _ with
                                              | left _ ⇒ left _
                                              | right _ ⇒ right _
                                              end
                                  end
          | (Var α, Var β) ⇒ λ eq _ _, if 𝕍.eq_dec α β then left _ else right _
          | _ ⇒ λ eq _ _, right _
          end eq_refl); inv eq; simpl in *;
    match goal with
```

```
  | |- main_algo_order _ _ ⇒ red; simpl; omega
  | |- ?σ ⩽ ?σ ⇒ reflexivity
  | H : ?x |- ?x ⇒ assumption
  | |- CANF _ ⇒ auto with SubtypeHints
  | |- DANF _ ⇒ auto with SubtypeHints
  (* Correctness *)
  | |- _ ⩽ U ⇒ auto with SubtypeHints
  | |- _ ⩽ _ ∩ _ ⇒ auto with SubtypeHints
  | |- _ ∪ _ ⩽ _ ⇒ auto with SubtypeHints
  | |- _ ∩ _ ⩽ _ ⇒ apply Inter_inf_dual; auto
  | |- _ ⩽ _ ∪ _ ⇒ apply Union_sup_dual; auto
  | |- _ → _ ⩽ _ → _ ⇒ apply R_CoContra; trivial
  (* Completeness *)
  | |- ¬ U ⩽ _ ⇒ apply Omega_IUANF; auto with SubtypeHints
  | |- ¬ _ ∪ _ ⩽ _ ⇒ intro; apply Union_sup' in H; auto
  | |- ¬ _ ⩽ _ ∩ _ ⇒ intro; apply Inter_inf' in H; auto
  | |- ¬ ?σ ⩽ _ ⇒ intro H; apply Ideal_complete in H; [|auto with SubtypeHints];
                    match σ with
                    | _ ∩ _ ⇒
                      apply IdealInter in H; inversion H as [H'|H'];
                        apply Ideal_correct in H'; auto
                    | _ → _ ⇒
                      inv H; [apply HAA; reflexivity| |]; auto with SubtypeHints
                    | _ ⇒ inv H; auto with SubtypeHints
                    end
  end.
Defined.
```

The final part is straightforward: we compose the previous algorithms without any difficulty:

```
Definition decide_subtype : ∀ σ τ, {σ ⩽ τ} + {¬ σ ⩽ τ}.
Proof.
  intros.
  refine (let (σ1,pfσ) := deleteOmega σ in let (Hσ1,pfσ) := pfσ in
          let (τ1,pfτ) := deleteOmega τ in let (Hτ1,pfτ) := pfτ in
          let (σ2,pfσ) := _DANF σ1 pfσ in let (Hσ2,pfσ) := pfσ in
          let (τ2,pfτ) := _CANF τ1 pfτ in let (Hτ2,pfτ) := pfτ in
          match main_algo (σ2,τ2) pfσ pfτ with
          | left H ⇒ left _
          | right H ⇒ right _
          end);
    rewrite ← Hτ1, ← Hσ1, ← Hτ2, ← Hσ2; assumption.
Defined.
```

5.4 Extracting the subtyping algorithm in OCaml

The extraction system of Coq is rather straightforward. We can replace the default extraction of inductive types with the `Extract Inductive` command. For instance, the Coq cartesian product `prod` can be replaced with the OCaml product. Non-inductive terms can be instantiated with `Extract Constant`. For instance, `eq_dec`, which is an uninstantiated

parameter in the source code, can be instantiated with the OCaml equality function. Moreover, there is no need to explicitly create a function `eq_dec` in OCaml, so we can tell Coq to inline it, with `Extraction Inline BDdL.eq_dec`. Here is the full `ExtractOcaml.v` source file:

```
Require Import Filter.
Require Extraction.

Module BDdL := VariableAlphabet <+ Types.

Extract Inductive prod => "(*)" [ "(,)" ].
Extract Constant fst => "fst".
Extract Constant snd => "snd".
Extract Inductive sumbool => "bool" [ "true" "false" ].
Extract Inductive sig => "" [ "" ].
Extract Constant BDdL.t => "int".
Extract Constant BDdL.eq_dec => "(fun x y -> x = y)".
Extraction Inline BDdL.eq_dec.

Extraction Language Ocaml.
Extraction "BDdL.ml" BDdL.SubtypeRelation.
```

5.5 The preorder tactic

We explain how we have implemented a *heuristic* tactic solving automatically goals involving preorders. The tactic proceeds by reflection, mainly using techniques given in [26]. Reflection has two components:

- *Reification*: the goal is converted into a data structure. This component is done with the Ltac language of Coq;
- *Denotation*: the data structure is converted back into a goal. This component is done with the Gallina language of Coq.

The denotation of the reification of a goal should be the original goal itself. In our case, we have a tactic `quote_formula` which change the goal into a term `denote_formula R c f`, where `R` is the preorder relation, `f` is the reification of the goal, and `c` is a vector which helps reconstruct the original goal. A function `preorder_heuristic` processes recursively the argument `f` and returns either a proof of `denote_formula R c f` or nothing. Then the `preorder` tactic:

```
Ltac preorder :=
  intros;
  match goal with
  | |- ?R _ _ => quote_formula R;
                 match goal with
                 | |- denote_formula R ?c ?f =>
                   exact (partialOut (preorder_heuristic _ c f))
                 end
  | _ => fail "preorder tactic unsuccessful"
  end.
```

5.5.1 Denotation

Let's assume we have a preorder relation $<$, some terms $t_1, \ldots, t_n, u_1, \ldots, u_n$ and a goal of the form:

$$\Gamma; H_1 : t_1 < u_1; H_2 : t_2 < u_2 \vdash t_n < u_n$$

The goal may have extra hypotheses that are irrelevant. The goal is then transformed into:

$$\Gamma \vdash t_1 < u_1 \to t_2 < u_2 \to \ldots \to t_n < u_n$$

We then store all the t_i and u_i in a vector (*i.e.* a sized list) v of size n, and reify the following formula as a list of pairs of indices:

$$t_1 < u_1 \to t_2 < u_2 \to \ldots \to t_n < u_n$$

The indices are the position of the terms in v. With the vector v and the list of pairs of indices, we can reconstruct the original formula.

We need to easily get a term in the vector from an index, but we don't want any undefined behavior in case the index is out-of-bound. Therefore, we implement vectors the following way:

```
Inductive vector (A : Type) : nat -> Type :=
| vnil : vector A 0
| vcons : forall n : nat, A -> vector A n -> vector A (S n).
```

An index is a natural number m along with a proof that m is smaller than the size of the vector.

```
Definition index (n : nat) := {m : nat | m < n}.
```

We consider that two indices are equivalent if they correspond to the same natural number:

```
Definition index_to_nat {n : nat} (i : index n) : nat :=
  match i with
  | exist _ i _ => i
  end.

(* Decidable proof-irrelevant equality for indexes *)
Definition index_eq {n : nat} (i j : index n) := index_to_nat i = index_to_nat j.
Local Notation "i == j" := (index_eq i j) (at level 70).
Definition index_eq_dec {n : nat} : forall i j : index n, {i == j} + {~ i == j}.
Proof.
  intros; unfold index_eq.
  decide equality.
Defined.
```

The function `_get` takes as input a vector `v` of size `n`, an natural number `i` and a proof that `i < n`, and returns the element of the vector at position `n`.

```
Fixpoint _get {A : Type} {n : nat} (v : vector A n) {i : nat} {struct v} : i < n -> A.
  refine (match v in vector _ n' return i < n' -> A with
          | vnil _ => fun p : i < 0 => _
          | vcons x v' => match i as x return x < _ -> A with
                          | 0 => fun _ => x
                          | S j => fun p : _ => _get A _ v' j _
                          end
        end).
```

The refiner manages to fill all the wildcards, except two of them, which become goals we have to solve on our own:

- case `vnil _`, where the vector is empty: since we have a proof `p : i < 0`, we can conclude by contradiction;
- case `S j`, where the vector is not empty and we do not want the element at position 0: this is the recursive case, but we need an arithmetical proof that the updated index is still valid.

These two logical steps are done interactively using Ltac:

```
- exfalso; inversion p.
- apply Lt.lt_S_n; assumption.
Defined.
```

Now we can write a simple and safe function `get`, such that `get v i` returns the term at index `i` in the vector `v`:

```
Definition get {A : Type} {n : nat} (v : vector A n) (i : index n) : A :=
  match i with
  | exist _ _ p ⇒ _get v p
  end.
```

Of course, if two indices `i` and `j` are equivalent, then for any vector `v`, `get v i = get v j`:

```
Lemma get_irrelevant : forall A n (v : vector A n) (i j : index n),
    i == j -> get v i = get v j.
```

A formula is reified as a list of pairs of indices for the hypotheses, along with a pair of indices for the conclusion:

```
Definition formula (n : nat) : Set := (list (index n * index n)) * (index n * index n).
```

The denotation of a formula is straightforward: we `get` back every term and reconstruct the original formula.

```
Definition denote_formula {n : nat} (v : vector A n) (f : formula n) : Prop :=
  match f with
  | (l, (i,j)) ⇒
    (fix loop l :=
       match l with
       | nil ⇒ get v i < get v j
       | cons (i,j) l' ⇒ get v i < get v j -> loop l'
       end) l
  end.
```

5.5.2 Implementation of the heuristic function

As the heuristic function returns either a proof or nothing, we need a type `[A]` which intuitively means "either a proof of `A` or nothing" and a function `partialOut` to extract the proof. The function `partialOut` cannot have type ∀ `A : Prop, [A] -> A`, because a proof of `A` may not exist, so `partialOut` returns a proof of `True` if we don't have any proof of `A`.

```
Inductive partial {A : Prop} : Set :=
| Yes : A -> partial
```

```
| No : partial.
Local Notation "[ A ]" := (@partial A).

Definition partialOut {A : Prop} (p : [A]) : match p with | Yes _ ⇒ A | No ⇒ True end :=
  match p with
  | Yes p ⇒ p
  | No ⇒ I
  end.

Lemma denote_hyp : forall n (v : vector _ n) l a b,
    get v a < get v b -> denote_formula v (l,(a,b)).
```

The following helper lemma allows us to partially denote a formula, which is very useful before using an induction hypothesis:

```
Lemma denote_cons : forall n (v: vector _ n) i j l a b,
    denote_formula v (cons (i,j) l, (a,b)) =
    (get v i < get v j -> denote_formula v (l,(a,b))).
```

Before giving the implementation of `preorder_heuristic`, we present its behaviour: if we want to prove that `a < b` under the list of hypotheses `l`, we examine three cases:

1. there is no hypothesis: `a < b` if, and only if, `a` and `b` are the same indices, and we conclude by reflexivity of the preorder;

2. the first hypothesis is useless: we can try to conclude by induction on the list of hypotheses, by simply discarding the first hypothesis;

3. the first two cases do not apply: the first hypothesis is `i < j`, so we can try to prove by induction on the list of hypotheses that `a < i` and `j < b`, then we conclude by transitivity. The transitivity property is given by the following lemma:

   ```
   Lemma denote_trans : forall {n} {v : vector _ n} {l x} y {z},
       denote_formula v (l,(x,y)) -> denote_formula v (l,(y,z)) ->
       denote_formula v (l,(x,z)).
   ```

   We also need to prove that `i < j` by hypothesis:

   ```
   Lemma denote_hyp : forall n (v : vector _ n) l a b, get v a < get v b ->
                                                       denote_formula v (l,(a,b)).
   ```

The function `preorder_heuristic` computes a result (`Yes` or `No`), which contains a proof. As usual, the computational part is given by the `refine` tactic:

```
Definition preorder_heuristic : forall (n : nat) (v : vector A n) (f : formula n),
    [ denote_formula v f ].
  intros ? ? [l (a,b)]; generalize a b; clear a b.
  (* Induction on the number of hypotheses *)
  induction l as [|(i,j) l is_less]; intros a b.
  - (* 1. Case with no hypothesis: a < b if a == b (same index) *)
    refine (if (index_eq_dec a b)
            then Yes _
            else No); simpl.
    erewrite get_irrelevant; [reflexivity | assumption].
  - (* 2. Inductive case: either induction or
```

```
      transitivity through the new hypothesis *)
    rewrite denote_cons.
    refine (match (is_less a b) with
            | Yes _ ⇒ Yes _
            (* 3. Transitive case *)
            | No ⇒ match (is_less a i) with
                    | Yes _ ⇒ match (is_less j b) with
                              | Yes _ ⇒ Yes _
                              | No ⇒ No
                              end
                    | No ⇒ No
                    end
            end); trivial; intros.
    apply (denote_trans i); trivial.
    apply (denote_trans j); trivial.
    apply denote_hyp; trivial.
Defined.
```

5.5.3 Reification

Now that we know how to solve reified goals, we only need to write tactics that automatically reify them. We need tactics to create the vector. We want to avoid duplicates, so we have a tactic that check if a term `x` is already in the vector `v`:

```
Ltac is_in_vector x v :=
  lazymatch v with
  | vnil _ ⇒ false
  | vcons x _ ⇒ true
  | vcons _ ?v' ⇒ is_in_vector x v'
  end.
```

We can see that Ltac's pattern-matching acts in a different way than Gallina's pattern-matching. There are three kinds of terms in Coq: Gallina's pattern-matching only matches linear patterns on inductive structures, and it silently performs computations on terms. Ltac's pattern-matching matches arbitrary Gallina terms. Moreover, the pattern-matching is first-order[3], which means that it does not apply any conversion rule: for instance, one can match $1 + _$ with $1 + 1$, but one cannot match $1 + 1$ with 2 or 2 with $1 + 1$. There is also a difference between variables and meta-variables. For instance, in the code of `is_in_vector`:

- in the pattern `vcons x _`, the variable `x` is the substitution of the first argument of the tactic, it only matches something which is syntactically the same term;
- in the pattern `vcons _ ?v'`, the meta-variable `?v'` can match anything, and then Ltac instantiate a new variable `v'` on the right-hand-side of the match clause.

In short, the tactic `is_in_vector x v` checks if the term `x` is *syntactically* a subterm of `v`. It does not care whether there is a term which is computationally convertible to `x`. We now can implement a function that add a term in a vector without doing any duplication:

[3]Actually, in Coq, there is a special notation for second-order pattern-matching, but it is not useful here.

```
Ltac add_in_vector x v :=
  lazymatch is_in_vector x v with
  | true ⇒ v
  | false ⇒ constr:(vcons x v)
  end.
```

In the code above, the badly documented "`constr:`" keyword is needed because Ltac is implemented in such a way that a tactic `vcons` can coexist with a Gallina term `vcons` (these two terms would be stored in two different namespaces in the Coq internals[4]). When we apply `vcons`, we have to explicitly state it lies in the `constr` namespace.
For getting the position of a term in a vector, we also need a tactic:

```
Ltac lookup_vector x v :=
  lazymatch v with
  | vcons x _ ⇒ O
  | vcons _ ?v' ⇒ let n := lookup_vector x v' in
                  constr:(S n)
  end.
```

However, this tactic does return a natural number, we need a proof that this natural number is not out-of-bound. We use the function `lt_dec` to do so, and we have to explicitly tell Ltac to evaluate the function, using the `eval` tactic:

```
Ltac nat_to_index i n :=
  (* lt_dec: forall n m : nat, {n < m} + {~ n < m} *)
  lazymatch eval lazy in (lt_dec i n) with
  | left _ ?p ⇒ constr:(exist (fun x ⇒ x < n) i p)
  end.
```

We can create the vector and the formula corresponding to the reification of the goal:

```
Ltac create_vector R e :=
  lazymatch e with
  | R ?x ?y -> ?e' ⇒ let v' := create_vector R e' in
                     add_in_vector x ltac:(add_in_vector y v')
  | R ?x ?y ⇒ add_in_vector x constr:(vcons y (vnil _))
  end.
Ltac create_formula R e v :=
  let n := lazymatch type of v with
           | vector _ ?n ⇒ n
           end in
  lazymatch e with
  | R ?x ?y -> ?e' ⇒ let x := nat_to_index ltac:(lookup_vector x v) n in
                     let y := nat_to_index ltac:(lookup_vector y v) n in
                     let f := create_formula R e' v in
                     match f with
                     | pair ?l ?h ⇒ constr:((cons (x,y) l, h))
                     end
  | R ?x ?y ⇒ let x := nat_to_index ltac:(lookup_vector x v) n in
              let y := nat_to_index ltac:(lookup_vector y v) n in
              constr:((@nil (index n * index n),(x,y)))
  end.
```

[4]We can see an analogy with *lisp-2* dialects such as Emacs Lisp which allows you to have the same name for both a function and a variable.

The tactic `quote_formula` reify the goal, and does the appropriate change:

```
Ltac quote_formula R :=
  repeat lazymatch goal with
         | H : R _ _ |- _ => revert H
         end;
  lazymatch goal with
  | |- ?e => let v := create_vector R e in
             let f := create_formula R e v in
             change (denote_formula R v f)
  end.
```

Chapter 6

Dependent types

This chapter provides a unifying framework for two hitherto unreconciled understandings of types: *i.e.* types-as-predicates *à la* Curry and types-as-propositions *à la* Church. The key to our unification consists in introducing *strong proof-functional connectives* [80, 7, 8] in a dependent type theory such as the Edinburgh Logical Framework (LF) [52]. Both Logical Frameworks and Proof-Functional Logic consider proofs as first-class citizens, albeit differently.

Strong proof-functional connectives take seriously into account the shape of logical proofs, thus allowing for polymorphic features of proofs to be made explicit in formulæ. Hence they provide a finer semantics than classical/intuitionistic connectives, where the meaning of a compound formula depends only on the *truth value* or the *provability* of its subformulæ. However, existing approaches to strong proof-functional connectives are all quite idiosyncratic in mentioning proofs. Existing Logical Frameworks, on the other hand, provide a uniform approach to proof terms in object logics, but they do not fully capitalize on subtyping.

This situation calls for a natural combination of the two understandings of types, which should benefit both worlds. On the side of Logical Frameworks, the expressive power of the metalanguage would be enhanced thus allowing for shallower encodings of logics, a more principled use of subtypes [74], and new possibilities for formal reasoning in existing interactive theorem provers. On the side of type disciplines for programming languages, a principled framework for proofs would be provided, thus supporting a uniform approach to "proof reuse" practices based on type theory [38, 77, 23, 47, 17].

Therefore, in this chapter, we extend LF with the connectives of *strong intersection* (corresponding to the intersection type), *strong union* (corresponding to the union type), and *minimal relevant implication* of Proof-Functional Logic [80, 8]. We call this extension the Δ-framework (LF_Δ), since it builds on the Δ-calculus [66]. Moreover, we illustrate by way of examples, that LF_Δ subsumes many expressive type disciplines in the literature [74, 7, 8, 77, 23].

It is not immediate to extend the Curry-Howard isomorphism to logics supporting strong proof-functional connectives, since these connectives need to compare the shapes of derivations and do not just take into account the provability of propositions, *i.e.* the inhabitation of the corresponding type. In order to capture successfully strong logical connectives such as $\cap$ or $\cup$, we need to be able to express the rules:

$$\frac{\mathcal{D}_1 : A \quad \mathcal{D}_2 : B \quad \mathcal{D}_1 \equiv \mathcal{D}_2}{A \cap B}\ (\cap I) \qquad \frac{\mathcal{D}_1 : A \supset C \quad \mathcal{D}_2 : B \supset C \quad A \cup B \quad \mathcal{D}_1 \equiv \mathcal{D}_2}{C}\ (\cup E)$$

where $\equiv$ is a suitable equivalence between logical proofs. Notice that the above rules suggest immediately intriguing applications in polymorphic constructions, *i.e.* the same evidence can be used as a proof for different statements.

Pottinger [80] was the first to study the strong connective $\cap$. He contrasted it to the intuitionistic connective $\wedge$ as follows: *"The intuitive meaning of $\cap$ can be explained by saying that to assert $A \cap B$ is to assert that one has a reason for asserting A which is also a reason for asserting B [while] to assert $A \wedge B$ is to assert that one has a pair of reasons, the first of which is a reason for asserting A and the second of which is a reason for asserting B".*

A logical theorem involving intuitionistic conjunction which does not hold for strong conjunction is $(A \supset A) \wedge (A \supset B \supset A)$, otherwise there should exist a closed λ-term having simultaneously both one and two abstractions. López-Escobar [67] and Mints [71] investigated extensively logics featuring both strong and intuitionistic connectives especially in the context of *realizability* interpretations.

Dually, it is in the $\cup$-elimination rule that proof equality needs to be checked. Following Pottinger, we could say that *asserting $(A \cup B) \supset C$ is to assert that one has a reason for $(A \cup B) \supset C$, which is also a reason to assert $A \supset C$ and $B \supset C$.* The two connectives differ since the intuitionistic theorem $((A \supset B) \vee B) \supset A \supset B$ is not derivable for $\cup$, otherwise there would exist a term which behaves both as $\mathbf{I}$ and as $\mathbf{K}$.

Following Barbanera and Martini [8], *minimal relevant implication*, denoted by $\supset_{\mathsf{r}}$, can be viewed as a special case of implication whose related function space is the simplest possible one, namely the one containing only the identity function. The operators $\supset$ and $\supset_{\mathsf{r}}$ differ, since $A \supset_{\mathsf{r}} B \supset_{\mathsf{r}} A$ is not derivable. Relevant implication allows for a natural introduction of subtyping, in that $A \supset_{\mathsf{r}} B$ morally means $A \leqslant B$. Relevant implication amounts to a notion of "proof-reuse". Combining the remarks in [8, 7], minimal relevant implication, strong intersection and strong union correspond respectively to the implication, conjunction and disjunction operators of Meyer and Routley's Minimal Relevant Logic B^+ [70]. A terminological comment is in order. We refer to $(\supset_{\mathsf{r}})$ as *relevant implication* in order to be faithful to the original logical literature, since this constructor satisfies the logical properties of implication in the minimal relevant logical system introduced in [70]. And precisely in this sense it was used later in [8]. This use of the word "relevant" is therefore more *constrained* than, but not totally unrelated to, the one arising in the context of $\lambda\mathsf{I}-$*calculus* and linear logic, where it expresses the requirement that the variable "is used at least once" in the function, in contrast to affine "at most one use" and linear "exactly one use".

Strong connectives arise naturally in investigating the propositions-as-types analogy for intersection and union type assignment systems. From a logical point of view, there are many proposals to find a suitable logic to fit intersection: among them we cite [71, 74, 94, 84, 72, 22, 18, 79].

The LF_Δ logical framework introduced in this chapter extends [66] with union types, dependent types and minimal relevant implication. The novelty of LF_Δ in the context of Logical Frameworks, lies in the full-fledged use of strong proof-functional connectives, which to our knowledge has never been explored before. Clearly, all Δ-terms have a computational counterpart.

Pfenning's work on Refinement Types [74] pioneered an extension of the Edinburgh Logical Framework with subtyping and intersection types. His approach capitalizes on an interesting and essentially *ad hoc* notion of subtyping. However, subtyping in LF_Δ arises naturally as a derived notion from the more fundamental concept of minimal relevant

$$\frac{\Gamma \vdash M : \sigma \quad \Gamma \vdash M : \tau}{\Gamma \vdash M : \sigma \cap \tau}\ (\cap I) \qquad \frac{\Gamma \vdash M : \sigma \cap \tau}{\Gamma \vdash M : \sigma}\ (\cap E_i) \qquad \frac{\Gamma \vdash M : \sigma \cap \tau}{\Gamma \vdash M : \tau}\ (\cap E_r)$$

$$\frac{\Gamma \vdash M : \sigma}{\Gamma \vdash M : \sigma \cup \tau}\ (\cup I_l) \qquad \frac{\Gamma \vdash M : \tau}{\Gamma \vdash M : \sigma \cup \tau}\ (\cup I_r)$$

$$\frac{\Gamma, x{:}\sigma \vdash M : \rho \quad \Gamma, x{:}\tau \vdash M : \rho \quad \Gamma \vdash N : \sigma \cup \tau}{\Gamma \vdash M[N/x] : \rho}\ (\cup E) \qquad \frac{\Gamma \vdash M : \sigma \quad \sigma \leqslant \tau}{\Gamma \vdash M : \tau}\ (Sub)$$

$$\frac{x{:}\sigma \in \Gamma}{\Gamma \vdash x : \sigma}\ (Var) \qquad \frac{\Gamma \vdash M : \sigma \to \tau \quad \Gamma \vdash N : \sigma}{\Gamma \vdash M\,N : \tau}\ (App) \qquad \frac{\Gamma, x{:}\sigma \vdash M : \tau}{\Gamma \vdash \lambda x.M : \sigma \to \tau}\ (Abs)$$

(1) $\sigma \leqslant \sigma \cap \sigma$

(2) $\sigma \cup \sigma \leqslant \sigma$

(3) $\sigma \cap \tau \leqslant \sigma, \sigma \cap \tau \leqslant \tau$

(4) $\sigma \leqslant \sigma \cup \tau, \tau \leqslant \sigma \cup \tau$

(5) $\sigma \leqslant \mathtt{U}$

(6) $\sigma \leqslant \sigma$

(7) $\sigma_1 \leqslant \sigma_2, \tau_1 \leqslant \tau_2 \Rightarrow \sigma_1 \cap \tau_1 \leqslant \sigma_2 \cap \tau_2$

(8) $\sigma_1 \leqslant \sigma_2, \tau_1 \leqslant \tau_2 \Rightarrow \sigma_1 \cup \tau_1 \leqslant \sigma_2 \cup \tau_2$

(9) $\sigma \leqslant \tau, \tau \leqslant \rho \Rightarrow \sigma \leqslant \rho$

(10) $\sigma \cap (\tau \cup \rho) \leqslant (\sigma \cap \tau) \cup (\sigma \cap \rho)$

(11) $(\sigma \to \tau) \cap (\sigma \to \rho) \leqslant \sigma \to (\tau \cap \rho)$

(12) $(\sigma \to \rho) \cap (\tau \to \rho) \leqslant (\sigma \cup \tau) \to \rho$

(13) $\mathtt{U} \leqslant \mathtt{U} \to \mathtt{U}$

(14) $\sigma_2 \leqslant \sigma_1, \tau_1 \leqslant \tau_2 \Rightarrow \sigma_1 \to \tau_1 \leqslant \sigma_2 \to \tau_2$

Figure 6.1: The type assignment system λ^{BDdL} of [7] and the type theory Ξ

implication, as illustrated in Section 6.1.

Miquel [72] discusses an extension of the Calculus of Constructions with implicit typing, which subsumes a kind of proof-functional intersection. His approach has opposite motivations to ours. While LF_Δ provides a Church-style version of Curry-style type assignment systems, Miquel's Implicit Calculus of Constructions encompasses some features of Curry-style systems in an otherwise Church-style Calculus of Constructions. In LF_Δ we can discuss also *ad hoc* polymorphism, while in the Implicit Calculus only structural polymorphism is encoded. Indeed, he cannot assign the type $((\sigma \cap \tau) \to \sigma) \cap (\rho \to \rho)$ to the identity $\lambda x.x$ [62]. Kopylov [61] adds a dependent intersection type constructor $x{:}A \cap B[x]$ to NuPRL, allowing the resulting system to support dependent records (which are a very useful data structure to encode mathematics). The implicit product-type of Miquel, together with the dependent intersection type of Kopylov, and a suitable equality-type is used by Stump [89] to enrich the impredicative second-order system $\lambda P2$, in order to derive induction.

In order to achieve our goals, we could have carried out simply the encoding of LF_Δ in LF. But, due to the side-conditions characterizing proof-functional connectives, this would have be achieved only through a deep encoding. As an example of this, in Section 8.2, we give an encoding of a subsystem of [7], where subtyping has been simulated using relevant arrows. This encoding illustrates the expressive power of LF in treating proofs as first-class citizens, and it was also a source of inspiration for LF_Δ.

We will discuss examples showing some encoding in the LF_Δ in Chapter 8: all examples have been checked by an experimental proof development environment for LF_Δ [88] (see

Kinds			
K	$::=$	$\mathsf{Type} \mid \Pi x{:}\sigma.K$	as in LF
Families			
σ, τ	$::=$	$a \mid \Pi x{:}\sigma.\tau \mid \sigma\,\Delta \mid$	as in LF
		$\sigma \to^{\mathsf{r}} \tau \mid$	relevant family
		$\sigma \cap \tau \mid$	intersection family
		$\sigma \cup \tau$	union family
Objects			
Δ	$::=$	$c \mid x \mid \lambda x{:}\sigma.\Delta \mid \Delta\,\Delta \mid$	as in LF
		$\lambda^{\mathsf{r}} x{:}\sigma.\Delta \mid$	relevant abstraction
		$\Delta \cdot^{\mathsf{r}} \Delta \mid$	relevant application
		$\langle \Delta, \Delta \rangle \mid$	intersection objects
		$[\Delta, \Delta] \mid$	union objects
		$\mathsf{pr}_1\,\Delta \mid \mathsf{pr}_2\,\Delta \mid$	projections objects
		$\mathsf{in}_1^{\sigma}\,\Delta \mid \mathsf{in}_2^{\sigma}\,\Delta$	injections objects

Figure 6.2: The syntax of the Δ-framework

Chapter 7 and Bull and Bull-Subtyping in [87]).

This chapter is organized as follows: in Section 6.1, we introduce LF_Δ. In Section 6.2 we outline its metatheory, together with a discussion of the main design decisions. In Section 6.3, we discuss the relation between subtyping and the relevant arrow operator. In Section 6.4, we give a Pure Type System presentation of LF_Δ. In Section 6.5, we outline the future work.

6.1 The Δ-framework: LF with proof-functional operators

The syntax of LF_Δ pseudo-terms is given in Figure 6.2. For the sake of simplicity, we suppose that α-convertible terms are equal. Signatures and contexts are defined as finite sequence of declarations, like in LF. Observe that we could formulate LF_Δ in the style of [53], using only canonical forms and without reductions, but we prefer to use the standard LF format to support better intuition. There are three proof-functional objects, namely strong conjunction (typed with $\sigma \cap \tau$) with two corresponding projections, strong disjunction (typed with $\sigma \cup \tau$) with two corresponding injections, and strong (or relevant) λ-abstraction (typed with $\to^{\mathsf{r}}$). Indeed, a relevant implication is not a dependent one because the essence of the inhabitants of type $\sigma \to^{\mathsf{r}} \tau$ is essentially the identity function as enforced in the typing rules. Note that injections in_i need to be decorated with the

$$
\begin{array}{rcl}
\wr \langle \Delta_1, \Delta_2 \rangle \wr & \stackrel{def}{=} & \wr \Delta_1 \wr \\
\wr \lambda^{\mathsf{r}} x{:}\sigma.\Delta \wr & \stackrel{def}{=} & \lambda x.\wr \Delta \wr \\
\wr \lambda x{:}\sigma.\Delta \wr & \stackrel{def}{=} & \lambda x.\wr \Delta \wr
\end{array}
\qquad
\begin{array}{rcl}
\wr [\Delta_1, \Delta_2] \wr & \stackrel{def}{=} & \wr \Delta_1 \wr \\
\wr \Delta_1\, \Delta_2 \wr & \stackrel{def}{=} & \wr \Delta_1 \wr \wr \Delta_2 \wr \\
\wr \Delta_1 \cdot^{\mathsf{r}} \Delta_2 \wr & \stackrel{def}{=} & \wr \Delta_2 \wr
\end{array}
\qquad
\begin{array}{rcl}
\wr \mathsf{pr}_i\, \Delta \wr & \stackrel{def}{=} & \wr \Delta \wr \\
\wr \mathsf{in}_i^{\sigma}\, \Delta \wr & \stackrel{def}{=} & \wr \Delta \wr \\
\wr c \wr & \stackrel{def}{=} & c \\
\wr x \wr & \stackrel{def}{=} & x
\end{array}
$$

Figure 6.3: The extended essence function

$$
\begin{array}{rcl}
(\lambda x{:}\sigma.\Delta_1)\, \Delta_2 & \longrightarrow_{\beta} & \Delta_1[\Delta_2/x] \\
\mathsf{pr}_1\, \langle \Delta_1, \Delta_2 \rangle & \longrightarrow_{\mathsf{pr}_1} & \Delta_1 \\
\mathsf{pr}_2\, \langle \Delta_1, \Delta_2 \rangle & \longrightarrow_{\mathsf{pr}_2} & \Delta_2 \\
[\Delta_1, \Delta_2]\, \mathsf{in}_1^{\sigma}\, \Delta_3 & \longrightarrow_{\mathsf{in}_1} & \Delta_1\, \Delta_3 \\
[\Delta_1, \Delta_2]\, \mathsf{in}_2^{\sigma}\, \Delta_3 & \longrightarrow_{\mathsf{in}_2} & \Delta_2\, \Delta_3 \\
(\lambda^{\mathsf{r}} x{:}\sigma.\Delta_1) \cdot^{\mathsf{r}} \Delta_2 & \longrightarrow_{\beta r} & \Delta_1[\Delta_2/x]
\end{array}
$$

$$
\frac{\Delta_1 \rightarrow_{\Delta} \Delta_1' \quad \Delta_2 \rightarrow_{\Delta} \Delta_2' \quad \wr \Delta_1' \wr =_{\eta} \wr \Delta_2' \wr}{\langle \Delta_1, \Delta_2 \rangle \rightarrow_{\Delta} \langle \Delta_1', \Delta_2' \rangle}\ (Congr_{\cap})
$$

$$
\frac{\Delta_1 \rightarrow_{\Delta} \Delta_1' \quad \Delta_2 \rightarrow_{\Delta} \Delta_2' \quad \wr \Delta_1' \wr =_{\eta} \wr \Delta_2' \wr}{[\Delta_1, \Delta_2] \rightarrow_{\Delta} [\Delta_1', \Delta_2']}\ (Congr_{\cup})
$$

Figure 6.4: The reduction semantics

injected type σ in order to ensure the unicity of typing.

We extend the notion of *essence* of Definition 2.3 to syntactically connect pure λ-terms (denoted by M) and type annotated LF_{Δ} terms (denoted by Δ). The essence function compositionally erases all type annotations, see Figure 6.3.

One could argue that the choice of Δ_1 in the definition of strong pairs/sums is arbitrary and could have been replaced with Δ_2: however, the typing rules will ensure that, if $\langle \Delta_1, \Delta_2 \rangle$ (resp. $[\Delta_1, \Delta_2]$) is typable, then we have that $\wr \Delta_1 \wr =_{\eta} \wr \Delta_2 \wr$. Thus, strong pairs/sums are constrained. The rule for the essence of a relevant application is justified by the fact that the operator amounts to just a type decoration.

The six basic reductions for LF_{Δ} objects appear in Figure 6.4. Congruence rules are as usual, except for the two cases dealing with strong pairs and sums. Here redexes need to be reduced "in parallel" in order to preserve identity of essences in the components. We denote by $=_{\Delta}$ the symmetric, reflexive, and transitive closure of $\rightarrow_{\Delta}$, *i.e.* the compatible closure of the reduction induced by the first six rules, with the addition of the last two congruence rules in the same figure. In order to make this definition truly functional as well as to be able to prove a simple subject reduction result, we need to constrain strong pairs and sums, *i.e.* objects of the form $\langle \Delta_i, \Delta_j \rangle$ and $[\Delta_i, \Delta_j]$ to have congruent components up-to erasure of type annotations. This is achieved by imposing $\wr \Delta_i \wr =_{\eta} \wr \Delta_j \wr$ in both constructs. We do not consider the relations $\wr \Delta_i \wr =_{\beta} \wr \Delta_j \wr$ or $\wr \Delta_i \wr =_{\beta\eta} \wr \Delta_j \wr$

Valid Signatures

$$\frac{}{\cdot \text{ sig}}\ (\epsilon\Sigma) \qquad \frac{\Sigma \text{ sig} \quad \vdash_\Sigma K \quad a \notin \mathsf{Dom}(\Sigma)}{\Sigma, a{:}K \text{ sig}}\ (K\Sigma)$$

$$\frac{\Sigma \text{ sig} \quad \vdash_\Sigma \sigma : \mathsf{Type} \quad c \notin \mathsf{Dom}(\Sigma)}{\Sigma, c{:}\sigma \text{ sig}}\ (\sigma\Sigma)$$

Valid Contexts

$$\frac{\Sigma \text{ sig}}{\vdash_\Sigma \cdot}\ (\epsilon\Gamma) \qquad \frac{\vdash_\Sigma \Gamma \quad \Gamma \vdash_\Sigma \sigma : \mathsf{Type} \quad x \notin \mathsf{Dom}(\Gamma)}{\vdash_\Sigma \Gamma, x{:}\sigma}\ (\sigma\Gamma)$$

Figure 6.5: Valid signatures and contexts

because they are undecidable. We will therefore assume that such strong pairs and sums are simply not well defined terms, if the components have a different "infrastructure". The effects of this choice are reflected in the congruence rules in the reduction relation, in order to ensure that reductions can only be carried out in parallel along the two components.

The restriction on reductions in strong pairs/sums and the new constructs do not cause any problems in showing that $\rightarrow_\Delta$ is confluent:

Theorem 6.1 (Confluence)**.** LF_Δ is confluent, *i.e.*:

1. If $K_1 \longrightarrow^*_\Delta K_2$ and $K_1 \longrightarrow^*_\Delta K_3$, then $\exists K_4$ such that $K_2 \longrightarrow^*_\Delta K_4$ and $K_3 \longrightarrow^*_\Delta K_4$;
2. If $\sigma_1 \longrightarrow^*_\Delta \sigma_2$ and $\sigma_1 \longrightarrow^*_\Delta \sigma_3$, then $\exists \sigma_4$ such that $\sigma_2 \longrightarrow^*_\Delta \sigma_4$ and $\sigma_3 \longrightarrow^*_\Delta \sigma_4$;
3. If $\Delta_1 \longrightarrow^*_\Delta \Delta_2$ and $\Delta_1 \longrightarrow^*_\Delta \Delta_3$, then $\exists \Delta_4$ such that $\Delta_2 \longrightarrow^*_\Delta \Delta_4$ and $\Delta_3 \longrightarrow^*_\Delta \Delta_4$.

Proof. Using the same technique as in Theorem 2.19. □

The extended type theory LF_Δ is a formal system for deriving judgments of the forms:

$\vdash \Sigma$	Σ is a valid signature	$\Gamma \vdash_\Sigma \sigma : K$	σ has kind K in Γ and Σ
$\vdash_\Sigma \Gamma$	Γ is a valid context in Σ	$\Gamma \vdash_\Sigma \Delta : \sigma$	Δ has type σ in Γ and Σ
$\Gamma \vdash_\Sigma K$	K is a kind in Γ and Σ		

Let Figure 6.5 denote Valid Signatures and Contexts and Figure 6.6 denote Valid Kinds and Families.The set of rules for object formation is defined in Figure 6.7, while the sets of rules for signatures, contexts, kinds and families are defined as in [52], and all typing rules (except the $(Conv)$ rules) are syntax-directed. Note that proof-functionality is enforced by the essence side-conditions in rules $(\rightarrow^{\mathsf{r}} I)$, $(\cap I)$, and $(\cup E)$.

In the rule $(Conv)$ we rely on the external notion of equality $=_\Delta$. An option could have be to add an internal notion of equality directly in the type system ($\Gamma \vdash_\Sigma \sigma =_\Delta \tau$), and prove that the external and the internal definitions of equality are equivalent, as was proved for semi-full Pure Type Systems [86]. Yet another possibility could be to compare type essences $\wr\sigma\wr =_\Delta \wr\tau\wr$, for a suitable extension of essence to types and kinds. Unfortunately, this would lead to undecidability of type checking, in connection with relevant implication, as the following example shows. Consider two constants c_1 of type $\sigma \rightarrow^{\mathsf{r}} (\Pi y{:}\sigma.\sigma)$ and c_2 of type $(\Pi y{:}\sigma.\sigma) \rightarrow^{\mathsf{r}} \sigma$: the Δ-term in Figure 6.8 is typable

Valid Kinds

$$\frac{\vdash_\Sigma \Gamma}{\Gamma \vdash_\Sigma \mathsf{Type}}\ (Type) \qquad \frac{\Gamma, x{:}\sigma \vdash_\Sigma K}{\Gamma \vdash_\Sigma \Pi x{:}\sigma.K}\ (\Pi K)$$

Valid Families

$$\frac{\vdash_\Sigma \Gamma \quad a{:}K \in \Sigma}{\Gamma \vdash_\Sigma a : K}\ (Const) \qquad \frac{\Gamma \vdash_\Sigma \sigma : K_1 \quad \Gamma \vdash_\Sigma K_2 \quad K_1 =_\Delta K_2}{\Gamma \vdash_\Sigma \sigma : K_2}\ (Conv)$$

$$\frac{\Gamma, x{:}\sigma \vdash_\Sigma \tau : \mathsf{Type}}{\Gamma \vdash_\Sigma \Pi x{:}\sigma.\tau : \mathsf{Type}}\ (\Pi I) \qquad \frac{\Gamma \vdash_\Sigma \sigma : \Pi x{:}\tau.K \quad \Gamma \vdash_\Sigma \Delta : \tau}{\Gamma \vdash_\Sigma \sigma\,\Delta : K[\Delta/x]}\ (\Pi E)$$

$$\frac{\Gamma \vdash_\Sigma \sigma : \mathsf{Type} \quad \Gamma \vdash_\Sigma \tau : \mathsf{Type}}{\Gamma \vdash_\Sigma \sigma \to^{\mathsf{r}} \tau : \mathsf{Type}}\ (\to^{\mathsf{r}} I)$$

$$\frac{\Gamma \vdash_\Sigma \sigma : \mathsf{Type} \quad \Gamma \vdash_\Sigma \tau : \mathsf{Type}}{\Gamma \vdash_\Sigma \sigma \cap \tau : \mathsf{Type}}\ (\cap I) \qquad \frac{\Gamma \vdash_\Sigma \sigma : \mathsf{Type} \quad \Gamma \vdash_\Sigma \tau : \mathsf{Type}}{\Gamma \vdash_\Sigma \sigma \cup \tau : \mathsf{Type}}\ (\cup I)$$

Figure 6.6: Valid kinds and families

Valid Objects

$$\frac{\vdash_\Sigma \Gamma \quad c{:}\sigma \in \Sigma}{\Gamma \vdash_\Sigma c : \sigma}\ (Const) \qquad \frac{\vdash_\Sigma \Gamma \quad x{:}\sigma \in \Gamma}{\Gamma \vdash_\Sigma x : \sigma}\ (Var)$$

$$\frac{\Gamma, x{:}\sigma \vdash_\Sigma \Delta : \tau}{\Gamma \vdash_\Sigma \lambda x{:}\sigma.\Delta : \Pi x{:}\sigma.\tau}\ (\Pi I) \qquad \frac{\Gamma \vdash_\Sigma \Delta_1 : \Pi x{:}\sigma.\tau \quad \Gamma \vdash_\Sigma \Delta_2 : \sigma}{\Gamma \vdash_\Sigma \Delta_1\,\Delta_2 : \tau[\Delta_2/x]}\ (\Pi E)$$

$$\frac{\Gamma, x{:}\sigma \vdash_\Sigma \Delta : \tau \quad \wr\Delta\wr =_\eta x}{\Gamma \vdash_\Sigma \lambda^{\mathsf{r}} x{:}\sigma.\Delta : \sigma \to^{\mathsf{r}} \tau}\ (\to^{\mathsf{r}} I) \qquad \frac{\Gamma \vdash_\Sigma \Delta : \sigma \cap \tau}{\Gamma \vdash_\Sigma \mathsf{pr}_1\,\Delta : \sigma}\ (\cap E_l)$$

$$\frac{\Gamma \vdash_\Sigma \Delta_1 : \sigma \to^{\mathsf{r}} \tau \quad \Gamma \vdash_\Sigma \Delta_2 : \sigma}{\Gamma \vdash_\Sigma \Delta_1 \cdot^{\mathsf{r}} \Delta_2 : \tau}\ (\to^{\mathsf{r}} E) \qquad \frac{\Gamma \vdash_\Sigma \Delta : \sigma \cap \tau}{\Gamma \vdash_\Sigma \mathsf{pr}_2\,\Delta : \tau}\ (\cap E_r)$$

$$\frac{\Gamma \vdash_\Sigma \Delta_1 : \sigma \quad \Gamma \vdash_\Sigma \Delta_2 : \tau \quad \wr\Delta_1\wr =_\eta \wr\Delta_2\wr}{\Gamma \vdash_\Sigma \langle \Delta_1, \Delta_2 \rangle : \sigma \cap \tau}\ (\cap I) \qquad \frac{\begin{array}{c}\Gamma \vdash_\Sigma \Delta : \sigma \\ \Gamma \vdash_\Sigma \tau : \mathsf{Type} \quad \sigma =_\Delta \tau\end{array}}{\Gamma \vdash_\Sigma \Delta : \tau}\ (Conv)$$

$$\frac{\Gamma \vdash_\Sigma \Delta : \sigma \quad \Gamma \vdash_\Sigma \sigma \cup \tau : \mathsf{Type}}{\Gamma \vdash_\Sigma \mathsf{in}_1^\tau\,\Delta : \sigma \cup \tau}\ (\cup I_l) \qquad \frac{\Gamma \vdash_\Sigma \Delta : \tau \quad \Gamma \vdash_\Sigma \sigma \cup \tau : \mathsf{Type}}{\Gamma \vdash_\Sigma \mathsf{in}_2^\sigma\,\Delta : \sigma \cup \tau}\ (\cup I_r)$$

$$\frac{\begin{array}{ll}\Gamma \vdash_\Sigma \Delta_1 : \Pi y{:}\sigma.\rho[\mathsf{in}_1^\tau\, y/x] & \wr\Delta_1\wr =_\eta \wr\Delta_2\wr \\ \Gamma \vdash_\Sigma \Delta_2 : \Pi y{:}\tau.\rho[\mathsf{in}_2^\sigma\, y/x] & \Gamma, x{:}\sigma \cup \tau \vdash_\Sigma \rho : \mathsf{Type}\end{array}}{\Gamma \vdash_\Sigma [\Delta_1, \Delta_2] : \Pi x{:}\sigma \cup \tau.\rho}\ (\cup E)$$

Figure 6.7: The type rules for valid objects

with σ and its essence is $\Omega \stackrel{def}{=} (\lambda x.x\,x)\,(\lambda x.x\,x)$. Since the intended meaning of relevant implication is "essentially" the identity, introducing variables or constants whose type is a relevant implication, amounts to assuming axioms corresponding to type inclusions such

$$\Delta_\Omega \stackrel{def}{=} (\lambda x{:}\sigma.c_1 \cdot^{\mathsf{r}} x\, x)\,(c_2 \cdot^{\mathsf{r}} (\lambda x{:}\sigma.c_1 \cdot^{\mathsf{r}} x\, x)) \qquad \wr \Delta_\Omega \wr \equiv \Omega$$

Figure 6.8: Encoding of Ω

$$x\,((\mathsf{I}\,y)\,z)\,((\mathsf{I}\,y)\,z) \begin{array}{l} \nearrow^\beta x\,(y\,z)\,((\mathsf{I}\,y)\,z) \searrow_\beta \\ \searrow_\beta x\,((\mathsf{I}\,y)\,z)\,(y\,z) \nearrow^\beta \end{array} x\,(y\,z)\,(y\,z),$$

Figure 6.9: Pierce's one-step reduction counter-example

as those that equate σ and $\sigma \to \sigma$. As a consequence, β-equality of essences becomes undecidable. Thus, we rule out such options in relating relevant implications in LF_Δ to subtypes in the type assignment system λ^{BDdL} of [7].

6.2 Relating LF_Δ to λ^{BDdL}

We compare and contrast certain design decisions of LF_Δ to the type assignment system λ^{BDdL} of [7]. The proof of strong normalization for LF_Δ will rely, in fact, on a forgetful mapping from LF_Δ to λ^{BDdL}. As pointed out in [7], the elimination rule for union types in λ^{BDdL} breaks subject reduction for one-step β-reduction, but this can be recovered using a suitable parallel β-reduction. The well-known counter-example for one-step reduction, due to Pierce, is in Figure 6.9 (where I is the identity).

In the typing context $B \stackrel{def}{=} x{:}(\sigma_1 \to \sigma_1 \to \tau) \cap (\sigma_2 \to \sigma_2 \to \tau), y{:}\rho \to (\sigma_1 \cup \sigma_2), z{:}\rho$, the first and the last terms can be typed with τ, while the terms in the fork cannot. The reason is that the subject in the conclusion of the $(\cup E)$ rule uses a context which can have more than one hole, as in the present case. It is interesting to note that the problem would not arise if $(\cup E)$ is replaced by the following rule schema:

$$\frac{\begin{array}{l}\Gamma, x_1{:}\sigma, \ldots, x_n{:}\sigma \vdash M : \rho \\ \Gamma, x_1{:}\tau, \ldots, x_n{:}\tau \vdash M : \rho \quad \Gamma \vdash N_i : \sigma \cup \tau \quad N_i =_\beta N_j \quad i, j = 1 \ldots n\end{array}}{\Gamma \vdash M[N_1/x_1, \ldots, N_n/x_n] : \rho}\ (\cup E')$$

However, removing the non-static clause on the N_i's would yield a more permissive type system than λ^{BDdL}.

In LF_Δ, the formulation of the $(\cup E)$ rule takes a different route which does not trigger the counterexample. Indeed, we have introduction and elimination constructs $\mathsf{in_1}$, $\mathsf{in_2}$ and $[-,-]$ which allow to reduce the term only if we know that the argument, stripped of the introduction construct, has one of the types of the disjunction. Pierce's counter-example can be expressed and typed in LF_Δ with the following judgment (the full derivation is in Subsection 6.2.1):

$$\Gamma \vdash_\Sigma [\underbrace{(\lambda x_1{:}\sigma_1.(\mathsf{pr_1}\,x)\,x_1\,x_1)}_{\Delta_1}, \underbrace{(\lambda x_2{:}\sigma_2.(\mathsf{pr_2}\,x)\,x_2\,x_2)}_{\Delta_2}]\,(\underbrace{(\lambda x_3{:}\rho \to \sigma_1 \cup \sigma_2.x_3)}_{\Delta_3}\,y\,z) : \tau$$

where $\Gamma \stackrel{def}{=} x{:}(\Pi x_1{:}\sigma_1.\Pi x_2{:}\sigma_1.\tau) \cap (\Pi x_1{:}\sigma_2.\Pi x_2{:}\sigma_2.\tau)$, $y{:}\rho \to \sigma_1 \cup \sigma_2, z{:}\rho$, and $\Sigma \stackrel{def}{=} \tau{:}\mathsf{Type}$. Notice that there is only one redex, namely $\Delta_3\,y$, and the reduction of this redex leads to $[\Delta_1, \Delta_2]\,(y\,z)$, and no other intermediate (untypable) Δ-terms are possible.

We recall the following result, proved in [7], will be useful in the following section.

6.2.1 Typed derivation of Pierce's example

Here is the typed derivation of Pierce's example (from Figure 6.9):

$$\Gamma \vdash_\Sigma \lambda x_1{:}\sigma_1.(\mathsf{pr}_1\, x)\, x_1\, x_1 : \Pi x_1{:}\sigma_1.(a\, x_4\, x_4)[\mathsf{in}_1^{\sigma_2}\, x_1/x_4]$$

$$\Gamma \vdash_\Sigma \lambda x_2{:}\sigma_2.(\mathsf{pr}_2\, x)\, x_2\, x_2 : \Pi x_2{:}\sigma_2.(a\, x_4\, x_4)[\mathsf{in}_2^{\sigma_1}\, x_2/x_4]$$

$$\Gamma, x_4{:}\sigma_1 \cup \sigma_2 \vdash_\Sigma a\, x_4\, x_4 : \mathsf{Type}$$

$$\wr\, \lambda x_1{:}\sigma_1.(\mathsf{pr}_1\, x)\, x_1\, x_1 \wr =_\eta \wr\, \lambda x_2{:}\sigma_2.(\mathsf{pr}_2\, x)\, x_2\, x_2) \wr$$

$$\dfrac{\dfrac{\dfrac{}{\Gamma \vdash_\Sigma [\lambda x_1{:}\sigma_1.(\mathsf{pr}_1\, x)\, x_1\, x_1, \lambda x_2{:}\sigma_2.(\mathsf{pr}_2\, x)\, x_2\, x_2] : \Pi x_4{:}\sigma_1 \cup \sigma_2.a\, x_4\, x_4} \quad \Gamma \vdash_\Sigma (\lambda x_3{:}\rho \to \sigma_1 \cup \sigma_2.x_3)\, y\, z : \sigma_1 \cup \sigma_2}{\Gamma \vdash_\Sigma [(\lambda x_1{:}\sigma_1.(\mathsf{pr}_1\, x)\, x_1\, x_1), (\lambda x_2{:}\sigma_2.(\mathsf{pr}_2\, x)\, x_2\, x_2)]\, ((\lambda x_3{:}\rho \to \sigma_1 \cup \sigma_2.x_3)\, y\, z) : a\, ((\lambda x_3{:}\rho \to \sigma_1 \cup \sigma_2.x_3)\, y\, z)\, ((\lambda x_3{:}\rho \to \sigma_1 \cup \sigma_2.x_3)\, y\, z)}}{\Gamma \vdash_\Sigma [\lambda x_1{:}\sigma_1.(\mathsf{pr}_1\, x)\, x_1\, x_1), \lambda x_2{:}\sigma_2.(\mathsf{pr}_2\, x)\, x_2\, x_2)]\, ((\lambda x_3{:}\rho \to \sigma_1 \cup \sigma_2.x_3)\, y\, z) : a\, (y\, z)\, (y\, z)}$$

where:

$\Gamma \stackrel{def}{=} x{:}\Pi x_1{:}\sigma_1.\Pi x_2{:}\sigma_1.a\, (\mathsf{in}_1^{\sigma_2}\ x_1)\, (\mathsf{in}_1^{\sigma_2}\ x_2) \cap \Pi x_1{:}\sigma_2.\Pi x_2{:}\sigma_2.a\, (\mathsf{in}_2^{\sigma_1}\ x_1)\, (\mathsf{in}_2^{\sigma_1}\ x_2), y{:}\rho \to \sigma_1 \cup \sigma_2, z{:}\rho$

$\Sigma \stackrel{def}{=} a{:}\sigma_1 \cup \sigma_2 \to \sigma_1 \cup \sigma_2 \to \mathsf{Type}$

Theorem 6.2 (Theorem 4.8 of [7])**.** The system λ^{BDdL} without `U` gives types only to strongly normalizing terms.

6.2.2 LF_Δ metatheory

LF_Δ can play the role of a Logical Framework only if decidable. The road map which we follow to establish decidability is the standard one, see *e.g.* [52]. In particular, we prove in order: uniqueness of types and kinds, structural properties, and normalization for well-formed terms. Then we prove the inversion property, the subderivation property, subject reduction, and finally decidability. But first, we prove the fundamental lemmas:

Lemma 6.3. Let α be either $\sigma : K$ or $\Delta : \sigma$. Then:

1. Weakening: If $\Gamma \vdash_\Sigma \alpha$ and $\vdash_\Sigma \Gamma, \Gamma'$, then $\Gamma, \Gamma' \vdash_\Sigma \alpha$;
2. Strengthening: If $\Gamma, x{:}\sigma, \Gamma' \vdash_\Sigma \alpha$, then $\Gamma, \Gamma' \vdash_\Sigma \alpha$, provided that $x \notin \mathsf{Fv}(\Gamma') \cup \mathsf{Fv}(\alpha)$;
3. Substitution: If $\Gamma \vdash_\Sigma \Delta : \sigma$ and $\Gamma, x{:}\sigma, \Gamma' \vdash_\Sigma \alpha$, then $\Gamma, \Gamma'[\Delta/x] \vdash_\Sigma \alpha[\Delta/x]$;
4. Permutation: If $\Gamma, x_1{:}\sigma, \Gamma', x_2{:}\tau, \Gamma'' \vdash_\Sigma \alpha$, then $\Gamma, x_2{:}\tau, \Gamma', x_1{:}\sigma, \Gamma'' \vdash_\Sigma \alpha$, provided that x_1 does not occur free in Γ' or in τ, and that τ is valid in Γ.

Proof. All the proofs are done by induction on the structure of the derivation. □

The first important step states that if a Δ-term is typable, then its type is unique up to $=_\Delta$.

Theorem 6.4 (Unicity of types and kinds)**.**

1. If $\Gamma \vdash_\Sigma \Delta : \sigma$ and $\Gamma \vdash_\Sigma \Delta : \tau$, then $\sigma =_\Delta \tau$;
2. If $\Gamma \vdash_\Sigma \sigma : K$ and $\Gamma \vdash_\Sigma \sigma : K'$, then $K =_\Delta K'$.

Proof. All the proofs are done by induction on the structure of the derivation. □

Strong normalization is proved as in LF. First we encode terms of LF_Δ into terms of the type assignment system λ^{BDdL} such that redexes in the source language correspond to redexes in the target language and we use Theorem 6.2. Then, we introduce two forgetful mappings, namely $\| - \|$ and $| - |$, defined in Figure 6.10, to erase dependencies in types and to drop proof-functional constructors in Δ-terms and we conclude. Special care is needed in dealing with redexes occurring in type-dependencies, because these need to be flattened at the level of terms.

Definition 6.1. Let the forgetful mappings $\| - \|$ and $| - |$ be defined as in Figure 6.10.

The forgetful mappings are extended to contexts and signatures in the obvious way. The clauses for strong pairs/sums are justified by the following lemma:

Lemma 6.5. If $\Gamma \vdash_\Sigma \langle \Delta_1, \Delta_2 \rangle : \sigma$ or $\Gamma \vdash_\Sigma [\Delta_1, \Delta_2] : \sigma$, then $|\Delta_1| =_\beta |\Delta_2|$.

Proof. By induction on Δ_1. Note that β-conversion is needed in the case where $\Delta_1 \equiv \lambda x{:}\sigma.\Delta_1'$ for some Δ_1'. In that case, it is necessary that $\Delta_2 \equiv \lambda x{:}\tau.\Delta_2'$, for some Δ_2', and we have that $|\Delta_1| =_\beta \lambda x.\ |\Delta_1'|$ and $|\Delta_2| =_\beta \lambda x.\ |\Delta_2'|$, where by induction hypothesis $|\Delta_1'| =_\beta |\Delta_2'|$. □

$$
\begin{array}{rcl}
\|\mathsf{Type}\| & = & \top \quad \text{(a special constant)} \\
\|\Pi x{:}\sigma.K\| & = & \|\sigma\| \to \|K\| \\
\|a\| & = & a \\
\|\Pi x{:}\sigma.\tau\| & = & \|\sigma\| \to \|\tau\|
\end{array}
\qquad
\begin{array}{rcl}
\|\sigma \to^{\mathsf{r}} \tau\| & = & \|\sigma\| \to \|\tau\| \\
\|\sigma\,\Delta\| & = & \|\sigma\| \\
\|\sigma \cap \tau\| & = & \|\sigma\| \cap \|\tau\| \\
\|\sigma \cup \tau\| & = & \|\sigma\| \cup \|\tau\|
\end{array}
$$

$$
\begin{array}{rcl}
|a| & = & a \\
|c| & = & c \\
|x| & = & x \\
|\sigma\,\Delta| & = & |\sigma|\,|\Delta| \\
|\Delta_1\,\Delta_2| & = & |\Delta_1|\,|\Delta_2| \\
|\Delta_1 \cdot^{\mathsf{r}} \Delta_2| & = & |\Delta_1|\,|\Delta_2| \\
|\mathsf{pr}_1\,\Delta| & = & |\Delta| \\
|\mathsf{pr}_2\,\Delta| & = & |\Delta| \\
|\Pi x{:}\sigma.\tau| & = & c_{\|\sigma\|}\ |\sigma|\ (\lambda x.\ |\tau|)
\end{array}
\qquad
\begin{array}{rcl}
|\sigma \to^{\mathsf{r}} \tau| & = & c_\times\ |\sigma|\,|\tau| \\
|\sigma \cap \tau| & = & c_\times\ |\sigma|\,|\tau| \\
|\sigma \cup \tau| & = & c_\times\ |\sigma|\,|\tau| \\
|\langle \Delta_1, \Delta_2 \rangle| & = & |\Delta_1| \\
|[\Delta_1, \Delta_2]| & = & |\Delta_1| \\
|\lambda x{:}\sigma.\Delta| & = & (\lambda y.\lambda x.\ |\Delta|)\ |\sigma|\ \ y \notin \mathsf{Fv}(\Delta) \\
|\lambda^{\mathsf{r}} x{:}\sigma.\Delta| & = & (\lambda y.\lambda x.\ |\Delta|)\ |\sigma|\ \ y \notin \mathsf{Fv}(\Delta) \\
|\mathsf{in}_1^\sigma\,\Delta| & = & (\lambda x.\ |\Delta|)\ |\sigma|\ \ x \notin \mathsf{Fv}(\Delta) \\
|\mathsf{in}_2^\sigma\,\Delta| & = & (\lambda x.\ |\Delta|)\ |\sigma|\ \ x \notin \mathsf{Fv}(\Delta)
\end{array}
$$

Figure 6.10: The forgetful mappings $\|-\|$ and $|-|$

The following lemmas are proved by straightforward structural induction, and using Lemma 6.5.

Lemma 6.6.

1. If $\sigma =_\Delta \tau$, then $\|\sigma\| =_\beta \|\tau\|$;
2. If $K_1 =_\Delta K_2$, then $\|K_1\| =_\beta \|K_2\|$.

Lemma 6.7.

1. $|\Delta_1[\Delta_2/x]| =_\beta |\Delta_1|\,[|\Delta_2|\,/x]$;
2. $|\sigma[\Delta/x]| =_\beta |\sigma|\,[|\Delta|\,/x]$.

Lemma 6.8.

1. If $\Gamma \vdash_\Sigma \sigma : K$, then $\|\Gamma\| \vdash_{\lambda^{\mathrm{BDdL}+}} |\sigma| : \|K\|$;
2. If $\Gamma \vdash_\Sigma \Delta : \sigma$, then $\|\Gamma\| \vdash_{\lambda^{\mathrm{BDdL}+}} |\Delta| : \|\sigma\|$.

where $\vdash_{\lambda^{\mathrm{BDdL}+}}$ denotes the type system λ^{BDdL}, augmented by $c_\times : \top \to \top \to \top$ and the infinite set of axioms $c_{\|\sigma\|} : \top \to (\|\sigma\| \to \top) \to \top$, for each type σ.

Proof. By induction on the derivation, using Lemmas 6.6 and 6.7 □

Notice that the function $\wr - \wr$ and $|-|$ treat relevant implication differently.

Lemma 6.9.

1. If $\sigma \longrightarrow_\beta \tau$, then $|\sigma| \longrightarrow_\beta^+ |\tau|$;
2. if $\Delta_1 \longrightarrow_\beta \Delta_2$, then $|\Delta_1| \longrightarrow_\beta^+ |\Delta_2|$.

Proof. By induction on the term. □

Parallel reduction enjoys the strong normalization property, *i.e.*

Theorem 6.10 (Strong normalization)**.**

1. LF_Δ is strongly normalizing, *i.e.*,
   (a) If $\Gamma \vdash_\Sigma K$, then K is strongly normalizing;
   (b) If $\Gamma \vdash_\Sigma \sigma : K$, then σ is strongly normalizing;
   (c) If $\Gamma \vdash_\Sigma \Delta : \sigma$, then Δ is strongly normalizing;
2. Every strongly normalizing pure λ-term is the essence of some Δ-term.

Proof.

1. Strong normalization derives directly from Lemmas 6.8, 6.9 and Theorem 6.2;
2. Every strongly normalizing pure λ-term can be typed with intersection types, and the derivation tree can be encoded as a Δ-term.

□

Then, we have subject reduction, whose proof relies on technical lemmas about inversion and subderivation properties.

The following lemmas (Lemma 6.11 and Lemma 6.12) can be easily proved by structural induction.

Lemma 6.11 (Inversion properties)**.**

1. If $\Pi x{:}\sigma.\tau =_\Delta \tau''$, then $\tau'' \equiv \Pi x{:}\sigma'.\tau'$, for some σ', τ', such that $\sigma' =_\Delta \sigma$, and $\tau' =_\Delta \tau$;
2. If $\sigma \to^{\mathsf{r}} \tau =_\Delta \tau''$, then $\tau'' \equiv \sigma' \to^{\mathsf{r}} \tau'$, for some σ', τ', such that $\sigma' =_\Delta \sigma$, and $\tau' =_\Delta \tau$;
3. If $\sigma \cap \tau =_\Delta \rho$, then $\rho \equiv \sigma' \cap \tau'$, for some σ', τ', such that $\sigma' =_\Delta \sigma$, and $\tau' =_\Delta \tau$;
4. If $\sigma \cup \tau =_\Delta \rho$, then $\rho \equiv \sigma' \cup \tau'$, for some σ', τ', such that $\sigma' =_\Delta \sigma$, and $\tau' =_\Delta \tau$;
5. If $\Gamma \vdash_\Sigma \lambda x{:}\sigma.\Delta : \Pi x{:}\sigma.\tau$, then $\Gamma, x{:}\sigma \vdash_\Sigma \Delta : \tau$;
6. If $\Gamma \vdash_\Sigma \lambda^{\mathsf{r}} x{:}\sigma.\Delta : \Pi x{:}\sigma.\tau$, then $\Gamma, x{:}\sigma \vdash_\Sigma \Delta : \tau$ and $\wr \Delta \wr =_\eta x$;
7. If $\Gamma \vdash_\Sigma \langle \Delta_1, \Delta_2 \rangle : \sigma \cap \tau$, then $\Gamma \vdash_\Sigma \Delta_1 : \sigma$, $\Gamma \vdash_\Sigma \Delta_2 : \tau$, and $\wr \Delta_1 \wr =_\beta \wr \Delta_2 \wr$;
8. If $\Gamma \vdash_\Sigma [\Delta_1, \Delta_2] : \Pi x{:}\sigma \cup \tau.\rho$, then $\Gamma \vdash_\Sigma \Delta_1 : \Pi y{:}\sigma.\rho\,(\mathsf{in}_1^\tau\ y)$, $\Gamma \vdash_\Sigma \Delta_2 : \Pi y{:}\tau.\rho\,(\mathsf{in}_2^\sigma\ y)$, and $\wr \Delta_1 \wr =_\beta \wr \Delta_2 \wr$;
9. If $\Gamma \vdash_\Sigma \mathsf{pr}_1\, \Delta : \sigma$, then $\Gamma \vdash_\Sigma \Delta{:}\sigma \cap \tau$, for some τ;
10. If $\Gamma \vdash_\Sigma \mathsf{pr}_2\, \Delta : \tau$, then $\Gamma \vdash_\Sigma \Delta{:}\sigma \cap \tau$, for some σ;
11. If $\Gamma \vdash_\Sigma \mathsf{in}_1^\tau\, \Delta : \sigma \cup \tau$, then $\Gamma \vdash_\Sigma \Delta : \sigma$ and $\Gamma \vdash_\Sigma \sigma \cup \tau : \mathsf{Type}$;

12. If $\Gamma \vdash_\Sigma \mathsf{in}_2^\sigma\, \Delta : \sigma \cup \tau$, then $\Gamma \vdash_\Sigma \Delta : \tau$ and $\Gamma \vdash_\Sigma \sigma \cup \tau : \mathsf{Type}$.

The following technical lemma will be useful to prove subject reduction.

Lemma 6.12 (Subderivation)**.**

1. A derivation of $\vdash_\Sigma \cdot$ has a subderivation of Σ sig;
2. A derivation of $\Sigma, a{:}K$ sig has subderivations of Σ sig and $\vdash_\Sigma K$;
3. A derivation of $\Sigma, f{:}\sigma$ sig has subderivations of Σ sig and $\vdash_\Sigma \sigma{:}\mathsf{Type}$;
4. A derivation of $\vdash_\Sigma \Gamma, x{:}\sigma$ has subderivations of Σ sig, $\vdash_\Sigma \Gamma$, and $\Gamma \vdash_\Sigma \sigma{:}\mathsf{Type}$;
5. A derivation of $\Gamma \vdash_\Sigma \alpha$ has subderivations of Σ sig and $\vdash_\Sigma \Gamma$;
6. Given a derivation of the judgment $\Gamma \vdash_\Sigma \alpha$, and a subterm occurring in the subject of this judgment, there exists a derivation of a judgment having this subterm as a subject.

Subject reduction can be proved by easy induction on the structure of the derivations.

Theorem 6.13 (Subject reduction of LF_Δ)**.**

1. If $\Gamma \vdash_\Sigma K$, and $K \to_\Delta K'$, then $\Gamma \vdash_\Sigma K'$;
2. If $\Gamma \vdash_\Sigma \sigma : K$, and $\sigma \to_\Delta \sigma'$, then $\Gamma \vdash_\Sigma \sigma' : K$;
3. If $\Gamma \vdash_\Sigma \Delta : \sigma$, and $\Delta \to_\Delta \Delta'$, then $\Gamma \vdash_\Sigma \Delta' : \sigma$.

Finally, we can prove decidability of all the judgments of LF_Δ.

Theorem 6.14 (Decidability)**.** All the type judgments of LF_Δ are recursively decidable.

Proof. We can easily check judgments in LF_Δ: because of Theorem 6.10 and because all the rules (except (*Conv*)) are syntax directed, we can compute a type or a kind for a term, and then test for *definitional equality*, *i.e.* $=_\Delta$, against the given type or kind; this is achieved by reducing both to their unique normal forms, and, thanks to the confluence property (Theorem 6.1), we only have to check if the normal forms are syntactically equal. □

6.3 Minimal relevant implications and type inclusion

Type inclusion and the rules of subtyping are related to the notion of minimal relevant implication, see [8, 39]. The insight is quite subtle, but ultimately very simple. This is what makes it appealing. The apparently intricate rules of subtyping and type inclusion, which occur in many systems, and might even appear *ad hoc* at times, can all be explained away in our principled approach, by proving that the relevant implication type is inhabited by a term whose essence is essentially a variable.

In the following theorem we show how relevant implication subsumes the type-inclusion rules of the theory Ξ of [7], without rules (5) and (13) (dealing with $\mathtt{U}$) and rule (10) (distributing $\cap$ over $\cup$) in Figure 6.1: we call Ξ' such restricted type theory. Note that the reason to drop subtype rule (10) is due to the fact that we cannot inhabit the type $\sigma \cap (\tau \cup \rho) \to^{\mathsf{r}} (\sigma \cap \tau) \cup (\sigma \cap \rho)$.

$$
\begin{array}{lrcl}
(1) & \|\sigma \leqslant \sigma \cap \sigma\|_\Delta & \stackrel{def}{=} & \langle \Delta, \Delta \rangle \\
(2) & \|\sigma \cup \sigma \leqslant \sigma\|_\Delta & \stackrel{def}{=} & [\lambda x{:}\sigma.x, \lambda x{:}\sigma.x]\,\Delta \\
(3) & \|\sigma_1 \cap \sigma_2 \leqslant \sigma_i\|_\Delta & \stackrel{def}{=} & \mathsf{pr}_i\,\Delta \\
(4) & \|\sigma_i \leqslant \sigma_1 \cup \sigma_2\|_\Delta & \stackrel{def}{=} & \mathsf{in}_i\,\Delta \\
(6) & \|\sigma \leqslant \sigma\|_\Delta & \stackrel{def}{=} & \Delta \\
(7) & \left\|\dfrac{\sigma_1 \leqslant \sigma_2 \quad \tau_1 \leqslant \tau_2}{\sigma_1 \cap \tau_1 \leqslant \sigma_2 \cap \tau_2}\right\|_\Delta & \stackrel{def}{=} & \langle \|\sigma_1 \leqslant \sigma_2\|_{(\mathsf{pr}_1\,\Delta)}, \|\tau_1 \leqslant \tau_2\|_{(\mathsf{pr}_2\,\Delta)} \rangle \\
(8) & \left\|\dfrac{\sigma_1 \leqslant \sigma_2 \quad \tau_1 \leqslant \tau_2}{\sigma_1 \cup \tau_1 \leqslant \sigma_2 \cup \tau_2}\right\|_\Delta & \stackrel{def}{=} & [\lambda x{:}\sigma_1.\mathsf{in}_1^{\tau_2}\ \|\sigma_1 \leqslant \sigma_2\|_x, \lambda x{:}\tau_1.\mathsf{in}_2^{\sigma_2}\ \|\tau_1 \leqslant \tau_2\|_x]\,\Delta \\
(9) & \left\|\dfrac{\sigma \leqslant \tau \quad \tau \leqslant \rho}{\sigma \leqslant \rho}\right\|_\Delta & \stackrel{def}{=} & \|\tau \leqslant \rho\|_{(\|\sigma \leqslant \tau\|_\Delta)} \\
(11) & \|(\sigma \to \tau) \cap (\sigma \to \rho) \leqslant \sigma \to (\tau \cap \rho)\|_\Delta & \stackrel{def}{=} & \lambda x{:}\sigma.\langle (\mathsf{pr}_1\,\Delta)\,x, (\mathsf{pr}_2\,\Delta)\,x \rangle \\
(12) & \|(\sigma \to \rho) \cap (\tau \to \rho) \leqslant (\sigma \cup \tau) \to \rho\|_\Delta & \stackrel{def}{=} & \lambda x{:}\sigma \cup \tau.[\lambda y{:}\sigma.(\mathsf{pr}_1\,\Delta)\,y, \lambda y{:}\tau.(\mathsf{pr}_2\,\Delta)\,y]\,x \\
(14) & \left\|\dfrac{\sigma_2 \leqslant \sigma_1 \quad \tau_1 \leqslant \tau_2}{\sigma_1 \to \tau_1 \leqslant \sigma_2 \to \tau_2}\right\|_\Delta & \stackrel{def}{=} & \lambda x{:}\sigma_2.\ \|\tau_1 \leqslant \tau_2\|_{(\Delta\,\|\sigma_2 \leqslant \sigma_1\|_x)}
\end{array}
$$

Figure 6.11: The coercion function

Theorem 6.15 (Type Inclusion)**.** The judgment $\vdash_\Sigma \Delta : \sigma \to^{\mathsf{r}} \tau$ (where both σ and τ do not contain dependencies or relevant families) holds iff $\sigma \leqslant \tau$ holds in the type theory Ξ' of λ^{BDdL} enriched with new axioms of the form $\sigma_1 \leqslant \sigma_2$ for each constant $c : \sigma_1 \to^{\mathsf{r}} \sigma_2 \in \Sigma$.

Proof.

(if). Follows directly from Lemma 6.11;

(only if). It is possible to write a Δ-term whose essence is an $\eta-$expansion of the identity $(\lambda x.x)$ corresponding to each of the axioms and rules in Ξ'. The Δ-term is obtained by defining a function $\|\sigma \leqslant \tau\|_\Delta$, where $\sigma \leqslant \tau$ is a subtyping derivation tree in the type theory Ξ', which coerce a Δ-term from type σ to type τ, as defined in Figure 6.11.

□

As far as the $\lambda^{\Pi\&}$ system of refinement types introduced by Pfenning in [74], we get the following example:

Example 6.1 (Pfenning's refinement types [74])**.** The judgment $\vdash_\Sigma \sigma \leqslant \tau$ in $\lambda^{\Pi\&}$ can be encoded in LF_Δ by adding a constant of type $\sigma \to^{\mathsf{r}} \tau$ to Σ', where the latter is the signature obtained from Σ by replacing each clause of the form $a_1 :: a_2$ or $a_1 \leqslant a_2$ in Σ by a constant of type $a_1 \to^{\mathsf{r}} a_2$.

Moreover, while Pfenning needs to add explicitly the rules of subtyping (*i.e.* the theory of $\leqslant$) in $\lambda^{\Pi\&}$, we inherit them naturally in LF_Δ from the rules for minimal relevant implication.

Let $\Gamma \stackrel{def}{=} \{x_1{:}\sigma_1, \ldots, x_n{:}\sigma_n\}$ ($i \neq j$ implies $x_i \not\equiv x_j$), and $\Gamma, x{:}\sigma \stackrel{def}{=} \Gamma \cup \{x{:}\sigma\}$

Let $\Sigma \stackrel{def}{=} \{c_1{:}\sigma_1, \ldots, c_n{:}\sigma_n\}$, and $\Sigma, c{:}\sigma \stackrel{def}{=} \Sigma \cup \{c{:}\sigma\}$

Valid Signatures

$$\frac{}{\cdot \text{ sig}}\ (\epsilon\Sigma) \qquad \frac{\Sigma \text{ sig} \quad \vdash_\Sigma \sigma : s \quad a \notin \mathsf{Dom}(\Sigma) \quad s \in \{\mathsf{Type}, \mathsf{Kind}\}}{\Sigma, c{:}\sigma \text{ sig}}\ (\Delta\Sigma)$$

Valid Contexts

$$\frac{\Sigma \text{ sig}}{\vdash_\Sigma \cdot}\ (\epsilon\Gamma) \qquad \frac{\vdash_\Sigma \Gamma \quad \Gamma \vdash_\Sigma \sigma : \mathsf{Type} \quad x \notin \mathsf{Dom}(\Gamma)}{\vdash_\Sigma \Gamma, x{:}\sigma}\ (\sigma\Gamma)$$

Figure 6.12: Pure Type System presentation of the Δ-framework (signature and context)

6.4 Pure Type System presentation of LF_Δ

Pure Type Systems (PTS) were introduced by Barendregt in [10]. The gist of PTSs is that we only have one syntactical set containing the terms, families and kinds. Some constants (here, Type and Kind) are called sorts. We call $\mathrm{LF}_\Delta^{\mathrm{PTS}}$ the PTS version of LF_Δ. For $\mathrm{LF}_\Delta^{\mathrm{PTS}}$, we have the following syntax:

$$\begin{array}{rcl} \Delta & ::= & \mathsf{Type} \mid \mathsf{Kind} \mid c \mid x \mid \lambda x{:}\sigma.\Delta \mid \Delta\,\Delta \mid \lambda^{\mathsf{r}} x{:}\Delta.\Delta \mid \Delta \cdot^{\mathsf{r}} \Delta \mid \langle \Delta, \Delta \rangle \mid \\ & & [\Delta, \Delta] \mid \mathsf{pr}_1\,\Delta \mid \mathsf{pr}_2\,\Delta \mid \mathsf{in}_1^\Delta\,\Delta \mid \mathsf{in}_2^\Delta\,\Delta \mid \Pi x{:}\Delta.\Delta \mid \Delta \to^{\mathsf{r}} \Delta \mid \\ & & \Delta \cup \Delta \mid \Delta \cap \Delta \end{array}$$

We define $\mathcal{R} \stackrel{def}{=} \{(\mathsf{Type}, \mathsf{Type}), (\mathsf{Type}, \mathsf{Kind})\}$. The typing rules are given in Figures 6.12 and 6.13.

We now define objects, families and kinds in $\mathrm{LF}_\Delta^{\mathrm{PTS}}$.

Definition 6.2 (Objects, families, and kinds)**.** In a signature Σ, for any Δ:

1. if there is some Γ, σ, such that $\Gamma \vdash_\Sigma \Delta : \sigma$ and $\Gamma \vdash_\Sigma \sigma : \mathsf{Type}$, then we say that Δ is an object in the signature Σ;

2. if there is some Γ, σ, such that $\Gamma \vdash_\Sigma \Delta : \sigma$ and $\Gamma \vdash_\Sigma \sigma : \mathsf{Kind}$, then we say that Δ is a family in the signature Σ;

3. if there is some Γ, such that $\Gamma \vdash_\Sigma \Delta : \mathsf{Kind}$, then we say that Δ is a kind in the signature Σ.

The next theorem states that all objects, families and kinds in LF_Δ remain respectively well-typed objects, families and kinds in $\mathrm{LF}_\Delta^{\mathrm{PTS}}$.

Theorem 6.16 (Preservation)**.**

1. for any signature Σ, if Σ sig in LF_Δ, then Σ sig in $\mathrm{LF}_\Delta^{\mathrm{PTS}}$ (assuming the alphabet for atom types and the alphabet for constants are the same);

Valid Terms

$$\frac{\vdash_\Sigma \Gamma}{\Gamma \vdash_\Sigma \mathsf{Type} : \mathsf{Kind}}\ (Type) \qquad \frac{\Gamma \vdash_\Sigma \sigma : s_1 \quad \Gamma, x{:}\sigma \vdash_\Sigma \tau : s_2 \quad (s_1, s_2) \in \mathcal{R}}{\Gamma \vdash_\Sigma \Pi x{:}\sigma.\tau : \tau}\ (sI)$$

$$\frac{\vdash_\Sigma \Gamma \quad c{:}\sigma \in \Sigma}{\Gamma \vdash_\Sigma c : \sigma}\ (Const) \qquad \frac{\vdash_\Sigma \Gamma \quad x{:}\sigma \in \Gamma}{\Gamma \vdash_\Sigma x : \sigma}\ (Var)$$

$$\frac{\Gamma \vdash_\Sigma \sigma : \mathsf{Type} \quad \Gamma \vdash_\Sigma \tau : \mathsf{Type}}{\Gamma \vdash_\Sigma \sigma \to^{\mathsf{r}} \tau : \mathsf{Type}}\ (\mathsf{r}I) \qquad \frac{\begin{array}{l}\Gamma \vdash_\Sigma \Delta : \sigma \\ \Gamma \vdash_\Sigma \tau : s \quad \sigma =_\Delta \tau\end{array}}{\Gamma \vdash_\Sigma \Delta : \tau}\ (Conv)$$

$$\frac{\Gamma \vdash_\Sigma \sigma : \mathsf{Type} \quad \Gamma \vdash_\Sigma \tau : \mathsf{Type}}{\Gamma \vdash_\Sigma \sigma \cap \tau : \mathsf{Type}}\ (\cap I) \qquad \frac{\Gamma \vdash_\Sigma \sigma : \mathsf{Type} \quad \Gamma \vdash_\Sigma \tau : \mathsf{Type}}{\Gamma \vdash_\Sigma \sigma \cup \tau : \mathsf{Type}}\ (\cup I)$$

$$\frac{\Gamma, x{:}\sigma \vdash_\Sigma \Delta : \tau \quad \Gamma \vdash_\Sigma \Pi x{:}\sigma.\tau : s}{\Gamma \vdash_\Sigma \lambda x{:}\sigma.\Delta : \Pi x{:}\sigma.\tau}\ (\Pi I) \qquad \frac{\Gamma \vdash_\Sigma \Delta_1 : \Pi x{:}\sigma.\tau \quad \Gamma \vdash_\Sigma \Delta_2 : \sigma}{\Gamma \vdash_\Sigma \Delta_1\, \Delta_2 : \tau[\Delta_2/x]}\ (\Pi E)$$

$$\frac{\begin{array}{l}\wr \Delta \wr =_\eta x \\ \Gamma, x{:}\sigma \vdash_\Sigma \Delta : \tau \quad \Gamma \vdash_\Sigma \sigma \to^{\mathsf{r}} \tau : \mathsf{Type}\end{array}}{\Gamma \vdash_\Sigma \lambda^{\mathsf{r}} x{:}\sigma.\Delta : \sigma \to^{\mathsf{r}} \tau}\ (\to^{\mathsf{r}} I) \qquad \frac{\Gamma \vdash_\Sigma \Delta_1 : \sigma \to^{\mathsf{r}} \tau \quad \Gamma \vdash_\Sigma \Delta_2 : \sigma}{\Gamma \vdash_\Sigma \Delta_1 \cdot^{\mathsf{r}} \Delta_2 : \tau}\ (\to^{\mathsf{r}} E)$$

$$\frac{\Gamma \vdash_\Sigma \Delta : \sigma \cap \tau}{\Gamma \vdash_\Sigma \mathsf{pr}_1\, \Delta : \sigma}\ (\cap E_l) \qquad \frac{\Gamma \vdash_\Sigma \Delta : \sigma \cap \tau}{\Gamma \vdash_\Sigma \mathsf{pr}_2\, \Delta : \tau}\ (\cap E_r)$$

$$\frac{\Gamma \vdash_\Sigma \Delta : \sigma \quad \Gamma \vdash_\Sigma \sigma \cup \tau : \mathsf{Type}}{\Gamma \vdash_\Sigma \mathsf{in}_1^\tau\, \Delta : \sigma \cup \tau}\ (\cup I_l) \qquad \frac{\Gamma \vdash_\Sigma \Delta : \tau \quad \Gamma \vdash_\Sigma \sigma \cup \tau : \mathsf{Type}}{\Gamma \vdash_\Sigma \mathsf{in}_2^\sigma\, \Delta : \sigma \cup \tau}\ (\cup I_r)$$

$$\frac{\begin{array}{l}\wr \Delta_1 \wr =_\eta \wr \Delta_2 \wr \\ \Gamma \vdash_\Sigma \Delta_1 : \sigma \qquad \Gamma \vdash_\Sigma \Delta_2 : \tau\end{array}}{\Gamma \vdash_\Sigma \langle \Delta_1, \Delta_2 \rangle : \sigma \cap \tau}\ (\cap I) \qquad \frac{\begin{array}{l}\Gamma \vdash_\Sigma \Delta_1 : \Pi y{:}\sigma.\rho[\mathsf{in}_1^\tau\, y/x] \qquad \wr \Delta_1 \wr =_\eta \wr \Delta_2 \wr \\ \Gamma \vdash_\Sigma \Delta_2 : \Pi y{:}\tau.\rho[\mathsf{in}_2^\sigma\, y/x] \quad \Gamma, x{:}\sigma \cup \tau \vdash_\Sigma \rho : \mathsf{Type}\end{array}}{\Gamma \vdash_\Sigma [\Delta_1, \Delta_2] : \Pi x{:}\sigma \cup \tau.\rho}\ (\cup E)$$

Figure 6.13: Pure Type System presentation of the Δ-framework (terms)

2. for any context Γ, if $\vdash_\Sigma \Gamma$ in LF_Δ, then $\vdash_\Sigma \Gamma$ in $\mathrm{LF}_\Delta^{\mathrm{PTS}}$;
3. for any kind K, if $\Gamma \vdash_\Sigma K$ in LF_Δ, then $\Gamma \vdash_\Sigma K : \mathsf{Kind}$ in $\mathrm{LF}_\Delta^{\mathrm{PTS}}$, and K is also a kind in $\mathrm{LF}_\Delta^{\mathrm{PTS}}$;
4. for any family σ, if $\Gamma \vdash_\Sigma \sigma : K$ in LF_Δ, then $\Gamma \vdash_\Sigma \sigma : K$ in $\mathrm{LF}_\Delta^{\mathrm{PTS}}$, and σ is a family in $\mathrm{LF}_\Delta^{\mathrm{PTS}}$;
5. for any object Δ, if $\Gamma \vdash_\Sigma \Delta : \sigma$ in LF_Δ, then $\Gamma \vdash_\Sigma \Delta : \sigma$ in $\mathrm{LF}_\Delta^{\mathrm{PTS}}$, and Δ is an object in $\mathrm{LF}_\Delta^{\mathrm{PTS}}$.

Proof. All the parts are easily proved by mutual induction on the derivation tree. □

6.5 Future Work

There is still a lot of research to do in the domain of Church-style λ-calculus with intersection, union, and dependent types. Among some interesting questions, we could

mention:

1. LF_Δ *in Canonical Form*: we presented LF_Δ in the standard LF format in order to support intuition, and in the PTS format for conciseness. It would be worthwhile however, to attempt to formulate LF_Δ in the style of Harper and Licata [53], using only canonical forms without reductions, especially in view of proving *adequacy results*. The terms peculiar to LF_Δ would then introduce new clauses in the definition of canonical and atomic terms. The principle to follow in this task is that atomic terms synthesize their type, while canonical terms are checked against their type. We are currently exploring the following extension:

$$\begin{array}{rcl} M & ::= & \ldots \mid \lambda^{\mathsf{r}} x.M \mid \langle M, M\rangle \mid [M, M] \mid \mathsf{in}_1\, M \mid \mathsf{in}_2\, M \\ R & ::= & \ldots \mid \mathsf{pr}_1\, R \mid \mathsf{pr}_2\, R \mid R \cdot^{\mathsf{r}} M \end{array}$$

   Notice the somewhat surprising treatment of the [,] constructor, which is not really an elimination construct but rather behaves as another form of abstraction. Accordingly hereditary substitution needs to be extended.

   An intriguing issue[1] is to explore the connections between strong implication and the *singleton type* of the identity function. This could lead also to an internalization of the essence function;

2. *Adequacy, Canonical Forms, Exotic terms*: in the presence of union types, we have to pay special attention to the exact formulation of adequacy results, as in the Harrop's formulæ example of Chapter 8. Otherwise exotic terms arise, such as $[\lambda x{:}\sigma.C[x], \lambda x{:}\tau.D[x]]\, y$, where $C[-]$ and $D[-]$ are distinct contexts (*i.e.* terms with holes), which cannot be naturally simplified even if $\wr C[-] \wr \equiv \wr D[-] \wr$. More work needs to be done to streamline how to exclude, or even capitalize, on exotic terms.

[1] Raised by one of the referees of [57].

Chapter 7

Implementation of the theorem prover Bull

This chapter presents the implementation of a prototype of an Interactive Theorem Prover (ITP) based on the Δ-framework. I have personally been writing this theorem prover from scratch for three years, and it is called *Bull*[1] [88, 87]. We have a command-line interface program where the user can declare axioms, terms, and perform computations. These terms can be incomplete, therefore the typechecking algorithm uses unification to try to construct the missing subterms.

In Chapter 5, we have implemented the subtyping algorithm which extends the well-known algorithm for intersection types, designed by Hindley [54], with union types. The subtyping algorithm has been mechanically proved correct in Coq, extending the mechanized proof of a subtyping algorithm for intersection types of Bessai [16].

We have implemented several features. A Read-Eval-Print-Loop allows to define axioms and definitions, and performs some basic terminal-style features like error pretty-printing, subexpressions highlighting, and file loading. Moreover, it can typecheck a proof or normalize it. We use the syntax of Pure Type Systems [14] to improve the compactness and the modularity of the kernel. Abstract and concrete syntax are mostly aligned: the concrete syntax is similar to the concrete syntax of Coq.

We have designed a *higher-order unification algorithm* for terms, while typechecking and partial type inference are done by our *bidirectional refinement algorithm*, similar to the one found in [6]. The refinement can be split into two parts: the essence refinement and the typing refinement. The bidirectional refinement algorithm aims to have partial type inference, and to give as much information as possible to the unifier. For instance, if we want to find a $?y$ such that $\vdash_\Sigma \langle \lambda x{:}\sigma.x, \lambda x{:}\tau.?y\rangle : (\sigma \to \sigma) \cap (\tau \to \tau)$, we can infer that $x{:}\tau \vdash ?y : \tau$ and that $\wr ?y \wr =_\beta x$.

This chapter is organized as follows: in Section 7.1, we explain the commonly-used *de Bruijn indices*. In Section 7.2, we introduce the language we have implemented. In Section 7.3, we define the reduction rules and explain the evaluation process. In Section 7.4, we present the subtyping algorithm. In Section 7.5, we present the unifier. In Section 7.6, we present the refiner which does partial typechecking and type reconstruction. In Section 7.7, we present the Read-Eval-Print-Loop. In Section 7.8, we present possible enhancements of the software.

[1] In reference to the Turin fountains.

7.1 de Bruijn indices in Bull

In this section, we review the de Bruijn indices technique [35] as it is implemented in Bull, using the pure λ-calculus for simplicity.

The principle is to replace every variable x is by its de Bruijn index i, which is the number of binders we encounter when traversing the term until we meet the binder of x. Constants c are still allowed, and are not replaced by an integer. We get the following grammar:

$$M ::= c \mid i \mid \lambda x.M \mid M\,M$$

The name x on the binder λx could seem superfluous, but it is convenient if we want to get back the names of the variables. N.G. de Bruijn counts from 1 in his historical paper, but we decided to count from 0 in the Bull implementation.

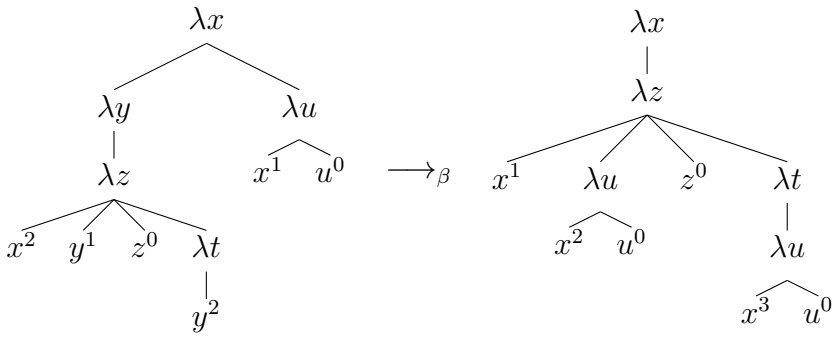
We will use $\lambda x.((\lambda y.\lambda z.x\,y\,z\,(\lambda t.y))(\lambda u.x\,u))$ as a running example which is translated as $\lambda x.((\lambda y.\lambda z.2\,1\,0\,(\lambda t.2))(\lambda u.1\,0))$. There is one β-redex, which consist of the function $\lambda y.\lambda z.x\,y\,z(\lambda t.y)$ applied to the argument $\lambda u.x\,u$. This term reduces to:

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

It is translated using de Bruijn indices as:

$$\lambda x.(\lambda z.1\,(\lambda u.2\,0)\,0\,(\lambda t.\lambda u.3\,0))$$

It is easier to read these terms by looking at their syntax tree, where variables are decorated with their de Bruijn index. Intuitively, we can see the de Bruijn indices as the distance between variables and their binders:



As you can see, β-reduction with de Bruijn indices requires a subtle update of the indices. Simply speaking, the β-reduction algorithm consists of a tree traversal where every de Bruijn index is compared to the depth of the consumed binder (λy in the example):

- if the de Bruijn index is strictly greater than the depth of the consumed binder, it corresponds to a variable bound to a binder above the consumed binder (λx in the example). Therefore this de Bruijn index is decremented;
- if the de Bruijn index is strictly smaller than the depth of the consumed binder, it corresponds to a variable bound to a binder below the consumed binder (λz in the example). Therefore this de Bruijn index is unchanged;
- if the reference binder is the depth of the consumed binder, it corresponds to the variable to be substituted. We replace the node with an updated version of the argument. It has to be computed anew for each replacement (it is done twice in the example), and it is a tree traversal where the de Bruijn indices are modified:

- for de Bruijn indices corresponding to variables bound outside the argument (x in the example), we update it by adding the number of extra binders (in the example, the extra binders are λz in the first case, and $\lambda z, \lambda t$ in the second case);
- for de Bruijn indices corresponding to variables bound inside the argument (u in the example), we do not change anything.

This auxiliary function is called `lift` in the literature (see *e.g.* [5]), where `lift k n t` updates the tree `t`, which is a subtree of the argument of the redex, where we are under `k` local binders, and there are `n` extra binders. The argument `k` is used to determine whether a de Bruijn index correspond to a variable inside or outside the argument, and `n` is the value to add, where appropriate.

7.2 Syntax of terms

We present a syntax for the language we have implemented. We use a Pure Type System approach, therefore all the terms are read through the same parser. The main differences with the Δ-framework presented in Chapter 6 are the additions of a placeholder and meta-variables, used by the refiner. We also added a **let** operator and changed the syntax of the strong sum so it looks more like the concrete syntax used in the implementation. A meta-variable $?x[\Delta_1; ...; \Delta_n]$ has the, so called, *suspended substitutions* $\Delta_1; ...; \Delta_n$, which will be explained clearly in Subsection 7.2.4. Finally, following the Cervesato-Pfenning jargon [25], applications are in *spine form*, *i.e.* the arguments of a function are stored together in a list, exposing the head of the term separately.

Δ, σ	$::=$	s	Sort
	$\mid$	c	Constant
	$\mid$	x	Variable
	$\mid$	$_$	Placeholder
	$\mid$	$?x[\Delta; ...; \Delta]$	Meta-variable
	$\mid$	$\mathbf{let}\ x{:}\sigma := \Delta\ \mathbf{in}\ \Delta$	Local definition
	$\mid$	$\Pi x{:}\sigma.\Delta$	Dependent product
	$\mid$	$\lambda x{:}\sigma.\Delta$	λ-abstraction
	$\mid$	$\Delta\, S$	Application
	$\mid$	$\sigma \cap \sigma$	Intersection
	$\mid$	$\sigma \cup \sigma$	Union
	$\mid$	$\langle \Delta, \Delta \rangle$	Strong pair
	$\mid$	$\mathsf{pr}_1\, \Delta$	Left projection
	$\mid$	$\mathsf{pr}_2\, \Delta$	Right projection
	$\mid$	$\mathbf{smatch}\ \Delta\ \mathbf{return}\ \sigma\ \mathbf{with}\, [x{:}\sigma \Rightarrow \Delta \mid x{:}\sigma \Rightarrow \Delta]$	Strong sum
	$\mid$	$\mathsf{in}_1\, \sigma\, \Delta$	Left injection
	$\mid$	$\mathsf{in}_2\, \sigma\, \Delta$	Right injection
	$\mid$	$\mathbf{coe}\, \sigma\, \Delta$	Coercion

Spines for Δ-terms have the following syntax:

$$S ::= () \mid (S; \Delta)$$

We also have a similar syntax for the essence:

$$
\begin{array}{rcll}
M, \varsigma & ::= & s & \text{Sort} \\
 & | & c & \text{Constant} \\
 & | & x & \text{Variable} \\
 & | & _ & \text{Placeholder} \\
 & | & ?x[M; ...; M] & \text{Meta-variable} \\
 & | & \textbf{let } x := M \textbf{ in } M & \text{Local definition} \\
 & | & \Pi x{:}\varsigma.\varsigma & \text{Dependent product} \\
 & | & \lambda x.M & \lambda\text{-abstraction} \\
 & | & M\,R & \text{Application} \\
 & | & \varsigma \cap \varsigma & \text{Intersection} \\
 & | & \varsigma \cup \varsigma & \text{Union}
\end{array}
$$

Spines for essences have the following syntax:

$$R ::= () \mid (R; M)$$

Note that essences of types (noted ς) belongs to the same syntactical set as essences of terms.

7.2.1 Concrete syntax

The concrete syntax of the terms has been implemented with OCamllex and OCamlyacc. Its simplified syntax is as follows:

```
term ::=
| Type
| let ID [args] [: term] := term in term
| ID                            # variables and constants
| forall args, term             # dependent product
| term -> term                  # non-dependent product
| fun args => term              # lambda-abstraction
| term term                     # application
| term & term                   # intersection of types
| term | term                   # union of types
| <term,term>                   # strong pair
| proj_l term                   # left projection of a strong pair
| proj_r term                   # right projection of a strong pair
| smatch term [as ID] [return term] with ID [: term] => term,
                                         ID [: term] => term end
                                # strong sum
| inj_l term term               # left injection of a strong sum
| inj_r term term               # right injection of a strong sum
| coe term term                 # coercion
| _                             # wildcard
```

Identifiers `ID` refers to any alphanumeric string (possibly with underscores and apostrophes). The non-terminal symbol `args` correspond to a non-empty sequence of arguments,

where an argument is an identifier, and can be given with its type. In the latter case, you should parenthesize it, for instance `(x : A)`, and if you want to assign the same type to several consecutive arguments, you can *e.g.* write `(x y z : A)`. Strong sums have a complicated syntax. For instance, consider this term:

```
smatch foo as x return T with y : T1 ⇒ bar, z : T2 ⇒ baz end
```

The above term in the concrete syntax corresponds to

$$\textbf{smatch}\ \texttt{foo}\ \textbf{return}\ \lambda x{:}_.\texttt{T}\ \textbf{with}\ [y{:}\texttt{T1} \Rightarrow \texttt{bar} \mid x{:}\texttt{T2} \Rightarrow \texttt{baz}]$$

in the abstract syntax. The concrete syntax thus guarantees that the returned type is a λ-abstraction, and it allows a simplified behaviour of the type reconstruction algorithm (see rule (*Ssum*) in Figure 7.3). The behaviour of the concrete syntax is intended to mimic Coq[2].

7.2.2 Implementation of the syntax

In the OCaml implementation, Δ-terms and their types along with essences and type essences are represented with a single type called `term`. It allows some functions (such as the normalization function) to be applied both on Δ-terms and on essences.

```
type term =
  | Sort of location * sort
  | Let of location * string * term * term * term (* let s : t1 := t2 in t3 *)
  | Prod of location * string * term * term (* forall s : t1, t2 *)
  | Abs of location * string * term * term (* fun s : t1 => t2 *)
  | App of location * term * term list (* t t1 t2 ... tn *)
  | Inter of location * term * term (* t1 & t2 *)
  | Union of location * term * term (* t1 | t2 *)
  | SPair of location * term * term (* < t1, t2 > *)
  | SPrLeft of location * term (* proj_l t1 *)
  | SPrRight of location * term (* proj_r t1 *)
  | SMatch of location * term * term * string * term * term * string * term * term
                 (* match t1 return t2 with s1 : t3 => t4 , s2 : t5 => t6 end *)
  | SInLeft of location * term * term (* inj_l t1 t2 *)
  | SInRight of location * term * term (* inj_r t1 t2 *)
  | Coercion of location * term * term (* coe t1 t2 *)
  | Var of location * int (* de Bruijn index *)
  | Const of location * string (* variable name *)
  | Underscore of location (* meta-variables before analysis *)
  | Meta of location * int * (term list) (* index and substitution *)
```

The constructors of `term` contain the location information retrieved by the parser that allows the typechecker to give the precise location of a subterm to the user, in case of error.

The `App` constructor takes as parameters the applied function and the list of all the arguments. The list of parameters is used as a stack, hence the rightmost argument is the head of the list, and can easily be removed in the OCaml recursive functions. The variables are referred to as strings in the `Const` constructor, and as de Bruijn indices in `Var` constructors.

[2]Even though I could not decipher the Coq parser.

The parser does not compute de Bruijn indices, it gives the variables as strings. The function `fix_index` replaces bound variables by de Bruijn indices. We still keep track of the string names of the variables, in case we have to print them back. Its converse function, `fix_id`, replaces the de Bruijn indices with the previous strings, possibly updating the string names in case of name clashes. For instance, the string printed to the user, showing the normalized form of (`fun (x y : nat) ⇒ x) y`, is `fun y0 : nat ⇒ y :` the bound variable `y` has been renamed `y0`. The meta-variables are generated by the typecheckers, and their identifier is an integer.

We have defined several helper functions to ease the process of terms: there is the most generic function `visit_term f g h t`, which looks at the children of the term `t`, and:

1. every child `t1` outside of a binder is replaced with `f t1`;
2. every child `t1` inside the binding of a variable whose name (a string) is `s` is replaced with `g s t1`, while `s` is replaced with `h s t1`. The functions `g` and `h` takes a string as an argument, for helping the implementation of the `fix_index` and `fix_id` functions.

The function `map_term` is a kind of mapping function: `map_term k f t` finds every variable `Var(l, n)` inside the term, and replaces it by `f (k+offset) l n`, where `offset` is the number of extra bindings.

```
let lift k n =
  map_term k
      (fun k l m → if m < k then Var (l, m) else Var (l, m+n))
```

The `lift` and `map_term` functions allow us to define a substitution in a clean way:

```
(* Transform (lambda x. t1) t2 into t1[t2/x] *)
let beta_redex t1 t2 =
  let subst k l m =
    if m < k then Var (l, m) (* bound variable *)
    else if m = k then (* x *)
      lift 0 k t2
    else (* the enclosing lambda goes away *)
      Var (l, m−1)
  in map_term 0 subst t1
```

7.2.3 Environments

There are four kinds of environments, namely

1. the *global environment* (noted Σ). The global environment holds constants which are fully typechecked:

$$\Sigma ::= \cdot \mid \Sigma, c{:}\varsigma@\sigma \mid \Sigma, c := M@\Delta : \varsigma@\sigma$$

   Intuitively, $c{:}\varsigma@\sigma$ is a declaration of a constant (or axiom), and $c := M@\Delta : \varsigma@\sigma$ corresponds to a global definition.

2. the *local environment* (noted Γ). It is used for the first step of typechecking, and looks like a standard environment:

$$\Gamma ::= \cdot \mid \Gamma, x{:}\sigma \mid \Gamma, x := \Delta : \sigma$$

   Intuitively, $x{:}\sigma$ is a variable introduced by a λ-abstraction, and $x := \Delta : \sigma$ is a local definition introduced by a **let**.

3. the *essence environment* (noted Ψ). It is used for the second step of typechecking, and holds the essence of the local variables:

$$\Psi ::= \cdot \mid \Psi, x \mid \Psi, x := M$$

Intuitively, x is a variable introduced by a λ-abstraction, and $x := M$ is a local definition introduced by a **let**. Notice that the variable x in the BNF expression Ψ, x carries almost no information. However, since local variables are referred to by their de Bruijn indices, and these indices are actually their position in the environment, it follows that they have to appear in the environment, even when there is no additional information.

4. the *meta-environment* (noted Φ). It is used for unification, and records meta-variables and their instantiation whenever the unification algorithm has found a solution:

$$\begin{array}{rcl} \Phi & ::= & \cdot \mid \Phi, \mathbf{sort}(?x) \mid \Phi, ?x := s \mid \Phi, (\Gamma \vdash ?x : \sigma) \mid \Phi, (\Gamma \vdash ?x := \Delta : \sigma) \mid \\ & & \Phi, \Psi \vdash ?x \mid \Phi, \Psi \vdash ?x := M \end{array}$$

Intuitively, since there are some meta-variables for which we know they have to be sorts, it follows that $\mathbf{sort}(?x)$ declares a meta-variable $?x$ which correspond either to Type or Kind, and $?x := s$ is the instantiation of a sort $?x$. Also, $\Gamma \vdash ?x : \sigma$ is the declaration of a meta-variable $?x$ of type σ which appeared in a local environment Γ, and $\Gamma \vdash ?x := \Delta : \sigma$ is the instantiation of the meta-variable $?x$. Concerning meta-variables inside essences, $\Psi \vdash ?x$ is the declaration of a meta-variable $?x$ in an essence environment Ψ, and $\Psi \vdash ?x := M$ is the instantiation of $?x$.

7.2.4 Suspended substitution

We shortly introduce suspended substitution, as presented in [6]. Let's consider the following example: if we want to unify $(\lambda x{:}\sigma.?y)\, c_1$ with c_1, we could unify $?y$ with c_1 or with x, the latter being the preferred solution. However, if we normalize $(\lambda x{:}\sigma.?y)\, c_1$, we should record the fact that c_1 can be substituted by any occurrence of x appearing the term to be replaced by $?y$. That is the purpose of suspended substitution: the term is actually noted $(\lambda x{:}\sigma.?y[x])\, c_1$ and reduces to $?y[c_1]$, noting that c_1 has replaced x.

Definition 7.1 (Type-erase function and suspended substitution)**.**

1. the vector $x_1; \ldots; x_n$ is created using the type-erase function $\widehat{\cdot}$, defined as

$$x_1 : \sigma_1 \widehat{; \ldots x_n} : \sigma_n \stackrel{def}{=} x_1; \ldots; x_n$$

2. when we want to create a new meta-variable in a local context $\Gamma = x_1 : \sigma_1; \ldots x_n : \sigma_n$, we create a meta-variable $?y[\widehat{\Gamma}] \equiv ?y[x_1; \ldots; x_n]$. The vector $\Delta_1; \ldots; \Delta_n$ inside $?y[\Delta_1; \ldots; \Delta_n]$ is the suspended substitution of $?y$. Substitutions for meta-variable and their suspended substitution are propagated as follows:

$$\begin{array}{rcl} ?y[\Delta_1; \ldots; \Delta_n][\Delta/x] & \stackrel{def}{=} & ?y[\Delta_1[\Delta/x]; \ldots; \Delta_n[\Delta/x]] \\ ?y[M_1; \ldots; M_n][N/x] & \stackrel{def}{=} & ?y[M_1[N/x]; \ldots; M_n[N/x]] \end{array}$$

7.3 The evaluator of Bull

The evaluator follows the applicative order strategy, which recursively normalizes all subterms from left to right (with the help of the `visit_term` function), then:

- if the resulting term is a redex, reduces it, then use the same strategy again;
- or else, the resulting term is in normal form.

7.3.1 Reduction rules

The reduction notions, from which we can defined one-step reduction, multistep reduction, and equivalence relation, are defined below.

1. for Δ-terms:

$$
\begin{array}{rcll}
(\lambda x{:}\sigma.\Delta_1)\,\Delta_2 & \mapsto_\beta & \Delta_1[\Delta_2/x] & \\
\lambda x{:}\sigma.\Delta\, x & \mapsto_\eta & \Delta & \text{if } x \notin \mathsf{Fv}(\Delta) \\
\mathsf{pr}_i\,\langle \Delta_1, \Delta_2\rangle & \mapsto_{\mathsf{pr}_i} & \Delta_i & \\
\multicolumn{4}{l}{\textbf{smatch}\ \mathsf{in}_i\,\Delta_3\ \textbf{return}\ \sigma\ \textbf{with}\,[x{:}\tau \Rightarrow \Delta_1 \mid x{:}\rho \Rightarrow \Delta_2]} \\
 & \mapsto_{\mathsf{in}_i} & \Delta_i[\Delta_3/x] & \\
\textbf{let}\ x{:}\sigma := \Delta_1\ \textbf{in}\ \Delta_2 & \mapsto_\zeta & \Delta_2[\Delta_1/x] & \\
c & \mapsto_{\delta\Sigma} & \Delta & \text{if } (c := M@\Delta : \varsigma@\sigma) \in \Sigma \\
x & \mapsto_{\delta\Gamma} & \Delta & \text{if } (x := \Delta : \sigma) \in \Gamma \\
?x[\Delta_1; \ldots; \Delta_n] & \mapsto_{\delta\Phi} & \Delta[\overrightarrow{\widehat{\Gamma}/\Delta_i}] & \text{if } (\Gamma \vdash ?x := \Delta : \sigma) \in \Phi \\
?x[\Delta_1; \ldots; \Delta_n] & \mapsto_{\delta\Phi} & s & \text{if } (\Gamma \vdash ?x := s) \in \Phi
\end{array}
$$

2. for pure λ-terms:

$$
\begin{array}{rcll}
(\lambda x{:}\sigma.M)\,N & \mapsto_\beta & M[N/x] & \\
\lambda x{:}\sigma.M\, x & \mapsto_\eta & M & \text{if } x \notin \mathsf{Fv}(M) \\
\textbf{let}\ x := M\ \textbf{in}\ N & \mapsto_\zeta & N[M/x] & \\
c & \mapsto_{\delta\Sigma} & M & \text{if } (c := M@\Delta : \varsigma@\sigma) \in \Sigma \\
x & \mapsto_{\delta\Psi} & M & \text{if } (x := M) \in \Psi \\
?x[M_1; \ldots; M_n] & \mapsto_{\delta\Phi} & N[\overrightarrow{\widehat{\Psi}/M_i}] & \text{if } (\Psi \vdash ?x := M) \in \Phi
\end{array}
$$

7.3.2 Implementation

When the user inputs a term, the refiner creates meta-variables and tries to instantiate them, but this should remain as much as possible invisible to the user. Therefore the term returned by the refiner should be meta-variable free, even though not in normal

form. Thus terms in the global signature Σ are meta-variable free, and the $\delta\Phi$ reductions are only used by the unifier and the refiner.

If we want to normalize a term, The function `strongly_normalize` works on both Δ-terms and pure λ-terms, and supposes that the given term is meta-variable free. Note that reductions can create odd spines, for instance if you consider the term $(\lambda x{:}\sigma.x\, S_1)\,(\Delta\, S_2)$, a simple β-redex would give $\Delta\, S_2\, S_1$, therefore we merge S_2 and S_1 in a single spine.

```
let rec strongly_normalize is_essence env ctx t =
  let sn_children = visit_term (strongly_normalize is_essence env ctx)
              (fun _ → strongly_normalize is_essence
                       env (Env.add_var ctx (DefAxiom ("",nothing))))
              (fun id _ → id)
  in let sn = strongly_normalize is_essence env ctx in
  (* Normalize the children *)
  let t = sn_children t in
  match t with
  (* Spine fix *)
  | App(l, App(l',t1,t2), t3) →
     sn (App(l, t1, List.append t2 t3))
  (* Beta-redex *)
  | App (l, Abs (l', _,_, t1), t2 :: [ ]) →
     sn (beta_redex t1 t2)
  | App (l, Abs (l', x,y, t1), t2 :: t3)
    → sn @@ app l (sn (App(l,Abs (l',x,y, t1), t3))) t2
  | Let (l, _, t1, t2, t3) → sn (beta_redex t2 t1)
  (* Delta-redex *)
  | Var (l, n) → let (t1, _) = Env.find_var ctx n in
       (match t1 with
        | Var _ → t1
        | _ → sn t1)
  | Const (l, id) → let o = Env.find_const is_essence env id in
              (match o with
               | None → Const(l, id)
               | Some (Const (_,id') as t1,_) when id = id' → t1
               | Some (t1,_) → sn t1)
  (* Eta-redex *)
  | Abs (l,_, _, App (l', t1, Var (_,0) :: l2))
    → if is_eta (App (l', t1, l2)) then
        let t1 = lift 0 (−1) t1 in
        match l2 with
        | [ ] → t1
        | _ → App (l', t1, List.map (lift 0 (−1)) l2)
      else t
  (* Pair-redex *)
  | SPrLeft (l, SPair (l', x,_)) → x
  | SPrRight (l, SPair (l', _, x)) → x
  (* inj-reduction *)
  | SMatch (l, SInLeft(l',_,t1), _, id1, _, t2, id2, _, _) →
     sn (beta_redex t2 t1)
  | SMatch (l, SInRight(l',_,t1), _, id1, _, _, id2, _, t2) →
     sn (beta_redex t2 t1)
  | _ → t
```

7.4 The subtyping algorithm of Bull

The subtyping algorithm is basically the same as the one described in Chapter 5. The only difference is that the types are normalized before applying the algorithm.

The functions rewriting the terms in normal forms are the following:

```
let rec anf a =
  let rec distr f a b =
    match (a,b) with
    | (Union(l,a1,a2),_) → Inter(l, distr f a1 b, distr f a2 b)
    | (_, Inter(l,b1,b2)) → Inter(l, distr f a b1, distr f a b2)
    | _ → f a b
  in
  match a with
  | Prod(l,id,a,b) → distr (fun a b → Prod(l,id,a,b)) (danf a) (canf b)
  | _ → a
and canf a =
  let rec distr a b =
    match (a,b) with
    | (Inter(l,a1,a2),_) → Inter(l, distr a1 b, distr a2 b)
    | (_,Inter(l,b1,b2)) → Inter(l, distr a b1, distr a b2)
    | _ → Union(dummy_loc,a,b)
  in
  match a with
  | Inter(l,a,b) → Inter(l, canf a, canf b)
  | Union(l,a,b) → distr (canf a) (canf b)
  | _ → anf a
and danf a =
  let rec distr a b =
    match (a,b) with
    | (Union(l,a1,a2),_) → Union(l, distr a1 b, distr a2 b)
    | (_,Union(l,b1,b2)) → Union(l, distr a b1, distr a b2)
    | _ → Inter(dummy_loc, a,b)
  in
  match a with
  | Inter(l,a,b) → distr (danf a) (danf b)
  | Union(l,a,b) → Union(l, danf a, danf b)
  | _ → anf a
```

The subtyping function is quite simple:

```
let is_subtype env ctx a b =
  let a = danf @@ strongly_normalize false env ctx a in
  let b = canf @@ strongly_normalize false env ctx b in
  let rec foo env ctx a b =
  match (a, b) with
  | (Union(_,a1,a2),_) → foo env ctx a1 b && foo env ctx a2 b
  | (_,Inter(_,b1,b2)) → foo env ctx a b1 && foo env ctx a b2
  | (Inter(_,a1,a2),_) → foo env ctx a1 b || foo env ctx a2 b
  | (_,Union(_,b1,b2)) → foo env ctx a b1 || foo env ctx a b2
  | (Prod(_,_,a1,a2),Prod(_,_,b1,b2))
    → foo env ctx b1 a1 && foo env (Env.add_var ctx (DefAxiom("",nothing))) a2 b2
  | _ → same_term a b
```

```
in foo env ctx a b
```

7.5 The unification algorithm of Bull

Higher-order unification of two terms Δ_1 and Δ_2 aims at finding a most general substitution for meta-variables such that Δ_1 and Δ_2 becomes convertible. Classical references are the work of Huet [59], and Dowek, Kirchner, and Hardin [41].

Our higher-order unification algorithm was inspired by the Reed [81] and Ziliani-Sozeau [97] papers. In [97], conversion of terms is quite involved because of the complexity of Coq. For simplicity, our algorithm supposes the terms to be in normal form.

The unification algorithm takes as input a meta-environment Φ, a global environment Σ, a local environment Γ, the two terms to unify Δ_1 and Δ_2, and either fails or returns the updated meta-environment Φ. The structural rules are given in Figure 7.1. The rest of the unification algorithm implements *Higher-Order Pattern Unification* (HOPU) [81]. In a nutshell, HOPU takes as an argument a unification problem $?f\,S \stackrel{?}{=} N$, where all the terms in S are free variables and each variable occurs once. For instance, for the unification problem $?f\,y\,x\,z \stackrel{?}{=} x\,c\,y$, it creates the solution $?f := \lambda y{:}\sigma_2.\lambda x{:}\sigma_1.\lambda z{:}\sigma_3.x\,c\,y$. The expected type of x, y, and z can be found in the local environment, but capturing correctly the free variables x, y, and z is quite tricky because we have to permute their de Bruijn indices. If HOPU does not work, we try to recursively unify every subterm.

7.6 The refinement algorithm of Bull

The Bull typechecker was inspired by the work on the Matita ITP [6]. It is defined using *bi-directionality*, in the style of Harper-Licata [53]. The bi-directional technique is a mix of typechecking and type reconstruction, in order to trigger the unification algorithm as soon as possible. Moreover, it gives more precise error messages than standard type reconstruction. For instance, if `f : (bool -> nat -> bool) -> bool`, then `f (fun x y ⇒ y)` is ill-typed. With a simple type inference algorithm, we would type `f`, then `fun x y ⇒ y` which would be given some type `?x -> ?y -> ?y`, and finally we would try to unify `bool -> nat -> bool` with `?x -> ?y -> ?y`, which fails. However, the failure is localized on the application, whereas it would better be localized inside the argument. More precisely, we would have the following error message:

```
f (fun x y ⇒ y)
                ^

Error: the term "y" has type "nat" while it is expected to have type "bool".
```

Our typechecker is also a *refiner*: intuitively, a refiner takes as input an incomplete term, and possibly an incomplete type, and tries to infer as much information as possible in order to reconstruct a well-typed term. For example, let's assume we have in the global environment the following constants:

```
(eq : nat -> nat -> Type), (eq_refl : forall x : nat, eq x x)
```

Then refining the term `eq_refl _ : eq _ 0` would create the following term:

```
eq_refl 0 : eq 0 0
```

Refinement also enable untyped abstractions: the refiner may recover the type of bound variables, because untyped abstractions are incomplete terms. The typechecking is done

$$\frac{s_1 \equiv s_2}{\Phi; \Sigma; \Gamma \vdash s_1 \stackrel{?}{=} s_2 \stackrel{\mathcal{U}}{\rightsquigarrow} \Phi}\ (Sort) \qquad \frac{c_1 \equiv c_2}{\Phi; \Sigma; \Gamma \vdash c_1 \stackrel{?}{=} c_2 \stackrel{\mathcal{U}}{\rightsquigarrow} \Phi}\ (Const) \qquad \frac{x_1 \equiv x_2}{\Phi; \Sigma; \Gamma \vdash x_1 \stackrel{?}{=} x_2 \stackrel{\mathcal{U}}{\rightsquigarrow} \Phi}\ (Var)$$

$$\frac{\Phi_1; \Sigma; \Gamma \vdash \sigma_1 \stackrel{?}{=} \sigma_2 \stackrel{\mathcal{U}}{\rightsquigarrow} \Phi_2 \quad \Phi_2; \Sigma; \Gamma, x{:}\sigma_1 \vdash \Delta_1 \stackrel{?}{=} \Delta_2 \stackrel{\mathcal{U}}{\rightsquigarrow} \Phi_3}{\Phi_1; \Sigma; \Gamma \vdash \lambda x{:}\sigma_1.\Delta_1 \stackrel{?}{=} \lambda x{:}\sigma_2.\Delta_2 \stackrel{\mathcal{U}}{\rightsquigarrow} \Phi_3}\ (Abs)$$

$$\frac{\Phi_1; \Sigma; \Gamma \vdash \sigma_1 \stackrel{?}{=} \sigma_2 \stackrel{\mathcal{U}}{\rightsquigarrow} \Phi_2 \quad \Phi_2; \Sigma; \Gamma \vdash \tau_1 \stackrel{?}{=} \tau_2 \stackrel{\mathcal{U}}{\rightsquigarrow} \Phi_3}{\Phi_1; \Sigma; \Gamma \vdash \sigma_1 \cap \tau_1 \stackrel{?}{=} \sigma_2 \cap \tau_2 \stackrel{\mathcal{U}}{\rightsquigarrow} \Phi_3}\ (\cap)$$

$$\frac{\Phi_1; \Sigma; \Gamma \vdash \sigma_1 \stackrel{?}{=} \sigma_2 \stackrel{\mathcal{U}}{\rightsquigarrow} \Phi_2 \quad \Phi_2; \Sigma; \Gamma \vdash \tau_1 \stackrel{?}{=} \tau_2 \stackrel{\mathcal{U}}{\rightsquigarrow} \Phi_3}{\Phi_1; \Sigma; \Gamma \vdash \sigma_1 \cup \tau_1 \stackrel{?}{=} \sigma_2 \cup \tau_2 \stackrel{\mathcal{U}}{\rightsquigarrow} \Phi_3}\ (\cup)$$

$$\frac{\Phi_1; \Sigma; \Gamma \vdash \Delta_1 \stackrel{?}{=} \Delta_2 \stackrel{\mathcal{U}}{\rightsquigarrow} \Phi_2 \quad \Phi_2; \Sigma; \Gamma \vdash \Delta_3 \stackrel{?}{=} \Delta_4 \stackrel{\mathcal{U}}{\rightsquigarrow} \Phi_3}{\Phi_1; \Sigma; \Gamma \vdash \langle \Delta_1, \Delta_3 \rangle \stackrel{?}{=} \langle \Delta_2, \Delta_4 \rangle \stackrel{\mathcal{U}}{\rightsquigarrow} \Phi_3}\ (Spair)$$

$$\frac{\Phi_1; \Sigma; \Gamma \vdash \sigma_1 \stackrel{?}{=} \sigma_2 \stackrel{\mathcal{U}}{\rightsquigarrow} \Phi_2}{\Phi_1; \Sigma; \Gamma \vdash \mathsf{pr}_i\, \Delta_1 \stackrel{?}{=} \mathsf{pr}_i\, \Delta_2 \stackrel{\mathcal{U}}{\rightsquigarrow} \Phi_2}\ (Proj)$$

$$\frac{\Phi_1; \Sigma; \Gamma \vdash \sigma_1 \stackrel{?}{=} \sigma_2 \stackrel{\mathcal{U}}{\rightsquigarrow} \Phi_2 \quad \Phi_2; \Sigma; \Gamma \vdash \Delta_1 \stackrel{?}{=} \Delta_2 \stackrel{\mathcal{U}}{\rightsquigarrow} \Phi_3}{\Phi_1; \Sigma; \Gamma \vdash \mathsf{in}_i\, \sigma_1 \Delta_1 \stackrel{?}{=} \mathsf{in}_i\, \sigma_2 \Delta_2 \stackrel{\mathcal{U}}{\rightsquigarrow} \Phi_3}\ (Inj)$$

$$\frac{\Phi_1; \Sigma; \Gamma \vdash \sigma_1 \stackrel{?}{=} \sigma_2 \stackrel{\mathcal{U}}{\rightsquigarrow} \Phi_2 \quad \Phi_2; \Sigma; \Gamma \vdash \Delta_1 \stackrel{?}{=} \Delta_2 \stackrel{\mathcal{U}}{\rightsquigarrow} \Phi_3}{\Phi_1; \Sigma; \Gamma \vdash \mathbf{coe}\, \sigma_1\, \Delta_1 \stackrel{?}{=} \mathbf{coe}\, \sigma_2\, \Delta_2 \stackrel{\mathcal{U}}{\rightsquigarrow} \Phi_3}\ (Coe)$$

$$\frac{\begin{array}{ll} \Phi_1; \Sigma; \Gamma \vdash \Delta \stackrel{?}{=} \Delta' \stackrel{\mathcal{U}}{\rightsquigarrow} \Phi_2 & \Phi_1; \Sigma; \Gamma \vdash \tau \stackrel{?}{=} \tau' \stackrel{\mathcal{U}}{\rightsquigarrow} \Phi_3 \\ \Phi_1; \Sigma; \Gamma \vdash \sigma_1 \stackrel{?}{=} \sigma_1' \stackrel{\mathcal{U}}{\rightsquigarrow} \Phi_4 & \Phi_1; \Sigma; \Gamma, x{:}\sigma_1 \vdash \Delta_1 \stackrel{?}{=} \Delta_1' \stackrel{\mathcal{U}}{\rightsquigarrow} \Phi_5 \\ \Phi_1; \Sigma; \Gamma \vdash \sigma_2 \stackrel{?}{=} \sigma_2' \stackrel{\mathcal{U}}{\rightsquigarrow} \Phi_6 & \Phi_1; \Sigma; \Gamma, x{:}\sigma_2 \vdash \Delta_2 \stackrel{?}{=} \Delta_2' \stackrel{\mathcal{U}}{\rightsquigarrow} \Phi_7 \end{array}}{\begin{array}{l} \Phi_1; \Sigma; \Gamma \vdash \mathbf{smatch}\ \Delta\ \mathbf{return}\ \tau\ \mathbf{with}\, [x{:}\sigma_1 \Rightarrow \Delta_1 \mid x{:}\sigma_2 \Rightarrow \Delta_2] \\ \stackrel{?}{=} \mathbf{smatch}\ \Delta'\ \mathbf{return}\ \tau'\ \mathbf{with}\, [x{:}\sigma_1' \Rightarrow \Delta_1 \mid x{:}\sigma_2' \Rightarrow \Delta_2'] \stackrel{\mathcal{U}}{\rightsquigarrow} \Phi_7 \end{array}}\ (Ssum)$$

$$\frac{\Phi_1; \Sigma; \Gamma, x{:}\sigma \vdash \Delta_1 \stackrel{?}{=} \Delta_2\, x \stackrel{\mathcal{U}}{\rightsquigarrow} \Phi_2 \quad \Delta_2 \text{ is not a } \lambda\text{-abstraction}}{\Phi_1; \Sigma; \Gamma \vdash \lambda x{:}\sigma.\Delta_1 \stackrel{?}{=} \Delta_2 \stackrel{\mathcal{U}}{\rightsquigarrow} \Phi_2}\ (\eta_l)$$

$$\frac{\Phi_1; \Sigma; \Gamma, x{:}\sigma \vdash \Delta_1\, x \stackrel{?}{=} \Delta_2 \stackrel{\mathcal{U}}{\rightsquigarrow} \Phi_2 \quad \Delta_1 \text{ is not a } \lambda\text{-abstraction}}{\Phi_1; \Sigma; \Gamma \vdash \Delta_1 \stackrel{?}{=} \lambda x{:}\sigma.\Delta_2 \stackrel{\mathcal{U}}{\rightsquigarrow} \Phi_2}\ (\eta_r)$$

Figure 7.1: Structural rules of the unification algorithm

in two steps: firstly the term is typechecked without caring about the essence, then we check the essence. The five typing judgment are defined as follows:

Definition 7.2 (Typing judgments)**.** We have five typing judgments, corresponding to five OCaml functions:

1. The function `reconstruct` takes as inputs a meta-environment Φ_1, a global environment Σ, a local environment Γ, and a term Δ_1. It either fails or fills the holes in Δ_1, which becomes Δ_2, and returns Δ_2 along with its type σ and the updated meta-environment Φ_2. The corresponding judgment is the following:

$$\Phi_1; \Sigma; \Gamma \vdash \Delta_1 \overset{\Uparrow}{\rightsquigarrow} \Delta_2 : \sigma; \Phi_2$$

   The rules are described in Figures 7.2 and 7.3;

2. The function `force_type` takes as inputs a meta-environment Φ_1, a global environment Σ, a local environment Γ, and a term σ_1. It either fails or fills the holes in σ_1, which becomes σ_2 while ensuring it is a type, *i.e.* its type is a sort s, and returns σ_2 along with s, and the updated meta-environment Φ_2. The corresponding judgment is the following:

$$\Phi_1; \Sigma; \Gamma \vdash \sigma_1 \overset{\mathcal{F}}{\rightsquigarrow} \sigma_2 : \tau; \Phi_2$$

   The rules are described in Figure 7.4. Intuitively, the function reconstruct the type τ of σ_1, then tries to unify τ with Type and Kind. If it can only do one unification, it keeps the successful one, if both unifications work, we choose unification with a sort meta-variable, so τ is convertible to a sort;

3. The function `reconstruct_with_type` takes as inputs a meta-environment Φ_1, a global environment Σ, a local environment Γ, a term Δ_1, and its expected type σ. It either fails or fills the holes in Δ_1, which becomes Δ_2 while ensuring its type is σ, and returns Δ_2 along the updated meta-environment Φ_2. The corresponding judgment is the following:

$$\Phi_1; \Sigma; \Gamma \vdash \Delta_1 : \sigma \overset{\Downarrow}{\rightsquigarrow} \Delta_2; \Phi_2$$

   The rules are described in Figure 7.5. There is a rule (*Default*) which applies only if none of the other rules work. The acute reader could remark two subtle things:

   (a) we chose not to add any inference rule for coercions, because we believe it would make error messages clearer: more precisely, if we want to check that $\mathbf{coe}\,\sigma\,\Delta$ has type τ, there could be two errors happening concurrently: it is possible that the type of Δ is not a subtype of σ, and at the same time σ is not unifiable with τ. We think that the error to be reported should be the first one, and in this case the (*Default*) rule is sufficient;

   (b) the management of de Bruijn indices for the (*Let*) is tricky: if we want to check that $\mathbf{let}\ x{:}\sigma := \Delta_1\ \mathbf{in}\ \Delta_2$ has type τ in some local context Γ, we recursively check that Δ_2 has type τ in the local context $\Gamma, x := \Delta_1' : \sigma'$ for some Δ_1', but the de Bruijn indices for τ correspond to the position of the local variables in the local context, which has been updated. We therefore have to increment all the de Bruijn indices in τ, in order to report the fact that there is one extra element in the local context;

4. The function `essence` takes as inputs a meta-environment Φ_1, a global environment Σ, an essence environment Ψ, and a term Δ. It either fails or construct its essence M, and returns M along with the updated meta-environment Φ_2. The corresponding judgment is the following:

$$\Phi_1; \Sigma; \Psi \vdash \Delta \overset{\mathcal{E}\Uparrow}{\rightsquigarrow} M; \Phi_2$$

$$\frac{}{\Phi; \Sigma; \Gamma \vdash \mathsf{Type} \overset{\Uparrow}{\rightsquigarrow} \mathsf{Type} : \mathsf{Kind}; \Phi} \; (T)$$

$$\frac{\begin{array}{l}\Phi_1; \Sigma; \Gamma \vdash \sigma \overset{\mathcal{F}}{\rightsquigarrow} \sigma'; \Phi_2 \\ \Phi_2; \Sigma; \Gamma \vdash \Delta : \sigma' \overset{\Downarrow}{\rightsquigarrow} \Delta'; \Phi_3 \quad \Phi_3; \Sigma; \Gamma, x := \Delta' : \sigma' \vdash \Delta_2 \overset{\Uparrow}{\rightsquigarrow} \Delta'_2 : \tau; \Phi_4\end{array}}{\Phi_1; \Sigma; \Gamma \vdash \mathbf{let}\ x{:}\sigma := \Delta_1\ \mathbf{in}\ \Delta_2 \overset{\Uparrow}{\rightsquigarrow} \mathbf{let}\ x{:}\sigma' := \Delta'_1\ \mathbf{in}\ \Delta'_2 : \tau[\sigma'/x]; \Phi_4} \; (Let)$$

$$\frac{\Phi_1; \Sigma; \Gamma \vdash \sigma_1 \overset{\mathcal{F}}{\rightsquigarrow} \sigma'_1 : s_1; \Phi_2 \quad \Phi_2; \Sigma; \Gamma \vdash \sigma_2 \overset{\mathcal{F}}{\rightsquigarrow} \sigma'_2 : s_2; \Phi_3 \quad \Phi_3 \vdash (s_1, s_2) \in \mathbf{LF}; \Phi_4}{\Phi_1; \Sigma; \Gamma \vdash \Pi x{:}\sigma_1.\sigma_2 \overset{\Uparrow}{\rightsquigarrow} \Pi x{:}\sigma'_1.\sigma'_2 : s_2; \Phi_4} \; (Prod)$$

$$\frac{\begin{array}{l}\Phi_1; \Sigma; \Gamma \vdash \sigma \overset{\mathcal{F}}{\rightsquigarrow} \sigma'; \Phi_2 \\ \Phi_2; \Sigma; \Gamma, x{:}\sigma' \vdash \Delta \overset{\Uparrow}{\rightsquigarrow} \Delta' : \tau; \Phi_3 \quad \Phi_3; \Sigma; \Gamma \vdash \Pi x{:}\sigma'.\tau \overset{\mathcal{F}}{\rightsquigarrow} \rho : s; \Phi_4\end{array}}{\Phi_1; \Sigma; \Gamma \vdash \lambda x{:}\sigma.\Delta \overset{\Uparrow}{\rightsquigarrow} \lambda x{:}\sigma'.\Delta' : \Pi x{:}\sigma'.\tau; \Phi_4} \; (Abs)$$

$$\frac{\Phi_1; \Sigma; \Gamma \vdash \Delta \overset{\Uparrow}{\rightsquigarrow} \Delta' : \sigma; \Phi_2}{\Phi_1; \Sigma; \Gamma \vdash \Delta\,() \overset{\Uparrow}{\rightsquigarrow} \Delta' : \sigma; \Phi_2} \; (App_1)$$

$$\frac{\begin{array}{l}\Phi_1; \Sigma; \Gamma \vdash \Delta_1\, S \overset{\Uparrow}{\rightsquigarrow} \Delta' : \sigma; \Phi_2 \\ \Phi_2; \Sigma; \Gamma \vdash \sigma =_\beta \Pi x{:}\sigma_1.\sigma_2 \qquad \Phi_2; \Sigma; \Gamma \vdash \Delta_2 : \sigma_1 \overset{\Downarrow}{\rightsquigarrow} \Delta'_2; \Phi_3\end{array}}{\Phi_1; \Sigma; \Gamma \vdash \Delta_1\,(S; \Delta_2) \overset{\Uparrow}{\rightsquigarrow} \Delta'_1\,\Delta'_2 : \sigma_2[\Delta'_2/x]; \Phi_3} \; (App_2)$$

$$\frac{\begin{array}{l}\Phi_1; \Sigma; \Gamma \vdash \Delta_2\, S \overset{\Uparrow}{\rightsquigarrow} \Delta' : \sigma; \Phi_2 \quad \Phi_2; \Sigma; \Gamma \vdash \Delta_2 \overset{\Uparrow}{\rightsquigarrow} \Delta'_2 : \sigma_1; \Phi_3 \\ \Phi_3, ?y, (\Gamma, x{:}\sigma_1 \vdash ?x : ?y[\,]); \Sigma; \Gamma \vdash \sigma \overset{?}{=} \Pi x{:}\sigma_1.?x[\widehat{\Gamma}; x] \overset{\mathcal{U}}{\rightsquigarrow} \Phi_4\end{array}}{\Phi_1; \Sigma; \Gamma \vdash \Delta_1\,(S; \Delta_2) \overset{\Uparrow}{\rightsquigarrow} \Delta'\,\Delta'_2 : ?x[\widehat{\Gamma}; x][\Delta'_2/x]; \Phi_4} \; (App_3)$$

Figure 7.2: Rules for $\overset{\Uparrow}{\rightsquigarrow}$ (1st part)

The rules are described in Figure 7.6;

5. The function `essence_with_hint` takes as inputs a meta-environment Φ_1, a global environment Σ, an essence environment Ψ, a term Δ, and its expected essence M. It either fails or succeeds by returning the updated meta-environment Φ_2. The corresponding judgment is the following:

$$\Phi_1; \Sigma; \Psi \vdash M@\Delta \overset{\mathcal{E}\Downarrow}{\rightsquigarrow} \Phi_2$$

The rules are described in Figure 7.7. There is a rule (*Default*) which applies only if none of the other rules work.

7.7 The Read-Eval-Print-Loop of Bull

The *Read-Eval-Print-Loop* (REPL) reads a command which is given by the parser as a list of atomic commands. For instance, if the user writes:

```
Axiom (a b : Type) (f : a -> b).
```

The parser creates the following list of three atomic commands:

1. the command asking `a` to be an axiom of type `Type`;
2. the command asking `b` to be an axiom of type `Type`;
3. the command asking `f` to be an axiom of type `a -> b`.

The REPL tries to process the whole list. If there is a single failure while processing the list of atomic commands, it backtracks so the whole commands fails without changing the environment.

These commands are similar to the vernacular Coq commands and are quite intuitive. Here is the list of the REPL commands, along with their description:

```
Help.                              show this list of commands
Load "file".                       for loading a script file
Axiom term : type.                 define a constant or an axiom
Definition name [: type] := term.  define a term
Print name.                        print the definition of name
Printall.                          print all the signature
                                   (axioms and definitions)
Compute name.                      normalize name and print the result
Quit.                              quit
```

7.8 Future work

The current version of Bull [87] (ver. 0.9, October 2019) is still a work-in-progress. We plan to implement the following features:

1. *Inductive types* are the most important feature to add, in order to have a really usable theorem prover. We plan to take inspiration from the works of Paulin-Mohring [73]. This should be reasonably feasible;

2. *Mixing subtyping and unification* is a difficult problem, especially with intersection and union types. The most extensive research which has been done in this domain is the work of Dudenhefner, Martens, and Rehof [42], where the authors study unification modulo subtyping with intersection types (but no union). It would be challenging to find a unification algorithm modulo subtyping for intersection and union types, but ideally it would allow us to do some implicit coercions. Take from example the code in Subsection 8.1.1, it would be interesting for the user to use implicit coercions in this way:

   ```
   Axiom (Neg Zero Pos T F : Type) (Test : Pos | Neg).
   Axiom Is_0 : (Neg -> F) & (Zero -> T) & (Pos -> F).
   Definition Is_0_Test : F := smatch Test with
                                 x => coe _ Is_0 x
                               , x => coe _ Is_0 x
                               end.
   ```

   The unification algorithm would then guess that the first wildcard should be replaced with `Pos -> F` and the second one should be replaced with `Neg -> F`, which does not seem feasible if the unification algorithm does not take subtyping into account;

3. *Relevant arrow*, as defined in Chapter 6, could be useful to add more expressivity to our system. Relevant implication allows for a natural introduction of subtyping, in that $A \supset_{\mathsf{r}} B$ morally means $A \leqslant B$. Relevant implication amounts to a notion of "proof-reuse". Combining the remarks in [8, 7], minimal relevant implication, strong intersection and strong union correspond respectively to the implication, conjunction and disjunction operators of Meyer and Routley's Minimal Relevant Logic B^+ [70]. This could lead to some implementation problem, because deciding β-equality for the essences in this extended system would be undecidable (see Figure 6.8);

4. A *Tactic language*, such as the one of Coq, should be useful. Currently, there is no such tactic language for Bull, conceiving such a language should be feasible even if it would be quite heavy.

$$\frac{\Phi_1;\Sigma;\Gamma \vdash \sigma_1 : \mathsf{Type} \overset{\Downarrow}{\rightsquigarrow} \sigma_1';\Phi_2 \quad \Phi_2;\Sigma;\Gamma \vdash \sigma_2 : \mathsf{Type} \overset{\Downarrow}{\rightsquigarrow} \sigma_2';\Phi_3}{\Phi_1;\Sigma;\Gamma \vdash \sigma_1 \cap \sigma_2 \overset{\Uparrow}{\rightsquigarrow} \sigma_1' \cap \sigma_2' : \mathsf{Type};\Phi_3} \ (\cap)$$

$$\frac{\Phi_1;\Sigma;\Gamma \vdash \sigma_1 : \mathsf{Type} \overset{\Downarrow}{\rightsquigarrow} \sigma_1';\Phi_2 \quad \Phi_2;\Sigma;\Gamma \vdash \sigma_2 : \mathsf{Type} \overset{\Downarrow}{\rightsquigarrow} \sigma_2';\Phi_3}{\Phi_1;\Sigma;\Gamma \vdash \sigma_1 \cup \sigma_2 \overset{\Uparrow}{\rightsquigarrow} \sigma_1' \cup \sigma_2' : \mathsf{Type};\Phi_3} \ (\cup)$$

$$\frac{\begin{array}{l}\Phi_1;\Sigma;\Gamma \vdash \Delta_1 \overset{\Uparrow}{\rightsquigarrow} \Delta_1' : \sigma_1;\Phi_2 \\ \Phi_2;\Sigma;\Gamma \vdash \Delta_2 \overset{\Uparrow}{\rightsquigarrow} \Delta_2' : \sigma_2;\Phi_3 \quad \Phi_3;\Sigma;\Gamma \vdash \sigma_1 \cap \sigma_2 : \mathsf{Type} \overset{\Downarrow}{\rightsquigarrow} \Phi_4\end{array}}{\Phi_1;\Sigma;\Gamma \vdash \langle \Delta_1, \Delta_2 \rangle \overset{\Uparrow}{\rightsquigarrow} \langle \Delta_1', \Delta_2' \rangle : \sigma_1 \cap \sigma_2;\Phi_4} \ (Spair)$$

$$\frac{\Phi_1;\Sigma;\Gamma \vdash \Delta \overset{\Uparrow}{\rightsquigarrow} \Delta' : \sigma;\Phi_2 \quad \Phi_1;\Sigma;\Gamma \vdash \sigma =_\beta \sigma_1 \cap \sigma_2}{\Phi_1;\Sigma;\Gamma \vdash \mathsf{pr}_i\,\Delta \overset{\Uparrow}{\rightsquigarrow} \mathsf{pr}_i\,\Delta' : \sigma_i;\Phi_2} \ (proj_1)$$

$$\frac{\begin{array}{l}\Phi_1;\Sigma;\Gamma \vdash \Delta \overset{\Uparrow}{\rightsquigarrow} \Delta' : \sigma;\Phi_2 \\ \Phi_2, (\Gamma \vdash ?x_1 : \mathsf{Type}), (\Gamma \vdash ?x_2 : \mathsf{Type});\Sigma;\Gamma \vdash \sigma \overset{?}{=} ?x_1[\widehat{\Gamma}] \cap ?x_2[\widehat{\Gamma}] \overset{\mathcal{U}}{\rightsquigarrow} \Phi_3\end{array}}{\Phi_1;\Sigma;\Gamma \vdash \mathsf{pr}_i\,\Delta \overset{\Uparrow}{\rightsquigarrow} \mathsf{pr}_i\,\Delta' : ?x_i[\widehat{\Gamma}];\Phi_3} \ (proj_2)$$

$$\frac{\begin{array}{ll}\Phi_1;\Sigma;\Gamma \vdash \Delta \overset{\Uparrow}{\rightsquigarrow} \Delta' : \sigma';\Phi_2 & \Phi_2;\Sigma;\Gamma \vdash \lambda x{:}\tau_1.\tau_2 : \Pi x{:}\sigma \to \mathsf{Type} \overset{\Downarrow}{\rightsquigarrow} \lambda x{:}\tau_1'.\tau_2';\Phi_3 \\ \Phi_3;\Sigma;\Gamma \vdash \sigma_1 : \mathsf{Type} \overset{\Downarrow}{\rightsquigarrow} \sigma_1';\Phi_4 & \Phi_4;\Sigma;\Gamma \vdash \sigma_2 : \mathsf{Type} \overset{\Downarrow}{\rightsquigarrow} \sigma_2';\Phi_5 \\ \Phi_5;\Sigma;\Gamma \vdash \sigma' \overset{?}{=} \sigma_1' \cup \sigma_2' \overset{\mathcal{U}}{\rightsquigarrow} \Phi_6 & \Phi_6;\Sigma;\Gamma, x{:}\sigma_1' \vdash \Delta_1 : \tau_2'[\mathsf{in}_1\ \sigma_2'\,x/x] \overset{\Downarrow}{\rightsquigarrow} \Delta_1';\Phi_7 \\ \Phi_7;\Sigma;\Gamma, x{:}\sigma_2' \vdash \Delta_2 : \tau_2'[\mathsf{in}_2\ \sigma_1'\,x/x] \overset{\Downarrow}{\rightsquigarrow} \Delta_2';\Phi_8 & \end{array}}{\begin{array}{c}\Phi_1;\Sigma;\Gamma \vdash \mathbf{smatch}\ \Delta\ \mathbf{return}\ \lambda x{:}\tau_1.\tau_2\ \mathbf{with}\,[x{:}\sigma_1 \Rightarrow \Delta_1 \mid x{:}\sigma_2 \Rightarrow \Delta_2] \overset{\Uparrow}{\rightsquigarrow} \\ \mathbf{smatch}\ \Delta'\ \mathbf{return}\ \tau'\ \mathbf{with}\,[x{:}\sigma_1' \Rightarrow \Delta_1' \mid x{:}\sigma_2' \Rightarrow \Delta_2'] : \tau_2'[\Delta'/x];\Phi_8\end{array}} \ (Ssum)$$

$$\frac{\Phi_1;\Sigma;\Gamma \vdash \sigma \overset{\mathcal{F}}{\rightsquigarrow} \sigma' : s;\Phi_2 \quad \Phi_2;\Sigma;\Gamma \vdash \Delta \overset{\Uparrow}{\rightsquigarrow} \Delta' : \tau;\Phi_3 \quad \Sigma;\Gamma \vdash \tau \leqslant \sigma'}{\Phi_1;\Sigma;\Gamma \vdash \mathbf{coe}\,\sigma\,\Delta \overset{\Uparrow}{\rightsquigarrow} \mathbf{coe}\,\sigma'\,\Delta' : \sigma';\Phi_3} \ (Coe)$$

$$\frac{(x{:}\sigma) \in \Gamma \text{ or } (x := \Delta : \sigma) \in \Gamma}{\Phi;\Sigma;\Gamma \vdash x \overset{\Uparrow}{\rightsquigarrow} x : \sigma;\Phi} \ (Var)$$

$$\frac{(c{:}\sigma) \in \Sigma \text{ or } (c := \Delta : \sigma) \in \Sigma}{\Phi;\Sigma;\Gamma \vdash c \overset{\Uparrow}{\rightsquigarrow} c : \sigma;\Phi} \ (Const)$$

$$\frac{}{\Phi;\Sigma;\Gamma \vdash _ \overset{\Uparrow}{\rightsquigarrow} ?x[\widehat{\Gamma}] : ?y[\widehat{\Gamma}];\Phi, ?z, (\Gamma \vdash ?y : ?z[\,]), (\Gamma \vdash ?x : ?y[\widehat{\Gamma}])} \ (Wildcard)$$

$$\frac{(\Gamma' \vdash ?x : \sigma) \in \Phi \text{ or } (\Gamma' \vdash ?x := \Delta : \sigma) \in \Phi \quad \Gamma' = \sigma_1, \ldots, \sigma_n \quad \begin{array}{c} i = 1 \ldots n \\ \Phi;\Sigma;\Gamma \vdash \Delta_i : \sigma_i \end{array}}{\Phi;\Sigma;\Gamma \vdash ?x[\Delta_1;\ldots;\Delta_n] \overset{\Uparrow}{\rightsquigarrow} ?x[\Delta_1;\ldots;\Delta_n] : \sigma[\overrightarrow{\Delta_i/\widehat{\Gamma'}}];\Phi} \ (Meta\text{-}Var)$$

Figure 7.3: Rules for $\overset{\Uparrow}{\rightsquigarrow}$ (2nd part)

$$\frac{\begin{array}{l}\Phi_1;\Sigma;\Gamma\vdash\sigma\overset{\Uparrow}{\rightsquigarrow}\sigma':\tau;\Phi_2 \quad \Phi_2;\Sigma;\Gamma\vdash\tau\overset{?}{=}\mathsf{Type}\overset{\mathcal{U}}{\rightsquigarrow}\Phi_3\\ \Phi_2;\Sigma;\Gamma\vdash\tau\overset{?}{=}\mathsf{Kind}\overset{\mathcal{U}}{\rightsquigarrow}\Phi_3' \quad \Phi_2,\mathbf{sort}(?x);\Sigma\vdash\tau\overset{?}{=}s\overset{\mathcal{U}}{\rightsquigarrow}\Phi_4\end{array}}{\Phi_1;\Sigma;\Gamma\vdash\sigma\overset{\mathcal{F}}{\rightsquigarrow}\sigma':\tau;\Phi_4}\ (\mathit{Force_1})$$

$$\frac{\Phi_1;\Sigma;\Gamma\vdash\sigma\overset{\Uparrow}{\rightsquigarrow}\sigma':\tau;\Phi_2 \quad \Phi_2;\Sigma;\Gamma\vdash\tau\overset{?}{=}\mathsf{Type}\overset{\mathcal{U}}{\rightsquigarrow}\Phi_3 \quad \Phi_2;\Sigma;\Gamma\not\vdash\tau\overset{?}{=}\mathsf{Kind}\overset{\mathcal{U}}{\rightsquigarrow}\Phi_3'}{\Phi_1;\Sigma;\Gamma\vdash\sigma\overset{\mathcal{F}}{\rightsquigarrow}\sigma':\tau;\Phi_3}\ (\mathit{Force_2})$$

$$\frac{\Phi_1;\Sigma;\Gamma\vdash\sigma\overset{\Uparrow}{\rightsquigarrow}\sigma':\tau;\Phi_2 \quad \Phi_2;\Sigma;\Gamma\not\vdash\tau\overset{?}{=}\mathsf{Type}\overset{\mathcal{U}}{\rightsquigarrow}\Phi_3 \quad \Phi_2;\Sigma;\Gamma\vdash\tau\overset{?}{=}\mathsf{Kind}\overset{\mathcal{U}}{\rightsquigarrow}\Phi_3'}{\Phi_1;\Sigma;\Gamma\vdash\sigma\overset{\mathcal{F}}{\rightsquigarrow}\sigma':\tau;\Phi_3'}\ (\mathit{Force_3})$$

Figure 7.4: Rules for $\overset{\mathcal{F}}{\rightsquigarrow}$

$$\frac{\Phi_1;\Sigma;\Gamma\vdash\Delta\overset{\Uparrow}{\rightsquigarrow}\Delta':\sigma;\Phi_2 \quad \Phi_2;\Sigma;\Gamma\vdash\sigma\overset{?}{=}\tau\overset{\mathcal{U}}{\rightsquigarrow}\Phi_3}{\Phi_1;\Sigma;\Gamma\vdash\Delta:\tau\overset{\Downarrow}{\rightsquigarrow}\Delta';\Phi_3}\ (\mathit{Default})$$

$$\frac{\begin{array}{l}\Phi_1;\Sigma;\Gamma\vdash\sigma\overset{\mathcal{F}}{\rightsquigarrow}\sigma':s;\Phi_2\\ \Phi_2;\Sigma;\Gamma\vdash\Delta_1:\sigma'\overset{\Downarrow}{\rightsquigarrow}\Delta_1';\Phi_3 \quad \Phi_3;\Sigma;\Gamma,x:=\Delta_1':\sigma'\vdash\Delta_2:\tau\overset{\Downarrow}{\rightsquigarrow}\Delta_2';\Phi_4\end{array}}{\Phi_1;\Sigma;\Gamma\vdash\mathbf{let}\ x{:}\sigma:=\Delta_1\ \mathbf{in}\ \Delta_2:\tau\overset{\Downarrow}{\rightsquigarrow}\mathbf{let}\ x{:}\sigma:=\Delta_1\ \mathbf{in}\ \Delta_2;\Phi_4}\ (\mathit{Let})$$

$$\frac{\begin{array}{l}\Phi_1;\Sigma;\Gamma\vdash\tau=_\beta\Pi x:\tau_1.\tau_2 \quad \Phi_1;\Sigma;\Gamma\vdash\sigma\overset{\mathcal{F}}{\rightsquigarrow}\sigma';\Phi_2\\ \Phi_2;\Sigma;\Gamma\vdash\sigma'\overset{?}{=}\tau_1\overset{\mathcal{U}}{\rightsquigarrow}\Phi_3 \quad \Phi_3;\Sigma;\Gamma,x{:}\sigma'\vdash\Delta:\tau_2\overset{\Downarrow}{\rightsquigarrow}\Delta';\Phi_4\end{array}}{\Phi_1;\Sigma;\Gamma\vdash\lambda x:\sigma.\Delta:\tau\overset{\Downarrow}{\rightsquigarrow}\lambda x:\sigma'.\Delta';\Phi_4}\ (\mathit{Abs})$$

$$\frac{\Phi_1;\Sigma;\Gamma\vdash\sigma=_\beta\sigma_1\cap\sigma_2 \quad \Phi_1;\Sigma;\Gamma\vdash\Delta_1:\sigma_1\overset{\Downarrow}{\rightsquigarrow}\Delta_1';\Phi_2 \quad \Phi_2;\Sigma;\Gamma\vdash\Delta_2:\sigma_2\overset{\Downarrow}{\rightsquigarrow}\Delta_2';\Phi_3}{\Phi_1;\Sigma;\Gamma\vdash\langle\Delta_1,\Delta_2\rangle:\sigma\overset{\Downarrow}{\rightsquigarrow}\langle\Delta_1',\Delta_2'\rangle;\Phi_3}\ (\mathit{Spair})$$

$$\frac{\Phi_1,(\Gamma\vdash ?x:\mathsf{Type});\Sigma;\Gamma\vdash\sigma\cap ?x:\mathsf{Type}\overset{\Downarrow}{\rightsquigarrow}\tau;\Phi_2 \quad \Phi_2;\Sigma;\Gamma\vdash\Delta:\sigma\cap ?x\overset{\Downarrow}{\rightsquigarrow}\Delta';\Phi_3}{\Phi_1;\Sigma;\Gamma\vdash\mathsf{pr}_1\,\Delta:\sigma\overset{\Downarrow}{\rightsquigarrow}\mathsf{pr}_1\,\Delta';\Phi_3}\ (\mathit{Proj_1})$$

$$\frac{\Phi_1,(\Gamma\vdash ?x:\mathsf{Type});\Sigma;\Gamma\vdash ?x\cap\sigma:\mathsf{Type}\overset{\Downarrow}{\rightsquigarrow}\tau;\Phi_2 \quad \Phi_2;\Sigma;\Gamma\vdash\Delta:?x\cap\sigma\overset{\Downarrow}{\rightsquigarrow}\Delta';\Phi_3}{\Phi_1;\Sigma;\Gamma\vdash\mathsf{pr}_2\,\Delta:\sigma\overset{\Downarrow}{\rightsquigarrow}\mathsf{pr}_2\,\Delta';\Phi_3}\ (\mathit{Proj_2})$$

$$\frac{\Phi_1;\Sigma;\Gamma\vdash\tau=_\beta\tau_1\cup\tau_2 \quad \Phi_1;\Sigma;\Gamma\vdash\sigma:\mathsf{Type}\overset{\Downarrow}{\rightsquigarrow}\sigma';\Phi_2 \quad \Phi_2;\Sigma;\Gamma\vdash\sigma'\overset{?}{=}\tau_i\overset{\mathcal{U}}{\rightsquigarrow}\Phi_3}{\Phi_1;\Sigma;\Gamma\vdash\mathsf{in}_i\,\sigma\Delta:\tau\overset{\Downarrow}{\rightsquigarrow}\mathsf{in}_i\,\sigma'\Delta';\Phi_3}\ (\mathit{Inj})$$

$$\frac{}{\Phi;\Sigma;\Gamma\vdash_:\sigma\overset{\Downarrow}{\rightsquigarrow}?x[\widehat{\Gamma}];\Phi,(\Gamma\vdash ?x:\sigma)}\ (\mathit{Wildcard})$$

Figure 7.5: Rules for $\overset{\Downarrow}{\rightsquigarrow}$

$$\frac{\Phi_1;\Sigma;\Psi \vdash \Delta_1 \overset{\mathcal{E}^{\Uparrow}}{\rightsquigarrow} M;\Phi_2 \quad \Phi_2;\Sigma;\Psi \vdash M@\Delta_2 \overset{\mathcal{E}^{\Downarrow}}{\rightsquigarrow} \Phi_3}{\Phi_1;\Sigma;\Psi \vdash \langle \Delta_1, \Delta_2 \rangle \overset{\mathcal{E}^{\Uparrow}}{\rightsquigarrow} M;\Phi_3} \ (Spair)$$

$$\frac{\Phi_1;\Sigma;\Psi \vdash \Delta \overset{\mathcal{E}^{\Uparrow}}{\rightsquigarrow} M;\Phi_2}{\Phi_1;\Sigma;\Psi \vdash \mathsf{pr}_i\,\Delta \overset{\mathcal{E}^{\Uparrow}}{\rightsquigarrow} M;\Phi_2} \ (Proj)$$

$$\frac{\begin{array}{ll} \Phi_1;\Sigma;\Psi \vdash \Delta \overset{\mathcal{E}^{\Uparrow}}{\rightsquigarrow} N;\Phi_2 & \Phi_2;\Sigma;\Psi \vdash \sigma \overset{\mathcal{E}^{\Uparrow}}{\rightsquigarrow} \varsigma;\Phi_3 \\ \Phi_3;\Sigma;\Psi \vdash \sigma_1 \overset{\mathcal{E}^{\Uparrow}}{\rightsquigarrow} \varsigma_1;\Phi_4 & \Phi_4;\Sigma;\Psi, x \vdash \Delta_1 \overset{\mathcal{E}^{\Uparrow}}{\rightsquigarrow} M\Phi_5 \\ \Phi_5;\Sigma;\Psi \vdash \sigma_2 \overset{\mathcal{E}^{\Uparrow}}{\rightsquigarrow} \varsigma_2;\Phi_6 & \Phi_6;\Sigma;\Psi, x \vdash M@\Delta_2 \overset{\mathcal{E}^{\Downarrow}}{\rightsquigarrow} \Phi_7 \end{array}}{\Phi_1;\Sigma;\Psi \vdash \textbf{smatch}\ \Delta\ \textbf{return}\ \sigma\ \textbf{with}\,[x{:}\sigma_1 \Rightarrow \Delta_1 \mid x{:}\sigma_2 \Rightarrow \Delta_2] \overset{\mathcal{E}^{\Uparrow}}{\rightsquigarrow} (\lambda x.M)\,N;\Phi_7} \ (Ssum)$$

$$\frac{\Phi_1;\Sigma;\Psi \vdash \Delta \overset{\mathcal{E}^{\Uparrow}}{\rightsquigarrow} M;\Phi_2}{\Phi_1;\Sigma;\Psi \vdash \mathsf{in}_i\,\sigma\ \Delta \overset{\mathcal{E}^{\Uparrow}}{\rightsquigarrow} M;\Phi_2} \ (Inj)$$

$$\frac{\Phi_1;\Sigma;\Psi \vdash \Delta \overset{\mathcal{E}^{\Uparrow}}{\rightsquigarrow} M;\Phi_2}{\Phi_1;\Sigma;\Psi \vdash \textbf{coe}\,\sigma\,\Delta \overset{\mathcal{E}^{\Uparrow}}{\rightsquigarrow} M;\Phi_2} \ (Coe)$$

$$\frac{\Phi_1;\Sigma;\Psi \vdash \sigma_1 \overset{\mathcal{E}^{\Uparrow}}{\rightsquigarrow} \varsigma_1;\Phi_2 \quad \Phi_2;\Sigma;\Psi, x \vdash \sigma_2 \overset{\mathcal{E}^{\Uparrow}}{\rightsquigarrow} \varsigma_2;\Phi_3}{\Phi_1;\Sigma;\Psi \vdash \Pi x{:}\sigma_1.\sigma_2 \overset{\mathcal{E}^{\Uparrow}}{\rightsquigarrow} \Pi x{:}\varsigma_1.\varsigma_2;\Phi_3} \ (Prod)$$

$$\frac{\Phi_1;\Sigma;\Psi \vdash \sigma \overset{\mathcal{E}^{\Uparrow}}{\rightsquigarrow} \varsigma;\Phi_2 \quad \Phi_2;\Sigma;\Psi, x \vdash \Delta \overset{\mathcal{E}^{\Uparrow}}{\rightsquigarrow} M;\Phi_3}{\Phi_1;\Sigma;\Psi \vdash \lambda x{:}\sigma.\Delta \overset{\mathcal{E}^{\Uparrow}}{\rightsquigarrow} \lambda x.M;\Phi_3} \ (Abs)$$

$$\frac{\Phi_1;\Sigma;\Psi \vdash \Delta \overset{\mathcal{E}^{\Uparrow}}{\rightsquigarrow} M;\Phi_2}{\Phi_1;\Sigma;\Psi \vdash \Delta\,() \overset{\mathcal{E}^{\Uparrow}}{\rightsquigarrow} M;\Phi_2} \ (App_1)$$

$$\frac{\Phi_1;\Sigma;\Psi \vdash \Delta_1\,S \overset{\mathcal{E}^{\Uparrow}}{\rightsquigarrow} M;\Phi_2 \quad \Phi_1;\Sigma;\Psi \vdash \Delta_2 \overset{\mathcal{E}^{\Uparrow}}{\rightsquigarrow} N;\Phi_3}{\Phi_1;\Sigma;\Psi \vdash \Delta_1\,(S;\Delta_2) \overset{\mathcal{E}^{\Uparrow}}{\rightsquigarrow} M\,N;\Phi_3} \ (App_2)$$

$$\frac{\Phi_1;\Sigma;\Psi \vdash \sigma_1 \overset{\mathcal{E}^{\Uparrow}}{\rightsquigarrow} \varsigma_1;\Phi_2 \quad \Phi_2;\Sigma;\Psi \vdash \sigma_2 \overset{\mathcal{E}^{\Uparrow}}{\rightsquigarrow} \varsigma_2;\Phi_3}{\Phi_1;\Sigma;\Psi \vdash \sigma_1 \cap \sigma_2 \overset{\mathcal{E}^{\Uparrow}}{\rightsquigarrow} \varsigma_1 \cap \varsigma_2;\Phi_3} \ (\cap)$$

$$\frac{\Phi_1;\Sigma;\Psi \vdash \sigma_1 \overset{\mathcal{E}^{\Uparrow}}{\rightsquigarrow} \varsigma_1;\Phi_2 \quad \Phi_2;\Sigma;\Psi \vdash \sigma_2 \overset{\mathcal{E}^{\Uparrow}}{\rightsquigarrow} \varsigma_2;\Phi_3}{\Phi_1;\Sigma;\Psi \vdash \sigma_1 \cup \sigma_2 \overset{\mathcal{E}^{\Uparrow}}{\rightsquigarrow} \varsigma_1 \cup \varsigma_2;\Phi_3} \ (\cup)$$

Figure 7.6: Rules for $\overset{\mathcal{E}^{\Uparrow}}{\rightsquigarrow}$

$$\frac{\Phi_1;\Sigma;\Psi \vdash \Delta \overset{\mathcal{E}\Uparrow}{\rightsquigarrow} M_2;\Phi_2 \quad \Phi_2;\Sigma;\Psi \vdash M_1 \overset{?}{=} M_2 \overset{\mathcal{U}}{\rightsquigarrow} \Phi_3}{\Phi_1;\Sigma;\Psi \vdash M_1@\Delta \overset{\mathcal{E}\Downarrow}{\rightsquigarrow} \Phi_3}\ (\mathit{Default})$$

$$\frac{\Phi_1;\Sigma;\Psi \vdash M@\Delta_1 \overset{\mathcal{E}\Downarrow}{\rightsquigarrow} \Phi_2 \quad \Phi_2;\Sigma;\Psi \vdash M@\Delta_1 \overset{\mathcal{E}\Downarrow}{\rightsquigarrow} \Phi_3}{\Phi_1;\Sigma;\Psi \vdash M@\langle\Delta_1,\Delta_2\rangle \overset{\mathcal{E}\Downarrow}{\rightsquigarrow} \Phi_3}\ (\mathit{Spair})$$

$$\frac{\Phi_1;\Sigma;\Psi \vdash M@\Delta \overset{\mathcal{E}\Downarrow}{\rightsquigarrow} \Phi_2}{\Phi_1;\Sigma;\Psi \vdash M@\mathsf{pr}_i\,\Delta \overset{\mathcal{E}\Downarrow}{\rightsquigarrow} \Phi_2}\ (\mathit{Proj})$$

$$\frac{\Phi_1;\Sigma;\Psi \vdash \sigma \overset{\mathcal{E}\Uparrow}{\rightsquigarrow} \varsigma;\Phi_2 \quad \Phi_2;\Sigma;\Psi \vdash M@\Delta \overset{\mathcal{E}\Downarrow}{\rightsquigarrow};\Phi_3}{\Phi_1;\Sigma;\Psi \vdash M@\mathsf{in}_i\,\sigma\,\Delta \overset{\mathcal{E}\Downarrow}{\rightsquigarrow} \Phi_3}\ (\mathit{Inj})$$

$$\frac{\Phi_1;\Sigma;\Psi \vdash \sigma \overset{\mathcal{E}\Uparrow}{\rightsquigarrow} \varsigma;\Phi_2 \quad \Phi_2;\Sigma;\Psi \vdash \Delta_1 \overset{\mathcal{E}\Uparrow}{\rightsquigarrow} M_1;\Phi_3 \quad \Phi_3;\Sigma;\Psi, x := M_1 \vdash M@\Delta_2 \overset{\mathcal{E}\Downarrow}{\rightsquigarrow} \Phi_4}{\Phi_1;\Sigma;\Psi \vdash M@\mathbf{let}\ x{:}\sigma := \Delta_1\ \mathbf{in}\ \Delta_2 \overset{\mathcal{E}\Downarrow}{\rightsquigarrow} \Phi_4}\ (\mathit{Let})$$

$$\frac{\Phi_1;\Sigma;\Psi \vdash M =_\beta \Pi x{:}\varsigma_1.\varsigma_2 \quad \Phi_1;\Sigma;\Psi \vdash \varsigma_1@\sigma_1 \overset{\mathcal{E}\Downarrow}{\rightsquigarrow} \Phi_2 \quad \Phi_2;\Sigma;\Psi, x \vdash \varsigma_2@\sigma_2 \overset{\mathcal{E}\Downarrow}{\rightsquigarrow} \Phi_3}{\Phi_1;\Sigma;\Psi \vdash M@\Pi x{:}\sigma_1.\sigma_2 \overset{\mathcal{E}\Downarrow}{\rightsquigarrow} \Phi_3}\ (\mathit{Prod})$$

$$\frac{\Phi_1;\Sigma;\Psi \vdash M_1 =_\beta \lambda x.M_2 \quad \Phi_1;\Sigma;\Psi, x \vdash M_2@\Delta \overset{\mathcal{E}\Downarrow}{\rightsquigarrow} \Phi_2}{\Phi_1;\Sigma;\Psi \vdash M_1@\lambda x{:}\sigma.\Delta \overset{\mathcal{E}\Downarrow}{\rightsquigarrow} \Phi_2}\ (\mathit{Abs})$$

$$\frac{\Phi_1;\Sigma;\Psi \vdash M =_\beta \varsigma_1 \cap \varsigma_2 \quad \Phi_1;\Sigma;\Psi \vdash \varsigma_1@\sigma_1 \overset{\mathcal{E}\Downarrow}{\rightsquigarrow} \Phi_2 \quad \Phi_2;\Sigma;\Psi \vdash \varsigma_2@\sigma_2 \overset{\mathcal{E}\Downarrow}{\rightsquigarrow} \Phi_3}{\Phi_1;\Sigma;\Psi \vdash M@\sigma_1 \cap \sigma_2 \overset{\mathcal{E}\Downarrow}{\rightsquigarrow} \Phi_3}\ (\cap)$$

$$\frac{\Phi_1;\Sigma;\Psi \vdash M =_\beta \varsigma_1 \cup \varsigma_2 \quad \Phi_1;\Sigma;\Psi \vdash \varsigma_1@\sigma_1 \overset{\mathcal{E}\Downarrow}{\rightsquigarrow} \Phi_2 \quad \Phi_2;\Sigma;\Psi \vdash \varsigma_2@\sigma_2 \overset{\mathcal{E}\Downarrow}{\rightsquigarrow} \Phi_3}{\Phi_1;\Sigma;\Psi \vdash M@\sigma_1 \cup \sigma_2 \overset{\mathcal{E}\Downarrow}{\rightsquigarrow} \Phi_3}\ (\cup)$$

Figure 7.7: Rules for $\overset{\mathcal{E}\Downarrow}{\rightsquigarrow}$

Chapter 8

Examples in Bull

The point of this chapter is to give examples which show a uniform and approach to the encoding of a plethora of type disciplines and systems which ultimately stem or can capitalize from strong proof-functional connectives and subtyping. The framework LF_Δ, presented in Chapter 6, is the first to accommodate all the examples and counterexamples that have appeared in the literature.

This chapter is organized as follows: in Section 8.1 we present some examples in LF_Δ along with their code in Bull, and in Section 8.2, we show some similar encodings done done in LF[1], in order to emphasize the benefits of LF_Δ.

8.1 Encodings in LF_Δ

We start by showing the expressive power of LF_Δ in encoding classical features of typing disciplines with strong intersection and union. For these examples, we set $\Sigma \stackrel{def}{=} \sigma{:}\mathsf{Type}, \tau{:}\mathsf{Type}$.

Auto application. The judgment $\vdash_{\lambda^{\mathrm{BDdL}}} \lambda x.x\,x : \sigma \cap (\sigma \to \tau) \to \tau$ in λ^{BDdL}, is rendered in LF_Δ by the LF_Δ judgment $\vdash_\Sigma \lambda x{:}\sigma \cap (\sigma \to \tau).(\mathsf{pr}_2\, x)\,(\mathsf{pr}_1\, x) : \sigma \cap (\sigma \to \tau) \to \tau$.

Polymorphic identity. The judgment $\vdash_{\lambda^{\mathrm{BDdL}}} \lambda x.x : (\sigma \to \sigma) \cap (\tau \to \tau)$ in λ^{BDdL}, is rendered in LF_Δ by the judgment $\vdash_\Sigma \langle \lambda x{:}\sigma.x, \lambda x{:}\tau.x\rangle : (\sigma \to \sigma) \cap (\tau \to \tau)$.

Commutativity of union. The judgment $\vdash_\Sigma \lambda x.x : (\sigma \cup \tau) \to (\tau \cup \sigma)$ in λ^{BDdL}, is rendered in LF_Δ by the judgment $\vdash_\Sigma \lambda x{:}\sigma\cup\tau.[\lambda y{:}\sigma.\mathsf{in}_2^\tau\, y, \lambda y{:}\tau.\mathsf{in}_1^\sigma\, y]\, x : (\sigma \cup \tau) \to (\tau \cup \sigma)$.

The Bull code corresponding to these examples is the following:

```
Axiom (s t : Type).
Definition auto_application (x : s & (s -> t)) := (proj_r x) (proj_l x).
Definition poly_id : (s -> s) & (t -> t) := let id1 x := x in
                                             let id2 x := x in < id1, id2 >.
Definition commut_union (x : s | t) := smatch x with
                                          x : s ⇒ inj_r t x
                                        , x : t ⇒ inj_l s x
                                        end.
```

[1]For convenience, we wrote and typechecked these examples in Coq, as LF is a sublanguage of the Calculus of Constructions.

8.1.1 Pierce's code

We recall the Pierce code from Figure 1.1:

$$\begin{array}{rcl} \text{Test} & \stackrel{def}{=} & \text{if } b \text{ then } 1 \text{ else } -1 : Pos \cup Neg \\ \text{Is_0} & : & (Neg \to F) \cap (Zero \to T) \cap (Pos \to F) \\ (\text{Is_0 Test}) & : & F \end{array}$$

The expressive power of union types highlighted by Pierce is rendered in LF_Δ by:

$$Neg : \mathsf{Type} \quad Zero : \mathsf{Type} \quad Pos : \mathsf{Type} \quad T : \mathsf{Type} \quad F : \mathsf{Type} \quad \text{Test} : Pos \cup Neg$$

$$\begin{array}{rcl} \text{Is_0} & : & (Neg \to F) \cap ((Zero \to T) \cap (Pos \to F)) \\ \text{Is_0_Test} & \stackrel{def}{=} & [\lambda x{:}Pos.(\mathsf{pr}_2\, \mathsf{pr}_2\, \text{Is_0})\, x, \lambda x{:}Neg.(\mathsf{pr}_1\, \text{Is_0})\, x]\, \text{Test} \end{array}$$

The Bull code corresponding to this example is the following:

```
Axiom (Neg Zero Pos T F : Type) (Test : Pos | Neg).
Axiom Is_0 : (Neg -> F) & (Zero -> T) & (Pos -> F).
Definition Is_0_Test := smatch Test with
                          x ⇒ coe (Pos -> F) Is_0 x
                        , x ⇒ coe (Neg -> F) Is_0 x
                        end.
```

As you can see, the code is quite short and readable, in contrast to the LF encoding of the same example found in Figure 8.2.

8.1.2 Hereditary Harrop formulæ

The encoding of Hereditary Harrop's Formulæ is one of the motivating examples given by Pfenning for introducing refinement types in [74]. In LF_Δ it can be expressed as in Figure 8.1 and type checked in Bull, without any reference to intersection types, by a subtle use of union types. We add also rules for solving and backchaining. Hereditary Harrop formulæ can be recursively defined using two mutually recursive syntactical objects called programs (π) and goals (γ):

$$\gamma \;:=\; \alpha \mid \gamma \wedge \gamma \mid \pi \Rightarrow \gamma \mid \gamma \vee \gamma \qquad\qquad \pi \;:=\; \alpha \mid \pi \wedge \pi \mid \gamma \Rightarrow \pi$$

Using Example 6.1, we can provide an alternative encoding of atoms, goals and programs which is more faithful to the one by Pfenning. Namely, we can introduce in the signature the constants $c_1 : \alpha \to^{\mathsf{r}} \gamma$ and $c_2 : \alpha \to^{\mathsf{r}} \pi$ in order to represent the axioms $atom \leqslant goal$ and $atom \leqslant prog$ in Pfenning's encoding. Our approach based on union types, while retaining the same expressivity permits to shortcut certain inclusions and to rule out also certain exotic goals and exotic programs. Indeed, for the purpose of establishing the adequacy of the encoding, it is sufficient to avoid variables involving union types in the derivation contexts.
The Bull code is the following:

```
(* three base types: atomic propositions, non-atomic goals and non-atomic programs *)
Axiom atom : Type.
Axiom non_atomic_goal : Type.
```

Atomic propositions, non-atomic goals and non-atomic programs: $\alpha, \gamma_0, \pi_0 : \mathsf{Type}$
Goals and programs: $\gamma = \alpha \cup \gamma_0 \qquad \pi = \alpha \cup \pi_0$
Constructors (implication, conjunction, disjunction).

$$
\begin{array}{lcl}
\mathsf{impl} & : & (\pi \to \gamma \to \gamma_0) \cap (\gamma \to \pi \to \pi_0) \\
\mathsf{impl}_1 & = & \lambda x{:}\pi.\lambda y{:}\gamma.\mathsf{in}_2^\alpha\,(\mathsf{pr}_1\,\mathsf{impl}\ x\ y) \qquad \mathsf{impl}_2 = \lambda x{:}\gamma.\lambda y{:}\pi.\mathsf{in}_2^\alpha\,(\mathsf{pr}_2\,\mathsf{impl}\ x\ y) \\
\mathsf{and} & : & (\gamma \to \gamma \to \gamma_0) \cap (\pi \to \pi \to \pi_0) \\
\mathsf{and}_1 & = & \lambda x{:}\gamma.\lambda y{:}\gamma.\mathsf{in}_2^\alpha\,(\mathsf{pr}_1\,\mathsf{and}\ x\ y) \qquad \mathsf{and}_2 = \lambda x{:}\pi.\lambda y{:}\pi.\mathsf{in}_2^\alpha\,(\mathsf{pr}_2\,\mathsf{and}\ x\ y) \\
\mathsf{or} & : & (\gamma \to \gamma \to \gamma_0) \qquad \mathsf{or}_1 = \lambda x{:}\gamma.\lambda y{:}\gamma.\mathsf{in}_2^\alpha\,(\mathsf{or}\ x\ y)
\end{array}
$$

$\mathsf{solve}\ p\ g$ indicates that the judgment $p \vdash g$ is valid.
$\mathsf{bchain}\ p\ a\ g$ indicates that, if $p \vdash g$ is valid, then $p \vdash a$ is valid.

$$\mathsf{solve} : \pi \to \gamma \to \mathsf{Type} \qquad \mathsf{bchain} : \pi \to \alpha \to \gamma \to \mathsf{Type}$$

Rules for solve:

$$
\begin{array}{lcl}
- & : & \Pi(p{:}\pi)(g_1, g_2{:}\gamma).\mathsf{solve}\ p\ g_1 \to \mathsf{solve}\ p\ g_2 \to \mathsf{solve}\ p\ (\mathsf{and}_1\ g_1\ g_2) \\
- & : & \Pi(p{:}\pi)(g_1, g_2{:}\gamma).\mathsf{solve}\ p\ g_1 \to \mathsf{solve}\ p\ (\mathsf{or}_1\ g_1\ g_2) \\
- & : & \Pi(p{:}\pi)(g_1, g_2{:}\gamma).\mathsf{solve}\ p\ g_2 \to \mathsf{solve}\ p\ (\mathsf{or}_1\ g_1\ g_2) \\
- & : & \Pi(p_1, p_2{:}\pi)(g{:}\gamma).\mathsf{solve}\ (\mathsf{and}_2\ p_1\ p_2)\ g \to \mathsf{solve}\ p_1\ (\mathsf{impl}_1\ p_2\ g) \\
- & : & \Pi(p{:}\pi)(a{:}\alpha)(g{:}\gamma).\mathsf{bchain}\ p\ a\ g \to \mathsf{solve}\ p\ g \to \mathsf{solve}\ p\ (\mathsf{in}_1^{\gamma_0}\ a)
\end{array}
$$

Rules for bchain:

$$
\begin{array}{lcl}
- & : & \Pi(a{:}\alpha)(g{:}\gamma).\mathsf{bchain}\ (\mathsf{impl}_2\ g\ (\mathsf{in}_1^{\pi_0}\ a))\ a\ g \\
- & : & \Pi(p_1, p_2{:}\pi)(a{:}\alpha)(g{:}\gamma).\mathsf{bchain}\ p_1\ a\ g \to \mathsf{bchain}\ (\mathsf{and}_2\ p_1\ p_2)\ a\ g \\
- & : & \Pi(p_1, p_2{:}\pi)(a{:}\alpha)(g{:}\gamma).\mathsf{bchain}\ p_2\ a\ g \to \mathsf{bchain}\ (\mathsf{and}_2\ p_1\ p_2)\ a\ g \\
- & : & \Pi(p{:}\pi)(a{:}\alpha)(g, g_1, g_2{:}\gamma).\mathsf{bchain} \\
 & & (\mathsf{impl}_2\ (\mathsf{and}_1\ g_1\ g_2)\ p)\ a\ g \to \mathsf{bchain}\ (\mathsf{impl}_2\ g_1\ (\mathsf{impl}_2\ g_2\ p))\ a\ g \\
- & : & \Pi(p_1, p_2{:}\pi)(a{:}\alpha)(g, g_1{:}\gamma).\mathsf{bchain}\ (\mathsf{impl}_2\ g_1\ p_1)\ a\ g \to \mathsf{bchain} \\
 & & (\mathsf{impl}_2\ g_1\ (\mathsf{and}_2\ p_1\ p_2))\ a\ g \\
- & : & \Pi(p_1, p_2{:}\pi)(a{:}\alpha)(g, g_1{:}\gamma).\mathsf{bchain} \\
 & & (\mathsf{impl}_2\ g_1\ p_2)\ a\ g \to \mathsf{bchain}\ (\mathsf{impl}_2\ g_1\ (\mathsf{and}_2\ p_1\ p_2))\ a\ g
\end{array}
$$

Figure 8.1: The LF_Δ encoding of Hereditary Harrop Formulæ

```
Axiom non_atomic_prog : Type.

(* goals and programs are defined from the base types *)
Definition goal := atom | non_atomic_goal.
Definition prog := atom | non_atomic_prog.

(* constructors (implication, conjunction, disjunction) *)
Axiom impl : (prog -> goal -> non_atomic_goal) & (goal -> prog -> non_atomic_prog).
Definition impl_1 p g := inj_r atom (proj_l impl p g).
Definition impl_2 g p := inj_r atom (proj_r impl g p).
Axiom and : (goal -> goal -> non_atomic_goal) & (prog -> prog -> non_atomic_prog).
Definition and_1 g1 g2 := inj_r atom (proj_l and g1 g2).
Definition and_2 p1 p2 := inj_r atom (proj_r and p1 p2).
Axiom or : (goal -> goal -> non_atomic_goal).
Definition or_1 g1 g2 := inj_r atom (or g1 g2).

(* solve p g means: the judgment p |- g is valid *)
Axiom solve : prog -> goal -> Type.

(* backchain p a g means: if p |- g is valid, then p |- a is valid *)
```

```
Axiom backchain : prog -> atom -> goal -> Type.

(* rules for solve *)
Axiom solve_and : forall p g1 g2, solve p g1 -> solve p g2 -> solve p (and_1 g1 g2).
Axiom solve_or1 : forall p g1 g2, solve p g1 -> solve p (or_1 g1 g2).
Axiom solve_or2 : forall p g1 g2, solve p g2 -> solve p (or_1 g1 g2).
Axiom solve_impl : forall p1 p2 g, solve (and_2 p1 p2) g -> solve p1 (impl_1 p2 g).
Axiom solve_atom : forall p a g, backchain p a g -> solve p g -> solve p (inj_l non_atomic_goal a).

(* rules for backchain *)
Axiom backchain_and1 :
 forall p1 p2 a g, backchain p1 a g -> backchain (and_2 p1 p2) a g.
Axiom backchain_and2 :
 forall p1 p2 a g, backchain p1 a g -> backchain (and_2 p1 p2) a g.
Axiom backchain_impl_atom :
 forall a g, backchain (impl_2 g (inj_l non_atomic_prog a)) a g.
Axiom backchain_impl_impl :
 forall p a g g1 g2, backchain (impl_2 (and_1 g1 g2) p) a g -> backchain (impl_2 g1 (impl_2 g2 p)) a g.
Axiom backchain_impl_and1 :
 forall p1 p2 a g g1, backchain (impl_2 g1 p1) a g -> backchain (impl_2 g1 (and_2 p1 p2)) a g.
Axiom backchain_impl_and2 :
 forall p1 p2 a g g1, backchain (impl_2 g1 p2) a g -> backchain (impl_2 g1 (and_2 p1 p2)) a g.
```

8.1.3 Natural deductions in normal form

The second motivating example for intersection types given in [74] is *natural deductions in normal form*. We recall that a natural deduction is in normal form if there are no applications of elimination rules of a logical connective immediately following their corresponding introduction, in the main branch of a subderivation.

$$
\begin{array}{rcl}
o & : & Type \qquad \supset : o \to o \to o \qquad Elim, \mathrm{Nf}^0 : o \to \mathsf{Type} \\
\mathrm{Nf} & \equiv & \lambda A{:}o.\mathrm{Nf}^0(A) \cup Elim(A) \\
\supset_I & : & \Pi A, B{:}o.(Elim(A) \to \mathrm{Nf}(B)) \to \mathrm{Nf}^0(A \supset B) \\
\supset_E & : & \Pi A, B{:}o.Elim(A \supset B) \to \mathrm{Nf}^0(A) \to Elim(B).
\end{array}
$$

The corresponding Bull code is the following:

```
Axiom (o : Type) (impl : o -> o -> o) (Elim Nf0 : o -> Type).
Definition Nf A := Nf0 A | Elim A.
Axiom impl_I : forall A B, (Elim A -> Nf B) -> (Nf0 (impl A B)).
Axiom impl_E : forall A B, Elim (impl A B) -> Nf0 A -> Elim B.
```

The encoding we give in LF_Δ is a slightly improved version of the one in [74]: as Pfenning, we restrict to the purely implicational fragment. As in the previous example, we use union types to define normal forms $\mathrm{Nf}(A)$ either as pure elimination-deductions from hypotheses $Elim(A)$ or normal form-deductions $\mathrm{Nf}^0(A)$. As above we could have used also intersection types. This example is interesting in itself, being the prototype of the encoding of type systems using canonical and atomic syntactic categories [53] and also of Fitch Set Theory [56].

Metacircular Encodings. This example uses an experimental implementation of relevant arrows in Bull. The following diagram summarizes the network of adequate encodings/inclusions between LF_Δ, LF, and λ^{BDdL} that can be defined:

$$\begin{tikzcd}
\mathrm{LF} \arrow[r, Rightarrow, "sh"] & \mathrm{LF}_\Delta \arrow[r, Rightarrow, "dp"] & \mathrm{LF} \\
\lambda^{\mathrm{BDdL}} \arrow[ur, Rightarrow, "sh"] \arrow[r, Rightarrow, "dp"] & \mathrm{LF} \arrow[u, hook] &
\end{tikzcd}$$

We denote by $\mathcal{S}_1 \Longrightarrow \mathcal{S}_2$ the encoding of system $\mathcal{S}_1$ in system $\mathcal{S}_2$, where the label *sh* (resp. *dp*), denotes a shallow (resp. deep) embedding. The notation $\mathcal{S}_1 \hookrightarrow \mathcal{S}_2$ denotes that $\mathcal{S}_2$ is an extension of $\mathcal{S}_1$.

With the intention of providing a better formal understanding of the semantics of strong intersection and union types in a logical framework, we provide in Section 8.2 a deep LF encoding of a presentation of λ^{BDdL} *à la* Church [39]. An encoding of λ^{BDdL} in LF_Δ can be mechanically type checked in the environment [88]. We even can add the relevant arrow. The corresponding Bull code is the following:

```
(* Object type *)
Axiom o : Type.

(* Type connectives *)
Axiom carrow : o -> o -> o.
Axiom cinter : o -> o -> o.
Axiom cunion : o -> o -> o.
Axiom crelev : o -> o -> o.

(* Transform the object types into real types *)
Axiom aOk : o -> Type.

(* Semantics *)
Axiom cabst   : forall s t, (aOk s -> aOk t) >> aOk (carrow s t).
Axiom capp    : forall s t, aOk (carrow s t) >> aOk s -> aOk t.
Axiom csabst  : forall s t,  (aOk s >> aOk t) >> aOk (crelev s t).
Axiom csapp   : forall s t, aOk (crelev s t) >> aOk s >> aOk t.
Axiom cpair   : forall s t, (aOk s & aOk t) >> aOk (cinter s t).
Axiom cpri    : forall s t, aOk (cinter s t) >> (aOk s & aOk t).
Axiom cini    : forall s t, (aOk s | aOk t) >> aOk (cunion s t).
Axiom ccopair : forall s t, aOk (cunion s t) >> (aOk s | aOk t).
```

Using this encoding, we can deeply encode self-application

$$\lambda x.x\,x : (\sigma \cap (\sigma \to \tau)) \to \tau$$

and commutativity of union

$$\lambda x.x : (\sigma \cup \tau) \to (\tau \cup \sigma)$$

```
Axiom s : o.
Axiom t : o.
Definition halfomega :=
 cabst (cinter s (carrow s t)) t (fun x => capp s t (proj_r (cpri s (carrow s t) x))
```

```
  (proj_l (cpri s (carrow s t) x))).
Definition idpair :=
 cpair (carrow s s) (carrow t t) <cabst s s (fun x ⇒ x), cabst t t (fun x ⇒ x)>.
Definition communion := cabst (cunion s t) (cunion t s)
 (fun x ⇒ smatch ccopair s t x with
            y ⇒ cini t s (inj_r (aOk t) y)
          , y ⇒ cini t s (inj_l (aOk s) y)
          end).
```

We also can show that the commutativity of union with a relevant arrow $\lambda x.x : (\sigma \cup \tau) \to^{\mathsf{r}} (\tau \cup \sigma)$:

```
Definition communion' := csabst (cunion s t) (cunion t s)
 (sfun x ⇒ smatch ccopair s t x with
            y ⇒ cini t s (inj_r (aOk t) y)
          , y ⇒ cini t s (inj_l (aOk s) y)
          end).
```

A shallow encoding of LF in LF_Δ making essential use of intersection types can be also type checked. The corresponding Bull code is the following:

```
Axiom obj' : Type.
Axiom fam' : Type.
Axiom knd' : Type.
Axiom sup' : Type.

Axiom same : obj' & fam' & knd' & sup'.
Axiom term : (obj' | fam' | knd' | sup') -> Type.
(* The obj, fam, knd, and sup types have the same essence (term same) *)
Definition obj := term (coe (obj' | fam' | knd' | sup') (coe obj' same)).
Definition fam := term (coe (obj' | fam' | knd' | sup') (coe fam' same)).
Definition knd := term (coe (obj' | fam' | knd' | sup') (coe knd' same)).
Definition sup := term (coe (obj' | fam' | knd' | sup') (coe sup' same)).

Axiom tp : knd & sup.
(* star and sqre have the same essence (tp) *)
Definition star := coe knd tp.
Definition sqre := coe sup tp.

Axiom lam : (fam -> (obj -> obj) -> obj) & (fam -> (obj -> fam) -> fam).
Definition lam_1 := coe (fam -> (obj -> obj) -> obj) lam.
Definition lam_2 := coe (fam -> (obj -> fam) -> fam) lam.

Axiom pi : (fam -> (obj -> fam) -> fam) & (fam -> (obj -> knd) -> knd).
Definition pi_1 := coe (fam -> (obj -> fam) -> fam) pi.
Definition pi_2 := coe (fam -> (obj -> knd) -> knd) pi.

Axiom app : (obj -> obj -> obj) & (fam -> obj -> fam).
Definition app_1 := coe (obj -> obj -> obj) app.
Definition app_2 := coe (fam -> obj -> fam) app.

Axiom of_1 : obj -> fam -> Type.
Axiom of_2 : fam -> knd -> Type.
Axiom of_3 : knd -> sup -> Type.
```

```
Axiom of_ax : of_3 star sqre.
(* Rules for lambda-abstraction are "essentially" the same *)
Definition of_lam1 := forall t1 t2 t3, of_2 t1 star ->
     (forall x, of_1 x t1 -> of_1 (t2 x) (t3 x)) -> of_1 (lam_1 t1 t2) (pi_1 t1 t3).
Definition of_lam2 := forall t1 t2 t3, of_2 t1 star ->
     (forall x, of_1 x t1 -> of_2 (t2 x) (t3 x)) -> of_2 (lam_2 t1 t2) (pi_2 t1 t3).
Axiom of_lam : of_lam1 & of_lam2.
(* Rules for product are ''essentially'' the same *)
Definition of_pi1 := forall t1 t2, of_2 t1 star ->
 (forall x, of_1 x t1 -> of_2 (t2 x) star) -> of_2 (pi_1 t1 t2) star.
Definition of_pi2 := forall t1 t2, of_2 t1 star ->
 (forall x, of_1 x t1 -> of_3 (t2 x) sqre) -> of_3 (pi_2 t1 t2) sqre.
Axiom of_pi : of_pi1 & of_pi2.
(* Rules for application are ''essentially'' the same *)
Definition of_app1 := forall t1 t2 t3 t4, of_1 t1 (pi_1 t3 t4) ->
 of_1 t2 t3 -> of_1 (app_1 t1 t2) (t4 t2).
Definition of_app2 := forall t1 t2 t3 t4, of_2 t1 (pi_2 t3 t4) ->
 of_1 t2 t3 -> of_2 (app_2 t1 t2) (t4 t2).
Axiom of_app : of_app1 & of_app2.
```

We finish this chapter by providing examples of encoding in LF.

8.2 Encodings in LF

We present a pure LF encoding of a presentation of λ^{BDdL} *à la* Church, using the Coq syntax, and the *Higher-Order Abstract Syntax* (HOAS) [75]. We use HOAS in order to take advantage of the higher-order features of the frameworks: other abstract syntax representation techniques would not be much different, but more verbose:

```
(* Define our types *)
Axiom o : Set.
(* Axiom omegatype : o. *)
Axioms (arrow inter union : o -> o -> o).

(* Transform our types into LF types *)
Axiom OK : o -> Set.

(* Define the essence equality as an equivalence relation *)
Axiom Eq : forall (s t : o), OK s -> OK t -> Prop.
Axiom Eqrefl : forall (s : o) (M : OK s), Eq s s M M.
Axiom Eqsymm : forall (s t : o) (M : OK s) (N : OK t), Eq s t M N -> Eq t s N M.
Axiom Eqtrans : forall (s t u : o) (M : OK s) (N : OK t) (O : OK u),
 Eq s t M N -> Eq t u N O -> Eq s u M O.

(* constructors for arrow (-> I and -> E) *)
Axiom Abst : forall (s t : o), ((OK s) -> (OK t)) -> OK (arrow s t).
Axiom App : forall (s t : o), OK (arrow s t) -> OK s -> OK t.

(* constructors for intersection *)
Axiom Proj_l : forall (s t : o), OK (inter s t) -> OK s.
Axiom Proj_r : forall (s t : o), OK (inter s t) -> OK t.
```

```
Axiom Pair : forall (s t : o) (M : OK s) (N : OK t), Eq s t M N -> OK (inter s t).

(* constructors for union *)
Axiom Inj_l : forall (s t : o), OK s -> OK (union s t).
Axiom Inj_r : forall (s t : o), OK t -> OK (union s t).
Axiom Sum : forall (s t u : o) (X : OK (arrow s u)) (Y : OK (arrow t u)),
 OK (union s t) -> Eq (arrow s u) (arrow t u) X Y -> OK u.

(* define equality wrt arrow constructors *)
Axiom Eqabst : forall (s t s' t' : o) (M : OK s -> OK t) (N : OK s' -> OK t'),
(forall (x : OK s) (y : OK s'), Eq s s' x y -> Eq t t' (M x) (N y)) ->
Eq (arrow s t) (arrow s' t') (Abst s t M) (Abst s' t' N).
Axiom Eqapp : forall (s t s' t' : o) (M : OK (arrow s t)) (N : OK s)
 (M' : OK (arrow s' t')) (N' : OK s'), Eq (arrow s t) (arrow s' t') M M' ->
 Eq s s' N N' -> Eq t t' (App s t M N) (App s' t' M' N').

(* define equality wrt intersection constructors *)
Axiom Eqpair : forall (s t : o) (M : OK s) (N : OK t) (pf : Eq s t M N),
 Eq (inter s t) s (Pair s t M N pf) M.
Axiom Eqproj_l : forall (s t : o) (M : OK (inter s t)),
 Eq (inter s t) s M (Proj_l s t M).
Axiom Eqproj_r : forall (s t : o) (M : OK (inter s t)),
 Eq (inter s t) t M (Proj_r s t M).

(* define equality wrt union *)
Axiom Eqinj_l : forall (s t : o) (M : OK s), Eq (union s t) s (Inj_l s t M) M.
Axiom Eqinj_r : forall (s t : o) (M : OK t), Eq (union s t) t (Inj_r s t M) M.
Axiom Eqsum : forall (s t u : o) (M : OK (arrow s u)) (N : OK (arrow t u))
 (O : OK (union s t)) (pf: Eq (arrow s u) (arrow t u) M N) (x : OK s),
 Eq s (union s t) x O -> Eq u u (App s u M x) (Sum s t u M N O pf).
```

The `Eq` predicate plays the same role of the essence function in LF_Δ, namely, it encodes the judgment that two proofs (*i.e.* two terms of type `(OK _)`) have the same structure. This is crucial in the `Pair` axiom (*i.e.* the introduction rule of the intersection type constructor) where we can inhabit the type `(inter s t)` only when the proofs of its component types `s` and `t` share the same structure (*i.e.* we have a witness of type `(Eq s t M N)`, where `M` has type `(OK s)` and `N` has type `(OK t)`). A similar role is played by the `Eq` premise in the `Sum` axiom (*i.e.* the elimination rule of the union type constructor). We have an `Eq` axiom for each proof rule.

Using this encoding, we can encode auto-application, polymorphic identity, and commutativity of union:

```
Section Examples.
  Hypotheses s t : o.

  (* lambda x. x x : (sigma inter (sigma -> tau)) -> tau *)
  Definition autoapp : OK (arrow (inter s (arrow s t)) t) :=
    Abst (inter s (arrow s t)) t (fun x : OK (inter s (arrow s t)) ⇒
     App s t (Proj_r s (arrow s t) x) (Proj_l s (arrow s t) x)).

  (* lambda x. x : (sigma -> sigma) inter (tau -> tau) *)
  Definition polyid : OK (inter (arrow s s) (arrow t t)) :=
```

```
    Pair (arrow s s) (arrow t t) (Abst s s (fun x : OK s ⇒ x))
     (Abst t t (fun x : OK t ⇒ x))
     (Eqabst s s t t (fun x : OK s ⇒ x) (fun x : OK t ⇒ x)
      (fun (x : OK s) (y : OK t) (Z : Eq s t x y) ⇒ Z)).

  (* lambda x. x : (sigma union tau) -> (tau union sigma) *)
  Definition commutunion : OK (arrow (union s t) (union t s)) :=
    Abst (union s t) (union t s)
         (fun x : OK (union s t) ⇒
            Copair s t (union t s) (Abst s (union t s) (fun y : OK s ⇒ Inj_r t s y))
                   (Abst t (union t s) (fun y : OK t ⇒ Inj_l t s y)) x
                   (Eqabst s (union t s) t (union t s) (fun y : OK s ⇒ Inj_r t s y)
                           (fun y : OK t ⇒ Inj_l t s y)
                           (fun (x0 : OK s) (y : OK t) (pf : Eq s t x0 y) ⇒
                              Eqtrans (union t s) s (union t s) (Inj_r t s x0) x0
                                      (Inj_l t s y)
                                      (Eqinj_r t s x0)
                                      (Eqtrans s t (union t s) x0 y (Inj_l t s y) pf
                                               (Eqsymm (union t s) t (Inj_l t s y) y
                                                       (Eqinj_l t s y))))))).
```

The definition of `commutunion` is quite unreadable, and has been created from the following Ltac script:

```
  Definition commutunion' : OK (arrow (union s t) (union t s)).
  Proof.
    apply (Abst (union s t) (union t s)).
    intro x.
    apply (Copair _ _ _ (Abst _ _ (fun y : _ ⇒ Inj_r _ _ y))
                  (Abst _ _ (fun y : _ ⇒ Inj_l _ _ y)) x).
    apply Eqabst.
    intros x0 y pf.
    assert (Eq _ _ (Inj_r t _ x0) x0) by apply Eqinj_r.
    assert (Eq _ _ y (Inj_l _ s y)). {
      apply Eqsymm.
      apply Eqinj_l.
    }
    eapply Eqtrans.
    now apply H.
    eapply Eqtrans.
    now apply pf.
    trivial.
  Defined.
End Examples.
```

Using the same encoding of LF_Δ in Coq, the Pierce's code from Figure 1.1 would be encoded as:

```
Section Test.
  Hypotheses (Pos Zero Neg T F : o).
  Hypotheses (Test : OK (union Pos Neg))
             (is_0 : OK (inter (arrow Neg F) (inter (arrow Zero T) (arrow Pos F)))).

  (* is_0 Test *)
```

```
Definition is0test : OK F.
  apply (Copair _ _ _ (Abst _ _ (fun x : _ ⇒ App _ _ (Proj_r _ _ (Proj_r _ _ is_0)) x))
                (Abst _ _ (fun x : _ ⇒ App _ _ (Proj_l _ _ is_0) x))).
  now apply Test.
  apply Eqabst.
  intros x y pf.
  apply Eqapp.
  - assert (H : Eq _ _ is_0 (Proj_r (arrow Neg F) (inter (arrow Zero T)
                                                          (arrow Pos F)) is_0))
      by apply Eqproj_r.
    assert (H0 : Eq _ _ (Proj_r (arrow Neg F) (inter (arrow Zero T)
                                                      (arrow Pos F)) is_0)
                    (Proj_r (arrow Zero T) (arrow Pos F)
                            (Proj_r (arrow Neg F) (inter (arrow Zero T)
                                                          (arrow Pos F)) is_0)))
      by apply Eqproj_r.
    assert (H1 : Eq _ _ is_0 (Proj_l (arrow Neg F) (inter (arrow Zero T)
                                                           (arrow Pos F)) is_0))
      by apply Eqproj_l.
    apply Eqsymm in H.
    apply Eqsymm in H0.
    eapply Eqtrans.
    apply H0.
    eapply Eqtrans.
    apply H.
    apply H1.
  - trivial.
Defined.

End Test.
```

The code of `is0test` has been generated by an Ltac script, the generated code is too huge to be humanly readable, as you can see in Figure 8.2.

```
Copair Pos Neg F
    (Abst Pos F
      (fun x : OK Pos ⇒
       App Pos F
         (Proj_r (arrow Zero T) (arrow Pos F)
            (Proj_r (arrow Neg F) (inter (arrow Zero T) (arrow Pos F)) is_0)) x))
    (Abst Neg F
       (fun x : OK Neg ⇒ App Neg F (Proj_l (arrow Neg F)
       (inter (arrow Zero T) (arrow Pos F)) is_0) x)) Test
    (Eqabst Pos F Neg F
      (fun x : OK Pos ⇒
       App Pos F
         (Proj_r (arrow Zero T) (arrow Pos F)
            (Proj_r (arrow Neg F) (inter (arrow Zero T) (arrow Pos F)) is_0)) x)
       (fun x : OK Neg ⇒ App Neg F (Proj_l (arrow Neg F)
       (inter (arrow Zero T) (arrow Pos F)) is_0) x)
      (fun (x : OK Pos) (y : OK Neg) (pf : Eq Pos Neg x y) ⇒
       Eqapp Pos F Neg F
         (Proj_r (arrow Zero T) (arrow Pos F)
            (Proj_r (arrow Neg F) (inter (arrow Zero T) (arrow Pos F)) is_0)) x
         (Proj_l (arrow Neg F) (inter (arrow Zero T) (arrow Pos F)) is_0) y
         (Eqtrans (arrow Pos F) (inter (arrow Zero T) (arrow Pos F)) (arrow Neg F)
            (Proj_r (arrow Zero T) (arrow Pos F)
               (Proj_r (arrow Neg F) (inter (arrow Zero T) (arrow Pos F)) is_0))
            (Proj_r (arrow Neg F) (inter (arrow Zero T) (arrow Pos F)) is_0)
            (Proj_l (arrow Neg F) (inter (arrow Zero T) (arrow Pos F)) is_0)
            (Eqsymm (inter (arrow Zero T) (arrow Pos F)) (arrow Pos F)
               (Proj_r (arrow Neg F) (inter (arrow Zero T) (arrow Pos F)) is_0)
               (Proj_r (arrow Zero T) (arrow Pos F)
                  (Proj_r (arrow Neg F) (inter (arrow Zero T) (arrow Pos F)) is_0))
               (Eqproj_r (arrow Zero T) (arrow Pos F)
                  (Proj_r (arrow Neg F) (inter (arrow Zero T) (arrow Pos F)) is_0)))
            (Eqtrans (inter (arrow Zero T) (arrow Pos F))
               (inter (arrow Neg F) (inter (arrow Zero T) (arrow Pos F))) (arrow Neg F)
               (Proj_r (arrow Neg F) (inter (arrow Zero T) (arrow Pos F)) is_0) is_0
               (Proj_l (arrow Neg F) (inter (arrow Zero T) (arrow Pos F)) is_0)
               (Eqsymm (inter (arrow Neg F) (inter (arrow Zero T) (arrow Pos F)))
                  (inter (arrow Zero T) (arrow Pos F)) is_0
                  (Proj_r (arrow Neg F) (inter (arrow Zero T) (arrow Pos F)) is_0)
                  (Eqproj_r (arrow Neg F) (inter (arrow Zero T) (arrow Pos F)) is_0))
               (Eqproj_l (arrow Neg F) (inter (arrow Zero T) (arrow Pos F)) is_0))) pf))
```

Figure 8.2: LF encoding of the Pierce's code (from Figure 1.1)

Bibliography

[1] Alexander Aiken. Introduction to set constraint-based program analysis. *Science of Computer Programming*, 35(2):79–111, 1999.

[2] Alexander Aiken and Edward L. Wimmers. Type inclusion constraints and type inference. In *Functional Programming languages and Computer Architecture (FPCA)*, pages 31–41. ACM, 1993.

[3] Fabio Alessi and Franco Barbanera. Strong conjunction and intersection types. In *Mathematical Foundations of Computer Science (MFCS)*, pages 64–73, 1991. URL: `http://dx.doi.org/10.1007/3-540-54345-7_49`, `doi:10.1007/3-540-54345-7_49`.

[4] Fabio Alessi, Franco Barbanera, and Mariangiola Dezani-Ciancaglini. Intersection types and lambda models. *Theoretical Computer Science*, 355(2):108–126, 2006.

[5] Andrea Asperti, Wilmer Ricciotti, Claudio Sacerdoti Coen, and Enrico Tassi. A compact kernel for the calculus of inductive constructions. *Sadhana*, 34(1):71–144, 2009.

[6] Andrea Asperti, Wilmer Ricciotti, Claudio Sacerdoti Coen, and Enrico Tassi. A bi-directional refinement algorithm for the calculus of (co)inductive constructions. *Logical Methods in Computer Science*, 8(1), 2012.

[7] Franco Barbanera, Mariangiola Dezani-Ciancaglini, and Ugo de'Liguoro. Intersection and union types: syntax and semantics. *Information and Computation*, 119(2):202–230, 1995. `doi:http://dx.doi.org/10.1006/inco.1995.1086`.

[8] Franco Barbanera and Simone Martini. Proof-functional connectives and realizability. *Archive for Mathematical Logic*, 33:189–211, 1994.

[9] Henk Barendregt. *The Lambda Calculus: Its Syntax and Semantics*, volume 103 of *Studies in Logic and the Foundations of Mathematics*. North-Holland, Amsterdam, revised edition, 1984.

[10] Henk Barendregt. Lambda-calculi with types. In *Handbook of Logic in Computer Science*, volume 2, pages 117–309. Oxford University Press, 1992.

[11] Henk Barendregt, Mario Coppo, and Mariangiola Dezani-Ciancaglini. A filter lambda model and the completeness of type assignment. *Journal of Symbolic Logic*, 48(4):931–940, 1983.

[12] Henk Barendregt, Wil Dekkers, and Richard Statman. *Lambda calculus with types*. Cambridge University Press, 2013.

[13] Choukri-Bey Ben-Yelles. *Type assignment in the lambda-calculus: syntax and semantics*. PhD thesis, University College of Swansea, 1979.

[14] Stefano Berardi. *Towards a mathematical analysis of the Coquand–Huet calculus of constructions and the other systems in Barendregt's cube*. PhD thesis, Dipartimento Matematica, Universita di Torino, 1988.

[15] Yves Bertot and Pierre Castéran. *Interactive theorem proving and program development: Coq'Art: the calculus of inductive constructions*. Springer Science & Business Media, 2013.

[16] Jan Bessai. Extracting a formally verified Subtyping Algorithm for Intersection Types from Ideals and Filters. Talk at TYPES, slides, 2016.

[17] Olivier Boite. Proof reuse with extended inductive types. In *Theorem Proving in Higher Order Logics (TPHOLs)*, pages 50–65, 2004.

[18] Viviana Bono, Betti Venneri, and Lorenzo Bettini. A typed lambda calculus with intersection types. *Theoretical Computer Science*, 398(1-3):95–113, 2008.

[19] George Boole. *The Mathematical Analysis of Logic, being an essay towards a calculus of deductive reasoning*. Cambridge: Macmillan, Barclay, & Macmillan; London: George Bell, 1847.

[20] Kim B. Bruce, Roberto Di Cosmo, and Giuseppe Longo. Provable isomorphisms of types. *Mathematical Structures in Computer Science*, 2(2):231–247, 1992.

[21] Antonio Bucciarelli, Adolfo Piperno, and Ivano Salvo. Intersection types and λ-definability. *Mathematical Structures in Computer Science*, 13(1):15–53, 2003.

[22] Beatrice Capitani, Michele Loreti, and Betti Venneri. Hyperformulae, Parallel Deductions and Intersection Types. *Electronic Notes in Theoretical Computer Science*, 50(2):180–198, 2001.

[23] Joshua E. Caplan and Mehdi T. Harandi. A logical framework for software proof reuse. In *Symposium on Software Reusability (SSR)*, pages 106–113, 1995.

[24] Luca Cardelli and Peter Wegner. On understanding types, data abstraction, and polymorphism. *ACM Computing Surveys*, 17(4):471–523, December 1985.

[25] Iliano Cervesato and Frank Pfenning. A linear spine calculus. *Journal of Logic and Computation*, 13(5):639–688, 2003.

[26] Adam Chlipala. *Certified Programming with Dependent Types – A Pragmatic Introduction to the Coq Proof Assistant*. MIT Press, 2013. URL: `http://adam.chlipala.net/cpdt/`.

[27] Mario Coppo and Mariangiola Dezani-Ciancaglini. An extension of the basic functionality theory for the λ-calculus. *Notre Dame Journal of Formal Logic*, 21(4):685–693, 1980.

[28] Mario Coppo, Mariangiola Dezani-Ciancaglini, and Patrick Sallé. Functional characterization of some semantic equalities inside λ-calculus. In *International Colloquium on Automata, Languages, and Programming (ICALP)*, pages 133–146. Springer-Verlag, 1979.

[29] Mario Coppo, Mariangiola Dezani-Ciancaglini, and Betti Venneri. Functional characters of solvable terms. *Zietschrift für Mathematische Logik und Grundlagen der Mathematik*, 27(2-6):45–58, 1981.

[30] Thierry Coquand and Gérard Huet. The calculus of constructions. *Information and Computation*, 76(2/3):95–120, 1988.

[31] Haskell B. Curry. Functionality in combinatory logic. *Proceedings of the National Academy of Sciences of the United States of America*, 20(11):584, 1934.

[32] Flemming Damm. Subtyping with union types, intersection types and recursive types. In *Theoretical and Applied Computer Science (TACS)*, pages 687–706, 1994.

[33] Rowan Davies. *Practical Refinement-Type Checking*. PhD thesis, Carnegie Mellon University, 2005. CMU–CS–05–110.

[34] N. G. de Bruijn. Automath, a language for mathematics. Technical Report T.H. Report 66-WSK-05, Department of Mathematics, Eindhoven University of Technology, November 1968.

[35] N. G. de Bruijn. Lambda calculus notation with nameless dummies, a tool for automatic formula manipulation, with application to the church-rosser theorem. In *Indagationes Mathematicae (Proceedings)*, volume 34(5), pages 381–392. Elsevier, 1972.

[36] The Coq development team, INRIA, CNRS, LIX, LRI, and PPS. The Coq Proof Assistant. `https://coq.inria.fr/`, 2019.

[37] Mariangiola Dezani-Ciancaglini, Silvia Ghilezan, and Betti Venneri. The "relevance" of intersection and union types. *Notre Dame Journal of Formal Logic*, 38(2):246–269, 1997.

[38] Roberto Di Cosmo. *Isomorphisms of types: from λ-calculus to information retrieval and language design*. Birkhauser, 1995.

[39] Daniel J. Dougherty, Ugo de'Liguoro, Luigi Liquori, and Claude Stolze. A realizability interpretation for intersection and union types. In *Asian Symposium on Programming Languages and Systems (APLAS)*, volume 10017 of *Lecture Notes in Computer Science*, pages 187–205. Springer-Verlag, 2016.

[40] Daniel J. Dougherty and Luigi Liquori. Logic and computation in a lambda calculus with intersection and union types. In *Logic Programming and Automated Reasoning (LPAR)*, volume 6355 of *Lecture Notes in Computer Science*, pages 173–191. Springer-Verlag, 2010.

[41] Gilles Dowek, Thérèse Hardin, and Claude Kirchner. Higher order unification via explicit substitutions. *Information and Computation*, 157(1-2):183–235, 2000.

[42] Andrej Dudenhefner, Moritz Martens, and Jakob Rehof. The intersection type unification problem. In *Formal Structures for Computation and Deduction (FSCD)*. Schloss Dagstuhl-Leibniz-Zentrum fuer Informatik, 2016.

[43] Andrej Dudenhefner, Moritz Martens, and Jakob Rehof. The algebraic intersection type unification problem. *Logical Methods in Computer Science*, 13(3), 2017.

[44] Joshua Dunfield. Refined typechecking with Stardust. In *Programming Languages meets Program Verification (PLPV)*, pages 21–32, 2007.

[45] Joshua Dunfield. Elaborating intersection and union types. *Journal of Functional Programming*, 24(2-3):133–165, 2014.

[46] Joshua Dunfield and Frank Pfenning. Tridirectional typechecking. In *Principles of Programming Languages (POPL)*, pages 281–292, 2004.

[47] Amy Felty and Douglas J. Howe. Generalization and reuse of tactic proofs. In *Logic Programming and Automated Reasoning (LPAR)*, pages 1–15, 1994.

[48] Timothy Freeman and Frank Pfenning. Refinement types for ML. In *Programming Language Design and Implementation (PLDI)*, pages 268–277, 1991.

[49] Alain Frisch. CDuce. `http://www.cduce.org/`, 2019.

[50] Alain Frisch, Giuseppe Castagna, and Véronique Benzaken. Semantic subtyping: Dealing set-theoretically with function, union, intersection, and negation types. *Journal of the ACM*, 55(4):19, 2008.

[51] Gerhard Gentzen. Untersuchungen über das logische Schließen I. *Mathematische Zeitschrift*, 39(1):176–210, 1935.

[52] Robert Harper, Furio Honsell, and Gordon Plotkin. A framework for defining logics. *Journal of the ACM*, 40(1):143–184, 1993. URL: `http://doi.acm.org/10.1145/138027.138060`, `doi:10.1145/138027.138060`.

[53] Robert Harper and Daniel R. Licata. Mechanizing metatheory in a logical framework. *Journal of Functional Programming*, 17(4–5):613–673, July 2007.

[54] J. Roger Hindley. The simple semantics for Coppo-Dezani-Sallé types. In *International Symposium on Programming*, pages 212–226, 1982.

[55] J. Roger Hindley. Coppo-Dezani types do not correspond to propositional logic. *Theoretical Computer Science*, 28:235–236, 1984.

[56] Furio Honsell, Marina Lenisa, Luigi Liquori, and Ivan Scagnetto. Implementing Cantor's paradise. In *Asian Symposium on Programming Languages and Systems (APLAS)*, pages 229–250, 2016.

[57] Furio Honsell, Luigi Liquori, Claude Stolze, and Ivan Scagnetto. The Delta-framework. In *Foundations of Software Technology and Theoretical Computer Science (FSTTCS)*, pages 37:1–37:21, 2018.

[58] William A. Howard. The Formulae-as-Types Notion of Construction. In *To H.B. Curry: Essays on Combinatory Logic, Lambda Calculus and Formalism*, pages 479–490. Academic press, 1980.

[59] Gérard Huet. A unification algorithm for typed lambda-calculus. *Theoretical Computer Science*, 1(1):27–57, 1975.

[60] Assaf Kfoury and Joe B. Wells. Principality and type inference for intersection types using expansion variables. *Theoretical Computer Science*, 311(1-3):1–70, 2004.

[61] Alexei Kopylov. Dependent intersection: a new way of defining records in type theory. In *Logic in Computer Science (LICS)*, pages 86–95, 2003.

[62] Luigi Liquori, Andreas Nuyts, and Claude Stolze. Private communications, 2017.

[63] Luigi Liquori and Simona Ronchi Della Rocca. Towards an intersection typed system *à la* Church. *Electronic Notes in Theoretical Computer Science*, 136:43–56, 2005.

[64] Luigi Liquori and Simona Ronchi Della Rocca. Intersection typed system *à la* Church. *Information and Computation*, 9(205):1371–1386, 2007.

[65] Luigi Liquori and Claude Stolze. A decidable subtyping logic for intersection and union types. In *Topics In Theoretical Computer Science (TTCS)*, volume 10608 of *Lecture Notes in Computer Science*, pages 74–90. Springer-Verlag, 2017.

[66] Luigi Liquori and Claude Stolze. The Delta-calculus: Syntax and types. In *Formal Structures for Computation and Deduction (FSCD)*, pages 28:1–28:20. Schloss Dagstuhl-Leibniz-Zentrum fuer Informatik, June 2019. URL: `https://hal.inria.fr/hal-01754707`.

[67] Edgar G. K. López-Escobar. Proof functional connectives. In *Methods in Mathematical Logic*, volume 1130 of *Lecture Notes in Mathematics*, pages 208–221. Springer-Verlag, 1985.

[68] David MacQueen, Gordon Plotkin, and Ravi Sethi. An ideal model for recursive polymorphic types. *Information and Control*, 71(1/2):95–130, 1986.

[69] Per Martin-Löf. *Intuitionistic type theory*, volume 1 of *Studies in proof theory*. Bibliopolis, 1984.

[70] Robert K. Meyer and Richard Routley. Algebraic analysis of entailment I. *Logique et Analyse*, 15:407–428, 1972.

[71] Grigori Mints. The completeness of provable realizability. *Notre Dame Journal of Formal Logic*, 30(3):420–441, 1989.

[72] Alexandre Miquel. The implicit calculus of constructions. In *Typed Lambda Calculi and Applications (TLCA)*, pages 344–359, 2001.

[73] Christine Paulin-Mohring. Inductive definitions in the system coq rules and properties. In *Typed Lambda Calculi and Applications (TLCA)*, pages 328–345. Springer, 1993.

[74] Frank Pfenning. Refinement types for logical frameworks. In *TYPES*, pages 285–299, 1993.

[75] Frank Pfenning and Conal Elliott. Higher-order abstract syntax. In *ACM SIGPLAN Notices*, volume 23(7), pages 199–208. ACM, 1988.

[76] Frank Pfenning and Christine Paulin-Mohring. Inductively defined types in the calculus of constructions. In *Mathematical Foundations of Programming Semantics (MFPS)*, pages 209–228. Springer, 1989.

[77] Benjamin C. Pierce. *Programming with intersection types, union types, and bounded polymorphism*. PhD thesis, Technical Report CMU-CS-91-205. Carnegie Mellon University, 1991.

[78] Benjamin C. Pierce. Programming with intersection types, union types, and polymorphism. Technical Report CMU–CS–91–106, Carnegie Mellon University, 1991.

[79] Elaine Pimentel, Simona Ronchi Della Rocca, and Luca Roversi. Intersection types from a proof-theoretic perspective. *Fundamenta Informaticae*, 121(1-4):253–274, 2012.

[80] Garrel Pottinger. A type assignment for the strongly normalizable λ-terms. In *To H.B. Curry: Essays on Combinatory Logic, Lambda Calculus and Formalism*, pages 561–577. Academic Press, 1980.

[81] Jason Reed. Higher-order constraint simplification in dependent type theory. In *Logical Frameworks and Meta-Languages: Theory and Practice (LFMTP)*, pages 49–56. ACM, 2009.

[82] Jakob Rehof and Pawel Urzyczyn. The complexity of inhabitation with explicit intersection. In *Logic and Program Semantics – Essays Dedicated to Dexter Kozen on the Occasion of His 60th Birthday*, pages 256–270, 2012.

[83] John C. Reynolds. Preliminary design of the programming language Forsythe. Report CMU–CS–88–159, Carnegie Mellon University, Pittsburgh, Pennsylvania, June 21 1988.

[84] Simona Ronchi Della Rocca and Luca Roversi. Intersection logic. In *Computer Science Logic (CSL)*, volume 2142 of *Lecture Notes in Computer Science*, pages 421–428. Springer-Verlag, 2001.

[85] Bertrand Russell. *The Principles of Mathematics*. Cambridge University Press, 1903.

[86] Vincent Siles and Hugo Herbelin. Equality is typable in semi-full pure type systems. In *Logic in Computer Science (LICS)*, pages 21–30, 2010.

[87] Claude Stolze. Bull. `https://github.com/cstolze/Bull`, 2019.

[88] Claude Stolze, Luigi Liquori, Furio Honsell, and Ivan Scagnetto. Towards a logical framework with intersection and union types. In *Logical Frameworks and Meta-languages: Theory and Practice (LFMTP)*, pages 1–9, 2017.

[89] Aaron Stump. From realizability to induction via dependent intersection. *Annals of Pure and Applied Logic*, 169(7):637–655, 2018.

[90] Masako Takahashi. Parallel reductions in λ-calculus. *Information and computation*, 118(1):120–127, 1995.

[91] Val Tannen, Thierry Coquand, Carl A. Gunter, and Andre Scedrov. Inheritance as implicit coercion. *Information and Computation*, 93(1):172–221, 1991.

[92] Pawel Urzyczyn. The emptiness problem for intersection types. *Journal of Symbolic Logic*, 64(3):1195–1215, 1999.

[93] Steffen van Bakel. Cut-elimination in the strict intersection type assignment system is strongly normalizing. *Notre Dame Journal of Formal Logic*, 45(1):35–63, 2004.

[94] Betti Venneri. Intersection types as logical formulae. *Journal of Logic and Computation*, 4(2):109–124, 1994.

[95] Joe B. Wells, Allyn Dimock, Robert Muller, and Franklyn Turbak. A calculus with polymorphic and polyvariant flow types. *Journal of Functional Programming*, 12(3):183–227, 2002.

[96] Joe B. Wells and Christian Haack. Branching types. In *European Symposium on Programming (ESOP)*, volume 2305 of *Lecture Notes in Computer Science*, pages 115–132. Springer-Verlag, 2002.

[97] Beta Ziliani and Matthieu Sozeau. A unification algorithm for Coq featuring universe polymorphism and overloading. In *ACM SIGPLAN Notices*, volume 50(9), pages 179–191. ACM, 2015.