Towards a Logical Framework with Intersection and Union Types

Claude Stolze, Luigi Liquori, Furio Honsell, Ivan Scagnetto

Towards a Logical Framework with Intersection and Union Types*

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

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

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

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

ABSTRACT

We present an ongoing implementation of a dependent-type theory (Δ-framework) based on the Edinburgh Logical Framework LF, extended with *Proof-functional* logical connectives such as intersection, union, and strong (or minimal relevant) implication. Their combination opens up new possibilities of formal reasoning on proof-theoretic semantics. We provide some examples in the extended type theory and we outline a type checker. The theory of the system is under investigation. Once validated *in vitro*, the proof-functional type theory could be successfully plugged into existing truth-functional proof-systems.

ACM Reference Format:
Claude Stolze, Luigi Liquori, Furio Honsell, and Ivan Scagnetto. 2017. Towards a Logical Framework with Intersection and Union Types. In *Proceedings of Logical Frameworks and Meta-Languages: Theory and Practice Affiliated with the Conference on Formal Structures for Computation and Deduction (FSCD 2017), colocated with ICFP 2017, Oxford, UK, September 8, 2017 (LFMTP2017)*, 9 pages.
https://doi.org/10.1145/nnnnnnn.nnnnnnn

1 INTRODUCTION

We extend the Edinburgh Logical Framework (LF) [14] with the *proof-functional* logical connectives of intersection, union, and minimal relevant implication. We call this extension the Δ-framework, since it is based on the Δ-calculus introduced in [9].

Proof-functional connectives take into account the shape of logical proofs, thus allowing polymorphic features of proofs in formulæ to be made explicit. This is in contrast to classical *Truth-functional* connectives where the meaning of a compound formula is only dependent on the truth value of its subformulæ. The aim of this research is to explore the expressiveness of proof-functional operators in a logical framework based on Type Theory. Both Logical Frameworks and proof functional logics consider proofs as first class citizens. But they do it differently, namely explicitly, in the former, while implicitly in the latter. Their combination opens up new possibilities of formal reasoning on proof-theoretic semantics.

Studying the behavior of proof-functional connectives will be beneficial to existing interactive theorem provers, and dependently typed programming languages. Furthermore, we outline the implementation of an experimental proof development environment for this extended theory and we experiment with it.

Intersection types [6] were first introduced as a form of *ad hoc* polymorphism in (pure) λ-calculi *à la* Curry. The paper by Barendregt, Coppo, and Dezani [6] is a classic reference, while [5] is a definitive reference. Union types [2, 21] were later introduced as a dual of intersection by MacQueen, Plotkin, and Sethi [21]: Barbanera, Dezani, and de'Liguoro [2] is a classic reference. The main (non syntax-directed) rules are shown in Figure 1 (left part). Intersection and union types were originally studied as (undecidable) type assignment systems for untyped λ-calculi, *i.e.* for finitary descriptions of denotational semantics.

Intersection and union type theories were also investigated in typed settings, *i.e.* λ-calculi *à la* Church: the programming language Forsythe, by Reynolds [30], is probably the first reference for intersection types: Pierce's dissertation [28], which combines unions and intersections, allows for a more powerful type discipline performing a limited form of abstract interpretation during type checking [17]. Other solutions were proposed by Castagna [13], Wells [32] and Dunfield [11], for example. To illustrate the significance of such type disciplines, we recall Pierce's example in an extension of Forsythe with union-types:

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

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

Dealing with a λ-calculus *à la* Church featuring intersection and union types is problematic: the usual approach of adding types to binders does not work, as the following type assignment derivation

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

LFMTP2017, September 8, 2017, Oxford, UK

ACM ISBN 978-x-xxxx-xxxx-x/YY/MM…$15.00
https://doi.org/10.1145/nnnnnnn.nnnnnnn

$$\frac{x{:}\sigma \in \Gamma}{\Gamma \vdash x : \sigma}\ (Var) \qquad \frac{\Gamma \vdash M : \sigma \to \tau \quad \Gamma \vdash N : \sigma}{\Gamma \vdash M\,N : \tau}\ (App) \qquad \frac{\Gamma, x{:}\sigma \vdash M : \tau}{\Gamma \vdash \lambda x.M : \sigma \to \tau}\ (Abs)$$

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

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

$$\frac{x{:}\sigma \in \Gamma}{\Gamma \vdash x : \sigma}\ (Var) \qquad \frac{\Gamma \vdash M : \sigma \to \tau \quad \Gamma \vdash N : \sigma}{\Gamma \vdash M\,N : \tau}\ (App) \qquad \frac{\Gamma, x{:}\sigma \vdash M : \tau}{\Gamma \vdash \lambda x{:}\sigma.M : \sigma \to \tau}\ (Abs)$$

$$\frac{\Gamma \vdash \Delta_1 : \sigma \quad \Gamma \vdash \Delta_2 : \tau \quad \wr\,\Delta_1\,\wr \equiv \wr\,\Delta_2\,\wr}{\Gamma \vdash \langle \Delta_1\,,\,\Delta_2 \rangle : \sigma \cap \tau}\ (\cap I) \qquad \frac{\Gamma \vdash \Delta : \sigma \cap \tau}{\Gamma \vdash pr_l\,\Delta : \sigma}\ (\cap E_l) \qquad \frac{\Gamma \vdash \Delta : \sigma \cap \tau}{\Gamma \vdash pr_r\,\Delta : \tau}\ (\cap E_r)$$

$$\frac{\Gamma \vdash \Delta : \sigma}{\Gamma \vdash in_l^{\tau}\,\Delta : \sigma \cup \tau}\ (\cup I_l) \qquad \frac{\Gamma \vdash \Delta : \tau}{\Gamma \vdash in_r^{\sigma}\,\Delta : \sigma \cup \tau}\ (\cup I_r) \qquad \frac{\Gamma, x{:}\sigma \vdash \Delta_1 : \rho \quad \Gamma, x{:}\tau \vdash \Delta_2 : \rho \quad \Gamma \vdash \Delta_3 : \sigma \cup \tau \quad \wr\,\Delta_1\,\wr \equiv \wr\,\Delta_2\,\wr}{\Gamma \vdash [\lambda x{:}\sigma.\Delta_1\,,\,\lambda x{:}\tau.\Delta_2]\,\Delta_3 : \rho}\ (\cup E)$$

Figure 1: The type assignment of [2] without subtyping (left) and the type system of the Δ-calculus [9] (right)

shows:

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

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

In [10, 18] and recently [9], two of the present authors proposed the Δ-calculus as an extension of the typed λ-calculus *à la* Church corresponding to the type assignment system *à la* Curry with intersection and union (without subtyping). The relation between Church-style and Curry-style λ-calculi were expressed using an *essence* function, denoted by $\wr - \wr$, that intuitively erases all the types in bound variables and all the truth-functional operators (the full definition is showed in Figure 5)

The Δ-calculus is intended to be a typed calculus corresponding to the type assignment system of [2] (actually without subtyping and ω[1]).

The implemented rules of the Δ-calculus are presented in Figure 1 (right part). The intuition of the current implementation (the metatheory is currently under investigation) is as follows: if $\Gamma \vdash_{[9]} \Delta : \sigma$ then $\Gamma \vdash_{[2]} M : \sigma$ where M is the essence of Δ, *i.e.* $\wr\,\Delta\,\wr \equiv M$, where $\equiv$ is meant to be α-conversion; conversely, if $\Gamma \vdash_{[2]} M : \sigma$ then there exists Δ such that $\Gamma \vdash_{[9]} \Delta : \sigma$ and $M \equiv \wr\,\Delta\,\wr$. Thus, each typable Δ-term of type σ encodes a type assignment derivation of [2] for M of type σ. For example, the Δ-term $\langle \lambda x{:}\sigma.x\,,\,\lambda x{:}\tau.x \rangle$ of type $(\sigma \to \sigma) \cap (\tau \to \tau)$ encodes a precise derivation tree $\mathcal{D}$ for $\lambda x.x$ of the same type; the Δ-term $\lambda x{:}\sigma \cup \tau.[\lambda y{:}\sigma.in_r^{\tau}\,y\,,\,\lambda y{:}\tau.in_l^{\sigma}\,y]\,x$ of type $\sigma \cup \tau \to \tau \cup \sigma$ encodes a precise type assignment derivation tree $\mathcal{D}$ for $\lambda x.x$ of the same type.

The proposition-as-types isomorphism of Curry-Howard is usually utilized to encode the derivations of a logic into a corresponding typed λ-term. For example, $\lambda x{:}A.M : A \to B$ encodes a derivation tree $\mathcal{D}$ for $A \supset B$. It is not immediate to extend this paradigm to the logics underpinning the Δ-calculus which permit to compare the shapes of the derivations and not only to check their validity, *i.e.* the inhabitation of the corresponding type. The logics of the Δ-calculus, called in the literature *Proof-functional*, and their connectives, called proof-functional connectives or simply strong connectives, need to take into account the very structure of proofs, thus implicitly they need to give a first-class status to the latter. Proof-functional connectives are therefore more restrictive than classical *Truth-functional* ones. Hence, because of the Curry-Howard isomorphism, the type rules $(\cap I)$ and $(\cup E)$ correspond to the Gentzen-style inference rules below:

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

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

where $\cap, \cup$ are strong logical connectives and $\equiv$ is a suitable equivalence between logical proofs.

Proof-functional connectives differ from intuitionistic connectives, where only the existence of proof objects matters, and not their actual shape. Proof-functional connectives represent evidence for polymorphic constructions, *i.e.* the same evidence can be used as a proof for different statements.

Pottinger [29] was the first to introduce a proof-theoretic conjunction, which he called *strong conjunction* $\cap$, by requiring something more than the mere existence of constructions proving the left and the right hand side of the conjuncts: namely, if we have a reason to assert A, which is also a reason to assert B, then the same reason asserts $A \cap B$. According to Pottinger: *"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"*. This interpretation makes inhabitants of $A \cap B$ behave as polymorphic evidence for both A and B. This is in contrast with the truth-functional logical operator of intuitionistic conjunction (usually denoted by $\wedge$ or &), stating that *"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 conjunction which does not hold for proof-theoretic conjunction is $(A \supset A) \wedge (A \supset B \supset A)$.

[1] Adding a subtyping relation and a suitable subtype type theory is subject of future work and partially introduced in [19].

Clearly, $(A \supset A) \cap (A \supset B \supset A)$ does not hold, since there is no common evidence for the two conjuncts (this corresponds to equate the two proof inhabitants **I** and **K**); another example, provided by Hindley [16] is $(A \supset A) \cap ((A \supset B \supset C) \supset (A \supset B) \supset A \supset C)$ since this would correspond to equate the two proof inhabitants **I** and **S**: other examples can be found in [29].

Later, Lopez-Escobar [20] presented the first proof-functional logic with strong conjunction as a special case of ordinary conjunction. Mints [23] 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 for both the formula $r_A[M]$ *and* $r_B[M]$.

Also in [9], two of the present authors extended the logical interpretation with union types as another proof-functional operator, the *strong union* $\cup$. Paraphrasing Pottinger's point of view, we could say that the intuitive meaning of $\cup$ is that if we have a reason to assert A (or B), then the same reason will also assert $A \cup B$. This interpretation makes inhabitants of $(A \cup B) \supset C$ be uniform evidence for both $A \supset C$ and $B \supset C$. To emphasize the difference between union and disjunction, consider the example $((A \supset B) \cup B) \supset A \supset B$, which is not derivable since this would again imply the equality of **I** and **K**. Symmetrical to intersection, and extending the Mints' logical interpretation, the logical predicate $r_{A\cup B}[M]$ succeeds if the pure λ-term M is a realizer for either the formula $r_A[M]$ or $r_B[M]$.

Strong (or *Minimal Relevant*) *Implication* $\rightarrow_r$ (or $\supset_r$) is another proof-functional connective: as explained in [3], 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. Relevant implication shares some similarities with singleton-types in Haskell [12]. In [3], relevant implication and intersection were proved to correspond respectively to the implication and conjunction operators of the Meyer and Routley's Minimal Relevant Logic B^+ [22]. Note that, for instance, $A \supset_r B \supset_r A$ is not derivable, therefore $\supset$ and $\supset_r$ are different operators. Relevant implication allows for a natural introduction of *subtyping*, in that $A \supset_r B$ morally means $A \leqslant B$. We do not elaborate further on this issue in this paper.

Encoding proof-functional connectives into a logical framework *à la* LF is not straightforward, because of the subtle side-conditions characterizing proof-functional connectives. However, this can be carried out through a deep encoding as illustrated in Figure 6: this encoding also illustrates the expressive power of a system such as LF which permits proofs as first-class citizens. Pfenning's work on Refinement Types [27] pioneered an extension of the Edinburgh Logical Framework with subtyping and intersection types: our aim is to study extensions of LF featuring full fledged proof-functional logical connectives like strong conjunction and strong disjunction in the presence of subtyping and minimal relevant implication. Also Miquel [24] discusses an extension of the Calculus of Constructions with implicit typing which subsumes a proof functional intersection. However his approach has opposite motivations to ours. While the Δ-calculus provides a Church-style system for the traditional Curry-style versions of type assignment systems. Miquel's Implicit Calculus of Constructions mimics some features of Curry-style systems in an otherwise Church-style Calculus of Constructions. More work needs to be done to understand how the two approaches complement each other. In our system we can discuss also *ad hoc* polymorphism, while in the Implicit Calculus only structural polymorphism is encoded.

As Pottinger said *"it would be of interest to figure out how to add* $\cap$ *(and* $\cup, \rightarrow_r$ *indeed) to intuitionistic logic and then consider the analysis of intuitionistic mathematical reasoning in the light of the resulting system"*: this is one of the aims of the present research. In this paper we introduce an implementation of the Δ-framework *i.e.* an extension of LF with proof-functional operators, allowing for an agile and shallow encoding of proof-functional connectives in presence of dependent-types. The Δ-framework is a direct extension of the Δ-calculus; its metatheory is currently under investigation. Thus the main contributions of the present paper are:

(i) introducing the strong (relevant) implication in the presence of intersection and union types to the Δ-calculus;
(ii) raising the type system of the Δ-calculus to a Δ-logical framework by introducing dependent types *à la* LF to [9];
(iii) encoding the Δ-calculus in plain LF: because of the relations of the Δ-calculus and the type assignment system of [2], this encoding could also be intended as an encoding for the type assignment with intersection and union;
(iv) testing the framework, presenting a quite compact shallow encoding of the Δ-calculus in the Δ-framework;
(v) implementing the type checking algorithm for the Δ-framework and a basic REPL.

The implementation is available at:

https://github.com/cstolze/Bull.

2 LF WITH PROOF-FUNCTIONAL OPERATORS

We could work with the original LF syntax (*i.e.* with abstractions in families) or use the more compact formulation of canonical LF [15] (*i.e.* no abstractions in families): we choose the original one.

The pseudo syntax is given by the following grammar:

Kinds	K	::=	$\text{Type} \mid \Pi x{:}\sigma.K$	as in LF
Families	σ, τ	::=	$a \mid \Pi x{:}\sigma.\tau \mid \sigma\ \Delta \mid$	as in LF
			$\Pi^r x{:}\sigma.\tau \mid$	relevant family
			$\sigma \cap \tau \mid$	intersection
			$\sigma \cup \tau$	union
Objects	Δ	::=	$c \mid x \mid \lambda x{:}\sigma.\Delta \mid \Delta\ \Delta \mid$	as in LF
			$\lambda^r x{:}\sigma.\Delta \mid$	relevant λ
			$\langle \Delta\,,\ \Delta \rangle \mid$	intersection
			$[\Delta\,,\ \Delta] \mid$	union
			$pr_l\ \Delta \mid pr_r\ \Delta \mid$	projections
			$in_l^\sigma\ \Delta \mid in_r^\sigma\ \Delta$	injections

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 the strong conjunction (typed with $\sigma \cap \tau$) with two corresponding projections, the strong disjunction (typed with $\sigma \cup \tau$) with two corresponding

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

Valid Signatures

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

Valid Contexts

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

Figure 2: Valid Signatures and Contexts

Valid Kinds

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

Valid Families

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

$$\frac{\Gamma, x{:}\sigma \vdash_\Sigma \tau : \text{Type}}{\Gamma \vdash_\Sigma \Pi x{:}\sigma.\tau : \text{Type}} \quad (\Pi I) \qquad \frac{\Gamma, x{:}\sigma \vdash_\Sigma \tau : \text{Type}}{\Gamma \vdash_\Sigma \Pi^r x{:}\sigma.\tau : \text{Type}} \quad (\Pi^r I)$$

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

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

Figure 3: Valid Kinds and Families

injections, and the 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. Intersection types can be viewed as constrained product types that are only inhabited by strong pairs, *i.e.* cartesian pairs whose left and right components share a "common infrastructure" (we will formalize this notion precisely).

In a similar way, union types can be viewed as constrained sum types that can only be destructed by strong pairs, *i.e.* the left and right components of the case-operator have to share the same "common infrastructure".

The five LF reductions for objects are:

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

We conjecture that these rules verify the confluence property. Note the similarity between the $\longrightarrow_{in_i}$ rules and the Coq ι-reduction rule. We denote $=$ as the symmetric, reflexive, and transitive closure of all the above reduction rules. The notion of *essence* was introduced in [9] to syntactically connect pure (noted M) and typed (noted Δ) λ-terms: intuitively, this partial function erases all operators corresponding to rules in [2] that are not syntax directed. It is compositional in all other cases and it is defined in Figure 5.

Valid Objects

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

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

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

$$\frac{\Gamma \vdash_\Sigma \Delta_1 : \sigma \quad \Gamma \vdash_\Sigma \Delta_2 : \tau \quad \wr \Delta_1 \wr_\Sigma^\Delta \equiv \wr \Delta_2 \wr_\Sigma^\Delta}{\Gamma \vdash_\Sigma \langle \Delta_1\,,\, \Delta_2 \rangle : \sigma \cap \tau}\ (\cap I)$$

$$\frac{\Gamma \vdash_\Sigma \Delta_1 : \Pi y{:}\sigma.\rho[in_l^\tau\ y/x] \quad \wr \Delta_1 \wr_\Sigma^\Gamma \equiv \wr \Delta_2 \wr_\Sigma^\Gamma \quad \Gamma \vdash_\Sigma \Delta_2 : \Pi y{:}\tau.\rho[in_r^\sigma\ y/x] \quad \Gamma \vdash_\Sigma \Delta_3 : \sigma \cup \tau}{\Gamma \vdash_\Sigma [\Delta_1\,,\, \Delta_2]\, \Delta_3 : \rho[\Delta_3/x]}\ (\cup E)$$

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

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

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

Figure 4: Valid Objects

$$\begin{aligned}
\wr c \wr_\Sigma^\Gamma &\triangleq c \\
\wr x \wr_\Sigma^\Gamma &\triangleq x \\
\wr \lambda x{:}\sigma.\Delta \wr_\Sigma^\Gamma &\triangleq \lambda x.\wr \Delta \wr_\Sigma^\Gamma \\
\wr \lambda^r x{:}\sigma.\Delta \wr_\Sigma^\Gamma &\triangleq \lambda x.\wr \Delta \wr_\Sigma^{\Gamma, x:\sigma} && \text{if } \wr \Delta \wr_\Sigma^{\Gamma, x:\sigma} \equiv x \\
\wr \langle \Delta_1\,,\, \Delta_2 \rangle \wr_\Sigma^\Gamma &\triangleq \wr \Delta_1 \wr_\Sigma^\Gamma && \text{if } \wr \Delta_1 \wr_\Sigma^\Gamma \equiv \wr \Delta_2 \wr_\Sigma^\Gamma \\
\wr [\Delta_1\,,\, \Delta_2] \wr_\Sigma^\Gamma &\triangleq \wr \Delta_1 \wr_\Sigma^\Gamma && \text{if } \wr \Delta_1 \wr_\Sigma^\Gamma \equiv \wr \Delta_2 \wr_\Sigma^\Gamma \\
\wr pr_i\, \Delta \wr_\Sigma^\Gamma &\triangleq \wr \Delta \wr_\Sigma^\Gamma \\
\wr in_i^\sigma\, \Delta \wr_\Sigma^\Gamma &\triangleq \wr \Delta \wr_\Sigma^\Gamma \\
\wr \Delta_1\, \Delta_2 \wr_\Sigma^\Gamma &\triangleq \begin{cases} \wr \Delta_2 \wr_\Sigma^\Gamma & \text{if } \Gamma \vdash_\Sigma \Delta_1 : \Pi^r x{:}\sigma.\tau \\ \wr \Delta_1 \wr_\Sigma^\Gamma \wr \Delta_2 \wr_\Sigma^\Gamma & \text{otherwise} \end{cases}
\end{aligned}$$

Figure 5: The essence function

The extended type theory is a formal system for deriving assertion of the following form:

	$\vdash$	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. The typing rules are syntax-directed, and we conjecture to have both decidability of type reconstruction and unicity of typing. In the rule (*Conv*) we rely on an external notion of equality, which for our purpose is the equality defined previously; using confluence and subject reduction we can guarantee that the equality is decidable.

Another option could be to add an internal notion of equality directly in the type system ($\Gamma \vdash_\Sigma \sigma = \tau$), and prove that the external and the internal definitions of equality are equivalent, as was proved for semi-full PTS [31].

Yet another possibility would be to compare type essences $\wr \sigma \wr = \wr \tau \wr$, for a suitable extension of essence to types and kinds: unfortunately, this would lead to undecidability of type checking, as the following example shows.

About type checking. The following example illustrates the fact that, for any pure λ-term M, you can create a Δ-term such that $\wr \Delta \wr = M$.

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

Because relevant implication is "essentially" the identity, this correspond to including in the type theory axioms such as those

that equate $\sigma \to (\sigma \to \sigma)$ with $(\sigma \to \sigma) \to \sigma$. As a consequence, β-equality of essences is undecidable.

3 EXAMPLES

In order to highlight the advantages of the system we propose the point of view of a user interested in formally reasoning with intersection and union types: we start by comparing a plain LF-encoding of the typing system of Δ-calculus with its representation in our Δ-framework. The former solution represents the constrained way one has to follow to embed intersection and union types in currently available Logical Frameworks. And this is in fact precisely what we do when we give the implementation of the Δ-calculus, only that we do it in a completely transparent way for the user, in the spirit of the Logical Frameworks, which factor out tedious bookkeeping issues.

On the other hand, a system whose metalanguage already provides built-in primitives, compatibles with the underlying logical meaning, is more appealing, since the user can delegate all the cumbersome and error-prone encodings to the metalevel. The aim of this section is to show that intersection and union types allow for a more shallow encoding in the following examples.

Embedding of the Δ-calculus in LF. Figure 6 presents our embedding in LF of the typing system of Δ-calculus in Coq syntax in HOAS. We use HOAS only for commodity reasons: using other abstract syntax representation techniques would not change much the encoding.

The `Eq` predicate plays the same role of the essence function in the Δ-calculus, namely, it allows to encode the fact 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 conclude with the introduction of `(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).

Embedding of Δ-calculus in the Δ-framework. The LF enriched with proof functional operators (intersection, union, and relevant implication) allows for a more shallow encoding because the metalanguage subsumes the source language. More precisely, since each type assignment derivation $\mathcal{D}$ for $\vdash M : \sigma$ (M closed) is isomorphic to a typed Δ-term, we conjecture that our encoding also applies to the type assignment system with intersection and union types, respectively. Let $\to$ and $\to_r$ denote a non-dependent product type Π and a relevant product type Π^r, respectively. The encoding is shown below:

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

This encoding is also available using our concrete syntax (file intersection_union.script).

Figure 6 shows, for comparison, an encoding of the same type assignment system in plain LF: note that it needs the constants used for defining equality of intersection, union and relevant implication ($c_{=abst}$, $c_{=sabst}$, $c_{=app}$, $c_{=sapp}$, $c_{=pr_{1,2}}$, $c_{=in_{1,2}}$, $c_{=sconj}$, and $c_{=sdisj}$) in order to capture equivalence of typed λ-terms having the same essence. In contrast, in the above encoding, intersection, union, and relevant types are implemented in a shallow way, thereby eliminating the need of encoding the essence side conditions via many lines of pure LF code. Thanks to the constants c_{pr_i} and c_{in_i}, we can use the projection and injection operators of the target language in a shallow way, instead of implementing explicitly the four constants. We conjecture that we could also encode the framework in the framework itself.

As an example of the expressivity of the extension, the Pierce's program shown in the Introduction could be encoded as follows:

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

An encoding for the same example in plain LF is presented in Figure 7): note that we have chosen to show the Coq LTAC script generating the function `Is_0_Test`, because the full λ-term is quite big.

Let us try to show now some examples of using the above encoding.

Auto application. Consider the following type assignment derivation:

$$
\dfrac{\dfrac{\dfrac{x{:}\sigma \cap (\sigma \to \tau) \vdash x : \sigma \cap (\sigma \to \tau)}{x{:}\sigma \cap (\sigma \to \tau) \vdash x : \sigma \to \tau} \qquad \dfrac{x{:}\sigma \cap (\sigma \to \tau) \vdash x : \sigma \cap (\sigma \to \tau)}{x{:}\sigma \cap (\sigma \to \tau) \vdash x : \sigma}}{x{:}\sigma \cap (\sigma \to \tau) \vdash x\,x : \tau}}{\vdash \lambda x.x\,x : (\sigma \cap (\sigma \to \tau)) \to \tau}
$$

This derivation is faithfully encoded by the Δ-term

$$\lambda x{:}\sigma \cap (\sigma \to \tau).(pr_r\ x)\,(pr_l\ x),$$

and an encoding in the Δ-framework is

$$
\begin{array}{l}
c_{abst}\ (c_{\cap}\ \sigma\ (c_{\to}\ \sigma\ \tau))\ \tau\ (\lambda x{:}obj\,(c_{\cap}\ \sigma\ (c_{\to}\ \sigma\ \tau)). \\
\qquad c_{app}\ \sigma\ \tau\ (pr_r\ \ (c_{pr_i}\ \sigma\ (c_{\to}\ \sigma\ \tau)\ x))\ (pr_l\ (c_{pr_i}\ \sigma\ (c_{\to}\ \sigma\ \tau)\ x))).
\end{array}
$$

Polymorphic identity. Consider the following type assignment derivation:

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

This derivation is faithfully encoded by the Δ-term

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

```
(* Define our types *)
Axiom o : Set.
(* Axiom omegatype : o. *)
Axioms (arrow relev 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 6: LF encoding of the Δ-calculus (Coq syntax)

```
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; trivial;
     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; [now apply H0|]; eapply Eqtrans; [apply H|apply H1].
 Defined.

End Test.
```

Figure 7: Pierce's example in plain LF (Coq syntax)

```
(* Note that the encoding of a derivation \ala Curry correspond to a typed Delta-term \ala Church *)
(* and the essence of the encodings \ala Church are equal to a pure lambda-term \ala Curry *)

Section Examples.
 Hypotheses s t : o.

 (* type derivation for the judgment \ala Curry
 (* Der : |- lambda x. x x : (s inter (s → t)) → t *)
 (* corresponding to the following derivable judgment \ala Church *)
 (* Judg : |- lambda x:s inter (s → t). (pr2 x) (pr1 x) : (s inter (s → t)) → t *)
 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)).

 (* type derivation for the judgment \ala Curry
 (* Der : |- lambda x. x : (s → s) inter (t → t) *)
 (* corresponding to the following derivable judgment \ala Church *)
 (* Judg : |- (lambda x:s. x) inter (lambda x:t. x) : (s → s) inter (t → t) *)
 Definition id1 : 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)).

 (* type derivation for the judgment \ala Curry *)
 (* Der : |- lambda x. x : (s union t) → (t union s) *)
 (* corresponding to the following derivable judgment \ala Church *)
 (* Judg : |- lambda x:s union t. (lambda y:s. injl y) union (lambda y:t. injr y) x : (s union t) → (t union s) *)
 Definition id2 : 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 (x : OK s) (y : OK t) (pf : Eq s t x y) ⇒ Eqtrans (union t s) t (union t s)
 (Inj_r t s x) y (Inj_l t s y) (Eqtrans (union t s) s t (Inj_r t s x) x y (Eqinj_r t s x) pf)
 (Eqsymm (union t s) t (Inj_l t s y) y (Eqinj_l t s y))))).
End Examples.
```

Figure 8: Examples in plain LF (Coq syntax)

and an encoding in the Δ-framework is

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

Commutativity of union. Consider the following type assignment derivation:

$$\dfrac{\dfrac{x{:}\sigma\cup\tau, y{:}\sigma\vdash y:\sigma}{x{:}\sigma\cup\tau, y{:}\sigma\vdash y:\tau\cup\sigma}\quad\dfrac{x{:}\sigma\cup\tau, y{:}\tau\vdash y:\tau}{x{:}\sigma\cup\tau, y{:}\tau\vdash y:\tau\cup\sigma}\quad x{:}\sigma\cup\tau\vdash x:\sigma\cup\tau}{\dfrac{x{:}\sigma\cup\tau\vdash x:\tau\cup\sigma}{\vdash\lambda x.x:(\sigma\cup\tau)\to(\tau\cup\sigma)}}$$

This derivation is faithfully encoded by the Δ-term

$$\lambda x{:}\sigma\cup\tau.[\lambda y{:}\sigma.in_r^{\tau}\,y\,,\ \lambda y{:}\tau.in_l^{\sigma}\ y]\,x,$$

and an encoding in the Δ-framework is

$$c_{abst}\,(c_{\cup}\,\sigma\,\tau)\,(c_{\cup}\,\tau\,\sigma)\,(\lambda x{:}obj\,(c_{\cup}\,\sigma\,\tau).$$
$$[\lambda y{:}obj\,\sigma.c_{in_i}\,(in_r^{obj\,\tau}\ y)\,,\ \lambda y{:}obj\,\tau.c_{in_i}\,(in_l^{obj\,\sigma}\ y)]\,(c_{sdisj}\,\sigma\,\tau\,x)).$$

4 IMPLEMENTATION DETAILS

Our current implementation uses a small kernel for a Logical Framework featuring union, intersection, and relevant implication. Type reconstruction and type checking algorithms use the pure functional part of the OCaml language: when possible we adopted the design patterns methodology (visitor); parser and lexer are implemented using `ocamllex` and `ocamlyacc`.

A *Read-Eval-Print-Loop* (REPL) 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, as introduced by Berardi in his dissertation [8] and further studied by Barendregt [4] to improve the compactness and the modularity of the kernel. Binders are implemented using de Brujin indexes. We implemented a strong reduction strategy: strong reduction applies whenever two dependent types (or two pure lambda terms in the essence definition) need to be compared. Abstract and concrete syntax are mostly aligned, and the concrete syntax is similar to the concrete syntax of Coq.

Remark. The implementation of $(\cup E)$ type rule is tricky because of the presence of types $\rho[in_l^{\tau}\ y/x]$ and $\rho[in_r^{\sigma}\ y/x]$ in premises requiring to deal with β-expansion and to start a premature kernel interaction with the user (activity usually delegated to the *refiner*). To avoid this, we implemented the following admissible rule:

$$\dfrac{\begin{array}{l}\Gamma\vdash_{\Sigma}\Delta_1:\Pi y{:}\sigma.\rho\,(in_l^{\tau}\ y)\quad \wr\,\Delta_1\,\wr_{\Sigma}^{\Gamma}\equiv\wr\,\Delta_2\,\wr_{\Sigma}^{\Gamma}\\ \Gamma\vdash_{\Sigma}\Delta_2:\Pi y{:}\tau.\rho\,(in_r^{\sigma}\ y)\quad \Gamma\vdash_{\Sigma}\rho:\Pi y{:}(\sigma\cup\tau).\mathrm{Type}\end{array}}{\Gamma\vdash_{\Sigma}[\Delta_1\,,\,\Delta_2]_{\rho}:\Pi x{:}\sigma\cup\tau.\rho\,x}\ (\cup E)_{\mathrm{adm}}$$

5 OUR AGENDA

Our agenda is as follows.

- To prove the "classical" metatheoretic properties of the Δ-framework, namely, strong normalization, confluence, subject reduction and decidability of type checking. Of course,

this process could require some modifications and corrections to the essence definition and/or rules of the typing system currently implemented.
- To integrate in the Δ-framework the minimal (sub)type theory Ξ, as described in [2], to and to add an explicit coercion expression $(\tau)\Delta$ of type τ if $\vdash \Delta : \sigma$ and $\sigma \leqslant \tau$, as described in [19]; this will raise the Δ-framework to the full power of intersection and union types.
- To extend the subtyping algorithm $\mathcal{A}$, for intersection and union types, as introduced in [19], with a dependent type theory: this algorithm (presented in functional style) is conceived to work for the minimal type theory Ξ (*i.e.* axioms 1 to 14, as presented in [2]). It is well known in the literature that subtyping in the presence of dependent-types is delicate: the paper [1] could fix the lines of how to add subtyping in the presence of dependent-types.
- To study the impact of proof-functional operators in *refiners*: when the user must prove *e.g.* a strong conjunction formula $\sigma_1 \cap \sigma_2$ obtaining (mostly interactively) a witness Δ_1 for σ_1, the prototype can "squeeze" the essence M of Δ_1 to accelerate, and in some cases automatize, the construction of a witness Δ_2 proof for the formula σ_2, having the same essence M of Δ_1. Because of this, we conjecture that existing proof-systems could benefit if they are extended with proof-functional operators.
- To explore strong operators, such as intersection and union types, in Martin Löf type theory and in programming with dependent types, as was recently studied in case of parametric quantifiers [25] and in [7, 26] in case of the Implicit Calculus of Constructions, and in Pure Type Systems, respectively.

Finally, note that the proof-functional connectives of the extended framework open interesting issues on the validity or the re-interpretation of the logical principle of proof irrelevance.

Acknowledgment. We are grateful to Daniel Dougherty and Ugo de'Liguoro for fruitful discussions on intersection and union types, to Andreas Nuyts for discussion on programming with dependent (and intersection) types, to the anonymous referees for the useful comments, and to Dave Ritchie for a careful reading of the document.

REFERENCES

[1] David Aspinall and Adriana B. Compagnoni. 2001. Subtyping dependent types. *Theor. Comput. Sci.* 266, 1-2 (2001), 273–309.
[2] Franco Barbanera, Mariangiola Dezani-Ciancaglini, and Ugo de'Liguoro. 1995. Intersection and union types: syntax and semantics. *Inf. Comput.* 119, 2 (1995), 202–230.
[3] Franco Barbanera and Simone Martini. 1994. Proof-functional connectives and realizability. *Archive for Mathematical Logic* 33 (1994), 189–211.
[4] Henk Barendregt. 1991. Introduction to generalized type systems. *Journal of Functional Programming* 1, 2 (1991), 125–154.
[5] H. 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] Bruno Barras and Bruno Bernardo. 2008. The Implicit Calculus of Constructions as a Programming Language with Dependent Types. In *FOSSACS (Lecture Notes in Computer Science)*, Vol. 4962. Springer, 365–379.
[8] Stefano Berardi. 1990. *Type dependence and Constructive Mathematics.* Ph.D. Dissertation. University of Turin.
[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 *Proceedings of the 17th ACM SIGPLAN International Conference on Functional Programming (ICFP '12)*. ACM, 17–28.
[12] Richard A. Eisenberg and Stephanie Weirich. 2012. Dependently typed programming with singletons. In *Proc. ACM SIGPLAN Symposium on Haskell*. 117–130.
[13] 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.
[14] Robert Harper, Furio Honsell, and Gordon Plotkin. 1993. A Framework for Defining Logics. *J. ACM* 40, 1 (1993), 143–184.
[15] Robert Harper and Daniel R. Licata. 2007. Mechanizing Metatheory in a Logical Framework. *Journal of Functional Programming* 17, 4–5 (July 2007), 613–673.
[16] J. Roger Hindley. 1984. Coppo-Dezani Types do not Correspond to Propositional Logic. *Theor. Comput. Sci.* 28 (1984), 235–236.
[17] Thomas P. Jensen. 1992. Disjunctive Strictness Analysis. In *Proc of LICS*. 174–185.
[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] Andreas Nuyts, Andrea Vezzosi, and Dominique Devriese. 2017. Parametric quantifiers for dependent type theory. In *ACM SIGPLAN ICFP*, Vol. 1. To appear.
[26] Nathan Mishra-Linger and Tim Sheard. 2008. Erasure and Polymorphism in Pure Type Systems. In *FOSSACS (Lecture Notes in Computer Science)*, Vol. 4962. Springer, 350–364.
[27] Frank Pfenning. 1993. Refinement types for logical frameworks. In *TYPES*. 285–299.
[28] 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.
[29] 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.
[30] John C. Reynolds. 1988. *Preliminary Design of the Programming Language Forsythe.* Report CMU–CS–88–159. Carnegie Mellon University.
[31] Vincent Siles and Hugo Herbelin. 2010. Equality Is Typable in Semi-full Pure Type Systems. In *Proc of LICS*. 21–30.
[32] Joe B. Wells and Christian Haack. 2002. Branching Types. In *ESOP (Lecture Notes in Computer Science)*, Vol. 2305. Springer-Verlag, 115–132.