Lecture Notes on
Concurrent Monadic Computations

15-816: Linear Logic
Frank Pfenning

Lecture 22
April 11, 2012

In this lecture we explore the parallelism inherent in forward-chaining logic programming. As we have seen in the last lecture, forward chaining takes place when the stable sequent has the form $\Gamma\ ;\ \Delta \vdash E \div A\ \mathit{lax}$. The material of this lecture is not well-documented in the literature, although the form of the equality on concurrent computations is mentioned in some of the publications on CLF [WCPW02, CPWW02, WCPW04b, WCPW04a].

1 Operations on Binary Numbers

We start with a review of operations on numbers in binary representation in Celf. These are *backward-chaining* logic programs, but they will be helpful as subcomputations in our forward-chaining programs.

First, the type declarations of natural numbers as a sequence of bits (represented by `b0` and `b1`) and the increment predicate. We provide a *mode declaration* which states that the first argument to increment should be input and the second output.

```
bit : type.
b0 : bit.
b1 : bit.

nat : type.
e : nat.
d : nat -o bit -o nat.

% Increment
```

```
inc : nat -> nat -> type.
#mode inc + -.

inc/e : inc e        (d e b1).
inc/0 : inc (d M b0) (d M b1).
inc/1 : inc (d M b1) (d R b0) o- inc M R.
```

It is tempting to use the `inc` predicate to perform a decrement when we need it, giving it the second argument as input and expecting the first argument as output. Even though the predicate would be well-moded in that direction, it does not work correctly because of nondeterminism: a query `inc M (d e b1)` to find the predecessor of 1, matches both the first and second clause. Upon first success it will return `e`, the second time `(d e b0)`. Both are valid representations of 0. This kind of nondeterminism is unacceptable in a backward-chaining program: because of backtracking, it will deliver multiple solutions which could wreak havoc on the calling predicates.

It is my experience that, in general, it is rarely useful to run a given predicate in multiple directions. Here, we just write an explicit decrement predicate and then addition.

```
% Decrement
dec : nat -> nat -> type.
#mode dec + -.

% no case for dec/e
% 2m-1 = 2(m-1)+1
dec/0 : dec (d M b0) (d N b1) o- dec M N.
dec/1 : dec (d M b1) (d M b0).

% Addition
plus : nat -> nat -> nat -> type.
#mode plus + + -.

p/e/e : plus e        e        e.
p/e/0 : plus e        (d N b0) (d N b0).
p/e/1 : plus e        (d N b1) (d N b1).

p/0/e : plus (d M b0) e        (d M b0).
p/0/0 : plus (d M b0) (d N b0) (d R b0) o- plus M N R.
p/0/1 : plus (d M b0) (d N b1) (d R b1) o- plus M N R.

p/1/e : plus (d M b1) e        (d M b1).
```

```
p/1/0 : plus (d M b1) (d N b0) (d R b1) o- plus M N R.
p/1/1 : plus (d M b1) (d N b1) (d R b0) o- plus M N K * inc K R.
```

There is another natural definition of addition that uses a carry bit instead of an increment. We finish our natural number "library" with predicates for less-or-equal and less-than.

```
% Less or equal
leq : nat -> nat -> type.
#mode leq + +.

leq/e   : leq        e        N.
leq/0/e : leq (d M b0)        e o- leq M e.
leq/0/0 : leq (d M b0) (d N b0) o- leq M N.
leq/0/1 : leq (d M b0) (d N b1) o- leq M N.
% no case for leq/1/e
% 2m+1 <= 2n iff 2m <= 2n-1 iff 2m <= 2n-2 iff m <= n-1
leq/1/0 : leq (d M b1) (d N b0) o- dec N N1 * leq M N1.
leq/1/1 : leq (d M b1) (d N b1) o- leq M N.

% Less than
lt : nat -> nat -> type.
#mode lt + +.

% no case for lt/e/e
lt/e/0 : lt        e (d N b0) o- lt e N.
lt/e/1 : lt        e (d N b1).
% no case for lt/0/e
lt/0/0 : lt (d M b0) (d N b0) o- lt M N.
lt/0/1 : lt (d M b0) (d N b1) o- leq M N.
% no case for lt/1/e
% 2m+1 < 2n iff 2m < 2n-1 iff 2m <= 2n-2 iff m <= n-1
lt/1/0 : lt (d M b1) (d N b0) o- dec N N1 * leq M N1.
lt/1/1 : lt (d M b1) (d N b1) o- lt M N.
```

2 Parallel Maximum

Our first exercise is to define a function to compute the parallel maximum of a collection of elements. We start with a list of elements, which we have to load into the context before the parallel algorithm itself starts.

Lists of natural numbers are very similar to lists of digits.

```
list : type.
```

```
nil : list.
cons : nat -o list -o list.
```

To load the elements into the context, we just recurse down the list in a backward-chaining program, assuming elem n for every element n in the list.

```
elem : nat -> type.
#mode elem -.

load : list -> nat -> type.
#mode load + -.

load/nil : load nil.
load/cons : load (cons N Ns) o- (elem N -o load Ns).
```

While this loads the elements into the context it doesn't start any computation. When the last element has been added as a resource, we want to start forward chaining, so the first clause has to be changed to look like

```
load/nil : load nil o- { ... }.
```

In the elided part, we have to read off the solution we want, namely the maximum element. We compute the maximum by picking any two elements, comparing then, and keeping only the larger one.

```
combine : elem M * elem N * leq M N -o {elem N}.
```

Eventually we must be left with only the largest one, assuming there are any. Going back to the first clause, once forward chaining reaches quiescence, we match against the only remaining (and therefore maximal) element and pass this back all the way through the calls that loaded the predicate.

```
load : list -> nat -> type.
#mode load + -.

load/nil : load nil Max o- {elem Max}.
load/cons : load (cons N Ns) Max o- (elem N -o load Ns Max).
```

Note that this requires the list to be non-empty. Otherwise, we could initialize the context with a smallest element 0, but we will not bother with this.

The core of this algorithm has a lot of parallelism because we can pick up any two elements and keep only the larger one. If there are many elements, a lot of these actions will be independent. The Celf implementation of CLF is currently not parallel, however, so while we model potential parallelism, we don't actually execute in parallel at present.

3 Proof Terms

Next we write a test query that finds the maximum of the list $[0, 3, 2, 0]$ and should return 3. There should be only one solution. We therefore ask `#query * 1 * 1`, which means we expect a unique solution and run the query just once.

```
#query * 1 * 1
load (cons e (cons (d (d e b1) b1) (cons (d (d e b1) b0) (cons e nil)))) Max.
```

Here is a record of our interaction, using the Celf source files on the course web pages.

```
% celf nat.clf list.clf max.clf
Celf ver 2.9. Copyright (C) 2011
[reading nat.clf]
...
[closing nat.clf]
[reading list.clf]
...
[closing list.clf]
[reading max.clf]
elem: nat -> type.
#mode elem { } -.
load: list -> nat -> type.
#mode load { } + -.
load/nil: Pi Max: nat. {elem Max} -o load nil Max.
load/cons: Pi Max: nat. Pi N: nat. Pi Ns: list. (elem N -o load Ns Max) -o load (cons N
combine: Pi M: nat. Pi N: nat. (elem M * (elem N * leq M N)) -o {elem N}.
Query (*, 1, *, 1) load (cons e (cons (d (d e b1) b1) (cons (d (d e b1) b0) (cons e nil
Solution: load/cons (\X1. load/cons (\X2. load/cons (\X3. load/cons (\X4. load/nil {
    let {X5} = combine [X1, [X3, leq/e]] in
    let {X6} = combine [X4, [X2, leq/e]] in
```

```
    let {X7} = combine [X5, [X6, leq/0/1 (leq/1/1 leq/e)]] in X7})))
 #Max = d (d e b1) b1
Query ok.
[closing max.clf]
```

We see that Celf also reports a solution, which consists of a proof term and a substitution for the free variables in the query, here only `Max`, printed as `#Max`, which luckily is 3.

Let's examine this proof term in more detail, recalling the crucial constants in the signature.

```
load/nil : load nil Max o- {elem Max}.
load/cons : load (cons N Ns) Max o- (elem N -o load Ns Max).

combine : elem M * elem N * leq M N -o {elem N}.
```

At the top level we see

```
load/cons (\X1. load/cons (\X2. load/cons (\X3. load/cons (\X4. load/nil {...}))))
```

which introduces resources

```
X1 : elem e,
X2 : elem (d (d e b1) b1),
X3 : elem (d (d e b1) b0),
X4 : elem e
```

into the linear context. Then it enters the monad, where we have elided the monadic expression representing the forward-chaining computation. This is

```
let {X5} = combine [X1, [X3, leq/e]] in
let {X6} = combine [X4, [X2, leq/e]] in
let {X7} = combine [X5, [X6, leq/0/1 (leq/1/1 leq/e)]] in
X7
```

We see that the concrete syntax for a multiplicative pair $M_1 \otimes M_2$ is `[M1, M2]`. So in the first step, `combine` is applied to `X1`, `X3`, and a proof that $0 \leq 2$. This consumes `X1 : elem e` and `X3 : elem (d (d e b1) b0)` and adds a new resource

```
X5 : elem (d (d e b1) b0)
```

The second step compares 0 and 3, consuming `X4` and `X2` and adding

```
X6 : elem (d (d e b1) b1)
```

At this point the linear context contains only

```
X5 : elem (d (d e b1) b0),
X6 : elem (d (d e b1) b1)
```

These are combined in the next step

```
let {X7} = combine [X5, [X6, leq/0/1 (leq/1/1 leq/e)]] in
```

where the `leq/0/1 (leq/1/1 leq/e)` represents the proof that $2 \leq 3$. In the process we consume `X5` and `X6` and add resource `X7 : elem (d (d e b1) b1)`.

At this point no further rules applies: forward chaining has reached quiescence. In this state we now solve the query `elem Max`, which has proof term `X7` and instantiates `Max` to 3.

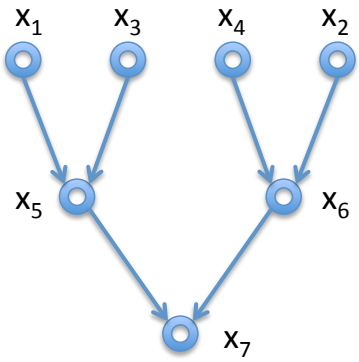
4 Dependency Graphs of Computations

Let's look again at this particular forward-chaining computation. Running the query again likely would yield a different term, due to some built-in non-determinism in the operational semantics and its implementation.

```
let {X5} = combine [X1, [X3, leq/e]] in
let {X6} = combine [X4, [X2, leq/e]] in
let {X7} = combine [X5, [X6, leq/0/1 (leq/1/1 leq/e)]] in
X7
```

The variables now induces some dependencies between the computation steps. When we start, we have variables x_1, x_2, x_3, and x_4. Then x_5 depends

on x_1 and x_3, and so on.



We should not be able to distinguish different ways to linearize the graph into a sequence of let-expressions. And, indeed, in the framework different interleavings of independent steps are treated as equal. This is discussed a little further in Section 7.

5 Permutations

As a warm-up for sorting a list, we want to write a program to compute an arbitrary *permutation* of a list. The difference to the above algorithm for maximum is that we cannot read out the answer from the state in a single step, but need to build a new list.

The basic idea is quite simple: we first load the context with propositions member n for each element of the list, then we just collect the elements back into a list. Since the order of resources in the linear context is irrelevant, we can pick up any permutation of the input list in this manner.

```
perm : list -> list -> type.
#mode perm + -.

member : nat -> type.
#mode member -.

collect : list -> type.
#mode collect -.
```

```
perm/load/cons : perm (cons N Ns) Ks o- (member N -o perm Ns Ks).
perm/load/nil  : perm nil Ks o- (collect nil -o {collect Ks}).

collect1 : member N * collect Ks -o {collect (cons N Ks)}.
```

Because the order in which the forward chaining rules fire is don't-care nondeterministic, we obtain a single answer which is some arbitrary permutation of the input list. We can try running a query 4 times, examining the results, with a query

```
#query * * * 4
perm (cons e (cons (d (d e b1) b1) (cons (d (d e b1) b0) (cons e nil)))) Ks.
```

and, indeed, you should see several variations.

6 Sorting

As a next example we try a "parallel bubblesort". If we had an ordered logical framework, we could load the elements into the *ordered* context

$$\mathsf{elem}\, x_1, \ldots, \mathsf{elem}\, x_n$$

and then use the single rule

$$\mathsf{elem}\, m \bullet \mathsf{elem}\, n \bullet \mathsf{lt}\, n\, m \twoheadrightarrow \{\mathsf{elem}\, n \bullet \mathsf{elem}\, m\}$$

When we reach quiescence the list must be sorted, because no two adjacent elements are out of order.

Unfortunately, Celf does not have an ordered context, so we have to encode this into linear logic. For this, we use the same technique that we used in substructural operational semantics: we thread *destinations* so that each element implicitly has pointers to the elements to its left and right.

To load elements into the context we start from some initial destination `D0` and create a new destination `d'` for every element in the list. At the end we have `D0` and `Dn` as the left and right endpoints of the elements loaded into the context.

```
dest : type.

ld : dest -> list -> dest -> list -> type.
#mode ld + + + -.
```

```
mem : dest -> nat -> dest -> type.
#mode mem - - -.

coll : dest -> list -> dest -> type.
#mode coll + - +.

ld/cons : ld D0 (cons N Ns) D Ks o- (Pi d'. mem D N d' -o ld D0 Ns d' Ks).
ld/nil : ld D0 nil Dn Ks o- {coll D0 Ks Dn}.
```

We use `coll D0 Ks Dn` to collect the elements between `D0` and `Dn` into the final list `Ks` which is passed backwards to the initial call to `ld`.

Now the swapping rule itself can exchange any two adjacent elements that are out of order.

```
swap : mem D1 M D2 * mem D2 N D3 * lt N M -o {mem D1 N D2 * mem D2 M D3}.
```

It is important that we preserve the two endpoints of the pair of elements, `D1` on the left and `D3` on the right.

To collect the elements we use a backward-chaining program, which maintains the destinations bracketing the segment of the list that still has to be collected. It stops and return `nil` if the right end of the list has been reached, which can be seen from the left and right end of the segment being identical.

```
coll/nil : coll Dn nil Dn.
coll/cons : coll D (cons N Ks) Dn o- mem D N D' o- coll D' Ks Dn.
```

Finally, we need a top-level call with an initial destination `d0` which represents the empty list (nothing has been loaded yet).

```
sort : list -> list -> type.
#mode sort + -.

sort/top : sort Ns Ks o- (Pi d0. ld d0 Ns d0 Ks).
```

Even though this algorithm is probably quite inefficient, it does have some parallelism, because out-of-order pairs that do not overlap can exchange themselves in parallel.

7 Capturing Dependency with Trace Equality

We now slightly reformulate the rules in the definition of CLF in order to isolate monadic computations as separate objects to we can study them separately. This is designed to separate the forward-chaining computation from the succedent of the sequent that is only executed when quiescence is reached.

We now have only one rule when we reach the judgment $E \div S\ lax$.

$$\frac{\Delta \to \epsilon : \Delta' \quad \Delta' \to M : [S]}{\Delta \to \mathsf{let}\ \ \epsilon\ \mathsf{in}\ M \div S\ lax}\ \mathsf{lax}$$

The new syntactic category of ϵ is intended to represented a computation starting from state Δ and ending with state Δ'. We have previously written

$$\Delta \longrightarrow \Delta'$$

now we write

$$\Delta \to \epsilon : \Delta'$$

to record the computation itself. In the context of the new lax rule we want Δ' to be quiescent, but that is not part of the formalism, only of the underlying operational semantics. For brevity, we omit the context Γ and its evolution. Adding this back in is not entirely straightforward, but we will not address it here.

So what are the possibilities for concurrent computations? The first possibility is to take no step at all. In that case the state does not change.

$$\frac{}{\Delta \to [\,] : \Delta}\ [\,]$$

We can also compose to computations if their interfaces match up: the second one starts in the state in which the first one ends.

$$\frac{\Delta_1 \to \epsilon_1 : \Delta_2 \quad \Delta_2 \to \epsilon_2 : \Delta_3}{\Delta_1 \to \epsilon_1\ ;\ \epsilon_2 : \Delta_3}\ ;$$

Finally, we need to perform a single step of forward chaining, which arises

from focusing on a forward-chaining clause.

$$\frac{\Delta, x{:}[A] \vdash \epsilon : \Delta'}{\Delta, x{:}A \to \epsilon : \Delta'}\ \mathsf{foc}L'$$

$$\frac{\Delta, p{:}S \to \Delta'}{\Delta, R : [\{S\}] \to \{p\} = R : \Delta'}\ \mathsf{blur}L'$$

$$\frac{\Delta\ \mathsf{stable}}{\Delta \vdash \Delta}\ \mathsf{id}_\Delta$$

There are analogous versions of the left focus rule for affine and persistent resources. The focus and inversion rules in the antecent have to be updated to have a different form of succedent now.

Next we look at some of the laws associated with concurrent computations. This is related to the theory of traces, but we will not pursue this in detail. First we note that the zero-step computation is the unit of composition, and that composition is associative.

$$\begin{array}{l}
[\,] \,;\, \epsilon = \epsilon \,;\, [\,] = \epsilon \\
(\epsilon_1 \,;\, \epsilon_2) \,;\, \epsilon_3 = \epsilon_1 \,;\, (\epsilon_2 \,;\, \epsilon_3)
\end{array}$$

Computations that are independent of each other can happen in either order. This means that no variable introduced by one is used in the other and vice versa. So we define the pre-set ($\bullet(\epsilon)$) and post-set ($(\epsilon)\bullet$) of variables for each computation. We write $\mathsf{fv}(R)$ for the free variables of an atomic term, and $\mathsf{bv}(p)$ for the variables bound in a pattern.

$$\begin{array}{lcl}
\bullet(p = R) & = & \mathsf{fv}(R) \\
(p = R)\bullet & = & \mathsf{bv}(p) \\
\bullet(\epsilon_1 \,;\, \epsilon_2) & = & \bullet(\epsilon_1) \cup (\bullet(\epsilon_2) - (\epsilon_1)\bullet) \\
(\epsilon_1 \,;\, \epsilon_2)\bullet & = & ((\epsilon_1)\bullet - \bullet(\epsilon_2)) \cup (\epsilon_2)\bullet) \\
\bullet([\,]) & = & \{\,\} \\
([\,])\bullet & = & \{\,\}
\end{array}$$

Now we can say when two computations can be exchanged:

$$\epsilon_1 \,;\, \epsilon_2 \quad = \quad \epsilon_2 \,;\, \epsilon_1 \quad \text{provided } (\epsilon_1)\bullet \cap \bullet(\epsilon_2) = \bullet(\epsilon_1) \cap (\epsilon_2)\bullet = \{\,\}$$

References

[CPWW02] 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.

[WCPW02] 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.

[WCPW04a] 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.

[WCPW04b] Kevin Watkins, Iliano Cervesato, Frank Pfenning, and David Walker. Specifying properties of concurrent computations in CLF. In C.Schürmann, editor, *Proceedings of the 4th International Workshop on Logical Frameworks and Meta-Languages (LFM'04)*, Cork, Ireland, July 2004. Electronic Notes in Theoretical Computer Science (ENTCS), vol 199, pp. 133–145, 2008.