Lecture Notes on Logical Frameworks

15-814: Types and Programming Languages
Frank Pfenning

Lecture 21
Thu Nov 19, 2025

1 Introduction

This is the second lecture on dependent types, this time on their use in logical frameworks. A *logical framework* is a metalanguage for

1. representing deductive systems that are common in the definition of logics and programming languages,
2. implementing algorithms such as proof checking, type checking, evaluation, translation between deductive systems, etc.,
3. proving metatheorems about deductive systems such as preservation and progress, logical interpretations, compiler correctness, etc.

There are several surveys [Basin and Matthews, 2001, Pfenning, 2001] so we restrict ourselves to the LF Logical Framework [Harper et al., 1987, 1993] and its implementation in the Twelf system [Pfenning and Schürmann, 1999, Twelf].

The type theory underlying LF called λ_Π is minimalistic, consisting only of the dependent function type inhabited by λ-abstractions. As such, the equational theory is well-understood and every term and type has a *canonical form* which allows for simple, compositional representations of the objects of interests. We illustrate this through the example of proving type preservation for (a fragment of) LAMBDA to give a flavor of how TWELF can be used.

The syntax of LF and two key rules are shown in Section 7 for reference.

2 Representing Syntax

We distinguish between the *object language* we are modeling and the *metalanguage* in which we represent the components of the object language. Each category of object is represented by a *type* in the metalanguage. For example, in LAMBDA we start with object-language types. We start only with function types and eager pairs.

```
tp : type.

arrow : tp -> tp -> tp.
times : tp -> tp -> tp.
```

The representation function from our mathematical notation to the logical framework:

$$\begin{array}{lcll} \ulcorner \tau \to \sigma \urcorner & = & \mathsf{arrow}\ \ulcorner \tau \urcorner\ \ulcorner \sigma \urcorner & : \ \mathsf{tp} \\ \ulcorner \tau \times \sigma \urcorner & = & \mathsf{times}\ \ulcorner \tau \urcorner\ \ulcorner \sigma \urcorner & : \ \mathsf{tp} \end{array}$$

The next step is to represent expressions. Here we see the next idea: we represent variables in the object language by corresponding variables in the metalanguage.

```
exp : type.

lam : (exp -> exp) -> exp.
app : exp -> exp -> exp.
pair : exp -> exp -> exp.
case/pair : exp -> (exp -> exp -> exp) -> exp.
```

The representation function now has to take variables into account.

$$\begin{array}{lcll} \ulcorner x \urcorner & = & x & : \ \mathsf{exp} \\ \ulcorner \lambda x.\, e \urcorner & = & \mathsf{lam}\ (\lambda x. \ulcorner e \urcorner) & : \ \mathsf{exp} \\ \ulcorner e_1\, e_2 \urcorner & = & \mathsf{app}\ \ulcorner e_1 \urcorner\ \ulcorner e_2 \urcorner & : \ \mathsf{exp} \\ \ulcorner \langle e_1, e_2 \rangle \urcorner & = & \mathsf{pair}\ \ulcorner e_1 \urcorner\ \ulcorner e_2 \urcorner & : \ \mathsf{exp} \\ \ulcorner \mathsf{case}\ e\ (\langle x, y \rangle \Rightarrow e') \urcorner & = & \mathsf{case/pair}\ \ulcorner e \urcorner\ (\lambda x.\, \lambda y. \ulcorner e' \urcorner) & : \ \mathsf{exp} \end{array}$$

The "λ" and also variables on the right-hand side of the representation function are components of LF, our metalanguage. In the concrete syntax of Twelf we use $[x]M$ for $\lambda x.\, M$. For example:

```
id : exp = (lam [x] x).
k : exp = (lam [x] lam [y] x).
```

By representing variables as variables, and variable binding as λ-abstraction we gain, for free, variable renaming and also substitution satisfying the characteristic equation

$$\ulcorner [e_1/x] e_2 \urcorner = [\ulcorner e_1 \urcorner / x] \ulcorner e_2 \urcorner =_\beta (\lambda x.\, e_2)\, e_1$$

It relieves us from having to define a concrete representation of variables, specify the action of substitution, and then prove properties of substitution. The most common technique nowadays is by deBruijn indices [de Bruijn, 1972].

3 Representing Deductions

The next task is to represent deductions. For this purpose we need to use some dependent types for the first time—so far, we only needed non-dependent function types in the metalanguage. We represent judgments as types of their deductions. In the example of the language dynamics, we will have:

$$\ulcorner \begin{array}{c} \mathcal{W} \\ e\ \mathit{value} \end{array} \urcorner \ : \ \mathsf{value}\ \ulcorner e \urcorner$$

$$\ulcorner \begin{array}{c} \mathcal{S} \\ e \mapsto e' \end{array} \urcorner \ : \ \mathsf{step}\ \ulcorner e \urcorner\ \ulcorner e' \urcorner$$

We see that value and step are *type families* indexed by one or two terms of type exp. Concretely, we declare:

```
value : exp -> type.
step : exp -> exp -> type.
```

Each inference rule now becomes a constructor for (metalevel) objects of the given indexed type. We show one example in mathematical style:

$$\ulcorner \frac{\overset{\mathcal{S}}{e_1 \mapsto e_1}}{e_1\, e_2 \mapsto e_1' e_2}\ \mathsf{step/app}_1 \urcorner = \mathsf{step/app1}\ \ulcorner e_1 \urcorner\ \ulcorner e_2 \urcorner\ \ulcorner e_1' \urcorner\ \ulcorner \mathcal{S} \urcorner : \mathsf{step}\ (\mathsf{app}\ \ulcorner e_1 \urcorner\ \ulcorner e_2 \urcorner)\ (\mathsf{app}\ \ulcorner e_1' \urcorner\ \ulcorner e_2 \urcorner)$$

In order to express the type of step/app1 we use dependent function types, written as $\Pi x{:}A.\, B$ where A and B are metalanguage types, and B may depend on x. For indexed types in the last lecture, we wrote this as $\forall x \in \mathbb{N}.\, \tau$.

$$\mathsf{step/app1} : \Pi e_1{:}\mathsf{exp}.\, \Pi e_2{:}\mathsf{exp}.\, \Pi e_1'{:}\mathsf{exp}.\, \mathsf{step}\ e_1\ e_1' \to \mathsf{step}\ (\mathsf{app}\ e_1\ e_2)\ (\mathsf{app}\ e_1'\ e_2)$$

Because in the vast majority of cases the index objects can inferred, they are generally omitted when we write a signature in Twelf.

```
step/app1 : step E1 E1' -> step (app E1 E2) (app E1' E2).
```

Most of the rules are now routine, but in the case of β reduction we have to see how to exploit the metalevel substitution to model object-level substitution.

$$\ulcorner \frac{\overset{\mathcal{W}_2}{v_2\ \mathit{value}}}{(\lambda x.\, e_1)\, v_2 \ \mapsto\ [v_2/x]e_1}\ \mathsf{step/beta}\ \urcorner \ = \ \mathsf{step/beta}\ (\lambda x.\ulcorner e_1 \urcorner)\ \ulcorner v_2 \urcorner\ \ulcorner \mathcal{W}_2 \urcorner$$

$$: \ \mathsf{step}\ (\mathsf{app}\ (\mathsf{lam}\ (\lambda x.\ulcorner e_1 \urcorner))\ \ulcorner v_2 \urcorner)\ (\ulcorner [v_2/x]e_1 \urcorner)$$

Because index objects are considered modulo definitional equality (which is $\beta\eta$-conversion), we can model the representation of $[v_2/x]e_1$ as $(\lambda x.\ulcorner e_1 \urcorner)\ulcorner v_2 \urcorner$. This gives us the following encoding in Twelf:

```
step/beta : value V2 -> step (app (lam [x] E1 x) V2) (E1 V2).
```

Here, implicitly, `E1 : exp -> exp`, so the representation of e_1 always comes with its bound variable x. We could also have written `lam E1`, which is η-equivalent to `lam [x] E1 x`.

The remaining rules introduce no new ideas, so we just show them in Twelf.

```
val/lam : value (lam ([x] E x)).
step/app1 : step E1 E1' -> step (app E1 E2) (app E1' E2).
step/app2 : value V1 -> step E2 E2' -> step (app V1 E2) (app V1 E2').
step/beta : value V2 -> step (app (lam [x] E1 x) V2) (E1 V2).

val/pair : value E1 -> value E2 -> value (pair E1 E2).
step/pair1 : step E1 E1' -> step (pair E1 E2) (pair E1' E2).
step/pair2 : value E1 -> step E2 E2' -> step (pair E1 E2) (pair E1 E2').

step/case/pair0 : step E1 E1' -> step (case/pair E1 B) (case/pair E1' B).
step/case/pair : value (pair V1 V2) -> step (case/pair (pair V1 V2) B) (B V1 V2).
```

4 Input/Output Modes

Before we move on to the typing judgment, we consider modes. We had checked for our judgment that e^+ *value* and $e^+ \mapsto e'^-$ so that the two judgments admit a computational interpretation for checking that expression is a value, and for stepping a value. Twelf can check the mode for us.

```
value : exp -> type.
step : exp -> exp -> type.
%mode value +E.
%mode step +E -E'.
```

We can also execute a query to construct, in a bottom-up manner, a stepping derivation for us.

```
id : exp = (lam [x] x).
k : exp = (lam [x] lam [y] x).

%query 1 * step (app k id) E'.
```

The query asks the Twelf server to verify that there is exactly one instance e' and deduction of $\mathsf{k}\ \mathsf{id} \mapsto e'$. It does that and reports

```
E' = lam ([y:exp] id).
```

which is the correct answer.

This highlights a point of comparison between functional programming and logical frameworks: while values of type $\tau \to \sigma$ are not directly observable in LAMBDA, in a logical framework they are. This harkens back to our original investigation of the λ-calculus and Church numerals, where we also assumed that the structure of functions is observable.

5 Representing Hypothetical Judgments

The typing judgment $\Gamma \vdash e : \tau$ requires us to confront the nature of *hypothetical judgments*, represented here by the context Γ. The technique for representing them extends the idea for representing variables. Starting with the straightforward, we declare

```
of : exp -> tp -> type.
```

For example,

$$\dfrac{\dfrac{\dfrac{}{x : \tau \vdash x : \tau}\ \mathsf{tp/var} \quad \dfrac{}{x : \tau \vdash x : \tau}\ \mathsf{tp/var}}{x : \tau \vdash \langle x, x \rangle : \tau \times \tau}\ \mathsf{tp/pair}}{\cdot \vdash \lambda x.\, \langle x, x \rangle : \tau \to (\tau \times \tau)}\ \mathsf{tp/lam}$$

would be represented as (omitting implicit indices)

$$\mathsf{tp/lam}\ (\lambda x.\, \lambda u.\, \mathsf{tp/pair}\ u\ u)$$

where $x : \mathsf{exp}$ and $u : \mathsf{of}\ x\ \ulcorner\tau\urcorner$.

Another way to put this is that hypotheses are variables ranging over deductions. As such, they are represented by variables in the metalanguage. The signature of the typing judgment using this representation is shown in Listing 1.

Perhaps surprisingly, the `of E T` judgment can still be used algorithmically even though `T` is neither input nor output. Proof construction will collect and solve equations over `T`. Because types are quite simple here, this is sufficient for a kind of Hindley-Milner type inference. For example, the first query finds the most general type of the identity function which is $\tau_1 \to \tau_1$ for any type τ_1, and the second query verifies that there is no type for $\lambda x.\, x\, x$.

```
%query 1 * of id A.
---------- Solution 1 ----------
A = arrow T1 T1.
```

```
tp/pair : of E1 T1
           -> of E2 T2
           -> of (pair E1 E2) (times T1 T2).
tp/case/pair : of E (times T1 T2)
                -> ({x:exp} of x T1 -> {y:exp} of y T2 -> of (E' x y) S)
                -> of (case/pair E ([x] [y] E' x y)) S.
tp/lam : ({x} of x T1 -> of (E x) T2)
          -> of (lam [x] E x) (arrow T1 T2).
tp/app : of E1 (arrow T2 T1)
          -> of E2 T2
          -> of (app E1 E2) T1.
```

Listing 1: Typing of functions and pairs

```
% query 0 * of (lam [x] app x x) A.
% no solution
```

We could also give a specification of bidirectional type-checking which is much more flexible, with the expected modes for synthesis and checking. It can easily be related to the typing embodied in the rules in this section.

6 Representing Metatheory

So far we have seen how to represent syntax and deductions, and how to give a computational interpretation to some well-moded judgments via proof construction. What remains is to illustrate how we represent metatheory. Let's recall the theorem of type preservation.

Theorem 1 (Type Preservation) *If* $\cdot \vdash e : \tau$ ($\mathcal{D}$) *and* $e \mapsto e'$ ($\mathcal{S}$) *then* $\cdot \vdash e' : \tau$ ($\mathcal{D}'$).

The proof is by rule induction on the derivation $\mathcal{S}$ of $e \mapsto e'$, applying inversion to the typing derivation. It is also possible to prove it by rule induction on the typing derivation and apply inversion to the stepping judgment.

How do we represent this in Twelf? First, we notice that the proof of type preservation induces a *total function* from derivations of $\cdot \vdash e : \tau$ and $e \mapsto e'$ to a derivation of $\cdot \vdash e' : \tau$. This function can **not** be represented as such in LF because the function space only allows us to use variable binding and substitution, but not functional computations over derivations. We could try to enrich the framework by a functional, computational layer, an approach taken by Beluga [Pientka and Cave, 2015]. In Twelf we represent this function instead as a *relation*, that is, a judgment relating the three derivations to each other. In short:

```
pres : of E T -> step E E' -> of E' T -> type.
%mode pres +D +S -D'.
```

The mode already expresses part of the content of the metatheorem: we think of the typing derivation $\mathcal{D}$ and the stepping derivation $\mathcal{S}$ as input, and the typing derivation $\mathcal{D}'$ (the typing derivation of the result of reduction) as output. If we could show that the relation obeys this mode, and furthermore is total in its first two arguments, then we know that preservation must hold.

Let's consider an example case in the proof.

Case:

$$\mathcal{S} \quad = \quad \dfrac{\begin{array}{c}\mathcal{S}_1\\ e_1 \mapsto e_1'\end{array}}{e_1\, e_2 \mapsto e_1'\, e_2}\ \mathsf{step/app}_1$$

$\cdot \vdash e_1\, e_2 : \tau$ (by $\mathcal{D}$) — Given

$\mathcal{D} = \dfrac{\begin{array}{cc}\mathcal{D}_1 & \mathcal{D}_2\\ \cdot \vdash e_1 : \tau_2 \to \tau & \cdot \vdash e_2 : \tau_2\end{array}}{\cdot \vdash e_1\, e_2 : \tau}\ \mathsf{tp/app}$ for some τ_2, $\mathcal{D}_1$, $\mathcal{D}_2$ — By inversion on $\mathcal{D}$

$\cdot \vdash e_1' : \tau_2 \to \tau$ (by $\mathcal{D}_1'$) — By ind. hyp. on $\mathcal{S}_1$ and $\mathcal{D}_1$

$\mathcal{D}' = \dfrac{\begin{array}{cc}\mathcal{D}_1' & \mathcal{D}_2\\ \cdot \vdash e_1' : \tau_2 \to \tau & \cdot \vdash e_2 : \tau_2\end{array}}{\cdot \vdash e_1'\, e_2 : \tau}\ \mathsf{tp/app}$ — By rule $\mathsf{tp/app}$

From this we can construct the following abbreviated description.

If $\mathcal{S} = \mathsf{step/app}_1\ \mathcal{S}_1$
and $\mathcal{D} = \mathsf{tp/app}\ \mathcal{D}_1\ \mathcal{D}_2$
then $\mathcal{D}' = \mathsf{tp/app}\ \mathcal{D}_1'\ \mathcal{D}_2$
where $\mathcal{D}_1'$ is the result of the induction hypothesis on $\mathcal{D}_1$ and $\mathcal{S}_1$ yielding $\mathcal{D}_1'$.

We can transliterate this into Twelf, where e_1, e_2, e_1', τ_2, and τ are determined by inference.

```
pres : of E T -> step E E' -> of E' T -> type.
%mode pres +D +S -D'.

pres/step/app1 : pres (tp/app D1 D2) (step/app1 S1) (tp/app D1' D2)
                  <- pres D1 S1 D1'.
```

The remaining cases are similar, with the most difficult when the proof include substituting a derivation for the use of hypotheses. Since hypotheses are represented by variables and therefore bound in the representation, we can use metalevel substitution to implement the object level substitution of derivations.

```
pres/beta      : pres (tp/app (tp/lam D1) D2) (step/beta W2) (D1 V2 D2).
```

Here,

$$D_1 : \Pi x : \mathsf{exp.\ of}\ x\ T_2 \to \mathsf{of}\ (E_1\, x)\ T$$

where $E_1 = \lambda x.\ulcorner e_1 \urcorner$, $T = \ulcorner \tau \urcorner$, and $T_2 = \ulcorner \tau_2 \urcorner$. So we can apply D_1 to $\ulcorner v_2 \urcorner = V_2$ to obtain

$$D_1\, V_2 : \mathsf{of}\ V_2\ T_2 \to \mathsf{of}\ (E_1\, V_2)\ T$$

and then to $D_2 : \mathsf{of}\ V_2\ T_2$ to get

$$D_1\, V_2\, D_2 : \mathsf{of}\ (E_1\, V_2)\ T$$

```
pres : of E T -> step E E' -> of E' T -> type.
%mode pres +D +S -D'.

pres/step/pair1 : pres (tp/pair D1 D2) (step/pair1 S1) (tp/pair D1' D2)
                   <- pres D1 S1 D1'.
pres/step/pair2 : pres (tp/pair D1 D2) (step/pair2 W1 S2) (tp/pair D1 D2')
                   <- pres D2 S2 D2'.
pres/case/pair0 : pres (tp/case/pair D0 D1) (step/case/pair0 S0) (tp/case/pair D0' D1)
                   <- pres D0 S0 D0'.
pres/case/pair : pres (tp/case/pair (tp/pair D1 D2) D3) (step/case/pair W12) (D3 V1 D1 V2 D

pres/step/app1 : pres (tp/app D1 D2) (step/app1 S1) (tp/app D1' D2)
                  <- pres D1 S1 D1'.
pres/step/app2 : pres (tp/app D1 D2) (step/app2 W1 S2) (tp/app D1 D2')
                  <- pres D2 S2 D2'.
pres/beta      : pres (tp/app (tp/lam D1) D2) (step/beta W2) (D1 V2 D2).
```

Listing 2: Relational representation of type preservation

which is the required output derivation for β-reduction.

The other cases are similar, so we just summarize them in Listing 2.

It remains to show that this is indeed a proof of preservation. In Twelf, this is decomposed into four steps:

1. The higher-level judgment is well-moded with respect to input (universal quantification) and output (existential quantification). This holds here as shown.

2. The judgment respects hypotheses in the judgment. In this example, all expression are closed, which is expressed by the empty parentheses in the worlds declaration.

   ```
   %worlds () (pres D S D').
   ```

3. The higher-level rules cover all possible cases for the input. This verifies not only the possible stepping derivations, but also the appeals to inversion in the mathematical proof. In Twelf, this is called a *coverage check*.

4. Some measure decreases in the premises of the higher-level rules. In this case, the mathematical proof is by rule induction on $\mathcal{S}$, so the premises can only reference subderivations of $\mathcal{S}$. In Twelf, this is called a *termination check*. We can easily see that it holds.

The declaration

```
%total S (pres D S D').
```

combines coverage checking for $\mathcal{D}$ and $\mathcal{S}$ together with termination check on $\mathcal{S}$. Everything is accepted by Twelf, so the proof of preservation has been verified.

We were able to carry out this formal proof for a nontrivial fragment of the LAMBDA language within a single lecture. You can see the complete code in lambda.elf. The remainder of the language has more cases, but requires no fundamentally different representation techniques. Proving progress requires just a little more effort because the conclusion of the progress theorem involves a disjunction. Since we don't have this directly available in Twelf, we have to represent it with an auxiliary judgment. Adding subtyping is more difficult since its rules are interpreted coinductively, but possible.

Where we hit real obstacles are logical relations and parametricity. See Rabe and Sojakova [2013] and Cave and Pientka [2018] for two different approaches.

7 The Structure of LF

We give here a very brief summary of the syntax of LF. We have *type families* a that are indexed by terms $M_1 \dots M_n$. If $n = 0$ we have just an ordinary type. We classify type families by declaring the types of their indices. The function type $A \to B$ is just a special case of $\Pi x{:}A.\, B$ when x does not occur in B. Signatures collect declarations for type families a and term constants c.

$$
\begin{array}{lllll}
\text{Kinds} & K & ::= & \Pi x{:}A.K \mid A \to K \mid \mathbf{type} \\
\text{Types} & A, B & ::= & \Pi x{:}A.\, B \mid A \to B \mid a\, M_1 \dots M_n \\
\text{Terms} & M, N & ::= & c \mid x \mid \lambda x.\, M \mid M\, N \\
 & & & \\
\text{Signatures} & \Sigma & ::= & \cdot \mid \Sigma, a : K \mid \Sigma, c : A \\
\text{Contexts} & \Gamma & ::= & \cdot \mid \Gamma, x : A
\end{array}
$$

We show only two typing rules because they illustrate the dependently typed nature of LF for the judgment $\Gamma \vdash_\Sigma M : A$.

$$
\frac{\Gamma, x : A \vdash_\Sigma M : B}{\Gamma \vdash_\Sigma \lambda x.\, M : \Pi x{:}A.\, B}\ \Pi I
\qquad\qquad
\frac{\Gamma \vdash_\Sigma M : \Pi x{:}A\, B \quad \Gamma \vdash_\Sigma N : B}{\Gamma \vdash_\Sigma M\, N : [N/x{:}A]B}\ \Pi E
$$

The operation $[N/x{:}A]B$ is called *hereditary substitution* [Watkins et al., 2002] that not only substitutes N for x in B, but always returns a canonical form. That allows us to keep types and the terms embedded in them always in canonical form which eliminates the need for explicit equational reasoning during type-checking.

References

David Basin and Seán Matthews. Logical frameworks. In Dov Gabbay and Franz Guenthner, editors, *Handbook of Philosophical Logic*. Kluwer Academic Publishers, 2nd edition, 2001.

Andrew Cave and Brigitte Pientka. Mechanizing proofs with logical relations—Kripke-style. *Mathematical Structures in Computer Science*, 28(9):1606–1638, October 2018.

N.G. de Bruijn. Lambda-calculus notation with nameless dummies: a tool for automatic formula manipulation with application to the Church-Rosser theorem. *Indagationes Mathematicae*, 34(5): 381–392, 1972.

Robert Harper, Furio Honsell, and Gordon Plotkin. A framework for defining logics. In *Symposium on Logic in Computer Science*, pages 194–204. IEEE Computer Society Press, June 1987.

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.

Frank Pfenning. Logical frameworks. In Alan Robinson and Andrei Voronkov, editors, *Handbook of Automated Reasoning*, chapter 17, pages 1063–1147. Elsevier Science and MIT Press, 2001.

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.

Brigitte Pientka and Andrew Cave. Inductive Beluga: Programming proofs. In A. Felty and A. Middeldorp, editors, *25th International Conference on Automated Deduction (CADE 2015)*, pages 272–281, Berlin, Germany, August 2015. Springer LNCS 9195.

Florian Rabe and Kristina Sojakova. Logical relations for a logical framework. *Transactions on Computational Logic*, 14(4):1–34, November 2013.

Twelf. The Twelf project. Available at http://twelf.org/wiki/Main_Page. Accessed January 15, 2024.

Kevin Watkins, Iliano Cervesato, Frank Pfenning, and David Walker. A concurrent logical framework I: Judgments and properties. Technical Report CMU-CS-02-101, Department of Computer Science, Carnegie Mellon University, 2002. Revised May 2003.