6-2008

Explicit Contexts in LF (Revised)

Karl Crary
Carnegie Mellon University

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,[1] 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:

```
www.cs.cmu.edu/~crary/papers/2009/excon-rev.elf
```

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.

```
tp     : type.
exp    : type.

o      : tp.
arrow  : tp -> tp -> tp.

b      : exp.
lam    : tp -> (exp -> exp) -> exp.
app    : exp -> exp -> exp.

of     : exp -> tp -> type.

of/b   : of b o.

of/lam : of (lam A ([x] M x)) (arrow A B)
          <- ({x} of x A -> of (M x) B).

of/app : of (app M N) B
          <- of M (arrow A B)
          <- of N A.

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

Figure 1: Simple types in LF

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]

```
subst : of M A
         -> ({x} of x A -> of (N x) B)
%%
         -> of (N M) B
         -> type.
%mode subst +X1 +X2 -X3.
%worlds (bind) (subst _ _ _).
```

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`:

Lemma 2.1 (Substitution)
If $\Gamma_1 \vdash M : A$ *and* $\Gamma_1, x{:}A, \Gamma_2 \vdash N : B$ *then* $\Gamma_1, \Gamma_2 \vdash N[M/x] : B$.

[2]For ease of readability, we will adopt the convention of placing all input arguments first, followed by the output arguments, with a blank comment line between.

```
- : subst
     %% inputs
     (Dm : of M A)
     ([x] [d:of x A]
        of/lam
        ([y] [e:of y B]
           Dn x d y e : of (N x y) C))
     %% output
     (of/lam
        ([y] [e:of y B] D y e)
        : of (lam B ([y] N M y)) (arrow B C))
     <- ({y} {e:of y B}  %% NB: y is outside x
           subst Dm ([x] [d] Dn x d y e)
           %%
           (D y e : of (N M y) C)).
```

Figure 2: Permuting `subst` proof (`of/lam` case)

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 Γ_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 Γ_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 Γ_1.) On the other hand, the inner context Γ_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 `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 "-".

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:

```
varof    : exp -> tp -> type.
of       : exp -> tp -> type.
path     : exp -> type.

of/var   : of X T
              <- varof X T.

path/var : path X
              <- varof X _.

%block bind
   : some {a:tp}
      block {x:exp} {d:varof x a}.
```

  In such a system, the substitution lemma:

```
subst
  : of M A
     -> ({x} varof X A  -> of (N x) (B x))
%%
     -> of (N M) (B M)
     -> type.
%mode subst +X1 +X2 -X3.
%worlds (bind) (subst _ _ _).
```

  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:

$$\frac{}{\Gamma \vdash X \leq X} \qquad \frac{(X{\leq}U) \in \Gamma \quad \Gamma \vdash U \leq T}{\Gamma \vdash X \leq T}$$

  This can be represented in LF by:

```
tp   : type.

assm : tp -> tp -> type.  %% assumption
sub  : tp -> tp -> type.  %% judgement

sub/refl  : sub X X
             <- assm X _.

sub/trans : sub X T
             <- assm X U
             <- sub U T.

%block tpbind
   : some {u:tp}
      block {x:tp} {d:assm x u}.
```

  A key lemma needed to prove the correctness of the $F_{\leq}$ algorithm is *narrowing,* which states that subtyping can be applied to assumptions:

```
narrow : ({x} assm x Q -> sub M N)
          -> sub P Q
%%
          -> ({x} assm x P -> sub M N)
          -> type.
%mode narrow +X1 +X2 -X3.
%worlds (tpbind) (narrow _ _ _).
```

  This is an inductively proven theorem with a distinguished bound variable, and the dependencies among subtyping bounds cause the same problems with variable permutation as dependent types do.

- **Functionality.** Functionality is the cousin of substitution where two equivalent terms are substituted into a term to obtain equivalent substitution instances:

```
of    : exp -> tp -> type.
equiv : exp -> exp -> tp -> type.

funct : equiv M1 M2 A
         -> ({x} of x A -> of (N x) (B x))
%%
         -> equiv (N M1) (N M2) (B M1)
         -> type.
%mode funct +X1 +X2 -X3.
%worlds (bind) (funct _ _ _).
```

  Functionality exhibits the same problem as substitution, but does not enjoy a trivial solution.

- **Hereditary substitution.** When defining LF and similar logical frameworks, it can be convenient to adopt a *canonical formulation,* in which only canonical forms are well-formed. A complication arises from substitution, since substitution instances of canonical forms are not necessarily canonical. To resolve this, one can define *hereditary substitution,* which reduces any redices resulting from substitution [15, 3].

  The top-level judgement defining hereditary substitution looks like:

```
atom : type.
term : type.

sub  : (atom -> term)
        -> term -> term -> type.
```

[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).

```
tp     : type.
exp    : type.

o      : tp.
p      : exp -> tp.
pi     : tp -> (exp -> tp) -> tp.

b      : exp.
lam    : tp -> (exp -> exp) -> exp.
app    : exp -> exp -> exp.

of     : exp -> tp -> type.

of/b   : of b o.

of/lam : of (lam A ([x] M x)) (pi A ([x] B x))
          <- ({x} of x A -> of (M x) (B x)).

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

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

Figure 3: Simple dependent types in LF

where "`sub ([x] N x) M O`" is read "hereditarily substituting `M` for `x` in `N x` yields `O`." In the dependently typed case, a similar judgement is required to substitute a term into a type.

Since hereditary substitution is a defined notion, its substitution lemma does not admit a trivial proof. (In fact, in the dependently typed case it is rather involved, even on paper.)

The explicit context method developed in this paper provides a principled, uniform, and dependable method to overcome these difficulties. For the first two examples, ad hoc workarounds are also known to exist. (As of this writing, we know of none that are published.) Also, for the narrowing example, Pientka [12] has shown that the difficulty can be circumvented by formulating the LF encoding differently. For the latter two, the explicit context method is the only known technique.

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 contructor `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.[5] The method generalizes smoothly from the simple dependently typed lambda calculus to other languages of interest, such as LF.

```
nat : type.

0 : nat.
s : nat -> nat.

lt : nat -> nat -> type.

lt/z : lt 0 (s _).
lt/s : lt (s N1) (s N2)
        <- lt N1 N2.

nat-eq : nat -> nat -> type.

nat-eq/i : nat-eq N N.
```

Figure 4: Natural numbers

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:

```
ctx  : type.

nil  : ctx.
cons : ctx -> exp -> tp -> ctx.
```

Thus, the context $x{:}o, y{:}o \to o$ is represented:

```
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 `isvar`. For every variable `x`, we assume `isvar x I`, for some natural number[6] `I`:

[5] In fact, the determining factor for Twelf as to whether the language 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] The definition of natural numbers is given in Figure 4.

```
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, $G \vdash M : A$.)

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.
```

```
lookup : ctx -> exp -> tp -> type.

lookup/hit  : lookup (cons G X A) X A.

lookup/miss : lookup (cons G Y _) X A
               <- lookup G X A.

append : ctx -> ctx -> ctx -> type.

append/nil  : append G nil G.

append/cons : append G1 (cons G2 X A)
               (cons G X A)
               <- append G1 G2 G.
```

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:

```
isvar-fun : isvar X I
             -> isvar X J
%%
             -> nat-eq I J
             -> type.
%mode isvar-fun +X1 +X2 -X3.
%worlds (ovar | bind)
        (isvar-fun _ _ _).
```

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

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

```
- :  isvar-fun D D nat-eq/i.
```

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

- As a corollary, `precedes` is a strict partial order:

```
precedes-irreflex : precedes X X
%%
                     -> false
                     -> type.
%mode precedes-irreflex +X1 -X2.
%worlds (ovar | bind)
        (precedes-irreflex _ _).

precedes-trans : precedes X Y
                  -> precedes Y Z
%%
                  -> precedes X Z
                  -> 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`:

```
extend-context
 : ordered G
%%
    -> ({x} isvar x I -> bounded G x)
    -> type.
%mode extend-context +X1 -X2.
%worlds (ovar | bind)
        (extend-context _ _).
```

- If `G` is well-formed and has the form `G1, x:A, G2` and looking up `x` returns `B x`, then `A` and `B x` are equal:

```
append-lookup-eq
 : ({x} append (cons G1 x A) (G2 x) (G x))
    -> ({x} isvar x I -> ordered (G x))
    -> ({x} lookup (G x) x (B x))
%%
    -> ({x} tp-eq A (B x))
    -> type.
%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:

```
lookup-pdv
 : ({x} append (cons G1 x A) (G2 x) (G x))
    -> append G1 (G2 M) G'
    -> ({x} lookup (G x) Y (B x))
%%
    -> lookup G' Y (B M)
    -> type.
%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 `G1` then it is well-typed under any `G` that extends `G1`:

```
weaken-ofe : append G1 G2 G
              -> ofe G1 M A
%%
              -> ofe G M A
              -> 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:

```
esubst : ({x} append (cons G1 x A) (G2 x) (G x))
          -> append G1 (G2 M) G'
          -> ({x} isvar x I -> ordered (G x))
          -> ofe G1 M A
          -> ({x} ofe (G x) (N x) (B x))
%%
          -> ofe G' (N M) (B  M)
          -> type.
%mode esubst +X1 +X2 +X3 +X4 +X5 -X6.
%worlds (ovar | bind)
        (esubst _ _ _ _ _ _).
```

It reads: if $\texttt{G x} = \texttt{G1}, \texttt{x}{:}\texttt{A}, (\texttt{G2 x})$ and $\texttt{G}' = \texttt{G1}, (\texttt{G2 M})$ and `G x` is well-formed, and if $\texttt{G1} \vdash \texttt{M} : \texttt{A}$ and $\texttt{G x} \vdash \texttt{N x} : \texttt{B x}$, then $\texttt{G'} \vdash \texttt{N M} : \texttt{B M}$. 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`

```
- : esubst
     %% inputs
     (DappendG
        : {x} append (cons G1 x A) (G2 x) (G x))
     (DappendG' : append G1 (G2 M) G')
     (Dordered : {x} isvar x I -> ordered (G x))
     (DofM : ofe G1 M A)
     ([x] ofe/lam
          (DofN x : {y} ofe (cons (G x) y (B x))
                              (N x y) (C x y))
          : ofe (G x) (lam (B x) ([y] N x y))
                    (pi (B x) ([y] C x y)))
     %% output
     (ofe/lam DofN'
        : ofe G' (lam (B M) ([y] N M y))
                 (pi (B M) ([y] C M y)))
     <- ({x} {d:isvar x I}
           extend-context
           (Dordered x d)
           %%
           (Dbounded x d : {y} isvar y J
                              -> bounded (G x) y))
     <- ({y} {e:isvar y J}
           esubst
           ([x] append/cons (DappendG x))
           (append/cons DappendG')
           ([x] [d:isvar x I]
              ordered/cons (Dbounded x d y e)
              : ordered (cons (G x) y (B x)))
           DofM
           ([x] DofN x y)
           %%
           (DofN' y : ofe (cons G' y (B M))
                            (N M y) (C M y))).
```

Figure 7: Explicit-context `esubst` proof (`ofe/lam` case)

`A` to `ofe G' M A`. The `ofe/app` case is a simple induction invocation.

The `ofe/lam` case in straightforward in the explicit-context setting. The LF binding for `y` is moved outside, but `y`'s typing assumption is now part of the `ofe` judgement, so it remains within the scope of `x`. An `isvar` assumption is added, using an order stamp obtained from `extend-context`. The complete `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 `ofe-to-of` lemma:

```
ofe-to-of : ofe nil M A
%%
             -> of M A
             -> type.
%mode ofe-to-of +X1 -X2.
%worlds (var | bind) (ofe-to-of _ _).
```

This states that if a typing judgement holds with an empty explicit context, it also hold with the context implicit. The worlds declaration shows that it can be used with `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 `isvar` assumption used in `ovar` blocks cannot appear within `ofe` or `of` derivations, we can use the simpler `var` block without loss of generality:

```
%block var : block {x:exp}.
```

The proof relies on a technical device, a judgement called `ofi`:

```
ofi : ctx -> exp -> tp -> type.

ofi/nil  : ofi nil M A
            <- of M A.

ofi/cons : ofi (cons G X A) M B
            <- (of X A -> ofi G M B).
```

The `ofi` judgement is an explicit-context typing judgement, like `ofe`, but it is based on the context rather than the term. While `ofe` has a rule for each term construct, `ofi` works by introducing an `of` assumption for each variable in the context, and then deferring to the `of` judgement.

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

```
ofe-to-ofi : ofe G M A
%%
              -> ofi G M A
              -> type.
%mode ofe-to-ofi +X1 -X2.
%worlds (var | bind) (ofe-to-ofi _ _).
```

We prove `ofe-to-ofi` by showing that each of the `ofe` rules applies to `ofi` as well. For example, for the `ofe/var` case:

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

we prove:

```
ofi-lookup : lookup G X A
%%
              -> ofi G X A
              -> type.
%mode ofi-lookup +X1 -X2.
%worlds (var | bind | ofblock)
        (ofi-lookup _ _).
```

and for the `ofe/app` case:

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

we prove:

```
ofi-app : ofi G M (pi A ([x] B x))
           -> ofi G N A
%%
           -> ofi G (app M N) (B N)
           -> type.
%mode ofi-app +X1 +X2 -X3.
%worlds (var | bind | ofblock)
        (ofi-app _ _ _).
```

Each of these lemmas is proven by a simple induction over the context `G`. Since the inductive case introduces a "disembodied" `of` assumption, their worlds must include the `ofblock` block:

```
%block ofblock : some {x:exp} {a:tp}
                   block {d:of x a}.
```

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:

```
of-to-ofe : of M A
%%
            -> ofe nil M A
            -> type.
%mode of-to-ofe +X1 -X2.
%worlds (bind) (of-to-ofe _ _).
```

This is trivial to prove, using the `ofe/closed` rule.

More often, however, there is at least one explicit free variable. (As in `subst`, for example.) Then we require a lemma such as:

```
of1-to-ofe
   : ({x} of x A -> of (M x) (B x))
%%
       -> ({x} ofe (cons nil x A) (M x) (B x))
       -> type.
%mode of1-to-ofe +X1 -X2.
%worlds (bind) (of1-to-ofe _ _).
```

5.1 Cut

The main lemma for proving `of1-to-ofe` (and similar results) is a cut principle that cuts a `lookup` judgement against an `of` assumption:[8]

```
cut-of : {M}  %% induction variable
             ({x} of x A -> of (M x) (B x))
          -> ({x} lookup (G x) x A)
%%
          -> ({x} ofe (G x) (M x) (B x))
          -> type.
%mode cut-of +X1 +X2 +X3 -X4.
%worlds (var | bind) (cut-of _ _ _ _).
```

[8] This is not quite the literal worlds declaration for `cut-of`, as Twelf requires that it be presented simultaneously with the worlds declaration for `cut-ofe`, discussed later.

The first argument is just the induction variable (Twelf requires the induction variable to be an explicit argument). The second argument states that `M x` has type `B x` assuming `x:A` in the implicit context. The third argument satisfies `x:A` using an explicit context lookup in `G x`. The lemma then provides that `M x` has type `B x` in the explicit context `G x`.

The proof proceeds by induction on the term `M` (not on its typing derivation). The variable case (that is, when the second argument is `([x] [d] d)`) uses `ofe/var` and the proffered `lookup` judgement. The closed case (that is, when the second argument is `([x] [d] D)`, where `D` does not depend on `d`) is trivial, using the `ofe/closed` rule. The `of/app` case is a simple induction invocation.

The interesting case is `of/lam`, for which the Twelf code is given in Figure 8. In that case, `M x` has the form `lam (B x) ([y] N x y)`. The typing judgement for the body has the LF type:

```
{x} of x A
   -> {y} of y (B x)
   -> of (N x y) (C x y)
```

We wish to cut the `lookup` judgement against the `of x A` assumption, but we have the `of y (B x)` assumption in the way. With simple types we could move the latter assumption outside the `x`, but that does not generalize to dependent types. Consequently, we must first cut against `y`'s typing assumption, before returning to `x`'s.

After the first cut, the typing judgement for the body has the LF type:

```
{x} of x A
  -> {y}
     ofe (cons (G x) y (B x)) (N x y) (C x y)
```

Now that `y`'s typing assumption has been moved into the explicit context, we can move `y` itself outside the scope of `x`. It remains to cut the original `lookup` assumption against `x`'s typing assumption within the new `ofe` derivation. For this we require a second lemma, proved simultaneously with `cut-of`:

```
cut-ofe : {M}  %% induction variable
              ({x} of x A
                 -> ofe (G x) (M x) (B x))
           -> ({x} lookup (G x) x A)
%%
           -> ({x} ofe (G x) (M x) (B x))
           -> type.
%mode cut-ofe +X1 +X2 +X3 -X4.
%worlds (var | bind) (cut-ofe _ _ _ _).
```

The `ofe/closed` case of `cut-ofe` defers back to `cut-of`. The remaining cases are all simple induction invocations. This includes `ofe/lam`, which is simple because its body already uses an `ofe` judgement, so only one recursive cut is required.

Let us return to the `of-lam` case of `cut-of` to make one final observation. Since the case makes two recursive calls, where the second operates on the result of the first, it is not possible to do this proof by induction on derivations. Instead, we do it by induction on the term itself, which is undisturbed by all the processing of typing derivations.

Once the cut lemma is established, the `ofe1-to-ofe` lemma is an easy corollary, since a derivation of `{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

[1] Brian E. Aydemir, Aaron Bohannon, Matthew Fairbairn, J. Nathan Foster, Benjamin C. Pierce, Peter Sewell, Dimitrios Vytiniotis, Geoffrey Washburn, Stephanie Weirich, and Steve Zdancewic. Mechanized metatheory for the masses: The POPLMark challenge. In *International Conference on Theorem Proving in Higher Order Logics*, pages 50–65, 2005.

[2] Karl Crary. Explicit contexts in LF. In *Workshop on Logical Frameworks and Meta-Languages: Theory and Practice*, Pittsburgh, Pennsylvania, 2008.

[3] Karl Crary. A syntactic account of singleton types via hereditary substitution. Technical report, Carnegie Mellon University, School of Computer Science, 2009.

[4] Robert Harper, Furio Honsell, and Gordon Plotkin. A framework for defining logics. *Journal of the ACM*, 40(1):143–184, January 1993.

[5] Robert Harper and Mark Lillibridge. A type-theoretic approach to higher-order modules with sharing. In *Twenty-First ACM Symposium on Principles of Programming Languages*, pages 123–137, Portland, Oregon, January 1994.

[6] Daniel K. Lee, Karl Crary, and Robert Harper. Towards a mechanized metatheory of Standard ML. In *Thirty-Fourth ACM Symposium on Principles of Programming Languages*, Nice, France, January 2007.

[7] Daniel K. Lee, Derek Dreyer, and Andreas Rossberg. Mechanizing the metatheory of a language with linear resources and context effects (abstract). Presented in *2008 Informal ACM SIGPLAN Workshop on Mechanizing Metatheory*, 2008.

[8] Xavier Leroy. Manifest types, modules and separate compilation. In *Twenty-First ACM Symposium on Principles of Programming Languages*, pages 109–122, Portland, Oregon, January 1994.

[9] Xavier Leroy. Applicative functors and fully transparent higher-order modules. In *Twenty-Second ACM Symposium on Principles of Programming Languages*, San Francisco, January 1995.

[10] Aleksandar Nanevski, Frank Pfenning, and Brigitte Pientka. A contextual modal type theory. *ACM Transactions on Computational Logic*, 2008. To appear.

[11] Frank Pfenning and Carsten Schürmann. *Twelf User's Guide, Version 1.4*, 2002. Available electronically at `http://www.cs.cmu.edu/~twelf`.

[12] Brigitte Pientka. Proof pearl: The power of higher-order encodings in the logical framework LF. In *Twentieth International Conference on Theorem Proving in Higher Order Logics*, volume 4732 of *Lecture Notes in Computer Science*, pages 246–261, Kaiserslautern, Germany, September 2007. Springer.

[13] Brigitte Pientka. A type-theoretic foundation for programming with higher-order abstract syntax and first-class substitutions. In *Thirty-Fifth ACM Symposium on Principles of Programming Languages*, pages 371–382, San Francisco, California, January 2008.

[14] Roberto Virga. Higher-order superposition for dependent types. In *Seventh International Conference on Rewriting Techniques and Applications*, 1995.

[15] Kevin Watkins, Iliano Cervesato, Frank Pfenning, and David Walker. A concurrent logical framework I: Judgments and properties. Technical Report CMU-CS-02-101, Carnegie Mellon University, School of Computer Science, 2002. Revised May 2003.

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-ofe ([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)