Towards an implementation of a logical framework based on intersection and union types*

(Extended abstract)

Claude Stolze[1] and Luigi Liquori[1]

Université Côte d'Azur, INRIA, France
Claude.Stolze@inria.fr Luigi.Liquori@inria.fr

Abstract. We present the dependent-type theory and a type checker implementation of an experimental theorem prover based on the Edinburgh Logical Framework LF, extended with *proof-functional* logical connectives such as intersection, union, and strong (or relevant) implication. Proof-functional connectives allow reasoning about the structure of logical proofs, in this way giving to the latter the status of *first-class* objects. 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 subformulas. Intersection, union types, and relevant implication have a natural interpretation in set theory as set intersection, set union, and subset, respectively. Some examples in the extended type theory are provided and a review of a type checker is presented. Once validated *in vitro*, the proof-functional type theory could be successfully plugged in existing truth-functional proof assistants.

1 Introduction

This extended abstract presents our ongoing research of extending the Edinburgh Logical Framework (LF) [HHP93] with *proof-functional* logical connectives like intersection, union, and relevant implication, together with the implementation of a type checker of an experimental theorem prover based on this extended type theory. The aim of this research is to experiment the expressiveness of the proof-functional operators in a LF based logical framework by considering proofs as genuine first-class objects. Studying the behavior of proof-functional connectives would be beneficial to existing interactive theorem provers, and dependently typed programming languages.

Intersection types [BCDC83] were first introduced as a form of *ad hoc* polymorphism in (pure) lambda-calculi *à la* Curry. Union types [MPS86,BDCd95] were later introduced as a *dual* of intersection. The main (non syntax-directed)

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

rules are:

$$\frac{B \vdash M : \sigma \quad B \vdash M : \tau}{B \vdash M : \sigma \cap \tau}\ (\cap I) \qquad \frac{B \vdash M : \sigma_1 \cap \sigma_2 \quad i \in \{1,2\}}{B \vdash M : \sigma_i}\ (\cap E_i)$$

$$\frac{B \vdash M : \sigma_i \quad i \in \{1,2\}}{B \vdash M : \sigma_1 \cup \sigma_2}\ (\cup I_i) \qquad \frac{B, x{:}\sigma \vdash M : \rho \quad B, x{:}\tau \vdash M : \rho \quad B \vdash N : \sigma \cup \tau}{B \vdash M[N/x] : \rho}\ (\cup E)$$

As intersection and union types had their classical development for (undecidable) type assignment systems, many researchers explored intersection and union type theories in (typed) lambda-calculi *à la* Church.

Proof-functional logical connectives represent evidences as "polymorphic" constructions, that is, a *same* evidence can be used as a proof for different sentences. This is in contrast to classical *truth-functional* connectives where the meaning of a compound formula is dependent only on the truth value of its subformulas.

Pottinger [Pot80] introduced a conjunction, called *strong conjunction* $\cap$, requiring more than the existence of constructions proving the left and the right hand side of the conjuncts: if we have a reason to assert A, which is also a reason to assert B, then the *same* reason asserts $A \cap B$. This interpretation makes inhabitants of $A \cap B$ function as polymorphic evidence for both A and B. Recently [DL10,DdLS16,LS17] extended the above logical interpretation with union types as another proof-functional operator, the *strong disjunction* $\cup$. Inspired by 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 $A \cup B$: the authors also explored the relationship between the proof-functional and the truth-functional nature of strong conjunction and strong disjunction, by introducing the notion of *essence* ($\wr \Delta \wr$) to connect typed (Δ) with their corresponding pure (M) lambda-terms. Specifically, a typed judgment $\Gamma \vdash \Delta : \sigma$ if and only if a type assignment judgment $B \vdash M : \sigma$ and $\wr \Gamma, \Delta \wr =_\beta B, M$.

Strong (or *Relevant*) *Implication* $\to_r$ (see Meyer and Routley's Minimal Relevant Logic B^+ [MR72]) is another well-known proof-functional connective: as explained in [BM94], 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.

In [LS17] we showed how strong conjunction (resp. disjunction) has a natural interpretation with set intersection (resp. union), by presenting a proof-functional logic using Mints' realizers [Min89], and we sketched how a sound and complete interpretation of subsets can be recovered by combination of relevant $\to_r$ and the subtyping relation $\leqslant$ in the type theory Ξ (*i.e.* axioms 1 to 14, as presented in [BDCd95]).

Encoding proof-functional connectives into a logical framework *à la* LF is not a simple task, because of the presence of the *side-conditions* constraints characterizing proof-functional evidence for intersection, union, and relevant implication. These can be coded in LF only with difficulty: moreover, the dependent-

types machinery has complications in the presence of intersection, union, and relevant types.

Here we have extended the LF logic with proof-functional operators, allowing an agile and shallow encoding of proof-functional connectives. Thus the main contributions of this work are (i) introducing the strong (relevant) implication in presence of intersection and union types; (ii) raising the type system to a logical framework by introducing dependent types *à la* LF; (iii) validating the framework, presenting a quite compact shallow encoding for [BDCd95] in the augmented LF; (iv) implementing the type checking algorithm and a basic REPL for the augmented LF.

A web artifact is available at `https://github.com/cstolze/Bull`.

2 Extending LF with Proof-functional Operators

We extend the syntax of families and objects of LF as follows:

$$\sigma, \tau ::= \text{as in LF} \mid \sigma \cap \tau \mid \sigma \cup \tau \mid \Pi^r x{:}\sigma.\tau$$
$$\Delta ::= \text{as in LF} \mid \Delta \cap \Delta \mid \mathsf{pr}_i\, \Delta \mid \Delta \cup \Delta \mid \mathsf{in}_i^\sigma\, \Delta \mid \lambda^r x{:}\sigma.\Delta \quad i \in \{1,2\}$$

There are three *proof-functional* objects, namely the strong conjunction (typed with $\sigma \cap \tau$) with corresponding two projections, the strong disjunction (typed with $\sigma \cup \tau$) with corresponding two injections, and the strong (or relevant) lambda-abstraction (typed with Π^r). Note the overloading of $\cap$ and $\cup$ in families and objects, and of the infix space " " in relevant and non relevant applications: they are not harmful. Note that injections in_i need to be decorated with the injected type σ to keep constructivism of the type system and decidability of type reconstruction. Kinds, contexts, and signatures are defined as in LF.

We extend the LF reductions for objects with the following four reductions:

$$\mathsf{pr}_i\,(\Delta_1 \cap \Delta_2) \longrightarrow_{\mathsf{pr}_i} \Delta_i \qquad (\Delta_1 \cup \Delta_2)\,\mathsf{in}_i^\sigma\,\Delta_3 \longrightarrow_{\mathsf{in}_i} \Delta_i\,\Delta_3 \qquad i \in \{1,2\}$$

The notion of *essence* was introduced in [DdLS16] to syntactically connect pure (*i.e.* M) and typed (*i.e.* Δ) lambda-terms: intuitively this function erases all operators having a *not syntax directed* type rule in [BDCd95]. It is compositional in all other cases. The presence of relevant lambda-abstraction requires the definition of the essence to be adjusted, as follows:

$$\wr \Delta_1 \cap \Delta_2 \wr_\Sigma^\Gamma \triangleq \wr \Delta_1 \wr_\Sigma^\Gamma \qquad \text{if } \wr \Delta_1 \wr_\Sigma^\Gamma =_\beta \wr \Delta_2 \wr_\Sigma^\Gamma$$
$$\wr \Delta_1 \cup \Delta_2 \wr_\Sigma^\Gamma \triangleq \wr \Delta_1 \wr_\Sigma^\Gamma \qquad \text{if } \wr \Delta_1 \wr_\Sigma^\Gamma =_\beta \wr \Delta_2 \wr_\Sigma^\Gamma$$
$$\wr \lambda^r x{:}\sigma.\Delta \wr_\Sigma^\Gamma \triangleq \lambda x.\wr \Delta \wr_\Sigma^{\Gamma, x:\sigma} \qquad \text{if } \wr \Delta \wr_\Sigma^{\Gamma,x:\sigma} =_\beta x$$
$$\wr \Delta_1\, \Delta_2 \wr_\Sigma^\Gamma \triangleq \begin{cases} \wr \Delta_2 \wr_\Sigma^\Gamma & \text{if } \Gamma \vdash_\Sigma \Delta_1 : \Pi^r x{:}\sigma.\tau \\ \wr \Delta_1 \wr_\Sigma^\Gamma \wr \Delta_2 \wr_\Sigma^\Gamma & \text{otherwise} \end{cases}$$

We extend the LF type rules for objects as follows:

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

$$\frac{\Gamma \vdash_\Sigma \Delta_1 : \sigma \quad \Gamma \vdash_\Sigma \Delta_2 : \tau \quad \wr \Delta_1 \wr_\Sigma^\Gamma =_\beta \wr \Delta_2 \wr_\Sigma^\Gamma}{\Gamma \vdash_\Sigma \Delta_1 \cap \Delta_2 : \sigma \cap \tau} \ (\cap I) \qquad \frac{\Gamma \vdash_\Sigma \Delta : \sigma_1 \cap \sigma_2 \quad i \in \{1,2\}}{\Gamma \vdash_\Sigma \mathsf{pr}_i\, \Delta : \sigma_i} \ (\cap E_i)$$

$$\frac{\begin{array}{l}\Gamma \vdash_\Sigma \Delta_1 : \Pi y{:}\sigma.\rho[\mathsf{in}_1^\tau y/x] \quad \wr \Delta_1 \wr_\Sigma^\Gamma =_\beta \wr \Delta_2 \wr_\Sigma^\Gamma \\ \Gamma \vdash_\Sigma \Delta_2 : \Pi y{:}\tau.\rho[\mathsf{in}_2^\sigma y/x] \quad \Gamma \vdash_\Sigma \Delta_3 : \sigma \cup \tau\end{array}}{\Gamma \vdash_\Sigma (\Delta_1 \cup \Delta_2)\, \Delta_3 : \rho[\Delta_3/x]} \ (\cup E) \qquad \frac{\begin{array}{ll}\Gamma \vdash_\Sigma \Delta : \sigma_i & i \neq j \\ \Gamma \vdash_\Sigma \sigma_1 \cup \sigma_2 : \mathsf{Type} & i,j \in \{1,2\}\end{array}}{\Gamma \vdash_\Sigma \mathsf{in}_i^{\sigma_j}\, \Delta : \sigma_1 \cup \sigma_2} \ (\cup I_i)$$

Because of the isomorphism between typed and type assignment derivations with intersection and union types [DdLS16], a plausible conjecture is that adding dependent types and relevant implication would not break Church-Rosser, subject reduction, strong normalization, unicity of typing, and decidability of type reconstruction ($\exists \sigma. \Gamma \vdash_\Sigma \Delta : \sigma$?) and of type checking (can we check if $\Gamma \vdash_\Sigma \Delta : \sigma$?) for the augmented proof-functional LF with intersection, union, and relevant implication [HLSC17].

3 Examples

We present an encoding for the type assignment system of [BDCd95] enriched with relevant abstractions and applications: the encoding makes fundamental use of the typed-calculus introduced in [LR07,DL10,DdLS16] and is improved in this paper with relevant products and dependent types. More precisely, since each type assignment derivation $\mathcal{D}$ for $\vdash M : \sigma$ (M closed) is isomorphic to a typed lambda term Δ, our encoding also applies to the typed calculus. Let $\to$ and $\to_r$ denote a non-dependent product type Π and a relevant product type Π^r, respectively. A *shallow* encoding in the enhanced LF, using all the new features is shown below: the source language and the target language are mostly overlapped.

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

Note the absence of the constants $c_{=abst}$, $c_{=sabst}$, $c_{=app}$, $c_{=sapp}$, $c_{=pr_{1,2}}$, $c_{=in_{1,2}}$, $c_{=sconj}$, and $c_{=sdisj}$ needed in a pure LF encoding to capture equivalence of typed lambda terms having the same essence: here 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 encode (in a shallow way and using HOAS) also the framework in the framework itself.

Let try to show some examples of using the above encoding. The derivation $\mathcal{D}_1$ for the type assignment judgment $\vdash \lambda x.x : (\sigma \to \sigma) \cap (\tau \to \tau)$ (isomorphic to the typed lambda term $\lambda x{:}\sigma.x \cap \lambda x{:}\tau.x$) and the derivation $\mathcal{D}_2$ for the type assignment judgment $\vdash \lambda x.x\,x : (\sigma \cap (\sigma \to \tau)) \to \tau$ (isomorphic to the typed lambda-term $\lambda x{:}\sigma \cap (\sigma \to \tau).(\mathsf{pr}_2\ x)\,(\mathsf{pr}_1\ x)$) can be encoded as follows:

$$c_{sconj}\,(c_{\to}\,s\,s)\,(c_{\to}\,t\,t)\,((c_{abst}\,s\,s\,(\lambda x{:}obj\ s.x)) \cap (c_{abst}\,t\,t(\lambda x{:}obj\ t.x)))$$

$$c_{abst}\,(c_{\cap}\,s\,(c_{\to}\,s\,t))\,t\,(\lambda x{:}obj\ (c_{\cap}\,s\,(c_{\to}\,s\,t)).\,(c_{app}\,s\,t\,(pr_2(c_{pr_i}\,s\,(c_{\to}\,s\,t)\,x))(pr_1(c_{pr_i}\,s\,(c_{\to}\,s\,t)\,x))))$$

Needless to say that, with a deep embedding in pure LF, they would be far more tedious. This encoding is also available in the web artifact using our concrete syntax.

4 Implementation

Our current implementation experiments with 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.

We use the syntax of Pure Type Systems, as introduced by Berardi in his Ph.D. and further studied by Barendregt [Bar91], 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 a special care were done to overlap (best effort) without conflicts our syntax with the one of Coq.

Remark. The implementation of $(\cup E)$ type rule is tricky because of the presence of types $\rho[\mathsf{in}_1^\tau\ y/x]$ and $\rho[\mathsf{in}_2^\sigma\ y/x]$ 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:

$$\frac{\begin{array}{l}\Gamma \vdash_\Sigma \Delta_1 : \Pi y{:}\sigma.\rho\,(\mathsf{in}_1^\tau\, y) \quad \wr\, \Delta_1 \wr_\Sigma^\Gamma =_\beta \wr\, \Delta_2 \wr_\Sigma^\Gamma \\ \Gamma \vdash_\Sigma \Delta_2 : \Pi y{:}\tau.\rho\,(\mathsf{in}_2^\sigma\, y) \quad \Gamma \vdash_\Sigma \rho : \Pi y{:}(\sigma \cup \tau).\mathsf{Type}\end{array}}{\Gamma \vdash_\Sigma \Delta_1 \cup_\rho \Delta_2 : \Pi x{:}\sigma \cup \tau.\rho\, x}\ (\cup E)_{\text{admissible}}$$

5 Our agenda

Our agenda is as follows.

- Integrate into the logical framework the minimal (sub)type theory Ξ and add an explicit coercion expression $(\tau)\Delta$ of type τ if $\vdash \Delta : \sigma$ and $\sigma \leqslant \tau$, as described in [LS17]; combining strong implication and subtyping would be useful in achieving a sound and complete interpretation of $\to, \cap, \cup$, and $\to_r$ with function space, set intersection, set union, and subset, respectively.
- Extend the prototype with the subtyping algorithm $\mathcal{A}$ introduced in [LS17]: Hindley gave a subtyping algorithm for intersection types but, as far as we know, there is no research also accounting for union types. To our knowledge, this is the first algorithm combining both union and intersection. The algorithm is conceived to work for the minimal type theory Ξ.
- 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 case automatize, the construction of a witness Δ_2 proof for the formula σ_2 having the same essence M of Δ_1. Existing proof assistants could get some benefit if extended with a proof-functional operators.

Acknowledgment. We are very grateful to Daniel Dougherty, Furio Honsell and Ivan Scagnetto for many fruitful discussions.

References

Bar91. Henk Barendregt. Introduction to generalized type systems. *Journal of Functional Programming*, 1(2):125–154, 1991.

BCDC83. Henk Barendregt, Mario Coppo, and Mariangiola Dezani-Ciancaglini. A Filter Lambda Model and the Completeness of Type Assignment. *Journal of Symbolic Logic*, 48(4):931–940, 1983.

BDCd95. Franco Barbanera, Mariangiola Dezani-Ciancaglini, and Ugo de'Liguoro. Intersection and union types: syntax and semantics. *Inf. Comput.*, 119(2):202–230, 1995.

BM94. Franco Barbanera and Simone Martini. Proof-functional connectives and realizability. *Archive for Mathematical Logic*, 33:189–211, 1994.

DdLS16. Daniel J. Dougherty, Ugo de'Liguoro, Luigi Liquori, and Claude Stolze. A realizability interpretation for intersection and union types. In *APLAS*, volume 10017 of *Lecture Notes in Computer Science*, pages 187–205. Springer, 2016.

DL10. Daniel J. Dougherty and Luigi Liquori. Logic and computation in a lambda calculus with intersection and union types. In *LPAR*, volume 6355 of *Lecture Notes in Computer Science*, pages 173–191. Springer, 2010.

HHP93. Robert Harper, Furio Honsell, and Gordon Plotkin. A framework for defining logics. *J. ACM*, 40(1):143–184, 1993.

HLSC17. Furio Honsell, Luigi Liquori, Ivan Scagnetto, and Stolze Claude. The strong logical framework and bull, its incarnation. Manuscript, work in progress, Inria, Università di Udine, 2017.

LR07. Luigi Liquori and Simona Ronchi Della Rocca. Intersection typed system *à la* Church. *Information and Computation*, 9(205):1371–1386, 2007.

LS17. Luigi Liquori and Claude Stolze. A Decidable Subtyping Logic for Intersection and Union Types. Research report https://hal.archives-ouvertes.fr/hal-01488428, Inria, March 2017.

Min89. Grigori Mints. The completeness of provable realizability. *Notre Dame Journal of Formal Logic*, 30(3):420–441, 1989.

MPS86. David B. MacQueen, Gordon D. Plotkin, and Ravi Sethi. An ideal model for recursive polymorphic types. *Information and Control*, 71(1/2):95–130, 1986.

MR72. Robert K. Meyer and Richard Routley. Algebraic analysis of entailment I. *Logique et Analyse*, 15:407–428, 1972.

Pot80. Garrel Pottinger. A type assignment for the strongly normalizable λ-terms. In *To H.B. Curry: Essays on Combinatory Logic, Lambda Calculus and Formalism*, pages 561–577. Academic Press, 1980.