Variable Arity for LF
Work in Progress

John Tang Boyland
ETH Zürich, Switzerland
University of Wisconsin-Milwaukee, USA
boylan@uwm.edu

Tian Zhao
University of Wisconsin-Milwaukee, USA
tzhao@uwm.edu

ABSTRACT

The dependently-typed lambda calculus (LF) supports use of meta-level binding in reasoning about bindings and hypotheticals in programming languages. That is, lambda expressions in LF can be used to model binders and hypothetical judgments depending on fixed-size contexts. However, because LF does not have a concept of variable-arity functions, a hypothetical judgment depending on a variable-size context cannot be modeled as an LF function. This paper extends LF to support variable-arity functions. As a result, one can model hypothetical judgments with variable contexts directly in the extended LF. The extended LF allows one to represent statements more transparently than previous work that uses complex meta-machinery to type the LF context. This is work in progress: we are still in the process of constructing a proof of correctness.

1. INTRODUCTION

Using LF as the metatheory for a proof system allows one to model name binding and hypotheticals as LF functions. For example, the classic substitution lemma has the following form in Pierce’s TAPL [8]

\[ \text{Lemma 1 (Substitution). If } \Gamma, \; x:S \vdash t : T \text{ and } \Gamma \vdash s : S, \text{ then } \Gamma \vdash t[x \mapsto s] : T \]

If we use the LF type families “exp” and “ty” for expressions and types respectively, and the LF type family “has-type” to represent both hypotheticals and also the typing judgment itself, and if we use the LF context itself to represent \( \Gamma \) (explained in more detail later), we end up with the following metatheorem (theorem about terms of LF):

\[ \text{Lemma 2. If we have LF terms of the following types:} \]
\[ T : ty, \; T' : ty, \; E : exp, \; F : \text{exp} \rightarrow \text{exp} \]
\[ t_1 : \Pi x : \text{exp}. \; \text{has-type} \; x \; T' \rightarrow \text{has-type} \; (F \; x) \; T' \]
\[ t_2 : \text{has-type} \; E \; T \]
\[ \text{then we can construct an LF term of type: has-type} \; (F \; E) \; T' . \]

The metatheorem can be expressed in the LF-based proof system Twelf [6] as

% theorem subst:
forall {T} {F} {T'} {E}
forall {T1: {x} has-type x T}
\rightarrow has-type (F x) T'
{F2: has-type E T}
exists {T3: has-type (F E) T'}
true.

As the reader may have noticed, the LF meta-theorem is trivial, since the desired result can be obtained by meta-level application: \( t_1 E t_2 \). The same is true of the Twelf formulation. This example shows the simplicity and power of LF-based higher order abstract syntax and in particular of using LF functions for hypotheticals. In general, one may not be able to use the same family for both hypotheticals and judgments, but instead has a separate type family (say “assm-type”) for hypotheticals. Then the substitution lemma requires an inductive proof which is straight-forward.\(^1\)

An important aspect is that the LF context is used for \( \Gamma \). In other words, the metatheorem itself operates in the scope of other bindings. Often a metatheorem is not true in all possible LF contexts (for example, one in which new “supernatural” numbers are positied) and thus the description of allowable LF contexts is part of the statement of a metatheorem. Twelf uses the concept of “regular worlds” [9] and Beluga [7] (another LF-based proof system) extends these with more powerful context descriptors. But the basic idea is shared between most (all?) LF-based proof systems.

When one cannot use the LF context to express a context of unknown size, the simple technique breaks down. Consider the following lemma for \( F_< \) from the POPLmark challenge [2]

% theorem subst:<
forall* {T} {F} {T'} {E}
exists {T1: has-type (F x) T'}
true.

Here, one cannot express the lemma in LF since \( \Delta \) may depend on \( x \). If \( \Delta \) is placed in the LF context, then any bindings in it that depend on \( x \) would be out of scope, a meta-level type error. Twelf’s POPLmark solution splits the assumption \( x:Q \) into two parts: the binding of the \( x \) which goes into the LF context whereas the subtyping assumption remains an explicit parameter. Basically, one could say that \( \Gamma, x:Q; \Delta \) is represented by \( \Gamma, x, \Delta, \; \text{assume} \; x:Q \) and then

\(^1\)Unless the type system (and \( \Gamma \)) permits dependent types, in which case, one must use more complex techniques, or indeed use the extension in this paper.
all but the last assumption are pushed into the LF context. Beluga and SASyLF [1] (an LF-based proof system geared to teaching) prove narrowing for the empty ∆ case using meta-level application (as with SUBSTITUTION above), thus avoiding the need to express or prove the narrowing lemma in full generality.

This paper proposes @LF, LF extended with variable-arity functions. Then the narrowing lemma can be represented as follows (for simplicity, we continue to use the LF context for Γ):

**Lemma 4.** If we have @LF terms of type

\[ P : ty, \quad Q : ty, \quad D : ty \rightarrow desc, \]
\[ M : \Pi X : ty.(ctx(D X)) \rightarrow ty, \]
\[ N : \Pi X : ty.(ctx(D X)) \rightarrow ty, \]
\[ t_1 : \Pi X : ty.\text{asm-sub X Q} \rightarrow \Pi \Sigma G : ctx(D X), \]
\[ \text{sub-type}(M X G)(N X G) \]
\[ t_2 : \text{sub-type}(P Q) \]

(here D represents ∆ with its dependency on the type variable being narrowed) then we can construct an @LF term of type

\[ \Pi X : ty.\text{asm-sub X P} \rightarrow \Pi \Sigma G : ctx(D X), \]
\[ \text{sub-type}(M X G)(N X G) \]

The extension includes variable-arity functions for which formal and actual parameters are marked with “@.” For example, the M term represents a type term but can depend on X (a normal parameter) and on ∆ (represented with variable parameter G of type “ctx(D X)”). Here D returns a context “descriptor”: a normal LF type whose values describe the shape of elements of an @LF “tuple type” family, here “ctx.” The “@” over the arrow on the type of M reminds us that this is a variable-arity function type.

The (LF) type family “desc” and the @LF type family “ctx” can be defined (the latter using equations to dependent tuple types U, written () or Σg.U):

- empty : desc
- newt : ΠD : desc. (ctx D @ ty) → desc
- ctx*empty = ()

The equations are used to replace known instances of @LF types families with dependent tuple types which are expanded in place. The syntax here is chosen for simplicity; in practice, a pattern-matching style would be preferable. To help understand how this works, consider the following examples. First suppose that D = λX : ty. empty; in other words, the context ∆ in the Lemma 3 is empty. In that case, because of the equation for ctx*empty, the type “ctx(D X)” is the empty tuple type which flattens to mean no parameters. In that case M and N have type ty → ty; they depend on X and the (implicit) LF context representing Γ but nothing else. Similarly the t1 will have the type:

\[ \Pi X : ty.\text{asm-sub X Q} \rightarrow \text{sub-type}(M X)(N X) \]

Next suppose instead that D ends up being “λX.newt empty X”, means that ∆ consists of a single (new) type variable Y assumed to be a subtype of X: ∆ = Y ∈: X. This correspondence is made using the “ctx*newt” equation (since “newt” is the head of “newt empty X”). We apply the expansion to the arguments and then after noting that “ctx*empty” expands to the empty tuple type, we end up with the following tuple type:

\[ ΣY : ty. Σb : (\text{asm-sub Y X}). () \]

When the explicit tuple here is substituted for the variable arity parameter, it is flattened out, Σ becomes Π, and we end up with the following as the type of M (and N):

\[ \Pi X : ty. \Pi Y : ty. \Pi b : (\text{asm-sub Y X}). ty \]

If ∆ is larger still, we will use a larger descriptor D. In the proof of Lemma 4, one would use induction on t1 and thus the fact that D gets bigger is not an issue.

Thus the main contribution of @LF is that it permits a context of unknown size to be typed using a descriptor in such a way that we can form types of judgments that depend on these contexts.

In general, since the descriptor is a normal LF type, using @LF, one can reason about contexts much more generally. One can define well-formedness conditions on contexts in the same way as for any other type. This ability would be welcome, since with more than seven years experience using Twelf), the first author finds it frustrating that one cannot express and use the fact that (say) all natural numbers used in a particular context are unique.

The remainder of this paper gives a precise definition of the extension, and then discusses current directions.

2. DEFINITIONS

Figure 1 gives the syntax of @LF including LF; the changes are highlighted. There are two kinds of bindings: normal bindings and variable bindings; the latter are typed by “b e,” a tuple type constructor b applied to an expression e, normally a variable. Similarly, we have two forms of kinds (K and @K) and types (T and @T). Variable application (in both types and expressions) is marked again with “@.” The
tuple types $U$ are only used in tuple constructor equations. The tuple lambda abstractions are used to accept the arguments from the constructor $c$ of the tuple constructor equation.

Figure 2 gives the type rules for $\@$LF. Following Harper and Licata [3], we only give the types of canonical forms. As in LF, abstractions are limited so that the head of an application is always a constructor, type family or variable. Since every formal or actual parameter can be of a tuple type, almost all the rules show changes, even if the only difference is the possible presence of "$\@$" noted with metavariable $\alpha$ in the figure, so that dependencies can be seen. Multiple occurrences of $\alpha$ must either all be $\emptyset$ or all vanish.

The first set of rules assign either kind or $\@$kind to each kind. Rule C-Abs has two independent ways ($\alpha$ and $\alpha'$) in which optional $\emptyset$’s can appear.

The next set of rules K-Xxx assign kinds to types. We have the familiar LF rules for type families, application and $\Pi$-types, where the last premise to K-Func checks subordination. The K-App rule uses substitution which refers to LF’s “hereditary substitution” [3] which is extended in $\@$LF. The new rule K-@Fam permits the type “$b \ e$” as the type for a variable-arity formal parameter only if $e$’s head be a variable (not a constructor). This restriction ensures that our canonical forms cannot include unexpanded types such as “ctx empty” in a way reminiscent of canonical LF not including any beta redexes in it.

We also have kinds for explicit tuple types $U$: K-@Emp, K-@Tuple and K-@Abs. The Kinding is qualified by $b$, the tuple type family being defined, allowing us to check subordination in K-@Tuple. The rule K-@Abs results in a non-atomic kind which means that it cannot be nested inside of instances of K-@Tuple.

So-called “atomic expressions” are typed using $\Pi$; the rules here are the same as for LF modulo variable-arity formals and actuals. An atomic expression with a non-function type in the signature, and as before, the last premise ensures uniqueness.

The final four rules handle signature elements with the normal LF signature elements being unchanged. The last premise of each rule ensures uniqueness in the signature.

Rule S-@Type checks a tuple type family. First we require that the descriptor is a legal type family without parameters. The next premise is short for saying that all constructors of the type must satisfy a “positivity” requirement (analogous to that used for inductive data types in Coq). In other words, no type that $\alpha$ is subordinate to may occur in negative position (i.e., as a function parameter type) in the type of a parameter to a constructor for $\alpha$. The type “newt” in our example is fine since “desc” is a (positive) parameter type and while “ctx $D$” is in negative position, it cannot expand to anything with a descriptor in it. Without this condition, we might have a constructor and associated tuple type equation:

$\text{bad : } \Pi f : (\Pi d : \text{desc}. \text{desc}). \text{desc}$

The expansion of “ctx (bad $\lambda x : \text{desc}. x$)” wouldn’t terminate. The third premise of S-@Type ensures that every constructor for the descriptor type has an associated tuple type equation, and as before, the last premise ensures uniqueness.

![Figure 2: Types for $\@$LF (extensions highlighted)](image-url)
In S-O@EQUATION, we check that b and c are defined in the signature and then abuse substitution notation to convert the type of c into the expected kind for the tuple type. The last premise, as always, ensures uniqueness.

Unfortunately, space precludes inclusion of the full extended definition of hereditary substitution. Suffice it that if we have a binding $\Omega: \beta e. B$ (for $\Omega \in \{\Pi, \Sigma, \Lambda\}$, and $B \in \{b, t, T, \#T, U, K, \#K\}$) and “be” expands to a tuple type $\Sigma_1 x_1 : t_1 T_1 \ldots \Sigma_n x_n : t_n T_n$, then the binding expands to $\Omega_1 x_1 : t_1 T_1 \ldots \Omega_2 x_2 : t_2 T_2 \ldots \Omega B[\Omega \rightarrow t_1 T_1 \ldots t_n T_n]$. The last substitution finds everywhere $\Omega$ is applied as an argument and substitutes in the new arguments. Variable-arity parameters can only be used in application which ensures we can get rid of all of them in this way.

3. APPLICATION

In the introduction, we motivated the need for an extension such as ours by showing how it enables a much more transparent expression of the narrowing lemma from POPLmark. But the extension is much more general. In particular, it gives a way to express any context using a descriptor; all the information about the shape of the context is present in the descriptor. This power permits us to model Twelf-style regular worlds, contexts more complicated than regular worlds, and to impose extra conditions on contexts as described in successive subsections. Finally we discuss how we envisage basing SASyLf on @LF.

3.1 Regular Worlds

In Twelf, a regular world is defined as a concatenation of any number of context fragments, each of which is described using one of a fixed set of “block” types. Each block is of the form:

\[ \%\text{block name : some bindings} \]

\[ \text{block bindings} \]

We can express the world with a new descriptor and a tuple type family (using a Twelf-style syntax in place of what we had earlier):

\[ \text{desc : type.} \]

\[ \text{world : desc }\rightarrow\text{ @type.} \]

We model this block with a constructor on the “some” bindings:

\[ \text{desc\text{/}name : bindings} \text{1 desc.} \]

Then we give a tuple type equation that takes the names from the first set of bindings and expands into a tuple type with the second set of bindings:

\[ \text{world (desc\text{/}name names} \text{1) = bindings} \text{2 ()}. \]

Here the () signifies that this is a tuple type, not a function type.

Using this construction, every world would need its own descriptor type and constructors, which is tedious. World subsumption type can be expressed in one direction by descriptor injection. In Twelf, the meta-machinery (e.g., coverage checker) “knows” about worlds and blocks and can perform world narrowing when subordination permits. In a system based on @LF, similar meta-machinery would be needed to avoid the need to prove narrowing separately for every type family (meta-theorem). We would expect to use context restrictions (See Sect. 3.3) in preference to a large number of worlds.

3.2 Other worlds

Twelf’s regular worlds are very restrictive; indeed not even powerful enough to describe all “regular worlds” in the usual sense: contexts describable by a regular expression over blocks. For instance, it’s not possible to declare a world in which the first part of the context uses one set of blocks, and the second part uses a distinct set of blocks. Such a world would be very useful for expressing LF in Twelf itself: the LF signature (declarations of type families and constructors) is not interleaved with the context for local bindings, but if we attempted to describe the whole context as a Twelf world, interleaving could not be ruled out. Some of these restrictions are lifted in Beluga, but we were not able to understand clearly what the expanded restrictions are.

3.3 Context Restrictions

Once the context is described by a normal LF type in the descriptor, we can now apply restrictions to it that would not fit in the regular world model. One may wish that express the fact that the elements of the context satisfy some non-regular well-formedness condition. For example, in the LF example (with a split context separating the signature from variable bindings), one may wish to express the fact that the signature is checked for type correctness, or that (say) every type family has at least one constructor.

In the past, the first author has wanted to be able to express concepts such as checking that every natural number bound to a variable is unique, or that all numbers in a specific range are covered. Neither of these concepts could be expressed in Twelf blocks. Surprisingly, it was still possible to prove things that required these restrictions, unfortunately at the cost of a large number of technical lemmas that worked around the shortcomings. Of course every system will have shortcomings, but the lack of the ability to restrict the context in Twelf is very frustrating.

3.4 (Not) Extending Twelf

Extending Twelf to support @LF would be a major undertaking, and despite the many references and examples that would suggest this approach, we do not intend this application, at least not in the near future.

We foresee difficulties especially with type reconstruction (type inference). Twelf infers the type of names, and permits variables to be implicitly bound. With @LF come many complications. First of all, would the @’s marking binders and applications be inferable as well? How does one handle inconsistencies where a variable is marked as a tuple type in one place but not others? How would one handle an omitted _ tuple argument? Depending on its value, a function may have different numbers of arguments. Would we expect type reconstruction to reconstruct the values of descriptor arguments?

Checking the proof of a meta-theorem includes coverage checking, which would need to be changed to handle @LF, in particular to handle the fact that one doesn’t know the number of arguments to a function from its type. The problems here don’t seem as difficult, however.

The extended Twelf would have both regular worlds and also the (new) context descriptor system. These address
similar problems with different techniques. Mixing the two (as we did in our motivating example) would be confusing. Best would be to integrate blocks with @LF so that (for instance) blocks could be seen as constructors of a descriptor type, as seen above. But working out such details would be further research.

Despite several proposals and at least one partial implementation, Twelf still has no module system that works with the meta-machinery. Since modules are less controversial than variable arity, it seems unlikely Twelf will address @LF.

### 3.5 Extending SASyLF

The @LF extension is being designed in order to support an extension of SASyLF. SASyLF aims to enable proofs to carried out close to the way they would be done in natural language, except being more explicit. In SASyLF contexts are declared using grammars in the same way as syntax. For example:

\[
\Gamma ::= \cdot \\
| \Gamma, x : T \\
| \Gamma, X <: T
\]

Currently such a declaration serves only for contexts, but we envisage that this declaration could at the same time serve as a declaration of the context descriptor. Then a judgment (such as typing) that uses \(\Gamma\) for a context indicates that it "assumes" \(\Gamma\) while one that uses it as a context descriptor would not. Then judgments about the context itself (as opposed to judgments that use the context) can be handled. For example:

```
judgment var-free: \(\Gamma\) var free

--------- empty-var-free
\cdot var free

\(\Gamma\) var free

--------------------------- type-var-free
\(\Gamma, X <: T\) var free
```

SASyLF provides an ideal platform for @LF technology since it already connects the descriptor and the context and be-operated on as syntax. With the former, we can use LF application for substitution and have built-in weakening, types are not "inductive" as explained (in more detail) by Schürmann [9]. But Coq already has tuples and many other data types available and so this work would be extraneous in any case.

In systems built on LF (Twelf, Beluga, Delphin and SASyLF), the lack of the ability to represent variable parameters means that other solutions are required.

The natural approach in Twelf is to use the LF context for metatheorems. When it works simply, this technique is very convenient. The mechanization of the rules in this paper uses Twelf and makes extensive use of regular worlds. However, when the technique of regular worlds does not work well, one must resort to contortions or revert to heavy-weight techniques. And in any case, the regular world technique can only describe the assumptions in the context, it cannot be used to change these assumptions as in the narrowing lemma.

Beluga represents LF context explicitly and it uses pattern matching to access the context passed to (recursive) functions that implement theorems. Similar to the Twelf’s regular worlds, Beluga places restrictions on the shape of the context using context schema. Also, the dependency among functions requires subsumption relations between the schemas of the function contexts. Beluga is able to decompose a closed term into a contextual variable and the substitution to its context. This enables the manipulation of open terms and the direct access to the binders in the context. However, the context in Beluga is still the LF context. It seems that the narrowing lemma shown earlier cannot be directly specified in Beluga since the context \(\Delta\) in \(\Gamma, X <: P, \Delta \vdash M <: N\), may be dependent on \(X\).

### 5. RELATED WORK

In systems with polymorphism (e.g. Coq [4]), HOAS [5] and the technique of representing hypothetical judgments by functions are less practical because functions are "too powerful" in such systems in that they can examine their arguments. There is also the problem that constructors such as \texttt{lam} use their type in a “negative” sense, which means their types are not "inductive" as explained (in more detail) by Schürmann [9]. But Coq already has tuples and many other data types available and so this work would be extraneous in any case.

Harper and Licata [3] prove that substitution terminates, commutes and preserves types. We are working on proving these properties for @LF but work is hampered because (unlike with LF) termination and commutativity must depend on typing. A counter-example for termination was shown above and, commutativity suffers from the fact that the composition of two substitutions may include tuple expansions that do not occur in either of the composite substitutions. Since type preservation requires termination and commutativity, it seems that all properties must be proved using mutual induction which is daunting. We are hoping to get some interaction and advice from attending the workshop and presenting our work in progress.

### 6. CONCLUSION

Our proposed extension @LF gives types for variable-arity functions, thus enabling the expression of metatheorems that refer to arbitrary-size contexts that depend on variables not in the LF context. Assuming we can prove correctness, this extension would enable LF-based proof assistants to express such metatheorems more naturally.

### 7. REFERENCES
APPENDIX

A. HEREDITARY SUBSTITUTION

Substitution for canonical LF avoids the creation of β-redexes by expanding them during the substitution process. As a result, substitution is more complex than standard substitution. In @LF, we further avoid the creation of situations where a tuple type family is applied to a constructed term. Figure 3 defines hereditary substitution for @LF. In the figure, B refers to any of the syntactic nonterminals in our syntax, and Ω refers to any of the binders (Π, Σ or λ).

A major aspect of the extension is that in parallel with the main substitution, we include a sequence ̂θ of expansions of variable-arity formals. Initially ̂θ is empty, but if we encounter a binding of a variable-arity argument whose type expands into a tuple type (see line 4), we expand the body with this argument being substituted by the sequence of the replacement arguments (see line 6).

Hereditary substitution distinguishes substitution in an atomic expression in which the result remains atomic (here with an “R” superscript) from substitution in which the head is the variable being substituted (here with a “RR” superscript). Secondary substitution occurs on lines 20 and 28 and is subscripted by t in case we have a variable-arity secondary substitution. The latter substitution is trivial (see line 25).

Hereditary substitution is not total, since we could fail to have a lambda expression at line 21. It is an important theorem of LF that all well-typed expressions have a substitution. We still need to establish this result for @LF.

When a variable-arity argument with a substitution is encountered in the “R” substitution (see line 15), we simply expand it in place. On the other hand, when doing the “RR” substitution (see line 22), we need to call out to a helper function to substitute the actual parameters into the lambda expression returned by the recursive call. This helper function can also be used to substitute in the constructor arguments when performing a “b+ic” reduction. Section B gives an example of substitution in operation.

B. FURTHER EXAMPLE

Figure 4 shows a simple example signature. First there are (standard) LF definitions of natural numbers and addition (“plus”). The type “plus m n o” is inhabited precisely when m + n = o where m is the natural associated with “m”, etc.

The second section defines sequences of natural numbers: “nats c” is the type of a sequence of c natural numbers (where c again is the value of “c”), as can be seen by nats+z (empty sequence) and nats+s (sequence of c naturals followed by one more).

Then “sum” takes a count c, a sequence of c naturals and a final natural; this type is inhabited precisely if the sum of the naturals in the sequence is equal to the final natural. For example, suppose the constructor sum/s is applied to the value z: the then perform the following hereditary substitution:

plus/z : Πm:nat. plus/∀n:n.
plus/s : Πm:nat. Πn:nat. Πo:nat. Πp:plus m n o.
plus (s (n (s (o)))).

nats : Πm:nat. @type.
  nats+z = ()
  nats+s = λc:nat. Σr: (nats c). Σi:nat. (),
  sum : Πc:nat. Π01: (nats c). Πm:nat. type.
  sum/z : sum z z.

Figure 4: Example: summing natural numbers.
we have a recursive call to substitute \(c \mid c \to z\) for which line 19 gives the result as simply “\(z\)”. Thus back at line 24 we need to look up \(\text{nats} + z\) in the signature yielding \(\emptyset\). Since there are no actual parameters, “app” simply returns \(\emptyset\) unchanged back to line 5. Next we must apply the expanded substitution to the body:

\[
\begin{align*}
\text{Ilm} &: \text{nat. In} : \text{nat. Io} : \text{nat. Ip} : \text{plus} \ m \ n \ o. \\
\text{Ilq} &: (\text{sum} \ (c \ 01 \ m). (s \ c) \ 01 \ n \ o) \\
& \quad [c \to z, 1 \to] \\
\end{align*}
\]

The first four abstractions include no uses of either argument and are passed through unchanged (using standard LF hereditary substitution). The interesting parts are the applications to the two types on the second line. First:

\[
\begin{align*}
\text{B} \ [x \to e, \emptyset] &= \text{B} = \alpha \text{type} \\
\alpha \text{type} &\quad \Omega \alpha x' : \alpha T'. B'[x \to e, \emptyset] = \alpha T' \\
\Omega \alpha x_1 : \alpha T_1 \ldots \Omega \alpha x_n : \alpha T_n. B' &= (\Omega \alpha x' : \alpha T'. B') \land \Omega \alpha x_1 : \alpha T_1 \ldots \Omega \alpha x_n : \alpha T_n. \emptyset \\
\emptyset &\quad B'[x \to e, \emptyset] = B'' \\
\emptyset &\quad B = \emptyset
\end{align*}
\]

Secondly, \(\emptyset\) is similarly substituted, and for the whole substitution, we have the following type:

\[
\begin{align*}
\text{Ilm} &: \text{nat. In} : \text{nat. Io} : \text{nat. Ip} : \text{plus} \ m \ n \ o. \\
\text{Ilq} &: \text{sum} \ z \ m. \text{sum} (s \ z) \ n \ o \\
\end{align*}
\]

Since the only term of type “\(\text{sum} \ z \ M\)” is \(\text{sum} /z\) and the latter requires the second term to be zero \(z\), this means that the only possibility for \(m\) is \(z\), and thus the only possibility for \(p\) is plus/\(z\) which means that \(o\) must be the same as \(n\), and thus our term can only have final type “\(\text{sum} \ z \ n\)”. In other words, the sum of the sequence of length one is exactly the only number in that sequence.

Likewise the type “\(\text{sum} \ (s \ z) \ n \ o\)” is only inhabited if \(m + n = o\), and so on. What makes these examples interesting is not the summation (which can be carried out in unextended LF on normal lists), but the fact that the type family \(\text{sum}\) takes multiple arguments.

**C. INFORMAL PROOF OUTLINE**

As mentioned in the main body of the paper, we have not
proved the correctness of our system, which, among other things require that we prove:

- totality of substitution for typed terms
- commutativity of substitution with typed terms
- preservation of types by substitution

And as also mentioned these three properties are mutually dependent:

- Totality of substitution requires that we have totality of the secondary substitution; it also requires that we have typed terms because we have non-well-typed counterexamples that fail to terminate. Unfortunately the secondary substitution is not on the input of the substitution but on the output of an earlier substitution which means we need preservation of types.

- Commutativity of substitution takes two substitutions on different variables and shows that they can be composed to a unique result whichever order is chosen. The problem is that one substitution could substitute the first variable with the second variable and the other could substitute the second variable with a constructed term. If the original term has a type “bx₁” then neither substitution on its own will cause this tuple type to expand, but the compositions will. As a result commutativity can only be proved for well-typed terms, and again secondary substitution will require us to have preservation of types after substitution.

- Finally, preservation of types after substitution needs commutativity of substitutions to handle cases such as T-App.

This mutual dependence may conceal a non-termination, and we would need to restrict our system in some way. If it doesn’t then we need to find a measure that is reduced over the mutual induction.

Setting aside this serious issue, the other main threats to termination of substitution are the secondary substitutions and, especially, tuple type expansion.

- Harper and Licata prove termination by showing that the type of the expression being substituted in is smaller in the secondary substitution. More precisely, termination uses the size of the erasure of the type, looking only at the function structure over type families, ignoring type parameters. We cannot use the same measure since type expansion can increase the size of a type, but with careful consideration of the maximum depth (number of switches between positive and negative positions) of an expanded type, we can use a similar argument, at least until we consider the expansion itself.

- A tuple type expansion can lead to further expansion, but termination here seems assured because the nested expansion must use constructed terms properly nested within the constructed term used for expansion. The expansion process itself cannot create any new constructed terms to be used for further expansion because of the positivity requirement ensuring that none of the available functions will be able to receive the constructed term.

Thus termination will either use a lexicographic termination condition, first on the depth of constructors of the type currently being expanded and then the depth of the function types of the variable being substituted, or perhaps a single measure that includes both. Subordination is important since secondary substitution can increase the constructor depth, but only for types “under” the current expansion level.

D. MECHANIZATION

We have mechanized the syntax, substitution and type system in Twelf including proofs of uniqueness of substitution among other more technical proofs. We have also mechanized two examples.