Lecture Notes on
The Concurrent Logical Framework

15-836: Substructural Logics
Frank Pfenning

Lecture 22
December 5, 2023

1 Introduction

In the last lecture we introduced Linear LF (LLF) as a substructural framework and we saw how to represents proofs in the linear semi-axiomatic sequent calculus. But it turns out that even some very simple systems of linear inference are difficult to represent, particularly those with multiple conclusions that we used at the beginning of the course.

In order to handle these we carefully extend the logical framework with some positive types, leading us to Concurrent LF (CLF) [Watkins et al., 2002, Cervesato et al., 2002, Watkins et al., 2004] which is implemented in the Celf language [Schack-Nielsen and Schürmann, 2008, Schack-Nielsen, 2011][1]. It turns out that this will allow us to capture some *concurrency* in the computations we represent. We then encode the dynamics of futures as an example that is significantly more direct than possible in LLF.

2 Coin Exchange Revisited

Recall the rules for a linear coin exchange from Lecture 1, where q is a quarter, d is a dime, and n is a nickel.

$$\frac{q}{d \quad d \quad n}\ \mathsf{fromQ} \qquad \frac{d \quad d \quad n}{q}\ \mathsf{toQ} \qquad \frac{d}{n \quad n}\ \mathsf{fromD} \qquad \frac{n \quad n}{d}\ \mathsf{toD}$$

Here is a simple proof that from a quarter and a nickel we can get three dimes.

$$\dfrac{\dfrac{q}{d \quad d}\ \mathsf{fromQ} \qquad \dfrac{n \qquad n}{d}\ \mathsf{toD}}{}$$

[1] https://github.com/clf/celf

In linear logic, we can internalize these as (structural!) propositions

$q \multimap (d \otimes d \otimes n)$
$(d \otimes d \otimes n) \multimap q$
$d \multimap (n \otimes n)$
$(n \otimes n) \multimap d$

We could make the second and the forth into something entirely negative by Currying and represent it in LLF, but not the other two.

$\mathsf{fromQ} : q \multimap (d \otimes d \otimes n)$??
$\mathsf{toQ} : d \multimap d \multimap n \multimap q$
$\mathsf{fromD} : d \multimap (n \otimes n)$??
$\mathsf{toD} : n \multimap n \multimap d$

How can we solve this problem and represent this form of inference in LLF? Think about it before you move on, because you actually have seen the technique we need already (specifically in Lecture 2).

The technique is to move the inference steps into the *antecedents*, also flipping their direction. The rules the are (with some arbitrary succedent C):

$$\frac{\Delta, d, d, n \vdash C}{\Delta, q \vdash C}\ \mathsf{fromQ} \qquad \frac{\Delta, q \vdash C}{\Delta, d, d, n \vdash C}\ \mathsf{toQ}$$

$$\frac{\Delta, n, n \vdash C}{\Delta, d \vdash C}\ \mathsf{fromD} \qquad \frac{\Delta, d \vdash C}{\Delta, n, n \vdash C}\ \mathsf{toD}$$

This we can represent in LLF, thinking of q, d, n, and C as judgments (and writing c in lowercase), rather than propositions to avoid an extra level of indirection.

$$\begin{aligned}
&\mathsf{fromQ} : (d \multimap d \multimap n \multimap c) \\
&\qquad\qquad \multimap (q \multimap c) \\
&\mathsf{toQ} : \quad (q \multimap c) \\
&\qquad\qquad \multimap (d \multimap d \multimap n \multimap c) \\
&\mathsf{fromD} : (n \multimap n \multimap c) \\
&\qquad\qquad \multimap (d \multimap c) \\
&\mathsf{toD} : \quad (d \multimap c) \\
&\qquad\qquad \multimap (n \multimap n \multimap c)
\end{aligned}$$

In functional programming this technique could be called *continuation-passing style* where c stands for the continuation.

Now if we want to show that we can get three dimes from a quarter and a nickel, it would be represented as

$$\vdash (d \multimap d \multimap d \multimap c) \multimap (q \multimap c)$$

which we prove as follows:

$$\dfrac{\dfrac{\dfrac{\vdots}{d \multimap d \multimap d \multimap c, d, d, d \vdash c}}{d \multimap d \multimap d \multimap c, d, d, n, n \vdash c}\ \mathsf{toD}}{d \multimap d \multimap d \multimap c, d, d, n, n \vdash c} = \qquad \dfrac{}{d \multimap d \multimap d \multimap c, q, n \vdash c}\ \mathsf{fromQ}}{\vdash (d \multimap d \multimap d \multimap c) \multimap (q \multimap n \multimap c)}\ \multimap R \times 3$$

The omitted part of the proof above is entirely straightforward.

If we represent this derivation as a term in LLF, it would be

$$\vdash \lambda f.\, \lambda q_1.\, \lambda n_1.\, \mathsf{fromQ}\ (\lambda d_1.\, \lambda d_2.\, \lambda n_2.\, \mathsf{toD}\ (\lambda d_3.\, f\ d_1\ d_2\ d_3)\ n_1\ n_2)\ q_1$$

where, as before, we write $h\ (M_1\ ;\ \ldots M_k)$ as $h\ M_1\ \ldots\ M_k$.

There are few notes about this particular term representation. One is that the coins have unique identities. For example, if we swap the arguments to f as in $f\ d_3\ d_1\ d_2$ we obtain a different term. In order to avoid this and reduce the number of possible proofs, we can use *proof irrelevance* [Ley-Wild and Pfenning, 2007].

The other is that no matter what coins and exchange opportunities we have, the proof term will always consist of nested constructors. One possible answer to this is *multifocusing* [Chaudhuri et al., 2008]. Another is to explicitly construct a logical framework with positive connectives, as we'll do now.

3 CLF

The Concurrent Logical Framework was explicitly designed to allow natural encodings of linear forward inference and also the kind of concurrency from the previous example. We only give here a very brief description before we start using it—you are referred to the technical reports and the implementation mentioned in the introduction to the lecture for more information.

Below we have in red the LLF additions to LF and in blue the further additions that CLF makes.

Negative types	A, B	$::=$	$P \mid A \to B \mid \Pi x : A.\, B(x) \mid A \multimap B \mid A \mathbin{\&} B \mid \{A^+\}$
Positive types	A^+, B^+	$::=$	$\mathbf{1} \mid A^+ \otimes B^+ \mid \exists x : A.\, B^+(x) \mid A$
Objects	M	$::=$	$c\ S \mid x\ S \mid \lambda x.\, M(x) \mid (M_1, M_2) \mid \{E\}$
Spines	S	$::=$	$M\ ;\ S \mid (\,) \mid \pi_1\ ;\ S \mid \pi_2\ ;\ S$
Expressions	E	$::=$	$\ldots$
Stable antecedents	Δ	$::=$	$\cdot \mid \Delta, x_{\mathsf{s}} : A \mid \Delta, x_{\mathsf{l}} : A$

There are a number of things to note here. Expressions E are the objects of positive type, yet to be specified. The positive types are included in the negative ones as $\{A^+\}$ which, nowadays, we would recognize as a form of shift. Similarly, the negative types are included directly in the positive ones (again, that should probably be via a shift). Also, there are no positive atoms, even though there should be because at the time we designed CLF we didn't understand the type theory well enough. Also, the antecedent A of $A \to B$, $A \multimap B$, and $\Pi x : A.\, B(x)$ should be positive, and probably also the A in $\exists x : A.\, B(x)$. This last set of issues was recognized and repaired in the design of Celf.

We omitted sums $A \oplus B$ because the branching nature of expressions complicated the form of equality on terms we wanted to allow; a few further remarks later. Before we come to the extension of objects/expressions, let's write some types to illustrate the use of positive types in the encoding of the coin exchange.

$\mathsf{fromQ} : q \multimap \{\, d \otimes d \otimes n \,\}$
$\mathsf{toQ} : d \multimap d \multimap n \multimap \{\, q \,\}$

$\mathsf{fromD} : d \multimap \{\, n \otimes n \,\}$
$\mathsf{toD} : n \multimap n \multimap \{\, d \,\}$

here, we need to wrap even the singletons in the conclusions of the implications because otherwise these clauses would be eligible for backchaining and not for forward chaining.

Now to the proof terms for forward chaining. Since we do not have expressions are just let bindings instead of general matches.

$$\begin{array}{lrcl} \text{Expressions} & E & ::= & \mathbf{let}\ \{p\} = M\ \mathbf{in}\ E \mid T \\ \text{Terms} & T & ::= & [\,] \mid [T_1, T_2] \mid [M, T] \mid M \\ \text{Patterns} & p & ::= & [\,] \mid [p_1, p_2] \mid [x, p] \mid x \end{array}$$

We now show the CLF signature including the proof term for the previous example.

```
1  q : type.
2  d : type.
3  n : type.
4
5  toQ : d -o d -o n -o { q }.
6  fromQ : q -o { d * d * n }.
7  toD : n -o n -o { d }.
8  fromD : d -o { n * n }.
9
10 ex1 : q -o n -o { d * d * d } =
11 \q1. \n1. { let {[d1, [d2, n2]]} = fromQ q1 in
12             let { d3 } = toD n1 n2 in
13             [d1, d2, d3] }.
```

In this particular example, fromQ must come before toD because the argument n_2 to toD is bound in the pattern that is matched against the result of fromQ.

More generally, though we consider two let expressions to be equivalent if they can be swapped without any variable capture. That is,

$$\begin{array}{rl} & (\mathbf{let}\ p = M\ \mathbf{in}\ \mathbf{let}\ q = N\ \mathbf{in}\ T) \\ = & (\mathbf{let}\ q = N\ \mathbf{in}\ \mathbf{let}\ p = M\ \mathbf{in}\ T) \\ & \text{provided } \mathsf{FV}(p) \cap \mathsf{FV}(N) = \emptyset = \mathsf{FV}(q) \cap \mathsf{FV}(M) \end{array}$$

This equality is baked into the definition of CLF at a fundamental level, just like the renaming of bound variables. This allows is to think of the expressions as capturing "*true concurrency*", that is, different independent interleavings of concurrent actions are indistinguishable.

It might be interesting to explore whether the notion of CBA-graphs we sketched in the first two lectures would be an abstract representation of the quivalence classes that arise from commuting independent actions. This may be related to the notion of multifocusing mentioned earlier.

4 Representing the Dynamics of Futures

Without further theory, which can be found in the given references, we show an encoding of the dynamics of the positive fragment of linear SAX, which gives us linear futures. This can easily be extended to encompass the whole language and is much more abstract than the implementation in SML we used in this course.

```
val : type.
exp : type.
cont : type.

addr : type.

unit : val.   % 1
pi1 : addr -o val. % A + B
pi2 : addr -o val.
pair : addr -o addr -o val. % A * B

unit_cont : exp -o cont.
plus_cont : (addr -o exp) & (addr -o exp) -o cont.
pair_cont : (addr -o addr -o exp) -o cont.

cut : (addr -o exp) -o (addr -o exp) -o exp.
id : addr -o addr -o exp.
write : addr -o val -o exp.
read : addr -o cont -o exp.

cell : addr -> val -> type.
proc : exp -> type.

pass : val -> cont -> exp -> type.
pass/unit : pass unit (unit_cont P) P.
pass/plus1 : pass (pi1 A) (plus_cont <(\x. P x), (\y. Q y)>) (P A).
pass/plus2 : pass (pi2 B) (plus_cont <(\x. P x), (\y. Q y)>) (Q B).
pass/pair : pass (pair A B) (pair_cont (\x. \y. P x y)) (P A B).

exec/cut : proc (cut (\x. P x) (\x. Q x))
        -o { Exists a:addr. proc (P a) * proc (Q a) }.
exec/id : proc (id A B) -o cell B V -o { cell A V }.
exec/write : proc (write A V) -o { cell A V }.
exec/read : proc (read A K) -o cell A V -o pass V K P
          -o { proc P }.
```

There are some subtle points here, such as the use of external choice in the encoding of `plus_cont`, but in most respects it is an entirely straightforward representation. Note also the use of the existential to create a fresh address dynamically when executing a cut.

Unlike the coin exchange, we can actually execute SAX programs in this en-

coding because of the don't care nondeterminism that underlies the dynamics. In the coin exchange, we never reach quiescence, but here we do for terminating programs using futures.

In the first two examples we execute a SAX program for negation of a boolean value represented by store with two cells.

```
#query * 1 * 1
Pi c0:addr. Pi c1:addr. Pi c2:addr.
    cell c0 unit * cell c1 (pi1 c0)
    * proc (read c1 (plus_cont (<(\u. write c2 (pi2 u)),
                                (\u. write c2 (pi1 u))>)))
    -o { cell c0 unit * cell c2 (pi2 c0) }.

#query * 1 * 1
Pi c0:addr. Pi c1:addr. Pi c2:addr.
    cell c0 unit * cell c1 (pi2 c0)
    * proc (read c1 (plus_cont (<(\u. write c2 (pi2 u)),
                                (\u. write c2 (pi1 u))>)))
    -o { cell c0 unit * cell c2 (pi1 c0) }.
```

The implementation will print a trace of the computations in the form of terms of the given type. It shows structural bindings with `\!x. M`, while linear bindings are just `\x. M`.

```
Query (*, 1, *, 1) ...
Solution: \!c0. \!c1. \!c2. \[X1, [X2, X3]]. {
    let {X4} = exec/read X3 X2 pass/plus1 in
    let {X5} = exec/write X4 in [X1, X5]}
Query ok.

Query (*, 1, *, 1) ...
Solution: \!c0. \!c1. \!c2. \[X1, [X2, X3]]. {
    let {X4} = exec/read X3 X2 pass/plus2 in
    let {X5} = exec/write X4 in [X1, X5]}
Query ok.
```

In the last query we use an existential quantifier in the succedent so as not to anticipate the answer and let CLF's forward inference compute it for us.

```
#query * 1 * 1
    Pi c0:addr. Pi c1:addr. Pi c2:addr.
    cell c0 unit * cell c1 (pi2 c0)
    * proc (read c1 (plus_cont (<(\u. write c2 (pi2 u)),
                                (\u. write c2 (pi1 u))>)))
    -o { Exists V0. Exists V2. cell c0 V0 * cell c2 V2 }.
```

Here, the answers `!unit` and `!(pi1 c0)` are presented in the results as witness for the existentials. The exclamation mark shows that they are not linear.

```
Query (*, 1, *, 1) ..
```

```
Solution: \!c0. \!c1. \!c2. \[X1, [X2, X3]]. {
    let {X4} = exec/read X3 X2 pass/plus2 in
    let {X5} = exec/write X4 in [!unit, [!(pi1 c0), [X1, X5]]]}
Query ok.
```

References

Iliano Cervesato, Frank Pfenning, David Walker, and Kevin Watkins. A concurrent logical framework II: Examples and applications. Technical Report CMU-CS-02-102, Department of Computer Science, Carnegie Mellon University, 2002. Revised May 2003.

Kaustuv Chaudhuri, Dale Miller, and Alexis Saurin. Canonical sequent proofs via multi-focusing. In *5th International Conference on Theoretical Computer Science*, pages 383–396, Milano, Italy, September 2008. IFIPAICT 273.

Ruy Ley-Wild and Frank Pfenning. Avoiding causal dependencies via proof irrelevance in a concurrent logical framework. Technical Report CMU-CS-07-107, Carnegie Mellon University, February 2007.

Anders Schack-Nielsen. *Implementing Substructural Logical Frameworks*. PhD thesis, IT University of Copenhagen, January 2011.

Anders Schack-Nielsen and Carsten Schürmann. Celf - a logical framework for deductive and concurrent systems. In A. Armando, P. Baumgartner, and G. Dowek, editors, *Proceedings of the 4th International Joint Conference on Automated Reasoning (IJCAR'08)*, pages 320–326, Sydney, Australia, August 2008. Springer LNCS 5195.

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

Kevin Watkins, Iliano Cervesato, Frank Pfenning, and David Walker. A concurrent logical framework: The propositional fragment. In S. Berardi, M. Coppo, and F. Damiani, editors, *Types for Proofs and Programs*, pages 355–377. Springer-Verlag LNCS 3085, 2004. Revised selected papers from the *Third International Workshop on Types for Proofs and Programs*, Torino, Italy, April 2003.