LF in LF: Mechanizing the Metatheory of LF in Twelf

Chris Martens Karl Crary

Carnegie Mellon University

{cmartens, crary}@cs.cmu.edu

Abstract

We present a mechanized proof of the metatheory of LF, i.e. the decidability of typechecking and the existence and uniqueness of canonical forms. We use a syntactic approach in which we define a translation from LF to its canonical forms presentation (in which only beta-short, eta-long terms are well-formed) and prove soundness and completeness of the translation, establishing that definitional equivalence in LF corresponds to syntactic equivalence in canonical forms. Much recent work is based on the system of canonical forms and hereditary substitution presented herein; our proof also serves to reconcile that presentation with the traditional version based on definitional equivalence.

Categories and Subject Descriptors F.3.3 [*Studies of Program Constructs*]: type structure; F.4.1 [*Mathematical logic*]: lambda calculus and related systems

General Terms languages, theory

Keywords logical frameworks, dependent type theory, hereditary substitution, mechanized metatheory, Twelf

1. Introduction

The logical framework LF [9] and its implementation Twelf [14] comprise a framework for defining and reasoning about deductive systems. Developments using this technology include a the mechanized definition of Standard ML [10], solutions to the POPLmark challenge [1], the metatheory of a framework for distributed mobile code [12], and a proof-carrying filesystem [6].

LF is a dependently-typed lambda calculus. When taking its type theory as the object of study, there are two extant classes of presentation: one in which terms may be written in the standard way and redexes in types are handled through *definitional equivalence*; the other in which terms must be written in *canonical* (roughly, normal) form, and substitution must be redefined to preserve that form. The former can be seen as the *programmer's interface* to LF—it is the original presentation, and it corresponds to a more standard notion of programming. All of the complexity lies in the metatheory of deciding the type system. On the other hand, the *canonical* presentation provides a convenient basis for studying and extending the LF type theory itself. Its metatheory is relatively straightforward and amenable to structural proof techniques.

[Copyright notice will appear here once 'preprint' option is removed.]

```
tm : type.
lam : (tm -> tm) -> tm.
app : tm -> tm -> tm.

eval : tm -> tm -> type.
eval/beta : eval (app (lam E) E') V
            <- eval (E E') V.
% ...
```

Figure 1. Untyped lambda calculus in Twelf

To illustrate more concretely the distinction between these two LF systems, consider the standard LF encoding of the untyped lambda calculus in Figure 1. We refer to LF as the *meta-level* and to the untyped lambda calculus (in general, the language being encoded) as the *object-level*. We give the syntax as well as one evaluation rule for the object-level. The following discussion refers to notions of reduction and expansion at the meta-level.

According to the original formulation of LF, the included evaluation rule may be written in several different ways. Consider the term E. Because it has (meta-level) type `tm -> tm`, it can be eta-expanded to λx:`tm`.E x. We could replace each occurrence of E in the rule with its η-expansion. Notably, if we did so in the second occurrence, we would create a β-redex.

The LF terms in this example are said to be in *canonical form* iff we η-expand in the first occurrence, where it is an argument (`lam` (λx.E x)) and do not in the second occurrence, where it is applied as a function (`E E'`). More generally, canonical forms are fully eta-expanded except where they would introduce beta-redexes.

This definition prescribes a *unique* way to write a term given its type, which comes up when considering *adequacy*. For an encoding to be *adequate* means that elements of the object language, such as terms or derivations, are in a compositional bijection with LF terms of the corresponding type. However, the space of *all LF terms* is too large; only by restricting to *canonical* LF terms of the appropriate type does the theorem hold.

Additionally, canonical forms provide a basis for decidability of type checking. Type checking reduces to type equivalence checking, and one way to tell if two terms are equivalent is to put them both in canonical form and compare (because canonical forms are unique). We should, e.g., be able to deduce decidably that `eval E E'` and `eval` (λx.E x) `E'` are equivalent types.

The notion of canonical form can be seen as an extrinsic property of an LF term, defined separately as a technical device. Up to a certain point in LF's history, that was exactly its treatment, e.g. in earlier proofs of decidability and existence of canonical forms, such as Harper and Pfenning's logical relations argument [8]. While this approach is elegant and has even been mechanized [20], the authors were interested in a syntactic approach amenable to formalization in Twelf. For that we needed theory developed later: the distinct type system *Canonical LF*.

Felty [5] first observed that you could restrict LF to β-normal terms from the get-go by changing the grammar and formation rules. However, non-canonical forms were still needed because canonicity is not preserved by substitution; in Felty's work, the notion of equality depends on β-reduction over non-canonical forms.

Watkins [21] later had the insight that one could work entirely within canonical forms if we *redefine substitution* to normalize any redexes it creates. It is this notion of *hereditary substitution*, similar to a sequent calculus notion of *cut*, that is central to this work's version of Canonical LF and to the simplicity and significance of our proof.

The canonical system gives rise to a straightforwardly decidable bidirectional type checking algorithm. Function applications, variables, and constants can *synthesize* their types while functions are *checked* against a provided type. No equivalence checking is necessary since terms are unique—or put another way, types are compared purely syntactically (i.e. by unification in the metalogic). Instead the subtle reasoning manifests in the metatheory of substitution, which is nonetheless amenable to a structural proof adapted from proofs of cut admissibility in sequent calculus.

Our theorem therefore is that type checking in the original LF can be faithfully reduced to type checking in Canonical LF. This boils down to defining an algorithm to translate the syntactic objects of LF to syntactic objects of canonical LF. We prove the algorithm total, sound, and complete. Because we are translating one system to another, we use the compilers terminology *EL* (external language) and *IL* (internal language) for original and canonical LF, respectively.

In the sense that this is a proof about the EL, soundness and completeness say that any judgments we wish to decide in the EL may first have their pieces translated and the corresponding judgment decided in the IL. Alternatively, the proof can also be viewed as a validation of Canonical LF. A considerable body of research on LF has been based on the canonical formulation [11, 15, 19, 21]; this proof confirms that they are talking about the same thing, and moreover that definitional equivalence gives a semantics to hereditary substitution.

The entire proof is formalized in Twelf. We borrow heavily from the methodology used in Crary's mechanization of the singleton calculus [3] in this development. Source code may be found at `www.cs.cmu.edu/~cmartens/lfinlf`, and this paper will make use of notation and judgments that should be recognizable in the source should the reader care to look.

2. LF

In this section we describe our formulation of LF, the EL. [1]

The top-level object of an encoding is a *signature*, for which we use the metavariable Σ. Every extension of the signature introduces a new type constant or term constant.

$$\Sigma ::= \cdot \mid \Sigma, c : \bar{A} \mid \Sigma, a : \bar{K}$$

(We will write every EL metavariable with a bar to distinguish it from IL metavariables, which come later.)

Constructors are:

$$\bar{A} ::= a \mid (\bar{A}\ \bar{M}) \mid \lambda x{:}\bar{A}.\,\bar{A} \mid \Pi x{:}\bar{A}.\,\bar{A}$$

They can be constants, applications of a constructor to a term, lambda expressions, or (dependent) function types. The variables x range over terms (there is no type polymorphism). We will use $\bar{A} \to \bar{B}$ as shorthand for $\Pi\_{:}\bar{A}.\bar{B}$ when there is no dependency in $\bar{B}$. We follow the original presentation by including family-level lambda. Several subsequent formulations omit it based on the folkloric observation that the system is equivalent without it, but it serves to be a convenient technical device in our proof, so we include it.

[1] The interested reader may follow along with the development by referring to `noncan.elf`. Where applicable, the inference rules herein are named after their corresponding signature declarations there.

$\bar{M} : \bar{A}$	$\bar{M}$ has type $\bar{A}$
$\bar{A} : \bar{K}$	$\bar{A}$ has kind $\bar{K}$
$\bar{K}$: kind	$\bar{K}$ is a kind
$\bar{M} \equiv \bar{N} : \bar{A}$	$\bar{M}$ and $\bar{N}$ are equivalent at type $\bar{A}$
$\bar{A} \equiv \bar{B} : \bar{K}$	$\bar{A}$ and $\bar{B}$ are equivalent at kind $\bar{K}$
$\bar{K} \equiv \bar{L}$	$\bar{K}$ and $\bar{L}$ are equivalent

Figure 2. Judgments of LF.

The (standard) grammar for terms and kinds follows.

$$\bar{M} ::= c \mid x \mid (\bar{M}\ \bar{M}) \mid \lambda x{:}\bar{A}.\bar{M}$$

$$\bar{K} ::= \mathsf{type} \mid \Pi x{:}\bar{A}.\,\bar{K}$$

The judgments of LF are summarized in Figure 2. For each of these judgments J, there is a hypothetical version $\Gamma \vdash_\Sigma J$ parameterized over a term context Γ and a signature Σ; in the typing rules, we stick to a notation closer to the Twelf development in which the context and the signature are implicit, residing in the ambient metalogic.

We will now inspect a few of the key typing and equivalence rules; the entirety of the type system can be found in the appendix.

Equivalence is introduced to the type system via the conversion rule (one each for terms and constructors):

$$\frac{\bar{M} : \bar{A} \quad \bar{A} \equiv \bar{B} : \mathsf{type}}{\bar{M} : \bar{B}}\ \texttt{eof/eqtp} \qquad \frac{\bar{A} : \bar{K} \quad \bar{K} \equiv \bar{K}'}{\bar{A} : \bar{K}'}\ \texttt{ekof/eqkind}$$

The premise $A \equiv \bar{B}$: type says that $\bar{A}$ and $\bar{B}$ are equivalent types, i.e. equivalent constructors *at kind* type. In general, we say that families are equivalent at kind K (and prove as an invariant that they each in fact inhabit kind K). Constructor equivalence ultimately appeals to term equivalence, which is where we will restrict our attention for this discussion.

Our notion of equivalence permits β and η conversion via the following rules:

$$\frac{\begin{array}{c}\bar{M} : \Pi x{:}\bar{A}.\bar{B}' \quad \bar{M} : \Pi x{:}\bar{A}.\bar{B}'' \\ x : \bar{A} \vdash (\bar{M}\ x) \equiv (\bar{N}\ x) : \bar{B}\end{array}}{\bar{M} \equiv \bar{N} : \Pi x{:}\bar{A}.\bar{B}}\ \texttt{eqtm/ext}$$

$$\frac{x : \bar{A} \vdash \bar{M} : \bar{B} \quad \bar{N} : \bar{A}}{(\lambda x{:}\bar{A}.\bar{M})\ \bar{N} \equiv [\bar{N}/x]\bar{M} : [\bar{N}/x]\bar{B}}\ \texttt{eqtm/beta}$$

The remainder of the rules simply close the relation under equivalence and compatibility.

The notion of substitution for the EL is the standard, uniform, capture-avoiding substitution that we see no need to include (and get for free in Twelf).

3. Canonical LF

In a *canonical presentation* of a proof system, we syntactically separate terms that are *canonical* (hereafter "terms") and terms that are *atomic* (hereafter "atoms"), ruling out beta redexes. eta-length is ensured by the typing rules, which are divided into two judgments, type *checking* and type *synthesis*, comprising a bidirectional, syntax-directed algorithm. Harper and Licata [7] provide

$$\frac{c : A \in \Sigma \quad A \Leftarrow \mathsf{type}}{c \Rightarrow A}\ \texttt{at-of/const}$$

$$\frac{x : A \in \Gamma \quad A \Leftarrow \mathsf{type}}{x \Rightarrow A}\ \texttt{at-of/var}$$

$$\frac{R \Rightarrow \mathsf{base}(P)}{\mathsf{at}(R) \Leftarrow \mathsf{base}(P)}\ \texttt{of/at} \qquad \frac{x{:}A \vdash M \Leftarrow B \quad A \Leftarrow \mathsf{type}}{\lambda x.M \Leftarrow \Pi x{:}A.B}\ \texttt{of/lam}$$

$$\frac{R \Rightarrow (\Pi x{:}A.B) \quad M \Leftarrow A \quad [M/x]B \ \mathsf{is}\ B'}{(R\ M) \Rightarrow B'}\ \texttt{at-of/app}$$

Figure 3. Type synthesis for atoms and checking for terms.

excellent exposition on this formulation of LF, which we attempt to recapitulate here. Such a system is known by a few other names, such as *intercalation* [2] and *verifications and uses* [13].

The canonical presentation of LF will be the IL for our proof. Because the only well-formed, well-typed terms are the beta-short and eta-long, there is no need to consider types up to a notion of equivalence. The heavy lifting gets passed off to *substitution*, which must be redefined to preserve canonicity. The grammar of Canonical LF is as follows: [2]

$$R ::= c \mid x \mid (R\ M)$$
$$M ::= \mathsf{at}(R) \mid \lambda x.\, M$$

By inspection, we see that no beta-redexes are admitted: the head of an application may never be a λ.

We have two distinct typing judgments for these syntactic classes:

$M \Leftarrow A$	M checks at type A
$R \Rightarrow A$	R synthesizes type A

Together, they constitute a *bidirectional* type checking algorithm. The judgment $M \Leftarrow A$ can be read operationally such that given a term and a type, it succeeds when they check—in logic programming terms, it has input-input mode. The judgment $R \Rightarrow A$ can be read with input-output mode: given an atom, it produces the type. Intuitively, atoms have the property that their types can be "read off" from the signature and constraints of the context, propagated top-down. Terms, on the other hand, must provide their type, from which information flows *upward* in the derivation tree. The key rule is where they meet in the middle, at the typing rule for $\mathsf{at}(R)$, the inclusion of atoms into terms. This rule precludes partial application (enforces eta-length) by requiring the atom to have a base type:

$$\frac{R \Rightarrow \mathsf{base}(P)}{\mathsf{at}(R) \Leftarrow \mathsf{base}(P)}\ \texttt{of/at}$$

Typing of terms and atoms is summarized in Figure 3. Note that the typing rule for application depends on substitution, which we write as a relation rather than a metalevel function to suggest that we need to define it—it is not the standard, uniform substitution granted by the metalogic.

In order to say what counts as a base type for the `of/at` rule, we need a similar stratification of atomic and canonical type families and corresponding kinding judgments. (Kinds are the same as previously.) "$\mathsf{base}(P)$" signifies base types, which are "atomic" type families.

$$P ::= a \mid (P\ M)$$

[2] The interested reader may follow along with `canonical.elf` for this section.

$$\frac{a : K \in \Sigma \quad K : \mathsf{kind}}{a : K}\ \texttt{at-kof/const}$$

$$\frac{A \Leftarrow \mathsf{type} \quad x{:}A \vdash B \Leftarrow \mathsf{type}}{\Pi x{:}A.B \Leftarrow \mathsf{type}}\ \texttt{kof/pi}$$

$$\frac{P \Rightarrow \Pi x{:}B.K \quad M \Leftarrow B \quad [M/x]K \ \mathsf{is}\ K'}{(P\ M) \Rightarrow K'}\ \texttt{at-kof/app}$$

$$\frac{A \Leftarrow \mathsf{type} \quad x{:}A \vdash B \Leftarrow K}{\lambda x{:}A.B \Leftarrow \Pi x{:}A.K}\ \texttt{kof/lam} \qquad \frac{P \Rightarrow \mathsf{type}}{\mathsf{base}(P) \Leftarrow \mathsf{type}}\ \texttt{kof/base}$$

Figure 4. Kinding for type families.

$$\frac{}{\mathsf{type} : \mathsf{kind}}\ \texttt{wfkind/tp} \qquad \frac{A \Leftarrow \mathsf{type} \quad x{:}A \vdash K : \mathsf{kind}}{\Pi x{:}A.K : \mathsf{kind}}\ \texttt{wfkind/pi}$$

Figure 5. Well-formedness of kinds.

$$A ::= \mathsf{base}(P) \mid \Pi x{:}A.B \mid \lambda x.\, A$$

The kinding and well-formedness rules for type families and kinds are listed in figures 4 and 5.

3.1 Substitution

Ordinary substitution, when applied to a canonical term, can create β-redexes; for an example, consider

$$[(\lambda y.N)/x](x\ M) \longrightarrow_R (\lambda y.N)\ M$$

Therefore we use *hereditary substitution* [21]. The key rule defines how to substitute into an application. Roughly speaking, to substitute N for x in $(R\ M)$,

- Let $M' = [N/x]M$.
- Compute $[N/x]R$, resulting in F, which may be either an atom or a term.
- If F is an atom, return $(F\ M)$.
- If F is a term, it must have the form $\lambda x.O$ (because it is a function). In that case, *hereditarily* compute $[M'/x]O$.

Whether $[N/x]R$ results in a term or an atom depends on whether x is the *head variable* of R—that is, whether $R = xN_1 \ldots N_n$ for $n \geq 0$.

Thus we formally define substitution (at the term level) as three judgments:

- $[N/x]M$ is M' for substituting into a term, resulting in a term
- $[N/x]R$ is R' for substituting into an atom, resulting in an atom (when x is not the head of R)
- $[N/x]R$ is M for substituting into an atom, resulting in a term (when x is the head of R)

The rules defining these judgments are summarized in Figure 6. The rules for constructors and kinds are similar, but without the `rm` cases for constructors, as there are no type variables.

Most of hereditary substitution just crawls the structure of the term to find variables to replace, as in uniform substitution. The hereditary case, `rmsub/app`, is where all the action takes place, as we previously described.

The only other rule of particular interest is the `sub/rm` rule, in which the substitution only occurs when the resulting term of the `rmsub` is an atom. This restriction adds a smidgen of sorting information to the rule needed to carry through the termination proof.

$$\frac{[N/x]R \text{ is } R'}{[N/x]R \text{ is at}(R')}\ \texttt{sub/rr} \qquad \frac{[N/x]R \text{ is at}(R')}{[N/x]R \text{ is at}(R')}\ \texttt{sub/rm}$$

$$\frac{y \vdash [N/x]M \text{ is } M'}{[N/x](\lambda y.M) \text{ is } \lambda y.M'}\ \texttt{sub/lam}$$

$$\frac{x \text{ does not occur free in } R}{[N/x]R \text{ is } R}\ \texttt{rrsub/closed}$$

$$\frac{}{[N/x]x \text{ is } N}\ \texttt{rmsub/var}$$

$$\frac{[N/x]R \text{ is } R' \quad [N/x]M \text{ is } M'}{[N/x](R\ M) \text{ is } (R'\ M')}\ \texttt{rrsub/app}$$

$$\frac{[N/x]R \text{ is } \lambda x.O \quad [N/x]M \text{ is } M' \quad [M'/x]O \text{ is } O'}{[N/x](R\ M) \text{ is } O'}\ \texttt{rmsub/app}$$

Figure 6. Hereditary substitution.

Most prior presentations of hereditary substitution track the simple types of the components of the substitution in the definition, making it manifestly total. We instead argue *externally* the totality of this relation over well-typed terms, allowing us to state hereditary substitution as a simple syntactic rewrite. To compensate, we require this extra restriction, which will prove necessary for our approach to the metatheory. Reed discusses an alternative typeless approach [17].

3.2 Metatheory of Hereditary Substitution

The primary result of this paper relates hereditary substitution and canonical forms to the standard notion of substitution and convertibility in the EL. Before that, we must set the stage by showing *internal* soundness and completeness of the canonical forms presentation, which is entirely independent of the EL.

The top-level Hereditary Substitution states [3] :

Proposition 1 (Hereditary Substitution)**.** *If $x : A \vdash M \Leftarrow B$ and $N \Leftarrow A$, then $[N/x]M$ is M', $[N/x]B$ is B', and $M' \Leftarrow B'$.*

We delay the proof sketch until after stating and proving the important lemmas. To untangle the dependencies of the proof, said lemmas must operate over derivations whose type dependencies have been erased. Below we define erasure $-^o$, which takes a dependent type to its corresponding "simple type" or "type skeleton".

$$T \quad ::= \quad \mathsf{o} \mid T \to T$$

$$\begin{aligned} (\mathsf{base}(P))^o &= \mathsf{o} \\ (\Pi x{:}A.\, B)^o &= A^o \to B^o \end{aligned}$$

We also define auxiliary typing judgments $M \Leftarrow^o T$ and $R \Rightarrow^o T$, obeying the same synthesis/checking stratification as before, such that $M \Leftarrow A$ and $A^o = T$ implies $M \Leftarrow^o T$. This definition allows us to prove that substitution preserves simple types by straightforward induction over the simple-typing derivation. Then we can prove that hereditary substitution always exists on well-simply-typed inputs.

Lemma 3.1 (Effectiveness of Hereditary Substitution)**.** *If $x :^o S \vdash M \Leftarrow^o T$ and $N \Leftarrow^o S$, then there exists an M' such that $[N/x]M$ is M'.*

Proof sketch. By lexicographic induction first on S and second on the derivation of $x{:}^oS \vdash M \Leftarrow^o T$.

The majority of cases follow by straightforward induction and rule application. As in proofs of cut in sequent calculus, only one case is interesting, because of its argument that the induction applies to smaller derivations according to the metric:

Case $x :^o S \vdash (R\ M) \Rightarrow^o U$ by the application typing rule and $N :^o S$. Specifically we are in the subcase where $[N/x]R$ is $\lambda y.P$, obtaining the well-typing of P from the fact that substitution preserves simple types.
However, since that derivation of is not a subderivation of that of $(R\ M)$, we need to show that the type T is smaller than S, which follows from the fact that x is the head variable of R.

□

Lemma 3.2 (Permutability of Hereditary Substitutions)**.** *If:*

$$\begin{array}{l} x_1 :^o T_1\ , \quad x_2 :^o T_2 \vdash M :^o S \\ x_1 :^o T_1 \vdash N_2 :^o T_2 \\ N_1 :^o T_1 \end{array}$$

$$\begin{array}{l} [N_1/x_1]N_2 \text{ is } \ulcorner N_2 \\ [N_1/x_1]M \text{ is } \ulcorner M \\ [N_2/x_2]M \text{ is } M\urcorner \end{array}$$

then there exists $\ulcorner M \urcorner$ such that $[N_1/x_1]\ulcorner M$ is $\ulcorner M \urcorner$ and $[\ulcorner N2/x_2]M\urcorner$ is $\ulcorner M\urcorner$.

The hardest part of this lemma is probably just writing down the statement. In addition to the above, we need a theorem statement for each possible combination of substitutions on atoms: two `rrsubs`, an `rrsub` followed by an `rmsub`, an `rmsub` followed by a `sub`. We omit the formal statement of these lemma subparts.

Proof sketch. The proof proceeds by induction over the simple types of the variables and the derivation of the inner substitution. Again, we require the simple types to be part of the termination metric to enable using the i.h. in cases involving `rmsub/app`.

Each case roughly has the form of inverting on the typing derivation to get well-formedness of the subterms in the substitution, then making an inductive call, and reassembling the substitution derivation on the results of the recursive call, and showing (through uniqueness of substitutions, e.g.) that the swapped-order substitution has the same result. □

Once permutability has been proven using simple typing derivations, we can use it on full typing derivations just by erasing the type dependencies.

Finally, we sketch the proof of hereditary substitution.

An instance of the theorem is needed for each syntactic class and relevant formation judgment, including synthesis and checking of terms, atoms, type families, and kinds. The statement varies depending on whether the substitution is into atoms (or atomic types) vs. terms (or canonical type families). Notably, whether the type substitution is an input or output has the same flavor as bidirectional typing, although, since all substitutions are effective and unique, this serves merely for convenience rather than provability.

Theorem 3.3 (Hereditary Substitution)**.** *Assume $M \Leftarrow A$.*

[3] The interested reader may follow along with the files `canonical-simple.elf`, `subst-effect.thm`, `substitution.thm`, and `expand.thm` for this section.

1. *If $x{:}A \vdash R \Rightarrow B$ and $[M/x]R$ is R', then $[M/x]B$ is B' and $R' \Rightarrow B'$.*
2. *If $x{:}A \vdash R \Rightarrow B$ and $[M/x]R$ is N, then $[M/x]B$ is B' and $M \Rightarrow B'$.*
3. *If $x{:}A \vdash N \Leftarrow B$, $[M/x]N$ is N', and $[M/x]B$ is B', then $N' \Leftarrow B'$.*
4. *If $x{:}A \vdash P \Rightarrow K$ and $[M/x]P$ is P', then $[M/x]K$ is K' and $P' \Rightarrow K'$.*
5. *If $x{:}A \vdash B \Leftarrow K$, $[M/x]B$ is B', and $[M/x]K$ is K', then $B' : K'$.*
6. *If $x{:}A \vdash K : \mathsf{kind}$ and $[M/x]K$ is K', then $K' : \mathsf{kind}$.*

Proof sketch. The proof by lexicographic induction A^o and the typing derivation of the open object (e.g. M).

In the application cases, we need to use Substitution Permutability to show permute the substitution into the type, a premise to the typing derivation, can happen after the substitution for the outer free variable referenced by the theorem. □

Thinking of Canonical LF as a logic, the Substitution proof establishes an internal soundness property, by analogy with Cut Admissibility. The corresponding internal *completeness* property is Expansion, which says we can eta-expand atoms into corresponding terms.

To expand an atom R of type A, we apply a function $\eta_{A^o}(R)$ defined on the structure of A^o as follows:

$$\begin{array}{rcl}\eta_o(R) & = & \mathsf{at}(R) \\ \eta_{S\to T}(R) & = & \lambda x.\, \eta_T(R\ \eta_S(x))\end{array}$$

It is worth noting that defining this function over simple types greatly simplifies the metatheory. The analogous theorem in the singleton calculus metatheory [3], defined over dependent types, requires the theorem to be mutually inductive with several seemingly unrelated lemmas. We can stage the proof much more directly in this case.

Again, a key lemma is *permutability*, this time referring to permuting substitution with expansion. Here we will state but not prove this lemma; its proof does not require techniques we have not already discussed.

Lemma 3.4 (Expansion-substitution permutability). *Permutability has five subparts.*

1. *If $[M/x](\eta_T(R))$ is N and x is not R's head, then $[M/x]R$ is R' and $\eta_T(R') = N$.*
2. *If $[M/x](\eta_T(R))$ is N and x is R's head, then $[M/x]R$ is N.*
3. *If $[\eta_T(R)/x]Q$ is Q', then $Q' = [R/x]_{LF}Q$.*
4. *If $[\eta_T(R)/x]Q$ is N, then $\eta_{\_}([R/x]_{LF}Q) = N$.*
5. *If $[\eta_T(R)/x]M$ is M', then $M' = [R/x]_{LF}M$.*

The notation $[-/-]_{LF}$ refers to the non-hereditary, atom-for-atom substitution that the metalogic grants us for free.

Theorem 3.5 (Expansion). *If $R \Rightarrow A$ then $\eta_{A^o}(R) \Leftarrow A$.*

Proof sketch. By induction over the expansion derivation. The base case is trivial. In the arrow case, we inductively get the well-typedness of the newly-introduced variable in the premise. It remains to show that substituting the expansion of the variable for the same variable results in the same type—this follows from permutability. Then, an inductive call on the inner application completes the proof. □

4. Translation

We now have all the necessary tools at our disposal to define the translation from EL to IL. [4]

Translation from EL to IL syntactic objects is defined with respect to the classifier (type or kind) that the IL object inhabits; accordingly, it will be written $\bar{M} \searrow M : A$ for terms, $\bar{A} \searrow A : K$ for type families, and $\bar{K} \searrow K$ for kinds. The notation $\bar{M} \searrow M : A$ means "$\bar{M}$ translates to M at type A", where A is the IL type which M inhabits. This type index exists mainly to treat the type label on the EL lambda, which would otherwise be lost upon translation to an IL lambda and is needed for metatheory.

To translate constants and variables, expand them at the type we read from the signature or context.

$$\frac{\Gamma \vdash x : A \qquad \eta_{A^o}(x) = M}{\Gamma \vdash x \searrow M}$$

$$\frac{\Sigma \vdash c : A \qquad \eta_{A^o}(c) = M}{\Gamma \vdash_\Sigma c \searrow M}$$

To translate an application, first we translate the function, which must become a lambda. Then we translate the argument and (hereditarily) substitute its translation into the body of the lambda.

$$\frac{\begin{array}{c}\Gamma \vdash \bar{M} \searrow \lambda x.\, M : \Pi x{:}B.\, A \quad \Gamma \vdash \bar{N} \searrow N \\ [N/x]M \text{ is } M' \quad [N/x]A \text{ is } A'\end{array}}{\Gamma \vdash (\bar{M}\ \bar{N}) \searrow M' : A'}$$

To translate a lambda, translate the type label, then translate the body under the assumption of the variable having that type.

$$\frac{\Gamma \vdash \bar{A} \searrow A : \mathsf{type} \quad \Gamma, x : A \vdash \bar{M} \searrow M : B}{\Gamma \vdash (\lambda x{:}\bar{A}.\, \bar{M}) \searrow (\lambda x.\, M) : \Pi x{:}A.B}$$

Analogous rules apply to translate constructors and kinds. Of note is the fact that, in order to keep the rules for type constructors analogous to the term case, we need a notion of expansion on type constructors (for the constant case). That is the motivation behind keeping the family-level lambda expression in the system; they are necessary if we want to expand constants of function kind. Alternatively, we could have changed translation to target a disjunctive datatype, but this would have potentially significantly changed the structure of the proof.

We prove the following invariant of translation:

Lemma 4.1 (Translation Regularity).

1. *If $\bar{M} \searrow M : A$, then $M \Leftarrow A$.*
2. *If $\bar{A} \searrow A : K$, then $A \Leftarrow K$.*

Proof sketch. By straightforward induction over the translation judgment. □

5. Completeness

Completeness says that definitionally equivalent terms translate to identical canonical forms, i.e. if $\bar{M} \equiv \bar{N} : \bar{A}$, then $\bar{A} \searrow A : \mathsf{type}$, $\bar{M} \searrow M : A$, $\bar{N} \searrow N : A$, and $M = N$. [5]

As before with substitution and expansion, we will need a lemma that allows us to permute the relevant operation with substitution, in this case translation.

[4] The interested reader may follow along with `translate.elf` for this section.

[5] The interested reader may follow along with the files `trans-sub.thm` and `complete.thm` for this section.

Theorem 5.1 (Translation-substitution permutability)**.** *Assume* $\bar{M} \searrow M : A$.

1. *If* $x{:}A \vdash \bar{N} \searrow N : B$ *then* $[M/x]N$ is N', $[M/x]B$ is B', *and* $[\bar{M}/x]\bar{N} \searrow N' : B'$.
2. *If* $x{:}A \vdash \bar{B} \searrow B : K$ *then* $[M/x]B$ is B', $[M/x]K$ is K', *and* $[\bar{M}/x]\bar{B} \searrow B' : K'$.

Proof sketch. By induction over the open translation derivation (e.g. $x{:}A \vdash \bar{N} \searrow N : B$).

Case $\bar{N} = x$, using rule `trans/var`, meaning $N = \eta_{A^o}(x)$. By Expansion, $x{:}A \vdash N : A$, so we may use Effectiveness to derive the substitution $[M/x]N$ is N'. Because we can derive $[M/x]x$ is M, by Expansion-substitution permutability, $N' = M$.

Case $\bar{N} = (\bar{L}\ \bar{N})$ using the rule `trans/app`, giving us subderivations of $\bar{L} \searrow \lambda y.L : \Pi x{:}B.C$, $\bar{N} \searrow N : B$, $[N/y]L$ is L_y, and $[N/y]C$ is C_y. By i.h., we get that the translations of $[\bar{M}/x]$ into $\bar{N}$, $\bar{L}$, and $\bar{C}$ are equivalent to the hereditary substitution into their translations, with respect to the inner variable, y. To apply the `trans/app` rule to these pieces, we need to swap the substitutions, i.e. show that $[M/x][N/y]L = [[M/x]N/y][M/x]L$ (and similar for C). This is satisfied by hereditary substitution permutability.

Other cases proceed straightforwardly. □

We are now ready to state and prove Completeness in full, which is several mutually inductive lemmas: completeness of typing/formation and completeness of equivalence for all syntactic forms.

Theorem 5.2 (Completeness)**.**

1. *If* $\bar{M} \equiv \bar{N} : \bar{A}$, *then* $\bar{A} \searrow A : \mathsf{type}$, $\bar{M} \searrow M : A$, $\bar{N} \searrow N : A$, *and* $M = N$.
2. *If* $\bar{A} \equiv \bar{B} : \bar{K}$, *then* $\bar{K} \searrow K$, $\bar{A} \searrow A : K$, $\bar{B} \searrow B : K$, *and* $A = B$.
3. *If* $\bar{K} \equiv \bar{K}'$, *then* $\bar{K} \searrow K$, $\bar{K}' \searrow K'$, *and* $K = K'$.
4. *If* $\bar{M} : \bar{A}$, *then* $\bar{A} \searrow A : \mathsf{type}$ *and* $\bar{M} \searrow M : A$.
5. *If* $\bar{A} : \bar{K}$, *then* $\bar{K} \searrow K$ *and* $\bar{A} \searrow A : K$.
6. *If* $\bar{K} : \mathsf{kind}$, *then* $\bar{K} \searrow K$.

It is also mutually inductive with the corresponding cases of (1) and (2) for type families and kinds.

Proof sketch. The proof proceeds by structural induction on the input derivation for all cases. The most interesting cases are for beta and extensionality equivalence rules. We outline these cases below.

Case

$$\frac{\bar{M} : \Pi x{:}\bar{A}.\bar{B}' \quad \bar{N} : \Pi x{:}\bar{A}.\bar{B}'' \qquad x{:}\bar{A} \vdash \bar{M}\ x \equiv \bar{N}\ x : \bar{B}}{\bar{M} \equiv \bar{N} : \Pi x{:}\bar{A}.\bar{B}}\ \texttt{eqtm/ext}$$

By i.h. and inversion, $\bar{A} \searrow A : \mathsf{type}$, $\bar{B}' \searrow B' : \mathsf{type}$, $\bar{B}'' \searrow B'' : \mathsf{type}$, $\bar{M} \searrow M : \Pi x{:}A.B'$, and $\bar{N} \searrow N : \Pi x{:}A.B''$. By i.h. on the equivalence between the applications, inversion on the output, and permutability of expansion with substitution, $\bar{M} \searrow \lambda x.L : \Pi x{:}A.B$ and $\bar{N} \searrow \lambda x.L : \Pi x{:}A.B$.

Case

$$\frac{\bar{N} : \bar{A} \quad x{:}\bar{A} \vdash \bar{M} : \bar{B}}{(\lambda x{:}\bar{A}.\bar{M})\ \bar{N} \equiv [\bar{N}/x]\bar{M} : [\bar{N}/x]\bar{B}}\ \texttt{eqtm/beta}$$

By i.h., $\bar{A} \searrow A : \mathsf{type}$, $\bar{N} \searrow N : A$, $x{:}A \vdash \bar{B} \searrow B : \mathsf{type}$, and $x{:}A \vdash \bar{M} \searrow M : B$.

By translation-substitution permutability,

$$[N/x]M \text{ is } M'$$
$$[N/x]B \text{ is } B'$$
$$[\bar{N}/x]\bar{M} \searrow M' : B'$$
$$[\bar{N}/x]\bar{B} \searrow B' : \mathsf{type}$$

The rules `trans/lam` and `trans/app` complete the case.

The remaining cases are straightforward. □

6. Soundness

Soundness says that terms that translate to identical canonical forms are definitionally equivalent, i.e. [6]

Proposition 2. *If* $\bar{M} \searrow M : A$, $\bar{N} \searrow M : A$, *and* $\bar{A} \searrow A : \mathsf{type}$, *then* $\bar{M} \equiv \bar{N} : \bar{A}$.

We achieve this result by defining *transliteration*, a back translation from IL syntactic forms to EL, and proving that translation composed with transliteration produces equivalent terms. Transliteration is summarized in the appendix.

At first glance, it appears that soundness should be the easy direction: canonical forms are "obviously" a subset of the full term language of LF. The tricky part is justifying hereditary substitution in terms of definitional equivalence. We want a lemma like we had in completeness about translation (transliteration) permuting with substitution, except in this case, it doesn't quite: if we substitute after transliterating, we may wind up with redexes, whereas performing hereditary substitution IL-side would eliminate those. Instead, we say that the results are definitionally equivalent:

Proposition 3 (Transliteration-substitution permutability)**.** *If* $x{:}A \vdash M \Leftarrow B \nearrow \bar{M}$, $N \Leftarrow A \nearrow \bar{N}$, $[N/x]M$ is M', *and* $[N/x]B$ is B', *then there exists* $\bar{O}$ *such that* $M' \Leftarrow B' \nearrow \bar{O}$ *and* $[\bar{N}/x]\bar{M} \equiv \bar{O}$.

The above lemma wound up being the most difficult and time-consuming part of the proof due to numerous unsuccessful attempts.

In order to maintain a strong enough invariant for the inductive calls to go through, we need to exploit the fact that the relation between $[\bar{N}/x]\bar{M}$ and $\bar{O}$ is *directional*, i.e. the former *reduces* to the latter. But we also needed to disentangle the theory about constructors from the theory about terms to avoid having circular dependencies in the proof.

Our insight was to define a notion of untyped computation separate from definitional equivalence. This lets us set the stage with the necessary lemmas over constructor-level equivalence, specifically *injectivity of Pi*, before we need to recall such a property for the metatheory of term-level equivalence.

First, we define term-level reduction $M \longrightarrow M'$ as follows.

$$\frac{}{\bar{M} \longrightarrow \bar{M}}\ \texttt{reduce/refl} \qquad \frac{\bar{A} \longrightarrow \bar{A}' \quad \bar{M} \longrightarrow \bar{M}'}{\lambda x{:}\bar{A}.\bar{M} \longrightarrow \lambda x{:}\bar{A}'.\bar{M}'}\ \texttt{reduce/lam}$$

$$\frac{}{(\lambda x{:}\bar{A}.\bar{M})\ \bar{N} \longrightarrow [\bar{N}/x]\bar{M}}\ \texttt{reduce/beta}$$

$$\frac{\bar{M} \longrightarrow \bar{M}' \quad \bar{N} \longrightarrow \bar{N}'}{\bar{M}\ \bar{N} \longrightarrow \bar{M}'\ \bar{N}'}\ \texttt{reduce/app}$$

Reduction on constructors just reduces the inner terms:

$$\frac{}{a \longrightarrow a}\ \texttt{tpreduce/const}$$

[6] The interested reader may follow along with the files `pi-inj.thm`, `convert-sub.thm`, `reduce-equiv.thm`, and `sound.thm` for this section.

$$\frac{\bar{A} \longrightarrow \bar{A}' \quad \bar{B} \longrightarrow \bar{B}'}{\Pi x{:}\bar{A}.\bar{B} \longrightarrow \Pi x{:}\bar{A}'.\bar{B}'}\ \texttt{tpreduce/pi}$$

$$\frac{\bar{A} \longrightarrow \bar{A}' \quad \bar{B} \longrightarrow \bar{B}'}{\lambda x{:}\bar{A}.\bar{B} \longrightarrow \lambda x{:}\bar{A}'.\bar{B}'}\ \texttt{tpreduce/lam}$$

$$\frac{\bar{A} \longrightarrow \bar{A}' \quad \bar{M} \longrightarrow \bar{M}'}{\bar{A}\ \bar{M} \longrightarrow \bar{A}'\ \bar{M}'}\ \texttt{tpreduce/app}$$

Notably, there is no beta rule for constructors. Reduction on kinds is similar.

$\longrightarrow^*$ denotes the reflexive, transitive closure of $\longrightarrow$. After proving a few easy compatibility lemmas for $\longrightarrow^*$, we can state the permutability lemma in terms of the transitive closure:

Lemma 6.1 (Transliteration-substitution permutability (reduction))**.** *Suppose $N \Leftarrow A \nearrow \bar{N}$.*

1. *If $x{:}A \vdash M \Leftarrow B \nearrow \bar{M}$, $[N/x]M$* is *M', and $[N/x]B$* is *B', there exists $\bar{O}$ s.t. $M' \Leftarrow B' \nearrow \bar{O}$ and $[\bar{N}/x]\bar{M} \longrightarrow^* \bar{O}$.*
2. *If $x{:}A \vdash R \Rightarrow B \nearrow \bar{M}$, $[N/x]R$* is *R', and $[N/x]B$* is *B', there exists $\bar{O}$ s.t. $R' \Rightarrow B' \nearrow \bar{O}$ and $[\bar{N}/x]\bar{M} \longrightarrow^* \bar{O}$*
3. *If $x{:}A \vdash R \Rightarrow B \nearrow \bar{M}$, $[N/x]R$* is *M, and $[N/x]B$* is *B', there exists $\bar{O}$ s.t. $M \Leftarrow B' \nearrow \bar{O}$ and $[\bar{N}/x]\bar{M} \longrightarrow^* \bar{O}$*
4. *If $x{:}A \vdash B \Leftarrow K \nearrow \bar{B}$, $[N/x]B$* is *B', and $[N/x]K$* is *K', there exists $\bar{C}$ s.t. $B' \Leftarrow K' \nearrow \bar{C}$ and $[\bar{N}/x]\bar{B} \longrightarrow^* \bar{C}$.*
5. *If $x{:}A \vdash P \Rightarrow K \nearrow \bar{B}$, $[N/x]P$* is *P', and $[N/x]K$* is *K', there exists $\bar{C}$ s.t. $P' \Rightarrow K' \nearrow \bar{C}$ and $[\bar{N}/x]\bar{B} \longrightarrow^* \bar{C}$.*
6. *If $x{:}A \vdash K \nearrow \bar{K}$ and $[N/x]K$* is *K', there exists $\bar{K}'$ s.t. $K' \nearrow \bar{K}'$ and $[\bar{N}/x]\bar{K} \longrightarrow^* \bar{K}'$.*

Proof. By lexicographic induction on A^o and the transliteration derivation for the open term. The case when the substitution is hereditary involves the `reduce/beta` rule, as expected. □

It remains to relate reduction and definitional equivalence. In fact, this implication is nontrivial, and in the course of the development we made sure to prove it before hinging the permutability lemma on this fact.

Consider proving the following proposition: if $\bar{M} : \bar{A}$ and $\bar{M} \longrightarrow \bar{N}$, then $\bar{M} \equiv \bar{N} : \bar{A}$.

Case

$$\frac{\lambda x{:}\bar{A}'.\bar{M} : \Pi x{:}\bar{A}.\bar{B} \quad \bar{N} : \bar{A}}{(\lambda x{:}\bar{A}'.\bar{M})\ \bar{N} : [\bar{N}/x]\bar{B}}\ \texttt{eof/app}$$

and $(\lambda x{:}\bar{A}'.\bar{M})\ \bar{N} \longrightarrow [\bar{N}/x]\bar{M}$.
Need to show $(\lambda x{:}\bar{A}'.\bar{M})\ \bar{N} \equiv [\bar{N}/x]\bar{M} : [\bar{N}/x]\bar{B}$

We cannot apply the `equiv/beta` rule here without a derivation of the well-typing of M, meaning we need to invert the typing of the lambda. So let us try to prove the following:

If $\lambda x{:}\bar{A}.\bar{M} : \bar{C}$, then $\bar{C} \equiv \Pi x{:}\bar{A}.\bar{B}$ and $x{:}\bar{A} \vdash \bar{M} : \bar{B}$.

Case

$$\frac{\lambda x{:}\bar{A}'.\bar{M} : \Pi x{:}\bar{A}.\bar{B}' \quad x{:}\bar{A} \vdash (\lambda x{:}\bar{A}'.\bar{M})\ x : \bar{B}}{\lambda x{:}\bar{A}'.\bar{M} : \Pi x{:}\bar{A}.\bar{B}}\ \texttt{eof/ext}$$

Need to show: $x{:}\bar{A} \vdash \bar{M} : \bar{B}$

In order to show the second output of the theorem, we will need to manipulate the second premise of the extensionality rule from typing a redex $(\lambda x{:}\bar{A}'.\bar{M})\ x$ to typing its contractum $\bar{M}$. But that follows from what we were proving in the first place: that reduction preserves typing. So we must make these lemmas mutually inductive.

In addition, both of these cases rely on the well-formation of the components of the Pi type involved, in order to invoke the needed substitution of $x{:}A$ for $x{:}A'$. This lemma amounts to "injectivity of Pi":

Proposition 4 (Injectivity of Pi)**.** *If $\Pi x{:}\bar{A}.\bar{B} \equiv \Pi x{:}\bar{A}'.\bar{B}' : \mathsf{type}$, then $\bar{A} \equiv \bar{A}' : \mathsf{type}$ and $x{:}\bar{A} \vdash \bar{B} \equiv \bar{B}' : \mathsf{type}$.*

Most of the naive proof of this proposition is easy, but it founders on the extensionality rule due to transitivity: two Pis might be equivalent through some mediating term which isn't a Pi. [7]

Therefore, we need to generalize this lemma to get it to go through. We define two more notions, *normalization* of constructors at the constructor level (maintaining term redexes), and *similarity* of constructors at a given kind, which is like equivalence only stronger: in particular, it implies injectivity.

Effectively, this gives us a way of relating two constructors that are made of equivalent Pis, under some arbitrary (but equal) number of lambdas. This equivalence is defined with respect to *constructor-level* reduction; term-level reduction is a completely separate notion.

$$\frac{}{\Pi x{:}\bar{A}.\bar{B} \downarrow \Pi x{:}\bar{A}.\bar{B}}\ \texttt{norm/pi}$$

$$\frac{\bar{B} \downarrow \bar{B}'}{\lambda x{:}\bar{A}.\bar{B} \downarrow \lambda x{:}\bar{A}.\bar{B}'}\ \texttt{norm/lam}$$

$$\frac{B \downarrow \lambda x{:}\bar{A}.\bar{C}}{(\bar{B}\ \bar{M}) \downarrow [\bar{M}/x]\bar{C}}\ \texttt{norm/app}$$

Similarity $\bar{A} \sim \bar{B} : \bar{K}$ reduces to equivalence in the case of Pi and simply strips off binders in the case of lambda.

$$\frac{\bar{A} \equiv \bar{A}' : \mathsf{type} \quad x{:}\bar{A} \vdash \bar{B} \equiv \bar{B}' : \mathsf{type}}{\Pi x{:}\bar{A}.\bar{B} \sim \Pi x{:}\bar{A}'.\bar{B}' : \mathsf{type}}\ \texttt{sim/pi}$$

$$\frac{x{:}\bar{C} \vdash \bar{B} \sim \bar{B}' : \bar{K}}{\lambda x{:}\bar{A}.\bar{B} \sim \lambda x{:}\bar{A}'.\bar{B}' : \Pi x{:}\bar{C}.\bar{K}}\ \texttt{sim/lam}$$

Lemma 6.2 (Generalized injectivity of Pi)**.** *If $\bar{A} \equiv \bar{B} : \bar{K}$ and $\bar{A} \downarrow \bar{A}'$, then $\bar{B} \downarrow \bar{B}'$ and $\bar{A}' \sim \bar{B}' : K$.*

Proof. By induction on the derivation of equivalence. □

Now Pi Injectivity is a special case:

Corollary 6.3 (Injectivity of Pi)**.** *If $\Pi x{:}\bar{A}.\bar{B} \equiv \Pi x{:}\bar{A}'.\bar{B}' : \mathsf{type}$, then $\bar{A} \equiv \bar{A}' : \mathsf{type}$ and $x{:}\bar{A} \vdash \bar{B} \equiv \bar{B}' : \mathsf{type}$.*

Now we can relate reduction and equivalence.

Lemma 6.4 (Reduction implies equivalence)**.** *This lemma is mutually inductive with inversion of lambda typing.*

1. *If $\bar{M} : \bar{A}$ and $\bar{M} \longrightarrow \bar{N}$, then $\bar{M} \equiv \bar{N} : \bar{A}$.*
2. *If $\lambda x{:}\bar{A}.\bar{M} : \bar{C}$, then $\bar{C} \equiv \Pi x{:}\bar{A}.\bar{B}$ and $x{:}\bar{A} \vdash \bar{M} : \bar{B}$.*

Proof.

Case of (1)

$$\frac{\lambda x{:}\bar{A}'.\bar{M} : \Pi x{:}\bar{A}.\bar{B} \quad \bar{N} : \bar{A}}{(\lambda x{:}\bar{A}'.\bar{M})\ \bar{N} : [\bar{N}/x]\bar{B}}\ \texttt{eof/app}$$

and $(\lambda x{:}\bar{A}'.\bar{M})\ \bar{N} \longrightarrow [\bar{N}/x]\bar{B}$.
$\Pi x{:}\bar{A}.\bar{B} \equiv \Pi x{:}\bar{A}'.\bar{B}' : \mathsf{type}$ and $x{:}\bar{A}' \vdash \bar{M} : \bar{B}'$ by i.h.(2).

[7] It was for this reason that Harper and Pfenning [8] left constructor-level lambda out of their formulation.

$\bar{A} \equiv \bar{A}' : \mathsf{type}$ by Pi Injectivity.
$\bar{N} : \bar{A}'$ by rule.

$$\frac{x{:}\bar{A}' \vdash \bar{M} : \bar{B} \quad \bar{N} : \bar{A}'}{(\lambda x{:}\bar{A}'.\bar{M})\ \bar{N} \equiv [\bar{N}/x]\bar{B} : [\bar{N}/x]\bar{B}}\ \texttt{equiv/beta}$$

Case of (2)

$$\frac{\lambda x{:}\bar{A}'.\bar{M} : \Pi x{:}\bar{A}.\bar{B}' \quad x{:}\bar{A} \vdash (\lambda x{:}\bar{A}'.\bar{M})\ x : \bar{B}}{\lambda x{:}\bar{A}'.\bar{M} : \Pi x{:}\bar{A}.\bar{B}}\ \texttt{eof/ext}$$

By rule, $(\lambda x{:}\bar{A}'.\bar{M})\ x \longrightarrow [x/x]\bar{M}$.
By i.h.(1), $x{:}\bar{A} \vdash (\lambda x{:}\bar{A}'.\bar{M})\ x \equiv \bar{M} : \bar{B}$.
By regularity of equivalence, $x{:}\bar{A} \vdash \bar{M} : \bar{B}$, which is almost what we need except for x's type.
By i.h.(2), $\Pi x{:}\bar{A}.\bar{B}' \equiv \Pi x{:}\bar{A}'.\_$.
By Pi Injectivity, $\bar{A} \equiv \bar{A}' : \mathsf{type}$; therefore we can substitute $x{:}\bar{A}'$ for $x{:}\bar{A}$, as needed.

The remaining cases are straightforward. □

We can extend the above to $\longrightarrow^*$ with a trivial induction over the transitive-reflective closure; finally, we can restate permutability in terms of equivalence, as in the original proposition, and prove it via appeal to Transliteration-substitution permutability (reduction) and Reduction equivalence.

Having closed off the key permutability lemma, we can finally prove the takeaway theorem about transliteration:

Theorem 6.5 (Translation inversion)**.** *If $\bar{M} \searrow M : A$, then $M : A \nearrow \bar{M}'$, $A : \mathsf{type} \nearrow \bar{A}$, and $\bar{M} \equiv \bar{M}' : \bar{A}$.*

Proof sketch. By induction on the translation derivation. The one hard case is application.

Case

$$\frac{\begin{array}{c}\bar{M} \searrow \lambda x.L : \Pi x{:}B.A \quad \bar{N} \searrow N : B \\ [N/x]L \text{ is } L' \quad [N/x]A \text{ is } A'\end{array}}{\bar{M}\ \bar{N} \searrow L' : A'}\ \texttt{trans/app}$$

By i.h.,

$$\begin{array}{c} N \Leftarrow B \nearrow \bar{N}', \\ B \Leftarrow \mathsf{type} \nearrow \bar{B}, \\ x{:}B \vdash L \Leftarrow A \nearrow \bar{L}, \\ x{:}B \vdash A \Leftarrow \mathsf{type} \nearrow \bar{A}, \end{array}$$

$\bar{N} \equiv \bar{N}' : \bar{B}$, and $\bar{M} \equiv \lambda x{:}\bar{B}.\bar{L} : \Pi x{:}\bar{B}.\bar{A}$.
By transliteration-substitution permutability, $L' \Leftarrow A' \nearrow \bar{L}'$ such that $\bar{L}' \equiv [\bar{N}'/x]\bar{L} : [\bar{N}'/x]\bar{A}$.
By substitution equivalence (a lemma we omitted), because $\bar{N} \equiv \bar{N}'$, we get $[\bar{N}/x]\bar{L} \equiv [\bar{N}'/x]\bar{L} : [\bar{N}/x]\bar{A}$.
Similar reasoning for A gets us the remainder of the premises we need; the rest follows from equivalence rules.

□

We derive the original soundness theorem as a corollary.

Corollary 6.6 (Soundness)**.** *If $\bar{M} \searrow M : A$, $\bar{N} \searrow M : A$, and $\bar{A} \searrow A : \mathsf{type}$, then $\bar{M} \equiv \bar{N} : \bar{A}$.*

Proof. From translation inversion on the first and second inputs, we get $\bar{M} \equiv \bar{M}' : \bar{A}'$ and $\bar{N} \equiv \bar{M}' : \bar{A}'$ (A' in both causes because type conversion is a function).

$A \equiv A'$ by the type translation analog of translation inversion.

The rest is just application of equivalence rules:

$$\frac{\dfrac{\bar{M} \equiv \bar{M}' : \bar{A}' \quad \dfrac{\bar{A} \equiv \bar{A}' : \mathsf{type}}{\bar{A}' \equiv \bar{A} : \mathsf{type}}}{\bar{M} \equiv \bar{M}' : \bar{A}} \quad \dfrac{\dfrac{\bar{N} \equiv \bar{M}' : \bar{A}' \quad \dfrac{\bar{A} \equiv \bar{A}' : \mathsf{type}}{\bar{A}' \equiv \bar{A} : \mathsf{type}}}{\bar{N} \equiv \bar{M}' : \bar{A}}}{\bar{M}' \equiv \bar{N} : \bar{A}}}{\bar{M} \equiv \bar{N} : \bar{A}}$$

□

7. Discussion

We presented a proof of the metatheory of LF using syntactic techniques that we have formalized in Twelf. The mechanization is approximately 36 KLOC (including comments and whitespace) and takes about 8 seconds to check on a dual-core Ubuntu laptop from 2008.

The proof represents a significant proof engineering effort as well as a theoretical contribution, which effort we have not discussed here. We use Crary's explicit contexts technique [4], needed in certain cases to refer to dependencies on variables in the ambient context, for a great deal of the metatheory, which unfortunately winds up bloating the proof with a lot of repetitive boilerplate. Having a way to write the technique as a library would improve matters; the authors have not seriously investigated whether the LF module system [16] would serve this purpose.

A potential contribution of this work is that extensions to LF, such as some of those mentioned in the introduction that form the basis of Ph.D. theses [11, 19], could be proportionally formalized as extensions to this development. Other projects requiring the metatheory of dependent type theories in general might use this development as a starting point. A litmus test of this claim of extensibility might be Reed's discussion of adding base type polymorphism to LF [18].

To summarize, our contributions are:

- A novel syntactic account of the metatheory of LF with constructor-level lambda and extensionality
- A validation of existing work based on Canonical LF and hereditary substitution
- A Twelf mechanization of this metatheory which may be freely referenced and extended.

Acknowledgments

The authors would like to acknowledge William Lovas, Daniel Licata, and Robert Simmons for helpful insights, comments, and encouragement throughout this work.

References

[1] B. E. Aydemir, A. Bohannon, M. Fairbairn, J. N. Foster, B. C. Pierce, P. Sewell, D. Vytiniotis, S. Weirich, and S. Zdancewic. Mechanized metatheory for the masses: The poplmark challenge. In *In Theorem Proving in Higher Order Logics: 18th International Conference, number 3603 in LNCS*, pages 50–65. Springer-Verlag, 2005.

[2] S. Cittadini. Intercalation calculus for intuitionistic propositional logic. Technical report, 1992. URL `http://repository.cmu.edu/philosophy/251/`. Available as a technical report as CMU-PHIL-29.

[3] K. Crary. A syntactic account of singleton types via hereditary substitution. In *Proceedings of the Fourth International Workshop on Logical Frameworks and Meta-Languages: Theory and Practice*, LFMTP '09, pages 21–29, New York, NY, USA, 2009. ACM. ISBN 978-1-60558-529-1.

[4] K. Crary. Explicit contexts in lf (extended abstract). *Electr. Notes Theor. Comput. Sci.*, 228:53–68, 2009.

[5] A. Felty. Encoding dependent types in an intuitionistic logic. In G. Huet and G. D. Plotkin, editors, *Logical Frameworks*, pages 214–251. Cambridge University Press, 1991.

[6] D. Garg and F. Pfenning. A proof-carrying file system. In D.Evans and G.Vigna, editors, *Proceedings of the 31st Symposium on Security and Privacy (Oakland 2010)*, Berkeley, California, May 2010. IEEE. Extended version available as Technical Report CMU-CS-09-123, June 2009.

[7] R. Harper and D. R. Licata. Mechanizing metatheory in a logical framework. *Journal of Functional Programming*, 17(4–5):613–673, July 2007.

[8] R. Harper and F. Pfenning. On equivalence and canonical forms in the LF type theory. *ACM Trans. on Computational Logic*, 6:61–101, Jan. 2005.

[9] R. Harper, F. Honsell, and G. Plotkin. A framework for defining logics. *Symposium on Logic in Computer Science*, pages 194–204, June 1987.

[10] D. K. Lee, K. Crary, and R. Harper. Towards a mechanized metatheory of Standard ML. In *2007 Symposium on Principles of Programming Languages*, 2007.

[11] W. Lovas. *Refinement Types for Logical Frameworks*. PhD thesis, Carnegie Mellon, September 2010. URL `http://www.cs.cmu.edu/~wlovas/papers/wjl-thesis-final.pdf`. Available as technical report CMU-CS-10-138.

[12] T. Murphy, VII. *Modal Types for Mobile Code*. PhD thesis, Carnegie Mellon, January 2008. URL `http://tom7.org/papers/`. Available as technical report CMU-CS-08-126.

[13] F. Pfenning. Judgments and propositions, Jan. 2010. URL `http://www.cs.cmu.edu/~fp/courses/15816-s12/lectures/09-focusing.pdf`. Lecture notes for 15-816: Modal Logic at Carnegie Mellon University.

[14] F. Pfenning and C. 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.

[15] J. Polakow. *Ordered Linear Logic and Applications*. PhD thesis, Carnegie Mellon, 2001. URL `http://www.cs.cmu.edu/~jpolakow/`. Available as technical report CMU-CS-01-152.

[16] F. Rabe and C. Schürmann. A practical module system for lf. In J. Cheney and A. P. Felty, editors, *LFMTP*, pages 40–48. ACM, 2009. ISBN 978-1-60558-529-1.

[17] J. Reed. Properties of hereditary substitution without type indices, May 2007. URL `http://www.cs.cmu.edu/~jcreed/papers/st2.pdf`. Unpublished draft.

[18] J. Reed. Base-type polymorphism in LF, Feb. 2008. URL `http://www.cs.cmu.edu/~jcreed/papers/lfb.pdf`. Unpublished draft.

[19] J. Reed. *A Hybrid Logical Framework*. PhD thesis, Carnegie Mellon, September 2009. URL `http://jcreed.org/papers/thesis-final-color.pdf`. Available as technical report CMU-CS-09-155.

[20] C. Urban, J. Cheney, and S. Berghofer. Mechanizing the metatheory of LF. *ACM Trans. Comput. Log.*, 12(2):15, 2011.

[21] K. Watkins, I. Cervesato, F. Pfenning, and D. Walker. A concurrent logical framework i: Judgments and properties. Technical report, 2003.

A. EL Rules

Full EL (LF) type system.

Signature checking

$\boxed{\Sigma\ \mathsf{ok}}$

$$\frac{}{\cdot\ \mathsf{ok}} \qquad \frac{\Sigma\ \mathsf{ok} \quad \vdash_\Sigma \bar{K} : \mathsf{kind}}{\Sigma, a : \bar{K}\ \mathsf{ok}} \qquad \frac{\Sigma\ \mathsf{ok} \quad \vdash_\Sigma \bar{A} : \mathsf{type}}{\Sigma, c : \bar{A}\ \mathsf{ok}}$$

$\boxed{\bar{A} : \bar{K}}$

$$\frac{a : \bar{K} \in \Sigma \quad \bar{K} : \mathsf{kind}}{\vdash_\Sigma a : \bar{K}}\ \texttt{ekof/const}$$

$$\frac{\bar{A} : \mathsf{type} \quad x : \bar{A} \vdash \bar{B} : \mathsf{type}}{\Pi x{:}\bar{A}.\bar{B} : \mathsf{type}}\ \texttt{ekof/pi}$$

$$\frac{\bar{A} : \mathsf{type} \quad X : \bar{A} \vdash \bar{B} : \bar{K}}{\lambda x{:}\bar{A}.\bar{B} : \Pi x{:}\bar{A}.\bar{K}}\ \texttt{ekof/lam}$$

$$\frac{\bar{A} : \Pi x{:}\bar{B}.\bar{K} \quad \bar{M} : \bar{B}}{(\bar{A}\ \bar{M}) : [\bar{M}/x]\bar{K}}\ \texttt{ekof/app}$$

$$\frac{\bar{C} : \Pi x{:}\bar{A}.\bar{K}' \quad x : \bar{A} \vdash (\bar{C}\ x) : \bar{K}}{\bar{C} : \Pi x{:}\bar{A}.\bar{K}}\ \texttt{ekof/ext}$$

$$\frac{\bar{A} : \bar{K} \quad \bar{K} \equiv \bar{L}}{\bar{A} : \bar{L}}\ \texttt{ekof/eqkind}$$

Term typing

$\boxed{\bar{M} : \bar{A}}$

$$\frac{c : \bar{A} \in \Sigma \quad \bar{A} : \mathsf{type}}{\vdash_\Sigma c : \bar{A}}\ \texttt{eof/const} \qquad \frac{x : \bar{A} \in \Gamma \quad \bar{A} : \mathsf{type}}{\Gamma \vdash x : \bar{A}}\ \texttt{eof/var}$$

$$\frac{x{:}\bar{A} \vdash \bar{M} : \bar{B}}{\lambda x{:}\bar{A}.\,\bar{M} : \Pi x{:}\bar{A}.\,\bar{B}}\ \texttt{eof/lam} \qquad \frac{\bar{M} : \Pi x{:}\bar{A}.\bar{B} \quad \bar{N} : \bar{A}}{(\bar{M}\ \bar{N}) : [\bar{N}/x]\bar{B}}\ \texttt{eof/app}$$

$$\frac{\bar{M} : \Pi x{:}\bar{A}.\bar{B}' \quad x : \bar{A} \vdash (\bar{M}\ x) : \bar{B}}{\bar{M} : \Pi x{:}\bar{A}.\bar{B}}\ \texttt{eof/ext}$$

$$\frac{\bar{M} : \bar{A} \qquad \bar{A} \equiv \bar{B} : \mathsf{type}}{\bar{M} : \bar{B}}\ \texttt{eof/eqtp}$$

Kind well-formedness

$\boxed{\bar{K} : \mathsf{kind}}$

$$\frac{}{\mathsf{type} : \mathsf{kind}}\ \texttt{ewfkind/tp} \qquad \frac{x : \bar{A} \vdash \bar{K} : \mathsf{kind}}{\Pi x{:}\bar{A}.\bar{K} : \mathsf{kind}}\ \texttt{ewfkind/pi}$$

Term equivalence

$\boxed{\bar{M} \equiv \bar{N} : \bar{A}}$

$$\frac{\bar{A} \equiv \bar{A}' : \mathsf{type} \quad x : \bar{A} \vdash \bar{M} \equiv \bar{M}' : \bar{B}}{\lambda x{:}\bar{A}.\bar{M} \equiv \lambda x{:}\bar{A}'.\bar{M}' : \Pi x{:}\bar{A}.\bar{B}}\ \texttt{eqtm/lam}$$

$$\frac{\bar{M} \equiv \bar{M}' : \Pi x{:}\bar{A}.\bar{B} \quad \bar{N} \equiv \bar{N}' : \bar{A}}{(\bar{M}\ \bar{N}) \equiv (\bar{M}'\ \bar{N}') : [\bar{M}'/x]\bar{B}}\ \texttt{eqtm/app}$$

$$\frac{\begin{array}{c}\bar{M} : \Pi x{:}\bar{A}.\bar{B}' \quad \bar{M} : \Pi x{:}\bar{A}.\bar{B}'' \\ x : \bar{A} \vdash (\bar{M}\ x) \equiv (\bar{N}\ x) : \bar{B}\end{array}}{\bar{M} \equiv \bar{N} : \Pi x{:}\bar{A}.\bar{B}}\ \texttt{eqtm/ext}$$

$$\frac{x : \bar{A} \vdash \bar{M} : \bar{B} \quad \bar{N} : \bar{A}}{(\lambda x{:}\bar{A}.\bar{M})\ \bar{N} \equiv [\bar{N}/x]\bar{M} : [\bar{N}/x]\bar{B}}\ \texttt{eqtm/beta}$$

$$\frac{\bar{M} \equiv \bar{N} : \bar{A}}{\bar{N} \equiv \bar{M} : \bar{A}}\ \texttt{eqtm/sym} \qquad \frac{\bar{M} \equiv \bar{N} : \bar{A} \quad \bar{N} \equiv \bar{O} : \bar{A}}{\bar{M} \equiv \bar{O} : \bar{A}}\ \texttt{eqtm/trans}$$

$$\frac{\bar{M} : \bar{A}}{\bar{M} \equiv \bar{M} : \bar{A}}\ \texttt{eqtm/refl}$$

$$\frac{\bar{M} \equiv \bar{N} : \bar{A} \quad \bar{A} \equiv \bar{B} : \mathsf{type}}{\bar{M} \equiv \bar{N} : \bar{B}}\ \texttt{eqtm/typecon}$$

Family equivalence

$\boxed{\bar A \equiv \bar B : \bar K}$

$$\frac{\bar A \equiv \bar A' : \Pi x{:}\bar B.\bar K \quad \bar M \equiv \bar M' : \bar B}{(\bar A\ \bar M) \equiv (\bar A'\ \bar M') : [\bar M/x]\bar K}\ \texttt{eqtp/app}$$

$$\frac{\bar A \equiv \bar A' : \mathsf{type} \quad x : \bar A \vdash \bar B \equiv \bar B' : \bar K}{\lambda x{:}\bar A.\bar B \equiv \lambda x{:}\bar A'.\bar B' : \Pi x{:}\bar A.\bar K}\ \texttt{eqtp/lam}$$

$$\frac{\begin{array}{c}\bar A : \Pi x{:}\bar C.\bar K' \quad \bar B : \Pi x{:}\bar C.\bar K'' \\ x : \bar C \vdash (\bar A\ x) \equiv (\bar B\ x) : \bar K\end{array}}{\bar A \equiv \bar B : \Pi x{:}\bar C.\bar K}\ \texttt{eqtp/ext}$$

$$\frac{x : \bar A \vdash \bar B : \bar K \quad \bar N : \bar A}{(\lambda x{:}\bar A.\bar B)\ \bar N \equiv [\bar N/x]\bar B : [\bar N/x]\bar K}\ \texttt{eqtp/beta}$$

$$\frac{\bar A \equiv \bar B : \bar K}{\bar B \equiv \bar A : \bar K}\ \texttt{eqtp/sym} \qquad \frac{\bar A \equiv \bar B : \bar K \quad \bar B \equiv \bar C : \bar K}{\bar A \equiv \bar C : \bar K}\ \texttt{eqtp/trans}$$

$$\frac{\bar A : \bar K}{\bar A \equiv \bar A : \bar K}\ \texttt{eqtp/refl} \qquad \frac{\bar A \equiv \bar B : \bar K \quad \bar K \equiv \bar L}{\bar A \equiv \bar B : \bar L}\ \texttt{eqtp/kcon}$$

Kind equivalence

$\boxed{\bar K \equiv \bar L}$

$$\frac{}{\mathsf{type} \equiv \mathsf{type}}\ \texttt{eqkind/tp} \qquad \frac{\bar K : \mathsf{kind}}{\bar K \equiv \bar K}\ \texttt{eqkind/refl}$$

$$\frac{\bar A \equiv \bar A' : \mathsf{type} \quad x : \bar A \vdash \bar K \equiv \bar K'}{\Pi x{:}\bar A.\bar K \equiv \Pi x{:}\bar A'.\bar K'}\ \texttt{eqkind/pi}$$

$$\frac{\bar L \equiv \bar L}{\bar L \equiv \bar K}\ \texttt{eqkind/sym} \qquad \frac{\bar K \equiv \bar L \quad \bar L \equiv \bar J}{\bar K \equiv \bar J}\ \texttt{eqkind/trans}$$

B. Transliteration

$\boxed{M \Leftarrow A \nearrow \bar M}$

$$\frac{R \Rightarrow \mathsf{base}(P) \nearrow \bar M}{\mathsf{at}(R) \Leftarrow \mathsf{base}(P) \nearrow \bar M}\ \texttt{convert/at}$$

$$\frac{\begin{array}{c}A \Leftarrow \mathsf{type} \nearrow \bar A \\ x{:}A \vdash M \Leftarrow B \nearrow \bar M\end{array}}{\lambda x.M \Leftarrow \Pi x{:}A.B \nearrow \lambda x{:}\bar A.\bar M}\ \texttt{convert/lam}$$

$\boxed{R \Rightarrow A \nearrow \bar M}$

$$\frac{c : A \in \Sigma}{c \Rightarrow A \nearrow \bar c}\ \texttt{convert/const}$$

$$\frac{x{:}A \in \Gamma}{x \Rightarrow A \nearrow \bar x}\ \texttt{convert/var}$$

$$\frac{R \Rightarrow \Pi x{:}A.B \nearrow \bar M \quad N \Leftarrow A \nearrow \bar N \quad [N/x]B \text{ is } B'}{(R\ N) \Rightarrow B' \nearrow (\bar M\ \bar N)}\ \texttt{convert/app}$$

$\boxed{A \Leftarrow K \nearrow \bar A}$

$$\frac{P \Rightarrow \mathsf{type} \nearrow A}{\mathsf{base}(P) \Leftarrow \mathsf{type} \nearrow \bar A}\ \texttt{tpconvert/base}$$

$$\frac{\begin{array}{c}A \Leftarrow \mathsf{type} \nearrow \bar A \\ x{:}A \vdash B \Leftarrow \mathsf{type} \nearrow \bar B\end{array}}{\Pi x{:}A.B \Leftarrow \mathsf{type} \nearrow \Pi x{:}\bar A.\bar B}\ \texttt{tpconvert/pi}$$

$$\frac{A \Leftarrow \mathsf{type} \nearrow \bar A \quad x{:}A \vdash B \Leftarrow K \nearrow \bar B}{\lambda x{:}A.B \Leftarrow \Pi x{:}A.K \nearrow \lambda x{:}\bar A.\bar B}\ \texttt{tpconvert/lam}$$

$\boxed{P \Rightarrow K \nearrow \bar A}$

$$\frac{a : K \in \Sigma}{a \Rightarrow K \nearrow \bar a}\ \texttt{atpconvert/const}$$

$$\frac{\begin{array}{c}P \Rightarrow \Pi x{:}A.K \nearrow \bar A \\ M \Leftarrow A \nearrow \bar M \quad [M/x]K \text{ is } K'\end{array}}{(P\ M) \Rightarrow K' \nearrow (\bar A\ \bar M)}\ \texttt{atpconvert/app}$$

$\boxed{K \nearrow \bar K}$

$$\frac{}{\mathsf{type} \nearrow \mathsf{type}}\ \texttt{kconvert/type}$$

$$\frac{A : \mathsf{type} \nearrow \bar A \quad x{:}A \vdash K \nearrow \bar K}{\Pi x{:}A.K \nearrow \Pi x{:}\bar A.\bar K}\ \texttt{kconvert/pi}$$