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, LFΔ, 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 formulae. 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 subformula. 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 possibilities of formal reasoning on proof-theoretic semantics. We study the metatheory of LFΔ 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:

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 (LFΔ)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

¹Work supported by the COST Action CA15123 EUTYPES "The European research network on types for programming and verification".
²Δ stands for "type Decorated term", although in Greek we should have rather called it the K-framework.

made explicit in formulæ. This differs from classical or intuitionistic connectives where the meaning of a compound formula is only dependent on the truth value or the provability of its subformula.

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:

\[
\begin{align*}
D_1 : A &\quad D_2 : B &\quad D_1 \equiv D_2 \\
A \land B
\end{align*}
\]

\[
\begin{align*}
D_1 : A \supset C &\quad D_2 : B \supset C &\quad A \lor B &\quad D_1 \equiv D_2 \\
C
\end{align*}
\]

where \(\equiv\) is a suitable equivalence between logical proofs.

Notice that the above rules suggest immediately intriguing applications in polymorphic constructions, i.e. the same evidence can be used as a proof for different statements.

Pottinger, [27], was the first to study the strong connective \(\land\). He contrasted it to the intuitionistic connective \(\wedge\) as follows: "The intuitive meaning of \(\land\) can be explained by saying that to assert \(A \land B\) is to assert that one has a reason for asserting \(A\) which is also a reason for asserting \(B\), while to assert \(A \land B\) 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 \((A \supset A) \land (A \supset B) \supset A\). Clearly, \((A \supset A) \land (A \supset B) \equiv A\) does not hold otherwise there should exist a closed \(\lambda\) term having simultaneously only one and at least two abstractions. Lopez-Escobar [20] and Mints [23] investigated extensively logics featuring both strong and
intuitionistic 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 intuitionistic type \( A \supset C \) and \( B \supset C \) is not derivable if \( \vee \) is replaced by \( \cup \). Otherwise there would exist a term which behaves both as \( I \) and as \( K \).

Strong (or Minimal Relevant) Implication \( \rightarrow_r \) 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 \( \rightarrow_r \) differ, since \( A \rightarrow_r B \rightarrow_r A \) is not derivable. Relevant implication allows for a natural introduction of subtyping, in that \( A \supset_r B \) morally means \( A \leq B \). 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 \emph{intersection} and \emph{union type assignment systems}. \emph{Intersection types} were introduced by Coppo, Dezani and Barendregt in the early 80’s [6] to support a form of \emph{ad hoc} polymorphism, for untyped \( \lambda \)-calculi, \( \lambda \) a Curry. Intersection types were used originally as an (undecidable) type assignment system for untyped \( \lambda \)-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. \emph{Union types} were introduced later, semantically, by MacQueen, Plotkin, and Sethi, [3, 21], again \( \lambda \) a 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. \( \lambda \) a 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:

\[
\begin{align*}
\text{Test} & \equiv b \text{ if } b \text{ then } 1 \text{ else } 1 : \text{Pos} \cup \text{Neg} \\
\text{Is}_0 & : (\text{Neg} \to F) \land (\text{Zero} \to T) \land (\text{Pos} \to F) \\
(\text{Is}_0\text{ Test}) & : F
\end{align*}
\]

Without union types the best information we can get for (Is\_0 Test) is a Boolean type.

Designing a \( \lambda \)-calculus \( \lambda \) a 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:

\[
\begin{align*}
\vdash x : \sigma & \vdash x : \sigma \\
\vdash \lambda x : \sigma. x : \sigma & \to \sigma \\
\vdash \lambda x : \sigma. x : \sigma \to \tau & \to \tau \\
\vdash \lambda x : \sigma. x : \sigma \to (\sigma \to \tau) & \to (\sigma \to \tau)
\end{align*}
\]

Liquori et al. were the first, [10, 18], to extend the theory of type assignment systems, \emph{i.e.} type disciplines \( \lambda \) a Church, to explicitly typed calculi, \emph{i.e.} type disciplines \( \lambda \) a Church, for all strong connectives. In [9], two of the present authors proposed the \( \Delta \)-calculus as a typed \( \lambda \)-calculus \( \lambda \) a Church corresponding to the type assignment system \( \lambda \) a Curry with intersection, union and subtyping, but without \( \omega \). The relation between Church-style and Curry-style \( \lambda \)-calculus was expressed using an \emph{essence} function, denoted by \( t \rightarrow \iota \), that intuitively erases all the type information in terms (the full definition is shown in Figure 5). The \( \Delta \)-calculus is a typed calculus corresponding to the type assignment system of [3], without \( \omega \) 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 \( \Delta \)-framework LF\( _\Delta \), which extends LF and features a full-fledged strong conjunction, strong disjunction, and minimal relevant implication, whereby all strongly normalizing \( \Delta \)-terms have a computational counterpart.

**Figure 1:** The type assignment of [3] and the subtype theory \( \Xi \).
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 allows 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, 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 \( \lambda x : \sigma \cdot (\sigma \cap \tau) \rightarrow \sigma \cap (\rho \rightarrow \rho) \) 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.

In this paper we outline the implementation of an experimental proof development environment for LF 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 meta-theory of LF, 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 \( \Delta \)-FRAMEWORK: LF WITH PROOF-FUNCTIONAL OPERATORS

We could formulate LF with 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 pseudo-terms is given by the following grammar:

\[
\begin{align*}
\text{Kinds} & \quad K ::= \text{Type} | \Pi \times \sigma. K & \quad \text{as in LF} \\
\text{Families} & \quad \sigma, \tau ::= a | \Pi \times \sigma. \tau | \sigma \Delta | \Pi' \times \sigma. \tau | \sigma \cap \tau | \sigma \cup \tau & \quad \text{as in LF} \\
\text{Objects} & \quad \Delta ::= c | x \cdot \lambda x : \sigma. \Delta | \Delta \Delta | \lambda x : \sigma. \Delta | \Delta \Delta | \Delta \Delta & \quad \text{as in LF} \\
& \quad \Delta \Delta & \quad \text{relevant abstraction} \\
& \quad \Delta \Delta & \quad \text{relevant application} \\
& \quad \Delta \Delta & \quad \text{intersection objects} \\
& \quad \Delta \Delta & \quad \text{union objects} \\
& \quad \Delta \Delta & \quad \text{projections objects} \\
& \quad \Delta \Delta & \quad \text{injections objects}
\end{align*}
\]

General terms, namely kinds, families, and objects are denoted by \( U \), and \( V \). For the sake of simplicity, we suppose that \( a \)-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) \( \lambda \)-abstraction (typed with \( \Pi'' \)). Note that injections \( in_i \) need to be decorated with the injected type \( \sigma \) 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 \( \lambda \)-terms (denoted by \( M \)) and type annotated LF terms (denoted by \( \Delta \)). 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 \( \langle \Delta_1, \Delta \rangle \) and \( \{ \Delta_1, \Delta \} \) to have congruent components “up-to” erasure of type annotations. This is achieved by imposing that the \( \iota \Delta_i \equiv \iota \Delta_i \) 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.

\[
\begin{align*}
\iota c & \equiv c \\
\iota x & \equiv x \\
\iota \lambda x : \sigma. \Delta & \equiv \lambda x : \iota \Delta \\
\iota \lambda x : \sigma. \Delta & \equiv \lambda x : \iota \Delta \\
\iota \langle \Delta_1, \Delta_2 \rangle & \equiv \iota \Delta_1 & i = 1, 2 \\
\iota \{ \Delta_1, \Delta_2 \} & \equiv \iota \Delta_1 & i = 1, 2 \\
\iota \{ \iota \Delta_1 \} & \equiv \iota \Delta_1 \\
\iota \{ \iota \Delta_2 \} & \equiv \iota \Delta_2 \\
\iota \Delta_1 \Delta_2 & \equiv \iota \Delta_1 \iota \Delta_2 \\
\iota \Delta_1 & \equiv \iota \Delta_2 \\
\iota \Delta_2 & \equiv \iota \Delta_1 \\
\iota \Delta_1 \Delta_2 & \equiv \iota \Delta_2 \iota \Delta_1
\end{align*}
\]

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 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 \( \equiv \Delta \) the symmetric, reflexive, and transitive closure of \( \rightarrow \Delta \), 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 \( \rightarrow \Delta \) is locally confluent:

**Proposition 2.1.** The reduction relation on well-formed LF terms is locally confluent.
Let \( \Gamma \vdash \{ x_1: \sigma_1, \ldots, x_n: \sigma_n \} \) (\( i \neq j \) implies \( x_i \neq x_j \)), and \( \Gamma, x: \sigma \vdash \Gamma \cup \{ x: \sigma \} \).

Let \( \Sigma \vdash \{ c_1: \sigma_1, \ldots, c_n: \sigma_n \} \), and \( \Sigma, c: \sigma \vdash \Sigma \cup \{ c: \sigma \} \).

Valid Signatures

\[
\begin{array}{ccc}
\Sigma \text{ sig} & \vdash \Sigma K & a \notin \text{dom}(\Sigma) \\
\Sigma, a: K \text{ sig} & \vdash \Sigma \sigma & c \notin \text{dom}(\Sigma) \\
\end{array}
\]

Valid Contexts

\[
\begin{array}{c}
\Sigma \text{ sig} \\
\vdash \Sigma (\epsilon) \\
\end{array}
\]

Figure 2: Valid Signatures and Contexts

Valid Kinds

\[
\begin{array}{c}
\Gamma \vdash \Sigma \Gamma \\
\vdash \Sigma \text{ Type} \\
\end{array}
\]

\[
\begin{array}{c}
\Gamma, x: \sigma \vdash \Sigma K \\
\vdash \Sigma \Pi x: \sigma.K \\
\end{array}
\]

Valid Families

\[
\begin{array}{c}
\Gamma, x: \sigma \vdash \Sigma \tau: \text{ Type} \\
\Gamma \vdash \Sigma \Pi x: \sigma, \tau: \text{ Type} \\
\end{array}
\]

\[
\begin{array}{c}
\Gamma, x: \sigma \vdash \Sigma \Delta: \tau \\
\Gamma \vdash \Sigma \Delta: \tau \\
\end{array}
\]

\[
\begin{array}{c}
\Gamma, x: \sigma \vdash \Sigma \tau: \text{ Type} \\
\Gamma \vdash \Sigma \tau: \text{ Type} \\
\end{array}
\]

\[
\begin{array}{c}
\Gamma \vdash \Sigma \sigma \cap \tau: \text{ Type} \\
\end{array}
\]

\[
\begin{array}{c}
\Gamma \vdash \Sigma \sigma \cup \tau: \text{ Type} \\
\end{array}
\]

Figure 3: Valid Kinds and Families

The extended type theory \( LF_{\Delta} \) is a formal system for deriving judgements of the following forms:

\( \vdash \sigma \)

\( \sigma \) is a valid signature

\( \vdash \Sigma \Gamma \)

\( \Gamma \) is a valid context in \( \Sigma \)

\( \vdash \Sigma K \)

\( K \) is a kind in \( \Gamma \) and \( \Sigma \)

\( \vdash \Sigma \sigma: K \)

\( \sigma \) has kind \( K \) in \( \Gamma \) and \( \Sigma \)

\( \vdash \Sigma \Delta: \sigma \)

\( \Delta \) has type \( \sigma \) in \( \Gamma \) and \( \Sigma \)

The set of rules are defined in Figures 2, 3, and 4. They are syntax-directed. In the rule (\( \text{Conv} \)) 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 \( \iota \sigma \iota =_\Delta \iota \tau \iota \), 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 \( \lambda \)-term \( M \), one can define a \( \Delta \)-term such that \( \iota \Delta \iota =_\beta M \).

\[
\begin{array}{c}
c_1 \ : \ \Pi x: \sigma. (\Pi y: \sigma. \sigma) \\
c_2 \ : \ \Pi x: \Pi y: \sigma. \sigma \\
\Omega \ : \ \lambda (\lambda x: \sigma. c_1 : x) \cdot (c_2 : (\lambda x: \sigma. c_1 : x) : x) : \sigma \\
\iota \Omega \iota \ : \ (\lambda x: x) \cdot (\lambda x: x) \\
\end{array}
\]

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 \( \sigma \) and \( \sigma \rightarrow \sigma \). As a consequence, \( \beta \)-equality of essences becomes undecidable. We will
Valid Objects

\[ \frac{\Gamma, c : e \in \Sigma}{\Gamma \vdash c : e} \quad \text{(Const)} \]
\[ \frac{\Gamma \vdash \Delta \vdash r}{\Gamma, x : \sigma, \Delta : \Pi \vdash \lambda x. \sigma, \Delta : \Pi \vdash \sigma, x} \quad \text{(II)} \]
\[ \frac{\Gamma, x : \sigma, \Delta : \tau \vdash \iota \Delta \equiv \Delta \iota}{\Gamma \vdash \lambda x. \sigma, \Delta : \Pi \vdash \sigma, x} \quad \text{(II') } \]
\[ \frac{\Gamma \vdash \Delta_1 : \sigma \quad \Gamma \vdash \Delta_2 : \tau \quad \Delta_1 \equiv \Delta_2 \iota}{\Gamma \vdash \Delta_1 \cdot \Delta_2 : \sigma \cap \tau} \quad \text{(\&I)} \]
\[ \frac{\Gamma \vdash \Delta : \sigma \cap \tau}{\Gamma \vdash \Gamma \vdash \Delta \vdash \sigma \cap \tau} \quad \text{(\&E \i)} \]
\[ \frac{\Gamma \vdash \Delta : \sigma \quad \Gamma \vdash \sigma \cup \tau : \text{Type}}{\Gamma \vdash \in_1^\tau \Delta : \sigma \cup \tau} \quad \text{(U\&I)} \]
\[ \frac{\Gamma \vdash \Delta : \sigma \quad \Gamma \vdash \sigma \cup \tau : \text{Type}}{\Gamma \vdash \in_1^\tau \Delta : \sigma \cup \tau} \quad \text{(U\&E \i)} \]
\[ \frac{\Gamma \vdash \Delta : \tau}{\Gamma \vdash \Gamma \vdash \Delta : \tau} \quad \text{(Conv)} \]

Figure 4: Valid Objects

\[ (\lambda x. \Delta_1) \Delta_2 \rightarrow_\beta \Delta_1[\Delta_2/x] \]
\[ pr_1(\Delta_1, \Delta_2) \rightarrow_{pr_1} \Delta_1 \]
\[ pr_r(\Delta_1, \Delta_2) \rightarrow_{pr_r} \Delta_2 \]
\[ [\Delta_1, \Delta_2] \in_1^\tau \Delta_3 \rightarrow_{in_1} \Delta_1 \Delta_3 \]
\[ [\Delta_1, \Delta_2] \in_1^\tau \Delta_3 \rightarrow_{in_r} \Delta_2 \Delta_3 \]
\[ (\lambda' x. \sigma, \Delta_1) \cdot \Delta_2 \rightarrow_{\beta r} \Delta_1[\Delta_2/x] \]
\[ \Delta_1 \rightarrow_{\Delta_1'} \Delta_2 \rightarrow_{\Delta_2'} \Delta_1' \equiv \Delta_2' \iota \iota \rightarrow_{\Delta_1''} \Delta_1' \Delta_2' \rightarrow_{\Delta_2''} \Delta_1' \equiv \Delta_2' \iota \iota \rightarrow_{\Delta_1'''} \Delta_1' \Delta_2' \rightarrow_{\Delta_2'''} \Delta_1' \equiv \Delta_2' \iota \iota \rightarrow_{\Delta_1''''} \Delta_1' \Delta_2' \rightarrow_{\Delta_2''''} \Delta_1' \equiv \Delta_2' \iota \iota \rightarrow_{\Delta_1'''''} \Delta_1' \Delta_2' \rightarrow_{\Delta_2'''''} \Delta_1' \equiv \Delta_2' \iota \iota \rightarrow_{\Delta_1''''''} \Delta_1' \Delta_2' \rightarrow_{\Delta_2''''''} \Delta_1' \equiv \Delta_2' \iota \iota \rightarrow_{\Delta_1'''''''} \Delta_1' \Delta_2' \rightarrow_{\Delta_2'''''''} \Delta_1' \equiv \Delta_2' \iota \iota \rightarrow_{\Delta_1''''''''} \Delta_1' \Delta_2' \rightarrow_{\Delta_2''''''''} \Delta_1' \equiv \Delta_2' \iota \iota \rightarrow_{\Delta_1''''''''' \text{ (Congr$_r$)}} \]
\[ \Delta_1 \rightarrow_{\Delta_1'} \Delta_2 \rightarrow_{\Delta_2'} \Delta_1' \equiv \Delta_2' \iota \iota \rightarrow_{\Delta_1''} \Delta_1' \Delta_2' \rightarrow_{\Delta_2''} \Delta_1' \equiv \Delta_2' \iota \iota \rightarrow_{\Delta_1'''} \Delta_1' \Delta_2' \rightarrow_{\Delta_2'''} \Delta_1' \equiv \Delta_2' \iota \iota \rightarrow_{\Delta_1''''} \Delta_1' \Delta_2' \rightarrow_{\Delta_2''''} \Delta_1' \equiv \Delta_2' \iota \iota \rightarrow_{\Delta_1''''} \Delta_1' \Delta_2' \rightarrow_{\Delta_2''''} \Delta_1' \equiv \Delta_2' \iota \iota \rightarrow_{\Delta_1'''''} \Delta_1' \Delta_2' \rightarrow_{\Delta_2''''' \text{ (Congr$_r$)}} \]

Figure 6: Reduction semantics

have to rule out such options in relating relevant implications in LF$_\Delta$ to subtypes in the type assignment system $B$ of [3].

2.1 Relating LF$_\Delta$ to $B$

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

As pointed out in [3], the elimination rule for union types in $B$ breaks subject reduction for one-step $\beta$-reduction, but this can be recovered using a suitable parallel $\beta$-reduction. The well-known counter-example for one-step reduction, due to Pierce, is the following:

\[ x ((\lambda y. z) ((\lambda y. z) y) z) \rightarrow_{\beta} x ((\lambda y. z) y) z \rightarrow_{\beta} x ((\lambda y. z) y) z, \]

where $I$ is the identity. In the typing context $B = x : (\sigma_1 \rightarrow \sigma_1 \rightarrow \tau) \cap (\sigma_2 \rightarrow \sigma_2 \rightarrow \tau), y : \rho \rightarrow \rho, z, x, \rho, \sigma_1, \sigma_2, \tau$, the first and the last terms can be typed with $\tau$, 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
\[
\begin{align*}
B, x_1 : \sigma, \ldots, x_n : \tau & \vdash M : \rho \\
B, x_1 : \tau, \ldots, x_n : \tau & \vdash M : \rho \\
B \vdash N_i : \sigma \cup \tau & \quad i = 1 \ldots n \\
\end{align*}
\]
(\(\cup E'\))
In LF\(_\Delta\) 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 \(in_1, in_x\) 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 disjunction.

Pierce's critical term can be expressed in LF\(_\Delta\) only in the form
\[
[(\lambda x_1 : \sigma_1.(pr_1 x) x_1 x_1), (\lambda x_2 : \sigma_2.(pr_2 x) x_2 x_2)]((\lambda x_3 : \rho'. x_3) y) z
\]
for \(\alpha: \sigma_1 \cup \sigma_2 \rightarrow \sigma_1 \cup \sigma_2 \rightarrow \text{Type}\) and \(\Pi x_1 : \sigma_1. a(in_i^{\sigma_1} x_1) (in_i^{\sigma_2} x_1) \Pi x_2 : \sigma_2(a(in_i^{\sigma_1} x_2)(in_i^{\sigma_2} x_2)y' z', \rho', \rho, \rho' \equiv \rho \rightarrow (\sigma_1 \cup \sigma_2)).\)
There is only one reducible, namely \(\Delta_3\), and the reduction of this redex leads to \([\Delta_1, \Delta_2](y, z)\) and no other intermediate (untypable) \(\Delta\)-terms are possible.

The following result will be useful in the following section.

**Theorem 2.2.** The systems \(B\) and \(B'\), 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\(_\Delta\) metatheory

LF\(_\Delta\) 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 \(\alpha\) be either \(\sigma : K\) or \(\Delta : \alpha\). Then:

1. **Weakening:** if \(\Gamma \vdash \alpha \Sigma \Gamma, \Gamma'\), then \(\Gamma, \Gamma' \vdash \Sigma \alpha\).
2. ** Strengthening:** if \(\Gamma, x, \alpha \Sigma \Gamma, \alpha\), then \(\Gamma, \Gamma' \vdash \Sigma \alpha\), provided that \(x \notin FV(\Gamma') \cup FV(\alpha)\).
3. **Transitivity:** if \(\Gamma, x, \sigma \Sigma \Gamma, \sigma \Sigma \alpha\), then \(\Gamma, \Gamma'[\Delta/x] \vdash \Sigma \alpha\).
4. **Permutation:** if \(\Gamma, x_1 : \sigma_1 \Sigma \Gamma', x_2 : \tau \Sigma \alpha\), then \(\Gamma, x_1, x_2 : \tau, \Gamma', x_1 : \sigma_1 \Sigma \alpha\), provided that \(x_1\) does not occur free in \(\Gamma'\) or in \(\tau\), and that \(\tau\) is valid in \(\Gamma\).

**Theorem 2.4 (Unicity of Types and Kinds).**

1. If \(\Gamma \vdash \alpha \Sigma \Delta, \alpha\) and \(\Gamma \vdash \alpha \Sigma \Delta, \tau\), then \(\sigma = \Delta \tau\).
2. If \(\Gamma \vdash \alpha \Sigma \Delta, \alpha\) and \(\Gamma \vdash \alpha \Sigma \Delta, \Sigma\), then \(K = \Delta \Sigma\).

**Strong Normalization.** In order to prove strong normalization we will map LF\(_\Delta\)-terms into \(B\)-terms taking advantage of Theorem 2.2.

**Definition 2.5.** Let the forgetful mappings \(\|\|\) and \(\|\|\) 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 \(\Gamma \vdash \Sigma \Delta_1, \Delta_2\) : \(\sigma \cap \tau\) is provable then \(\|\Delta_1\| = \|\Delta_2\|\) and similarly if \(\Gamma \vdash \Sigma \Delta_1, \Delta_2\) : \(\Pi x : \sigma \cap \tau, \rho(x)\) then \(\|\Delta_1\| = \|\Delta_2\|\).

The following lemmas are proved by straightforward structural induction.

**Lemma 2.7.**

1. If \(\sigma = \alpha \tau\), then \(\|\sigma\| = \|\alpha\| \|\tau\|\).
2. If \(K = \alpha \beta\), then \(\|K\| = \|\alpha\| \|\beta\|\).

**Lemma 2.8.**

1. If \(\|\Delta\| = \|\alpha\| \|\tau\|\).
2. If \(\|\Delta\| = \|\alpha\| \|\beta\|\).

**Lemma 2.9.**

1. If \(\Gamma \vdash \Sigma \sigma : K\), then \(\|\Gamma\| = \|\Sigma\| = \|\sigma\| = \|K\|\).
2. If \(\Gamma \vdash \Sigma \Delta : \sigma\), then \(\|\Gamma\| = \|\Sigma\| = \|\Delta\| = \|\sigma\|\).

where \(\vdash \Sigma\) denotes the type system \(B\), augmented by the infinite set of axioms for kinds (\(K\)) \(c_\sigma : \omega \rightarrow (\sigma \rightarrow \omega) \rightarrow \omega\), for each type \(\sigma\).

**Lemma 2.10.**

1. If \(\alpha \rightarrow \beta, \tau\), then \(\|\alpha\| = \|\beta\| \|\tau\|\).
2. If \(\Delta_1 \rightarrow \beta, \Delta_2\), then \(\|\Delta_1\| = \|\beta\| \|\Delta_2\|\).

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

**Theorem 2.11.**

1. The LF\(_\Delta\) is strongly normalizing, i.e.,
   1. If \(\Gamma \vdash \Sigma \alpha : K\), then \(K\) is strongly normalizing.
   2. If \(\Gamma \vdash \Sigma \alpha : K\), then \(\alpha\) is strongly normalizing.
   3. If \(\Gamma \vdash \Sigma \Delta : \alpha\), then \(\Delta\) is strongly normalizing.

2. Every strongly normalizing term can be type-annotated so as to be the essence of a \(\Delta\)-term.

**Proof.** 1) Strong normalization derives directly from Lemma 2.10 and Theorem 2.2.
2) Use rules (\(\cap E\)) and (\(\cap I\)) to encode the SN terms of \(\lambda\)-calculus inductively defined by: \(\lambda_1 \ldots \lambda_n \in SN \Rightarrow \lambda_1 \ldots \lambda_n \in SN\) and \(\Delta(\lambda' / x) \Delta_1 \ldots \Delta_n \in SN\) and \(\Delta' \in SN \Rightarrow (\lambda x. \sigma. \Delta') \Delta_1 \ldots \Delta_n \in SN\)

Incidentally, we point out that to our knowledge LF\(_\Delta\) 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\(_\Delta\) is confluent, i.e.,

1. If \(K_1 \rightarrow^* K_2\) and \(K_1 \rightarrow^* K_3\), then there exists an \(K_4\) such that \(K_2 \rightarrow^* K_4\) and \(K_3 \rightarrow^* K_4\).
2. If \(\sigma_1 \rightarrow^* \sigma_2\) and \(\sigma_1 \rightarrow^* \sigma_3\), then there exists a \(\sigma_4\) such that \(\sigma_2 \rightarrow^* \sigma_4\) and \(\sigma_3 \rightarrow^* \sigma_4\).
3. If \(\Delta_1 \rightarrow^* \Delta_2\) and \(\Delta_1 \rightarrow^* \Delta_3\), then there exists a \(\Delta_4\) such that \(\Delta_2 \rightarrow^* \Delta_4\) and \(\Delta_3 \rightarrow^* \Delta_4\).

The following lemmas are proved by structural induction.

**Lemma 2.13 (Inversion Properties).**
The $\Delta$-framework

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

Theorem 2.15 (Subject reduction of LF).
Finally, we define a possible algorithm for checking judgements in $\text{LF}_\Delta$ by computing a type or a kind for a term, and then testing for definitional equality, i.e., $\equiv_\Delta$, 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 $\alpha$-conversion. Therefore we finally have:

\[ \text{Figure 7: The forgetful mappings } \vdash \text{ and } \cdot. \]

Theorem 2.16 (Decidability). All the type judgments of $\text{LF}_\Delta$ 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 $\Sigma$ of [3], without rule (10); we call $\Sigma'$ the resulting set.

Theorem 2.17 (Type Inclusion). The judgement $\vdash \Gamma \rightarrow \Delta$ holds iff $\sigma \subseteq \sigma'$ holds in the subtype theory $\Sigma'$ of $B$.

Proof. For the “only if” part, it is possible to write a “type-annotated” term whose essence is an $\eta$-expansion of the identity corresponding to each of the axioms and rules in $\Sigma$. The converse follows from Lemma 2.13.

To encompass also rule (10) of the subtype theory $\Sigma$, we need to strengthen rule (P') taking essence up to $\beta$-equality and use the term $\lambda x.\sigma(\rho(z))$, $\lambda x.\sigma(\rho(x), z)$, or $\lambda x.\sigma(\rho(x, z))$, for some $\tau$.

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

3 EXAMPLES

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

The intuition behind these encodings, which can be substantiated by an appropriate adequacy result, is the following: in the given signatures, if $\Gamma \vdash \Delta : \alpha$ then there exist $M, B, \sigma'$ such that $B \vdash_M \sigma'$ where $M, B, \sigma'$ are the “decodings” respectively of $\Delta, \Gamma$, and $\sigma$, and conversely, if $B \vdash_M \sigma'$ then there exist $\Delta, \sigma$, and $\Gamma$, i.e. “encodings” of $M, B$ and $\sigma$ respectively, such that $\Gamma \vdash \Delta : \sigma$.

Thus, in the appropriate signature and context, each $\text{LF}_\Delta$-typable term $\Delta$ of type encoding of $\sigma$ encodes a type assignment derivation in $B$, i.e. the decoding of $\Delta$, of type $\sigma$.

The $\text{LF encoding of } L$. Figure 8 presents a pure $\text{LF}$ encoding of $L$ in Coq syntax using HOAS. We use HOAS in order to take advantage
of the higher order features of the frameworks: other abstract syntax representation techniques would not be much different, but more verbose. The Eq predicate plays the same role of the essence function in LFs, namely, it encodes the judgement that two proofs (i.e., two types of terms of type \( \text{OK} \)) 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 \( \text{OK} \) terms of type \( \text{OK} \).

Figure 8: LF encoding of \( \mathcal{L} \) (Coq syntax)
Three base types: atomic propositions, non-atomic goals and non-atomic programs.

\[ \alpha, \gamma_0, \pi_0 : \text{Type} \]

Goals and programs are defined from the base types.

\[ \gamma = \alpha \lor \gamma_0 \quad \pi = \alpha \lor \pi_0 \]

Constructors (implication, conjunction, disjunction).

\[ \text{impl} : (\pi \rightarrow \gamma \rightarrow \gamma_0) \cap (\gamma \rightarrow \pi \rightarrow \pi_0) \]
\[ \text{impl}_1 = \lambda x: \pi. \lambda y: \gamma. \text{in}^\pi_1 (\text{pr}_1 \text{impl} x y) \]
\[ \text{impl}_2 = \lambda x: \pi. \lambda y: \gamma. \text{in}^\pi_2 (\text{pr}_2 \text{impl} x y) \]
\[ \text{and} : (\pi \rightarrow \gamma \rightarrow \gamma_0) \cap (\pi \rightarrow \pi \rightarrow \pi_0) \]
\[ \text{and}_1 = \lambda x: \pi. \lambda y: \gamma. \text{in}^\pi_1 (\text{pr}_1 \text{and} x y) \]
\[ \text{and}_2 = \lambda x: \pi. \lambda y: \gamma. \text{in}^\pi_2 (\text{pr}_2 \text{and} x y) \]
\[ \text{or} : (\gamma \rightarrow \gamma \rightarrow \gamma_0) \cap (\gamma \rightarrow \pi \rightarrow \pi_0) \]
\[ \text{or}_1 = \lambda x: \gamma. \lambda y: \gamma. \text{in}^\gamma_1 (\text{pr}_1 \text{or} x y) \]
\[ \text{or}_2 = \lambda x: \gamma. \lambda y: \gamma. \text{in}^\gamma_2 (\text{pr}_2 \text{or} x y) \]

Solve \( p \rightarrow q \) indicates that the judgment \( p \rightarrow q \) is valid. and \( \text{bchain} \ p \rightarrow q \) indicates that, if \( p \rightarrow q \) is valid, then \( p \rightarrow a \) is valid.

Solve:

\[ \pi : \gamma \rightarrow \text{Type} \quad \text{bchain} : \pi \rightarrow \alpha \rightarrow \gamma \rightarrow \text{Type} \]

Rules for solve:

\[ \Pi_1 \rightarrow \Pi_2 \rightarrow \Pi_3 \rightarrow \Pi_4 \rightarrow \Pi_5 \rightarrow \Pi_6 \rightarrow \text{solve} \]

\[ \Pi_1 \rightarrow \Pi_2 \rightarrow \Pi_3 \rightarrow \Pi_4 \rightarrow \Pi_5 \rightarrow \Pi_6 \rightarrow \text{solve} \]

\[ \Pi_1 \rightarrow \Pi_2 \rightarrow \Pi_3 \rightarrow \Pi_4 \rightarrow \Pi_5 \rightarrow \Pi_6 \rightarrow \text{solve} \]

\[ \Pi_1 \rightarrow \Pi_2 \rightarrow \Pi_3 \rightarrow \Pi_4 \rightarrow \Pi_5 \rightarrow \Pi_6 \rightarrow \text{solve} \]

\[ \Pi_1 \rightarrow \Pi_2 \rightarrow \Pi_3 \rightarrow \Pi_4 \rightarrow \Pi_5 \rightarrow \Pi_6 \rightarrow \text{solve} \]

Figure 9: LF\(_\alpha\) encoding of Hereditary Harrop Formulas (LF\(_\alpha\) concrete syntax).

\[ \text{obj} : \text{Type} \quad \text{fam} : \text{Type} \quad \text{sup} : \text{Type} \quad \text{same} : \text{obj} \cap (\text{fam} \cap (\text{fam} \cup \text{sup})) \quad \text{term} : (\text{obj} \cup (\text{fam} \cup (\text{fam} \cup \text{sup}))) \rightarrow \text{Type} \]

\[ \text{obj} = \text{term}(\text{in}_{\text{fam}}^\text{fam} \cup (\text{fam} \cup \text{sup})) \]
\[ \text{fam} = \text{term}(\text{in}_{\text{obj}}^\text{obj} \cup (\text{obj} \cup \text{sup})) \]
\[ \text{sup} = \text{term}(\text{in}_{\text{sup}}^\text{sup} \cup (\text{obj} \cup \text{sup})) \]
\[ \text{tp} = \text{kn} \cap \text{sup} \cap \text{type} \]
\[ \text{lam} = (\text{fam} \rightarrow \text{obj} \rightarrow \text{obj}) \cap (\text{fam} \rightarrow (\text{obj} \rightarrow \text{fam}) \rightarrow \text{fam}) \]
\[ \text{pi} = (\text{obj} \rightarrow \text{fam}) \cap (\text{fam} \rightarrow (\text{obj} \rightarrow \text{fam}) \rightarrow \text{fam}) \]
\[ \text{app} = (\text{obj} \rightarrow \text{obj} \rightarrow \text{obj}) \]
\[ \text{of} = (\text{obj} \rightarrow \text{fam} \rightarrow \text{Type}) \]

Figure 10: LF in LF\(_\alpha\).

In the following we show the expressive power of LF\(_\alpha\) encoding classical features of typing systems with strong intersection and union.

**Auto application.** The provable judgement \( \text{r}_{BG} \lambda x. x : \sigma \cap (\sigma \rightarrow \tau) \rightarrow \tau \in \mathcal{B} \) is faithfully rendered in LF\(_\alpha\) by the provable LF\(_\alpha\)-judgment of \( \text{r}_{\sigma \cap (\sigma \rightarrow \tau) \rightarrow \tau} \lambda x : \sigma \cap (\sigma \rightarrow \tau) \rightarrow \tau \rightarrow \tau \).

**Polymorphic identity.** The provable judgement \( \text{r}_{BG} \lambda x. x : (\sigma \rightarrow \sigma) \cap (\tau \rightarrow \tau) \in \mathcal{B} \) is faithfully rendered in LF\(_\alpha\) by the provable judgement \( \text{r}_{\sigma \rightarrow \sigma} \lambda x. x : (\sigma \rightarrow \sigma) \cap (\tau \rightarrow \tau) \).

**Commutativity of union.** The provable judgement \( \lambda x. x : (\sigma \cup \tau) \rightarrow (\tau \cup \sigma) \) in \( \mathcal{B} \) is faithfully rendered in LF\(_\alpha\) by the provable judgement \( \lambda x. \sigma \cup \tau. \lambda y. \sigma. \text{in}_\sigma^\tau y, \lambda y. \tau. \text{in}_\tau^\sigma y \) : \((\sigma \cup \tau) \rightarrow (\tau \cup \sigma) \).

**LF\(_\alpha\) 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).

\[ γ := a \mid γ \land γ \mid π \Rightarrow γ \mid γ \lor γ \]

\[ π := a \mid π \land π \mid γ \Rightarrow π \]

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

**Pierce’s program** is faithfully rendered in LF by

\[
\begin{align*}
\text{Neg} : & \quad \text{Type} \\
T & \quad \text{Type} \\
\text{Test} & \quad \text{Pos} \lor \text{Neg} \\
\text{Is}_0 & \quad (\text{Neg} \rightarrow F) \cap (\text{Zero} \rightarrow T) \cap (\text{Pos} \rightarrow F) \\
\text{Is}_0 \_ \text{Test} & \quad [\lambda x: \text{Pos}. \text{pr}_r, (pr, \text{Is}_0) \_ x, \lambda x: \text{Neg}. \text{pr}_l \text{Is}_0 \_ x] \_ \text{Test}
\end{align*}
\]

The above example clearly illustrates the advantages of taking the LF\( _A \) framework. In plain LF we would render such an example only deeply encoding \( \exists \) 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 prototyope 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 Bruijn 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 development is available on Bull and Bull-Subtyping.

As future work, we are planning to design a higher-order unification algorithm for \( \Delta \)-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 \( \Lambda \)-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 \( \gamma \) such that \( γ \Rightarrow μ \_ x, \lambda x: \text{pr}_r, (pr, \text{Is}_0) \_ x, \lambda x: \text{Neg}. \text{pr}_l \text{Is}_0 \_ x] \_ \text{Test} \)

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

**REFERENCES**


