Redundancy Elimination for LF

Jason Reed [1]

Carnegie Mellon University
Pittsburgh, Pennsylvania
jcreed@cs.cmu.edu

Abstract

We present a type system extending the dependent type theory LF, whose terms are more amenable to compact representation. This is achieved by carefully omitting certain subterms which are redundant in the sense that they can be recovered from the types of other subterms. This system is capable of omitting more redundant information than previous work in the same vein, because of its uniform treatment of higher-order and first-order terms. Moreover the 'recipe' for reconstruction of omitted information is encoded directly into annotations on the types in a signature. This brings to light connections between bidirectional (synthesis vs. checking) typing algorithms of the object language on the one hand, and the bidirectional flow of information in the ambient encoding language. The resulting system is a compromise seeking to retain both the effectiveness of full unification-based term reconstruction such as is found in implementation practice, and the logical simplicity of pure LF.

Key words: Proof Compression, Dependent Type Theory, Bidirectional Type Checking

1 Introduction

The use of logical frameworks in domains such as proof-carrying code [Nec97] makes the efficiency of proof representation and manipulation a nontrivial issue. Proofs of safety for realistic programs can be, if naïvely represented, unfeasibly large. Necula and Lee [NL98] developed one technique which addressed this issue. They give a way of representing proof terms in the logical framework LF [HHP93] in a more efficient way, by rewriting them with whole subtrees of the proofs erased. They then describe an algorithm which recovers these omitted parts, using typing information found in other parts of the proof.

[1] This work was supported by NSF Grant CCR 0306313 "Efficient Logical Frameworks".

Their experimental results are good: proofs so represented tend to have size roughly $O(\sqrt{n})$ of the originals, with similar improvements in checking time.

To get a flavor of how omission works, consider the following example of encoding a natural deduction proof theory in LF. We have a signature

$$o : \mathsf{type} \qquad pf : o \to \mathsf{type}$$

$$\supset : o \to o \to o \qquad \wedge : o \to o \to o$$

where o is declared as the type of propositions, pf is the type family of proofs, indexed by proposition, and $\supset$ and $\wedge$ are the familiar logical connectives. Take one of the two natural deduction elimination rules for $\wedge$:

$$\frac{A \wedge B}{A} \wedge E_1$$

In LF it becomes

$$ande1 : \Pi a{:}o.\Pi b{:}o.pf\ (\wedge\ a\ b) \to pf\ a$$

Consider a use of this proof rule, $ande1\ a\ b\ d$. Here d must be a derivation of $a \wedge b$, and this larger proof $ande1\ a\ b\ d$ is a proof of a. This is excessively verbose, in a sense: knowing what type d is supposed to have (that is, $pf(\wedge\ a\ b)$) reveals what a and b must be. We would like to just write $ande1\ d$. It is not at all obvious, however, that the object d itself uniquely determines its type. This is a central issue, and we return to it below.

Another sort of apparent redundancy appears if we examine the introduction rule for implication. The natural deduction rule is

$$\frac{\begin{array}{c}\overline{A}\\ \vdots\\ B\end{array}}{A \supset B} \supset I$$

The hypothetical derivation of B under the hypothesis A is represented by higher-order abstract syntax [PE98] as a function from $pf\ a$ to $pf\ b$, and the rule is encoded as

$$impi : \Pi a{:}o.\Pi b{:}o.(pf\ a \to pf\ b) \to pf\ (\supset\ a\ b)$$

Here we may notice that if we have a term $impi\ a\ b\ f$, and if we know it, as a whole, is being checked against a certain type, $pf\ (\supset\ a\ b)$ then we can read off what a and b have been. If we knew that the type is going to be provided 'by the environment' somehow, then we can simply write $impi\ f$ instead.

It is this sort of omission of arguments that LF_i obtains its savings from. However, the technique uses a notion of 'reconstruction recipes' external to the type system to control which arguments are omitted. This work aims to put the basic idea of Necula and Lee on firmer type-theoretical footing, explaining the mechanism of omission in the types themselves. We describe an extension of the LF language, called LF_*, such that the same sort of arguments to type families and constants can be omitted. Our priorities are, in order, (1) making sure that the extension is conservative, (2) making the theory logically well-motivated, (3) making sure that an eventual implementation is simple and easy to trust, and only finally (4) maximizing the number of subterms that can be omitted omitted.

It should be noted that this general idea of 'implicit' syntax is not new: It can be found in the earlier work of Hagiya and Toda [HT94] with LEGO, and Miquel [Miq01] and Luther [Lut01] with the Calculus of Constructions.

However, some approaches (such as [Miq01]) do not treat implicit terms as anything more than a user-interface convenience. Though the front-end reconstructs arguments omitted by the user, and erases them once again when terms are printed, the core of the implementation works with fully explicit terms. The meaning of the implicit calculus is in any event *defined* in terms of the explicit calculus: an implicit term is well-typed if it can be elaborated uniquely into an explicit term. Both [Lut01] and [HT94] agree that it seems "difficult to directly give a foundation to the implicit calculus." That is exactly the aim of this work.

The remainder of the paper is structured as follows. We first present the type theory of LF_*, followed by a description of a decision procedure for the judgments therein. The proof of correctness of this algorithm is sketched. We give a description of a translation from LF to LF_* and argue that it preserves typing and is bijective on terms, so that it witnesses the equivalence of the new language and the old.

2 Type Theory

The two critical questions left unanswered in the introductory example are *when does an object uniquely determine its type?* and *when do we already know, from the surrounding context, what type an object must have?* These are answered by organizing the language and type-checking algorithms of a system so as to support bidirectional type-checking.

The terms are divided into *normal* terms, which can be type-checked if a type is provided as input, *atomic* terms, which can be type-checked in such a way that uniquely determines (one says it *synthesizes*) a type as output if type-checking succeeds. Ordinarily in λ-calculi, we know that functions are normal, and application of a constant or variable to a list (or *spine*) S of arguments is atomic. That is, our grammar of terms looks something like

$$\text{terms } M ::= N \mid R$$
$$\text{normal } N ::= \lambda x.M$$

$$\text{atomic } R ::= x \cdot S \mid c \cdot S$$
$$\text{spines } S ::= () \mid (M; S)$$

Our reasoning about the example, however, suggests that we may want some constants c — such as *impi* from the example — to require that $c \cdot S$ receive a type as input before type-checking proceeds, so that some omitted arguments in S can be recovered. We divide, therefore, the constants into two halves, the *synthesizable* constants c^+ and the *checkable* constants c^-. Therefore we write $\mathit{ande}1^+$ instead of $\mathit{ande}1$, and impi^- instead of *impi*, for the latter will depend on the 'inherited' type information for reconstruction, where the former does not. In general a spine headed by a c^- constant is a normal term, rather than atomic.

Now we have a further problem, however. The fate of constants such as $\mathit{ande}1$ is in doubt, because they require certain of their arguments to be synthesizing. What if the proof we have in mind of $A \wedge B$ uses impi^- as its last step? There are two conflicting requirements: $\mathit{ande}1^+$ wants to get type information from impi^- to proceed with reconstruction, and vice versa.

We fill this gap by allowing type ascriptions to appear inside spines, so that when an argument does not provide its type, and the constant which it is an argument of requires it, the type can be simply written down in the term. We make a production rule for spine elements

$$E ::= M \mid M^+ \mid *$$

which says that an argument may either be an ordinary term, a term which is adequate when a synthesizing term is required, (see immediately below) or else a placeholder for an omitted argument. Spines are then given by

$$S ::= () \mid (E; S)$$

Terms are now

$$M ::= N \mid R$$
$$N ::= \lambda x.M \mid c^- \cdot S$$
$$R ::= x \cdot S \mid c^+ \cdot S$$

and the M^+ used above has the production

$$M^+ ::= (N : A) \mid (R :)$$

The new syntax $(R :)$ here seems peculiar: it would seem more natural to put simply R. For an atomic term is adequate when a synthesizing term is required, and so is a normal term with a type ascription. However, when we define substitution, it is necessary to know syntactically when we come across an atomic argument in a spine, whether it is in a position that actually *requires* a synthesizing term or not. The $(R :)$ signals that if substitution produces a normal term, then a type ascription must be introduced.

Now we turn to the language of types in LF_*. They are given by the grammar

$$\begin{aligned}\text{basic types } A, B &::= a \cdot S \mid \Pi^- x{:}A.B\\ \text{general types } Z &::= a \cdot S \mid \Pi^\rho x{:}A.Z\\ \text{omission modes } \mu &::= s \mid i\\ \text{polarities } \sigma &::= + \mid -\\ \Pi\text{-annotations } \rho &::= \sigma \mid [\mu]\end{aligned}$$

Expanding out the grammar, there are four dependent function types, each of which determines how its argument functions with regard to omission and reconstruction. The Π^- functions are just the ordinary dependent functions from LF. They receive a $-$ superscript to make them stand in contrast with Π^+, which require their argument to be synthesizing. When there are Π^+ arguments, earlier arguments may be omitted via making their functional dependency $\Pi^{[s]}$, which marks a function whose argument is omitted *by synthesis*. Finally, $\Pi^{[i]}$ indicates a function whose argument is omitted *by inheriting* it from the result type the function application is checked against. In this language, the types of the proof rules in the example are (writing $A \to B$ for $\Pi^- x{:}A.B$ when x doesn't appear in B)

$$ande1 : \Pi^{[s]} a{:}o.\Pi^{[s]} b{:}o.\Pi^+ pf\ (\wedge\ a\ b).pf\ a$$

$$impi : \Pi^{[i]} a{:}o.\Pi^{[i]} b{:}o.(pf\ a \to pf\ b) \to pf\ (\supset\ a\ b)$$

Note that there is a distinction between the A, B are 'basic' types, which variables in a context may have, and Z which are the more general types that c^- constants can have. It would be more felicitously uniform if we could have simply one notion of type which constants and variables shared, but so far we have not been able to overcome the technical difficulties that arise when function variables are allowed to omit some of their arguments.

We elide for space reasons the grammar for kinds, and often refrain from mentioning the cases for kinds in the results below. Extending the definitions and results to that level is easy and uninteresting. Sometimes it is useful to write W, V as a 'wildcard' standing for a term or type or kind, for a briefer treatment of judgments and statements that are relevant for all three levels.

2.1 Substitutions

We elect a style of presentation which follows that of the concurrent logical framework CLF [WCPW03], in that we keep all terms in canonical form, that is, β-normal η-long form. This saves us from the complexity of dealing directly with $\beta\eta$-convertibility and the ensuing complex logical relations proofs of decidability of equality (for an example, see [HP01]) This complexity doesn't wholly disappear, though it reappears in a more tractable form: it is delegated to the definition of substitution. Substitution of a normal term in for a variable may create a redex, and the definition of substitution must carry out the reduction to ensure that the result is still canonical. To show that this process terminates we must pay attention to the decrease in the size of types of redices, logically parallel to the induction in structural cut elimination [Pfe00]. For this reason, CLF indexes the substitution operators with the type at which they operate. In fact, to show just termination of the substitution algorithm, only the skeleton of the type is required, but for our purposes, we need the full type for an independent reason.

Namely, it is possible that a variable-headed term, say, $x \cdot ()$ appears in a spine in a position which needs to be synthesizing. As the matter stands, this is perfectly acceptable, for variables applied to spines are synthesizing. However, we may substitute a term for x, say $c^- \cdot ()$, that produces a result which no longer synthesizes. Therefore, before we set out on the substitution, we must specify what type the substituted object has, so that we can create a type ascription to ensure that the result synthesizes.

We define, therefore, partial operations $[M/x]^A M'$, $[M/x]^A A'$, $[M/x]^A S$, substitution of M for x in M', A', S, respectively, at the type A. Since substituting for a variable in a synthesizing term may require wrapping it in a type, we have have a σ-indexed partial operations $[M/x]^A_\sigma R$. When σ is plus it outputs an M^+, and when it's $-$, an M. The operation $[M \cdot S]^A_\sigma$ resolves the redex $M \cdot S$, for M at type A, and similarly produces an M^+ or M according to σ.

To see that this definition is well-founded, one can analyze the *simple type* of the type in the superscript, that is, the result of erasing all dependencies and changing every Π to a mere $\rightarrow$.

The term $subj(M^+)$ is defined by $subj(R :) = R$ and $subj(N : A) = N$. We write $[M^+/x]^A$ to mean $[subj(M^+)/x]^A$.

$$[M \cdot ()]^{a \cdot S}_{-} = M$$
$$[R \cdot ()]^{a \cdot S}_{+} = (R :)$$
$$[N \cdot ()]^{a \cdot S}_{+} = (N : a \cdot S)$$
$$[\lambda x.M \cdot (M'; S)]^{\Pi^- x:A.B}_{\sigma} = [[M'/x]^A M \cdot S]^{[M/x]^A B}_{\sigma}$$

$$[M/x]^A_\sigma x \cdot S = [M \cdot [M/x]^A S]^A_\sigma$$
$$[M/x]^A_\sigma y \cdot S = y \cdot [M/x]^A S$$
$$[M/x]^A_\sigma c^+ \cdot S = c^+ \cdot [M/x]^A S$$

$$[M/x]^A c^- \cdot S = c^- \cdot [M/x]^A S$$
$$[M/x]^A \lambda y.M = \lambda y.[M/x]^A M$$
$$[M/x]^A R = [M/x]^A_- R$$

$$[M/x]^A \mathsf{type} = \mathsf{type}$$

$$[M/x]^A a \cdot S = a \cdot [M/x]^A S$$
$$[M/x]^A (\Pi^\rho y{:}B.Z) = \Pi^\rho y{:}[M/x]^A B.[M/x]^A Z$$

$$[M/x]^A () = ()$$
$$[M/x]^A (E; S) = ([M/x]^A E; [M/x]^A S)$$

$$[M/x]^A (R :) = [M/x]^A_+ R$$
$$[M/x]^A (N : B) = ([M/x]^A N : [M/x]^A B)$$

$$[M/x]^A * = *$$

2.2 Strictness

We have still so far neglected to pin down formally what it means for, say, one argument to have a sufficiently good occurrence in another argument to allow the former to be omitted. We can see that clearly a has an occurrence in $pf\ (\wedge\ a\ b)$ in such a way that we can 'read it off,' but the general higher-order case can be more complicated. The variable simply appearing in the syntax tree of the type is not enough, for the process of substituting in other arguments may cause β-reductions which make that appearance vanish. We therefore need to define *strict occurrences*, so that an argument which *strictly* occurs in the type of a synthesizing argument, or in the result type of a c^- constant, may be safely omitted.

The definition of strict occurrences that follows closely follows the definition of Pfenning and Schürmann [PS98] used to describe the theory of notational definitions. The notion of *pattern spine* at the heart of it is originally due to Miller [Mil91]. The guiding idea is that a strict position cannot be eliminated by other substitutions, and that, as a result, the operation of substituting $[M/x]N$ is injective in the argument M when x is strict in N. This injectivity means that we can uniquely recover M from $[M/x]N$. That is, the important consequence is that the corresponding matching problem is decidable and has a unique closed solution.

A key limitation of the way strictness is defined here, from the standpoint that more strict occurrences means more opportunities to omit redundant information, is that x cannot generally have a strict occurrence in $(*, S)$, even if it does have a strict occurrence in S. This is because we actually need more than just the term being uniquely determined when it is substituted for a strictly occurring variable: for technical reasons in the unification algorithm, we need its type to be uniquely determined as well.

The strictness judgments are $\Gamma \Vdash^s x \in Z$, (x has a strict occurrence in some argument of the type Z) $\Gamma \Vdash^i x \in Z$, (x has a strict occurrence in the output of the type Z) $\Gamma; \Delta \Vdash x \in W$, ($x$ has a strict occurrence in W in the presence of local bound variables Δ) and $\Delta \vdash S\,\mathsf{pat}$. ($S$ is a pattern spine, that is, a sequence of distinct bound variables)

2.2.1 Top-level

$$\frac{\Gamma; \cdot \Vdash x \in S}{\Gamma \Vdash^{i} x \in a \cdot S} \qquad \frac{\Gamma; \cdot \Vdash x \in A}{\Gamma \Vdash^{s} x \in \Pi^{+} y{:}A.B}$$

$$\frac{\Gamma, y : A \Vdash^{\mu} x \in B}{\Gamma \Vdash^{\mu} x \in \Pi^{\rho} y{:}A.B}$$

2.2.2 Types

$$\frac{\Gamma; \Delta \Vdash x \in S}{\Gamma; \Delta \Vdash x \in a \cdot S}$$

$$\frac{\Gamma; \Delta, y \Vdash x \in B}{\Gamma; \Delta \Vdash x \in \Pi^{\rho} y{:}A.B} \qquad \frac{\Gamma; \Delta \Vdash x \in A}{\Gamma; \Delta \Vdash x \in \Pi^{\rho} y{:}A.B}$$

2.2.3 Spines

$$\frac{\Gamma; \Delta \Vdash x \in M}{\Gamma; \Delta \Vdash x \in (M; S)} \qquad \frac{\Gamma; \Delta \Vdash x \in S}{\Gamma; \Delta \Vdash x \in (M; S)}$$

$$\frac{\Gamma; \Delta \Vdash x \in S}{\Gamma; \Delta \Vdash x \in (M^{+}; S)} \qquad \frac{\Gamma; \Delta \Vdash x \in R}{\Gamma; \Delta \Vdash x \in ((R :); S)}$$

$$\frac{\Gamma; \Delta \Vdash x \in N}{\Gamma; \Delta \Vdash x \in ((N : A); S)} \qquad \frac{\Gamma; \Delta \Vdash x \in A}{\Gamma; \Delta \Vdash x \in ((N : A); S)}$$

2.2.4 Pattern Spines

Since all terms are in η-long form, define $x \rightarrow_{\bar{\eta}}^{*} H$ ("H is an η-expansion of the variable x") by

$$\frac{y_1 \rightarrow_{\bar{\eta}}^{*} H_1 \qquad \cdots \qquad y_n \rightarrow_{\bar{\eta}}^{*} H_n}{x \rightarrow_{\bar{\eta}}^{*} \lambda y_1 \ldots \lambda y_n . x \cdot (H_1; \cdots; H_n)}$$

Then the definition of pattern spine is

$$\frac{}{\Delta \vdash () \,\mathsf{pat}} \qquad \frac{x \rightarrow_{\bar{\eta}}^{*} H \qquad \Delta_1, \Delta_2 \vdash S \,\mathsf{pat}}{\Delta_1, x, \Delta_2 \vdash (H; S) \,\mathsf{pat}}$$

2.2.5 Terms

$$\frac{\Gamma; \Delta, y \Vdash x \in M}{\Gamma; \Delta \Vdash x \in \lambda y.M} \qquad \frac{\Delta \vdash S \,\mathsf{pat}}{\Gamma; \Delta \Vdash x \in x \cdot S}$$

$$\frac{y \in \Delta \qquad \Gamma; \Delta \Vdash x \in S}{\Gamma; \Delta \Vdash x \in y \cdot S} \qquad \frac{\Gamma; \Delta \Vdash x \in S}{\Gamma; \Delta \Vdash x \in c^{\sigma} \cdot S}$$

2.3 Type Checking

We define over the language of LF_* two typing judgments $\Gamma \vdash_{def} M : A$ and $\Gamma \vdash_{alg} M : A$, with analogous judgments at the type and kind levels. The former is definitionally simpler, and consequently far easier to reason about, but nonalgorithmic. The latter, however, is transparently decidable, and can be implemented directly.

Establishing correctness of the system as a whole now has two parts. The first part is to show that the algorithm embodied by $\Gamma \vdash_{alg} M : A$ is sound and complete relative to $\Gamma \vdash_{def} M : A$. After that we must still connect $\Gamma \vdash_{def} M : A$ over LF_* to the same typing judgment over the original language of LF, which we construe as a syntactic subset of LF_*.

In a diagram, the task ahead looks like

$$\begin{tikzcd} LF/\vdash_{def} \arrow[r, "(\text{—})^*"] & LF_*/\vdash_{def} \arrow[r, equal] & LF_*/\vdash_{alg} \end{tikzcd}$$

Where $(\text{—})^*$ is a bijective translation from LF to LF_*.

We first give the rules that $\Gamma \vdash_{def} M : A$, $\Gamma \vdash_{alg} M : A$ have in common. This consists of all of the objects in the theory except for spines. Think of each rule with $\vdash$ as implicitly quantified by 'for all $\vdash \in \{\vdash_{def}, \vdash_{alg}\}$, ...'.

When we come to assigning types to spines there are two directions which a spine can be checked. The more familiar one is $\Gamma \vdash S : Z > C$, where the type Z and the spine S are given, and the type C is output. This is read as meaning that if a head (i.e. variable or constant) of type Z is applied S, the result will be of type C. However, we have introduced constants that require the output type to be known, so we also require a judgment $\Gamma \vdash S : Z < C$ which is identical in meaning to the other judgment, except that the type C is input rather than output.

2.3.1 Kinding

$$\frac{a : K \in \Sigma \qquad \Gamma \vdash S : K > \mathsf{type}}{\Gamma \vdash a \cdot S : \mathsf{type}}$$

$$\frac{\Gamma \vdash A : \mathsf{type} \qquad \Gamma, x : A \vdash B : \mathsf{type}}{\Gamma \vdash \Pi^{\sigma} x{:}A.B : \mathsf{type}}$$

Notice here that Π^{μ} types are well-kinded only in the event that the variable they bind actually has a strict occurrence. This is a key property when proving soundness of the system.

$$\frac{\Gamma \vdash A : \mathsf{type} \quad \Gamma, x : A \vdash B : \mathsf{type} \quad \Gamma, x : A \Vdash^{\mu} x \in B}{\Gamma \vdash \Pi^{[\mu]} x{:}A.B : \mathsf{type}}$$

2.4 Typing

$$\frac{\Gamma \vdash A : \mathsf{type}}{\Gamma \vdash (N : A) : A} \qquad \frac{\Gamma \vdash R : A}{\Gamma \vdash (R :) : A}$$

$$\frac{x : A \in \Gamma \qquad \Gamma \vdash S : A > C}{\Gamma \vdash x \cdot S : C}$$

$$\frac{\Gamma, x : A \vdash M : B}{\Gamma \vdash \lambda x.M : \Pi^{-}x{:}A.B}$$

$$\frac{c^{+} : Z \in \Sigma \qquad \Gamma \vdash S : Z > C}{\Gamma \vdash c^{+} \cdot S : C}$$

$$\frac{c^{-} : Z \in \Sigma \qquad \Gamma \vdash S : Z < C}{\Gamma \vdash c^{-} \cdot S : C}$$

2.5 Spines: Definitional Typing

The definitional typing system $\vdash_{def}$ uses the following rules to typecheck spines. So that we can write down rules only once that work the same way for both $>$ and $<$, say $\gtrless^{s}$ means $>$ and $\gtrless^{i}$ means $<$. Recall that μ,μ' are variables standing for either s or i.

$$\frac{}{\Gamma \vdash_{def} () : \mathsf{type} \gtrless^{\mu'} \mathsf{type}}$$

$$\frac{}{\Gamma \vdash_{def} () : a \cdot S \gtrless^{\mu'} a \cdot S}$$

$$\frac{\Gamma \vdash_{def} M : A \qquad \Gamma \vdash_{def} S : [M/x]^{A} V \gtrless^{\mu'} W}{\Gamma \vdash_{def} (M; S) : \Pi^{-}x{:}A.V \gtrless^{\mu'} W}$$

$$\frac{\begin{array}{l} A = A' \\ \Gamma \vdash_{def} M^{+} : A' \qquad \Gamma \vdash_{def} S : [M^{+}/x]^{A} V \gtrless^{\mu'} W \end{array}}{\Gamma \vdash_{def} (M^{+}; S) : \Pi^{+}x{:}A.V \gtrless^{\mu'} W}$$

$$\frac{\Gamma \vdash_{def} M : A \qquad \Gamma \vdash_{def} S : [M/x]^{A} V \gtrless^{\mu'} W}{\Gamma \vdash_{def} (*, S) : \Pi^{[\mu]}x{:}A.V \gtrless^{\mu'} W}$$

These rules as a system are impractical for an implementation because of the final rule. If read bottom-up, it requires the omitted argument M of a spine to be nondeterministically guessed.

2.6 Algorithmic Typing

The algorithmic type checking judgment does higher-order matching (that is, unification where all of the right-hand sides of equations have no free variables) to recover missing arguments.

2.6.1 Matching

We use P to denote sets of equations:

$$P ::= \top \mid (E_1 \doteq E_2) \wedge P \mid (S_1 \doteq S_2) \wedge P \mid (A_1 \doteq A_2) \wedge P$$

Q for sets of typing constraints:

$$Q ::= \top \mid (M : A) \wedge Q$$

and U for unification problems that track two sets of equality constraints, and one set of typing constraints:

$$U ::= \exists \Psi.(P, P', Q)$$

where Ψ denotes a list of variables

$$\Psi ::= \cdot \mid \Psi, x : A$$

It will also be necessary to talk about lists θ of substitutions:

$$\theta ::= \cdot \mid [M/x]^A \theta$$

There are several technical details about such substitutions θ that must be treated (not least of which, typing them) but for space reasons we do not cover them here.

The idea at a high level is that to solve a unification problem

$$\exists x_1{:}A_1, \ldots, x_n{:}A_n.(P, P', Q)$$

is to find a set of instantiations for $x_1, \ldots, x_n$ that make P, P', Q all true. Given that every x_i has a strict occurrence in P, which is maintained as an invariant of the algorithm, we can decompose equations in P while preserving any solutions that might exist, either instantiating variables, or postponing equations by transferring them to P', until P is empty, and all that remains is P' and Q. Since P is empty, our invariant says that no variables remain, so both P' and Q are closed, and can be checked directly. The only potential difficulty is the fact that we recursively call the typechecker on Q. But by inspection, the algorithm only puts strictly smaller type-checking problems into Q.

We define a transition relation $\Longrightarrow_\theta$ 'takes one step, resulting in substitution θ' via the following rules. The basic rules for working on a set of equations are quite straightforward, and all result in the empty substitution.

$$(a \cdot S_1 \doteq a \cdot S_2) \wedge P \Longrightarrow (S_1 \doteq S_2) \wedge P$$
$$(\Pi^\rho x{:}A_1.B_1 \doteq \Pi^\rho x{:}A_2.B_2) \wedge P \Longrightarrow$$
$$(A_1 \doteq A_2) \wedge (B_1 \doteq B_2) \wedge P$$
$$(\lambda x.M_1 \doteq \lambda x.M_2) \wedge P \Longrightarrow (M_1 \doteq M_2) \wedge P$$
$$(x \cdot S_1 \doteq x \cdot S_2) \wedge P \Longrightarrow (S_1 \doteq S_2) \wedge P$$
$$(c^\sigma \cdot S_1 \doteq c^\sigma \cdot S_2) \wedge P \Longrightarrow (S_1 \doteq S_2) \wedge P$$
$$(() \doteq ()) \wedge P \Longrightarrow P$$
$$((E_2; S_1) \doteq (E_2; S_2)) \wedge P \Longrightarrow (E_1 \doteq E_2) \wedge (S_1 \doteq S_2) \wedge P$$
$$(* \doteq *) \wedge P \Longrightarrow P$$
$$(R_1 :) \doteq (R_2 :) \wedge P \Longrightarrow (R_1 \doteq R_2) \wedge P$$
$$(N_1 : A_1) \doteq (N_2 : A_2) \wedge P \Longrightarrow (N_1 \doteq N_2) \wedge (A_1 \doteq A_2) \wedge P$$

These are used via

$$\frac{P \Longrightarrow P_0}{\exists\Psi'(P, P', Q) \Longrightarrow \exists\Psi'(P_0, P', Q)}$$

These less trivial rules handle the occurrence of variable on the left. Recall that we are doing matching, not full unification, so $\exists$-quantified variables do not occur on the right.

$$\exists\Psi, x : A, \Psi'.((x \cdot (H_1; \cdots; H_n) \doteq R) \wedge P, P'Q) \Longrightarrow_{[M/x]^A}$$
$$\exists\Psi, ([M/x]^A\Psi').[M/x]^A(P, P', Q)$$

(if $x_i \rightarrow^*_{\bar\eta} H_i$ where $x_1, \ldots, x_n$ are distinct variables not among those in Ψ, x, Ψ', Γ where $M = \lambda x_1 \ldots x_n.R$, if M has no free variables except those in Γ)

$$\exists\Psi, x : A, \Psi'.((x \cdot (H_1; \cdots; H_n) \doteq R) \wedge P, P'Q) \Longrightarrow$$
$$\exists\Psi, x : A, \Psi'.(P, (x \cdot (H_1; \cdots; H_n) \doteq R) \wedge P', Q)$$

(if the above rule doesn't apply)

Iterated $\Longrightarrow_\theta$ is the relation $\Longrightarrow^*_\theta$, defined by

$$\frac{}{U \Longrightarrow^*_\cdot U} \qquad \frac{U \Longrightarrow_\theta U' \qquad U' \Longrightarrow^*_{\theta'} U''}{U \Longrightarrow^*_{\theta'\theta} U''}$$

$\models$ is defined, like $\vdash$, uniformly over $\models_{def}$ and $\models_{alg}$ as follows:

$$\frac{}{\Gamma \models \top} \qquad \frac{\Gamma \vdash M : A \qquad \Gamma \models Q}{\Gamma \models (M : A) \wedge Q}$$

$$\frac{\Gamma \models P}{\Gamma \models (W \doteq W) \wedge P}$$

$$\frac{\Gamma \models P \qquad \Gamma \models P' \qquad \Gamma \models Q}{\Gamma \models (P, P', Q)}$$

Now we are able to give a definition of the core of the algorithm, the constraint generation judgment, which takes the form

$$\Gamma; \Psi; \Psi' \vdash S : Z \lessgtr^{\mu'} C/(P, Q)$$

This claims that if we are trying to apply a head of type Z to S, and the resulting type is C, then we must find instantiations for the variables in Ψ' to satisfy the equations P and type constraints Q. Γ, Ψ, S, Z are input to this judgment, and Ψ', P, Q are output. C is input if $\mu' = i$, and output if $\mu' = s$. The judgment is defined by the following rules.

$$\frac{}{\Gamma; \Psi; \cdot \vdash () : a \cdot S < a \cdot S' / (a \cdot S \doteq a \cdot S' \wedge \top, \top)}$$

$$\frac{}{\Gamma; \Psi; \cdot \vdash () : a \cdot S > a \cdot S / (\top, \top)}$$

$$\frac{\Gamma; \Psi, x : A; \Psi' \vdash S : Z \lessgtr^{\mu'} C/(P, Q)}{\Gamma; \Psi; x : A, \Psi' \vdash (*; S) : \Pi^{[\mu]} x{:}A.Z \lessgtr^{\mu'} C/(P, Q)}$$

$$\frac{\Gamma \vdash_{alg} M^+ : A' \qquad \Gamma; \Psi; \Psi' \vdash S : [M^+/x]^A Z \lessgtr^{\mu'} C/(P, Q)}{\Gamma; \Psi; \Psi' \vdash (M^+; S) : \Pi^+ x{:}A.Z \lessgtr^{\mu'} C/((A \doteq A') \wedge P, Q)}$$

$$\frac{\Gamma; \Psi; \Psi' \vdash S : [M/x]^A Z \lessgtr^{\mu'} C/(P, Q)}{\Gamma; \Psi; \Psi' \vdash (M; S) : \Pi^- x{:}A.Z \lessgtr^{\mu'} C/(P, (M : A) \wedge Q)}$$

Finally, the toplevel rules which tell how to algorithmically typecheck a spine are

$$\frac{\Gamma; \cdot; \Psi' \vdash S : Z < C/(P, Q) \qquad \exists \Psi'.(P, \top, Q) \Longrightarrow^*_{\theta'} (\top, P', Q') \qquad \Gamma \models_{alg} (P', Q')}{\Gamma \vdash_{alg} S : Z < C}$$

$$\frac{\Gamma; \cdot; \Psi' \vdash S : Z > C/(P, Q) \qquad \exists \Psi'.(P, \top, Q) \Longrightarrow^*_{\theta'} (\top, P', Q') \qquad \Gamma \models_{alg} (P', Q')}{\Gamma \vdash_{alg} S : Z > \theta' C}$$

When we have the type as input ($\Gamma \vdash_{alg} S : Z < C$) we invoke constraint generation to produce Ψ', P, Q, and call unification to check that the constraints are satisfied. If unification succeeds, then type-checking does. If we are to output a type ($\Gamma \vdash_{alg} S : Z < C$) then we furthermore use the substitution returned by unification, and apply it to the type C which constraint generation produced, and return this as the result type of S.

2.7 Correctness

The statements of soundness and completeness of unification are somewhat technical:

Lemma 2.1 (Soundness of Unification) *Suppose that*

$$\exists \Psi'.(P, P', Q) \Longrightarrow^*_{\theta_0} (\top, P'', Q')$$

and $\Gamma \models (\top, P'', Q)$. *Then there is a* θ' *such that* $\theta' = \theta_0$ *and* $\Gamma \vdash \theta' : \Psi'$ *and* $\Gamma \models \theta'(P, P', Q)$.

Lemma 2.2 (Completeness of Unification) *Suppose there exists* θ' *such that* $\Gamma \vdash \theta' : \Psi'$ *and* $\Gamma \models \theta'(P, P', Q)$. *Suppose further that for every* $x \in \Psi'$ *that there is an equation* $W \doteq W'$ *in* P *and a set* Δ_x *of variables disjoint from those declared in* Γ, Ψ' *such that* $\Gamma; \Delta_x \Vdash x \in W$. *Then there exist* P'', Q, θ_0 *such that* $\exists \Psi'.(P, P', Q) \Longrightarrow^*_{\theta_0} (\top, P'', Q')$ *and* $\theta' = \theta_0$ *and* $\Gamma \models (\top, P'', Q')$.

The main thrust of them, however, as is standard with such transition systems, is that (a) all of the individual transitions preserve solutions, and in our case, preserve strict occurrences as well, and (b) each transition decreases the size of the problem, so that solvability of a problem is decidable. The correctness of unification then leads to the correctness of the typing algorithm $\vdash_{alg}$ with respect to the definition $\vdash_{def}$.

Lemma 2.3 (Soundness and Completeness of $\vdash_{alg}$)

(i) *If* $\Gamma \vdash_{alg} M : A$, *then* $\Gamma \vdash_{def} M : A$.

(ii) *If* $\Gamma \vdash_{def} M : A$, *then* $\Gamma \vdash_{alg} M : A$.

3 Equivalence

Having defined LF_* and establishing that the definitional typing judgment is decidable, we turn now to the issue of showing that it is equivalent to LF. As mentioned previously, we construe the language of LF as a strict subset of the language of LF_*. Henceforth we syntactically distinguish every LF object with a $\circ$ in the subscript and every LF_* object with a $*$ subscript. The grammar of LF is

$$M_\circ ::= N_\circ \mid R_\circ$$
$$N_\circ ::= \lambda x.M_\circ$$
$$R_\circ ::= x \cdot S_\circ \mid c^+ \cdot S_\circ$$
$$E_\circ ::= M_\circ$$
$$S_\circ ::= () \mid (E_\circ; S_\circ)$$

$$A_\circ, B_\circ ::= a \cdot S_\circ \mid \Pi^- x{:}A_\circ.B_\circ$$
$$K_\circ ::= \mathsf{type} \mid \Pi^- x{:}A_\circ.K_\circ$$

This is simply the LF_* grammar with $c^-, \Pi^+, \Pi^{[\mu]}, (*; S), (M^+; S)$ removed. The typing judgments and rules that apply to this subset of LF_* are exactly the ordinary typing rules for LF. The only difference is cosmetic: here we say c^+, Π^- where one would of course find merely c, Π in a normal treatment of LF.

It remains to show that LF_* is isomorphic to LF, in the sense that every proof term in LF_* corresponds to one and only one proof term in LF. Fix for the sake of discussion signatures $\Sigma_\circ$ and Σ_*, in LF and LF_* respectively, and assume that they assign types and kinds to exactly the same constant and type family symbols, except that whenever Σ has c^+, we find exactly one of c^+ or c^- in Σ_*. Under suitable further assumptions (described below) that $\Sigma_\circ$ and Σ_* are in fact equivalent signatures, we aim to show that there is a translation from well-formed objects in $\Sigma_\circ$ to well-formed objects in Σ_* that is bijective, homomorphic with respect to typing, and so on.

One difficulty in establishing this result via such a translation comes from the fact that neither LF nor LF_* *prima facie* bears strictly more information than the other: LF_* signatures have more information in the form of Π-annotations, and its terms contain type ascriptions foreign to LF, while an LF term generally contains subterms that are omitted in its LF_* counterpart. Because of this, we cannot simply define an erasure function $W \mapsto W^*$ from LF to LF_* that erases some subterms to $*$. We need another erasure $W \mapsto W^\circ$ which erases Π-annotations, and we need $W \mapsto W^*$ to fill in necessary type ascriptions.

This notation is chosen to suggest that $(\text{—})^*$ takes objects into LF_*, and that $(\text{—})^\circ$ takes objects back to LF, though this latter statement is not strictly true. The general idea is that both mappings erase some information, and that objects $W_\circ$ and W_* ought to be considered equivalent when the mappings bring them together, when '$(W_\circ)^* = (W_*)^\circ$'.

The mapping $(\text{—})^\circ$ for Π-types is defined by $(\Pi^\rho x{:}A.W)^\circ = \Pi^- x{:}A.(W^\circ)$. Otherwise, $W^\circ = W$. However, the definition of $(\text{—})^*$ is less simple. Since it needs to insert type ascriptions, it cannot be merely a function from terms to terms, types to types, and so on. To know which type to insert, we must carry along the type, and in order to know the type of variables, we must carry along a context as well. We write this translation, then, using the same syntax as the typing judgment itself, as $(\Gamma_\circ \vdash M_\circ : A_\circ)^*$ for terms, and $(\Gamma_\circ \vdash A_\circ : \mathsf{type})^*$ for types.

For spines it is still not enough to write something of the form $(\Gamma_\circ \vdash S_\circ : Z_\circ > C_\circ)^*$. We need an additional argument Z_*, because its Π binders carry the required extra annotations required to translate the spine, (dictating, importantly, which arguments to erase) whereas $Z_\circ$ does not. Therefore the translation function for spines takes the form $(\Gamma_\circ \vdash S_\circ : Z_\circ > C_\circ)^*_{Z_*}$.

The translation is defined as follows:

Terms

$$(\Gamma_\circ \vdash \lambda x.M_\circ : \Pi^- x{:}A_\circ.B_\circ)^* =$$
$$\lambda x.(\Gamma_\circ, x : A_\circ \vdash M_\circ : B_\circ)^*$$
$$(\Gamma_\circ \vdash x \cdot S_\circ : C_\circ)^* = x \cdot (\Gamma_\circ \vdash S_\circ : A_\circ > C_\circ)^*_{(\Gamma_\circ \vdash A_\circ : \mathsf{type})^*}$$

(if $x : A_\circ \in \Gamma_\circ$)

$$(\Gamma_\circ \vdash c^+ \cdot S_\circ : C_\circ)^* = c^\sigma \cdot (\Gamma_\circ \vdash S_\circ : A_\circ > C_\circ)^*_{A_*}$$

(if $c^\sigma : A_* \in \Sigma_*$ and $c^+ : A_\circ \in \Sigma_\circ$)

Spines We mention only the case for typed (not kinded) spines. The other case is analogous. We split cases on the subscript Z_*. Make the abbreviations $A_* = (\Gamma_\circ \vdash A_\circ : \mathsf{type})^*$, and $S_* = (\Gamma_\circ \vdash S_\circ : [M_\circ/x]^{A_\circ} Z_\circ > C_\circ)^*_{[M_*/x]^{A_*} Z_*}$, and $M_* = (\Gamma_\circ \vdash M_\circ : A_\circ)^*$. Then for Π^σ we do

$$(\Gamma_\circ \vdash (M_\circ; S_\circ) : \Pi^- x{:}A_\circ.Z_\circ > C_\circ)^*_{\Pi^\sigma x:A_*.Z_*}$$

$$= \begin{cases} ((M_* : A_*); S_*) & \text{if } \sigma = +,\ M_* \text{ normal;} \\ ((M_*:); S_*) & \text{if } \sigma = +,\ M_* \text{ atomic;} \\ (M_*; S_*) & \text{otherwise.} \end{cases}$$

Observe that we only add the type annotation A_* when it is necessary. For $\Pi^{[\mu]}$ we simply erase the argument, and make the same recursive call on $S_\circ$ as before:

$$(\Gamma_\circ \vdash (M_\circ; S_\circ) : \Pi^- x{:}A_\circ.Z_\circ > C_\circ)^*_{\Pi^{[\mu]} x:A_*.Z_*} = (*; S_*)$$
$$(\Gamma_\circ \vdash () : C_\circ > C_\circ)^*_{C_*} = ()$$

Types

$$(\Gamma_\circ \vdash \Pi^- x{:}A_\circ.B_\circ : \mathsf{type})^* =$$
$$\Pi^- x{:}(\Gamma_\circ \vdash A_\circ : \mathsf{type})^*.(\Gamma_\circ, x : A_\circ \vdash B_\circ : \mathsf{type})^*$$
$$(\Gamma_\circ \vdash a \cdot S_\circ : \mathsf{type})^* = a \cdot (\Gamma_\circ \vdash S_\circ : K_\circ > \mathsf{type})^*_{K_*}$$

(if $a : K_* \in \Sigma_*$ and $a : K_\circ \in \Sigma_\circ$)

We may also translate contexts in the evident way, namely by translating each of the types in them. With these maps we can state the correspondence condition for the two signatures:

Definition 3.1 $\Sigma_\circ$ and Σ_* are *equivalent* if

- For every c, we have that $c^\sigma : A_* \in \Sigma_*$ and $c^+ : A_\circ \in \Sigma_\circ$ implies $(A_*)^\circ = (\cdot \vdash A_\circ : \mathsf{type})^*$.
- For every a, we have that $a : K_* \in \Sigma_*$ and $a : K_\circ \in \Sigma_\circ$ implies $(K_*)^\circ = (\cdot \vdash K_\circ : \mathsf{type})^*$.

When two signatures are equivalent, the theories they generate should be equivalent. This essentially amounts to two properties, that the image under the translation of the terms of a type actually belong to the translation of the type itself, and that the translation restricted to any one type is a bijection.

Theorem 3.2 (Type Preservation) *Suppose that* $\Sigma_\circ$ *and* Σ_* *are equivalent. Then*

- *if* $\Gamma_\circ \vdash_{\Sigma_\circ} M_\circ : A_\circ$*, then* $(\Gamma_\circ)^* \vdash_{\Sigma_*} (\Gamma_\circ \vdash M_\circ : A_\circ)^* : (\Gamma \vdash A_\circ : \mathsf{type})^*$
- *if* $\Gamma_\circ \vdash_{\Sigma_\circ} S_\circ : A_\circ > C_\circ$ *and* $(\Gamma_\circ \vdash A_\circ : \mathsf{type})^* = (A_*)^\circ$*, then* $(\Gamma_\circ)^* \vdash_{\Sigma_*} (\Gamma_\circ \vdash S_\circ : A_\circ > C_\circ)^*_{A_*} : A_* > (\Gamma \vdash C : \mathsf{type})^*$
- *If* $\Gamma_\circ$ *is a* $\Sigma_\circ$*-context, then* $(\Gamma_\circ)^*$ *is a* Σ_**-context.*
- *if* $\Gamma_\circ \vdash_{\Sigma_\circ} A_\circ : \mathsf{type}$ *then* $(\Gamma_\circ)^* \vdash_{\Sigma_*} (\Gamma_\circ \vdash A_\circ : \mathsf{type})^* : \mathsf{type}$.
- *if* $\Gamma_\circ \vdash_{\Sigma_\circ} K_\circ : \mathsf{kind}$ *then* $(\Gamma_\circ)^* \vdash_{\Sigma_*} (\Gamma_\circ \vdash K_\circ : \mathsf{kind})^* : \mathsf{kind}$.

Stating and proving the bijectivity of the translation, though important, is considerably more difficult, and so we do not develop it here.

4 Conclusion

We have described a type system which internalizes facts about which parts of terms can be safely omitted, while preserving representational adequacy. An implementation can achieve significant savings by not representing these omitted parts at all, and still ‘prove the same theorems’ as before.

The empirical advantage of this species of change of representation has been confirmed by earlier work. Ours retains several of its key properties. By working in a system derived from *LF*, we have at our disposal all of its representational techniques, such as higher-order abstract syntax. Like LF_i, full unification is not used, and instead only a subset — in our case, higher-order matching — is necessary. This is important for a maximally simple and trustable implementation.

The divergence from LF_i is that LF_* seeks to make type theoretic sense out of the possibility that subterms can be redundant. We do not have LF_i's ability to assign both an ‘inference recipe’ and a ‘checking recipe’ to a single constant, since we impose the restriction that a constant has a single type, which gives its reconstruction recipe once and for all. However, preliminary investigation suggests that in many cases — most especially when the object language is a type theory admitting a bidirectional typing algorithm itself — a constant is consistently always or almost always used in one way or the other. Thus, only one recipe is really necessary most of the time.

There is also a possible answer to this difficulty from using notational definitions. It is still an open problem whether notational definitions could feasibly be combined with this system, but if they could, then we could regain the ability to freely use different recipes by introducing a constant as being definitionally equal to an old one: one which, by virtue of being exposed at a new type, specifies a different reconstruction strategy for its arguments.

On the other side of the balance, there are forms of omission which LF_* can handle, which LF_i cannot. Since LF_* places a priority on pushing the mechanics of omission into the language itself at as fundamental a level as possible, the design of it is such that all terms, types, and kinds can contain placeholders for omitted information as a matter of course: the indices to a type family are general terms, and terms may contain placeholders. LF_i, on the other hand, has restrictions on when placeholders can appear in types. We anticipate, therefore, that encoding techniques that use more high-order and high-level constructions may benefit from the uniform treatment of omission afforded by LF_*. A more precise evaluation of the effectiveness of the proposed system still awaits implementation and experimentation, which we hope to complete soon.

Acknowledgements

Many thanks are due Frank Pfenning for his encouragement and help with both the conceptual and technical portions of this work, and to Kevin Watkins for the original formulation of the type system.

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.

[HP01] Robert Harper and Frank Pfenning. On the equivalence and canonical forms in the LF type theory. Technical report, Carnegie Mellon University, 2001.

[HT94] Masami Hagiya and Yozo Toda. On implicit arguments. In *Logic, Language and Computation*, pages 10–30, 1994.

[Lut01] Marko Luther. More on implicit syntax. In *Automated Reasoning. First International Joint Conference (IJCAR'01), Siena, Italy, June 18–23, 2001, Proceedings*, volume 2083 of *Lecture Notes in Artificial Intelligence*, pages 386–400, Berlin, 2001. Springer-Verlag.

[Mil91] Dale Miller. A logic programming language with lambda-abstraction, function variables, and simple unification. *Journal of Logic and Computation*, 1(4):497–536, 1991.

[Miq01] Alexandre Miquel. The implicit calculus of constructions: Extending pure type systems with an intersection type binder and subtyping. In S. Abramsky, editor, *Proc. of 5th Int. Conf. on Typed Lambda Calculi and Applications, TLCA'01, Krakow, Poland, 2–5 May 2001*, volume 2044, pages 344–359. Springer-Verlag, Berlin, 2001.

[Nec97] George C. Necula. Proof-carrying code. In *Proceedings of the 24th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Langauges (POPL '97)*, pages 106–119, Paris, January 1997.

[NL98] George C. Necula and Peter Lee. Efficient representation and validation of logical proofs. In *Proceedings of the 13th Annual Symposium on Logic in Computer Science (LICS'98)*, pages 93–104, Indianapolis, Indiana, 1998. IEEE Computer Society Press.

[PE98] 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 1998.

[Pfe00] Frank Pfenning. Structural cut elimination I. intuitionistic and classical logic. *Information and Computation*, 157(1/2):84–141, mar 2000.

[PS98] Frank Pfenning and Carsten Schürmann. Algorithms for equality and unification in the presence of notational definitions. In T. Altenkirch, W. Naraschewski, and B. Reus, editors, *Types for Proofs and Programs*, pages 179–193, Kloster Irsee, Germany, March 1998. Springer-Verlag LNCS 1657.

[WCPW03] K. Watkins, I. Cervesato, F. Pfenning, and D. Walker. A concurrent logical framework I: Judgments and properties. Technical report, Carnegie Mellon University, 2003.