Base-Type Polymorphism in LF

Jason Reed

February 26, 2008

1 Introduction

The type theory of LF [HHP93] is particularly useful for representation of other logical systems because it enjoys such a robust notion of *canonical forms* of terms. With only a very weak, intensional function type, the canonical forms of functions are typically isomorphic to terms with bound variables, making possible powerful representational techniques such as higher-order abstract syntax [PE89]. Lacking polymorphism, one immediately knows in LF how to fully η-expand and β-normalize a term. As Watkins et al. showed in the development of CLF [WCPW03], these canonical forms admit a substitution principle embodied by the operation of hereditary substitution, allowing the entirety of the metatheory of LF to be carried out on canonical forms alone.

Now on the other hand, Nanevski, Morrisett, and Birkedal [NMB06] devised a way to add polymorphism without sacrificing the benefits of canonical forms entirely. The price to be paid, however, is a mixing of the term and type levels, and a mechanism for 'on-the-fly' η-expansion. Their technique is to add a wrapper eta_α around any term whose type is the type variable α, so that when a substitution later replaces α with (for example) a function type, $\mathsf{eta}_\alpha(M)$ can transform itself into the appropriate eta-expansion of M.

The purpose of this note is to make the observation that we can get a surprising amount of mileage out of examining the subset of polymorphic extension to LF where precisely this 'on-the-fly' η-expansion is not required: that is, allowing quantification not over arbitrary types, but over arbitrary *base* types, for which no η-expansion is required. Call this language LFB.

One advantage of this arrangement is an immediate simplification of the definition of LF itself, in that the signature no longer needs to be fundamentally distinguished from the context, for assumptions of existence of types and type families may now occur in the context.

2 Language

The syntax of the language LFB is as follows.

$$\begin{array}{rlll} \text{Classifiers} & V & ::= & \Pi x{:}V_1.V_2 \mid v \\ \text{Base Classifiers} & v & ::= & \text{type} \mid R \\ \text{Normal Terms} & M & ::= & \lambda x.M \mid R \\ \text{Atomic Terms} & R & ::= & x \cdot S \\ \text{Spines} & S & ::= & () \mid (M; S) \end{array}$$

The complete type system is just these four judgments:

Classifier Well-Formedness ($\Gamma \vdash V \Leftarrow \text{class}$)

$$\frac{\Gamma \vdash V_1 \Leftarrow \text{class} \qquad \Gamma, x : V_1 \vdash V_2 \Leftarrow \text{class}}{\Gamma \vdash \Pi x{:}V_1.V_2 \Leftarrow \text{class}}$$

$$\frac{}{\Gamma \vdash \text{type} \Leftarrow \text{class}} \qquad \frac{\Gamma \vdash R \Leftarrow \text{type}}{\Gamma \vdash R \Leftarrow \text{class}}$$

Checking ($\Gamma \vdash M \Leftarrow V$)

$$\frac{\Gamma, x : V_1 \vdash M \Leftarrow V_2}{\Gamma \vdash \lambda x.M \Leftarrow \Pi x{:}V_1.V_2} \qquad \frac{\Gamma \vdash R \Rightarrow v' \qquad v = v'}{\Gamma \vdash R \Leftarrow v}$$

Synthesis ($\Gamma \vdash R \Rightarrow v$)

$$\frac{x : V \in \Gamma \qquad \Gamma \vdash S : V > v}{\Gamma \vdash x \cdot S \Rightarrow v}$$

Spine Checking ($\Gamma \vdash S : V > v$)

$$\frac{}{\Gamma \vdash () : v > v} \qquad \frac{\Gamma \vdash M \Leftarrow V_1 \qquad \Gamma \vdash S : [M_1/x]V_2 > v}{\Gamma \vdash (M; S) \Rightarrow \Pi x{:}V_1.V_2 > v}$$

We assume that all terms are *simply well-typed*, which means that they are well-typed according to the above rules with all v in classifier positions replaced by $\bullet$ and all Π replaced by $\rightarrow$. This means that hereditary substitution, defined as follows, is total and manifestly terminating:

Abbreviate $\sigma = [M/x]$, assume $x \neq y$.

$$\begin{array}{rcl} \sigma(x \cdot S) & = & [M \mid \sigma S] \\ \sigma(y \cdot S) & = & y \cdot (\sigma S) \\ \sigma(\lambda y.N) & = & \lambda y.(\sigma N) \\ \sigma(\Pi y{:}V_1.V_2) & = & \Pi y{:}(\sigma V_1).(\sigma V_2) \\ \sigma(\text{type}) & = & \text{type} \\ \sigma() & = & () \\ \sigma(N; S) & = & (\sigma N; \sigma S) \\ [\lambda x.N \mid (M; S)] & = & [[M/x]N \mid S] \\ [R \mid ()] & = & R \end{array}$$

This system satisfies the evident substitution and identity principles, proven by standard techniques.

Lemma 2.1 *If* $\Gamma \vdash M \Leftarrow V$ *and* $\Gamma, x : V, \Gamma' \vdash J$, *then* $\Gamma, [M/x]\Gamma' \vdash [M/x]J$.

Lemma 2.2 *If* $\Gamma \vdash V \Leftarrow \text{class}$, *then there exists a unique term* $\eta_V(x)$ *such that*

- $\Gamma, x : V \vdash \eta_V(x) \Leftarrow V$
- $[\eta_V(x)/x]X = X$
- $[M/x]\eta_V(x) = M$

Remark The lemma above does not allow substitution of $\Pi x{:}V_1.V_2$ for a variable! Such a thing is already ruled out syntactically. An M can be an R which can satisfy $R \Rightarrow$ type (so base types can be substituted for type variables) but higher types cannot.

3 Embedding LFB into LF

Although this system seems easily marketed as an extension to LF, from another perspective it is just a cheap subset of (or to put it another way, an encoding style in) LF. The translation is as follows: Let Σ be the LF signature containing two declarations t : type and $o : t \rightarrow$ type. Any LFB classifier V is mapped to V^* according to the recursive definition

$$\begin{array}{rcl} (\Pi x{:}V_1.V_2)^* & = & \Pi x{:}(V_1^*).(V_2^*) \\ \text{type}^* & = & t \\ R^* & = & o\ R \end{array}$$

Then

Lemma 3.1 $\Sigma; \cdot \vdash_{LF} M \Leftarrow V^*$ *iff* $\vdash_{LFB} M \Leftarrow V$.

Proof By induction on the structure of the derivations. ■

4 Applications

The stereotypical thing that the workaday LF hacker wants polymorphism for is lists. This is easily and directly encodable. In pseudo-Twelf syntax,

```
list : type -> type.
nil : list T.
cons : T -> list T -> list T.
```

Then we can go on to write polymorphic logic programs over this type. For instance, `snoc`, which appends an item to the end of a list.

```
snoc : T -> list T -> list T -> type.
snoc/nil : snoc X nil (cons X nil).
snoc/cons : snoc X (cons H TL) (cons H TL')
             <- snoc X TL TL'.
```

Under the above translation, this code becomes

```
t : type.
o : t -> type.
list : t -> t.
nil : o (list T).
cons : o T -> o (list T) -> o (list T).
snoc : o T -> o (list T) -> o (list T) -> t.
snoc/nil : o (snoc X nil (cons X nil)).
snoc/cons : o (snoc X (cons H TL) (cons H TL'))
             <- o (snoc X TL TL').
```

And we can actually then run it in Twelf.

```
bool : t.
true : o bool.
false : o bool.
%query 1 1
o (snoc true (cons true (cons false (cons false nil))) X).
---------- Solution 1 ----------
X = cons true (cons false (cons false (cons true nil))).
```

What's more, being able to abstract over type operators allows for higher-order functions in logic programming. We can write map as follows:

```
map : (A -> B -> type) -> list A -> list B -> type.
map/nil : map F nil nil.
map/cons : map F (cons H TL) (cons H' TL')
    <- F H H'
    <- map F TL TL'.
```

Which again, after suitable translation, is runnable in Twelf.

```
map : (o A -> o B -> t) -> o (list A) -> o (list B) -> t.
map/nil : o (map F nil nil).
map/cons : o (map F (cons H TL) (cons H' TL'))
    <- o (F H H')
    <- o (map F TL TL').

not : o bool -> o bool -> t.
not/t : o (not true false).
not/f : o (not false true).

%query 1 1
o (map not (cons true (cons false (cons false nil))) X).
---------- Solution 1 ----------
X = cons false (cons true (cons true nil)).
```

5 Challenges

Clearly, doing mode-, coverage-, and termination-checking directly on this language — or even basic algorithms such as subordination — are difficult open problems. However, these difficulties are already implicit in doing the same for LF extended with a module language, for we'd hope to be able to *assume* such properties in module signatures.

It appears *prima facie* that we are limited to building polymorphic data structures of only first-order things, since we are unable to instantiate type variables with higher types. However in some cases one can get around this by extracting the binding outside the type operator: instead of $list\ (tm \to tm)$ one can get by with $tm \to (list\ tm)$.

Some of the idioms of higher-order logic do not work as well as might be hoped. Standard polymorphic definitions of propositional connectives like

$$A \vee B \equiv_{def} \Pi\alpha{:}\mathrm{type}.(A \to B \to \alpha) \to \alpha$$

'infect' all types with eliminations — just as would happen with a naïve extension of LF with sums. What might be expected for a definition of Leibniz equality

$$Eq_A(M, N) \equiv_{def} \Pi P{:}A \to \mathrm{type}.P\ M \to P\ N$$

fails to enjoy symmetry, for the standard proof of it involves a higher-order (in the sense that it involves an implication) predicate substituted for P. I would conjecture that *no* single such definition could suitably capture equality as would be defined in HOL.

References

[HHP93] 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.

[NMB06] Aleksandar Nanevski, Greg Morrisett, and Lars Birkedal. Polymorphism and separation in hoare type theory. In *ICFP '06: Proceedings of the eleventh ACM SIGPLAN international conference on Functional programming*, pages 62–73, New York, NY, USA, 2006. ACM.

[PE89] Frank Pfenning and Conal Elliott. Higher-order abstract syntax. In *Proceedings of the ACM SIGPLAN '88 Symposium on Language Design and Implementation*, pages 199–208, Atlanta, Georgia, June 1989.

[WCPW03] 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, 2003.