Lecture Notes on Logical Frameworks

15-836: Substructural Logics
Frank Pfenning

Lecture 20
November 28, 2023

1 Introduction

A *logical framework* consists of a *formal metalanguage* for the definition of logics and other deductive systems and a *representation methodology*. The seminal work on logical frameworks is LF [Harper et al., 1987, 1993], with a full-scale implementation in the Twelf system [Pfenning and Schürmann, 1999], available at `www.twelf.org`. Logical frameworks distill the essence of the conceptual notions that are used to define logics and other deductive systems, such as the statics and dynamics of programming languages. They are distinguished from general type theories and their implementations in systems such as Agda[1] and Coq[2] in that they designed for the specific domain of deductive systems rather than general (constructive or classical) mathematics.

LF itself is *structural*, and this limitation has led to substructural generalizations in the form of Linear LF (LLF) [Cervesato and Pfenning, 1996, 2002][3] and Concurrent LF (CLF) [Watkins et al., 2002, Cervesato et al., 2002, Schack-Nielsen and Schürmann, 2008, Schack-Nielsen, 2011][4] Each of these is designed to address some shortcomings of its predecessors, suitably extending both the formal metalanguage and the representation methodology.

We will follow a similar path, introducing LF in today's lecture and then consider substructural extensions in the next two lectures. We emphasize the universality of the underlying principles, which mirror the principles we have employed throughout in this course. One might even say that these principles are manifest in the design of the logical frameworks we represent.

[1] `https://wiki.portal.chalmers.se/agda/pmwiki.php`
[2] `https://coq.inria.fr/`
[3] `https://github.com/clf/llf`
[4] `https://github.com/clf/celf`

What is a logical framework used for? In this and the three remaining lectures we don't have the time to cover the full rangle of applications, but we can broadly categorize them as follows:

Definition: Logical frameworks are used to *define* logics and other deductive systems under consideration, ideally at a very high level of abstraction. Logics are generally characterized by propositions and rules of inference, programming languages by programs, type systems, and rules of computation. The main principle used in the definition of logics is summarized as *judgments as types* and *proofs as objects*.

Algorithms: The most fundamental algorithm is that of proof checking for an object logic represented in a logical framework, but there are many others such as proof search, proof reduction, translations between logics, type checking, or evaluation of programs on an object language. The logical frameworks we consider support algorithms via *computation as proof construction* which encompasses both backward [Miller et al., 1991, Miller and Nadathur, 2012] and forward proof construction [López et al., 2005].

Metareasoning: Once a deductive system has been defined, we usually prove a number of important properties of it. For logics, these include cut elimination, identity elimination, focusing, soundness and completeness of translations, etc. For programming languages they are progress and preservation, soundness and completeness of type-checking algorithms, compiler correctness, etc. We can exploit the nature of representations in logical framework to formally prove such metatheorems Pfenning and Schürmann [1999], Schürmann [2000]. The general methodology is to represent that computational content of proof of the metatheorem algorithmically and then verify its totality. There are some gaps in our understanding of how to achieve this for substructural frameworks (see some approaches by McCreight and Schürmann [2008], Reed [2009], Georges et al. [2017])

For today's lecture where we only consider LF we will focus on logic definition and proof checking.

2 Judgments as Types

One of the fundamental representation techniques is *judgments as types*. A *judgment* in this context are what is subject to deductive inference, as mapped out by Martin-Löf [1983]. Common judgments are $A\ \mathit{true}$ or $A\ \mathit{false}$ or $A\ \mathit{valid}$. We use the very simply example from Lecture 1 of defining a path through a directed graph.

$$\frac{\mathsf{edge}(x,y)}{\mathsf{path}(x,y)}\ \mathsf{step} \qquad \frac{\mathsf{path}(x,y) \quad \mathsf{path}(y,z)}{\mathsf{path}(x,z)}\ \mathsf{trans}$$

We previously thought of edge(x, y) and path(x, y) as propositions, but we will now think of them as judgments since they are directly subject to inference. This means that edge(x, y) and path(x, y) for vertices x and y should be represented by *types* in our formal metalanguage (which we have yet to define). We think of both edge and path as constructors for types, taking *vertices* as arguments. So we have

```
vertex : type.
edge : vertex -> vertex -> type.
path : vertex -> vertex -> type.
```

In the terminology of logical frameworks we call edge and path *type families*, each indexed by two vertices.

It is easy to see what the type vertex represents. Here is the example from Lecture 1: Our initial state of knowledge is edge(a, b), edge(b, c), edge(b, d) for some vertices a, b, c, and d. Therefore:

```
a : vertex.
b : vertex.
c : vertex.
d : vertex.
```

We also have to have objects of type edge a b, edge b c and edge b d. These are constants represent (trivial) *proofs* of these judgments.

```
eab : edge a b.
ebc : edge b c.
ebd : edge b d.
```

What happens to the inference rules? They become *proof constructors*. As a first approximation, we might write

```
step : edge x y -> path x y.
trans : path x y -> path y z -> path x z.
```

It remains to clarify the status of x, y and z. Somehow we have to express that any instantiation of the constructor with vertices x, y and z is a valid instance of the rule.

When we developed predicate calculus, we used universal quantification to express this, where we purposely let the quantifier range over arbitrary "individuals". In LF we would like to be more precise and specify that they must be vertices. Syntactically, we just replace $\forall x.\, B(x)$ with $\Pi x{:}A.B(x)$ where A is a type. Our concrete syntax for this is $\{x : A\}B$.

```
step : {x:vertex} {y:vertex} edge x y -> path x y.
trans : {x:vertex} {y:vertex} {z:vertex}
           path x y -> path y z -> path x z.
```

Now we can show a little example of a proof representation. We represent

$$\dfrac{\dfrac{}{\mathsf{edge}(a,b)}\,\mathsf{eab}}{\mathsf{path}(a,b)}\,\mathsf{step}$$

as

```
step a b eab : path a b
```

Here is a slightly larger proof:

$$\dfrac{\dfrac{\dfrac{}{\mathsf{edge}(a,b)}\,\mathsf{eab}}{\mathsf{path}(a,b)}\,\mathsf{step} \quad \dfrac{\dfrac{}{\mathsf{edge}(b,c)}\,\mathsf{ebc}}{\mathsf{path}(b,c)}\,\mathsf{step}}{\mathsf{path}(a,c)}\,\mathsf{trans}$$

which becomes

```
trans a b c (step a b eab) (step b c ebc) : path a c
```

3 The Formal Metalanguage

So far, we have learned about "*judgments as types*" and "*proofs as objects*" through a very simple example. A proof of a judgment is represented by an object of the corresponding type. Clearly, this should be a *bijection*: every valid proof should be an object that has the expected type, and every object that has a given type should represent a type. If typing in the framework is decidable (which it will be) this means we can model proof checking by type checking.

Before we go further, we should be more precise about the metalanguage that we have written some code in without actually defining it. Even though LF was not originally conceived this way, we think of it as arising from [*drum roll*] *focusing*. This point of view is helpful because it will extend to the substructural frameworks we start discussing in the next lecture.

What have we used to far? We have used atomic types vertex, edge x y and path x y. We have also used function types $A \to B$ and quantification $\Pi x{:}A.B(x)$ that generalizes $\forall x.\, B(x)$. We observe that all of these are *negative*! This also alleviates any stress regarding atoms: let's just be consistent and also make them negative. Here is what we have so far.

Negative types	A, B	$::=$	$P \mid A \to B \mid \Pi x{:}A.\, B(x)$
Atoms	P	$::=$	$\ldots$
Objects	M	$::=$	$\ldots$
Kinds	K	$::=$	$\mathbf{type} \mid A \to K \mid \ldots$
Signatures	Σ	$::=$	$\cdot \mid \Sigma, a : K \mid \Sigma, c : A$

A *signature* has declarations for term constructors c and also for type families a that may depend on objects like edge and path.

The language of objects (and, by analogy, the language of types) is now determined by what it means to *focus* on a type among the antecedents and what it means to *invert* a type as a succedent. The declarations $c : A$ in signature Σ (which is generally fixed for a particular encoding) act as antecedents.

We start with left focus, first the rules that starts left focus from a *stable sequent*. At the end of this section we will see what the succedent δ of a stable sequent must look like.

$$\frac{c : A \in \Sigma \quad \Gamma, [A] \vdash_\Sigma \delta}{\Gamma \vdash_\Sigma \delta}\ \mathsf{FL/C} \qquad \frac{c : A \in \Sigma \quad \Gamma, [A] \vdash_\Sigma S : \delta}{\Gamma \vdash_\Sigma c\ S : \delta}\ \mathsf{FL/C}$$

The kind of proof term to we assign to the left focus judgment is called a *spine* [Cervesato and Pfenning, 2003], which we write as S. This harkens back to earlier term assignments, although with a different purpose.

We now omit the signature Σ from the turnstile for brevity since it never changes in the typing of objects and spines.

$$\frac{\Gamma \vdash [A] \quad \Gamma, [B] \vdash \delta}{\Gamma, [A \to B] \vdash \delta}\ {\to}L \qquad \frac{\dfrac{\Gamma \vdash M : A}{\Gamma \vdash M : [A]}\ \mathsf{IR/FR} \quad \Gamma, [B] \vdash S : \delta}{\Gamma, [A \to B] \vdash (M\ ;\ S) : \delta}\ {\to}L$$

Since A is negative, we will lose focus on $[A]$ in the first premise and start inversion which is the judgment to type objects (not spines).

Universal quantification is interesting. Recall that a proof of $\forall x.\, B(x)$ was a function which for every individual t returned a proof of $B(t)$. So both quantification and implication correspond to functions. Here, there is no separate class of terms t—we just use objects M.

$$\frac{\dfrac{\Gamma \vdash M : A}{\Gamma \vdash M : [A]}\ \mathsf{IR/FR} \quad \Gamma, [B(M)] \vdash S : \delta}{\Gamma, [\Pi x{:}A.\, B(x)] \vdash (M\ ;\ S) : \delta}\ \Pi L$$

The tricky part of this rule is the substitution $B(M)$. So the only difference between $A \to B$ and $\Pi x{:}A.\, B(x)$ that in the latter, B may depend on x, while not so in the former. We discuss this operation further in Section 4.

For the final left rule, we consider atoms, which in LF are all considered negative. The left focus only succeeds if the succedent is the same suspended atom. Because of this, there is no real information content in the rule and the spine is just empty.

$$\frac{}{\Gamma, [P] \vdash \langle P \rangle}\ \mathsf{id}^- \qquad \frac{}{\Gamma, [P] \vdash (\,) : \langle P \rangle}\ \mathsf{id}^-$$

We revisit our grammar. Atoms are just like constants applied to spines, except that the constant itself is a type family. Also, variables can be used just like constants.

Negative types	A, B	$::=$	$P \mid A \to B \mid \Pi x{:}A.\, B(x)$
Atoms	P	$::=$	$a\ S$
Objects	M	$::=$	$c\ S \mid x\ S \mid \ldots$
Spines	S	$::=$	$(M\ ;\ S) \mid (\,)$
Kinds	K	$::=$	$\mathbf{type} \mid A \to K \mid \ldots$
Signatures	Σ	$::=$	$\cdot \mid \Sigma, a : K \mid \Sigma, c : A$

With this, we can give a formal representation of our earlier example, abbreviating $c\ (\,)$ and $x\ (\,)$ as just c and x. We also omit trailing empty spines and write $(M_1\ ;\ \ldots\ ;\ M_n\ ;\ (\,))$ as $(M_1\ ;\ \ldots\ ;\ M_n)$.

$\mathsf{vertex} : \mathbf{type}$
$\mathsf{edge} : \mathsf{vertex} \to \mathsf{vertex} \to \mathbf{type}$
$\mathsf{path} : \mathsf{vertex} \to \mathsf{vertex} \to \mathbf{type}$
$\mathsf{step} : \Pi x{:}\mathsf{vertex}.\, \Pi y{:}\mathsf{vertex}.\, \mathsf{edge}\ (x\ ;\ y) \to \mathsf{path}\ (x\ ;\ y)$
$\mathsf{trans} : \Pi x{:}\mathsf{vertex}.\, \Pi y{:}\mathsf{vertex}.\, \Pi z{:}\mathsf{vertex}.$
$\qquad \mathsf{path}\ (x\ ;\ y) \to \mathsf{path}\ (y\ ;\ z) \to \mathsf{path}\ (x\ ;\ z)$

$a : \mathsf{vertex}$
$b : \mathsf{vertex}$
$c : \mathsf{vertex}$
$d : \mathsf{vertex}$

$\mathsf{eab} : \mathsf{edge}\ (a\ ;\ b)$
$\mathsf{ebc} : \mathsf{edge}\ (b\ ;\ c)$
$\mathsf{ebd} : \mathsf{edge}\ (b\ ;\ d)$

$\vdash \mathsf{trans}\ (a\ ;\ b\ ;\ c\ ;\ (\mathsf{step}\ (a\ ;\ b\ ;\ \mathsf{eab}))\ ;\ (\mathsf{step}\ (b\ ;\ c\ ;\ \mathsf{ebc}))) : \mathsf{path}\ (a\ ;\ b)$

There is still a lot of redundancy in this representation with multiple occurrences of a, b, and c, but implementations can further mitigate this by allowing the user to elide some of these, and in some cases even eliminate them from the representation altogether.

Even if it is only needed to suspend atoms in this example, we should return to the inversion phase of focusing. Since all constructors are negative, these will be the right rules. For the same reason, we can dispense with the usual ordered antecedents Ω. For example, the right rule for $A \to B$ would add A to the ordered context, but since A is negative and therefore stable, it will be immediately transferred to the structural context Γ that consists entirely of negative types (since suspended positive atoms are not part of the language).

$$\frac{\Gamma, A \vdash B}{\Gamma \vdash A \to B}\ {\to}R \qquad \frac{\Gamma, x : A \vdash M(x) : B}{\Gamma \vdash \lambda x.\, M(x) : A \to B}\ {\to}R$$

As we might expect by now, quantifiers are just dependent function types can behave the same.

$$\frac{\Gamma, x : A \vdash M(x) : B(x)}{\Gamma \vdash \lambda x.\, M(x) : \Pi x{:}A.\, B(x)}\ \Pi R$$

Finally, atoms on the right are suspended because their are negative.

$$\frac{\Gamma \vdash \langle P \rangle}{\Gamma \vdash P}\ \mathsf{C/IR} \qquad \frac{\Gamma \vdash M : \langle P \rangle}{\Gamma \vdash M : P}\ \mathsf{C/IR}$$

We see that in stable sequents the succedent δ always has the form $\langle P \rangle$ for some atom P.

This allows us to complete the grammar, where we additional allow kinds to be dependent.

Negative types	A, B	$::=$	$P \mid A \to B \mid \Pi x{:}A.\, B(x)$
Atoms	P	$::=$	$a\ S$
Objects	M	$::=$	$c\ S \mid x\ S \mid \lambda x.\, M(x)$
Spines	S	$::=$	$(M\ ;\ S) \mid (\,)$
Kinds	K	$::=$	$\mathbf{type} \mid A \to K \mid \Pi x{:}A.\, B(x)$
Stable antecedents	Γ	$::=$	$\cdot \mid \Gamma, x : A$
Stable succedents	δ	$::=$	$\langle P \rangle$
Signatures	Σ	$::=$	$\cdot \mid \Sigma, a : K \mid \Sigma, c : A$

We have already seen the most critical typing rules; they are summarized in Figure 1. Others are similar and elided and can be found in the literature. Still missing is the definition of $B(M)$, which looks like ordinary substitution but is more complicated.

In the next lecture we will extend the representation methodology and show some representations of the sequent calculus and the semi-axiomatic sequent calculus.

4 Hereditary Substitution

Consider a type $\Pi x{:}A.\, B(x)$. When typing an application we need to substitute a term $M : A$ for x, written so far as $B(M)$. But does this substitution actually make sense? Consider the term $x\ (\,)$ that is typed from $x : P$ with left focus. Just plugging in the object $M : P$ would be $M\ (\,)$, but that's not actually a valid object. Similarly, if $A = P \to Q$ then the term will be $\lambda y.\, N(y)$ and after just plugging into $x\ (M\ ;\ (\,))$ we would have $(\lambda y.\, N(y))\ (M\ ;\ (\,))$, again not even syntactically valid.

We use a more traditional notation $[M/x]_A B(x)$ instead of $B(M)$, indexing the operation also with the type A of x. This quickly reduces to $[M/x]_A N$ and $[M/x]_A S$. The idea is that if x is at the head of a spine we then reduce further, initiating more

$$\frac{c : A \in \Sigma \quad \Gamma, [A] \vdash S : \delta}{\Gamma \vdash c\ S : \delta}\ \mathsf{FL/C} \qquad \frac{x : A \in \Gamma \quad \Gamma, [A] \vdash S : \delta}{\Gamma \vdash c\ S : \delta}\ \mathsf{FL/C/}$$

$$\frac{\Gamma \vdash M : A \quad \Gamma, [B] \vdash S : \delta}{\Gamma, [A \to B] \vdash (M\ ;\ S) : \delta}\ {\to}L \qquad \frac{\Gamma \vdash M : A \quad \Gamma, [B(M)] \vdash S : \delta}{\Gamma, [\Pi x{:}A.\ B(x)] \vdash (M\ ;\ S) : \delta}\ \Pi L$$

$$\frac{}{\Gamma, [P] \vdash (\,) : \langle P \rangle}\ \mathsf{id}^-$$

$$\frac{\Gamma, x : A \vdash M(x) : B}{\Gamma \vdash \lambda x.\ M(x) : A \to B}\ {\to}R \qquad \frac{\Gamma, x : A \vdash M(x) : B(x)}{\Gamma \vdash \lambda x.\ M(x) : \Pi x{:}A.\ B(x)}\ \Pi R$$

$$\frac{\Gamma \vdash M : \langle P \rangle}{\Gamma \vdash M : P}\ \mathsf{C/IR}$$

Figure 1: LF Type Theory (excerpt), given a fixed signature Σ

substitutions and so on. Why does this terminate? Similar to cut elimination, it is by a nested induction first on the type A and second on the object M and spine S we substitute into. In fact, it is the operational reading of cut elimination for the focusing calculus on negative types.

We write h for a *head*, that is, a constant c or a variable y.

$$\begin{array}{lcll}
[M/x]_A(\lambda y.\ N) & = & \lambda y.\ [M/x]_A N & y \text{ not free in } M \\
[M/x]_A(h\ S) & = & h\ [M/x]_A S & \text{where } x \neq h \\
[M/x]_A(x\ S) & = & M\ |_A\ [M/x]_A S & \text{(application)} \\
[M/x]_A(N\ ;\ S) & = & [M/x]_A N\ ;\ [M/x]_A S & \\
[M/x]_A(\,) & = & (\,) & \\
 & & & \\
(h\ S)\ |_P\ (\,) & = & h\ S & \\
(\lambda x.\ M)\ |_{A \to B}\ (N\ ;\ S) & = & [N/x]_A M\ |_B\ S & \\
(\lambda x.\ M)\ |_{\Pi x:A.\ B(x)}\ (N\ ;\ S) & = & [N/x]_A M\ |_{B(x)}\ S &
\end{array}$$

The condition in the first case can be satisfied by renaming the bound variable y, which is always (silently) possible. For the purpose of hereditary substitution, ordinary and dependent function types are treated identically; the free variable x in $B(x)$ is not relevant to the termination argument.

Another interesting point is that hereditary substitution may be undefined, but is always computable by the nested induction argument. The notion of hereditary substitution was originally developed for a substructural logical framework [Watkins et al., 2002] which contains LF as a fragment.

References

Iliano Cervesato and Frank Pfenning. A linear logical framework. In E. Clarke, editor, *Proceedings of the Eleventh Annual Symposium on Logic in Computer Science*, pages 264–275, New Brunswick, New Jersey, July 1996. IEEE Computer Society Press.

Iliano Cervesato and Frank Pfenning. A linear logical framework. *Information & Computation*, 179(1):19–75, November 2002. Revised and expanded version of an extended abstract, LICS 1996, pp. 264-275.

Iliano Cervesato and Frank Pfenning. A linear spine calculus. *Journal of Logic and Computation*, 13(5):639–688, 2003.

Iliano Cervesato, Frank Pfenning, David Walker, and Kevin Watkins. A concurrent logical framework II: Examples and applications. Technical Report CMU-CS-02-102, Department of Computer Science, Carnegie Mellon University, 2002. Revised May 2003.

Aïna Linn Georges, Agata Murawska, Shawn Otis, and Brigitte Pientka. LINCX: A linear logical frameworks with first-class contexts. In Hongseok Yang, editor, *26th European Symposium on Programming (ESOP 2017)*, pages 530–555, Uppsala, Sweden, April 2017. Springer LNCS 10201.

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.

Pablo López, Frank Pfenning, Jeff Polakow, and Kevin Watkins. Monadic concurrent linear logic programming. In A.Felty, editor, *Proceedings of the 7th International Symposium on Principles and Practice of Declarative Programming (PPDP'05)*, pages 35–46, Lisbon, Portugal, July 2005. ACM Press.

Per Martin-Löf. On the meanings of the logical constants and the justifications of the logical laws. Notes for three lectures given in Siena, Italy. Published in *Nordic Journal of Philosophical Logic*, 1(1):11-60, 1996, April 1983. URL http://www.hf.uio.no/ifikk/forskning/publikasjoner/tidsskrifter/njpl/vol1no1/meaning.pdf.

Andrew McCreight and Carsten Schürmann. A meta linear logical framework. In *4th International Workshop on Logical Frameworks and Meta-Languages (LFM 2004)*, volume 199 of *Electronic Notes in Theoretical Computer Science*, pages 129–147, February 2008.

Dale Miller and Gopalan Nadathur. *Programming with Higher-Order Logic*. Cambridge University Press, 2012.

Dale Miller, Gopalan Nadathur, Frank Pfenning, and Andre Scedrov. Uniform proofs as a foundation for logic programming. *Annals of Pure and Applied Logic*, 51:125–157, 1991.

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.

Jason C. Reed. *A Hybrid Logical Framework*. PhD thesis, Carnegie Mellon University, September 2009. Available as Technical Report CMU-CS-09-155.

Anders Schack-Nielsen. *Implementing Substructural Logical Frameworks*. PhD thesis, IT University of Copenhagen, January 2011.

Anders Schack-Nielsen and Carsten Schürmann. Celf - a logical framework for deductive and concurrent systems. In A. Armando, P. Baumgartner, and G. Dowek, editors, *Proceedings of the 4th International Joint Conference on Automated Reasoning (IJCAR'08)*, pages 320–326, Sydney, Australia, August 2008. Springer LNCS 5195.

Carsten Schürmann. *Automating the Meta Theory of Deductive Systems*. PhD thesis, Department of Computer Science, Carnegie Mellon University, August 2000. Available as Technical Report CMU-CS-00-146.

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.