Chapter 5

A Linear Logical Framework

[*a lot of stuff omitted here for now which is easily accessible in published papers*]

5.1 Representation of Meta-Theory

The features which make the linear logical framework so suitable for the representation of deductive systems with linear or imperative features, also make it a good candidate to represent proofs of properties of such deductive systems. If one follows the natural line of development, however, it turns out that proofs of meta-theorems are representable, but that their validity cannot be guaranteed by linear type-checking alone. Presently, there exist no good tools to verify the necessary additional properties, but their development is the subject of ongoing research.

In this section, we will examine two relatively simple examples of meta-theoretic properties and their proofs in the context of LLF encodings. We will pay particular attention to the additional checks required to verify the proofs, since they must currently be carried out by the user. The examples are soundness and completeness of sequent derivations (including cut) with respect to natural deductions. The proofs we give here is somewhat different from the ones in Chapter 2 since here we are not interested in the fine-grained analysis which relates cut-free sequent derivations to normal deductions.

We present all the LLF encodings in the concrete syntax of linear Twelf introduced in prior lectures and documented in [CP97] and Section 5.2. In particular, we will take full advantage of term reconstruction to recover the types of all free variables, which are implicitly Π-quantified over each declaration.

We begin with the encoding of the intuitionistic, propositional fragment of linear logic. Here we omit propositions $\bot$, $\neg A$, and $?A$ which require a second judgment of "possible truth".

```
%%% Propositions
o : type.                    %name o A

%%% Multiplicatives
lolli  : o -> o -> o.        % A -o B
tensor : o -> o -> o.        % A * B
one    : o.                  % 1

%%% Additives
with : o -> o -> o.          % A & B
top : o.                     % T
plus : o -> o -> o.          % A + B
zero : o.                    % 0

%%% Exponentials
imp : o -> o -> o.           % A -> B
bang : o -> o.               % ! A
```

We encode the main judgment of linear logic, $\Gamma; \Delta \vdash A$ using the standard encoding technique for hypothetical and linear hypothetical judgments. A natural deduction $\mathcal{D}$ of $\Gamma; \Delta \vdash A$ will therefore be represented as a canonical LLF object M such that $\ulcorner\Gamma\urcorner; \ulcorner\Delta\urcorner \overset{LF}{\vdash} M : \mathrm{nd}\ulcorner A\urcorner$ where

$$\begin{array}{rcl} \ulcorner . \urcorner & = & . \\ \ulcorner \Gamma, u{:}A \urcorner & = & \ulcorner\Gamma\urcorner, u{:}\mathrm{nd}\ulcorner A\urcorner \\ \ulcorner \Delta, x{:}A \urcorner & = & \ulcorner\Delta\urcorner, x{:}\mathrm{nd}\ulcorner A\urcorner \end{array}$$

With this idea it is almost possible to achieve a perfect bijection between canonical LLF objects of type $\mathrm{nd}\ulcorner A\urcorner$ and natural deductions of A.[1]

```
nd : o -> type.              %name nd D

%%% Multiplicatives

% A -o B
lolliI : (nd A -o nd B) -o nd (lolli A B).
lolliE : nd (lolli A B) -o nd A -o nd B.

% A * B
tensorI : nd A -o nd B -o nd (tensor A B).
tensorE : nd (tensor A B)
           -o (nd A -o nd B -o nd C)
           -o nd C.

% 1
```

[1]For the exception consider the two derivations of $A \multimap (\top \otimes \top)$ and their representation.

```
oneI : nd (one).
oneE : nd (one)
        -o nd C
        -o nd C.

%%% Additives

% A & B
withI : nd A & nd B -o nd (with A B).
withE1 : nd (with A B) -o nd A.
withE2 : nd (with A B) -o nd B.

% T
topI : <T> -o nd (top).
% no topE

% A + B
plusI1 : nd A -o nd (plus A B).
plusI2 : nd B -o nd (plus A B).
plusE : nd (plus A B)
         -o ((nd A -o nd C) & (nd B -o nd C))
         -o nd C.

% 0
% no zeroI
zeroE : nd (zero)
         -o <T>
         -o nd C.

%%% Exponentials

% A -> B
impI : (nd A -> nd B) -o nd (imp A B).
impE : nd (imp A B) -o nd A -> nd B.

% ! A
bangI : nd A -> nd (bang A).
bangE : nd (bang A)
         -o (nd A -> nd C)
         -o nd C.
```

Representation of the sequent calculus is discussed in some detail in the literature [CP97]. Briefly, it arises by considering two basic judgments, "*A is a hypothesis*" and "*A is the conclusion*". We represent this via two corresponding type families, left$\ulcorner A \urcorner$ for hypotheses and right$\ulcorner A \urcorner$ for the conclusion. So a sequent derivation of $\Gamma; \Delta \Longrightarrow A$ is represented by a canonical LLF object M

such that $\ulcorner\Gamma\urcorner;\ulcorner\Delta\urcorner \overset{LF}{\vdash} M : \text{right}\ulcorner A\urcorner$, where

$$\begin{array}{rcl}\ulcorner\cdot\urcorner & = & \cdot \\ \ulcorner\Gamma, u{:}A\urcorner & = & \ulcorner\Gamma\urcorner, u{:}\text{left}\ulcorner A\urcorner \\ \ulcorner\Delta, x{:}A\urcorner & = & \ulcorner\Delta\urcorner, x{:}\text{left}\ulcorner A\urcorner\end{array}$$

Unlike natural deduction, we now need to explicitly connect the left and right judgments, which is done in two dual ways. Initial sequents allow us to conclude right$\ulcorner A\urcorner$ from left$\ulcorner A\urcorner$. The rules of Cut allow us to substitute a derivation of right$\ulcorner A\urcorner$ for a hypothesis left$\ulcorner A\urcorner$. There are two forms of cut, since we might replace a linear or unrestricted hypothesis.

```
left  : o -> type.          %name left L
right : o -> type.          %name right R

%%% Initial sequents
init : (left A -o right A).

%%% Cuts
cut : right A
       -o (left A -o right C)
       -o right C.

cut! : right A
        -> (left A -> right C)
        -o right C.
```

The renaming rules are the left and right rules for each connective, cast into LLF.

```
%%% Multiplicatives

% A -o B
lolliR : (left A -o right B) -o right (lolli A B).
lolliL : right A
          -o (left B -o right C)
          -o (left (lolli A B) -o right C).

% A * B
tensorR : right A -o right B -o right (tensor A B).
tensorL : (left A -o left B -o right C)
           -o (left (tensor A B) -o right C).

% 1
oneR : right (one).
oneL : (right C)
        -o (left (one) -o right C).
```

```
%%% Additives

% A & B
withR : right A & right B -o right (with A B).
withL1 : (left A -o right C)
          -o (left (with A B) -o right C).
withL2 : (left B -o right C)
          -o (left (with A B) -o right C).

% T
topR : <T> -o right (top).
% no topL

% A + B
plusR1 : right A -o right (plus A B).
plusR2 : right B -o right (plus A B).
plusL : (left A -o right C) & (left B -o right C)
        -o (left (plus A B) -o right C).

% 0
% no zeroR
zeroL : <T> -o (left (zero) -o right C).

%%% Exponentials

% A -> B
impR : (left A -> right C) -o right (imp A C).
impL : right A
        -> (left B -> right C)
        -o (left (imp A B) -o right C).

% ! A
bangR : right A -> right (bang A).
bangL : (left A -> right C)
         -o (left (bang A) -o right C).
```

Note that LLF allows a one-by-one representation of the rules without any auxiliary judgment forms. It is this directness of encoding which makes it feasible to write out the proof of soundness and completeness of the sequent calculus with respect to natural deduction in a similar manner. We begin with the slightly easier direction of soundness.

Theorem 5.1 (Soundness of Sequent Calculus with Cut)
If $\Gamma; \Delta \Longrightarrow A$ *then* $\Gamma; \Delta \vdash A$.

Proof: By induction over the structure of the given sequent derivation. This constructive proof contains a method by which a natural deduction $\mathcal{D}$ can be constructed from a sequent derivation $\mathcal{R}$. Under the Curry-Howard isomorphism, one might expect this method to be represented as a dependently typed function

$$\mathrm{sd} : \Pi A{:}o.\ \mathrm{right}\, A \to \mathrm{nd}\, A.$$

Unfortunately, such a function cannot be defined within LLF, since it lacks the means to define functions recursively, or distinguish cases based on the possible input derivations.[2]

Instead, we represent the proof as a *higher-level judgment* sd relating the derivations $\mathcal{D}$ and $\mathcal{R}$. We do not give the awkward informal presentation of this relation, only its implementation in linear Twelf. Since hypotheses on the left of the sequent are represented by a different judgment, we also need to explicitly relate the hypotheses in the two judgments, using sd$'$.

```
sd : right A -> nd A -> type.
sd' : left A -> nd A -> type.
```

Each case in the proof corresponds to an inference rule defining this higher-level judgment.

Case: $\mathcal{R} = \dfrac{}{\Gamma; x{:}A \Longrightarrow A}\,\mathrm{I}.$

Then

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

is the corresponding derivation. In the formalization, the labels of the hypothesis A are different, but related by sd$'$. Thus we declare:

```
initSD : sd (init ^ L) X
          o- sd' L X.
```

where L stands for label of the sequent hypothesis (of type left$\ulcorner A \urcorner$) and X stands for the label of the natural deduction hypothesis (of type nd$\ulcorner A \urcorner$).

Case: $\mathcal{R} = \dfrac{\begin{matrix}\mathcal{R}_1 \\ \Gamma; \Delta_1 \Longrightarrow A\end{matrix} \qquad \begin{matrix}\mathcal{R}_2 \\ \Gamma; \Delta_2 \Longrightarrow B\end{matrix}}{\Gamma; (\Delta_1 \times \Delta_2) \Longrightarrow A \otimes B}\,\otimes\mathrm{R}.$

Then we reason:

$\mathcal{D}_1 :: (\Gamma; \Delta_1 \vdash A)$	By i.h. on $\mathcal{R}_1$
$\mathcal{D}_2 :: (\Gamma; \Delta_2 \vdash B)$	By i.h. on $\mathcal{R}_2$
$\mathcal{D} :: (\Gamma; (\Delta_1 \times \Delta_2) \vdash A \otimes B)$	By $\otimes$I from $\mathcal{D}_1$ and $\mathcal{D}_2$

[2]The lack of definition by cases and recursion is essential to obtain adequate encodings (see [SDP97])

The appeals to the induction hypothesis are implemented by "recursive calls" in the sd judgment.

```
tensorRSD : sd (tensorR ^ R1 ^ R2) (tensorI ^ D1 ^ D2)
             o- sd R1 D1
             o- sd R2 D2.
```

Case: $\mathcal{R} = \dfrac{\begin{matrix}\mathcal{R}_1\\ \Gamma; (\Delta_1, x_1{:}A, x_2{:}B) \Longrightarrow C\end{matrix}}{\Gamma; (\Delta_1, x{:}A \otimes B) \Longrightarrow C} \otimes\mathrm{L}.$

In this case we reason:

$\mathcal{D}_1 :: (\Gamma; (\Delta_1, x_1{:}A, x_2{:}B) \vdash C)$	By i.h. on $\mathcal{R}_1$
$\mathcal{X} :: (\Gamma; x{:}A \otimes B \vdash A \otimes B)$	By hypothesis x
$\mathcal{D} :: (\Gamma; (\Delta_1, x{:}A \otimes B) \vdash C)$	By $\otimes$E from $\mathcal{X}$ and $\mathcal{D}_1$

In the implementation, we have to take care to introduce the new hypotheses in the premisses, that is, the parameters x_1 and x_2. Further, we need to relate the hypotheses in the two different judgments (as indicated already in the case for initial sequents above), by using the sd$'$ judgment. We label the hypotheses in the sequent calculus with l, l_1, and l_2.

```
tensorLSD : sd (tensorL ^ R1 ^ L) (tensorE ^ X ^ D1)
             o- ({l1:left A} {l2:left B} {x1:nd A} {x2:nd B}
                   sd' l1 x1
                   -o sd' l2 x2
                   -o sd (R1 ^ l1 ^ l2) (D1 ^ x1 ^ x2))
             o- sd' L X.
```

Case: $\mathcal{R} = \dfrac{\begin{matrix}\mathcal{R}_1\\ \Gamma; \Delta \Longrightarrow A\end{matrix} \qquad \begin{matrix}\mathcal{R}_2\\ \Gamma; \Delta \Longrightarrow B\end{matrix}}{\Gamma; \Delta \Longrightarrow A \& B} \&\mathrm{R}.$

$\mathcal{D}_1 :: (\Gamma; \Delta \vdash A)$	By i.h. on $\mathcal{R}_1$
$\mathcal{D}_2 :: (\Gamma; \Delta \vdash B)$	By i.h. on $\mathcal{R}_2$
$\mathcal{D} :: (\Gamma; \Delta \vdash A \& B)$	By &I from $\mathcal{D}_1$ and $\mathcal{D}_2$

In the formalization of this case, we have to be careful to use the alternative conjunction at the meta-level as well, since the assumptions about the connection between hypotheses left$\ulcorner A \urcorner$ and nd$\ulcorner A \urcorner$ are linear and may be needed in both branches.

```
withRSD : sd (withR ^ (R1, R2)) (withI ^ (D1, D2))
           o- sd R1 D1 & sd R2 D2.
```

Case: $\mathcal{R} = \dfrac{\overset{\mathcal{R}_1}{\Gamma; (\Delta_1, x_1{:}A) \Longrightarrow C}}{\Gamma; (\Delta_1, x{:}A\&B) \Longrightarrow C}\,\&\mathrm{L}_1.$

This case is slightly more complicated than the previous ones, since we need to appeal to the substitution lemma.

$\mathcal{D}_1 :: (\Gamma; \Delta_1, x_1{:}A \vdash C)$ By i.h. on $\mathcal{R}_1$
$\mathcal{D}_2 :: (\Gamma; x{:}A\&B \vdash A)$ By $\&\mathrm{E}_1$ from hypothesis x
$\mathcal{D} :: (\Gamma; (\Delta_1, x{:}A\&B) \vdash C)$ By the substitution lemma from $\mathcal{D}_1$ and $\mathcal{D}_2$

The implementation exploits the compositionality of the representation to model the appeal to the substitution lemma by application in the LLF. We have (in the LLF representation):

$$\begin{array}{rcl}
\ulcorner\Delta_1\urcorner & \overset{LF}{\vdash} & (\hat{\lambda} x_1{:}\mathrm{nd}\ulcorner A\urcorner.\ \ulcorner\mathcal{D}_1\urcorner) : \mathrm{nd}\ulcorner A\urcorner \multimap \mathrm{nd}\ulcorner C\urcorner \\
x{:}\mathrm{nd}\,(\mathrm{with}\,\ulcorner A\urcorner\ulcorner B\urcorner) & \overset{LF}{\vdash} & \mathrm{withE}_1\,\hat{}\,x : \mathrm{nd}\ulcorner A\urcorner \\
\ulcorner\Delta_1, x{:}A\&B\urcorner & \overset{LF}{\vdash} & (\hat{\lambda} x_1{:}\mathrm{nd}\ulcorner A\urcorner.\ \ulcorner\mathcal{D}_1\urcorner)\,\hat{}\,(\mathrm{withE}_1\,\hat{}\,x) : \mathrm{nd}\ulcorner C\urcorner \\
\ulcorner\Delta_1, x{:}A\&B\urcorner & \overset{LF}{\vdash} & [(\mathrm{withE}_1\,\hat{}\,x)/x_1]\ulcorner\mathcal{D}_1\urcorner : \mathrm{nd}\ulcorner C\urcorner
\end{array}$$

In the concrete code, `D1` will be bound to $(\hat{\lambda} x_1{:}\mathrm{nd}\ulcorner A\urcorner.\ \ulcorner\mathcal{D}_1\urcorner)$, so the application in the second to the last line is written as `D1 ^ (withE1 ^ X)`.

```
withL1SD : sd (withL1 ^ R1 ^ L) (D1 ^ (withE1 ^ X))
            o- ({l1:left A} {x1:nd A}
                  sd' l1 x1 -o sd (R1 ^ l1) (D1 ^ x1))
            o- sd' L X.
```

Case: $\mathcal{R} = \dfrac{\overset{\mathcal{R}_1}{\Gamma; \Delta_1 \Longrightarrow A} \qquad \overset{\mathcal{R}_2}{\Gamma; (\Delta_2, x{:}A) \Longrightarrow C}}{\Gamma; (\Delta_1 \times \Delta_2) \Longrightarrow C}\,\mathrm{Cut}.$

$\mathcal{D}_1 :: (\Gamma; \Delta_1 \vdash A)$ By i.h. on $\mathcal{R}_1$
$\mathcal{D}_2 :: (\Gamma; (\Delta_2, x{:}A) \vdash C)$ By i.h. on $\mathcal{R}_2$
$\mathcal{D} :: (\Gamma; (\Delta_1 \times \Delta_2) \vdash C)$ By the substitution lemma from $\mathcal{D}_1$ and $\mathcal{D}_2$

The representation uses the same technique of meta-level application as the case for $\&\mathrm{R}_1$ above.

```
cutSD : sd (cut ^ R1 ^ R2) (D2 ^ D1)
         o- sd R1 D1
         o- ({l:left A} {x:nd A}
               sd' l x -o sd (R2 ^ l) (D2 ^ x)).
```

□

Among the remaining cases, the exponentials and the cut of an unrestricted hypothesis require some additional case to make sure the meta-level hypothesis are also unrestricted. Similarly, in the case for ⊤R, care must be taken so the linearity of the meta-reasoning is not violated by allowing weakening with the ⊤ of LLF. We leave it to the reader to write out these cases and relate them to the complete code given below.

```
sd : right A -> nd A -> type.
sd' : left A -> nd A -> type.

%%% Initial sequents
initSD : sd (init ^ L) X
          o- sd' L X.

%%% Cuts
cutSD : sd (cut ^ R1 ^ R2) (D2 ^ D1)
         o- sd R1 D1
         o- ({l:left A} {x:nd A}
               sd' l x -o sd (R2 ^ l) (D2 ^ x)).

cut!SD : sd (cut! R1 ^ R2) (D2 D1)
          <- sd R1 D1
          o- ({l:left A} {u:nd A}
                sd' l u -> sd (R2 l) (D2 u)).

%%% Multiplicatives

% A -o B
lolliRSD : sd (lolliR ^ R1) (lolliI ^ D1)
            o- ({l:left A} {x:nd A}
                  sd' l x -o sd (R1 ^ l) (D1 ^ x)).

lolliLSD : sd (lolliL ^ R1 ^ R2 ^ L) (D2 ^ (lolliE ^ X ^ D1))
            o- sd R1 D1
            o- ({l:left B} {x:nd B}
                  sd' l x -o sd (R2 ^ l) (D2 ^ x))
            o- sd' L X.

% A * B
tensorRSD : sd (tensorR ^ R1 ^ R2) (tensorI ^ D1 ^ D2)
             o- sd R1 D1
             o- sd R2 D2.

tensorLSD : sd (tensorL ^ R1 ^ L) (tensorE ^ X ^ D1)
             o- ({l1:left A} {l2:left B} {x1:nd A} {x2:nd B}
                   sd' l1 x1
                   -o sd' l2 x2
                   -o sd (R1 ^ l1 ^ l2) (D1 ^ x1 ^ x2))
             o- sd' L X.
```

```
% 1
oneRSD : sd (oneR) (oneI).

oneLSD : sd (oneL ^ R1 ^ L) (oneE ^ X ^ D1)
          o- sd R1 D1
          o- sd' L X.

%%% Additives

% A & B
withRSD : sd (withR ^ (R1, R2)) (withI ^ (D1, D2))
           o- sd R1 D1 & sd R2 D2.

withL1SD : sd (withL1 ^ R1 ^ L) (D1 ^ (withE1 ^ X))
            o- ({l1:left A} {x1:nd A}
                  sd' l1 x1 -o sd (R1 ^ l1) (D1 ^ x1))
            o- sd' L X.

withL2SD : sd (withL2 ^ R2 ^ L) (D2 ^ (withE2 ^ X))
            o- ({l2:left B} {x2:nd B}
                  sd' l2 x2 -o sd (R2 ^ l2) (D2 ^ x2))
            o- sd' L X.

% T
topRSD : sd (topR ^ ()) (topI ^ ())
          o- <T>.
% no topL

% A + B
plusR1SD : sd (plusR1 ^ R1) (plusI1 ^ D1)
            o- sd R1 D1.

plusR2SD : sd (plusR2 ^ R2) (plusI2 ^ D2)
            o- sd R2 D2.

plusLSD : sd (plusL ^ (R1, R2) ^ L) (plusE ^ X ^ (D1, D2))
           o- (({l1:left A} {x1:nd A}
                  sd' l1 x1 -o sd (R1 ^ l1) (D1 ^ x1))
                 & ({l2:left B} {x2:nd B}
                      sd' l2 x2 -o sd (R2 ^ l2) (D2 ^ x2)))
           o- sd' L X.

% 0
% no zeroR
zeroLSD : sd (zeroL ^ () ^ L) (zeroE ^ X ^ ())
           o- <T>
           o- sd' L X.

%%% Exponentials
```

```
% A -> B
impRSD : sd (impR ^ R1) (impI ^ D1)
          o- ({l:left A} {u:nd A}
                sd' l u -> sd (R1 l) (D1 u)).

impLSD : sd (impL R1 ^ R2 ^ L) (D2 (impE ^ X D1))
          <- sd R1 D1
          o- ({l:left B} {u:nd B}
                sd' l u -> sd (R2 l) (D2 u))
          o- sd' L X.

% ! A
bangRSD : sd (bangR R1) (bangI D1)
           <- sd R1 D1.

bangLSD : sd (bangL ^ R1 ^ L) (bangE ^ X ^ D2)
           o- ({l:left A} {u:nd A}
                 sd' l u -> sd (R1 l) (D2 u))
           o- sd' L X.
```

The signature above is type-correct in linear Twelf. But what does this establish? For example, assume that we had forgotten the last clause—the signature would still have been correctly typed, but it would no longer represent a proof, since not all possible cases have been covered. So we need to verify the following properties *in addition to the type correctness* in order to be sure that the signature represents a proof.

1. The signature is *well-moded*. This means that when we appeal to the induction hypothesis we have actually constructed an of the appropriate type, and in the end we have fully constructed the object whose existence is postulated. In concrete terms, when we make a recursive call, we need to make sure the input arguments to the type family are fully instantiated, and before we return the output arguments to the type family are also fully instantiated.

   For example, the first argument of the type family `sd` is an input argument, the second an output argument. We then reason as follows:

   ```
   tensorRSD : sd (tensorR ^ R1 ^ R2) (tensorI ^ D1 ^ D2)
                o- sd R1 D1
                o- sd R2 D2.
   ```

   we reason as follows:

   > When we use this clause, the first argument to `sd` is given,
   > so `R1` and `R2` are ground.
   > Therefore, the input arguments to both recursive calls are ground.
   > They yield ground outputs `D1` and `D2`.
   > Therefore the output `(tensorI ^ D1 ^ D2)` will be ground.

Failure of mode-correctness are often simple typographical mistakes which leave the clause well-typed, such as in the following cases.

```
tensorRSD : sd (tensorR ^ R1 ^ R2) (tensorI ^ D1 ^ D2)
             o- sd R1 D1
             o- sd R3 D2.

tensorRSD : sd (tensorR ^ R1 ^ R2) (tensorI ^ D1 ^ D3)
             o- sd R1 D1
             o- sd R2 D2.
```

In the first, `R3` is not ground, in the second `D3` is not ground.

2. The signatures is *terminating*. This means all recursive calls are carried out on smaller terms. These will just be proper subterms if the informal induction argument is over the structure of a derivation. There is one slight complication in that we must allow instantiation of bound variables by parameters so that, for example,

```
tensorLSD : sd (tensorL ^ R1 ^ L) (tensorE ^ X ^ D1)
             o- ({l1:left A} {l2:left B}
                 {x1:nd A} {x2:nd B}
                   sd' l1 x1
                   -o sd' l2 x2
                   -o sd (R1 ^ l1 ^ l2) (D1 ^ x1 ^ x2))
             o- sd' L X.
```

is terminating since `(R1 ^ l1 ^ l2)` is considered a subterm of the input argument `(tensorL ^ R1 ^ L)` because `l1` and `l2` are parameters.

3. The signature *covers* all possible cases. This is the most error-prone property which must be verified, and failures can be subtle. There are three principles classes of failures: A case for a constructor has been omitted, a case for a parameter is missing, or a case is given, but not general enough. For example, a missing case for a parameter arises if we omit the assumption `sd' l1 x1` from the declaration of `tensorLSD` above:

```
tensorLSD : sd (tensorL ^ R1 ^ L) (tensorE ^ X ^ D1)
             o- ({l1:left A} {l2:left B}
                 {x1:nd A} {x2:nd B}
                   sd' l2 x2
                   -o sd (R1 ^ l1 ^ l2) (D1 ^ x1 ^ x2))
             o- sd' L X.
```

When `l1` is encountered in an initial sequence or the principal proposition of a left rule, it will be impossible to proceed with the translation, since `l1` has not been related to the natural deduction hypothesis `x2`.

The more subtle case of insufficient generality is exhibited by the following two examples, both of which mean that perfectly valid translations cannot be carried out. The first fails to allow weakening when translating instances of the $\top$R rule.

```
topRSD : sd (topR ^ ()) (topI ^ ())
          o- <T>.
```

The second applies only to instances where the conclusion of the $\otimes$R rule has the form $A \otimes A$.

```
tensorRSD : sd (tensorR ^ R1 ^ R2) (tensorI ^ D2 ^ D1)
             o- sd R1 D1
             o- sd R2 D2.
```

This fact is only apparent when we look at the reconstructed form of this clause.

```
tensorRSD : {A1:o} {R2:right A1} {D2:nd A1} {R1:right A1} {D1:nd A1}
              sd R2 D2 -o sd R1 D1
                 -o sd (tensorR ^ R1 ^ R2) (tensorI ^ D2 ^ D1).
```

Both `R1` and `R2` are derivations of sequents with conclusion `A1`. The reconstruction of the correct clause is

```
tensorRSD : {A1:o} {R2:right A1} {D2:nd A1} {A2:o}
           {R1:right A2} {D1:nd A2}
              sd R2 D2 -o sd R1 D1
                 -o sd (tensorR ^ R1 ^ R2) (tensorI ^ D1 ^ D2).
```

From the above examples one can see that bugs in proof representations can be subtle. Some may be caught by running examples, other by inspection, but the need for a reliable verification procedure should be clear.

The second example is the completeness property: whenever A can be derived by natural deduction, it can also be derived in the sequent calculus. The proof is very direct, but makes rather heavy use of the cut rule (in contrast to the proof in Chapter 2).

Theorem 5.2 (Completeness of Sequent Calculus with Cut)
If $\Gamma; \Delta \vdash A$ *then* $\Gamma; \Delta \Longrightarrow A$.

Proof: By induction of the structure over the given natural deduction. The main insight is that all introduction rules can be translated straightforwardly to right rules, while all elimination rules require one or more uses of the cut rule in the sequent calculus.[3] □

[3] [*show a few cases*]

The representation of this proof is straightforward along the lines of the soundness proof. This time, we need only one meta-level judgment since we can relate hypotheses in natural deduction directly to initial sequent derivations.

```
cp : nd A -> right A -> type.

%%% Multiplicatives

% A -o B
lolliICP : cp (lolliI ^ D1) (lolliR ^ R1)
            o- ({x:nd A} {l:left A}
                  cp x (init ^ l) -o cp (D1 ^ x) (R1 ^ l)).

lolliECP : cp (lolliE ^ D1 ^ D2)
            (cut ^ R2
               ^ ([l^left A] cut ^ R1
                    ^ ([k^left (lolli A B)]
                         lolliL ^ (init ^ l) ^ ([r^left B] init ^ r) ^ k)))
            o- cp D1 R1
            o- cp D2 R2.

% A * B
tensorICP : cp (tensorI ^ D1 ^ D2) (tensorR ^ R1 ^ R2)
             o- cp D1 R1
             o- cp D2 R2.

tensorECP : cp (tensorE ^ D1 ^ D2)
             (cut ^ R1
                ^ [l^left (tensor A B)] tensorL ^ R2 ^ l)
             o- cp D1 R1
             o- ({x:nd A} {l:left A} {y:nd B} {k:left B}
                   cp x (init ^ l)
                   -o cp y (init ^ k)
                   -o cp (D2 ^ x ^ y) (R2 ^ l ^ k)).
% 1
oneICP : cp (oneI) (oneR).

oneECP : cp (oneE ^ D1 ^ D2) (cut ^ R1 ^ ([l^left (one)] oneL ^ R2 ^ l))
          o- cp D1 R1
          o- cp D2 R2.

%%% Additives

% A & B

withICP : cp (withI ^ (D1, D2)) (withR ^ (R1, R2))
           o- cp D1 R1 & cp D2 R2.

withE1CP : cp (withE1 ^ D1)
            (cut ^ R1
               ^ ([l^left (with A B)] withL1 ^ ([k^left A] init ^ k) ^ l))
            o- cp D1 R1.

withE2CP : cp (withE2 ^ D1)
            (cut ^ R1
               ^ ([l^left (with A B)] withL2 ^ ([k^left B] init ^ k) ^ l))
            o- cp D1 R1.

% T
topICP : cp (topI ^ ()) (topR ^ ())
          o- <T>.

% no topE

% A + B
plusI1CP : cp (plusI1 ^ D1) (plusR1 ^ R1)
            o- cp D1 R1.

plusI2CP : cp (plusI2 ^ D2) (plusR2 ^ R2)
            o- cp D2 R2.

plusECP : cp (plusE ^ D1 ^ (D2 , D3))
           (cut ^ R1
              ^ ([l^left (plus A B)] plusL ^ (R2, R3) ^ l))
           o- cp D1 R1
           o- ({x:nd A} {l:left A}
                 cp x (init ^ l)
                 -o cp (D2 ^ x) (R2 ^ l))
              & ({y:nd B} {k:left B}
                   cp y (init ^ k)
                   -o cp (D3 ^ y) (R3 ^ k)).

% 0
% no zeroI
zeroECP : cp (zeroE ^ D1 ^ ())
           (cut ^ R1 ^ ([l^left (zero)] zeroL ^ () ^ l))
           o- cp D1 R1
           o- <T>.

%%% Exponentials

% A -> B
impICP : cp (impI ^ D1) (impR ^ R1)
          o- ({u:nd A} {l:left A}
                cp u (init ^ l) -> cp (D1 u) (R1 l)).

impECP : cp (impE ^ D1 D2)
          (cut! R2
             ^ ([l:left A] cut ^ R1
                  ^ ([k^left (imp A B)]
                     impL (init ^ l) ^ ([r:left B] init ^ r) ^ k)))
          o- cp D1 R1
          <- cp D2 R2.

% ! A
bangICP : cp (bangI D1) (bangR R1)
           <- cp D1 R1.

bangECP : cp (bangE ^ D1 ^ D2)
           (cut ^ R1 ^ ([l^left (bang A)] bangL ^ R2 ^ l))
           o- cp D1 R1
           o- ({u:nd A} {l:left A}
                 cp u (init ^ l) -o cp (D2 u) (R2 l)).
```

5.2 Concrete Syntax of Linear Twelf

In this section, we extend the concrete syntax of *Elf* [Pfe94] to express the linear operators of *LLF*. In doing so, we want to fulfill two constraints: first of all, existing *Elf* programs should not undergo any syntactic alteration (unless they declare some of the reserved identifiers that we will introduce) if we were to execute them in an implementation of *LLF* relying on the new syntax. In other words, the extension we propose should be conservative with respect to the syntax of *Elf.* Second, we want to avoid a proliferation of operators: keeping their number as small as possible will make future extensions easier to accommodate if their inclusion appears beneficial.

The set of special characters of *Elf* consists of `% : . ) ( ] [ } {`. We extend these with two symbols: `,` and `^`. LLF object and type family constants are consequently represented as identifiers consisting of any non-empty string that does not contain spaces or the characters `% : . ) ( ] [ } { , ^`. As in *Elf,* identifiers must be separated from each other by whitespace (i.e., blanks, tabs, and new lines) or special characters. We augment the set of reserved identifiers of *Elf* (`type`, `->` and `<-`) with `<T>`, `&`, `-o`, `o-`, `<fst>` and `<snd>`. Although not properly an identifier, the symbol `()` is also reserved; this string is forbidden in *Elf.*

The following table associates every $\lambda^{\Pi,\multimap,\&,\top}$ operator to its concrete representation. Terms in the λ^{Π} sublanguage of *LLF* are mapped to the syntax of *Elf.* This language offers the convenience of writing `->` as `<-` with the arguments reversed in order to give a more operational reading to a program, when desired: under this perspective, we read the expression A `<-` B as "A *if* B". We extend this possibility to linear implication, `-o`. Clearly, when we use `o-`, the arguments should be swapped: A `o-` B is syntactic sugar for B `-o` A.

	Abstract syntax	*Concrete syntax*
Kinds	type	`type`
	$\Pi x{:}A.\ K$	`{`x`:`A`}`K A `->` K K `<-` A
Types	$P\,M$	$P\ \ M$
	$\top$	`<T>`
	$A\&B$	A `&` B
	$A \multimap B$	A `-o` B B `o-` A
	$\Pi x{:}A.\ B$	`{`x`:`A`}`B A `->` B B `<-` A
Objects	$\langle\,\rangle$	`()`
	$\langle M, N\rangle$	M`,`N
	$\mathrm{fst}\,M$	`<fst>` M
	$\mathrm{snd}\,M$	`<snd>` M
	$\hat{\lambda} x{:}A.\ M$	`[`x`^`A`]`M
	$M\,\hat{}\,N$	M `^` N
	$\lambda x{:}A.\ M$	`[`x`:`A`]`M
	$M\,N$	$M\ \ N$

The next table gives the relative precedence and associativity of these operators. Parentheses are available to override these behaviors. Note that `-o`, `->`, `o-`, and `<-` all have the same precedence.

Precedence	*Operator*	*Position*
highest	`<fst> _` `<snd> _`	left prefix
	`_ _` `_ ^ _`	left associative
	`_ & _`	right associative
	`_ -o _` `_ -> _`	right associative
	`_ o- _` `_ <- _`	left associative
	`_,_`	right associative
	`_:_`	left associative
lowest	`{_:_}_` `[_:_]_` `[_^_]_`	left prefix

As in *Elf*, a signature declaration $c : A$ is represented by the program clause:

$$c\ \texttt{:}\ A\texttt{.}$$

Type family constants are declared similarly. For practical purposes, it is convenient to provide a means of declaring linear assumptions. Indeed, whenever the object formalism we want to represent requires numerous linear hypotheses, it is simpler to write them as program clauses than to rely on some initialization routine that assumes them in the context during its execution. To this end, we permit declarations of the form

$$c\ \texttt{\^{}}\ A\texttt{.}$$

with the intent that this declaration should be inserted in the context as a linear assumption.[4]

We retain from *Elf* the use of `%` for comments and interpreter directives. Delimited comments have the form `%{... }%`, where embedded delimited comments must be nested properly. We adopt the conventions available in that language in order to enhance the readability of *LLF* programs [Pfe91]. In particular, we permit keeping the type of bound variables implicit whenever they can be effectively reconstructed by means techniques akin to those currently implemented in *Elf*.

We write `{`x`}`B, `[`x`]`B and `[`x`^]`B when maintaining implicit the type A of the variable x in `{`x`:`A`}`B, `[`x`:`A`]`B and `[`x`^`A`]`B, respectively. Similar conventions apply to dependent kinds. As in *Elf*, the binders for variables quantified at the head of a clause can be omitted altogether if we write these variables with identifiers starting with a capital letter. Moreover, the arguments instantiating them can be kept implicit when using these declarations.

Finally, we relax the requirement of writing *LLF* declarations only in η-long form. With sufficient typing information it is always possible to transform a signature to that format.

[4][*currently, this is unimplemented*]

Bibliography

[ABCJ94] D. Albrecht, F. Bäuerle, J. N. Crossley, and J. S. Jeavons. Curry-Howard terms for linear logic. In ??, editor, *Logic Colloquium '94*, pages ??–?? ??, 1994.

[Abr93] Samson Abramsky. Computational interpretations of linear logic. *Theoretical Computer Science*, 111:3–57, 1993.

[Bar96] Andrew Barber. Dual intuitionistic linear logic. Draft manuscript, March 1996.

[Bie94] G. Bierman. On intuitionistic linear logic. Technical Report 346, University of Cambridge, Computer Laboratory, August 1994. Revised version of PhD thesis.

[CF58] H. B. Curry and R. Feys. *Combinatory Logic*. North-Holland, Amsterdam, 1958.

[Chu41] Alonzo Church. *The Calculi of Lambda-Conversion*. Princeton University Press, Princeton, New Jersey, 1941.

[CP97] Iliano Cervesato and Frank Pfenning. A linear spine calculus. Technical Report CMU-CS-97-125, Department of Computer Science, Carnegie Mellon University, April 1997.

[Cur30] H.B. Curry. Grundlagen der kombinatorischen Logik. *American Journal of Mathematics*, 52:509–536, 789–834, 1930.

[Doš93] Kosta Došen. A historical introduction to substructural logics. In Peter Schroeder-Heister and Kosta Došen, editors, *Substructural Logics*, pages 1–30. Clarendon Press, Oxford, England, 1993.

[Gen35] Gerhard Gentzen. Untersuchungen über das logische Schließen. *Mathematische Zeitschrift*, 39:176–210, 405–431, 1935. Translated under the title *Investigations into Logical Deductions* in [Sza69].

[Gir87] J.-Y. Girard. Linear logic. *Theoretical Computer Science*, 50:1–102, 1987.

[GMW79] Michael J. Gordon, Robin Milner, and Christopher P. Wadsworth. *Edinburgh LCF*. Springer-Verlag LNCS 78, 1979.

[Her30] Jacques Herbrand. Recherches sur la théorie de la démonstration. *Travaux de la Société des Sciences et de Lettres de Varsovic*, 33, 1930.

[Her95] Hugo Herbelin. *Séquents qu'on calcule*. PhD thesis, Universite Paris 7, January 1995.

[Hil22] David Hilbert. Neubegründung der Mathematik (erste Mitteilung). In *Abhandlungen aus dem mathematischen Seminar der Hamburgischen Universität*, pages 157–177, 1922. Reprinted in [Hil35].

[Hil35] David Hilbert. *Gesammelte Abhandlungen*, volume 3. Springer-Verlag, Berlin, 1935.

[How69] W. A. Howard. The formulae-as-types notion of construction. Unpublished manuscript, 1969. Reprinted in To H. B. Curry: Essays on Combinatory Logic, Lambda Calculus and Formalism, 1980.

[Hue76] Gérard Huet. *Résolution d'équations dans des langages d'ordre* $1, 2, \ldots, \omega$. PhD thesis, Université Paris VII, September 1976.

[Kni89] Kevin Knight. Unification: A multi-disciplinary survey. *ACM Computing Surveys*, 2(1):93–124, March 1989.

[Lin92] P. Lincoln. Linear logic. *ACM SIGACT Notices*, 23(2):29–37, Spring 1992.

[LS86] Joachim Lambek and Philip J. Scott. *Introduction to Higher Order Categorical Logic*. Cambridge University Press, Cambridge, England, 1986.

[ML85a] Per Martin-Löf. On the meanings of the logical constants and the justifications of the logical laws. Technical Report 2, Scuola di Specializzazione in Logica Matematica, Dipartimento di Matematica, Università di Siena, 1985.

[ML85b] Per Martin-Löf. Truth of a proposition, evidence of a judgement, validity of a proof. Notes to a talk given at the workshop *Theory of Meaning*, Centro Fiorentino di Storia e Filosofia della Scienza, June 1985.

[ML94] Per Martin-Löf. Analytic and synthetic judgements in type theory. In Paolo Parrini, editor, *Kant and Contemporary Epistemology*, pages 87–99. Kluwer Academic Publishers, 1994.

[MM76] Alberto Martelli and Ugo Montanari. Unification in linear time and space: A structured presentation. Internal Report B76-16, Ist. di Elaborazione delle Informazione, Consiglio Nazionale delle Ricerche, Pisa, Italy, July 1976.

[MM82] Alberto Martelli and Ugo Montanari. An efficient unification algorithm. *ACM Transactions on Programming Languages and Systems*, 4(2):258–282, April 1982.

[MOM91] N. Martí-Oliet and J. Meseguer. From Petri nets to linear logic through categories: A survey. *Journal on Foundations of Computer Science*, 2(4):297–399, December 1991.

[MPP92] Dale Miller, Gordon Plotkin, and David Pym. A relevant analysis of natural deduction. Talk given at the workshop on *Types for Proofs and Programs*, Båstad, Sweden, June 1992.

[Par92] Michel Parigot. $\lambda\mu$-calculus: An algorithmic interpretation of classical natural deduction. In A. Voronkov, editor, *Proceedings of the International Conference on Logic Programming and Automated Reasoning*, pages 190–201, St. Petersburg, Russia, July 1992. Springer-Verlag LNCS 624.

[Pfe91] Frank Pfenning. Logic programming in the LF logical framework. In Gérard Huet and Gordon Plotkin, editors, *Logical Frameworks*, pages 149–181. Cambridge University Press, 1991.

[Pfe94] Frank Pfenning. Elf: A meta-language for deductive systems. In A. Bundy, editor, *Proceedings of the 12th International Conference on Automated Deduction*, pages 811–815, Nancy, France, June 1994. Springer-Verlag LNAI 814. System abstract.

[Pfe95] Frank Pfenning. Structural cut elimination. In D. Kozen, editor, *Proceedings of the Tenth Annual Symposium on Logic in Computer Science*, pages 156–166, San Diego, California, June 1995. IEEE Computer Society Press.

[Pra65] Dag Prawitz. *Natural Deduction.* Almquist & Wiksell, Stockholm, 1965.

[PW78] M. S. Paterson and M. N. Wegman. Linear unification. *Journal of Computer and System Sciences*, 16(2):158–167, April 1978.

[Rob65] J. A. Robinson. A machine-oriented logic based on the resolution principle. *Journal of the ACM*, 12(1):23–41, January 1965.

[Rob71] J. A. Robinson. Computational logic: The unification computation. *Machine Intelligence*, 6:63–72, 1971.

[Sce93] A. Scedrov. A brief guide to linear logic. In G. Rozenberg and A. Salomaa, editors, *Current Trends in Theoretical Computer Science*, pages 377–394. World Scientific Publishing Company, 1993. Also in Bulletin of the European Association for Theoretical Computer Science, volume 41, pages 154–165.

[SDP97] Carsten Schürmann, Jöelle Despeyroux, and Frank Pfenning. Primitive recursion for higher-order abstract syntax. Submitted to TCS. Revised version of Technical Report CMU-CS-96-172, December 1997.

[SHD93] Peter Schroeder-Heister and Kosta Došen, editors. *Substructural Logics.* Number 2 in Studies in Logic and Computation. Clarendon Press, Oxford, England, 1993.

[Sza69] M. E. Szabo, editor. *The Collected Papers of Gerhard Gentzen.* North-Holland Publishing Co., Amsterdam, 1969.

[Tro92] A. S. Troelstra. *Lectures on Linear Logic.* CSLI Lecture Notes 29, Center for the Study of Language and Information, Stanford, California, 1992.

[Tro93] A. S. Troelstra. Natural deduction for intuitionistic linear logic. Unpublished manuscript, May 1993.