A Representation of F_ω in LF

Carsten Schürmann, Dachuan Yu*, and Zhaozhong Ni

Department of Computer Science
Yale University
{carsten,yu,nzz}cs.yale.edu

Abstract. We study how the type theory F_ω can be adequately represented in the meta-logical framework Twelf [16]. This development puts special emphasis on the way how terms, types, and kinds are represented in that it uses higher-order abstract syntax to model variable binding and dependent types to model typing constraints. Furthermore our design ensures that only well-typed terms and well-kinded types can be constructed. A possible application of this work lies in the development of safe intermediate languages for compilation.

1 Introduction

Modern compilers employ sophisticated compilation technology to guarantee safety conditions of the generated binary. An important class of safety conditions is captured by type systems with which compilers attempt to maintain type information across an entire cascade of intermediate languages throughout a compilation process. The cascade starts with a source language and typically ends in a machine language. A variety of techniques have been proposed to express safety conditions, such as PCC [12] and TAL [11].

Intermediate languages are typically designed in such a way that the conceptual difference between the individual languages is small and manageable, the properties of each particular language are provable, and the relationship between different intermediate languages is analyzable. In this paper we concentrate on one particular intermediate language F_ω and its properties, which forms the basis of FLINT [18] and TILT [8].

In general, compilation from one language to another is expressed by judgments and inference rules. The soundness argument is often left to language designers who typically reason about the language's properties informally, with pencil and paper. Given that the design of those intermediate languages is a challenging engineering task in itself, and that in terms of safety so much depends on it, the question of whether the desired properties are really satisfied is of crucial importance. Examples of those properties include the correctness of static and dynamic semantics, subject reduction, progress, termination properties, observational equivalence, and soundness and completeness of the compilation. Informal proofs are often error prone. From an engineering point of view, it is difficult to

* This work was supported in part by NSF Grants CCR-9901011 and CCR-0081590

maintain a valid set of theorems and proofs while a formal development evolves. For this reason we advocate in this paper the use of meta-logical frameworks to specify, implement, and verify designs.

Different meta-logical frameworks have different advantages. Coq [4], Nuprl [1], and Isabelle/HOL [13] for example offer extremely elaborate and sophisticated interactive proof search tools. In order to use those tools, one must commit to a particular way of representing the inference systems involved. In particular, representations of typing relations, and operational semantics, for example, which have in general extremely elegant and expressive higher-order encodings are not directly supported in Coq, Nuprl, or Isabelle/HOL, because the question what induction principles to use is problematic [3, 9]. However, true higher-order encodings of those systems provide enormous advantages in that certain lemmas related to substitution and weakening are *implicitly* supported, and need not be implemented by the language designer. In this paper, we use Twelf [16] as representation language of specifications, algorithms, and their meta-theory.

From an implementors point of view, each intermediate language and each types system requires a different implementation of a type checker. We show in this paper with F_ω as example, how static typing can become part of the representation. Consequently, the LF type checker can decide if a term is well-typed or not. Ill-typed terms in F_ω are simply not typable in our representation. We show an implemented proof of type soundness for this design.

This paper is organized as follows. We discuss Twelf in Section 2, introduce F_ω in Section 3, discuss issues concerning substitution in Section 4, and give a reduction semantics in Section 5 before we show type soundness in Section 6. An example of how to use our encoding is given in Section 7. Section 8 outlines future work and assesses results.

2 Twelf

The Twelf system [16] is a meta-logical framework and a tool for experimentation in the theory of programming languages and logics. It supports a variety of tasks such as the *specification* of object languages and their semantics, implementations of *algorithms* manipulating object-language expressions and deductions, and formal development of the *meta-theory* of an object language. Twelf implements the logical framework LF [7] and it employs the judgments-as-types, and derivations-as-objects methodology for specification. Our formulation of LF is standard.

$$\begin{array}{lll}
\text{Kinds} & K & ::= \mathrm{type} \mid \Pi x : A.\, K \mid A \to K \\
\text{Types} & A & ::= a \mid A\ M \mid \Pi x : A_1.\, A_2 \mid A_1 \to A_2 \\
\text{Objects} & M & ::= c \mid x \mid \lambda x : \tau.\, e \mid M_1\ M_2 \\
\text{Signature} & \Sigma & ::= \cdot \mid \Sigma, c : K \mid \Sigma, a : A \\
\text{Signature} & \Gamma & ::= \cdot \mid \Gamma, x : A
\end{array}$$

We use a for type constants, c for objects constant, and x for variables. The signature Σ is used below to declare the constants related to our encoding.

Following standard practice [14] we assume substitutions to be capture avoiding and we omit all leading Π-abstractions prefixes from types. $\beta\eta$-convertibility is taken as the underlying notion of definitional equality [2]. $A \to K$ and $A_1 \to A_2$ are used as abbreviations for $\Pi x : A.\, K$ and $\Pi x : A_1 .\, A_2$ if x does not occur free in K and A_2, respectively. Sometimes we write $A_2 \leftarrow A_1$ for $A_1 \to A_2$.

As typing judgments for LF we write $\Gamma \vdash M : A$ if object M has type A in context Γ, and $\Gamma \vdash M :_c A$ if M is well-typed and in addition a canonical (β-normal, η-long) form. The corresponding inference rules can be found in [7].

3 F_ω

F_ω is a type theory which has been introduced by Girard in his thesis [5] as a tool to prove properties about higher-order logics. In type-directed compilation, F_ω's expressive power has made it an attractive choice for the core of the FLINT system [18] and TILT [8]. It extends the simply-typed λ-calculus by polymorphism and type constructors.

$$
\begin{array}{l}
\text{Kinds } \kappa ::= o \mid \kappa_1 \Rrightarrow \kappa_2 \\
\text{Types } \tau ::= \alpha \mid \Rightarrow \mid \tau_1\ \tau_2 \mid \lambda\alpha : \kappa.\, \tau \mid \forall \alpha : \kappa.\, \tau \\
\text{Terms } e ::= x \mid \lambda x : \tau.\, e \mid e_1\ e_2 \mid \Lambda\alpha : \kappa.\, e \mid e[\tau]
\end{array}
$$

There are different ways to encode F_ω into LF. One way, for example, is to represent expressions, types, and kinds as individual syntactic categories, and then to encode the related typing relations explicitly. For the purpose of this work however, we have chosen an "implicit" representation to index types by kinds, and terms by types (see below).

$$
\begin{array}{ll}
\mathrm{kd} & : \mathrm{type} \\
\mathrm{o} & : \mathrm{kd} \\
\Rrightarrow & : \mathrm{kd} \to \mathrm{kd} \to \mathrm{kd}
\end{array}
$$

We write $\ulcorner\cdot\urcorner$ for the (polymorphic) representation function that embeds the syntactic categories of F_ω into LF.

Theorem 1 (Adequacy of encoding: kinds). *If κ is a kind, then $\cdot \vdash \ulcorner\kappa\urcorner :_c$ kd. And conversely, if $\Gamma \vdash M :_c$ kd for some object K, then, there exists a kind κ, such that $\ulcorner\kappa\urcorner = M$.*

The type system of F_ω is strongly normalizing which seems to make LF's simply typed λ-calculus a good candidate to represent F_ω's type level directly. However such a encoding would be unsatisfactory because polymorphic quantification cannot be directly supported. Therefore we encode F_ω-types in LF as type family indexed by their respective kinds.

$$
\begin{array}{ll}
\mathrm{tp} & : \mathrm{kd} \rightarrow \mathrm{type} \\
\Rightarrow' & : \mathrm{tp}\ (\mathrm{o} \Rrightarrow \mathrm{o} \Rrightarrow \mathrm{o}) \\
@' & : \mathrm{tp}\ (K_1 \Rrightarrow K_2) \rightarrow \mathrm{tp}\ K_1 \rightarrow \mathrm{tp}\ K_2 \\
\hat{\lambda}' & : (\mathrm{tp}\ K_1 \rightarrow \mathrm{tp}\ K_2) \rightarrow \mathrm{tp}\ (K_1 \Rrightarrow K_2) \\
\forall' & : (\mathrm{tp}\ K \rightarrow \mathrm{tp}\ \mathrm{o}) \rightarrow \mathrm{tp}\ \mathrm{o}
\end{array}
$$

We mark these newly defined constants with a prime, because we reuse the same names for other inference rules defining atomic and canonical forms below. This choice makes signatures that encode theorems and proofs more readable.

Theorem 2 (Adequacy of encoding: types). *If τ is a type of kind κ with free type variables $\alpha_1 : \kappa_1, \ldots, \alpha_n : \kappa_n$, then $\alpha_1 : \ulcorner\kappa_1\urcorner, \ldots, \alpha_n : \ulcorner\kappa_n\urcorner \vdash \ulcorner\tau\urcorner : tp\ \ulcorner\kappa\urcorner$. And conversely, if $\alpha_1 : \ulcorner\kappa_1\urcorner, \ldots, \alpha_n : \ulcorner\kappa_n\urcorner \vdash M : tp\ \ulcorner\kappa\urcorner$ then there exists a type $\tau : \kappa$, s.t. $\ulcorner\tau\urcorner = M$. $\ulcorner\cdot\urcorner$ is compositional in that $\ulcorner[\tau/\alpha]\tau'\urcorner = (\lambda\alpha : \ulcorner\kappa\urcorner . \ulcorner\tau'\urcorner)\ \ulcorner\tau\urcorner$.*

F_ω allows β-reduction on the type level which induces an equality relation among well-kinded types which we denote by $\Gamma \vdash \tau_1 \equiv \tau_2 : \kappa$. In order to avoid notational clutter, we suppress context Γ and kind κ and simply write $\tau_1 \equiv \tau_2$ for this judgment. However, it is important to note that all of the defining inference rules rely on the participating types to be well-kinded.

$$
\frac{}{(\lambda\alpha : \kappa.\ \tau_1)\ \tau_2 \equiv [\tau_2/\alpha]\tau_1}\ \mathrm{tbeta} \qquad \frac{}{\tau \equiv \lambda\alpha : \kappa . \tau\ \alpha}\ \mathrm{teta}\ \ (\alpha \text{ not free in } \tau)
$$

$$
\frac{\tau \equiv \tau'}{\forall\alpha : \kappa.\ \tau \equiv \forall\alpha : \kappa.\ \tau'}\ \mathrm{tall} \qquad \frac{\tau \equiv \tau'}{\lambda\alpha : \kappa.\ \tau \equiv \lambda\alpha : \kappa.\ \tau'}\ \mathrm{tlam}
$$

$$
\frac{}{\Rightarrow \equiv \Rightarrow}\ \mathrm{tarr} \qquad \frac{\tau_1 \equiv \tau_1' \qquad \tau_2 \equiv \tau_2'}{\tau_1\ \tau_2 \equiv \tau_1'\ \tau_2'}\ \mathrm{tapp}
$$

$$
\frac{}{\tau \equiv \tau}\ \mathrm{tref} \qquad \frac{\tau_1 \equiv \tau_2 \qquad \tau_2 \equiv \tau_3}{\tau_1 \equiv \tau_3}\ \mathrm{ttra} \qquad \frac{\tau_1 \equiv \tau_2}{\tau_2 \equiv \tau_1}\ \mathrm{tsymm}
$$

The type level of F_ω forms a strongly normalizing λ-calculus [5]. Without further discussion and formalization, we assume this fact as given and leave a formulation of meta-level properties about this congruence relation to future work. F_ω also satisfies several inversion principles, two of which are important for this work.

Lemma 1 (Admissible rules of inference).

1. *If $\Rightarrow\ \tau_1\ \tau_2 \equiv \Rightarrow\ \tau_1'\ \tau_2'$ then $\tau_2 \equiv \tau_2'$.*
2. *If $\forall\alpha : \kappa . \tau_1 \equiv \forall\alpha : \kappa . \tau_2$ and $\tau_1' \equiv \tau_2'$ then $[\tau_1'/\alpha]\tau_1 \equiv [\tau_2'/\alpha]\tau_2$.*

The congruence relation and the two parts of Lemma 1 are expressed in LF by the signature depicted in Figure 1. The encoding of the judgment alone forces the left hand side and right hand side of a congruence to be of the same kind.

$$
\begin{array}{ll}
\equiv & : \mathrm{tp}\ K \to \mathrm{tp}\ K \to \mathrm{type} \\
\mathrm{tbeta} & : ((\hat{\lambda}'\ (\lambda a : \mathrm{tp}\ K.\, T_1\ a))\ @'\ T_2) \equiv T_1\ T_2 \\
\mathrm{teta} & : T \equiv \hat{\lambda}'\ (\lambda a : \mathrm{tp}\ K.\, T\ @'\ a) \\
\mathrm{tarr} & : \Rightarrow \equiv \Rightarrow \\
\mathrm{ttra} & : T_1 \equiv T_2 \to T_2 \equiv T_3 \to T_1 \equiv T_3 \\
\mathrm{tall} & : (\Pi a : \mathrm{tp}\ K.\, a \equiv a \to T_1\ a \equiv T_1'\ a) \to \forall'\ T_1 \equiv \forall'\ T_1' \\
\mathrm{tapp} & : T_1 \equiv T_1' \to T_2 \equiv T_2' \to T_1\ @'\ T_2 \equiv T_1'\ @'\ T_2' \\
\mathrm{tlam} & : (\Pi a : \mathrm{tp}\ K.\, a \equiv a \to T_1\ a \equiv T_1'\ a) \to \hat{\lambda}'\ T_1 \equiv \hat{\lambda}'\ T_1' \\
\mathrm{tsymm} & : T_1 \equiv T_2 \to T_2 \equiv T_1 \\
\mathrm{tinv} & : \Rightarrow\ @'\ T_1\ @'\ T_2 \equiv \Rightarrow\ @'\ T_1'\ @'\ T_2' \to T_2 \equiv T_2' \\
\mathrm{tinvall} & : \forall' T_1 \equiv \forall' T_1' \to T_2 \equiv T_2' \to T_1\ T_2 \equiv T_1'\ T_2'
\end{array}
$$

Fig. 1. Encoding of $\equiv$ and Lemma 1

Theorem 3 (Adequacy of encoding: congruence relation). *If $\mathcal{R}$ is a derivation of $\tau_1 \equiv \tau_2$ with free variables among $\alpha_1 : \kappa_1, \ldots, \alpha_n : \kappa_n$, then $\alpha_1 : \ulcorner\kappa_1\urcorner, u_1 : \alpha_1 \equiv \alpha_1, \ldots, \alpha_n : \ulcorner\kappa_n\urcorner, u_n : \alpha_n \equiv \alpha_n \vdash \ulcorner\mathcal{R}\urcorner : \ulcorner\tau_1\urcorner \equiv \ulcorner\tau_2\urcorner$. And conversely, if $\alpha_1 : \ulcorner\kappa_1\urcorner, u_1 : \alpha_1 \equiv \alpha_1, \ldots, \alpha_n : \ulcorner\kappa_n\urcorner, u_n : \alpha_n \equiv \alpha_n \vdash M : \ulcorner\tau_1\urcorner \equiv \ulcorner\tau_2\urcorner$ then there exists a derivation $\mathcal{R}$ of $\tau_1 \equiv \tau_2$, s.t. $\ulcorner\mathcal{R}\urcorner = M$.*

In addition $\ulcorner\cdot\urcorner$ is compositional, but only in as far as derivations of $\tau \equiv \tau$ are concerned. This limited property of compositionality alone, however, is insufficient for the general case. That even derivations of $\tau \equiv \tau'$ can be substituted for any of the u_i's in Theorem 3 is the main result of Lemma 4.

An implementation of this congruence relation is given in [17]. The rule 'tref' from above is also an admissible rule of inference, however we have chosen not to encode it as such, but instead to implement the admissibility proof.

Lemma 2 (Identity lemma). *For all types τ it holds that $\tau \equiv \tau$.*

Its encoding in Twelf as type family with a set of defining constant declarations can be found in [17].

$$\mathrm{id} : \Pi T : \mathrm{tp}\ K.\, T \equiv T \to \mathrm{type}$$

In F_ω, every equivalence class of types modulo congruence has a unique representative. They are called canonical forms and they are in β(tbeta)-normal η(teta)-long form [5]. Canonical forms are defined in terms of two judgments, one for canonical types whose definition is kind-directed and one for atomic types which is type directed. The "|" rule holds only for types of kind o.

$$
\frac{\Gamma, \alpha \downarrow \kappa \vdash \tau \Uparrow \mathrm{o}}{\Gamma \vdash \forall \alpha : \kappa.\, \tau \Uparrow \mathrm{o}}\,\forall
\qquad
\frac{\Gamma, \alpha \downarrow \kappa_1 \vdash \tau \Uparrow \kappa_2}{\vdash \lambda \alpha : \kappa.\, \tau \Uparrow \kappa_1 \Rrightarrow \kappa_2}\,\hat{\lambda}
\qquad
\frac{\Gamma \vdash \tau \downarrow \mathrm{o}}{\Gamma \vdash \tau \Uparrow \mathrm{o}}\,|
$$

$$
\frac{\alpha \downarrow \kappa \in \Gamma}{\Gamma \vdash \alpha \downarrow \kappa}\,\epsilon
\qquad
\frac{}{\Gamma \vdash \Rightarrow \downarrow \mathrm{o} \Rrightarrow \mathrm{o} \Rrightarrow \mathrm{o}}\Rightarrow
\qquad
\frac{\Gamma \vdash \tau_1 \downarrow \kappa_2 \Rrightarrow \kappa_1 \qquad \Gamma \vdash \tau_2 \Uparrow \kappa_2}{\Gamma \vdash \tau_1\ \tau_2 \downarrow \kappa_1}\,@
$$

$$
\begin{array}{ll}
\mathrm{can} & : \mathrm{tp}\ K \to \mathrm{type} \\
\mathrm{at} & : \mathrm{tp}\ K \to \mathrm{type} \\
\forall & : (\Pi a : \mathrm{tp}\ K.\, \mathrm{at}\ a \to \mathrm{can}\ (T\ a)) \to \mathrm{can}\ (\forall'\ T) \\
\hat{\lambda} & : (\Pi a : \mathrm{tp}\ K.\, \mathrm{at}\ a \to \mathrm{can}\ (T\ a)) \to \mathrm{can}\ (\hat{\lambda}'\ T) \\
| & : \mathrm{at}\ (T : \mathrm{tp}\ \mathrm{o}) \to \mathrm{can}\ T \\
\Rightarrow & : \mathrm{at}\ \Rightarrow' \\
@ & : \mathrm{at}\ T_1 \to \mathrm{can}\ T_2 \to \mathrm{at}\ (T_1\ @'\ T_2)
\end{array}
$$

Fig. 2. Encoding of canonical and atomic types.

The representation of atomic and canonical terms in LF is given in Figure 2. The type ascription $(T : \mathrm{tp\ o})$ in the declaration of $|$ is necessary because Twelf would otherwise infer the more general type 'tp K' as argument type.

Theorem 4 (Adequacy of encoding: canonical and atomic forms). *If $\mathcal{C}$ is a canonical form derivation of type τ with free type variables $\alpha_1 : \kappa_1, \ldots, \alpha_n : \kappa_n$, then $\alpha_1 : \ulcorner\kappa_1\urcorner, u_1 : at\ \alpha_1, \ldots, \alpha_n : \ulcorner\kappa_n\urcorner, u_n : at\ \alpha_n \vdash \ulcorner\mathcal{C}\urcorner : can\ \ulcorner\tau\urcorner$. And conversely, if $\alpha_1 : \ulcorner\kappa_1\urcorner, u_1 : at\ \alpha_1, \ldots, \alpha_n : \ulcorner\kappa_n\urcorner, u_n : at\ \alpha_n \vdash M : can\ \ulcorner\tau\urcorner$ then there exists a derivation $\mathcal{C}$ that τ is canonical, s.t. $\ulcorner\mathcal{C}\urcorner = M$. A symmetric property holds for atomic forms.*

Note that in this case $\ulcorner\cdot\urcorner$ is compositional, but only in the sense that atomic derivations can be substituted for atomic assumptions. The more general case of substituting canonical derivations for atomic assumptions also holds and is shown in Lemma 5. The introduction of canonical and atomic forms brings other benefits such as additional inversion lemmas (of which we only show one here).

Lemma 3 (Inversion). *If $\forall\alpha : \kappa.\, \tau \equiv \tau'$ and $\Gamma \vdash \tau' \Uparrow o$ then $\tau' = \forall\alpha : \kappa.\, \tau''$ for some τ''.*

What differentiates Lemma 3 from Lemma 1 is that τ' and $\forall\alpha : \kappa.\, \tau''$ are syntactical identical and not only convertible. Consequently this lemma is directly supported by LF and need not to be encoded extra.

Following our original proposal we make the well-typedness condition part of the Twelf encoding and avoid therefore an explicit encoding of the typing relationship. This technique relieves us from having to run a separate type-checking and type-normalization phase once a term has been constructed in LF. The well-typedness condition is built into the representation.

$$
\begin{array}{ll}
\mathrm{exp} & : \mathrm{can}\ (T : \mathrm{tp}\ o) \to \mathrm{type} \\
\mathrm{abs} & : (\mathrm{exp}\ C_1 \to \mathrm{exp}\ C_2) \to \mathrm{exp}\ (|\ (\Rightarrow\ @\ C_1)\ @\ C_2) \\
\mathrm{app} & : \mathrm{exp}\ (|\ (\Rightarrow\ @\ C_1)\ @\ C_2) \to \mathrm{exp}\ C_1 \to \mathrm{exp}\ C_2 \\
\mathrm{Abs} & : (\Pi a : \mathrm{tp}\ K.\, \Pi u : \mathrm{at}\ a.\, a \equiv a \to \mathrm{exp}\ (C\ a\ u)) \to \mathrm{exp}\ (\forall\ C) \\
\mathrm{App} & : \mathrm{exp}\ (\forall (C : \Pi a : \mathrm{tp}\ K.\, \mathrm{at}\ a \to \mathrm{can}\ (T_1\ a))) \\
 & \quad \to \Pi T : \mathrm{tp}\ K.\, \Pi C' : \mathrm{can}\ T.\, \Pi R : T_1 T \equiv T_2.\, \Pi C'' : \mathrm{can}\ T_2.\, \mathrm{exp}\ C''
\end{array}
$$

Fig. 3. Encoding of terms

$$
\frac{\Gamma(x) = \tau}{\Gamma \vdash x : \tau}\,\mathrm{var} \qquad \frac{\Gamma \vdash e : \tau \qquad \tau \equiv \tau'}{\Gamma \vdash e : \tau'}\,\mathrm{cong}
$$

$$
\frac{\Gamma, x : \tau_1 \vdash e : \tau_2}{\Gamma \vdash \lambda x : \tau_1 .\, e : \tau_1 \to \tau_2}\,\mathrm{abs} \qquad \frac{\Gamma \vdash e_1 : \tau_2 \to \tau_1 \qquad \Gamma \vdash e_2 : \tau_2}{\Gamma \vdash e_1\ e_2 : \tau_1}\,\mathrm{app}
$$

$$
\frac{\Gamma, \alpha : \kappa \vdash e : \tau}{\Gamma \vdash \Lambda\alpha : \kappa .\, e : \forall\alpha : \kappa .\, \tau}\,\mathrm{Abs} \qquad \frac{\Gamma \vdash e : \forall\alpha : \kappa .\, \tau_1 \qquad \Gamma \vdash \tau_2 : \kappa}{\Gamma \vdash e[\tau_2] : [\tau_2/\alpha]\tau_1}\,\mathrm{App}
$$

How should terms be represented? The obvious solution to introduce a type family and index it by types is insufficient, because typing is not unique. Each term has several types modulo applications of the 'cong' rule, and consequently none of the desired inversion principles on the type level exist. The alternative and successful solution is to stipulate that all types in the rules must be canonical. This solution amounts to omitting 'cong' from the list of rules above, and rewriting 'App' rule in such a way that it satisfies this new constraint. $[\tau_2/\alpha]\tau_1$ does not necessarily yield a type of canonical form, but it is known be to congruent to one.

$$
\frac{\Gamma \vdash e : \forall\alpha : \kappa .\, \tau_1 \qquad \Gamma \vdash \tau_2 : \kappa \qquad [\tau_2/\alpha]\tau_1 \equiv \tau'}{\Gamma \vdash e[\tau_2] : \tau'}\,\mathrm{App}
$$

If all types of terms are canonical, the term formation rules are directly representable as type family in LF, indexed by the a proof object that certifies the canonicity of its type. The respective constant declarations are given in Figure 3.

We would like to make two comments about this representation. First, the annotation $(T : \mathrm{tp}\ \mathrm{o})$ in the first declaration restricts terms to be of a type of kind o. Second, the formulation of all rules presented so far, but these in particular, take full advantage of Twelf's powerful type reconstruction abilities. $(|\ (\Rightarrow\ @\ C_1)\ @\ C_2)$, for example, is a proof of 'abs E' canonicity. It almost reads as its type. For reasons related to adequacy, Abs's type is parametric in a proof that parameter a is atomic. The additional hypothesis $a \equiv a$ extends the convertibility relation on types by reflexivity on parameters.

Theorem 5 (Adequacy of encoding: terms). *Let e be a term of type τ and $\mathcal{C}$ a derivation that τ is canonical. If e contains free type variable $\alpha_1 : \kappa_1, \dots \alpha_m : \kappa_m$ and free term variables $x_1 : \tau_1, \dots, x_n : \tau_n$, (and $\mathcal{C}_i$ proofs of their canonicity) , then $\alpha_1 : tp \ulcorner\kappa_1\urcorner, \dots \alpha_m : tp \ulcorner\kappa_m\urcorner, x_1 : exp \ulcorner\mathcal{C}_1\urcorner, \dots, x_n : exp \ulcorner\mathcal{C}_n\urcorner \vdash \ulcorner e\urcorner : exp \ulcorner\mathcal{C}\urcorner$. And conversely, if $\alpha_1 : tp \ulcorner\kappa_1\urcorner, \dots \alpha_m : tp \ulcorner\kappa_m\urcorner, x_1 : exp \ulcorner\mathcal{C}_1\urcorner, \dots, x_n : exp \ulcorner\mathcal{C}_n\urcorner \vdash M : exp \ulcorner\mathcal{C}\urcorner$. then there exists a term $e : \tau'$, s.t. $\ulcorner e\urcorner = M$ and $\tau \equiv \tau'$.*

The bijection between terms and their representation in LF is compositional only for terms, but not for types.

4 Substitutions

The particular encoding of F_ω's syntactic categories from the previous section brings many advantages, but also some disadvantages. Only well-typed F_ω terms are representable, however, whenever polymorphic application is used, explicit proofs for the equivalence of types and the corresponding canonical forms must be provided. Consequently, even though we are using higher-order abstract syntax to encode the rule for polymorphic abstraction 'Abs', we cannot use LF application to mimic substitution. The representation of terms is not compositional when it comes to instantiating free type variables assumed to be atomic by types that are canonical. Instead we have to instantiate all free hypotheses in a hypothetical canonicity proof and convert the result into a canonical form proof by the means of a substitution lemma. In this section, we discuss the appropriate substitution lemmas for the congruence relation, canonical forms, and terms. In fact, those lemmas establish generalized compositionality properties for Theorem 3, Theorem 4, and Theorem 5, respectively. For the remainder of this section, recall, that we assume all terms to be well-typed, and all types to be well-kinded.

Lemma 4 (Substitution into the congruence relation). *Let $\tau_3 \equiv \tau_4$ be of kind κ'. If, under the hypothesis that α is a type variable of kind κ', $\tau_1 \equiv \tau_2$ is of kind κ then $[\tau_3/\alpha]\tau_1 \equiv [\tau_4/\alpha]\tau_2$.*

This proof extends Pfenning's representation of the substitution lemma [15] by polymorphic quantification. It is encoded as a type family

$$\begin{aligned}\text{thm-sub-c} : &(\Pi a : \text{tp } K'.\, a \equiv a \to (T_1\ a : \text{tp } K) \equiv T_2\ a)\\ &\to T_3 \equiv T_4\\ &\to T_1\ T_3 \equiv T_2\ T_4\\ &\to \text{type.}\end{aligned}$$

and the implementation of the proof is given in [17]. The type annotation ($T_1\ a$: tp K) signals Twelf's type reconstruction algorithm, that K cannot depend on types. The substitution lemma holds not only for the congruence relation, but also for types.

Lemma 5 (Substitution into canonical/atomic forms).

1. *For all proofs that* $\Gamma, \alpha : \kappa_1 \vdash \tau' \Uparrow \kappa_2$ *and* $\Gamma \vdash \tau \Uparrow \kappa_1$ *there exists a* τ''*, such that* $[\tau/\alpha]\tau' \equiv \tau''$ *and a proof of* $\Gamma \vdash \tau'' \Uparrow \kappa_2$*.*
2. *For all proofs that* $\Gamma, \alpha : \kappa_1 \vdash \tau' \downarrow \kappa_2$ *and* $\Gamma \vdash \tau \Uparrow \kappa_1$ *there exists a* τ''*, such that* $[\tau/\alpha]\tau' \equiv \tau''$ *and either a proof that* $\Gamma \vdash \tau'' \Uparrow \kappa_2$ *or* $\Gamma \vdash \tau'' \downarrow \kappa_2$*.*

And again, this lemma can be formalized in LF by two mutual dependent type families. An encoding of this proof is given in [17]. The main difficulty is the disjunction in the second part of the lemma: τ'' is either canonical or atomic. Pushing this logical connective into LF suggests an auxiliary intermediate type family "can∨at".

$$
\begin{array}{ll}
\text{can}\vee\text{at} & : \text{tp } K \to \text{type}.\\
\text{iscan} & : \text{can } T \to \text{can}\vee\text{at } T.\\
\text{isat} & : \text{at } T \to \text{can}\vee\text{at } T.\\
\\
\text{substc} & : (\Pi a : \text{tp } K.\, \text{at } a \to \text{can } (T'\ a)) \to \text{can } T\\
 & \quad \to (T'\ T) \equiv T'' \to \text{can } T'' \to \text{type}\\
\text{substa} & : (\Pi a : \text{tp } K.\, \text{at } a \to \text{at } (T'\ a)) \to \text{can } T\\
 & \quad \to (T'\ T) \equiv T'' \to \text{can}\vee\text{at } T'' \to \text{type}
\end{array}
$$

Lastly, we define substitution $[\sigma]e$ on the term level. Substitutions are defined as $\sigma = \tau_1/\alpha_1, \ldots, \tau_n/\alpha_n$ and all τ_i are in canonical form. As usual, we assume that these substitutions are capture avoiding through tacit variable renaming.

$$
\begin{array}{rcll}
[\sigma](x) & = & \sigma(x) & \\
[\sigma](\lambda x : \tau.\, e) & = & \lambda x : \tau'.\, [\sigma, x/x]e & \text{where } [\sigma](\tau) \equiv \tau' \text{ and } \tau' \text{ canonical}\\
[\sigma](e_1\ e_2) & = & ([\sigma]e_1)\ ([\sigma]e_2) & \\
[\sigma](\Lambda\alpha : \kappa.\, e) & = & \Lambda\alpha : \kappa.\, [\sigma, \alpha/\alpha]e & \\
[\sigma](e[\tau]) & = & ([\sigma]e)[\tau'] & \text{where } [\sigma](\tau) \equiv \tau' \text{ and } \tau' \text{ canonical}\\
\\
[\sigma](\alpha) & = & \sigma(\alpha) & \\
[\sigma](\Rightarrow) & = & \Rightarrow & \\
[\sigma](\tau_1\ \tau_2) & = & \tau' & \text{where } ([\sigma]\tau_1)\ ([\sigma]\tau_2) \equiv \tau'\\
 & & & \text{and } \tau' \text{ canonical}\\
[\sigma](\lambda\alpha : \kappa.\, \tau) & = & \Lambda\alpha : \kappa.\, \tau' & \text{where } [\sigma, \alpha/\alpha]\tau \equiv \tau' \text{ and } \tau' \text{ canonical}\\
[\sigma](\forall\alpha : \kappa.\, \tau) & = & \forall\alpha : \kappa.\, \tau' & \text{where } [\sigma, \alpha/\alpha]\tau \equiv \tau' \text{ and } \tau' \text{ canonical}
\end{array}
$$

Unlike applications of term substitutions e/x, which are encoded as β-redices, the representation of type substitution application τ/α cannot take advantage β-redices. Both, term variables and type variables are represented via higher-order abstract syntax, but β-reduction models substitution only for the former. For the latter we observe that with any instantiation of free type variables the canonicity proofs recorded with "exp" are likely to change. Consequently, this form of substitution application must be defined externally. Its definition is quite involved and implements the proof of the following substitution lemma.

Lemma 6 (Substitution into terms). *For all proofs that $\Gamma, \alpha : \kappa' \vdash e : \tau$ and $\Gamma \vdash \tau' \Uparrow \kappa'$ there exists a τ'', such that $[\tau'/\alpha]\tau \equiv \tau''$ and a proof of $\Gamma \vdash \tau'' \Uparrow \kappa_2$ and $\Gamma \vdash [\tau'/\alpha]e : \tau''$.*

Proof. By induction on e. We consider only case $e = e_1[\tau_1]$. All other cases are similar.

$\mathcal{C}' :: \Gamma \vdash \tau' \Uparrow \kappa'$	by assumption
$\mathcal{D} :: \Gamma, \alpha : \kappa' \vdash e_1[\tau_1] : \tau$	by assumption
$\mathcal{E}_1 :: \Gamma, \alpha : \kappa' \vdash e_1 : \forall\beta : \kappa''.\, \tau_2$	by inversion on $\mathcal{D}$
$\mathcal{C}_1 :: \Gamma, \alpha : \kappa' \vdash \tau_1 \Uparrow \kappa''$	by inversion on $\mathcal{D}$
$\mathcal{R} :: [\tau_1/\beta]\tau_2 \equiv \tau$	by inversion on $\mathcal{D}$
$\mathcal{R}'' :: [\tau'/\alpha](\forall\beta : \kappa''.\, \tau_2) \equiv \tau'''$	by induction hypothesis on $\mathcal{E}_1$
$\tau''' = \forall\beta : \kappa''.\, \tau_3'$	by Lemma 3
$\mathcal{C}''' :: \Gamma \vdash \forall\beta : \kappa''.\, \tau_3' \Uparrow \mathrm{o}$	by induction hypothesis on $\mathcal{E}_1$
$\mathcal{E}'' :: \Gamma \vdash [\tau'/\alpha]e_1 : \forall\beta : \kappa''.\, \tau_3'$	by induction hypothesis on $\mathcal{E}_1$
$\mathcal{R}_1' :: [\tau'/\alpha]\tau_1 \equiv \tau_1'$	by Lemma 5 on $\mathcal{C}_1, \mathcal{C}'$
$\mathcal{C}_1' :: \Gamma \vdash \tau_1' \Uparrow \kappa''$	by Lemma 5 on $\mathcal{C}_1, \mathcal{C}'$
$\mathcal{R}'' :: [\tau'/\alpha][\tau_1/\beta]\tau_2 \equiv [\tau_1'/\beta]\tau_3'$	by Lemma 1 (2) on $\mathcal{R}'', \mathcal{R}_1'$
$\mathcal{C}'' :: \Gamma, \beta : \kappa'' \vdash \tau_3' \Uparrow \mathrm{o}$	by inversion on $\mathcal{C}'''$
$\mathcal{R}_2' :: [\tau_1'/\beta]\tau_3' \equiv \tau_4$	by Lemma 5 on $\mathcal{C}'', \mathcal{C}_1'$
$\mathcal{C}_2' :: \Gamma \vdash \tau_4 \Uparrow \mathrm{o}$	by Lemma 5 on $\mathcal{C}'', \mathcal{C}_1'$
$\mathcal{I} :: \tau' \equiv \tau'$	by Lemma 2 on τ'
$\mathcal{Q}_1 :: [\tau'/\alpha][\tau_1/\beta]\tau_2 \equiv [\tau'/\alpha]\tau$	by Lemma 4 on $\mathcal{R}, \mathcal{I}$
$\mathcal{Q}_2 :: [\tau'/\alpha]\tau \equiv [\tau'/\alpha][\tau_1/\beta]\tau_2$	by tsymm on $\mathcal{Q}_1$
$\mathcal{Q}_3 :: [\tau'/\alpha]\tau \equiv [\tau_1'/\beta]\tau_3' \equiv \tau_4$	by ttra on $\mathcal{Q}_2, \mathcal{R}'', \mathcal{R}_2'$
$\mathcal{Q} :: \Gamma \vdash ([\tau'/\alpha]e_1)[\tau_1'] : \tau_4$	by App on $\mathcal{E}'', \mathcal{C}_1', \mathcal{R}_2'$
$\mathcal{Q} :: \Gamma \vdash [\tau'/\alpha](e_1[\tau_1]) : \tau_4$	by definition substitution

When substituting a type for a type variable in an expression, it is the proof of Lemma 6 that contains an algorithm on how to reestablish the canonicity proofs of the types of the resulting objects. In fact, this algorithm *defines* how to apply a substitution and it is formalized in Twelf as follows:

$$\begin{aligned}\mathrm{subst} : \Pi C : (&\Pi a : \mathrm{tp}\ K.\, \Pi u : \mathrm{at}\ a.\, \mathrm{can}\ (T\ a)).\\ &(\Pi a : \mathrm{tp}\ K.\, \Pi u : \mathrm{at}\ a.\, \Pi e : a \equiv a.\, \mathrm{exp}\ (C\ a\ u))\\ &\to \mathrm{can}\ T' \to (T\ T') \equiv T''\\ &\to \Pi C'' : \mathrm{can}\ T''.\, \mathrm{exp}\ C'' \to \mathrm{type}\end{aligned}$$

The one discussed case of the proof is depicted in Figure 4. All other cases can be found in [17]. In this representation, substituting types for type variables is more than just an algorithm acting on terms. It simultaneously enforces that all representation invariants are satisfied during execution. With this substitution lemma at hand, it is possible to define an evaluation semantics for F_ω and prove progress, termination, and eventually type soundness.

$$
\begin{array}{l}
\text{spapp} : \text{subst}(\lambda a : \text{tp } K'.\, \lambda u : \text{at } a.\, D\ a\ u) \\
\qquad\qquad (\lambda a : \text{tp}K'.\, \lambda u : \text{at } a.\, \lambda i : a \equiv a. \\
\qquad\qquad\quad \text{app}(E_1\ a\ u\ i)\ (T_1\ a)\ (C_1\ a\ u)\ (R\ a\ i)\ (D\ a\ u))\ C' \\
\qquad\qquad (\text{ttra } (\text{ttra } (\text{tsymm } Q_1)\ (\text{tinvall } R''\ R_1'))\ R_2')\ C_2' \\
\qquad\qquad (\text{App } E''\ T_6\ C_1'\ R_2'\ C_2') \\
\quad \leftarrow \text{subst } (\lambda a : \text{tp } K'.\, \lambda u : \text{at } a.\, \forall\ (\lambda a_1 : \text{tp } K.\, \lambda a_2 : \text{at } a_1.\ C_2\ a\ u\ a_1\ a_2)) \\
\qquad\qquad (\lambda a : \text{tp } K'.\, \lambda u : \text{at } a.\, E_1\ a\ u)\ C'\ R''\ (\forall'\ C'')\ E'' \\
\quad \leftarrow\ \text{substc } C_1\ C'\ R_1'\ C_1' \\
\quad \leftarrow\ \text{substc } C''\ C_1'\ R_2'\ C_2' \\
\quad \leftarrow\ \text{id } T'\ I \\
\quad \leftarrow\ \text{thm-sub-c } (\lambda a : \text{tp } K'.\ \lambda i : a \equiv a.\ R\ a\ i)\ I\ Q_1
\end{array}
$$

Fig. 4. Representation of Lemma 6, case $e = e_1[\tau_1]$

5 Operational Semantics

The evaluation semantics of F_ω preserves types. Only well-typed expressions evaluate to expressions of the same type. We write $e \mapsto e'$ for the judgment that denotes that expression e evaluates to e' in one step and $e \overset{*}{\mapsto} e'$ if it does so in arbitrary but finitely many steps. The left and the right hand side of the evaluation symbols are always well-typed.

$$
\frac{}{(\lambda x : \tau.\, e)v \mapsto [v/x]e}\ \text{ev\_beta} \qquad \frac{}{(\Lambda\alpha : \kappa.\, e)[\tau] \mapsto [\tau/\alpha]e}\ \text{ev\_pbeta}
$$

$$
\frac{e_1 \mapsto e_1'}{e_1\ e_2 \mapsto e_1'\ e_2}\ \text{ev\_app}_1 \qquad \frac{e_2 \mapsto e_2'}{e_1\ e_2 \mapsto e_1\ e_2'}\ \text{ev\_app}_2 \qquad \frac{e \mapsto e'}{e[\tau] \mapsto e'[\tau]}\ \text{ev\_papp}
$$

Without loss of generality, F_ω's evaluation semantics requires the argument of a β-redex to be a value. Only λ- and Λ-abstractions are considered values. The representation of the rules is given in Figure 5.

6 Type Soundness

By construction, the operational semantics of F_ω is type preserving. A quick inspection of the evaluation rules reveals that repeated applications of individual reduction steps must terminate, because the number of β-redices in a term (on term and type level) decreases with every individual step.

Theorem 6 (Termination). *If $\cdot \vdash e : \tau$, then all sequences of evaluation steps originating from e are finite.*

We can assume the context in which e is well-typed to be empty, because the evaluation semantics does not evaluate under λ-binders. Therefore, the question of type soundness reduces to the question of progress which ensures that the evaluation never gets stuck.

$$
\begin{array}{ll}
\text{val} & : \text{exp } C \to \text{type} \\
\text{vallam} & : \text{val (abs } C) \\
\text{valplam} & : \text{val (Abs } C) \\
\mapsto & : \text{exp } (C : \text{can } T) \to \text{exp } (C : \text{can } T) \to \text{type.} \\
\text{ev\_beta} & : (\text{app (abs } E)\ V) \mapsto (E\ V) \leftarrow \text{val } V. \\
\text{ev\_pbeta} & : (\text{App (Abs } E)\ T\ C_T\ R\ C') \mapsto E' \leftarrow \text{subst } C_0\ E\ C_T\ R'\ C'\ E'. \\
\text{ev\_app}_1 & : \text{app } E1\ E2 \mapsto \text{app } E1'\ E2 \leftarrow E1 \mapsto E1'. \\
\text{ev\_app}_2 & : \text{app } V\ E2 \mapsto \text{app } V\ E2' \leftarrow \text{val } V \leftarrow E2 \mapsto E2'. \\
\text{ev\_papp} & : \text{App } E\ A\ C_A\ R\ P \mapsto \text{App } E'\ A\ C_A\ R\ P \leftarrow E \mapsto E'. \\
\overset{*}{\mapsto} & : \text{exp } C \to \text{exp } C \to \text{type.} \\
\text{ev\_trans} & : E2 \overset{*}{\mapsto} E3 \to E1 \mapsto E2 \to E1 \overset{*}{\mapsto} E3. \\
\text{ev\_refl} & : E \overset{*}{\mapsto} E.
\end{array}
$$

Fig. 5. Encoding of the operational semantics

Theorem 7 (Progress). *If* $\cdot \vdash e : \tau$ *then either* e *is a value or there exists a* e'*, s.t.* $e \mapsto e'$*.*

The proof of this theorem is by induction on e. Unfortunately, Twelf's automatic deduction facilities are currently still in preliminary state and cannot be employed to prove the progress theorem automatically. A hand-coded proof is feasible and can be found in [17]. The disjunction used in the formulation of Theorem 7 is pushed down again to the LF level and called "val∨eval".

$$
\begin{array}{ll}
\text{val}\vee\text{eval} & : \text{exp } C \to \text{type} \\
\text{ve\_val} & : \text{val } E \to \text{val}\vee\text{eval } E \\
\text{ve\_eval} & : E \mapsto E' \to \text{val}\vee\text{eval } E \\
\text{progress} & : \Pi E : \text{exp } C.\, \text{val}\vee\text{eval } E \to \text{type}
\end{array}
$$

The representation of the progress proof in LF is mostly straightforward, however it relies on the property that substitution of types into terms from the previous section is total. Informally true, this property must be formalized explicitly in LF. For a complete development of this (and related) theorems and their proofs consult [17].

7 Example

F_ω can be used to define new types, values, and the corresponding elimination principles for Booleans, natural numbers, pairs, and sum types [6]. By construction all formally encoded objects, types and kinds are well-typed and well-kinded, respectively. We demonstrate how to use our encoding of F_ω by defining Booleans values and the corresponding elimination principle as depicted in Figure 6.

$$
\begin{array}{l}
\text{bool} = \forall' \ (\lambda a : \text{tp}\ o.\ \lambda u : \text{at}\ a.\ |\ \Rightarrow\ @\ (|\ u)\ @\ (|\ \Rightarrow\ @\ (|\ u)\ @\ (|\ u)))\\
\text{true}\ \ :\ \text{exp bool}\\
\qquad = \Lambda' \ (\lambda a : \text{tp o}.\ \lambda u : \text{at}\ a.\ \lambda i : a \equiv a.\\
\qquad\qquad \lambda' \ (\lambda t : \text{exp}\ (|\ u).\ \lambda' \ (\lambda f : \text{exp}\ (|\ u).\ t)))\\
\text{false}\ :\ \text{exp bool}\\
\qquad = \Lambda' \ (\lambda a : \text{tp o}.\ \lambda u : \text{at}\ a.\ \lambda i : a \equiv a.\\
\qquad\qquad \lambda' \ (\lambda t : \text{exp}\ (|\ u).\ \lambda' \ (\lambda f : \text{exp}\ (|\ u).\ f)))\\
\text{if}\quad\ :\ \text{exp}\ (\forall' \ (\lambda a : \text{tp o}.\ \lambda u : \text{at}\ a.\\
\qquad\qquad |\ \Rightarrow\ @\ (|\ u)\ @\ (|\ \Rightarrow\ @\ (|\ u)\ @\ (|\ \Rightarrow\ @\ \text{bool}\ @\ (|\ u)))))\\
\qquad = \Lambda' \ (\lambda a : \text{tp}\ o.\ \lambda u : \text{at}\ a.\ \lambda i : a \equiv a.\\
\qquad\qquad \lambda' \ (\lambda x : \text{exp}\ (|\ u).\ \lambda' \ (\lambda y : \text{exp}\ (|\ u).\ \lambda' \ (\lambda e : \text{exp bool}.\\
\qquad\qquad\quad (\text{app}\ (\text{app}\ (\text{App}\ e\ a\ (|\ u)\\
\qquad\qquad\qquad (\text{tapp}\ (\text{tapp tarr}\ i)\ (\text{tapp}\ (\text{tapp tarr}\ i)\ i))\\
\qquad\qquad\qquad (|\ \Rightarrow\ @\ (|\ u)\ @\ (|\ \Rightarrow\ @\ (|\ u)\ @\ (|\ u))))\ x)\ y))))
\end{array}
$$

Fig. 6. Encoding of Booleans in F_ω

Using Twelf, we can experiment easily with this encoding, and verify that the standard properties hold, namely that

$$
\begin{array}{l}
\text{if}[\alpha]\ e_1\ e_2\ \text{true} \overset{*}{\mapsto} e_1\\
\text{if}[\alpha]\ e_1\ e_2\ \text{false} \overset{*}{\mapsto} e_2
\end{array}
$$

for any type α, and well-typed terms e_1, e_2. On the other hand, all terms that use the rule of polymorphic application, such as 'if[α]', tend to become very large. This is due to the fact that all assumptions about canonicity and congruence relation are expected to be made explicit. On the other hand these annotations can be automatically generated especially in the setting where we foresee this encoding of F_ω to be used: as a target language inside the implementation of a compiler.

8 Conclusion

The main contribution of this paper is an encoding of F_ω in the meta-logical framework Twelf. It is elegant, precise, and its main benefits include a formal representation of well-typed terms and a type-preserving operational semantics. We have shown in Twelf that F_ω is type sound. The main benefits of this encoding is that the LF type checker suffices to decide well-typedness. Terms are indexed by proofs that witness the canonicity of its type, which in turn contain all necessary kind information.

We view this paper as a case study on how to use meta-logical frameworks in the design, implementation, and verification process of datastructures and algorithms. Specifically, the main motivation of this work stems from the area of safe intermediate languages. The encoding and the properties of F_ω are not just of significant theoretical interest, but have also many practical applications, related to compilers and proof carrying code.

In future work we plan to extend F_ω to Mini-FLINT [10] by adding other features namely row polymorphism, type tuples, sum types, existentials, fixed-point and contextual recursive types. Mini-FLINT can serve as target language for compiling Featherweight Java [10]. We also plan to develop a compiler from mini-FLINT to typed assembly language [18] and to develop verifiable safe compilation techniques within the meta-logical framework, possibly all the way down to machine code. Finally, we plan to extend F_ω to support intensional type analysis in the spirit of [8, 19]. Applications of intensional type analysis include tagless garbage collection and polymorphic marshalling.

Acknowledgements We would like to thank Valery Trifonov and Zhong Shao for many helpful discussions.

References

1. Robert L. Constable et al. *Implementing Mathematics with the Nuprl Proof Development System.* Prentice-Hall, Englewood Cliffs, New Jersey, 1986.
2. Thierry Coquand. An algorithm for testing conversion in type theory. In Gérard Huet and Gordon Plotkin, editors, *Logical Frameworks*, pages 255–279. Cambridge University Press, 1991.
3. Joëlle Despeyroux, Frank Pfenning, and Carsten Schürmann. Primitive recursion for higher-order abstract syntax. In R. Hindley, editor, *Proceedings of the Third International Conference on Typed Lambda Calculus and Applications (TLCA'97)*, pages 147–163, Nancy, France, April 1997. Springer-Verlag LNCS. An extended version is available as Technical Report CMU-CS-96-172, Carnegie Mellon University.
4. Gilles Dowek, Amy Felty, Hugo Herbelin, Gérard Huet, Chet Murthy, Catherine Parent, Christine Paulin-Mohring, and Benjamin Werner. The Coq proof assistant user's guide. Rapport Techniques 154, INRIA, Rocquencourt, France, 1993. Version 5.8.
5. J.-Y. Girard. Interpretation fonctionelle et elimination des coupures de l'arithmetique d'ordre superieur. These D'Etat, Universite Paris VII, 1972.
6. J.-Y. Girard, Y. Lafont, and P. Taylor. *Proofs and Types.* Cambridge Tracts in Theoretical Computer Science 7. Cambridge University Press, 1988.
7. Robert Harper, Furio Honsell, and Gordon Plotkin. A framework for defining logics. *Journal of the Association for Computing Machinery*, 40(1):143–184, January 1993.
8. Robert Harper and Greg Morrisett. Compiling polymorphism using intensional type analysis. In *Conference Record of POPL '95: 22nd ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, pages 130–141, San Francisco, California, 1995.
9. Martin Hofmann. Semantical analysis for higher-order abstract syntax. In G. Longo, editor, *Proceedings of the 14th Annual Symposium on Logic in Computer Science (LICS'99)*, pages 204–213, Trento, Italy, July 1999. IEEE Computer Society Press.

10. Christopher League, Valery Trifonov, and Zhong Shao. Type-preserving compilation of featherweight Java. In *Foundations of Object-Oriented Languages (FOOL8)*, London, January 2001.
11. G. Morrisett, D. Walker, K. Crary, and N. Glew. From system F to typed assembly language. In *In 25th ACM SIGPLANSIGACT Symposium on Principles of Programming Languages*, pages 85–97, San Diego California, USA, 1998. ACM Press.
12. George C. Necula. Proof-carrying code. In Neil D. Jones, editor, *Conference Record of the 24th Symposium on Principles of Programming Languages (POPL'97)*, pages 106–119, Paris, France, January 1997. ACM Press.
13. Lawrence C. Paulson. *Isabelle: A Generic Theorem Prover*. Springer-Verlag LNCS 828, 1994.
14. Frank Pfenning. Logic programming in the LF logical framework. In Gérard Huet and Gordon Plotkin, editors, *Logical Frameworks*, pages 149–181. Cambridge University Press, 1991.
15. Frank Pfenning. A proof of the Church-Rosser theorem and its representation in a logical framework. *Journal of Automated Reasoning*, 1993. To appear. A preliminary version is available as Carnegie Mellon Technical Report CMU-CS-92-186, September 1992.
16. Frank Pfenning and Carsten Schürmann. System description: Twelf — a meta-logical framework for deductive systems. In H. Ganzinger, editor, *Proceedings of the 16th International Conference on Automated Deduction (CADE-16)*, pages 202–206, Trento, Italy, July 1999. Springer-Verlag LNAI 1632.
17. Carsten Schürmann, Dachuan Yu, and Zhaozhong Ni. A representation of F_ω in LF. `http://www.cs.yale.edu/~carsten/public/merlin01.elf`, 2001.
18. Zhong Shao. Implementing typed intermediate language. In *Proc. 1998 ACM SIGPLAN International Conference on Functional Programming (ICFP'98)*, pages 313–323, Baltimore, Maryland, September 1998.
19. Valery Trifonov, Bratin Saha, and Zhong Shao. Fully reflexive intensional type analysis. In *Proc. 2000 ACM SIGPLAN International Conference on Functional Programming (ICFP'00)*, pages 82–93. ACM Press, September 2000.