2009

Higher-order Representation of Substructural Logics

Karl Crary
Carnegie Mellon University, crary@cs.cmu.edu

Published In
.

Higher-order Representation of Substructural Logics

Karl Crary

Carnegie Mellon University

Abstract

We present a technique for higher-order representation of substructural logics such as linear or modal logic. We show that such logics can be encoded in the (ordinary) Logical Framework, without any linear or modal extensions. Using this encoding, metatheoretic proofs about such logics can easily be developed in the Twelf proof assistant.

1 Introduction

The Logical Framework (or LF) [6] provides a powerful and flexible framework for encoding deductive systems such as programming languages and logics. LF employs an elegant account of binding structure by identifying object-language variables with LF variables, object-language contexts with (fragments of) the LF context, and object-language binding occurrences with LF lambda abstraction. This account of binding, often called higher-order abstract syntax [12], automatically handles most operations that pertain to binding, including alpha-equivalence, substitution, and variable-freshness conventions [3].

Since the object-language context is maintained implicitly, as part of the built-in LF context, the structural properties of LF contexts (such as weakening and contraction) automatically apply to the object language as well. Ordinarily this is desirable, but it poses a problem for encoding substructural logics that do not possess those properties.[1] For example, linear logics (by design) satisfy neither weakening nor contraction, so it would seem that they cannot be encoded in LF.

One solution to this problem is to extend LF with linear features. Linear LF [4] extends LF with linear assumptions and connectives. This provides the ability to encode linear logics. However, linearity has yet to be implemented in Twelf [13], the proof assistant that implements LF, in part due to unresolved complications that linearity creates in its metalogical apparatus. Consequently, Linear LF is not currently an option for those engaged in formalizing metatheory. Moreover, Linear LF does not give us any assistance with other substructural logics, such as affine, strict, or modal logic.

Another option is to break with standard LF practice and model object-language contexts explicitly [5]. Explicit contexts can be reconciled with higher-order abstract syntax, thereby retaining many of the benefits of LF. Once contexts are explicit, it is easy to state inference rules that handle the context in an appropriate way for a substructural logic. However, the explicit context method is clumsy to work with and sacrifices some of the advantages of LF. For example, although substitution is still free (since the syntax of terms is unchanged), the *substitution lemma* is not. The explicit context method is typically used internally within a proof, rather than in the "official" formalization of a logic.

In this paper we advocate a more general and workable approach in which we look at substructural logic from a slightly different perspective. Rather than viewing a substructural logic from the perspective of its contexts (that is, collections of assumptions), we suggest it is profitable to look at it from the perspective of its individual assumptions.

The essence of linear logic is not that type-checking splits the context when it checks a (multiplicative) term with multiple subterms. The essence of linear logic is that an assumption is used exactly once. The latter property can be stated on an assumption-by-assumption basis, without reference to contexts. Thus, wherever an assumption is introduced, as part of the typing rule that introduced it, we can check that that assumption is used linearly.

At first glance, it might appear that linearity must be a meta-judgement, tracing the use of assumptions throughout a typing derivation. That would make it very awkward to use in practice. Fortunately, however, to check the linear use of assumptions, we need look only at proof terms; there is no need to examine typing derivations.

The idea of linearity as a judgement over proof terms dates to the early days of LF. Avron *et al.* [2, 1] suggested that linearity can be expressed by imposing a lattice structure on proof terms and defining linear proof terms as those that are strict and distributive, when viewed as a function of their linear variables.

In this paper, we suggest a simpler formulation of linearity, based on tracking variables through the proof terms of linear logic. This allows for a clean, practical definition of linearity.

We express linear logic using two judgements, the usual typing judgement:

```
of : term -> tp -> type.
```

and a linearity judgement:

```
linear : (term -> term) -> type.
```

[1] Substructural logics may be defined in various different ways. For our purposes, we define substructural logic to mean any logic in which it is not the case that every bound variable can be freely used, or not, throughout its scope.

The judgement `linear`$(\lambda \mathtt{x}.M_{\mathtt{x}})$ should be read as "the variable `x` is used linearly (*i.e.,* is used exactly once) in $M_{\mathtt{x}}$."

In this paper, we illustrate the use of a substructural judgement (such as `linear`) in three settings: linear logic, dependently typed linear logic, and judgemental modal logic [11]. Many other substructural logics including affine logic and strict logic can be handled analogously. Some others, such as ordered logic [15, 14], cannot, because the rules of the logic make it impossible to handle assumptions independently. We briefly discuss the latter in Section 5.

The full Twelf development can be found on-line at:

```
www.cs.cmu.edu/~crary/papers/2009/substruct.tar
```

In our discussion, we assume familiarity with the Logical Framework, and with linear and modal logic. Some familiarity with Twelf may also be helpful. The sections on adequacy are technical, but the remainder of the paper should be accessible to the casual practitioner.

2 Linear Logic

We begin by representing the syntax of linear logic in the usual fashion. The LF encoding, with the standard on-paper notation written alongside it for reference, is shown in Figure 1. The type `atom` ranges over a fixed set of atomic propositions.

On paper, we represent linear logic with the typing judgment $\Gamma; \Delta \vdash M : A$. In this, the first context, Γ, represents the unrestricted context (*i.e.,* truth), and the second context, Δ, represents the linear context (*i.e.,* resources). To simplify the notation, we adopt the convention that the linear context is unordered. Thus (Δ, Δ') refers to a context that can be split into two pieces Δ and Δ' that may possibly be interleaved. We also adopt the convention that all the variables appearing in either context must be distinct.

The encoding of the static semantics, as discussed previously, is given by two judgements:

```
of     : term -> tp -> type.
linear : (term -> term) -> type.
```

We read "`of M A`" as "`M` is of type `A`," and we read "`linear ([x:term] M x)`" as "`x` is used linearly in `(M x)`." Note that `[x:term]` is Twelf's concrete syntax for LF lambda abstraction[2] (λ`x:term.`). Twelf can usually infer the domain type, leaving just `[x]`.

We proceed rule-by-rule to show the encoding of the static semantics.

Variables The rule for linear variables states that a linear variable may be used provided there are no *other* linear variables in scope:

$$\frac{}{\Gamma; x{:}A \vdash x : A}$$

There is no typing rule for variables in the encoding; that is handled automatically by higher-order representations. However, there is a linearity rule that states that `x` is linear in `x`:

[2]Keep in mind the distinction between lambda abstraction in LF, which represents binding, and lambda abstraction in the object language (`llam`).

```
tp : type.                          A  ::=

atomic : atom -> tp.                    a
lolli  : tp -> tp -> tp.              | A ⊸ A
tensor : tp -> tp -> tp.              | A ⊗ A
with   : tp -> tp -> tp.              | A & A
plus   : tp -> tp -> tp.              | A + A
one    : tp.                          | 1
zero   : tp.                          | 0
top    : tp.                          | ⊤
!      : tp -> tp.                    | !A

term : type.                        M  ::=
                                          x
llam   : (term -> term) -> term.      | λx.M
lapp   : term -> term -> term.        | M M
tpair  : term -> term -> term.        | M ⊗ M
lett   : term
         -> (term -> term -> term)
         -> term                      | let x ⊗ x = M in M
pair   : term -> term -> term.        | ⟨M, M⟩
pi1    : term -> term.                | π1 M
pi2    : term -> term.                | π2 M
in1    : term -> term.                | in1 M
in2    : term -> term.                | in2 M
case   : term
         -> (term -> term)
         -> (term -> term)
         -> term.                     | case(M, x.M.x.M)
star   : term.                        | *
leto   : term -> term -> term.
                                      | let * = M in M
any    : term -> term.                | any M
unit   : term.                        | ⟨⟩
bang   : term -> term.                | !M
letb   : term -> (term -> term) -> term.
                                      | let !x = M in M
```

Figure 1: Linear logic syntax

```
linear/var : linear ([x] x).
```

The rule for unrestricted variables states that an unrestricted variable may be used provided there any no linear variables in scope:

$$\frac{\Gamma(x) = A}{\Gamma; \epsilon \vdash x : A}$$

As with linear variables, there is no typing rule for unrestricted variables in the encoding. There is also no linearity rule for unrestricted variables.

Linear implication The introduction rule for linear implication is:

$$\frac{\Gamma; (\Delta, x{:}A) \vdash M : B}{\Gamma; \Delta \vdash \lambda x.M : A \multimap B}$$

This is encoded using two rules:

```
of/llam
  : of (llam ([x] M x)) (lolli A B)
     <- ({x:term} of x A -> of (M x) B)
     <- linear ([x] M x).

linear/llam
  : linear ([y] llam ([x] M y x))
     <- ({x:term} linear ([y] M y x)).
```

Note that `{x:term}` is Twelf's concrete syntax for the dependent function space (Πx:term.). Again, Twelf can usually infer the domain type, leaving just `{x}`.

The typing rule has the usual typing premise, plus a second premise that requires that the argument be used linearly in the body. The linearity rule says that a variable `y` is linear in a function (`llam ([x] M y x)`) if it is linear in its body (`M y x`) for any choice of `x`.

The elimination rule splits the linear context between the function and argument:

$$\frac{\Gamma;\Delta \vdash M : A \multimap B \quad \Gamma;\Delta' \vdash N : A}{\Gamma;(\Delta,\Delta') \vdash MN : B}$$

This is encoded using three rules:

```
of/lapp
  : of (lapp M N) B
     <- of M (lolli A B)
     <- of N A.

linear/lapp1
  : linear ([x] lapp (M x) N)
     <- linear ([x] M x).

linear/lapp2
  : linear ([x] lapp M (N x))
     <- linear ([x] N x).
```

The typing rule is standard. There are two linearity rules, one for each way a linear variable might be used. The first linearity rule says that `x` is linear in `(lapp (M x) N)` if it is linear in `(M x)` and does not appear in `N`. (Since implicitly bound meta-variables such as `M` and `N` are quantified on the outside, stating `N` without a dependency on `x` means that `N` is closed with respect to `x`.) The second linearity rule provides the symmetric case.

Multiplicative conjunction The introduction rule for tensor is:

$$\frac{\Gamma;\Delta \vdash M : A \quad \Gamma;\Delta' \vdash N : B}{\Gamma;(\Delta,\Delta') \vdash M \otimes N : A \otimes B}$$

This is encoded using three rules, in a similar fashion to function application:

```
of/tpair
  : of (tpair M N) (tensor A B)
     <- of M A
     <- of N B.

linear/tpair1
  : linear ([x] tpair (M x) N)
     <- linear ([x] M x).

linear/tpair2
  : linear ([x] tpair M (N x))
     <- linear ([x] N x).
```

The elimination rule is:

$$\frac{\Gamma;\Delta \vdash M : A \otimes B \quad \Gamma;(\Delta', x{:}A, y{:}B) \vdash N : C}{\Gamma;(\Delta,\Delta') \vdash \mathsf{let}\, x \otimes y = M \,\mathsf{in}\, N : C}$$

In the encoding, the typing rule requires that `x` and `y` are linear in N. As in previous cases where the linear context is split, there are two linearity rules depending on whether a linear variable is used in the let-bound term or the body:

```
of/lett
  : of (lett M ([x] [y] N x y)) C
     <- of M (tensor A B)
     <- ({x} of x A
          -> {y} of y B -> of (N x y) C)
     <- ({y} linear ([x] N x y))
     <- ({x} linear ([y] N x y)).

linear/lett1
  : linear ([z] lett (M z) ([x] [y] N x y))
     <- linear ([z] M z).

linear/lett2
  : linear ([z] lett M ([x] [y] N z x y))
     <- ({x} {y} linear ([z] N z x y)).
```

Additive conjunction The introduction rule for "with" does not split the context:

$$\frac{\Gamma;\Delta \vdash M : A \quad \Gamma;\Delta \vdash N : B}{\Gamma;\Delta \vdash \langle M, N\rangle : A \,\&\, B}$$

In the encoding, there is one linearity rule, requiring that linear variables be linear in both constituents of the pair:

```
of/pair
  : of (pair M N) (with A B)
     <- of M A
     <- of N B.

linear/pair
  : linear ([x] pair (M x) (N x))
     <- linear ([x] M x)
     <- linear ([x] N x).
```

The elimination rules are straightforward:

$$\frac{\Gamma;\Delta \vdash M : A \,\&\, B}{\Gamma;\Delta \vdash \pi_1 M : A} \qquad \frac{\Gamma;\Delta \vdash M : A \,\&\, B}{\Gamma;\Delta \vdash \pi_2 M : B}$$

```
of/pi1     : of (pi1 M) A
               <- of M (with A B).

of/pi2     : of (pi2 M) B
               <- of M (with A B).

linear/pi1 : linear ([x] pi1 (M x))
               <- linear ([x] M x).

linear/pi2 : linear ([x] pi2 (M x))
               <- linear ([x] M x).
```

Disjunction The introduction rules for plus are straightforward:

$$\frac{\Gamma;\Delta \vdash M : A}{\Gamma;\Delta \vdash \mathsf{in}_1 M : A + B} \qquad \frac{\Gamma;\Delta \vdash M : B}{\Gamma;\Delta \vdash \mathsf{in}_2 M : A + B}$$

```
of/in1     : of (in1 M) (plus A B)
               <- of M A.

of/in2     : of (in2 M) (plus A B)
               <- of M B.

linear/in1 : linear ([x] in1 (M x))
               <- linear ([x] M x).

linear/in2 : linear ([x] in2 (M x))
               <- linear ([x] M x).
```

The elimination rule splits the context into two pieces, one for the discriminant and one used by both arms:

$$\frac{\Gamma;\Delta \vdash M : A + B \quad \Gamma;(\Delta', x{:}A) \vdash N_1 : C \quad \Gamma;(\Delta', x{:}B) \vdash N_2 : C}{\Gamma;(\Delta,\Delta') \vdash \mathsf{case}(M, x.N_1, x.N_2) : C}$$

In the encoding, the typing rule requires that each arm's bound variable be used linearly. The linearity rules provide the two cases, one when the variable is used linearly in the discriminant, and one in which is it used linearly in both arms:

```
of/case
  : of (case M ([x] N1 x) ([x] N2 x)) C
     <- of M (plus A B)
     <- ({x} of x A -> of (N1 x) C)
     <- ({x} of x B -> of (N2 x) C)
     <- linear ([x] N1 x)
     <- linear ([x] N2 x).

linear/case1
  : linear ([y] case (M y) ([x] N1 x) ([x] N2 x))
     <- linear ([y] M y).

linear/case2
  : linear ([y] case M ([x] N1 y x) ([x] N2 y x))
     <- ({x} linear ([y] N1 y x))
     <- ({x} linear ([y] N2 y x)).
```

Exponentiation The introduction rule for exponentiation requires that the linear context be empty:

$$\frac{\Gamma;\epsilon \vdash M : A}{\Gamma;\epsilon \vdash\ !M : !A}$$

In the encoding, this means there is no linearity rule, since variables cannot be linear in exponents:

```
of/bang : of (bang M) (! A)
            <- of M A.
```

The elimination rule splits the context and adds the newly bound variable to the unrestricted context:

$$\frac{\Gamma;\Delta \vdash M : !A \quad (\Gamma, x{:}A);\Delta' \vdash N : C}{\Gamma;(\Delta,\Delta') \vdash \mathsf{let}\, !x = M \,\mathsf{in}\, N : C}$$

In the encoding, the unrestricted nature of `x` is handled by *not* checking that `x` is linear in `(N x)`. The linearity rules work in the usual fashion:

```
of/letb
  : of (letb M ([x] N x)) B
     <- of M (! A)
     <- ({x} of x A -> of (N x) B).

linear/letb1
  : linear ([y] letb (M y) N)
     <- linear M.

linear/letb2
   : linear ([y] letb M ([x] N y x))
      <- ({x} linear ([y] N y x)).
```

Units The unit for tensor is 1:

$$\frac{}{\Gamma;\epsilon \vdash * : 1} \qquad \frac{\Gamma;\Delta \vdash M : 1 \quad \Gamma;\Delta' \vdash N : C}{\Gamma;(\Delta,\Delta') \vdash \mathsf{let} * = M \,\mathsf{in}\, N : C}$$

The encoding is straightforward, with no linearity rule for introduction since variables cannot be linear in $*$:

```
of/star      : of star one.

of/leto      : of (leto M N) C
                 <- of M one
                 <- of N C.

linear/leto1 : linear ([x] leto (M x) N)
                 <- linear ([x] M x).

linear/leto2 : linear ([x] leto M (N x))
                 <- linear ([x] N x).
```

The unit for "with", $\top$, is more interesting. It stands for an unknown collection of resources, and consequently has an introduction form but no elimination form:

$$\frac{}{\Gamma;\Delta \vdash \langle\rangle : \top}$$

The encoding provides that any variable is linear in `unit`:

```
of/unit     : of unit top.

linear/unit : linear ([x] unit).
```

The unit for plus, 0, represents falsehood. Accordingly, it has an elimination form but no introduction form. The elimination form behaves a little bit like $\langle\rangle$; any resources not used to prove 0 may be discarded:

$$\frac{\Gamma;\Delta \vdash M : 0}{\Gamma;(\Delta,\Delta') \vdash \mathsf{any}\, M : C}$$

In the encoding there are two linearity rules. A variable is linear in $(\mathsf{any}\, M)$ if is is linear in M or if it does not appear in M at all:

```
of/any      : of (any M) T
                <- of M zero.

linear/any1 : linear ([x] any (M x))
                <- linear M.

linear/any2 : linear ([x] any M).
```

Note that it is tempting but incorrect to simplify this to the single rule:

```
linear/any-wrong : linear ([x] any (M x)).
```

That rule would allow `x` to be used multiple times in `(M x)`, which is not permitted. It would be tantamount to moving the entire linear context into the unrestricted context, rather than merely discarding any unused resources.

2.1 Adequacy

It seems intuitively clear that the preceding is a faithful representation of linear logic. We wish to go further and make the correspondence rigorous, following the adequacy argument of Harper *et al.* [6]. Adequacy establishes a isomorphism between the object language (linear logic in this case) and its encoding in LF. As usual, an isomorphism is a bijection that respects the relevant operations.

For syntax, the only primitively meaningful operation is substitution. (Other operations are given by defined semantics.) Thus, an isomorphism for syntax is a bijective translation that respects substitution. Our translation for syntax (written $\ulcorner - \urcorner$) is standard, so we will omit the obvious details of its definition and simply state its adequacy theorem for reference:

Definition 2.1 *Translation of variable sets is defined:*

$$\ulcorner \{\mathtt{x}_1, \ldots, \mathtt{x}_n\} \urcorner = \mathtt{x}_1{:}\mathtt{term}, \ldots, \mathtt{x}_n{:}\mathtt{term}$$

Theorem 2.2 (Syntactic adequacy)

1. *Let Type be the set of linear logic types. Then there exists a bijection $\ulcorner - \urcorner$ between Type and LF canonical forms* `P` *such that* $\vdash_{LF}$ `P : tp`*. (Variables cannot appear within types, so there is no substitution to respect.)*
2. *Let S be a set of variables and let $Term_S$ be the set of linear logic terms whose free variables are contained in S. Then there exists a bijection $\ulcorner - \urcorner$ between $Term_S$ and LF canonical forms* `P` *such that $\ulcorner S \urcorner \vdash_{LF}$ `P : term`. Moreover, $\ulcorner - \urcorner$ respects substitution: $\ulcorner [M/\mathtt{x}]N \urcorner = [\ulcorner M \urcorner / \mathtt{x}] \ulcorner N \urcorner$.*

For semantic adequacy, we wish to establish a bijective translation between typing derivations and LF canonical forms.[3] The usual statement of adequacy for typing is something to the effect of:

Definition 2.3 *Translation of contexts is defined:*

$$\ulcorner \mathtt{x}_1{:}A_1, \ldots, \mathtt{x}_n{:}A_n \urcorner = \begin{array}{l} \mathtt{x}_1{:}\mathtt{term}, \mathtt{dx}_1{:}\mathtt{of}\ \mathtt{x}_1\ \ulcorner A_1 \urcorner, \ldots, \\ \mathtt{x}_n{:}\mathtt{term}, \mathtt{dx}_n{:}\mathtt{of}\ \mathtt{x}_n\ \ulcorner A_n \urcorner \end{array}$$

Non-Theorem 2.4 *There exists a bijection between derivations of the judgement $\Gamma \vdash M : A$ and LF canonical forms* `P` *such that $\ulcorner \Gamma \urcorner \vdash_{LF}$ `P : of` $\ulcorner M \urcorner \ulcorner A \urcorner$.*

[3] That is, we view typing derivations as having no operations to respect. Harper *et al.* suggest that substitution of derivations for assumptions is a meaningful operation on typing derivations, and prove that their translation respects such substitutions. This could be done in our setting as well. However, we take the view that when substituting derivations for assumptions, we care only that the resulting derivation exists (this being the standard substitution lemma), and not about the identity of that resulting derivation.

Unfortunately, this simple statement of adequacy does not work in the presence of linearity. Consider the judgement $\epsilon; x{:}a \vdash \langle\rangle \otimes \langle\rangle : \top \otimes \top$. It has two derivations, depending on which conjunct is chosen to consume the assumption:

$$\dfrac{\dfrac{}{\epsilon; x{:}a \vdash \langle\rangle : \top} \quad \dfrac{}{\epsilon; \epsilon \vdash \langle\rangle : \top}}{\epsilon; x{:}a \vdash \langle\rangle \otimes \langle\rangle : \top \otimes \top} \qquad \dfrac{\dfrac{}{\epsilon; \epsilon \vdash \langle\rangle : \top} \quad \dfrac{}{\epsilon; x{:}a \vdash \langle\rangle : \top}}{\epsilon; x{:}a \vdash \langle\rangle \otimes \langle\rangle : \top \otimes \top}$$

However, the LF type corresponding to that judgement,

```
{x:term} of x (atomic a)
 -> of (tpair unit unit) (tensor top top)
```

contains only one canonical form, namely:

```
[x:term] [dx:of x (atomic a)]
    of/tpair of/unit of/unit
```

So linear-logic typing derivations are not in bijection with the LF encoding of typing in general. Our isomorphism must take linearity into account, and not only where linearity is a premise of a typing rule.

Consequently, we establish a correspondence between each linear-logic typing derivation on the one hand, and an LF proof of typing paired with a collection of LF proofs of linearity on the other. Alas, this is notationally awkward when compared with the usual adequacy theorem.

Definition 2.5 *An* encoding structure for $\Gamma; \Delta \vdash M : A$ *is a pair* $(\mathtt{P}, H)$ *of an LF canonical form* `P` *and a finite mapping H from variables to LF canonical forms, such that:*

- $\ulcorner \Gamma, \Delta \urcorner \vdash_{LF}$ `P : of` $\ulcorner M \urcorner \ulcorner A \urcorner$*, and*
- $\mathrm{Domain}(H) = \mathrm{Domain}(\Delta)$*, and*
- *For each variable* `y` *in* $\mathrm{Domain}(\Delta)$*,* $\ulcorner S_{\mathtt{y}} \urcorner \vdash_{LF} H(\mathtt{y}) : \mathtt{linear\ ([y:term]}\ulcorner M \urcorner\mathtt{)}$*, where* $S_{\mathtt{y}} = \mathrm{Domain}(\Gamma, \Delta) \setminus \{\mathtt{y}\}$.

Theorem 2.6 (Semantic adequacy) *There exists a bijection $\ulcorner - \urcorner$ between derivations of the judgement $\Gamma; \Delta \vdash M : A$ and encoding structures for $\Gamma; \Delta \vdash M : A$.*

Proving adequacy is typically straightforward but tedious once it is stated correctly. The same is true here, but the tedium is a bit more pronounced because of the need to manipulate encoding structures, rather than just canonical forms. We give a few cases by way of example:

Proof Sketch

First, by induction on derivations, we construct the translation and show it is type correct.

- Suppose ∇ is the derivation:

  $$\dfrac{}{\Gamma; x{:}A \vdash x : A}$$

  Then $\ulcorner \nabla \urcorner \stackrel{\mathrm{def}}{=} (\mathtt{dx}, \{\mathtt{x} \mapsto \mathtt{linear/var}\})$.
- Suppose ∇ is the derivation:

  $$\dfrac{\Gamma(x) = A}{\Gamma; \epsilon \vdash x : A}$$

  Then $\ulcorner \nabla \urcorner \stackrel{\mathrm{def}}{=} (\mathtt{dx}, \emptyset)$.

- Suppose ∇ is the derivation:

$$\dfrac{\begin{array}{c}\nabla_1\\ \vdots\\ \Gamma;(\Delta, x{:}A) \vdash M : B\end{array}}{\Gamma;\Delta \vdash \lambda x.M : A \multimap B}$$

Let $\ulcorner\nabla_1\urcorner = (\mathtt{P}_1, H_1)$. By induction, $(\mathtt{P}_1, H_1)$ is an encoding structure for $\Gamma;(\Delta, x{:}A) \vdash M : B$, so:

$$\ulcorner\Gamma, \Delta\urcorner, \mathtt{x{:}term}, \mathtt{dx{:}of\ x}\ulcorner A\urcorner \vdash_{LF} \mathtt{P}_1 : \mathtt{of}\ulcorner M\urcorner\ulcorner B\urcorner$$

and

$$\ulcorner\mathrm{Domain}(\Gamma, \Delta)\urcorner \vdash_{LF} H_1(\mathtt{x}) : \mathtt{linear\ ([x]}\ulcorner M\urcorner\mathtt{)}$$

Therefore:

$$\begin{array}{l}\ulcorner\Gamma;\Delta\urcorner \vdash_{LF} \mathtt{of/llam}\\ \qquad\qquad (H_1(\mathtt{x}))\\ \qquad\qquad (\mathtt{[x]\ [dx]\ P}_1)\\ \qquad\quad : \mathtt{of\ (llam\ ([x]}\ulcorner M\urcorner\mathtt{))}\\ \qquad\qquad \mathtt{(lolli}\ulcorner A\urcorner\ulcorner B\urcorner\mathtt{)}\end{array}$$

So let

$$\ulcorner\nabla\urcorner = (\mathtt{of/llam}\ (H_1(\mathtt{x}))\ (\mathtt{[x]\ [dx]\ P}_1), H)$$

where for each $\mathtt{y}$ in $\mathrm{Domain}(\Delta)$, $H(\mathtt{y}) \stackrel{\text{def}}{=} \mathtt{linear/llam\ ([x]}\ H_1(\mathtt{y})\mathtt{)}$.

- Suppose ∇ is the derivation:

$$\dfrac{\begin{array}{cc}\nabla_1 & \nabla_2\\ \vdots & \vdots\\ \Gamma;\Delta_1 \vdash M : A \multimap B & \Gamma;\Delta_2 \vdash N : A\end{array}}{\Gamma;(\Delta_1, \Delta_2) \vdash MN : B}$$

Let $\ulcorner\nabla_1\urcorner = (\mathtt{P}_1, H_1)$ and let $\ulcorner\nabla_2\urcorner = (\mathtt{P}_2, H_2)$. By induction $(\mathtt{P}_1, H_1)$ is an encoding structure for $\Gamma;\Delta_1 \vdash M : A \multimap B$ and $(\mathtt{P}_2, H_2)$ is an encoding structure for $\Gamma;\Delta_2 \vdash N : A$.

Let $\mathtt{y} \in \mathrm{Domain}(\Delta_1, \Delta_2)$ be arbitrary. Let $S = \mathrm{Domain}(\Gamma)$ and $S_i = \mathrm{Domain}(\Delta_i)$. Then either $\mathtt{y} \in S_1$ and $\mathtt{y} \notin S_2$ or vice versa. Suppose the former. Then:

$$\ulcorner S \cup S_1 \setminus \{\mathtt{y}\}\urcorner \vdash_{LF} H_1(\mathtt{y}) : \mathtt{linear\ ([y]}\ulcorner M\urcorner\mathtt{)}$$

Also, since $\mathtt{y} \notin \mathrm{Domain}(\Delta_2)$, $\mathtt{y}$ is not free in N or (consequently) in $\ulcorner N\urcorner$. Therefore:

$$\begin{array}{l}\ulcorner S \cup S_1 \cup S_2 \setminus \{\mathtt{y}\}\urcorner \vdash_{LF} \mathtt{linear/lapp1}\ (H_1(\mathtt{y}))\\ \qquad\qquad\qquad\qquad : \mathtt{linear\ ([y]\ lapp}\ulcorner M\urcorner\ulcorner N\urcorner\mathtt{)}\end{array}$$

The other case is symmetric.

So let $\ulcorner\nabla\urcorner = (\mathtt{of/lapp\ P}_2\ \mathtt{P}_1, H)$, where for each $\mathtt{y}$ in $\mathrm{Domain}(\Delta_1, \Delta_2)$,

$$H(\mathtt{y}) \stackrel{\text{def}}{=} \begin{cases}\mathtt{linear/lapp1}\ (H_1(\mathtt{y})) & (\text{if } \mathtt{y} \in S_1)\\ \mathtt{linear/lapp2}\ (H_2(\mathtt{y})) & (\text{if } \mathtt{y} \in S_2)\end{cases}$$

- Et cetera.

It remains to show that $\ulcorner -\urcorner$ is a bijection. To do so, we exhibit an inverse $\llcorner -\lrcorner$. The interesting cases are those that split the context. We give the application case as an example.

Suppose $(\mathtt{of/lapp\ P}'_2\ \mathtt{P}'_1, H')$ is an encoding structure for $\Gamma;\Delta \vdash O : B'$. Then O has the form $M'N'$, and $\ulcorner\Gamma;\Delta^{\urcorner} \vdash_{LF} \mathtt{P}'_1 : \mathtt{of}\ulcorner M'\urcorner\ulcorner A' \multimap B'\urcorner$, and $\ulcorner\Gamma;\Delta^{\urcorner} \vdash_{LF} \mathtt{P}'_2 : \mathtt{of}\ulcorner N'\urcorner\ulcorner A'\urcorner$.

We must sort Δ into two pieces. Define:

$$\begin{array}{l}\Delta_1 = \{(\mathtt{y}{:}C) \in \Delta \mid \exists \mathtt{R}.H'(\mathtt{y}) = \mathtt{linear/lapp1\ R}\}\\ \Delta_2 = \{(\mathtt{y}{:}C) \in \Delta \mid \exists \mathtt{R}.H'(\mathtt{y}) = \mathtt{linear/lapp2\ R}\}\\ H'_1 = \{\mathtt{y} \mapsto \mathtt{R} \mid H'(\mathtt{y}) = \mathtt{linear/lapp1\ R}\}\\ H'_2 = \{\mathtt{y} \mapsto \mathtt{R} \mid H'(\mathtt{y}) = \mathtt{linear/lapp2\ R}\}\end{array}$$

Note that $\Delta = \Delta_1, \Delta_2$. Also note that no variable in Δ_1 appears free in N' or vice versa. Therefore it is easy to show that no assumption in $\ulcorner\Delta_1\urcorner$ appears free in $\mathtt{P}'_2$ and vice versa. Hence[4] $\ulcorner\Gamma;\Delta_1^{\urcorner} \vdash_{LF} \mathtt{P}'_1 : \mathtt{of}\ulcorner M'\urcorner\ulcorner A' \multimap B'\urcorner$ and $\ulcorner\Gamma;\Delta_2^{\urcorner} \vdash_{LF} \mathtt{P}'_2 : \mathtt{of}\ulcorner N'\urcorner\ulcorner A'\urcorner$. Also, $\mathrm{Domain}(H'_i) = \mathrm{Domain}(\Delta_i)$.

Therefore $(\mathtt{P}'_1, H'_1)$ is an encoding structure for $\Gamma;\Delta_1 \vdash M' : A' \multimap B'$ and $(\mathtt{P}'_2, H'_2)$ is an encoding structure for $\Gamma;\Delta_2 \vdash N' : A'$. Let $\nabla_i = \llcorner(\mathtt{P}'_i, H'_i)\lrcorner$. Then ∇_1 is a derivation of $\Gamma;\Delta_1 \vdash M' : A' \multimap B'$ and ∇_2 is a derivation of $\Gamma;\Delta_2 \vdash N' : A'$. So let $\llcorner(\mathtt{of/lapp\ P}'_2\ \mathtt{P}'_1, H')\lrcorner$ be the derivation:

$$\dfrac{\begin{array}{cc}\nabla_1 & \nabla_2\\ \vdots & \vdots\\ \Gamma;\Delta_1 \vdash M' : A' \multimap B' & \Gamma;\Delta_2 \vdash N' : A'\end{array}}{\Gamma;(\Delta_1, \Delta_2) \vdash M'N' : B'}$$

We can show, by induction over LF canonical forms, that $\llcorner -\lrcorner$ is fully defined over encoding structures. It is easy to verify that $\ulcorner -\urcorner$ and $\llcorner -\lrcorner$ are inverses. Therefore $\ulcorner -\urcorner$ is bijective. □

When the linear context is empty, the H portion of an encoding structure is empty, and we recover the usual notion of adequacy:

Corollary 2.7 *There exists a bijection between derivations of the judgement* $\Gamma;\epsilon \vdash M : A$ *and LF canonical forms* $\mathtt{P}$ *such that* $\ulcorner\Gamma\urcorner \vdash \mathtt{P} : \mathtt{of}\ulcorner M\urcorner\ulcorner A\urcorner$.

2.2 Metatheory

To demonstrate the practicality of our encoding, we proved the subject reduction theorem in Twelf. We give the definition of reduction in Figure 2. Reduction is encoded with the judgement:

```
reduce : term -> term -> type.
```

We will not discuss the encoding of reduction and its adequacy, as they are standard.

We prove subject reduction by a series of four metatheorems. To make the development more accessible to readers not familiar with Twelf's logic programming notation for proofs, we give those metatheorems in English.

[4] This fact, that non-appearing variables may be omitted from the context, requires a strengthening lemma for LF that is proved by Harper and Pfenning [7, Theorem 6.6].

$$\frac{}{(\lambda x.M)N \longrightarrow [N/x]M} \qquad \frac{}{\mathsf{let}\, x \otimes y = M \otimes N \,\mathsf{in}\, O \longrightarrow [M,N/x,y]O} \qquad \frac{}{\pi_1\langle M,N\rangle \longrightarrow M} \qquad \frac{}{\pi_2\langle M,N\rangle \longrightarrow N}$$

$$\frac{}{\mathsf{case}(\mathsf{in}_1 M, x.N_1, x.N_2) \longrightarrow [M/x]N_1} \qquad \frac{}{\mathsf{case}(\mathsf{in}_2 M, x.N_1, x.N_2) \longrightarrow [M/x]N_2} \qquad \frac{}{\mathsf{let} * = * \,\mathsf{in}\, M \longrightarrow M}$$

$$\frac{}{\mathsf{let}\,!x = !M \,\mathsf{in}\, N \longrightarrow [M/x]N} \qquad \frac{M \longrightarrow M'}{\lambda x.M \longrightarrow \lambda x.M'} \qquad \frac{M \longrightarrow M' \quad N \longrightarrow N'}{MN \longrightarrow M'N'} \qquad \frac{M \longrightarrow M' \quad N \longrightarrow N'}{M \otimes N \longrightarrow M' \otimes N'}$$

$$\frac{M \longrightarrow M' \quad N \longrightarrow N'}{\mathsf{let}\, x \otimes y = M \,\mathsf{in}\, N \longrightarrow \mathsf{let}\, x \otimes y = M' \,\mathsf{in}\, N'} \qquad \frac{M \longrightarrow M' \quad N \longrightarrow N'}{\langle M,N\rangle \longrightarrow \langle M',N'\rangle} \qquad \frac{M \longrightarrow M'}{\pi_1 M \longrightarrow \pi_1 M'} \qquad \frac{M \longrightarrow M'}{\pi_2 M \longrightarrow \pi_2 M'}$$

$$\frac{M \longrightarrow M'}{\mathsf{in}_1 M \longrightarrow \mathsf{in}_1 M'} \qquad \frac{M \longrightarrow M'}{\mathsf{in}_2 M \longrightarrow \mathsf{in}_2 M'} \qquad \frac{M \longrightarrow M' \quad N_1 \longrightarrow N_1' \quad N_2 \longrightarrow N_2'}{\mathsf{case}(M, x.N_1, x.N_2) \longrightarrow \mathsf{case}(M', x.N_1', x.N_2')} \qquad \frac{M \longrightarrow M' \quad N \longrightarrow N'}{\mathsf{let} * = M \,\mathsf{in}\, N \longrightarrow \mathsf{let} * = M' \,\mathsf{in}\, N'}$$

$$\frac{M \longrightarrow M'}{\mathsf{any}\, M \longrightarrow \mathsf{any}\, M'} \qquad \frac{M \longrightarrow M'}{!M \longrightarrow !M'} \qquad \frac{M \longrightarrow M' \quad N \longrightarrow N'}{\mathsf{let}\,!x = M \,\mathsf{in}\, N \longrightarrow \mathsf{let}\,!x = M' \,\mathsf{in}\, N'} \qquad \frac{}{M \longrightarrow M}$$

Figure 2: Linear logic reduction

Lemma 2.8 (Composition of linearity) *Suppose the ambient context is made up of bindings of the form* `x:term` *(and other bindings not subordinate*[5] *to* `linear`*). If* `linear ([x] M1 x)` *and* `linear ([x] M2 x)` *are derivable, then* `linear ([x] M1 (M2 x))` *is derivable.*

The next lemma is usually glossed over in proofs on paper:

Lemma 2.9 (Reduction of closed terms) *Suppose the ambient context is made up of bindings of the form* `x:term` *(and other bindings not subordinate to* `reduce`*). If* `({x:term} reduce M1 (M2 x))` *is derivable, then there exists* `M2':term` *such that* `M2 = ([_] M2')`*.*

Lemma 2.10 (Subject reduction for `linear`) *Suppose the ambient context is made up of bindings of the form* `x:term,dx:of x A` *(and other bindings not subordinate to* `reduce` *or* `of`*). If* `({x} reduce (M x) (M' x))` *and* `({x} of x A -> of (M x) B)` *and* `linear ([x] M x)` *are derivable, then* `linear ([x] M' x)` *is derivable.*

Proof Sketch

By induction on the first derivation. Cases involving substitution (most of the beta-reduction cases) use Lemma 2.8. Multiple-subterm compatibility cases use Lemma 2.9 to show that reduction of subterms not mentioning a linear variable will not create such a reference.

Theorem 2.11 (Subject reduction for `of`) *Suppose the ambient context is made up of bindings of the form* `x:term,dx:of x A` *(and other bindings not subordinate to* `reduce` *or* `of`*). If* `reduce M M'` *and* `of M T` *are derivable, then* `of M' T` *is derivable.*

Proof Sketch

By induction on the first derivation. Cases with linearity premises (`reduce/llam`, `reduce/lett`, and `reduce/case`) use Lemma 2.10 to show that the linearity premises are preserved by reduction.

[5] "Subordinate" is a term of art in Twelf. Informally, `s` is subordinate to `t` if `s` can contribute to `t`. More precisely, a type family `s` is subordinate to an type family `t` if there exist types `S` and `T` belonging to `s` and `t` such that objects of type `S` can appear within objects of type `T` [17]. If `s` is *not* subordinate to `t`, then assumptions whose types belong to `s` can be ignored while considering `t`.

`tp : type.`	A ::=
`...`	...
~~`atomic : atom -> tp.`~~	~~a~~
`const : constant -> term -> tp.`	$c(M)$
`pi : tp -> (term -> tp) -> tp.`	$\Pi x{:}A.B$
`term : type.`	M ::=
`...`	...
`ulam : (term -> term) -> term.`	$\lambda^! x.M$
`uapp : term -> term -> term.`	$M \circledcirc M$

Figure 3: Linear logic syntax (dependently typed)

Corollary 2.12 *If* $\Gamma;\Delta \vdash M : A$ *and* $M \longrightarrow M'$ *then* $\Gamma;\Delta \vdash M' : A$*.*

Proof

Immediate from Subject Reduction and Adequacy.

3 Dependently Typed Linear Logic

Adding dependent types to linear logic is straightforward syntactically. The revised syntax is shown in Figure 3. We delete atomic propositions, and replace them with constants that take a single term parameter. (That parameter may be a unit or tuple, which provides implicit support for zero or multiple parameters.)

In the static semantics, a new wrinkle arises. Now that terms can appear within types, the typing rules must ensure that linear variables are not used within types. However, a variable can appear within a term's type without appearing in the term itself. This is obvious because our lambda abstractions are unlabelled, but it would still be the case even if all bindings were labeled with types. This is because of the equivalence rule:

$$\frac{\Gamma;\Delta \vdash M : A \quad \Gamma \vdash A' \,\mathsf{type} \quad A \equiv_\beta A'}{\Gamma;\Delta \vdash M : A'}$$

Using the equivalence rule, a term's type can mention any variable in scope.

One solution to this problem is to make linearity a judgement over typing derivations, rather than over proof terms. However, that would make linearity a dependently typed meta-judgement, which would be too cumbersome to work with in practice. It is better to maintain `linear` as a judgement over proof terms.

Instead, we change our view of unrestricted variables. In non-dependently typed linear logic, we viewed unrestrictedness as merely the absence of a linearity restriction. Now we will view unrestrictedness as conferring an *affirmative capability;* specifically, the capability to appear within types.

We add a new judgement `unrest` that applies to unrestricted variables. We extend that judgement to terms by saying that a term is unrestricted if all its free variables are unrestricted:

```
unrest : term -> type.

unrest/llam : unrest (llam ([x] M x))
               <- ({x} unrest x
                    -> unrest (M x)).

unrest/lapp : unrest (lapp M N)
               <- unrest M
               <- unrest N.
...
```

Note that, within the `unrest` judgement, all bound variables are taken to be unrestricted, even linear ones.

Only unrestricted terms are permitted to serve as the parameter to a constant. On paper, this is written

$$\frac{c : A \to \mathsf{type} \quad \Gamma; \epsilon \vdash M : A}{\Gamma \vdash c(M)\ \mathsf{type}}$$

where we assume some pre-specified collection of axioms of the form $c : A \to \mathsf{type}$. In our encoding, the well-formedness judgement for types is `wf : tp -> type`. The constant rule is written:

```
wf/const : wf (const C M)
            <- cparam C A
            <- of M A
            <- unrest M.
```

We assume there exists a unique `cparam` rule for each axiom $c : A \to \mathsf{type}$. The remaining `wf` rules are uninteresting (but note that the rule for `pi` introduces an unrestricted variable).

Our existing typing rules must be altered in two ways. First, now that types can be ill-formed, several rules must add a `wf` premise. This is straightforward. Second, the rules for the exponential must be rewritten to use the `unrest` judgement:

```
of/bang : of (bang M) (! A)
           <- of M A
           <- unrest M.

of/letb : of (letb M ([x] N x)) B
           <- of M (! A)
           <- ({x} of x A
                -> unrest x -> of (N x) B)
           <- wf B.
```

We also have the new rules for unrestricted functions and application:

$$\frac{\Gamma \vdash A\ \mathsf{type} \quad (\Gamma, x{:}A); \Delta \vdash M : B}{\Gamma; \Delta \vdash (\lambda^{!} x.M) : \Pi x{:}A.B}$$

$$\frac{\Gamma; \Delta \vdash M : \Pi x{:}A.B \quad \Gamma; \epsilon \vdash N : A}{\Gamma; \Delta \vdash M \mathbin{@} N : [N/x]B}$$

```
of/ulam : of (ulam ([x] M x)) (pi A ([x] B x))
           <- wf A
           <- ({x} of x A
                -> unrest x -> of (M x) (B x)).

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

linear/ulam : linear ([y] ulam ([x] M y x))
               <- ({x} linear ([y] M y x)).

linear/uapp : linear ([x] uapp (M x) N)
               <- linear ([x] M x).
```

And finally equivalence:

```
of/equiv : of M A'
            <- of M A
            <- wf A'
            <- equiv A A'.
```

The addition of dependent types complicates the proof of subject reduction in a number of ways, but nearly all are orthogonal to linearity. One issue that does relate to linearity is we require one additional lemma to show that unrestrictedness is preserved by reduction:

Lemma 3.1 (Subject reduction for `unrest`) *Suppose the ambient context is made up of bindings of the form* `x:term,ex:unrest x` *and bindings of the form* `x:term` *(and other bindings not subordinate to* `reduce` *or* `unrest`*). If* `reduce M M'` *and* `unrest M` *are derivable, then* `unrest M'` *is derivable.*

3.1 Adequacy

Adequacy for dependently typed linear logic proceeds in much the same fashion as before. We must make four changes. First, we revise syntactic adequacy of types, now that types are not closed:

Theorem 3.2 (Syntactic adequacy)

1. *Let S be a set of variables and let Type_S be the set of linear logic types whose free variables are contained in S. Then there exists a bijection $\ulcorner - \urcorner$ between Type_S and LF canonical forms* `P` *such that $\ulcorner S \urcorner \vdash_{LF}$* `P : tp`*. Moreover, $\ulcorner - \urcorner$ respects substitution: $\ulcorner [M/\mathtt{x}]A \urcorner = [\ulcorner M \urcorner/\mathtt{x}]\ulcorner A \urcorner$.*
2. *Let S be a set of variables and let Term_S be the set of linear logic terms whose free variables are contained in S. Then there exists a bijection $\ulcorner - \urcorner$ between Term_S and LF canonical forms* `P` *such that $\ulcorner S \urcorner \vdash_{LF}$* `P : term`*. Moreover, $\ulcorner - \urcorner$ respects substitution: $\ulcorner [M/\mathtt{x}]N \urcorner = [\ulcorner M \urcorner/\mathtt{x}]\ulcorner N \urcorner$.*

Second, we define a translation for unrestricted contexts:

$$\begin{aligned}&\ulcorner \mathtt{x}_1{:}A_1, \ldots, \mathtt{x}_n{:}A_n \urcorner \\ &\quad = \mathtt{x}_1{:}\mathtt{term}, \mathtt{dx}_1{:}\mathtt{of}\ \mathtt{x}_1\ \ulcorner A_1 \urcorner, \mathtt{ex}_1{:}\mathtt{unrest}\ \mathtt{x}_1 \ldots, \\ &\qquad \mathtt{x}_n{:}\mathtt{term}, \mathtt{dx}_n{:}\mathtt{of}\ \mathtt{x}_n\ \ulcorner A_n \urcorner, \mathtt{ex}_n{:}\mathtt{unrest}\ \mathtt{x}_n\end{aligned}$$

and we alter the first clause of the definition of encoding structures to read:

$$\ulcorner \Gamma \urcorner, \ulcorner \Delta \urcorner \vdash_{LF} \mathtt{P} : \mathtt{of}\ \ulcorner M \urcorner\ \ulcorner A \urcorner$$

Third, we state adequacy for typing and for well-formedness of types simultaneously:

Theorem 3.3 (Semantic adequacy)

1. *There exists a bijection $\ulcorner - \urcorner$ between derivations of the judgement $\Gamma; \Delta \vdash M : A$ and encoding structures for $\Gamma; \Delta \vdash M : A$.*
2. *There exists a bijection $\ulcorner - \urcorner$ between derivations of the judgement $\Gamma \vdash A$ type and LF canonical forms $\mathtt{P}$ such that $\ulcorner \Gamma \urcorner \vdash_{LF} \mathtt{P} : \mathtt{wf}\ \ulcorner A \urcorner$.*

Fourth, we state a new lemma to deal with `unrest` derivations:

Lemma 3.4

1. *Suppose $\Gamma; \Delta \vdash M : A$. Then there exists a unique LF canonical form $\mathtt{P}$ such that $\ulcorner \Gamma, \Delta \urcorner \vdash_{LF} \mathtt{P} : \mathtt{unrest}\ \ulcorner M \urcorner$.*
2. *Suppose there exists an LF canonical form $\mathtt{P}$ such that $\ulcorner \Gamma \urcorner, \ulcorner \Delta \urcorner \vdash_{LF} \mathtt{P} : \mathtt{unrest}\ \ulcorner M \urcorner$. Then no variable in* Domain(Δ) *appears free in M.*
3. *Suppose there exists an LF canonical form $\mathtt{P}$ such that $\ulcorner \Gamma \urcorner, \ulcorner \Delta \urcorner \vdash_{LF} \mathtt{P} : \mathtt{wf}\ \ulcorner A \urcorner$. Then no variable in* Domain(Δ) *appears free in A.*

We give the adequacy case for unrestricted application to illustrate how Lemma 3.4 is used.

Proof Sketch of Theorem 3.3

Suppose ∇ is the derivation:

$$\dfrac{\begin{array}{cc}\nabla_1 & \nabla_2 \\ \vdots & \vdots \\ \Gamma; \Delta \vdash M : \Pi x{:}A.B & \Gamma; \epsilon \vdash N : A\end{array}}{\Gamma; \Delta \vdash M \mathbin{@} N : [N/x]B}$$

Let $\ulcorner \nabla_1 \urcorner = (\mathtt{P}_1, H_1)$ and let $\ulcorner \nabla_2 \urcorner = (\mathtt{P}_2, H_2)$. By induction $(\mathtt{P}_1, H_1)$ is an encoding structure for $\Gamma; \Delta \vdash M : \Pi x{:}A.B$ and $(\mathtt{P}_2, H_2)$ is an encoding structure for $\Gamma; \epsilon \vdash N : A$.

By Lemma 3.4, there exists a unique $\mathtt{Q}$ such that $\ulcorner \Gamma \urcorner \vdash_{LF} \mathtt{unrest}\ \ulcorner N \urcorner$. So let $\ulcorner \nabla \urcorner \stackrel{\text{def}}{=} (\mathtt{of/uapp}\ \mathtt{Q}\ \mathtt{P}_2\ \mathtt{P}_1, H)$, where for each $\mathtt{y}$ in Domain(Δ), $H(\mathtt{y}) = \mathtt{linear/uapp}\ (H_1(\mathtt{y}))$.

As an example of the definition of the inverse, suppose $(\mathtt{of/uapp}\ \mathtt{Q}'\ \mathtt{P}'_2\ \mathtt{P}'_1, H')$ is an encoding structure for $\Gamma; \Delta \vdash O : C$. Then O has the form $M' \mathbin{@} N'$ and C has the form $[N'/x]B'$. Also, $\ulcorner \Gamma \urcorner, \ulcorner \Delta \urcorner \vdash_{LF} \mathtt{P}'_1 : \mathtt{of}\ \ulcorner M' \urcorner\ \ulcorner \Pi x{:}A'.B' \urcorner$, and $\ulcorner \Gamma \urcorner, \ulcorner \Delta \urcorner \vdash_{LF} \mathtt{P}'_2 : \mathtt{of}\ \ulcorner N' \urcorner\ \ulcorner A' \urcorner$, and $\ulcorner \Gamma \urcorner, \ulcorner \Delta \urcorner \vdash_{LF} \mathtt{Q}' : \mathtt{unrest}\ \ulcorner N' \urcorner$.

Let $H'_1 = \{\mathtt{y} \mapsto \mathtt{R} \mid H'(\mathtt{y}) = \mathtt{linear/uapp}\ \mathtt{R}\}$. Then $(\mathtt{P}'_1, H'_1)$ is an encoding structure for $\Gamma; \Delta \vdash M' : \Pi x{:}A'.B'$. Let $\nabla'_1 = \llcorner (\mathtt{P}'_1, H'_1) \lrcorner$.

By Lemma 3.4, no variable in Domain(Δ) appears free in N'. (In this case—but not in some others—this fact could also be ascertained by inspection of H'.) Therefore, $\ulcorner \Gamma \urcorner \vdash_{LF} \mathtt{P}'_2 : \mathtt{of}\ \ulcorner N' \urcorner\ \ulcorner A' \urcorner$. Consequently, $(\mathtt{P}'_2, \emptyset)$ is an encoding structure for $\Gamma; \epsilon \vdash N' : A'$. Let $\nabla'_2 = \llcorner (\mathtt{P}'_2, \emptyset) \lrcorner$.

Then let $\llcorner (\mathtt{of/uapp}\ \mathtt{Q}'\ \mathtt{P}'_2\ \mathtt{P}'_1, H') \lrcorner$ be the derivation:

$$\dfrac{\begin{array}{cc}\nabla'_1 & \nabla'_2 \\ \vdots & \vdots \\ \Gamma; \Delta \vdash M' : \Pi x{:}A'.B' & \Gamma; \epsilon \vdash N' : A'\end{array}}{\Gamma; \Delta \vdash M' \mathbin{@} N' : [N'/x]B'}$$

Since $\mathtt{Q}'$ is uniquely determined by Lemma 3.4, it is easy to verify that $\ulcorner - \urcorner$ and $\llcorner - \lrcorner$ are inverses. □

4 Modal Logic

There are (at least) two ways to specify modal logic. One is using an explicit notion of Kripke worlds and accessibility [16]. Such a formulation does not behave as a substructural logic (in that all assumptions are available throughout their scope) and can be encoded in LF without difficulty [8]. A second, which we consider here, is judgemental modal logic [11].

Judgemental modal logic distinguishes between two sorts of assumption, truth and validity. Although judgemental modal logic has no explicit notion of Kripke worlds, one can think of truth as applying to only the current world, and validity as applying to all worlds. Consequently, the introduction rule for $\Box A$, which internalizes validity, must require that no truth assumptions are used.

This is accomplished with the rule:

$$\dfrac{\Gamma; \epsilon \vdash M : A}{\Gamma; \Delta \vdash \mathsf{box}\, M : \Box A}$$

Here, Γ is the validity context and Δ is the truth context. Whatever truth assumptions exist are discarded while type checking M. Since assumptions in Δ, are unavailable in M despite being in scope, judgemental modal logic behaves as a substructural logic.

We express this restriction using a judgement reminiscent of `linear`, indicating that an assumption is used locally to the current world:

```
local : (term -> term) -> type.
```

The judgement `local([x]` $M_{\mathtt{x}}$`)` should be read as "the variable `x` is used locally (*i.e.,* not within boxes) in $M_{\mathtt{x}}$."

The syntax of modal logic is given in Figure 4. In the interest of brevity, we omit discussion of the possibility modality here. A treatment of possibility appears in the full Twelf development.

Variables The rules for variables allow the use of any variable in the context:

$$\dfrac{\Delta(x) = A}{\Gamma; \Delta \vdash x : A} \qquad \dfrac{\Gamma(x) = A}{\Gamma; \Delta \vdash x : A}$$

As usual, there is no typing rule for variables in the encoding, but there are two locality rules. First, `x` is local in `x`:

```
tp : type.                    A  ::=

atomic : atom -> tp.                a
arrow  : tp -> tp -> tp.          | A → A
box    : tp -> tp.               | □A

term : type.                  M  ::=
                                    x
lam    : (term -> term) -> term.  | λx.M
app    : term -> term -> term.    | M M
bx     : term -> term.            | box M
letbx  : term -> (term -> term) -> term.
                                  | let box x = M in M
```

Figure 4: Modal logic syntax

```
local/var : local ([x] x).
```

Second, we wish to say that `x` is local in every variable (truth or validity) other than `x`. The easiest way to express this is to generalize to all terms `M` that do not contain `x`:

```
local/closed : local ([x] M).
```

Implication The introduction rule for implication is:

$$\frac{\Gamma; (\Delta, x{:}A) \vdash M : B}{\Gamma; \Delta \vdash \lambda x.M : A \to B}$$

This is encoded using two rules, reminiscent of the ones for linear implication:

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

local/lam : local ([y] lam ([x] M y x))
              <- ({x} local ([y] M y x)).
```

The function's argument is a truth assumption, so it must be used locally in the body.

The elimination rule for implication is straightforward:

$$\frac{\Gamma; \Delta \vdash M : A \to B \quad \Gamma; \Delta \vdash N : A}{\Gamma; \Delta \vdash MN : B}$$

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

local/app : local ([x] app (M x) (N x))
              <- local ([x] M x)
              <- local ([x] N x).
```

Necessity Recall the introduction rule for necessity:

$$\frac{\Gamma; \epsilon \vdash M : A}{\Gamma; \Delta \vdash \mathsf{box}\, M : \Box A}$$

This is encoded with the single rule:

```
of/bx : of (bx M) (box A)
          <- of M A.
```

The important thing here is the absence of any locality rule for `bx`. The only way to show that a variable is local in `(bx M)` is using the `local/closed` rule, which requires that the variable not appear in `M`, as desired.

The elimination rule for necessity is:

$$\frac{\Gamma; \Delta \vdash M : \Box A \quad (\Gamma, x{:}A); \Delta \vdash N : C}{\Gamma; \Delta \vdash \mathsf{let\ box}\, x = M \,\mathsf{in}\, N : C}$$

This is encoded using two rules:

```
of/letbx
  : of (letbx M ([x] N x)) B
     <- of M (box A)
     <- ({x} of x A -> of (N x) B).

local/letbx
  : local ([x] letbx (M x) ([y] N x y))
            <- local ([x] M x)
            <- ({y} local ([x] N x y)).
```

Since the variable introduced by letbx is a validity assumption, we do not check that it is local in the body.

Metatheory Subject reduction for modal logic follows the same development as for linear logic in Section 2.2, with `local` standing in for `linear`. One lemma must be generalized: since local variables can appear multiple times in modal logic, composition of locality must allow the local variable to appear (locally) in the scope of substitution (`M1` below), as well as in the substitutend (`M2` below):

Lemma 4.1 (Composition of locality) *Suppose the ambient context is made up of bindings of the form* `x:term` *(and other bindings not subordinate to* `local`*). If* `({y} local ([x] M1 x y))` *and* `({x} local ([y] M1 x y)` *and* `local ([x] M2 x)` *are derivable, then* `local ([x] M1 x (M2 x))` *is derivable.*

4.1 Adequacy

Syntactic adequacy for modal logic is again standard:

Definition 4.2 *Translation of variable sets is defined:*

$$\ulcorner \{\mathtt{x}_1, \ldots, \mathtt{x}_n\} \urcorner = \mathtt{x}_1{:}\mathtt{term}, \ldots, \mathtt{x}_n{:}\mathtt{term}$$

Theorem 4.3 (Syntactic adequacy)

1. *Let Type be the set of modal logic types. Then there exists a bijection* $\ulcorner - \urcorner$ *between Type and LF canonical forms* `P` *such that* $\vdash_{LF}$ `P : tp`*. (Variables cannot appear within types, so there is no substitution to respect.)*
2. *Let* S *be a set of variables and let* $Term_S$ *be the set of modal logic terms whose free variables are contained in* S*. Then there exists a bijection* $\ulcorner - \urcorner$ *between* $Term_S$ *and LF canonical forms* `P` *such that* $\ulcorner S \urcorner \vdash_{LF}$ `P : term`*. Moreover,* $\ulcorner - \urcorner$ *respects substitution:* $\ulcorner [M/\mathtt{x}]N \urcorner = [\ulcorner M \urcorner / \mathtt{x}] \ulcorner N \urcorner$*.*

Semantic adequacy again encounters a challenge; this time the opposite problem from the one we saw with linear logic. In the encoding of linear logic there were too few typing derivations; here there are too many.

The problem lies in the `local` judgement. Unlike `linear`, which expressed a property that could be satisfied in many

ways, `local` expresses a fact that essentially can be satisfied in only one way, by the variable not appearing in any boxes. In this regard, `local` is more like `unrest` than `linear`. However, unlike `unrest`, derivations of `local` are not unique.

The problem stems from the fact that the `local/closed` rule can apply to terms that also have another rule. For example, suppose `M` and `N` are closed terms. Then `local ([x] app M N)` has at least two derivations: `local/closed` and `(local/app local/closed local/closed)`.

One solution to the problem would be to restrict `local/closed` to variables (and add another rule for closed boxes). This would ensure that `local` derivations are unique (like `unrest` derivations). We could impose the restriction by creating a judgement (say, `var`) to identify variables, and then rewrite the `local/closed` rule as:

```
local/closed-varonly : local ([y] X)
                         <- var X.
```

However, this solution has a significant shortcoming; the substitution lemma would no longer be a free consequence of higher-order representation. Under such a regime, variable assumptions would take the form `({x:term} of x A -> var x -> ...whatever...)`. Consequently, we would only obtain substitution for free when the substitutend possesses a `var` derivation; that is, when the substitutend is another variable. The general substitution lemma would have to be proved and used explicitly.

A better solution is to rephrase adequacy to quotient out the excess derivations:

Definition 4.4 *Translation of contexts is defined:*

$$\ulcorner \mathtt{x}_1{:}A_1, \ldots, \mathtt{x}_n{:}A_n \urcorner = \begin{array}{l} \mathtt{x}_1{:}\mathtt{term}, \mathtt{dx}_1{:}\mathtt{of}\ \mathtt{x}_1 \ulcorner A_1 \urcorner, \ldots, \\ \mathtt{x}_n{:}\mathtt{term}, \mathtt{dx}_n{:}\mathtt{of}\ \mathtt{x}_n \ulcorner A_n \urcorner \end{array}$$

Definition 4.5 *Let $\cong$ be the least congruence over LF canonical forms such that $\mathtt{P} \cong \mathtt{P}'$ for any $\mathtt{P}, \mathtt{P}'$: `local` F (where F : `term -> term`).*

An encoding structure for $\Gamma; \Delta \vdash M : A$ *is a nonempty equivalence class (under $\cong$) of LF canonical forms* `P` *such that:*

- $\ulcorner \Gamma, \Delta \urcorner \vdash_{LF} \mathtt{P} : \mathtt{of} \ulcorner P \urcorner \ulcorner A \urcorner$, *and*
- *For every* `y` *in* Domain(Δ), *there exists an LF canonical form* $\mathtt{Q}_\mathtt{y}$ *such that* $\ulcorner S_\mathtt{y} \urcorner \vdash_{LF} \mathtt{Q}_\mathtt{y}$: `local ([y:term]` $\ulcorner M \urcorner$`)`, *where* $S_\mathtt{y} = \text{Domain}(\Gamma, \Delta) \setminus \{\mathtt{y}\}$.

Theorem 4.6 (Semantic adequacy) *There exists a bijection between derivations of the judgement $\Gamma; \Delta \vdash M : A$ and encoding structures for $\Gamma; \Delta \vdash M : A$.*

Proof Sketch

We give one case in each direction, by way of example. Suppose ∇ is the derivation:

$$\dfrac{\begin{array}{c}\nabla_1 \\ \vdots \\ \Gamma; (\Delta, x{:}A) \vdash M : B\end{array}}{\Gamma; \Delta \vdash \lambda x.M : A \to B}$$

Let $\ulcorner \nabla_1 \urcorner = (\mathtt{P}_1, H_1)$. By induction, $(\mathtt{P}_1, H_1)$ is an encoding structure for $\Gamma; (\Delta, x{:}A) \vdash M : B$, so:

$$\ulcorner \Gamma, \Delta \urcorner, \mathtt{x}{:}\mathtt{term}, \mathtt{dx}{:}\mathtt{of}\ \mathtt{x} \ulcorner A \urcorner \vdash_{LF} \mathtt{P}_1 : \mathtt{of} \ulcorner M \urcorner \ulcorner B \urcorner$$

and, for every $\mathtt{y} \in \text{Domain}(\Delta, x{:}A)$, there exists a $\mathtt{Q}_\mathtt{y}$ such that:

$$\ulcorner \text{Domain}(\Gamma, \Delta, x{:}A) \setminus \{\mathtt{y}\} \urcorner \vdash_{LF} \mathtt{Q}_\mathtt{y} : \mathtt{local\ ([y:term]} \ulcorner M \urcorner\mathtt{)}$$

In particular, $\mathtt{x} \in \text{Domain}(\Delta, x{:}A)$, so:

$$\ulcorner \text{Domain}(\Gamma, \Delta) \urcorner \vdash_{LF} \mathtt{Q}_\mathtt{x} : \mathtt{local\ ([x:term]} \ulcorner M \urcorner\mathtt{)}$$

Therefore:

$$\ulcorner \Gamma, \Delta \urcorner \vdash_{LF} \mathtt{of/lam}\ \mathtt{Q}_\mathtt{x}\ \mathtt{P}_1 : \mathtt{of} \ulcorner \lambda x.M \urcorner \ulcorner A \to B \urcorner$$

Also, for every $\mathtt{y} \in \text{Domain}(\Delta)$,

$$\ulcorner \text{Domain}(\Gamma, \Delta) \setminus \{\mathtt{y}\} \urcorner \vdash_{LF} \mathtt{local/lam\ ([x:term]}\ \mathtt{Q}_\mathtt{y}\mathtt{)} : \mathtt{local\ ([y:term]} \ulcorner \lambda x.M \urcorner\mathtt{)}$$

So let $\ulcorner \nabla \urcorner$ be the equivalence class containing `of/lam` $\mathtt{Q}_\mathtt{x}\ \mathtt{P}_1$, which is an encoding structure for $\Gamma; \Delta \vdash \lambda x.M : A \to B$.

As an example of the definition of the inverse, suppose `(of/bx P′)` belongs to an encoding structure for $\Gamma; \Delta \vdash O : C$. Then O has the form $\mathsf{box}\, M'$, and C has the form $\Box A'$. Also, $\ulcorner \Gamma, \Delta \urcorner \vdash_{LF} \mathtt{P}' : \mathtt{of} \ulcorner M' \urcorner \ulcorner A' \urcorner$.

Further, for every `y` in Domain(Δ), there exists $\mathtt{Q}_\mathtt{y}$ such that $\ulcorner S_\mathtt{y} \urcorner \vdash_{LF} \mathtt{Q}_\mathtt{y}$: `local ([y:term] bx` $\ulcorner M' \urcorner$`)`, where $S_\mathtt{y} = \text{Domain}(\Gamma, \Delta) \setminus \{\mathtt{y}\}$. Each $\mathtt{Q}_\mathtt{y}$ must be `linear/closed`, so no `y` in Domain(Δ) appears in M'. Therefore $\ulcorner \Gamma \urcorner \vdash_{LF} \mathtt{P}' : \mathtt{of} \ulcorner M' \urcorner \ulcorner A' \urcorner$.

The second criterion of encoding structures is vacuously satisfied for an empty truth context, so $\mathtt{P}'$ belongs to an encoding structure for $\Gamma; \epsilon \vdash M' : A'$. Let $\nabla' = \llcorner \mathtt{P}' \lrcorner$.

Then let $\llcorner \mathtt{of/bx}\ \mathtt{P}' \lrcorner$ be the derivation:

$$\dfrac{\begin{array}{c}\nabla' \\ \vdots \\ \Gamma; \epsilon \vdash M' : A'\end{array}}{\Gamma; \Delta \vdash \mathsf{box}\, M' : \Box A'}$$

It is easy to verify that, for the appropriate ∇ and $\mathtt{P}$, $\llcorner \ulcorner \nabla \urcorner \lrcorner = \nabla$ and $\ulcorner \llcorner \mathtt{P} \lrcorner \urcorner \cong \mathtt{P}$. Therefore $\ulcorner - \urcorner$ and $\llcorner - \lrcorner$ are inverses. □

5 Conclusion

The Logical Framework is not only (nor even primarily) a type theory. More importantly, it is a methodology for representing deductive systems using higher-order representation of syntax and semantics, and a rigorous account of adequacy. Where applicable, the LF methodology provides a powerful and elegant tool for formalizing programming languages and logics.

There are two reasons it might not apply. First, limitations of existing tools for LF, such as Twelf, might prevent one from carrying out the desired proofs once a system were encoded in LF. Second, there might be an inherent problem representing the desired deductive system adequately using a higher-order representation. When a language cannot be cleanly represented in a higher-order fashion, it often indicates that something about the language is suspect, such as an incorrect (or at least nonstandard) notion of binding and/or scope.

In some cases, however, languages with unconventional notions of binding or scope are nevertheless sensible. Substructural logics are probably the most important example. In this paper, we show that many substructural logics can be given a clean higher-order representation by isolating its "substructuralness" (*e.g.,* linearity or locality) and expressing that as a judgement over proof terms.

Our strategy applies to other substructural logics as well. For example, affine logic and strict logic can each be encoded along very similar lines as linear logic. We conjecture that contextual modal logic [9] is encodable along similar lines as judgemental modal logic. This is a good avenue for future work. The logic of bunched implications [10] is another.

On the other hand, since our method relies on enforcing "substructuralness" on an assumption-by-assumption basis, there are some substructural logics it does not support, such as ordered logic [15, 14]. In ordered logic, the context is taken to be ordered and assumptions must be processed in order. We cannot enforce this restriction on assumptions independently, as the very nature of the restriction is that assumptions are not independent. The usability of one assumption can depend on the disposition of every other assumption in scope.

References

[1] Arnon Avron, Furio Honsell, and Ian A. Mason. Using typed lambda calculus to implement formal systems on a machine. Technical Report ECS-LFCS-87-31, Department of Computer Science, University of Edinburgh, July 1987.

[2] Arnon Avron, Furio Honsell, and Ian A. Mason. An overview of the Edinburgh Logical Framework. In Graham Birtwistle and P. A. Subrahmanyam, editors, *Current Trends in Hardware Verification and Automated Theorem Proving*. Springer, 1989.

[3] Brian Aydemir, Arthur Charguéraud, Benjamin C. Pierce, Randy Pollack, and Stephanie Weirich. Engineering formal metatheory. In *Thirty-Fifth ACM Symposium on Principles of Programming Languages*, San Francisco, California, January 2008.

[4] Iliano Cervesato and Frank Pfenning. A linear logical framework. In *Eleventh IEEE Symposium on Logic in Computer Science*, pages 264–275, New Brunswick, New Jersey, July 1996.

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

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

[7] Robert Harper and Frank Pfenning. On equivalence and canonical forms in the LF type theory. *ACM Transactions on Computational Logic*, 6(1), 2005.

[8] Tom Murphy, VII. *Modal Types for Mobile Code*. PhD thesis, Carnegie Mellon University, School of Computer Science, Pittsburgh, Pennsylvania, May 2008.

[9] Aleksandar Nanevski, Frank Pfenning, and Brigitte Pientka. A contextual modal type theory. *ACM Transactions on Computational Logic*, 9(3), 2008.

[10] Peter W. O'Hearn and David J. Pym. The logic of bunched implications. *Bulletin of Symbolic Logic*, 5(2), 1999.

[11] Frank Pfenning and Rowan Davies. A judgmental reconstruction of modal logic. *Mathematical Structures in Computer Science*, 11(4):511–540, 2001.

[12] Frank Pfenning and Conal Elliott. Higher-order abstract syntax. In *1988 SIGPLAN Conference on Programming Language Design and Implementation*, pages 199–208, Atlanta, Georgia, June 1988.

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

[14] Jeff Polakow. *Ordered Linear Logic and Applications*. PhD thesis, Carnegie Mellon University, School of Computer Science, Pittsburgh, Pennsylvania, August 2001.

[15] Jeff Polakow and Frank Pfenning. Natural deduction for intuitionistic non-commutative linear logic. In *1999 International Conference on Typed Lambda Calculi and Applications*, volume 1581 of *Lecture Notes in Computer Science*, L'Aquila, Italy, April 1999. Springer.

[16] Alex Simpson. *The Proof Theory and Semantics of Intuitionistic Modal Logic*. PhD thesis, University of Edinburgh, 1994.

[17] Roberto Virga. *Higher-Order Rewriting with Dependent Types*. PhD thesis, Carnegie Mellon University, School of Computer Science, Pittsburgh, Pennsylvania, 1999.