The Δ-framework*

Furio Honsell
University of Udine
Udine, Italy
furio.honsell@uniud.it

Luigi Liquori
Université Côte d'Azur, INRIA
Sophia Antipolis, France
Luigi.Liquori@inria.fr

Ivan Scagnetto
University of Udine
Udine, Italy
ivan.scagnetto@uniud.it

Claude Stolze
Université Côte d'Azur, INRIA
Sophia Antipolis, France
Claude.Stolze@inria.fr

ABSTRACT

We introduce a dependent-type theory Δ-framework, LF_Δ, based on the Edinburgh Logical Framework LF, extended with the *strong proof-functional connectives* intersection, union, and relevant implication. Proof-functional connectives take into account the shape of logical proofs, thus allowing the user to reflect polymorphic features of proofs in formulæ. This is in contrast to classical and intuitionistic connectives where the meaning of a compound formula is dependent only on the *truth value* or the *provability* of its subformulæ. Both Logical Frameworks and Proof Functional Logics consider proofs as first class citizens albeit differently. The former mention proofs explicitly, while the latter mention proofs implicitly. Their combination therefore opens up new possibilites of formal reasoning on proof-theoretic semantics. We study the metatheory of LF_Δ and provide various examples of applications. Moreover, we discuss a prototype implementation of a type checker and a refiner allowing the user to accelerate and possibly automate, the proof development process. This proof-functional type theory can be plugged in existing common proof assistants.

ACM Reference Format:
Furio Honsell, Luigi Liquori, Ivan Scagnetto, and Claude Stolze. 2018. The Δ-framework. In *Proceedings of ACM Conference (Conference'17).* ACM, New York, NY, USA, 10 pages. https://doi.org/10.1145/nnnnnnn.nnnnnnn

1 INTRODUCTION

We extend the Edinburgh Logical Framework (LF) [13] with the *proof-functional* logical connectives of intersection, union, and minimal relevant implication of Proof Functional Logics [3, 4, 27]. We call this extension the Δ-framework (LF_Δ)[1], since it builds on the Δ-calculus introduced in [9, 18].

Proof functional connectives 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 intuitionsitic connectives where the meaning of a compound formula is only dependent on the *truth value* or the *provability* of its subformulæ.

Quite remarkably, both Logical Frameworks and Proof Functional Logics consider proofs as first class citizens albeit differently. The former mention proofs explicitly, while the latter mention proofs implicitly. This calls for a natural combination of the two, which should enhance the expressiveness of Logical Frameworks based on Type Theory. Hence new possibilities open up to formal reasoning on proof-theoretic semantics, beneficial to existing interactive theorem provers and dependently typed programming languages.

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. Proof-functional connectives, or simply *strong connectives*, need to consider the very structure of proofs and thus they need to give a first-class status to the latter, at least implicitly. In order to capture successfully strong logical connectives such as ∩ or ∪ we need to be able to express the rules:

$$\frac{\mathcal{D}_1 : A \quad \mathcal{D}_2 : B \quad \mathcal{D}_1 \equiv \mathcal{D}_2}{A \cap B} \ (\cap I)$$

$$\frac{\mathcal{D}_1 : A \supset C \quad \mathcal{D}_2 : B \supset C \quad A \cup B \quad \mathcal{D}_1 \equiv \mathcal{D}_2}{C} \ (\cup E)$$

where ≡ 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, [27], was the first to study the strong connective ∩. He contrasted it to the intuitionistic connective ∧ as follows: *"The intuitive meaning of ∩ 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 strong conjunction is $(A \supset A) \wedge (A \supset B \supset A)$. Clearly, $(A \supset A) \cap (A \supset B \supset A)$ does not hold otherwise there should exist a closed λ term having simultaneously only one and at least two abstractions. Lopez-Escobar [20] and Mints [23] investigated extensively logics featuring both strong and

*Work supported by the COST Action CA15123 EUTYPES "The European research network on types for programming and verification".

[1] Δ stands for "type Decorated term", although in Greek we should have rather called it the *K*-framework.

Conference'17, July 2017, Washington, DC, USA

ACM ISBN 978-x-xxxx-xxxx-x/YY/MM.
https://doi.org/10.1145/nnnnnnn.nnnnnnn

$$\frac{x{:}\sigma \in B}{B \vdash x : \sigma}\ (Var) \qquad \frac{B \vdash M : \sigma \to \tau \quad B \vdash N : \sigma}{B \vdash M\,N : \tau}\ (App) \qquad \frac{B \vdash M : \sigma \quad \sigma \leq \tau}{B \vdash M : \tau}\ (Sub)$$

$$\frac{B, x{:}\sigma \vdash M : \tau}{B \vdash \lambda x.M : \sigma \to \tau}\ (Abs) \qquad \frac{B \vdash M : \sigma \quad B \vdash M : \tau}{B \vdash M : \sigma \cap \tau}\ (\cap I)$$

$$\frac{B \vdash M : \sigma \cap \tau}{B \vdash M : \sigma}\ (\cap E_l) \qquad \frac{B \vdash M : \sigma \cap \tau}{B \vdash M : \tau}\ (\cap E_r)$$

$$\frac{B \vdash M : \sigma}{B \vdash M : \sigma \cup \tau}\ (\cup I_l) \qquad \frac{B \vdash M : \tau}{B \vdash M : \sigma \cup \tau}\ (\cup I_r)$$

$$\frac{B, x{:}\sigma \vdash M : \rho \quad B, x{:}\tau \vdash M : \rho \quad B \vdash N : \sigma \cup \tau}{B \vdash M[N/x] : \rho}\ (\cup E)$$

(1)	$\sigma \leqslant \sigma \cap \sigma$		(8)	$\sigma_1 \leqslant \sigma_2, \tau_1 \leqslant \tau_2 \Rightarrow \sigma_1 \cup \tau_1 \leqslant \sigma_2 \cup \tau_2$
(2)	$\sigma \cup \sigma \leqslant \sigma$		(9)	$\sigma \leqslant \tau, \tau \leqslant \rho \Rightarrow \sigma \leqslant \rho$
(3)	$\sigma \cap \tau \leqslant \sigma, \sigma \cap \tau \leqslant \tau$		(10)	$\sigma \cap (\tau \cup \rho) \leqslant (\sigma \cap \tau) \cup (\sigma \cap \rho)$
(4)	$\sigma \leqslant \sigma \cup \tau, \tau \leqslant \sigma \cup \tau$		(11)	$(\sigma \to \tau) \cap (\sigma \to \rho) \leqslant \sigma \to (\tau \cap \rho)$
(5)	$\sigma \leqslant \omega$		(12)	$(\sigma \to \rho) \cap (\tau \to \rho) \leqslant (\sigma \cup \tau) \to \rho$
(6)	$\sigma \leqslant \sigma$		(13)	$\omega \leqslant \omega \to \omega$
(7)	$\sigma_1 \leqslant \sigma_2, \tau_1 \leqslant \tau_2 \Rightarrow$	$\sigma_1 \cap \tau_1 \leqslant \sigma_2 \cap \tau_2$	(14)	$\sigma_2 \leqslant \sigma_1, \tau_1 \leqslant \tau_2 \Rightarrow$ $\sigma_1 \to \tau_1 \leqslant \sigma_2 \to \tau_2$

Figure 1: The type assignment of [3] and the subtype theory Ξ.

intuitionsitic connectives especially in the context of *realizability* interpretations.

Dually, it is in the ∪ 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 intuitionitic theorem $((A \supset B) \vee B) \supset A \supset B$ is not derivable if ∨ is replaced by ∪. Otherwise there would exist a term which behaves both as **I** and as **K**.

Strong (or *Minimal Relevant*) *Implication* $\to_r$ is yet another proof-functional connective. As explained in [4], 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. The operators $\supset$ and $\to_r$ differ, since $A \to_r B \to_r A$ is not derivable. Relevant implication allows for a natural introduction of *subtyping*, in that $A \supset_r B$ morally means $A \leqslant B$. Relevant implication amounts to a notion of "proof-reuse". Combining the remarks in [3, 4], 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^+ [22].

Strong connectives arise naturally in investigating the propositions-as-types analogy for *intersection* and *union type assignment systems*. *Intersection types* were introduced by Coppo, Dezani and Barendregt in the early 80's [6] to support a form of *ad hoc* polymorphism, for untyped λ-calculi, *à la* Curry. Intersection types were used originally as an (undecidable) type assignment system for untyped λ-calculi, *i.e.* for finitary descriptions of denotational semantics [8]. This line of research was later expanded by Abramsky [1] in a full-fledged Stone duality. *Union types* were introduced later, semantically, by MacQueen, Plotkin, and Sethi, [3, 21], again *à la* Curry. In [3] strong intersection, union and subtyping were thoroughly studied in the context of type-assignment systems, see Fig. 1.

Intersection and union type disciplines started to be investigated in typed settings,*i.e. à la* Church, much later, in the context of the programming language Forsythe, by Reynolds and Pierce [26, 28]. Other solutions were proposed by Castagna [12], Wells [31] and Dunfield [11]. A classical example of the expressiveness of such type disciplines, due to Pierce, is the following:

$$\begin{array}{rcl} \text{Test} & \triangleq & \text{if } b \text{ then } 1 \text{ else } -1 : \mathit{Pos} \cup \mathit{Neg} \\ \text{Is\_0} & : & (\mathit{Neg} \to F) \cap (\mathit{Zero} \to T) \cap (\mathit{Pos} \to F) \\ \text{(Is\_0 Test)} & : & F \end{array}$$

Without union types the best information we can get for (Is_0 Test) is a Boolean type.

Designing a λ-calculus *à la* Curry featuring intersection and union types is problematic. The usual approach of simply adding types to binders does not work, as the following example shows:

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

Liquori et al. were the first, [10, 18], to extend the theory of type assignment systems, *i.e.* type disciplines *à la* Curry, to explicitly typed calculi, *i.e.* type disciplines *à la* Church, for all strong connectives. In [9], two of the present authors proposed the Δ-calculus as a typed λ-calculus *à la* Church corresponding to the type assignment system *à la* Curry with intersection, union and subtyping, but without ω. The relation between Church-style and Curry-style λ-calculi was expressed using an *essence* function, denoted by $\wr - \wr$, that intuitively erases all the type information in terms (the full definition is shown in Figure 5). The Δ-calculus is a typed calculus corresponding to the type assignment system of [3], without ω and subtyping. Throughout the paper we will call this system $\mathcal{B}$.

Pfenning's work on Refinement Types [25] pioneered an extension of the Edinburgh Logical Framework with subtyping and intersection types. The logical strength of this system however, did not go beyond LF.

In this paper, building on [9], we introduce the Δ-framework LF_Δ, which extends LF and features a full-fledged strong conjunction, strong disjunction, and minimal relevant implication, whereby all strongly normalizing Δ-terms have a computational counterpart.

Miquel [24] 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. In fact not all terms typed by intersection types have an equivalent, for instance $\lambda x.x : ((\sigma \cap \tau) \to \sigma) \cap (\rho \to \rho))$ appears to be problematic [17].

Of course we could have carried out an encoding of the system $\mathcal{B}$ in LF. But due to the side-conditions characterizing proof-functional connectives, however, this can be achieved only through a deep encoding (see Fig. 8). 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.

In this paper we outline the implementation of an experimental proof development environment for LF_Δ and discuss experiments with it. It allows, of course, an agile and shallow encoding of proof-functional connectives in presence of dependent-types.

The main contributions of the present paper are the definition and the metatheory of LF_Δ, together with a discussion of the main design decisions. We provide also some motivating examples and outline the details of the implementation. Throughout the paper we assume the reader familiar with [3, 5].

2 THE Δ-FRAMEWORK: LF WITH PROOF-FUNCTIONAL OPERATORS

We could formulate LF_Δ in the style of [14], using only canonical forms and without reductions, but we use the standard LF format to better support the intuition. The syntax of LF_Δ-pseudo-terms is given by the following grammar:

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
			$\Pi^{\mathsf{r}} x{:}\sigma.\tau \mid$	relevant family
			$\sigma \cdot \Delta \mid$	relevant dependency
			$\sigma \cap \tau \mid$	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
			$\Delta \cdot \Delta \mid$	relevant application
			$\langle \Delta\ ,\ \Delta \rangle \mid$	intersection objects
			$[\Delta\ ,\ \Delta] \mid$	union objects
			$pr_l\ \Delta \mid pr_r\ \Delta \mid$	projections objects
			$in_l^\tau\ \Delta \mid in_r^\sigma\ \Delta$	injections objects

General terms, namely kinds, families, and objects are denoted by U, and V. 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.

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 Π^{r}). Note that injections in_i need to be decorated with the injected type σ in order to ensure the unicity of typing. We need to generalize the notion of *essence*, which was introduced in [9] 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 Fig. 5.

In order to be able to prove a simple subject reduction result, we need to constrain pairs and co-pairs, *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 that the $\wr \Delta_i \wr \equiv \wr \Delta_j \wr$ in both constructs. We will therefore assume that such pairs and co-pairs 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.

$$
\begin{array}{rcll}
\wr c \wr & \triangleq & c & \\
\wr x \wr & \triangleq & x & \\
\wr \lambda x{:}\sigma.\Delta \wr & \triangleq & \lambda x.\wr \Delta \wr & \\
\wr \lambda^{\mathsf{r}} x{:}\sigma.\Delta \wr & \triangleq & \lambda x.\wr \Delta \wr & \\
\wr \langle \Delta_1\ ,\ \Delta_2 \rangle \wr & \triangleq & \wr \Delta_i \wr & i = 1,2 \\
\wr [\Delta_1\ ,\ \Delta_2] \wr & \triangleq & \wr \Delta_i \wr & i = 1,2 \\
\wr pr_i\ \Delta \wr & \triangleq & \wr \Delta \wr & \\
\wr in_i\ \Delta \wr & \triangleq & \wr \Delta \wr & \\
\wr \Delta_1\ \Delta_2 \wr & \triangleq & \wr \Delta_1 \wr \wr \Delta_2 \wr & \\
\wr \Delta_1 \cdot \Delta_2 \wr & \triangleq & \wr \Delta_2 \wr &
\end{array}
$$

Figure 5: The essence function

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 on the left in Fig. 6. Congruence rules are as usual, except for the two cases dealing with pairs and co-pairs which appear on the right of Fig. 6. 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 $\to_\Delta$, *i.e.* the reduction induced by the rules in Fig. 6.

The restriction on reductions in pairs/co-pairs and the new constructs do not cause any problems in showing that $\to_\Delta$ is locally confluent:

Proposition 2.1. *The reduction relation on well-formed LF_Δ-terms is locally confluent.*

Let $\Gamma \triangleq \{x_1{:}\sigma_1, \ldots, x_n{:}\sigma_n\}$ ($i \neq j$ implies $x_i \not\equiv x_j$), and $\Gamma, x{:}\sigma \triangleq \Gamma \cup \{x{:}\sigma\}$

Let $\Sigma \triangleq \{c_1{:}\sigma_1, \ldots, c_n{:}\sigma_n\}$, and $\Sigma, c{:}\sigma \triangleq \Sigma \cup \{c{:}\sigma\}$

Valid Signatures

$$\frac{}{\langle\,\rangle \text{ sig}}\ (\epsilon\Sigma) \qquad \frac{\Sigma \text{ sig} \quad \vdash_\Sigma K \quad a \notin \text{dom}(\Sigma)}{\Sigma, a{:}K \text{ sig}}\ (K\Sigma) \qquad \frac{\Sigma \text{ sig} \quad \vdash_\Sigma \sigma : \text{Type} \quad c \notin \text{dom}(\Sigma)}{\Sigma, c{:}\sigma \text{ sig}}\ (\sigma\Sigma)$$

Valid Contexts

$$\frac{\Sigma \text{ sig}}{\vdash_\Sigma \langle\,\rangle}\ (\epsilon\Gamma) \qquad \frac{\vdash_\Sigma \Gamma \quad \Gamma \vdash_\Sigma \sigma : \text{Type} \quad x \notin \text{dom}(\Gamma)}{\vdash_\Sigma \Gamma, x{:}\sigma}\ (\sigma\Gamma)$$

Figure 2: Valid Signatures and Contexts

Valid Kinds

$$\frac{\vdash_\Sigma \Gamma}{\Gamma \vdash_\Sigma \text{Type}}\ (Type) \qquad \frac{\Gamma, x{:}\sigma \vdash_\Sigma K}{\Gamma \vdash_\Sigma \Pi x{:}\sigma.K}\ (\Pi K)$$

Valid Families

$$\frac{\vdash_\Sigma \Gamma \quad a{:}K \in \Sigma}{\Gamma \vdash_\Sigma a : K}\ (Const) \qquad \frac{\Gamma \vdash_\Sigma \sigma : K_1 \quad \Gamma \vdash_\Sigma K_2 \quad K_1 =_\Delta K_2}{\Gamma \vdash_\Sigma \sigma : K_2}\ (Conv)$$

$$\frac{\Gamma, x{:}\sigma \vdash_\Sigma \tau : \text{Type}}{\Gamma \vdash_\Sigma \Pi x{:}\sigma.\tau : \text{Type}}\ (\Pi I) \qquad \frac{\Gamma \vdash_\Sigma \sigma : \Pi x{:}\tau.K \quad \Gamma \vdash_\Sigma \Delta : \tau}{\Gamma \vdash_\Sigma \sigma\,\Delta : K[\Delta/x]}\ (\Pi E)$$

$$\frac{\Gamma, x{:}\sigma \vdash_\Sigma \tau : \text{Type}}{\Gamma \vdash_\Sigma \Pi^{\mathsf{r}} x{:}\sigma.\tau : \text{Type}}\ (\Pi^{\mathsf{r}} I) \qquad \frac{\Gamma \vdash_\Sigma \sigma : \Pi^{\mathsf{r}} x{:}\tau.K \quad \Gamma \vdash_\Sigma \Delta : \tau}{\Gamma \vdash_\Sigma \sigma{\cdot}\Delta : K[\Delta/x]}\ (\Pi^{\mathsf{r}} E)$$

$$\frac{\Gamma \vdash_\Sigma \sigma : \text{Type} \quad \Gamma \vdash_\Sigma \tau : \text{Type}}{\Gamma \vdash_\Sigma \sigma \cap \tau : \text{Type}}\ (\cap I) \qquad \frac{\Gamma \vdash_\Sigma \sigma : \text{Type} \quad \Gamma \vdash_\Sigma \tau : \text{Type}}{\Gamma \vdash_\Sigma \sigma \cup \tau : \text{Type}}\ (\cup I)$$

Figure 3: Valid Kinds and Families

The extended type theory LF_Δ is a formal system for deriving judgements of the following forms:

	$\vdash$	σ	σ is a valid signature
	$\vdash_\Sigma$	Γ	Γ is a valid context in Σ
Γ	$\vdash_\Sigma$	K	K is a kind in Γ and Σ
Γ	$\vdash_\Sigma$	$\sigma : K$	σ has kind K in Γ and Σ
Γ	$\vdash_\Sigma$	$\Delta : \sigma$	Δ has type σ in Γ and Σ

The set of rules are defined in Figures 2, 3, and 4. They are syntax-directed. 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 PTS [29].

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. For any pure λ-term M, one can define a Δ-term such that $\wr\,\Delta\,\wr =_\beta M$.

$$\begin{array}{rcl} c_1 & : & \Pi^{\mathsf{r}} x{:}\sigma.(\Pi y{:}\sigma.\sigma) \\ c_2 & : & \Pi^{\mathsf{r}} x{:}(\Pi y{:}\sigma.\sigma).\sigma \\ \Omega & \triangleq & (\lambda x{:}\sigma.c_1{\cdot}x\,x)\,(c_2{\cdot}(\lambda x{:}\sigma.c_1{\cdot}x\,x)) : \sigma \\ \wr\,\Omega\,\wr & = & (\lambda x.x\,x)\,(\lambda x.x\,x) \end{array}$$

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 as those that equate σ and $\sigma \to \sigma$. As a consequence, β-equality of essences becomes undecidable. We will

Valid Objects

$$\frac{\vdash_\Sigma \Gamma \quad c{:}\sigma \in \Sigma}{\Gamma \vdash_\Sigma c : \sigma}\ (Const) \qquad \frac{\vdash_\Sigma \Gamma \quad x{:}\sigma \in \Gamma}{\Gamma \vdash_\Sigma x : \sigma}\ (Var)$$

$$\frac{\Gamma, x{:}\sigma \vdash_\Sigma \Delta : \tau}{\Gamma \vdash_\Sigma \lambda x{:}\sigma.\Delta : \Pi x{:}\sigma.\tau}\ (\Pi I) \qquad \frac{\Gamma \vdash_\Sigma \Delta_1 : \Pi x{:}\sigma.\tau \quad \Gamma \vdash_\Sigma \Delta_2 : \sigma}{\Gamma \vdash_\Sigma \Delta_1\, \Delta_2 : \tau[\Delta_2/x]}\ (\Pi E)$$

$$\frac{\Gamma, x{:}\sigma \vdash_\Sigma \Delta : \tau \quad \wr \Delta \wr =_\eta x}{\Gamma \vdash_\Sigma \lambda^{\mathrm{r}} x{:}\sigma.\Delta : \Pi^{\mathrm{r}} x{:}\sigma.\tau}\ (\Pi^{\mathrm{r}} I) \qquad \frac{\Gamma \vdash_\Sigma \Delta_1 : \Pi^{\mathrm{r}} x{:}\sigma.\tau \quad \Gamma \vdash_\Sigma \Delta_2 : \sigma}{\Gamma \vdash_\Sigma \Delta_1 \cdot \Delta_2 : \tau[\Delta_2/x]}\ (\Pi^{\mathrm{r}} E)$$

$$\frac{\Gamma \vdash_\Sigma \Delta_1 : \sigma \quad \Gamma \vdash_\Sigma \Delta_2 : \tau \quad \wr \Delta_1 \wr \equiv \wr \Delta_2 \wr}{\Gamma \vdash_\Sigma \langle \Delta_1 , \Delta_2 \rangle : \sigma \cap \tau}\ (\cap I) \qquad \frac{\begin{array}{l}\Gamma \vdash_\Sigma \Delta_1 : \Pi y{:}\sigma.\rho\,(in_l^\tau\ y) \quad \wr \Delta_1 \wr \equiv \wr \Delta_2 \wr \\ \Gamma \vdash_\Sigma \Delta_2 : \Pi y{:}\tau.\rho\,(in_r^\sigma\ y) \quad \Gamma \vdash_\Sigma \rho : \Pi y{:}(\sigma \cup \tau).\mathsf{Type}\end{array}}{\Gamma \vdash_\Sigma [\Delta_1 , \Delta_2] : \Pi x{:}\sigma \cup \tau.\rho\, x}\ (\cup E)$$

$$\frac{\Gamma \vdash_\Sigma \Delta : \sigma \cap \tau}{\Gamma \vdash_\Sigma pr_l\, \Delta : \sigma}\ (\cap E_l) \qquad \frac{\Gamma \vdash_\Sigma \Delta : \sigma \cap \tau}{\Gamma \vdash_\Sigma pr_r\, \Delta : \tau}\ (\cap E_r)$$

$$\frac{\Gamma \vdash_\Sigma \Delta : \sigma \quad \Gamma \vdash_\Sigma \sigma \cup \tau : \mathsf{Type}}{\Gamma \vdash_\Sigma in_l^\tau\, \Delta : \sigma \cup \tau}\ (\cup I_l) \qquad \frac{\Gamma \vdash_\Sigma \Delta : \tau \quad \Gamma \vdash_\Sigma \sigma \cup \tau : \mathsf{Type}}{\Gamma \vdash_\Sigma in_r^\sigma\, \Delta : \sigma \cup \tau}\ (\cup I_r)$$

$$\frac{\Gamma \vdash_\Sigma \Delta : \sigma \quad \Gamma \vdash_\Sigma \tau : \mathsf{Type} \quad \sigma =_\Delta \tau}{\Gamma \vdash_\Sigma \Delta : \tau}\ (Conv)$$

Figure 4: Valid Objects

$$\begin{array}{rcl}(\lambda x{:}\sigma.\Delta_1)\,\Delta_2 & \longrightarrow_\beta & \Delta_1[\Delta_2/x] \\ pr_l\,\langle \Delta_1 , \Delta_2 \rangle & \longrightarrow_{pr_l} & \Delta_1 \\ pr_r\,\langle \Delta_1 , \Delta_2 \rangle & \longrightarrow_{pr_r} & \Delta_2 \\ [\Delta_1 , \Delta_2]\, in_l^\sigma\, \Delta_3 & \longrightarrow_{in_l} & \Delta_1\, \Delta_3 \\ [\Delta_1 , \Delta_2]\, in_r^\sigma\, \Delta_3 & \longrightarrow_{in_r} & \Delta_2\, \Delta_3 \\ (\lambda^{\mathrm{r}} x{:}\sigma.\Delta_1)\cdot\Delta_2 & \longrightarrow_{\beta r} & \Delta_1[\Delta_2/x]\end{array}$$

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

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

Figure 6: Reduction semantics

have to rule out such options in relating relevant implications in LF_Δ to subtypes in the type assignment system $\mathcal{B}$ of [3].

2.1 Relating LF_Δ to $\mathcal{B}$

In this subsection we compare and contrast certain design decisions of LF_Δ to the type assignment system introduced in [3], without subtyping rules and the type ω, which we call $\mathcal{B}$. The proof of strong normalization for LF_Δ will rely, in fact, on a *forgetful mapping* from LF_Δ to $\mathcal{B}$.

As pointed out in [3], the elimination rule for union types in $\mathcal{B}$ 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 the following:

$$x\,((\mathsf{I}\,y)\,z)\,((\mathsf{I}\,y)\,z) \longrightarrow_\beta \begin{array}{l} \nearrow^\beta x\,(y\,z)\,((\mathsf{I}\,y)\,z) \searrow_\beta \\ \searrow_\beta x\,((\mathsf{I}\,y)\,z)\,(y\,z) \nearrow^\beta \end{array} x\,(y\,z)\,(y\,z),$$

where I is the identity. In the typing context $B \triangleq x{:}(\sigma_1 \to \sigma_1 \to \tau) \cap (\sigma_2 \to \sigma_2 \to \tau), y{:}\rho \to (\sigma_1 \cup \sigma_2), z{:}\rho$, the first and the last terms can be typed with τ, while the terms in the "fork" cannot. The root 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. We point out that the problem would not arise if $(\cup E)$

is replaced by the rule schema

$$\frac{\begin{array}{l} B, x_1{:}\sigma, \ldots, x_n{:}\sigma \vdash M : \rho \\ B, x_1{:}\tau, \ldots, x_n{:}\tau \vdash M : \rho \quad B \vdash N_i : \sigma \cup \tau \quad i = 1 \ldots n \end{array}}{B \vdash M[N_1/x_1 \ldots N_n/x_n] : \rho} \quad (\cup E')$$

In LF_Δ the formulation of the $(\cup E)$ rule takes a completely different route which does not trigger the counterexample. Indeed, we have introduction and elimination constructs in_l , in_r and $[\,,\,]$ which allow to reduce the term only if we know that the argument, stripped off from the construct, has one of the types of the undisjunction. Pierce's critical term can be expressed in LF_Δ only in the form

$$[\underbrace{(\lambda x_1{:}\sigma_1.(pr_l\ x)\ x_1\ x_1)}_{\Delta_1}, \underbrace{(\lambda x_2{:}\sigma_2.(pr_r\ x)\ x_2\ x_2)}_{\Delta_2}]\ (((\underbrace{\lambda x_3{:}\rho'.x_3}_{\Delta_3})\ y)\ z)$$

for $a{:}\sigma_1 \cup \sigma_2 \to \sigma_1 \cup \sigma_2 \to \mathrm{Type}$ and $x{:}\Pi x_1 : \sigma_1 a\ (in_l^{\sigma_2}\ x_1)\ (in_l^{\sigma_2}\ x_1) \cap \Pi x_2 : \sigma_2 a\ (in_r^{\sigma_1}\ x_2)\ (in_r^{\sigma_1}\ x_2), y{:}\rho', z{:}\rho$, where $\rho' \equiv \rho \to (\sigma_1 \cup \sigma_2)$. 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.

The following result will be useful in the following section.

Theorem 2.2. *The systems $\mathcal{B}$ and $\mathcal{B}'$, where the rule $(\cup E)$ is replaced by $(\cup E')$, are strongly normalizing.*

The proof is embedded in Theorem 4.8 of [3]. But it can also be straightforwardly obtained using the general Computability Method presented in [16] Section 4.

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.* [13]. In particular, we prove in order: uniqueness of types and kinds, structural properties, normalization for raw well-formed terms, and hence confluence. Then we prove the inversion property, the subderivation property, subject reduction, and finally decidability. Apart from normalization all the other results are long structural inductions, albeit ultimately routine.

Theorem 2.3. *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 $x \notin FV(\Gamma') \cup FV(\alpha)$.*
3. *Transitivity: 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.*

Theorem 2.4 (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 $K =_\Delta K'$.*

Strong Normalization. In order to prove strong normalization we will map LF_Δ-terms into $\mathcal{B}$-terms taking advantage of Theorem 2.2.

Definition 2.5. Let the forgetful mappings $\|\cdot\|$ and $|\cdot|$ be defined as in Figure 7.

The forgetful mappings are extended to contexts and signatures in the obvious way. The clauses for strong pairs/co-pairs are justified by the following lemma:

Lemma 2.6. *If $\Gamma \vdash_\Sigma \langle \Delta_1\ , \Delta_2 \rangle : \sigma \cap \tau$ is provable then $|\Delta_1| \equiv |\Delta_2|$ and similarly if $\Gamma \vdash_\Sigma [\Delta_1\ , \Delta_2] : \Pi x : \sigma \cup \tau.\rho(x)$ then $|\Delta_1| \equiv |\Delta_2|$.*

The following lemmas are proved by straightforward structural induction.

Lemma 2.7.

1. *If $\sigma =_\Delta \tau$, then $\|\sigma\| =_\beta \|\tau\|$.*
2. *If $K_1 =_\Delta K_2$, then $\|K_1\| =_\beta \|K_2\|$.*

Lemma 2.8.

1. *$|\Delta_1[\Delta_2/x]| =_\beta |\Delta_1|\ [|\Delta_2|\ /x]$.*
2. *$|\sigma[\Delta/x]| =_\beta |\sigma|\ [|\Delta|\ /x]$.*

Lemma 2.9.

1. *If $\Gamma \vdash_\Sigma \sigma : K$, then $\|\Gamma\| \vdash_{\mathcal{B}^+} |\sigma| : \|K\|$.*
2. *If $\Gamma \vdash_\Sigma \Delta : \sigma$, then $\|\Gamma\| \vdash_{\mathcal{B}^+} |\Delta| : \|\sigma\|$.*

where $\vdash_{\mathcal{B}^+}$ denotes the type system $\mathcal{B}$, augmented by the infinite set of axioms for kinds (K) $c_\sigma : \omega \to (\sigma \to \omega) \to \omega$, for each type σ.

Lemma 2.10.

1. *If $\sigma \longrightarrow_\beta \tau$, then $|\sigma| = \longrightarrow_\beta^+ |\tau|$.*
2. *if $\Delta_1 \longrightarrow_\beta \Delta_2$, then $|\Delta_1| \longrightarrow_\beta^+ |\Delta_2|$.*

Parallel reduction enjoys the strong normalization property, *i.e.*

Theorem 2.11.

1. *The 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 term can be type-annotated so as to be the essence of a Δ-term.*

Proof. 1) Strong normalization derives directly from Lemma 2.10 and Theorem 2.2.
2) Use rules $(\cap I)$ and $(\cap E)$ to encode the SN terms of λ-calculus inductively defined by: $\Delta_1 \ldots \Delta_n \in SN \Rightarrow x\ \Delta_1 \ldots \Delta_n \in SN$ and $\Delta[\Delta'/x]\ \Delta_1 \ldots \Delta_n \in SN$, and $\Delta' \in SN \Rightarrow (\lambda x{:}\sigma.\Delta)\ \Delta'\ \Delta_1\ \ldots\ \Delta_n \in SN$. □

Incidentally, we point out that to our knowledge LF_Δ is the first fully typed system to exhibit this property.

Local confluence (Proposition 2.1) and strong normalization (Theorem 2.11) entail confluence, so we have

Theorem 2.12. *LF_Δ is confluent, i.e.:*

1. *If $K_1 \longrightarrow_\Delta^* K_2$ and $K_1 \longrightarrow_\Delta^* K_3$, then there exists an 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 there exists a σ_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 there exists a Δ_4 such that $\Delta_2 \longrightarrow_\Delta^* \Delta_4$ and $\Delta_3 \longrightarrow_\Delta^* \Delta_4$.*

The following lemmas are proved by structural induction.

Lemma 2.13 (Inversion properties).

$$
\begin{array}{rcl}
\|\mathsf{Type}\| & = & \omega \\
\|\Pi x{:}\sigma.K\| & = & \|\sigma\| \to \|K\| \\
\|a\| & = & a \\
\|\Pi x{:}\sigma.\tau\| & = & \|\sigma\| \to \|\tau\| \\
\|\Pi^{\mathsf{r}} x{:}\sigma.\tau\| & = & \|\sigma\| \to \|\tau\| \\
\|\sigma\,\Delta\| & = & \|\sigma\| \\
\|\sigma \cdot \Delta\| & = & \|\sigma\| \\
\|\sigma \cap \tau\| & = & \|\sigma\| \cap \|\tau\| \\
\|\sigma \cup \tau\| & = & \|\sigma\| \cup \|\tau\|
\end{array}
\qquad
\begin{array}{rcl}
|x| & = & x \\
|c| & = & c \\
|\sigma\,\Delta| & = & |\sigma|\,|\Delta| \\
|\sigma \cdot \Delta| & = & |\sigma|\,|\Delta| \\
|\Delta_1\,\Delta_2| & = & |\Delta_1|\,|\Delta_2| \\
|\lambda x{:}\sigma.\Delta| & = & (\lambda y.\lambda x.\,|\Delta|)\;|\sigma| \quad y \notin fv(\Delta) \\
|\lambda^{\mathsf{r}} x{:}\sigma.\Delta| & = & (\lambda y.\lambda x.\,|\Delta|)\;|\sigma| \quad y \notin fv(\Delta)
\end{array}
\qquad
\begin{array}{rcl}
|\Pi x{:}\sigma.\tau| & = & c_{\|\sigma\|}\;\|\sigma\|\;(\lambda x.\,\|\tau\|) \\
|\Pi^{\mathsf{r}} x{:}\sigma.\tau| & = & c_{\|\sigma\|}\;\|\sigma\|\;(\lambda x.\,\|\tau\|) \\
|\langle \Delta_1\,,\,\Delta_2\rangle| & = & |\Delta_1| \\
|[\Delta_1\,,\,\Delta_2]| & = & |\Delta_1| \\
|pr_l\,\Delta| & = & |\Delta| \\
|pr_r\,\Delta| & = & |\Delta| \\
|in_l^{\sigma}\,\Delta| & = & |\Delta| \\
|in_r^{\sigma}\,\Delta| & = & |\Delta|
\end{array}
$$

Figure 7: The forgetful mappings $\|\cdot\|$ and $|\cdot|$.

(1) If $\Pi x{:}\sigma.T =_\Delta T''$, then $T'' \equiv \Pi x{:}\sigma'.T'$, for some σ', T', such that $\sigma' =_\Delta \sigma$, and $T' =_\Delta T$.
(2) If $\Pi^{\mathsf{r}} x{:}\sigma.T =_\Delta T''$, then $T'' \equiv \Pi^{\mathsf{r}} x{:}\sigma'.T'$, for some σ', T', such that $\sigma' =_\Delta \sigma$, and $T' =_\Delta T$.
(3) If $\sigma \cap \tau =_\Delta T$, then $T \equiv \sigma' \cap \tau'$, for some σ', τ', such that $\sigma' =_\Delta \sigma$, and $\tau' =_\Delta \tau$.
(4) If $\sigma \cup \tau =_\Delta T$, then $T \equiv \sigma' \cup \tau'$, 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 $\Gamma \vdash_\Sigma \Delta_1 : \Pi y{:}\sigma.\rho(in_l^{\tau}\,y)$, $\Gamma \vdash_\Sigma \Delta_2 : \Pi y{:}\tau.\rho(in_r^{\sigma}\,y)$, and $\wr\,\Delta_1\,\wr =_\beta \wr\,\Delta_2\,\wr$.
(9) If $\Gamma \vdash_\Sigma pr_l\,\Delta : \sigma$, then $\Gamma \vdash_\Sigma \Delta{:}\sigma \cap \tau$, for some τ.
(10) If $\Gamma \vdash_\Sigma pr_r\,\Delta : \tau$, then $\Gamma \vdash_\Sigma \Delta{:}\sigma \cap \tau$, for some σ.
(11) If $\Gamma \vdash_\Sigma in_l^{\tau}\,\Delta : \sigma \cup \tau$, then $\Gamma \vdash_\Sigma \Delta : \sigma$ and $\Gamma \vdash_\Sigma \sigma \cup \tau$: Type.
(12) If $\Gamma \vdash_\Sigma in_r^{\sigma}\,\Delta : \sigma \cup \tau$, then $\Gamma \vdash_\Sigma \Delta : \tau$ and $\Gamma \vdash_\Sigma \sigma \cup \tau$: Type.

Proposition 2.14 (Subderivation).

(1) A derivation of $\vdash_\Sigma \langle\rangle$ 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 $\vdash_\Sigma \sigma$:Type.
(4) A derivation of $\vdash_\Sigma \Gamma, x{:}\sigma$ has subderivations of Σ sig, $\vdash_\Sigma \Gamma$, and $\Gamma \vdash_\Sigma \sigma$:Type.
(5) A derivation of $\Gamma \vdash_\Sigma \alpha$ has subderivations of Σ sig and $\vdash_\Sigma \Gamma$.
(6) Given a derivation of the judgement $\Gamma \vdash_\Sigma \alpha$, and a subterm occurring in the subject of this judgement, there exists a derivation of a judgement having this subterm as a subject.

Theorem 2.15 (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 define a possible algorithm for checking judgements in LF_Δ by computing a type or a kind for a term, and then testing for *definitional equality*, *i.e.* $=_\Delta$, against the given type or kind. This is achieved by reducing both to their unique normal forms and checking that they are identical up to α-conversion. Therefore we finally have:

Theorem 2.16 (Decidability). *All the type judgments of LF_Δ are recursively decidable.*

Relevant Implications and Type Inclusion. Relevant implication is related to the notion of *type inclusion* and the rules of *subtyping*, see [4, 9]. In the following theorem we show how relevant implication subsumes the type-inclusion rules of the theory Ξ of [3], without rule (10): we call Ξ' the resulting set.

Theorem 2.17 (Type Inclusion). *The judgement $\langle\rangle \vdash_{\langle\rangle} \Delta : \Pi^{\mathsf{r}} x{:}\tau.\sigma$ holds iff $\tau \leq \sigma$ holds in the subtype theory Ξ' of $\mathcal{B}$.*

Proof. For the "only if" part, it is possible to write a "type-annotated" term whose essence is an η−expansion of the identity corresponding to each of the axioms and rules in Ξ. The converse follows from Lemma 2.13. □

To encompass also rule (10) of the subtype theory Ξ, we need to strenghten rule (Π^{r}) taking essence up to β-equality and use the term $\lambda z^{\sigma\cap(\rho\cup\tau)}.\lambda x^{\sigma}.[\lambda z^{\rho}.in_l{}^{\sigma\cap\tau}\langle x\,,\,z\rangle, \lambda z^{\tau}.in_r{}^{\sigma\cap\rho}\langle x\,,\,z\rangle](pr_l\,z)\,(pr_r\,z){:}(\sigma\cap\rho)\cup(\sigma\cap\tau)$.

Similarly if we make assumptions of relevant type these induce new axioms for $\leq$.

3 EXAMPLES

In order to highlight the advantages of LF_Δ we assume the viewpoint of a user interested in reasoning formally with a logical system featuring strong intersection and union types, which we call $\mathcal{L}$ (inspired by the logical systems appearing in [4, 20, 23]). It amounts essentially to a system for inhabitability in $\mathcal{B}$ or in the system of [9]. We start by comparing the signatures of $\mathcal{L}$ in LF and in LF_Δ.

The intuition behind these encodings, which can be substantiated by an appropriate *adequacy result*, is the following: in the given signatures, if $\Gamma \vdash_\Sigma \Delta : \sigma$ then there exist B, M, σ' such that $B \vdash_{\mathcal{B}} M : \sigma'$ where M, B, and σ' are the "decodings" respectively of Δ, Γ, and σ, and conversely, if $B \vdash_{\mathcal{B}} M : \sigma'$ then there exist Δ, σ, and Γ, *i.e.* "encodings" of M, B and σ respectively, such that $\Gamma \vdash_\Sigma \Delta : \sigma$.

Thus, in the appropriate signature and context, each LF_Δ-typable term Δ of type *encoding of* σ, encodes a type assignment derivation in $\mathcal{B}$, *i.e.* the *decoding of* Δ, of type σ.

LF encoding of $\mathcal{L}$. Figure 8 presents a pure LF encoding of $\mathcal{L}$ in Coq syntax using HOAS. We use HOAS in order to take advantage

```
(* 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 strong/relevant arrows *)
%Axiom Sabst : forall (s t : o) (M : OK s → OK t), (forall (N : OK s), (Eq s t N (M N))) → OK (relev s t).
%Axiom Sapp : forall (s t : o), OK (relev 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 Copair : 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 strong/relevant arrow constructors *)
%Axiom Eqsabst : forall (s t s' t' : o) (M : OK (relev s t)) (N : OK (relev s' t')), Eq (relev s t) (relev s' t') M N.
%Axiom Eqsapp : forall (s t : o) (M : OK (relev s t)) (x : OK s), Eq s t x (Sapp s t M x).

(* 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 Eqcopair : 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) (Copair s t u M N O pf).
```

Figure 8: LF encoding of $\mathcal{L}$ (Coq syntax)

of the higher order features of the frameworks: other abstract syntax representation techniques would not be much different, but more verbose. The Eq predicate plays the same role of the *essence* function in LF_Δ, namely, it encodes the judgement 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 Copair axiom (*i.e.*, the elimination rule of the union type constructor). We have an Eq axiom for each proof rule. Examples of this encoding can be found in intersection_union.v.

LF_Δ encoding of $\mathcal{L}$. LF_Δ allows for a shallow encoding of $\mathcal{L}$ because the metalanguage subsumes the source language. Let $\to$ and $\to_r$ denote the non-dependent product type Π and the relevant product type Π^r, respectively. The encoding is given below:

$$
\begin{array}{rcl}
o & : & \mathsf{Type} \qquad c_{\to}, c_{\cap}, c_{\cup} : o \to o \to o \\
obj & : & o \to \mathsf{Type} \\
c_{abst} & : & \Pi s\,t{:}o.(obj\ s \to obj\ t) \to_r obj\ (c_{\to}\ s\ t) \\
c_{app} & : & \Pi s\,t{:}o.obj\ (c_{\to}\ s\ t) \to_r obj\ s \to obj\ t \\
c_{in_i} & : & \Pi s\,t{:}o.(obj\ s \cup obj\ t) \to_r obj\ (c_{\cup}\ s\ t) \\
c_{sconj} & : & \Pi s\,t{:}o.(obj\ s \cap obj\ t) \to_r obj\ (c_{\cap}\ s\ t) \\
c_{sdisj} & : & \Pi s\,t{:}o.obj\ (c_{\cup}\ s\ t) \to_r (obj\ s \cup obj\ t)
\end{array}
$$

This encoding can be typechecked using our concrete syntax (file intersection_union.bull). Notice the use of the relevant arrow in the type signature constants. It is not mandatory but it paves the way for a more perspicuous adequacy theorem, where the decoding of LF_Δterm would be the very essence function!

Three base types: atomic propositions, non-atomic goals and non-atomic programs.

$\alpha, \gamma_0, \pi_0 \quad : \quad \text{Type}$

Goals and programs are defined from the base types.

$\gamma \quad = \quad \alpha \cup \gamma_0 \qquad \pi = \alpha \cup \pi_0$

Constructors (implication, conjunction, disjunction).

$\text{impl} \quad : \quad (\pi \to \gamma \to \gamma_0) \cap (\gamma \to \pi \to \pi_0)$

$\text{impl}_1 \quad = \quad \lambda x{:}\pi.\lambda y{:}\gamma.in_r^{\alpha}\ (pr_l\ \text{impl}\ x\ y) \qquad \text{impl}_2 = \lambda x{:}\gamma.\lambda y{:}\pi.in_r^{\alpha}\ (pr_r\ \text{impl}\ x\ y)$

$\text{and} \quad : \quad (\gamma \to \gamma \to \gamma_0) \cap (\pi \to \pi \to \pi_0) \qquad \text{and}_1 = \lambda x{:}\gamma.\lambda y{:}\gamma.in_r^{\alpha}\ (pr_l\ \text{and}\ x\ y) \qquad \text{and}_2 = \lambda x{:}\pi.\lambda y{:}\pi.in_r^{\alpha}\ (pr_r\ \text{and}\ x\ y)$

$\text{or} \quad : \quad (\gamma \to \gamma \to \gamma_0) \qquad \text{or}_1 = \lambda x{:}\gamma.\lambda y{:}\gamma.in_r^{\alpha}\ (\text{or}\ x\ y)$

solve $p\ g$ indicates that the judgment $p \vdash g$ is valid. and bchain $p\ a\ g$ indicates that, if $p \vdash g$ is valid, then $p \vdash a$ is valid.

$\text{solve} \quad : \quad \pi \to \gamma \to \text{Type} \qquad \text{bchain} : \pi \to \alpha \to \gamma \to \text{Type}$

Rules for solve:

$- \quad : \quad \Pi_{(p:\pi)(g_1,g_2:\gamma)}\text{solve}\ p\ g_1 \to \text{solve}\ p\ g_2 \to \text{solve}\ p\ (\text{and}_1\ g_1\ g_2)$

$- \quad : \quad \Pi_{(p:\pi)(g_1,g_2:\gamma)}\text{solve}\ p\ g_1 \to \text{solve}\ p\ (\text{or}_1\ g_1\ g_2)$

$- \quad : \quad \Pi_{(p:\pi)(g_1,g_2:\gamma)}\text{solve}\ p\ g_2 \to \text{solve}\ p\ (\text{or}_1\ g_1\ g_2)$

$- \quad : \quad \Pi_{(p_1,p_2:\pi)(g:\gamma)}\text{solve}\ (\text{and}_2\ p_1\ p_2)\ g \to \text{solve}\ p_1\ (\text{impl}_1\ p_2\ g)$

$- \quad : \quad \Pi_{(p:\pi)(a:\alpha)(g:\gamma)}\text{bchain}\ p\ a\ g \to \text{solve}\ p\ g \to \text{solve}\ p\ (in_l^{\gamma_0}\ a)$

Rules for bchain:

$- \quad : \quad \Pi_{(a:\alpha)(g:\gamma)}\text{bchain}\ (\text{impl}_2\ g\ (in_l\ \pi_0 a))\ a\ g$

$- \quad : \quad \Pi_{(p_1,p_2:\pi)(a:\alpha)(g:\gamma)}\text{bchain}\ p_1\ a\ g \to \text{bchain}\ (\text{and}_2\ p_1\ p_2)\ a\ g$

$- \quad : \quad \Pi_{(p_1,p_2:\pi)(a:\alpha)(g:\gamma)}\text{bchain}\ p_2\ a\ g \to \text{bchain}\ (\text{and}_2\ p_1\ p_2)\ a\ g$

$- \quad : \quad \Pi_{(p:\pi)(a:\alpha)(g,g_1,g_2:\gamma)}\text{bchain}\ (\text{impl}_2\ (\text{and}_1\ g_1\ g_2)\ p)\ a\ g \to \text{bchain}\ (\text{impl}_2\ g_1\ (\text{impl}_2\ g_2\ p))\ a\ g$

$- \quad : \quad \Pi_{(p_1,p_2:\pi)(a:\alpha)(g,g_1:\gamma)}\text{bchain}\ (\text{impl}_2\ g_1\ p_1)\ a\ g \to \text{bchain}\ (\text{impl}_2\ g_1\ (\text{and}_2\ p_1\ p_2))\ a\ g$

$- \quad : \quad \Pi_{(p_1,p_2:\pi)(a:\alpha)(g,g_1:\gamma)}, \text{bchain}\ (\text{impl}_2\ g_1\ p_2)\ a\ g \to \text{bchain}\ (\text{impl}_2\ g_1\ (\text{and}_2\ p_1\ p_2))\ a\ g$

Figure 9: LF_Δ encoding of Hereditary Harrop Formulas (LF_Δ concrete syntax).

$\text{obj}' \quad : \quad \text{Type} \qquad \text{fam}' : \text{Type} \qquad \text{knd}' : \text{Type} \qquad \text{sup}' : \text{Type} \qquad \text{same} : \text{obj}' \cap (\text{fam}' \cap (\text{knd}' \cap \text{sup}')) \qquad \text{term} : (\text{obj}' \cup (\text{fam}' \cup (\text{knd}' \cup \text{sup}'))) \to \text{Type}$

$\text{obj} \quad = \quad \text{term}\ (in_l^{\text{fam}' \cup (\text{knd}' \cup \text{sup}')}\ (pr_l\ \text{same}))$

$\text{fam} \quad = \quad \text{term}\ (in_r^{\text{obj}'}\ in_l^{\text{knd}' \cup \text{sup}'}\ (pr_l\ pr_r\ \text{same}))$

$\text{knd} \quad = \quad \text{term}\ (in_r^{\text{obj}'}\ in_r^{\text{fam}'}\ in_l^{\text{sup}'}\ (pr_l\ pr_r\ pr_r\ \text{same}))$

$\text{sup} \quad = \quad \text{term}\ (in_r^{\text{obj}'}\ in_r^{\text{fam}'}\ in_r^{\text{knd}'}\ (pr_r\ pr_r\ pr_r\ \text{same}))$

$\text{tp} \quad : \quad \text{knd} \cap \text{sup} \qquad * = pr_l\ \text{tp} \qquad \Box = pr_r\ \text{tp}$

$\text{lam} \quad : \quad (\text{fam} \to (\text{obj} \to \text{obj}) \to \text{obj}) \cap (\text{fam} \to (\text{obj} \to \text{fam}) \to \text{fam}) \qquad \text{lam}_1 = pr_l\ \text{lam} \qquad \text{lam}_2 = pr_r\ \text{lam}$

$\text{pi} \quad : \quad (\text{fam} \to (\text{obj} \to \text{fam}) \to \text{fam}) \cap (\text{fam} \to (\text{obj} \to \text{knd}) \to \text{knd}) \qquad \text{pi}_1 = pr_l\ \text{pi} \qquad \text{pi}_2 = pr_r\ \text{pi}$

$\text{app} \quad : \quad (\text{obj} \to \text{obj} \to \text{obj}) \cap (\text{fam} \to \text{obj} \to \text{fam}) \qquad \text{app}_1 = pr_l\ \text{app} \qquad \text{app}_2 = pr_r\ \text{app}$

$\text{of} \quad : \quad (\text{obj} \to \text{fam} \to \text{Type}) \cap ((\text{fam} \to \text{knd} \to \text{Type}) \cap (\text{knd} \to \text{sup} \to \text{Type})) \qquad \text{of}_1 = pr_l\ \text{of} \qquad \text{of}_2 = pr_l\ pr_r\ \text{of} \qquad \text{of}_3 = pr_r\ pr_r\ \text{of}$

$- \quad : \quad \text{of}\ *\ \Box$

$\text{oflam}_1 \quad : \quad \Pi_{(t_1:\text{fam})(t_2:\text{obj}\to\text{obj})(t_3:\text{obj}\to\text{fam})}\text{of}_2\ t_1\ * \to (\Pi_{(x:\text{obj})}\text{of}_1\ x\ t_1 \to \text{of}_1\ (t_2\ x)\ (t_3\ x)) \to \text{of}_1\ (\text{lam}_1\ t_1\ t_2)\ (\text{pi}_1\ t_1\ t_3)$

$\text{oflam}_2 \quad : \quad \Pi_{(t_1:\text{fam})(t_2:\text{obj}\to\text{fam})(t_3:\text{obj}\to\text{knd})}\text{of}_2\ t_1\ * \to (\Pi_{(x:\text{obj})}\text{of}_1\ x\ t_1 \to \text{of}_2\ (t_2\ x)\ (t_3\ x)) \to \text{of}_2\ (\text{lam}_2\ t_1\ t_2)\ (\text{pi}_2\ t_1\ t_3)$

$\text{ofpi}_1 \quad : \quad \Pi_{(t_1:\text{fam})(t_2:\text{obj}\to\text{fam})}\text{of}_2\ t_1\ * \to (\Pi_{(x:\text{obj})}\text{of}_1\ x\ t_1 \to \text{of}_2\ (t_2\ x)\ *) \to \text{of}_2\ (\text{pi}_1\ t_1\ t_2)\ *$

$\text{ofpi}_2 \quad : \quad \Pi_{(t_1:\text{fam})(t_2:\text{obj}\to\text{knd})}\text{of}_2\ t_1\ * \to (\Pi_{(x:\text{obj})}\text{of}_1\ x\ t_1 \to \text{of}_3\ (t_2\ x)\ \Box) \to \text{of}_3\ (\text{pi}_2\ t_1\ t_2)\ \Box$

$\text{ofapp}_1 \quad : \quad \Pi_{(t_1:\text{obj})(t_2:\text{obj})(t_3:\text{fam})(t_4:\text{obj}\to\text{fam})}\text{of}_1\ t_1\ (\text{pi}_1\ t_3\ t_4) \to \text{of}_1\ t_2\ t_3 \to \text{of}_1\ (\text{app}_1\ t_1\ t_2)\ (t_4\ t_2)$

$\text{ofapp}_2 \quad : \quad \Pi_{(t_1:\text{fam})(t_2:\text{obj})(t_3:\text{fam})(t_4:\text{obj}\to\text{knd})}\text{of}_2\ t_1\ (\text{pi}_2\ t_3\ t_4) \to \text{of}_2\ t_2\ t_3 \to \text{of}_2\ (\text{app}_2\ t_1\ t_2)\ (t_4\ t_2)$

$\text{oflameq} \quad = \quad \langle \text{oflam}_1, \text{oflam}_2 \rangle \qquad \text{ofpieq} = \langle \text{ofpi}_1, \text{ofpi}_2 \rangle \qquad \text{ofappeq} = \langle \text{ofapp}_1, \text{ofapp}_2 \rangle$

Figure 10: LF in LF_Δ.

In the following we show the expressive power of LF_Δ encoding classical features of typing systems with strong intersection and union.

Auto application. The provable judgement $\vdash_{\mathcal{B}} \lambda x.x\,x : \sigma \cap (\sigma \to \tau) \to \tau$ in $\mathcal{B}$, is faithfully rendered in LF_Δ by the provable LF_Δ-judgement of $\vdash_\Sigma \lambda x{:}\sigma \cap (\sigma \to \tau).(pr_r\ x)\ (pr_l\ x) : \sigma \cap (\sigma \to \tau) \to \tau$.

Polymorphic identity. The provable judgement $\vdash_{\mathcal{B}} \lambda x.x : (\sigma \to \sigma) \cap (\tau \to \tau)$ in $\mathcal{B}$, is faithfully rendered in LF_Δ by the provable judgement $\vdash_\emptyset \langle \lambda x{:}\sigma.x\,, \lambda x{:}\tau.x \rangle : (\sigma \to \sigma) \cap (\tau \to \tau)$.

Commutativity of union. The provable judgement $\lambda x.x : (\sigma \cup \tau) \to (\tau \cup \sigma)$ in $\mathcal{B}$ is faithfully rendered in LF_Δ by the provable judgement $\lambda x{:}\sigma \cup \tau.[\lambda y{:}\sigma.in_r^{\tau}\ y\,, \lambda y{:}\tau.in_l^{\sigma}\ y]\ x : (\sigma \cup \tau) \to (\tau \cup \sigma)$.

LF_Δ encoding of Hereditary Harrop Formulas. Pfenning's encoding of [25] can be expressed as in Fig. 9 (file pfenning_harrop.bull). We also add rules for solving and backchaining. The encoding makes a subtle use of unions and intersections. Without them the encoding would be far more tedious. Hereditary Harrop formulas can

be recursively defined using two mutually recursive syntactical objects called programs (π) and goals (γ).

$$\gamma \quad := \quad \alpha \mid \gamma \wedge \gamma \mid \pi \Rightarrow \gamma \mid \gamma \vee \gamma$$
$$\pi \quad := \quad \alpha \mid \pi \wedge \pi \mid \gamma \Rightarrow \pi$$

LF_Δ encoding of LF. We show a LF_Δ encoding of LF making essential use of intersection types. It can be typechecked using the concrete syntax in Fig. 10 (file lf.bull). Having intersection permits to highlight that some rules are "essentially" the same. For instance, all the rules for typing LF's λ-abstractions are the same, and it is shown by oflameq.

Pierce's program is faithfully rendered in LF_Δ by

$$\begin{array}{rcl} Neg & : & \text{Type} \qquad Zero : \text{Type} \qquad Pos : \text{Type} \\ T & : & \text{Type} \qquad F : \text{Type} \\ Test & : & Pos \cup Neg \\ Is\_0 & : & (Neg \to F) \cap ((Zero \to T) \cap (Pos \to F)) \\ Is\_0\_Test & \triangleq & [\lambda x{:}Pos.pr_r\ (pr_r\ Is\_0)\ x\ ,\ \lambda x{:}Neg.pr_l\ Is\_0\ x]\ Test \end{array}$$

The above example clearly illustrates the advantages of taking the LF_Δ framework. In plain LF we would render such an example only deeply encoding $\mathcal{B}$ ending up with the verbose code in file pierce_program.v.

4 IMPLEMENTATION AND FUTURE WORK

We have implemented in Ocaml suitable algorithms for type reconstruction, and type and subtype checking; these algorithms are used as modules to build a prototype kernel for a Logical Framework featuring strong union, strong intersection, and relevant implication. We have implemented the subtyping algorithm [19] which extends the well-known Hindley algorithm for intersection types [15] with union types. The subtyping algorithm has been mechanically proved correct in Coq, in the spirit of the Bessai's mechanized proof of a subtyping algorithm for intersection types [7]. 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 type-check a proof or normalize it, using a strong reduction evaluator. We use the syntax of Pure Type Systems to improve the compactness and the modularity of the kernel. Binders are implemented using de Brujin indexes. We implemented the conversion rule in the simplest way possible: whenever we need to compare types, we syntactically compare their normal form. Abstract and concrete syntax are mostly aligned, and the concrete syntax is similar to the concrete syntax of Coq. The developpement is available on Bull and Bull-Subtyping.

As future work, we are planning to design a higher-order unification algorithm for Δ-terms and a bidirectional refinement algorithm, similar to the one found in [2]. The refinement can be split into two parts: the essence refinement and the typing refinement. In the same way, there will be a unification algorithm for the essence terms, and a unification algorithm for Δ-terms.

The bidirectional refinement algorithm aims to have partial type inference, and to give as much information as possible to a hypothetical solver, or 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 $\wr\, ?y \,\wr = x$.

Acknowledgment. We are grateful to D. Dougherty, U. de'Liguoro and A. Nuyts. And also A. Momigliano, B. Pientka and A. Appel and the referees of [30] for their useful remarks.

REFERENCES

[1] Samson Abramsky. Domain theory in logical form. *Annals of Pure and Applied Logic* 51, 1 (1991), 1–77

[2] Andrea Asperti and Wilmer Ricciotti and Claudio Sacerdoti Coen and Enrico Tassi. 2017. A bidirectional refinement algorithm for the calculus of (co)inductive constructions. In *Logical Methods in Computer Science*, Vol. 8.

[3] Franco Barbanera, Mariangiola Dezani-Ciancaglini, and Ugo de'Liguoro. 1995. Intersection and union types: syntax and semantics. *Inf. Comput.* 119, 2 (1995).

[4] Franco Barbanera and Simone Martini. 1994. Proof-functional connectives and realizability. *Archive for Mathematical Logic* 33 (1994), 189–211.

[5] Henk Barendregt. 2013. *The λ-Calculus with types.* Cambridge University Press.

[6] Henk Barendregt, Mario Coppo, and Mariangiola Dezani-Ciancaglini. 1983. A Filter Lambda Model and the Completeness of Type Assignment. *Journal of Symbolic Logic* 48, 4 (1983), 931–940.

[7] Jan Bessai. Extracting a formally veri ed Subtyping Algorithm for Intersection Types from Ideals and Filters. Talk at COST Types, slides, 2016.

[8] Mario Coppo, Mariangiola Dezani-Ciancaglini, Furio Honsell, Giuseppe Longo. *Extended type structures and filter lambda models.* G. Lolli, et al. (Eds.), Logic Colloquium '82, North-Holland, Amsterdam (1983), pp. 241–262.

[9] Daniel J. Dougherty, Ugo de'Liguoro, Luigi Liquori, and Claude Stolze. 2016. A Realizability Interpretation for Intersection and Union Types. In *APLAS (Lecture Notes in Computer Science)*, Vol. 10017. Springer, 187–205.

[10] Daniel J. Dougherty and Luigi Liquori. 2010. Logic and Computation in a Lambda Calculus with Intersection and Union Types. In *LPAR (Lecture Notes in Computer Science)*, Vol. 6355. Springer, 173–191.

[11] Joshua Dunfield. 2012. Elaborating Intersection and Union Types. In *ICFP '12*. ACM, 17–28.

[12] Alain Frisch, Giuseppe Castagna, and Véronique Benzaken. 2008. Semantic subtyping: Dealing set-theoretically with function, union, intersection, and negation types. *J. ACM* 55, 4 (2008), 19:1–19:64.

[13] Robert Harper, Furio Honsell, and Gordon Plotkin. 1993. A Framework for Defining Logics. *J. ACM* 40, 1 (1993), 143–184.

[14] Robert Harper and Daniel R. Licata. 2007. Mechanizing Metatheory in a Logical Framework. *Journal of Functional Programming* 17, 4–5 (July 2007), 613–673.

[15] J. Roger Hindley. 1992. The Simple Semantics for Coppo-Dezani-Sallé Types. In *International Symposium on Programming. (Lecture Notes in Computer Science)*, Vol. 137. Springer, 212?226.

[16] Furio Honsell, and Marina Lenisa. 1993. Semantical Analysis of Perpetual Strategies in lambda-Calculus. *Theor. Comput. Sci.* 212, 1–2 (1999), 183–209.

[17] Luigi Liquori, Andreas Nuyts and Claude Stolze Private communications, 2017.

[18] Luigi Liquori and Simona Ronchi Della Rocca. 2007. Intersection Typed System *à la* Church. *Information and Computation* 9, 205 (2007), 1371–1386.

[19] Luigi Liquori and Claude Stolze. 2017. A Decidable Subtyping Logic for Intersection and Union Types. In *TTCS (Lecture Notes in Computer Science)*. Springer. To appear. Also on https://hal.archives-ouvertes.fr/hal-01488428.

[20] Edgar G. K. Lopez-Escobar. 1985. Proof functional connectives. In *Methods in Mathematical Logic (Lecture Notes in Mathematics)*, Vol. 1130. Springer-Verlag, 208–221.

[21] David B. MacQueen, Gordon D. Plotkin, and Ravi Sethi. 1986. An Ideal Model for Recursive Polymorphic Types. *Information and Control* 71, 1/2 (1986), 95–130.

[22] Robert K. Meyer and Richard Routley. 1972. Algebraic analysis of entailment I. *Logique et Analyse* 15 (1972), 407–428.

[23] Grigori Mints. 1989. The Completeness of Provable Realizability. *Notre Dame Journal of Formal Logic* 30, 3 (1989), 420–441.

[24] Alexandre Miquel. 2001. The Implicit Calculus of Constructions. In *TLCA. (Lecture Notes in Computer Science)*, Vol. 2044. Springer, 344–359.

[25] Frank Pfenning. 1993. Refinement types for logical frameworks. In *Types.* 285–299.

[26] Benjamin C. Pierce. 1991. *Programming with intersection types, union types, and bounded polymorphism.* Ph.D. Dissertation. Technical Report CMU-CS-91-205. Carnegie Mellon University.

[27] Garrel Pottinger. 1980. A Type Assignment for the Strongly Normalizable λ-terms. In *To H.B. Curry: Essays on Combinatory Logic, Lambda Calculus and Formalism.* Academic Press, 561–577.

[28] John C. Reynolds. 1988. *Preliminary Design of the Programming Language Forsythe.* Report CMU–CS–88–159. Carnegie Mellon University.

[29] Vincent Siles and Hugo Herbelin. 2010. Equality Is Typable in Semi-full Pure Type Systems. In *Proc of LICS.* 21–30.

[30] Claude Stolze and Luigi Liquori and Furio Honsell and Ivan Scagnetto. 2017. Towards a Logical Framework with Intersection and Union Types. In *LFMTP.* ACM, 1–9.

[31] Joe B. Wells and Christian Haack. 2002. Branching Types. In *ESOP (Lecture Notes in Computer Science)*, Vol. 2305. Springer-Verlag, 115–132.