Explicit Contexts in LF (Revised)

Karl Crary
Carnegie Mellon University

Follow this and additional works at: http://repository.cmu.edu/compsci
Explicit Contexts in LF  
(Revised)
Karl Crary  
Carnegie Mellon University

Abstract
The standard methodology for representing deductive systems in LF identifies the object’s language’s context with the LF context. Consequently, any variable dealt with explicitly by any judgement or metatheorem must be last in the context. When the object language is dependently typed, this can pose a problem for establishing some metatheoretic results, since dependent hypotheses cannot be re-ordered at will.

This paper presents a general technique that addresses such problems, based on representing the object language’s context as an explicit object in LF while retaining the use of higher-order representation for the object language’s syntax. A central result is that it is possible to convert between explicit and implicit contexts, which makes it feasible to use the standard methodology for most developments, but use explicit contexts where necessary. We do not propose any extensions to LF; the technique can be utilized in standard LF.

1 Introduction
There are at least two different ways one may interpret the typing judgement \( x:A \vdash M : B \). One is as a hypothetical judgement built over a context-free (2-place) typing judgement. It states that if \( x \) is a term, and if one assumes that \( x \) has the type \( A \), then it follows that \( M \) has the type \( B \). Another is as a categorical judgement relating three syntactic objects: a context, a term, and a type. It states that, relative to the context \( x:A \), \( M \) has the type \( B \).

It is usually not difficult to prove, on paper at least, that these two interpretations are equivalent. However, they can look quite different when formalized in a logical framework. In particular, the LF logical framework emphasizes higher-order syntax and higher-order judgements, and therefore it lends itself to the hypothetical interpretation [4]. Although nothing in LF prevents one from employing a first-order encoding of syntax (including contexts) and then utilizing the categorical interpretation, to do so would sacrifice many of the strongest advantages of LF.

When formalizing meta-theorems in Twelf [11], the hypothetical interpretation occasionally causes difficulties. Specifically, in settings that include dependent types, theorems that involve a distinguished bound variable (such as substitution or functionality) cannot be proven directly by induction. In the next section we illustrate the difficulty that arises.

In this paper we give a technique that makes it possible to prove such theorems. It is based on a hybrid interpretation that is hypothetical in regard to the variables themselves, but categorical in regard to contexts that assign their types. That is, the judgement \( x:A \vdash M : B \) states that if (hypothetically) \( x \) is a variable, then (categorically) \( M \) has the type \( B \) relative to the context \( x:A \). Importantly, we can prove the equivalence of the hypothetical and hypothetical-categorical interpretations as a Twelf meta-theorem. Therefore, one can utilize the standard LF strategy as a matter of course, and resort to this paper’s technique only when necessary.

Concretely, we illustrate how to define in LF a type system that employs an explicit context, and how to prove some necessary properties (e.g., looking up a variable in a context returns a unique type). When using an explicit-context formulation, no problems arise when proving meta-theorems involving a distinguished bound variable. Finally, we show how to prove in Twelf that the explicit-context formulation is equivalent to the standard, implicit-context formulation. The latter result is the paper’s central technical contribution.

The remainder of this paper is structured as follows: In Section 2 we illustrate the problem that arises from distinguished bound variables and dependent types. In Section 3 we formalize explicit contexts in LF and give an example of their use. In Sections 4 and 5 we show how to convert derivations between implicit and explicit contexts. Throughout the paper we assume the reader is familiar with LF and with Twelf.

The Twelf code contained in this paper is available online at:

2 Motivation

2.1 An Illustrative Example
Consider the simply typed lambda calculus with a single base type \( o \) inhabited by a term \( b \). Its encoding in LF is given in Figure 1, and is standard except for the \%block declaration.

Twelf’s \%block declarations specify fragments that can be used to construct (implicit) LF contexts. The bind block provides an \( x:exp \) and \( d:of x a \), for some choice of \( a:tp \). (The names \( a \), \( x \) and \( d \) are bound, and are significant only

\(^{1}\)It will prove to be significant that we do not assume merely that \( x \) is a term.
Lemma 2.1 (Substitution)
If $\Gamma_1 \vdash M : A$ and $\Gamma_1, x:A, \Gamma_2 \vdash N : B$ then $\Gamma_1, \Gamma_2 \vdash N[M/x] : B$.

\[\begin{array}{rcl}
\text{tp} & : & \text{type}.\\
\text{exp} & : & \text{type}.\\
o & : & \text{tp}.\\
arow & : & \text{tp} \rightarrow \text{tp} \rightarrow \text{tp}.\\
b & : & \text{exp}.\\
lam & : & \text{tp} \rightarrow (\text{exp} \rightarrow \text{exp}) \rightarrow \text{exp}.\\
app & : & \text{exp} \rightarrow \text{exp} \rightarrow \text{exp}.\\
of & : & \text{exp} \rightarrow \text{tp} \rightarrow \text{type}.\\
of/b & : & \text{of} b o.\\
of/lam & : & \text{of} (\text{lam} A (\{x\} M x)) (\text{arrow} A B)\
& \leftarrow & \text{of} M (\text{arrow} A B)\
& \leftarrow & \text{of} N A.
\end{array}\]

%block bind
: some \{a:tp\}
block \{x:exp\} \{d:of x a\}.

Figure 1: Simple types in LF

\[\begin{array}{rcl}
\text{of/app} & : & \text{of} (\text{app} M N) B\
& \leftarrow & \text{of} M (\text{arrow} A B)\
& \leftarrow & \text{of} N A.
\end{array}\]

\[\begin{array}{rcl}
\text{subst} & : & \text{of} M A\
& \rightarrow & (\{x\} \text{of} x A \rightarrow \text{of} (N x) B)\
& \% & \rightarrow \text{of} (N M) B\
& \rightarrow & \text{type}.
\end{array}\]

\[\begin{array}{rcl}
\% \text{mode subst} & \times & \times X 2 - \times X 3.
\% \text{worlds (bind) (subst \_ \_ \_).}
\end{array}\]

It is important to note that this is an illustrative example, not a motivating one. Since substitution is provided primitively in LF by function application, this theorem has a trivial non-inductive proof. Nevertheless, we prefer this example due to its simplicity. We will briefly give some motivating examples in Section 2.2.

Note the treatment of the distinguished variable $x$, which is bound in the second input but free in $N x$.

A typical proof on paper would proceed by strengthening the theorem to allow additional assumptions after the assumption for $x$:

The key case of the proof is the typing rule for lambda, wherein the binding for the lambda-bound variable is shifted into the context $\Gamma_2$ before invoking induction on the body.

This proof strategy appears to be closed to us because in LF, the object-language context is absorbed into the LF context, and therefore it cannot be referenced explicitly. In this example, the outer context $\Gamma_1$ is absorbed into the surrounding LF context and so it is present implicitly in subst. (The worlds declaration gives an explicit indication that the surrounding context is permitted to contain $\Gamma_1$.) On the other hand, the inner context $\Gamma_2$ must appear after the explicit bound variable $x$, so there is nowhere to write it in subst.

When proving subst in Twelf, without the benefit of an inner context, we encounter difficulties in the lambda case. The usual way to resolve the difficulty is to permute variables, as shown in Figure 2.\(^3\)

In this proof case, $x$ is the substitution variable and $y$ is the lambda’s bound variable. The typing derivation for the body ($Dn x d y e$) depends on both $x$ and $y$ and their typing hypotheses. The proof proceeds by quantifying over $y$ and recursing, obtaining a typing derivation ($D y e$) for the body’s substitution instance $N M y$, which is used to reconstruct a typing derivation for the lambda.

This proof works because we are able to reverse the order in which $x$ and $y$ are bound. Initially, $y$ is within the scope of $x$. However, when we recurse, we move $y$ to the outside, while $x$ is still bound by the theorem itself.

Unfortunately, this strategy does not work in a dependently typed setting, where it is not generally possible to re-order variables in the context. Were the example dependently typed, the function’s domain would not be $B$ but $B x$, making it impossible to move $e$ (the typing assumption for $y$, whose type would then be $\text{of} y (B x)$) outside of the binding for $x$. The proof cannot be recovered.

2.2 Motivating Examples

The example above fails to be a motivating example because there exists a trivial proof of substitution. Of course, this

\(^3\)Since the names of theorem cases are insignificant, we save space by naming them all “\_\_\_”.

Within the declaration.) That is, it provides a variable binding, encoded in LF. Contexts constructed with the bind block may contain arbitrarily many instances of that fragment. Twelf’s %worlds declaration (an example appears below) specifies the possible contexts in which a metatheorem can be used, by listing the blocks from which an acceptable context can be constructed.

Now suppose that we wish to give an inductive proof of the substitution lemma, written in Twelf as:\(^2\)

\[\begin{array}{rcl}
\text{subst} & : & \text{of} M A\
& \rightarrow & (\{x\} \text{of} x A \rightarrow \text{of} (N x) B)\
& \% & \rightarrow \text{of} (N M) B\
& \rightarrow & \text{type}.
\end{array}\]

\[\begin{array}{rcl}
\% \text{mode subst} & \times & \times X 1 + \times X 2 - \times X 3.
\% \text{worlds (bind) (subst \_ \_ \_).}
\end{array}\]

Figure 2: Permuting subst proof (of/lam case)
is a fluke of the example; most interesting theorems require inductive proof.

In general, the problem arises for theorems with three properties:

1. The theorem involves a distinguished bound variable.
2. We require an inductive proof.
3. The type system is dependently typed (where types can
depend on the distinguished bound variable\(^4\)).

When working in a dependently typed setting, such theorems are not uncommon. A few examples are:

- **Substitution with different judgements on the left and right.** It is often necessary for typing assumptions to utilize a different judgement than the primary typing judgement. This most often happens because of a need to treat variables specially. For example, many module type theories ascribe special privileges to paths (where a path is defined as a series of actions, such as projection, acting on a variable) [5, 8, 9]. This might be represented in LF by:

  \[
  \begin{align*}
  \text{varof} & : \text{exp} \rightarrow \text{tp} \rightarrow \text{type}. \\
  \text{of} & : \text{exp} \rightarrow \text{tp} \rightarrow \text{type}. \\
  \text{path} & : \text{exp} \rightarrow \text{type}. \\
  \text{of/var} & : \text{of} X T \leftarrow \text{varof} X T. \\
  \text{path/var} & : \text{path} X \leftarrow \text{varof} X \_.
  \end{align*}
  \]

  In such a system, the substitution lemma:

  \[
  \begin{align*}
  \text{sub} & : \text{of} M A \\
  & \rightarrow (\{x\} \text{varof} X A \rightarrow \text{of} (N x) (B x)) \\
  & \%
  \rightarrow \text{of} (N M) (B M) \\
  & \rightarrow \text{type}. \\
  \%
  \text{mode} \text{sub} +X1 +X2 -X3. \\
  \%
  \text{worlds} \text{(bind)} (\text{sub} \_ \_ \_).
  \end{align*}
  \]

  cannot be proven trivially. (Above, we give the dependently typed formulation of substitution. Without dependent types it can be proven by permuting assumptions.)

- **Narrowing in Algorithmic \(F_\leq\).** A similar issue arises in the subtyping algorithm for \(F_\leq\), proposed as part of the Poplmark challenge [1]. In the algorithm, the reflexivity and transitivity rules are available only for variables:

  \[
  \Gamma \vdash X \leq X \quad \Gamma \vdash U \leq T
  \]

  This can be represented in LF by:

  \[
  \begin{align*}
  \text{atom} & : \text{type}. \\
  \text{term} & : \text{type}. \\
  \text{sub} & : (\text{atom} \rightarrow \text{term}) \\
  & \rightarrow \text{term} \rightarrow \text{term} \rightarrow \text{type}.
  \end{align*}
  \]

  \(^4\)For example, the problem does not arise in term substitution for the polymorphic lambda calculus, even though it is dependently typed, since types cannot depend on terms. However, it would arise for type substitution (if not for the trivial solution, of course).
3 Explicit Contexts

We will illustrate the explicit context method using a variant of the simply typed lambda calculus from Section 2.1. We will call this variant the simple dependently typed lambda calculus. It is obtained from the simply typed language by adding an additional type constructor \( p \) that depends on terms, and generalizing the arrow type to a dependent \( \pi \) type. The syntax and static semantics are given in Figure 3.

A useful application would also add some formation requirements and interesting structure to the \( p \) type, but we will not, since our interest here is in the technique, not the language itself.\(^6\) The method generalizes smoothly from the simple dependently typed lambda calculus to other languages of interest, such as LF.

Turning now to explicit contexts, the first important observation is that the encoding of syntax need not be changed at all. This is important because it means that explicit-context developments can co-exist with conventional implicit-context developments. Thus, the syntax of terms and types remains exactly that given in Figure 3.

3.1 Contexts

Of central interest in the method, of course, are contexts. A context is represented as a list of pairs associating types with term variables:

\[
\text{ctx : type.} \\
\text{nil : ctx.} \\
\text{cons : ctx -> exp -> tp -> ctx.}
\]

Thus, the context \( x : o, y : o \rightarrow o \) is represented:

\[
\text{cons (cons nil x o) y (arrow o o)}
\]

The intention is that the terms appearing within the context are always variables. However, nothing in the syntax enforces this. Instead, the task of enforcing that property is left to a context formation judgement. The context formation judgement checks another important property as well. We need to enforce the property that each variable appearing in the context is distinct. (This is important, for example, for establishing that looking up a variable in the context returns a unique type.)

These properties are tricky, because a priori we have no way in LF to say that a term is a variable, much less that two variables are distinct. We resolve both issues using the judgement \( \text{isvar} \). For every variable \( x \), we assume \( \text{isvar} x \), for some natural number\(^6\):\(^6\)

\[
\text{guage is dependently typed is whether exp is subordinate to tp [14]; that is, whether or not Twelf permits terms of type exp to appear within terms of type tp. Twelf infers the subordination relation from the signature, and either the p or pi declaration is sufficient to add the desired edge. The explicit context method would work similarly for any other formulation of dependent types that induced that edge.}

\(^6\)In fact, the determining factor for Twelf as to whether the lan-
precedes : exp -> exp -> type.

precedes/i : precedes X Y
<- isvar X I
<- isvar Y J
<- lt I J.

bounded : ctx -> exp -> type.

ordered : ctx -> type.

bounded/nil : bounded nil X
<- isvar X _.

bounded/cons : bounded (cons G Y _) X
<- precedes Y X
<- bounded G Y.

ordered/nil : ordered nil.

ordered/cons : ordered (cons G X _) <- bounded G X.

Figure 5: Context formation

isvar : exp -> nat -> type.

%block ovar
  : some {i:nat}
  block
  {x:exp}
  {d:isvar x i}.

The invariant that each variable has an isvar assumption is specified by the ovar block.

In the assumption isvar x I, we call I the order stamp for x. We use order stamps to impose a strict partial order on variables (x < y if the order stamp of x is less than that of y). We may then enforce that variables in a context are distinct by requiring them to be strictly increasing. (This is no limitation because we can choose the order stamps as desired.) Note that if x < y, it follows that x and y are variables, so the variables-only property follows directly from the strictly-increasing property.

These definitions are summarized in Figure 5. We consider a context G to be well-formed if ordered G. The auxiliary judgement bounded G x indicates that G is ordered and all its variables are strictly less than x.

Two other important judgements are lookup and append; their definitions are given in Figure 6.

3.2 Typing

To type a term relative to an explicit context, we use the ofe judgement:

ofe : ctx -> exp -> tp -> type.

The judgement ofe G M A is read “in context G, M has the type A.” (That is, G ⊢ M : A.)

If looking up the variable X in the context yields A then X has the type A:

ofe/var : ofe G X A
<- lookup G X A.

Figure 6: Lookup and append

In a lambda abstraction, we hypothetically assume a new variable x, then place it in the explicit context when checking the body:

ofe/lam
 : ofe G (lam A ([x] M x)) (pi A ([x] B x))
<- ({x} ofe (cons G x A) (M x) (B x)).

This rule expresses the essence of the hypothetical-categorical hybrid interpretation. We take x as a hypothetical variable — thereby allowing the use of higher-order abstract syntax — but treat x’s type assignment categorically: M x has type B in a context including x : A.

Note that the rule’s premise does not introduce an isvar assumption. Since isvar is not used in lookup or ofe derivations, there is no purpose to introducing such a dependency, and leaving it out eliminates the need to choose arbitrary order stamps within typing derivations. Instead, we maintain isvar assumptions in the proofs, for those proofs that depend on an ordered context.\(^7\)

The application rule is standard:

ofe/app : ofe G (app M N) (B N)
<- ofe G M (pi A ([x] B x))
<- ofe G N A.

Finally, we have one more rule for terms that are closed with respect to the explicit context:

ofe/closed : ofe G M A
<- of M A.

This rule states that if M has type A independently of the explicit context (that is, using only the implicit context) then M has type A in any context G.

It is convenient to use this rule for typing b, since it happens to be closed. More importantly, we use it for importing assumptions from the implicit-context setting into this explicit-context setting. This is essential because we wish to be able to shift into the explicit-context method at any point in a proof, not just when the implicit context is empty. Formally this is reflected in the worlds declaration for our lemmas allowing bind blocks as well as ovar blocks.

\(^7\)Many do not, it turns out.
3.3 Lemmas

We require several lemmas about the management of contexts:

- The order stamp of a variable is unique:

  \[
  \text{isvar-fun} : \text{isvar} \ X \ I \\
  \rightarrow \text{isvar} \ X \ J \\
  \rightarrow \text{nat-eq} \ I \ J \\
  \rightarrow \text{type}.
  \]

  \%mode isvar-fun +X1 +X2 -X3.

  \%worlds (ovar | bind) (isvar-fun _ _ _).

  As noted above, \text{isvar-fun} (and the lemmas that follow) works within a context consisting of both \text{ovar} and \text{bind} blocks, so the explicit-context method can be used within a larger implicit-context development.

  We can prove \text{isvar-fun} in a single line, by requiring the two inputs to unify and then simply returning the introduction rule for \text{nat-eq}:

  \[
  - : \text{isvar-fun} \ D \ D \ \text{nat-eq}/i.
  \]

  This works because within the world (ovar | bind) there is at most one \text{isvar} assumption for any given variable, so two \text{isvar} assumptions for the same variable must unify.

- As a corollary, \text{precedes} is a strict partial order:

  \[
  \text{precedes-irreflex} : \text{precedes} \ X \ X \\
  \rightarrow \text{false} \\
  \rightarrow \text{type}.
  \]

  \%mode precedes-irreflex +X1 -X2.

  \%worlds (ovar | bind) (precedes-irreflex _ _).

  \[
  \text{precedes-trans} : \text{precedes} \ X \ Y \\
  \rightarrow \text{precedes} \ Y \ Z \\
  \rightarrow \text{type}.
  \]

  \%mode precedes-trans +X1 +X2 -X3.

  \%worlds (ovar | bind) (precedes-trans _ _ _).

- A well-formed context can be extended. More precisely, for any well-formed context \(G\), there exists an order stamp \(I\) such that a fresh variable given that stamp will bound \(G\):

  \[
  \text{extend-context} \\
  : \text{ordered} \ G \\
  \rightarrow (\{x\} \text{isvar} \ x \ I \rightarrow \text{bounded} \ G \ x) \\
  \rightarrow \text{type}.
  \]

  \%mode extend-context +X1 -X2.

  \%worlds (ovar | bind) (extend-context _ _ _).

- If \(G\) is well-formed and has the form \(G_1, x : A, G_2\) and looking up \(x\) returns \(B \ x\), then \(A\) and \(B \ x\) are equal:

  \[
  \text{append-lookup-eq} \\
  : (\{x\} \text{append} \ (\text{cons} \ G_1 \ x \ A) \ (G_2 \ x) \ (G \ x)) \\
  \rightarrow (\{x\} \text{isvar} \ x \ I \rightarrow \text{ordered} \ (G \ x)) \\
  \rightarrow (\{x\} \text{lookup} \ (G \ x) \ x \ (B \ x))
  \]

  \%mode append-lookup-eq +X1 +X2 +X3 -X4.

  \%worlds (ovar | bind) (append-lookup-eq _ _ _ _).

- Lookup is preserved by the deletion of variables other than the one being looked up:

  \[
  \text{lookup-pdv} \\
  : (\{x\} \text{append} \ (\text{cons} \ G_1 \ x \ A) \ (G_2 \ x) \ (G \ x)) \\
  \rightarrow \text{append} \ G_1 \ (G_2 \ M) \ G' \\
  \rightarrow (\{x\} \text{lookup} \ (G \ x) \ Y \ (B \ x))
  \]

  \%mode lookup-pdv +X1 +X2 +X3 -X4.

  \%worlds (ovar | bind) (lookup-pdv _ _ _ _).

  Note that since Twelf meta-variables are implicitly quantified on the outside, \(Y\) cannot depend on \(x\), and therefore cannot be \(x\).

- If a term is well-typed under \(G_1\) then it is well-typed under any \(G\) that extends \(G_1\):

  \[
  \text{weaken-ofe} : \text{append} \ G_1 \ G_2 \ G \\
  \rightarrow \text{ofe} \ G_1 \ M \ A \\
  \rightarrow \text{ofe} \ G \ M \ A \\
  \rightarrow \text{type}.
  \]

  \%mode weaken-ofe +X1 +X2 +X3 -X4.

  \%worlds (ovar | bind) (weaken-ofe _ _ _ _).

3.4 Substitution Proof

With these lemmas in hand, we can prove the explicit-context substitution lemma:

\[
\text{esubst} : (\{x\} \text{append} \ (\text{cons} \ G_1 \ x \ A) \ (G_2 \ x) \ (G \ x)) \\
\rightarrow (\{x\} \text{isvar} \ x \ I \rightarrow \text{ordered} \ (G \ x)) \\
\rightarrow (\{x\} \text{lookup} \ (G \ x) \ x \ (B \ x))
\]

\%mode esubst +X1 +X2 +X3 +X4 +X5 -X6.

\%worlds (ovar | bind) (esubst _ _ _ _ _ _).

It reads: if \(G \ x = G_1, x : A, G_2 \ x\) and \(G' = G_1, \ (G_2 \ M)\) and \(G \ x\) is well-formed, and if \(G_1 \vdash M : A\) and \(G \ x \vdash N : B \ x\), then \(G' \vdash N : B \ M\). This is exactly the standard, on-paper formulation in Lemma 2.1 (generalized for dependent types).

In the proof, the ofe/closed case is immediate. The ofe/var case for variables other than \(x\) uses lookup-pdv to obtain a lookup from the context with \(x\) removed. The ofe/var case for \(x\) uses weaken-ofe to weaken ofe G1 M.
to \text{ofe} G' M A. The \text{ofe/app} case is a simple induction invocation.

The \text{ofe/lam} case is straightforward in the explicit-context setting. The LF binding for \( y \) is moved outside, but \( y \)'s typing assumption is now part of the \text{ofe} judgement, so it remains within the scope of \( x \). An isvar assumption is added, using an order stamp obtained from \text{extend-context}. The complete \text{ofe/lam} case is given in Figure 7.

This completes the explicit-context substitution proof, but recall that our ultimate aim is to prove the result for the original, implicit-context system. Thus, it remains to show that we can shift from implicit to explicit form and back.

4 Translation to Implicit Form

To convert from explicit form back to implicit form, we wish to prove the \text{ofe-to-of} lemma:

\[
\text{ofe-to-of}: \text{ofe} \ \text{nil} \ M \ A \\
\text{ofe-to-of}: \rightarrow \text{of} \ M \ A \\
\rightarrow \text{type}. \\
\text{mode ofe-to-of} +X_1 -X_2. \\
\text{worlds (var | bind) (ofe-to-of \_ \_)}. \\
\]

This states that if a typing judgement holds with an empty explicit context, it also holds with the context implicit. The worlds declaration shows that it can be used with \text{bind} blocks, that is, in the middle of a larger proof. It also can be used within explicit-context proofs, although in practice it would rarely be used that way. Since the \text{isvar} assumption used in \text{ovar} blocks cannot appear within \text{ofe} or \text{of} derivations, we can use the simpler \text{var} block without loss of generality:

\[
\text{block var} : \text{block} \ {x:exp}. \\
\]

The proof relies on a technical device, a judgement called \text{ofi}:

\[
ofi: \text{ctx} \rightarrow \text{exp} \rightarrow \text{tp} \rightarrow \text{type}. \\
\]

\text{mode ofi} +X_1 -X_2. \\
\text{worlds (var | bind | ofblock) (ofi \text{\_ \_)}. \\
\]

By definition, \text{of M A} follows immediately from \text{ofi nil M A}, so it remains to show that the latter follows from \text{ofe nil M A}:

\[
ofe-to-ofi: \text{ofe} \ G \ M \ A \\
\text{ofe-to-ofi}: \rightarrow \text{of} \ G \ M \ A \\
\rightarrow \text{type}. \\
\text{mode ofe-to-ofi} +X_1 -X_2. \\
\text{worlds (var | bind | ofblock) (ofe-to-ofi \_ \_)}. \\
\]

4 Translation to Implicit Form

To convert from explicit form back to implicit form, we wish to prove the \text{ofe-to-of} lemma:
The main lemma for proving

\[ \text{cut-ofe} \]

such as:

\[ \text{subst} \]

variable. (As in terms that are closed with respect to explicit variables:

\[ \text{convert back to implicit form (Section 4), thereby obtaining out a proof using explicit contexts (Section 3.4), and then implicit to explicit form. Once we have done so, we can carry The key result of this paper is that we can convert from 5 Translation to Explicit Form

The simplest version of translation to explicit form is for

This block appears nowhere else in the proof.

5 Translation to Explicit Form

The key result of this paper is that we can convert from implicit to explicit form. Once we have done so, we can carry out a proof using explicit contexts (Section 3.4), and then convert back to implicit form (Section 4), thereby obtaining a general result with no mention of explicit contexts. The simplest version of translation to explicit form is for terms that are closed with respect to explicit variables:

\[ \text{of-to-ofe} : \text{of} \ A \]

This is trivial to prove, using the \text{ofe/closed} rule. More often, however, there is at least one explicit free variable. (As in \text{subst}, for example.) Then we require a lemma such as:

\[ \text{of1-to-ofe} \]

\[ \text{block} \]

of x A

\[ \text{to} \]

\[ \text{of} \]

\[ \text{type} \]

\[ \text{mode cut-ofe +X1 +X2 -X3} \]

\[ \text{worlds} \]

of i-app +X1 +X2 -X3.

\[ \text{ofi-app} : \text{ofi} \ G \ M (\pi \ A (\[x\] B x)) \]

\[ \text{to} \]

\[ \text{type} \]

\[ \text{mode ofi-app +X1 +X2 -X3} \]

\[ \text{worlds} \]

of i-app +X1 +X2 -X3.

\[ \text{ofi-app} : \text{ofi} \ G \ M (\pi \ A (\[x\] B x)) \]

\[ \text{to} \]

\[ \text{type} \]

\[ \text{mode ofi-app +X1 +X2 -X3} \]

\[ \text{worlds} \]

of i-app +X1 +X2 -X3.

\[ \text{ofi-app} : \text{ofi} \ G \ M (\pi \ A (\[x\] B x)) \]

\[ \text{to} \]

\[ \text{type} \]

\[ \text{mode ofi-app +X1 +X2 -X3} \]

\[ \text{worlds} \]

of i-app +X1 +X2 -X3.

\[ \text{ofi-app} : \text{ofi} \ G \ M (\pi \ A (\[x\] B x)) \]

\[ \text{to} \]

\[ \text{type} \]

\[ \text{mode ofi-app +X1 +X2 -X3} \]

\[ \text{worlds} \]

of i-app +X1 +X2 -X3.

\[ \text{ofi-app} : \text{ofi} \ G \ M (\pi \ A (\[x\] B x)) \]

\[ \text{to} \]

\[ \text{type} \]

\[ \text{mode ofi-app +X1 +X2 -X3} \]

\[ \text{worlds} \]

of i-app +X1 +X2 -X3.

\[ \text{ofi-app} : \text{ofi} \ G \ M (\pi \ A (\[x\] B x)) \]

\[ \text{to} \]

\[ \text{type} \]

\[ \text{mode ofi-app +X1 +X2 -X3} \]

\[ \text{worlds} \]

of i-app +X1 +X2 -X3.

\[ \text{ofi-app} : \text{ofi} \ G \ M (\pi \ A (\[x\] B x)) \]

\[ \text{to} \]

\[ \text{type} \]

\[ \text{mode ofi-app +X1 +X2 -X3} \]

\[ \text{worlds} \]

of i-app +X1 +X2 -X3.

\[ \text{ofi-app} : \text{ofi} \ G \ M (\pi \ A (\[x\] B x)) \]

\[ \text{to} \]

\[ \text{type} \]

\[ \text{mode ofi-app +X1 +X2 -X3} \]

\[ \text{worlds} \]

of i-app +X1 +X2 -X3.

\[ \text{ofi-app} : \text{ofi} \ G \ M (\pi \ A (\[x\] B x)) \]

\[ \text{to} \]

\[ \text{type} \]

\[ \text{mode ofi-app +X1 +X2 -X3} \]

\[ \text{worlds} \]

of i-app +X1 +X2 -X3.

\[ \text{ofi-app} : \text{ofi} \ G \ M (\pi \ A (\[x\] B x)) \]

\[ \text{to} \]

\[ \text{type} \]

\[ \text{mode ofi-app +X1 +X2 -X3} \]

\[ \text{worlds} \]

of i-app +X1 +X2 -X3.

\[ \text{ofi-app} : \text{ofi} \ G \ M (\pi \ A (\[x\] B x)) \]

\[ \text{to} \]

\[ \text{type} \]

\[ \text{mode ofi-app +X1 +X2 -X3} \]

\[ \text{worlds} \]

of i-app +X1 +X2 -X3.

\[ \text{ofi-app} : \text{ofi} \ G \ M (\pi \ A (\[x\] B x)) \]

\[ \text{to} \]

\[ \text{type} \]

\[ \text{mode ofi-app +X1 +X2 -X3} \]

\[ \text{worlds} \]

of i-app +X1 +X2 -X3.

\[ \text{ofi-app} : \text{ofi} \ G \ M (\pi \ A (\[x\] B x)) \]

\[ \text{to} \]

\[ \text{type} \]

\[ \text{mode ofi-app +X1 +X2 -X3} \]

\[ \text{worlds} \]

of i-app +X1 +X2 -X3.

\[ \text{ofi-app} : \text{ofi} \ G \ M (\pi \ A (\[x\] B x)) \]

\[ \text{to} \]

\[ \text{type} \]

\[ \text{mode ofi-app +X1 +X2 -X3} \]

\[ \text{worlds} \]

of i-app +X1 +X2 -X3.

\[ \text{ofi-app} : \text{ofi} \ G \ M (\pi \ A (\[x\] B x)) \]

\[ \text{to} \]

\[ \text{type} \]

\[ \text{mode ofi-app +X1 +X2 -X3} \]

\[ \text{worlds} \]

of i-app +X1 +X2 -X3.

\[ \text{ofi-app} : \text{ofi} \ G \ M (\pi \ A (\[x\] B x)) \]

\[ \text{to} \]

\[ \text{type} \]

\[ \text{mode ofi-app +X1 +X2 -X3} \]

\[ \text{worlds} \]

of i-app +X1 +X2 -X3.

\[ \text{ofi-app} : \text{ofi} \ G \ M (\pi \ A (\[x\] B x)) \]

\[ \text{to} \]

\[ \text{type} \]

\[ \text{mode ofi-app +X1 +X2 -X3} \]

\[ \text{worlds} \]

of i-app +X1 +X2 -X3.

\[ \text{ofi-app} : \text{ofi} \ G \ M (\pi \ A (\[x\] B x)) \]

\[ \text{to} \]

\[ \text{type} \]

\[ \text{mode ofi-app +X1 +X2 -X3} \]

\[ \text{worlds} \]

of i-app +X1 +X2 -X3.

\[ \text{ofi-app} : \text{ofi} \ G \ M (\pi \ A (\[x\] B x)) \]

\[ \text{to} \]

\[ \text{type} \]

\[ \text{mode ofi-app +X1 +X2 -X3} \]

\[ \text{worlds} \]

of i-app +X1 +X2 -X3.
(x) lookup (cons nil x A) A can be constructed directly, as [x] lookup/\hit.

We can deal with terms with more than one bound variable in a similar manner to the of/\lam case above, by cutting the last variable with cut-of and all preceding variables with cut-ofe.

6 Conclusion

The explicit context method provides a general proof technique for theorems involving dependent types and one or more distinguished bound variables. In general, explicit contexts are much clumsier than LF’s ordinary usage with implicit contexts. (For example, explicit weakening is more of a bother than ordinary LF practice, where one can simply bind variables and not use them.) Therefore we do not advocate using explicit contexts throughout a development.

Instead, we recommend carrying out developments using LF in its conventional style, shifting into explicit form only when necessary. For example, Lee et al. [6] use explicit contexts to prove functionality of type-constructor substitution, and Crary [3] uses explicit contexts to prove the substitution lemma for a form of hereditary substitution arising in the metatheory of singleton kinds.

The existence of this general method applicable to conventional LF formalizations means that one can begin a Twelf formalization without worrying about being tripped up on this sort of issue. Moreover, in contrast to contextual modal type theory [10, 13], the method works without any extensions to LF or Twelf, so such formalizations can be carried out today.

Author’s Note

This revised version of this paper incorporates several simplifications to the explicit context method that were not used in the original paper [2]. The most significant simplification, the disentanglement of typing and context formation, so that ofe need not depend on isvar, was suggested by Daniel Lee [7].

References


Explicit Contexts in LF, Revision 3.
- : cut-of

%% inputs
([x] lam (B x) ([y] N x y))  %% induction variable
([x] [d:of x A]
  of/lam
  (Dof x d : {y} of y (B x) -> of (N x y) (C x y)))
(Dlookup : {x} lookup (G x) x A)

%% outputs
([x] ofe/lam ([y] Dofe' x y))
<- ({x} {d:of x A}
  cut-of ([y] N x y)
  ([y] [e:of y (B x)] Dof x d y e)
  ([y] lookup/hit)
  %%
  ([y] Dofe x d y : ofe (cons (G x) y (B x)) (N x y) (C x y)))
<- ({y}
  cut-of ([x] N x y)
  ([x] [d:of x A] Dofe x d y)
  ([x] lookup/miss (Dlookup x))
  %%
  ([x] Dofe' x y : ofe (cons (G x) y (B x)) (N x y) (C x y))).

Figure 8: Cut proof (of/lam case)