The LF Seminar

Meeting 1

Term Representation

Lecture: Frank Pfenning
Notes: Robert J. Simmons

February 20, 2007

We describe *spine form LF*, a version of LF syntactically restricted to canonical forms, and use this framework to begin a logical presentation of the term language used within Twelf, which is (among other things) an implementation of LF. This is not a self-contained document by any means, it is incomplete and furthermore assumes various levels of background in focusing systems, dependent types, LF, the Twelf metalogical framework, and logic programming.

1.1 Canonical LF

Modern presentations of LF generally use the *canonical LF* style developed by Watkins et al. in [9] and most recently presented in [4]; the syntax of canonical LF is reviewed briefly in Figure 1. This presentation syntactically restricts LF to only allow the existence of canonical (β-normal, η-long) forms while preserving the natural deduction style of the the earliest presentations of LF [3]. This presentation is, however, problematic in terms of an implementation for a number of reasons. One reason is that, in any implementation of LF, type checking, term reconstruction, logic programming, theorem proving, etc. will all frequently need to compare two terms for equality, for instance:

$$((c\ M_1)\ M_2)\ M_3 = \big(\big(c'\ M_1'\big)\ M_2'\big)\ M_3'$$

In order to compare the two atomic terms, the implementation must "burrow down" into the two terms to get at c and c' to check them for equality.

$$
\begin{array}{rrcl}
\text{Kinds} & K & ::= & \mathsf{type} \mid \Pi x{:}A.K \\
\text{Atomic Types} & P & ::= & a \mid PM \\
\text{Canonical Types} & A & ::= & P \mid \Pi x{:}A_2.A_1 \\
\text{Atomic Terms} & R & ::= & c \mid x \mid RM \\
\text{Canonical Terms} & M & ::= & R \mid \lambda x.M \\
\\
\text{Contexts} & \Gamma & ::= & \cdot \mid \Gamma, x{:}A \\
\text{Signatures} & \Sigma & ::= & \cdot \mid \Sigma, a{:}K \mid \Sigma, c{:}A
\end{array}
$$

Figure 1: Canonical LF

Another perspective on this complication is that type checking in canonical LF passes information both up and down through a type derivation. Consider, for instance, the typing rule for term application:

$$\frac{\Gamma \vdash R_1 \Rightarrow \Pi x : A_2.A_1 \quad \Gamma \vdash M_2 \Leftarrow A_2 \quad [M_2/x]_{(A_2)^-} A_1 = A}{\Gamma \vdash R_1 M_2 \Rightarrow A}$$

In order to synthesize the type of $R_1 M_2$ (which then moves *down* the derivation tree), the type of R_1 must be synthesized from the left branch and then used to check the type of M_2 against A_2, which passes information up the right branch. This pattern is typical of natural deduction systems. The next section presents *spine form LF*, which corresponds by Curry-Howard to a focusing sequent calculus in the same way that canonical LF corresponds to a natural deduction system. Spine form LF is a way to present "LF with only canonical forms," but in this set of notes we will use "canonical LF" to refer specifically to the presentation in Figure 1 and "spine form LF" to refer to the presentation in the following section.

1.2 Spine form LF

The modern LF presentation is the natural deduction presentation described above; an alternative is the *spine form LF* which was investigated by Cervesato and Pfenning in [2] as both an explanation of the Curry-Howard correspondence for focusing and a means to efficiently implement canonical forms in Linear LF. The presentation in this note borrows heavily from Reed's presentation of spine form LF with proof irrelevance in [7].

$$
\begin{array}{rcll}
\text{Kinds} & K & ::= & \mathsf{type} \mid \Pi x{:}A.K \\
\text{Types} & A, B & ::= & a \cdot S \mid \Pi x{:}A.B \\
\text{Terms} & M & ::= & H \cdot S \mid \lambda x.M \\
\text{Heads} & H & ::= & c \mid x \\
\text{Spines} & S & ::= & \mathsf{nil} \mid M; S \\
 & & & \\
\text{Contexts} & \Gamma & ::= & \cdot \mid \Gamma, x : A \\
\text{Signatures} & \Sigma & ::= & \cdot \mid \Sigma, a : K \mid \Sigma, c : A
\end{array}
$$

Figure 2: Spine form LF

As mentioned in the previous section, spine form LF, like canonical LF, is a presentation of LF where the syntax is restricted to only permit canonical forms. Spine form LF further leverages the nature of η-long terms to separate a head (a bare variable or constant) from the elimination forms (i.e. applications) applied to it. For example, the two atomic terms being checked for equality in the previous section appear in spine form in a way that makes it simple to quickly check c for equality with c'.

$$c \cdot (M_1; M_2; M_3; \mathsf{nil}) = c' \cdot (M_1'; M_2'; M_3'; \mathsf{nil})$$

The effect of this modification, as shown in Figure 2, is to collapse down the series of applications applied to atomic types and terms into a single spine application. We will, however, continue to use P to refer to an atomic type $a \cdot S$ and R to refer to an atomic term $H \cdot S$ where doing so is convenient. Substitution looks a bit different in this presentation than it does in canonical LF presentations. Canonical LF presentations have to deal with the fact that the head of an atomic term R is not immediately apparent, meaning it is not obvious whether substitution will force a reduction step to happen (Watkins deals with this by first checking the term, Harper and Licata do so by presenting hereditary substitution in a non-deterministic style). In the case of substitution in spine form LF, the nature of the head of an atomic term is known immediately, removing this complication. The ability to immediately examine the head of an atomic term also makes erasure to simple types slightly simpler.

$$
\begin{array}{c}
(a \cdot S)^- = a \\
(\Pi x{:}A.B)^- = (A)^- \to (B)^-
\end{array}
$$

$$
\begin{array}{rcll}
\rho(\mathsf{type}) & = & \mathsf{type} & \\
\rho(\Pi y : A.K) & = & \Pi y{:}\rho A.\rho K & \\
\rho(a \cdot S) & = & a \cdot (\rho S) & \\
 & & & \\
\rho(\Pi y : A.B) & = & \Pi y{:}\rho A.\rho B & \\
\rho(c \cdot S) & = & c \cdot (\rho S) & \\
\rho(x \cdot S) & = & (M @_\tau \rho S) & \text{(reduction, remember } \rho \equiv [M/x]_\tau) \\
\rho(y \cdot S) & = & y \cdot (\rho S) & (y \neq x, \text{otherwise reduction starts}) \\
\rho(\lambda y.M) & = & \lambda y.\rho N & \\
 & & & \\
\rho() & = & () & \\
\rho(N; S) & = & (\rho N; \rho S) &
\end{array}
$$

Figure 3: Hereditary substitution for spine form LF. $[M/x]_\tau$ is abbreviated as ρ, and the usual caveats of capture-avoiding substitution apply.

$$
\begin{array}{c}
((\lambda x.M) @_{\tau_1 \to \tau_2} (N; S)) = (([N/x]_{\tau_1} M) @_{\tau_2} S) \\
((H \cdot S) @_a \mathsf{nil}) = (H \cdot S)
\end{array}
$$

Figure 4: Hereditary reduction for spine form LF. If the judgment does not fit into the above forms (i.e. $(H \cdot S) @_\tau (N; S)$ or $(\lambda x.M) @_\tau \mathsf{nil}$) it is an error.

The only interesting case in substitution, defined in Figure 3, is (as usual) the one where a term is substituted for a variable at the head of a term. The reduction judgment $(M @_\tau S)$, defined in Figure 4, can be thought of as normalizing the non-canonical "spine" $M \cdot S$ that has a head M in canonical form (a head which is presumed to have simple type τ) and a spine S of terms in canonical form.

1.3 Typing for spine forms LF

The typing judgments are where the connection between spine forms LF and focusing systems come into play; however, if you aren't well-versed in focused sequent calculi, then these rules should make sense on their own. Because all terms in LF are, depending on who you're talking to, "neg-

ative," "right-asynchronous," or "left-synchronous," we only have to be concerned with two of the four phases of general focusing systems. The right inversion phase corresponds to type checking, and the left focusing phase corresponds to type synthesis. The typing judgments are as follows, annotated with the positions that act as inputs ($+$) and outputs ($-$) during bi-directional typechecking.

	Checking (right inversion)	Synthesis (left focusing)
Terms	$\overset{+}{\Gamma} \vdash \overset{+}{M} \Leftarrow \overset{+}{A}$	$\overset{+}{\Gamma} \vdash \overset{+}{S} : \overset{+}{A} > \overset{-}{P}$
Types	$\overset{+}{\Gamma} \vdash \overset{+}{A} \Leftarrow \overset{+}{K}$	$\overset{+}{\Gamma} \vdash \overset{+}{S} : \overset{+}{K} > \overset{-}{\mathsf{type}}$
Kinds	$\overset{+}{\Gamma} \vdash \overset{+}{K} \Leftarrow \overset{+}{\mathsf{kind}}$	*n/a*

The notation used here attempts to span a gap between a system that looks very much like focusing and a system that looks very much like bi-directional type checking. A sequent that is left-focused on V would be represented in a focused sequent calculus as $\Gamma; V \ll P$, whereas in both Cervesato and Pfenning's presentation, as well as in Reed's presentation, it is represented as $\Gamma \vdash S : V > P$ to concentrate on the fact that a P is considered the *output* in a bi-directional typechecking system, a fact which we will return to in section 1.3.5.

While we adopt the notation $\Gamma \vdash S : V > P$, it is worth mentioning that the proposition we are left-focused on (i.e. V) is presented as a hypothetical assumption in a presentation of focusing without proof terms, and so placing it on the right-hand side of the judgment stroke $\vdash$ is strange in this respect; an alternative would have been to follow Ruy Ley-Wild's (unpublished) formulation of spine form CLF and use the judgment $\Gamma; S : V \ll P$ for the left-focused sequent.

1.3.1 Term validity

The term checking judgment corresponds to the right-inversion phase of focusing.

$$\frac{\Gamma, x : A \vdash M \Leftarrow B}{\Gamma \vdash \lambda x.M \Leftarrow \Pi x{:}A.B}\ \textsc{check-term}$$

Term synthesis is initiated when term checking encounters an atomic term $H \cdot S$. H is then looked up in either the signature or the context, as appropriate.

$$\frac{c : A \in \Sigma \quad \Gamma \vdash S : A > P' \quad P = P'}{\Gamma \vdash c \cdot S \Leftarrow P} \text{ FOC-VAR}$$

$$\frac{x : A \in \Gamma \quad \Gamma \vdash S : A > P \quad P = P'}{\Gamma \vdash x \cdot S \Leftarrow P} \text{ FOC-CON}$$

The rules initiating focusing do not have to explicitly require that the type P be an atomic type $a \cdot S$, even though an atomic term $H \cdot S$ should always have an atomic type. We know that P and P' will both be atomic by virtue of the equality judgment and the fact that the focusing stage can only return an type P' that is of atomic type, generated by the rule INIT-TERM.

$$\frac{\Gamma \vdash M \Leftarrow A \quad \Gamma \vdash S : [M/x]_{(A)^-} B > P}{\Gamma \vdash (M; S) : \Pi x{:}A.B > P} \text{ SYN-TERM}$$

$$\frac{}{\Gamma \vdash \mathsf{nil} : a \cdot S > a \cdot S} \text{ INIT-TERM}$$

1.3.2 Type validity

The rules for type validity are similar; the primary difference is that no equality judgment is needed at the transition to focusing (i.e. synthesis) because type is the *only* atomic kind.

$$\frac{\Gamma \vdash A \Leftarrow \mathsf{type} \quad \Gamma, x : A \vdash B \Leftarrow \mathsf{type}}{\Gamma \vdash \Pi x{:}A.B \Leftarrow \mathsf{type}} \text{ CHECK-TYPE}$$

$$\frac{a : K \in \Sigma \quad \Gamma \vdash S : K > \mathsf{type}}{\Gamma \vdash a \cdot S \Leftarrow \mathsf{type}} \text{ FOC-TCON}$$

$$\frac{\Gamma \vdash M \Leftarrow A \quad \Gamma \vdash S : [M/x]_{(A)^-} K > \mathsf{type}}{\Gamma \vdash (M; S) : \Pi x{:}A.K > \mathsf{type}} \text{ SYN-TYPE}$$

$$\frac{}{\Gamma \vdash \mathsf{nil} : \mathsf{type} > \mathsf{type}} \text{ INIT-TYPE}$$

1.3.3 Kind validity

No focusing phase is needed here, as there is only one hyperkind kind, so that the end of the left inversion (i.e. checking) phase is initial.

$$\frac{\Gamma \vdash A \Leftarrow \mathsf{type} \quad \Gamma, x{:}A \vdash K \Leftarrow \mathsf{kind}}{\Gamma \vdash \Pi x{:}A.K \Leftarrow \mathsf{kind}}\ \text{CHECK-KIND}$$

$$\frac{}{\Gamma \vdash \mathsf{type} \Leftarrow \mathsf{kind}}\ \text{INIT-KIND}$$

1.3.4 Signature and context validity

This whole series of judgments has been defined in terms of valid signatures and contexts. The rules for valid contexts are straightforward.

$$\frac{}{\vdash \cdot\ \mathsf{ctx}}\ \text{CTX-EMPTY} \qquad \frac{\vdash \Gamma\ \mathsf{ctx} \quad \Gamma \vdash A \Leftarrow \mathsf{type}}{\vdash \Gamma, x : A\ \mathsf{ctx}}\ \text{CTX-TERM}$$

Furthermore, every judgment above has been implicitly parametrized by a signature Σ, which must be a valid signature as defined by the following judgments (which are mutually recursive with all the others):

$$\frac{}{\vdash \cdot\ \mathsf{sig}}\ \text{SIG-EMPTY} \qquad \frac{\vdash \Sigma\ \mathsf{sig} \quad \cdot \vdash_\Sigma A \Leftarrow \mathsf{type}}{\vdash \Sigma, c : A\ \mathsf{sig}}\ \text{SIG-TERM}$$

$$\frac{\vdash \Sigma\ \mathsf{sig} \quad \cdot \vdash_\Sigma K \Leftarrow \mathsf{kind}}{\vdash \Sigma, a : K\ \mathsf{sig}}\ \text{SIG-FAM}$$

1.3.5 A note on modes and equality

As we explained before, the left focusing judgment is represented here as $\Gamma \vdash S : V > P$ where V represents an arbitrary classifier (a type or a kind) and P represents an arbitrary atomic classifier. This is in line with other presentations, but the notation most consistent with a focusing system would seem to be Ley-Wild's formulation, $\Gamma; S : V \ll P$.

The difference between these two notations can be understood as two interpretations of where an equality check happens in a focusing system. Focusing systems often consider both the V and P in $\Gamma; V \ll P$ to be inputs and enforce an equality check at the left initial sequent, whereas the notation $\Gamma \vdash S : V > P$ interprets P as an output; this is why this output is checked for equality at the judgment that began the left focusing phase.

$$
\begin{array}{rrcl}
\text{Levels} & L & ::= & \mathsf{kind} \mid \mathsf{type} \\
\text{Expressions} & U, V & ::= & L \mid \Pi^d(V_1, V_2) \mid \lambda([V], U) \mid H \cdot S \\
 & & & \mid U@S \quad \text{(redex)} \\
 & & & \mid U[\sigma] \quad \text{(closure)} \\
 & & & \\
\text{Heads} & H & ::= & c \mid i \ (i \geq 1) \\
\text{Spines} & S & ::= & \mathsf{nil} \mid U; S \mid S[\sigma] \\
 & & & \\
\text{Substitution} & \sigma & ::= & U.\sigma \mid i.\sigma \mid \uparrow^k \ (k \geq 0) \\
\text{Contexts} & \Gamma & ::= & \cdot \mid \Gamma, A \\
\text{Signatures} & \Sigma & ::= & \cdot \mid \Sigma, c : V
\end{array}
$$

Figure 5: Term representation LF - this presentation is incomplete and will be extended with metavariables in the next set of notes.

Another point worth mentioning is that the equality judgments in LF generally happen without the context Γ, although it would be reasonable, for instance, to write $(\Gamma \vdash P = P' : \text{type})$ instead of $(P = P')$ in the rule FOC-CON. In fact, the comments in the Twelf implementation often use the former notation, even though if Γ is used only for debugging and pretty-printing in equality checks within Twelf. Doing so primarily has documentation value, as checking terms at different types (and equivalently for types and kinds) can cause Twelf to throw match exceptions.

1.4 Term representation language

Having presented the rules of spine form LF, we will now begin to describe a variant of spine form LF that is much closer to an efficient implementation. The syntax of this language, which we will call *term representation LF*, is (partially) described in Figure 5. This section describes the major differences between spine form LF and term representation LF.

1.4.1 de Bruijn indices

When several different implementations were done at the beginning, a nominal approach (requiring capture-avoiding substitution) was more opaque and error-prone, though there was some question at the meeting whether that might be alleviated by using a "wizard" pattern [8]. However, in the

implementation de Bruijn indices (specifically de Bruijn indices with explicit substitution, see Section 1.4.5) are used instead. This means that variable labels are omitted in contexts, λ-bindings, and dependent Π-bindings. These indices start at one, not zero.[1]

1.4.2 Fewer syntactic categories

Observing the similarity between judgments like CHECK-TYPE and CHECK-KIND, we collapse types and terms into a single syntactic category of *expressions*, and we combine the base kind type and the base hyperkind kind into the syntactic category of *levels*, denoted by L. This simplifies signatures as well: they now only have entries of the form $c : V$ where V is a type or a kind. When there is a thing being classified being related to its classifier, the convention is to use U for the classified thing (a type, term, or kind) and V for the classifier (a type, kind, or the token kind, the only hyperkind). For instance, the rules CHECK-TYPE and CHECK-KIND can be rolled into a single function.

$$\frac{\Gamma \vdash V_1 \Leftarrow \mathsf{type} \quad \Gamma, V_1 \vdash V_2 \Leftarrow L}{\Gamma \vdash \Pi(V_1, V_2) \Leftarrow L} \text{ CHECK-PI}$$

When type checking has reached an atomic term or type, the head H is either a de Bruijn index i that can be looked up in the context Γ or a constant c that can be looked up in the signature Σ, and the judgment $\Gamma \vdash H \Rightarrow V$ represents looking up $H : V$ in either the signature or the context. The constants c are implemented as integers called *cids* in the Twelf implementation. The symbols P and P' in the rule below represent an atomic classifier, either $c \cdot S$ or L.

$$\frac{\Gamma \vdash H \Rightarrow V \quad \Gamma \vdash S : V > P' \quad P = P'}{\Gamma \vdash H \cdot S \Leftarrow P} \text{ FOCUS}$$

1.4.3 Dependency flags

The pi-binding $\Pi^d V_1, V_2$ comes with a dependency flag d which may be no or maybe, where a no indicates that the classifier V_2 definitely does not depend on V_1. To give an example in spine form LF, in the type

$$\Pi^{\mathsf{no}} x{:}\mathsf{nat}.\mathsf{nat}$$

[1] Sorry about that.

the type nat is independent of x, whereas in the type

$$\Pi^{\mathsf{maybe}} x{:}\mathsf{nat}.\mathsf{plus} \cdot (\mathsf{z}; x; x; \mathsf{nil})$$

the type $\mathsf{plus} \cdot (\mathsf{z}; x; x; \mathsf{nil})$ is dependent on x.

This flag is important for avoiding spurious dependencies and has implication for the operational semantics of logic programming in Twelf. The flag also has different meanings before and after type reconstruction. Twelf initially does a first order approximation of the user's input in order to reconstruct that input in Twelf's internal syntax. After that approximation, the no flag means that the user wrote an arrow (i.e. `A -> B`), so that there is no dependency, and the maybe flag means that the user used brackets (i.e. `{x:A} B`), so that there may be a dependency. After reconstruction, the maybe flags that are unnecessary will be changed to no flags, which will happen, for instance, if the user wrote `s : {x:nat} nat.`

Even after the full type reconstruction, the maybe flag does not guarantee the presence of a dependency if there are metavariables, as unification during proof search may cause dependencies to be eliminated.

1.4.4 Optional type annotations

Type annotations are notably present (though optional) for lambda-bindings; these are not present in other presentations of canonical forms LF. During type reconstruction type checking is not always bidirectional and the information is sometimes needed. These annotations generate overhead when they are present; an extra equality check is required when checking the type of a lambda that has an annotation. It is not clear that this was the best choice, and it is not clear that it will be the right choice in other systems such as CLF.

$$\frac{V_1 = V_2 \qquad \Gamma, V_1 \vdash U \Leftarrow V}{\Gamma \vdash \lambda(V_1, U) \Leftarrow \Pi(V_2, V)} \text{ CHECK-ANNOT-LAM}$$

1.4.5 Lazy canonical forms

During the early implementations, it was not just de Bruijn indices but de Bruijn indices with explicit substitutions that were found to lead to the most efficient and natural formulations.[2] It allows for a lazy approach where

[2] Bob Harper noted that this approach still seemed quite natural even after spending a long time away from it.

terms are normalized towards canonical (spine) form only as far as they needed to be in order to perform a given operation.

This lazy implementation requires two non-canonical syntactic additions, which correspond to the two parts of hereditary substitution - a redex $U@S$ representing partial reduction and a closure $U[\sigma]$ representing partial substitution. Type checking is never performed on these syntactic objects, they are pushed along to normalize terms only as far as is needed. Incidentally, testing for equality (as is done in the FOCUS rule) will force the two terms to be fully normalized, as non-canonical forms are never checked for equality, and generally have no first-class status within the implementation.

The primary means of performing these reductions is called *weak head normalization*, which is just a process of driving through reductions and substitutions until the top layer of a term looks like a canonical form (i.e. $L, H \cdot S$, $\lambda([V], U)$, or $\Pi(V, V)$). There are many functions in the Twelf implementation that have two forms, a `check()` that can be called on arbitrary terms and an optimized `checkW()` that can be called if the term is known to be in weak head normal form - the functions that can be called on arbitrary terms just reduce the term to weak head normal form and then call the optimized version of the function.

Aleksey Kliger asked about whether these applied reconstructions were saved, as they were in his LSL system. The Twelf implementation does not save these, but the reason they seem to be efficient in LSL is that there is no backtracking - backtracking is one of the major reasons that memoizing reconstruction isn't worth the cost in Twelf or other systems such as Lambda Prolog where it was tested.

1.5 References

[1] M. Abadi, L. Cardelli, P.-L. Curien, and J.-J. Levy. Explicit substitutions. In *POPL '90: Proceedings of the 17th ACM SIGPLAN-SIGACT symposium on Principles of programming languages*, pages 31–46, New York, NY, USA, 1990. ACM Press.

[2] Iliano Cervesato and Frank Pfenning. A Linear Spine Calculus. *Journal of Logic and Computation*, 13(5):639–688, 2003.

[3] 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.

[4] Robert Harper and Daniel R. Licata. Mechanizing metatheory in

a logical framework. Submitted for publication. Available from `http://www.cs.cmu.edu/~drl/`, October 2006.

[5] Aleksander Nanevski, Frank Pfenning, and Brigette Pientka. Contextual Modal Type Theory, 2005. CHECK.

[6] Brigitte Pientka. *Tabled Higher-Order Logic Programming*. PhD thesis, Carnegie Mellon University, December 2003.

[7] Jason C. Reed. Proof Irrelevance with Hereditary Substitution, May 2006. Unpublished notes.

[8] Tom Murphy VII. The Wizard of TILT: Efficient, convenient and abstract type representations. Technical Report CMU-CS-02-120, School of Computer Science, Carnegie Mellon University, March 2002.

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