# Lecture Notes on Concurrent Cost Semantics

15-816: Substructural Logics
Frank Pfenning

Lecture 24
November 29, 2016

In the last lecture we finally formally introduced the Concurrent Logical Framework (CLF) [WCPW02, CPWW02] and its implementation in Celf [SN11]. In this lecture we will use CLF to develop a high level implementation of SILL, the core of a session-typed programming language. As we will see, with some thought, it turns out the CLF is an almost perfect vehicle for specifying SILL. This approach exhibits a perfect isomorphism with the CLF specification of the sequent calculus for linear logic [BBMS16] that has previously been presented on purely logical grounds [Pfe94, Ree09].

We then proceed to instrument our semantics with costs, to compute the work and span of concurrent computations which together can be seen as measuring the "parallel complexity" of the computation. In our setting, we count the total number of communication steps that have to be performed, but other cost measures can be derived in a similar manner.

## 1 Representing Channels and Process Expressions

At the outset, we assume the following types (to be revised later):

```
ch : type.
exp : type.
```

where `ch` represents channels and `exp` represents process expressions. Channels should remain abstract, as in our previous encoding, which means the type `ch` is inhabited only by parameters that are introduced during the computation. Expressions represent concurrent programs.

At the level of our linear inference semantics, we write $\mathsf{proc}(c, P)$ for the state of a process computing $P$ and providing a service along $c$. This will be represented here as `proc P C` for an expression `P` and channel `C`, so we have

```
proc : exp -> ch -> type.
```

Note that we "curry" our propositions, so there are no explicit parentheses around the arguments of a predicate. The other point to note is the `proc P C` is a *type*, rather than a proposition. This is because we are working in a type theory, so our process configuration will look like

```
p1 : proc P1 C1, p2 : proc P2 C2, ..., pn : proc Pn Cn
```

where each `p` represents a means to reference the process when describing the computation. For now, it is perfectly sensible to just think of it as a proposition.

Next, we consider a simple program

$$w{:}A \vdash \mathsf{send}\ c\ w\ ; \mathsf{close}\ c :: (c : A \otimes \mathbf{1})$$

where $A$ is arbitrary and therefore a propositional variable. The first idea of representation would be

$$\ulcorner \mathsf{send}\ c\ w\ ; \mathsf{close}\ c \urcorner = \texttt{send C W (close C)}$$

which would give us the straightforward types

```
send : ch -> exp -> exp.
close : ch -> exp.
```

While these types are workable, they are not fully satisfactory as we will see, and do not fully exploit the expressive power of the framework.

On the receiving side, we might have a matching process

$$c{:}A \otimes 1 \vdash y \leftarrow \mathsf{recv}\ c\ ; \mathsf{wait}\ c\ ; P_y :: (d : D)$$

The receive construct binds the variable $y$ with scope $\mathsf{wait}\ c\ ;\ P_y$. We represent is by a corresponding binder with corresponding scope in the logical framework. Recall that in CLF, the binders are represented with a $\lambda$-abstraction and written as `\x. M`. Using this idea we obtain

$$\ulcorner y \leftarrow \mathsf{recv}\ c\ ; \mathsf{wait}\ c\ ; P_y \urcorner = \texttt{recv C (\textbackslash y. wait C (P y))}$$

An interesting part of this representation is that we indicate the possible dependence of $P_y$ on $y$ by writing `P y`, which means that `P` will have type `ch -> exp`. Overall, this gives us

```
recv : ch -> (ch -> exp) -> exp.
wait : ch -> exp -> exp.
```

Again, this is servicable, but uninspired. Why? Note that our process language is *linearly typed* and in fact we have seen it as being in a Curry-Howard correspondence with the linear sequent calculus! But our representation above is *not* linearly typed. In both the sender and the receiver example expressions

```
send C W (close C)          % sender
recv C (\y. wait C (P y))   % receiver
```

the channel `C` is apparently not linear. This means that we can write bogus expressions, namely programs that use their channels not linearly and they will type-check in the framework, which is unfortunate.

We will sharpen our representation so that *only* process expression that are properly linear will be well-typed in the framework. We use different techniques for the provider and the client of a channel, although other choices are certainly possible. On the provider side, we note that an executing process

```
proc (send C W (close C)) C
```

has a lot of redundancy, because the channel `C` along which we communicate is mentioned multiple times. What we do instead is for the provider expressions to leave the provider channel (here `C`) *implicit*, so the above becomes

```
proc (send_ W (close_)) C
```

where the underscore suffix in the name of the `send_` and `close_` are there to remind us that they implicitly refer to the channel that is provided. With that, we can then use linear typing:

```
send_ : ch -o exp -o exp.
close_ : exp.
```

To be formal, the representation function now takes a parameter $c$ (the channel along which the process provides) so we can recognize the appropriate syntactic form.

$$\begin{array}{lcl} \ulcorner \mathsf{send}\ c\ w\ P \urcorner^c & = & \texttt{send\_ W } \ulcorner P \urcorner^c \\ \ulcorner \mathsf{close}\ c \urcorner^c & = & \texttt{close\_} \end{array}$$

Unfortunately, on the client side this particular device fails. This is because a client can use many different processes, and the name of a channel is critical to identify which channel we want to communicate with. So how do we deal with the apparent non-linearity of `C` in an expression such as

```
recv C (\y. wait C (P y))
```

which uses `C` twice? A clue to the answer was provided many lectures ago when we gave the *asynchronous semantics*. Recall that, for example

$$\frac{\mathsf{proc}(c, \mathsf{send}\ c\ w; P)}{\mathsf{proc}(c', P) \qquad \mathsf{msg}(c, \mathsf{send}\ c\ w\ ;\ c \leftarrow c')}\ \otimes C^{c'}$$

where $c'$ is a freshly chosen continuation channel. So we'll use this idea here, even though for now our semantics is synchronous: sending will create a fresh continuation channel for further communication. We bake this into our representation, rather than using it only as a feature of our semantics. So we define

$$\ulcorner y \leftarrow \mathsf{recv}\ c\ ;\ Q_y \urcorner^d \quad = \quad \texttt{recv C (\textbackslash y. \textbackslash c.}\ \ulcorner Q_y \urcorner^d\ \texttt{y c)}$$

Of course, $\mathsf{wait}\ c$ does not receive a continuation channel, since the associated process has terminated. So we get

$$\ulcorner \mathsf{wait}\ c\ ;\ Q \urcorner^d \quad = \quad \texttt{wait C}\ \ \ulcorner Q \urcorner^d$$

These constructs can now be linearly typed

```
recv : ch -o (ch -o ch -o exp) -o exp.
wait : ch -o exp -o exp.
```

Let's think about forwarding and spawning. Within a process, forwarding is

$$\mathsf{proc}(c, c \leftarrow d)$$

which means that forwarding has $c$ as an implicit argument. Similarly, spawning

$$\mathsf{proc}(d, x \leftarrow P_x\ ;\ Q_x)$$

creates a new channel $c$, which is provided by $P_c$ (and therefore implicit in $P_c$ and used by $Q_c$, where it is explicit. This means we have

$$\begin{array}{lcl} \ulcorner c \leftarrow d \urcorner^c & = & \texttt{fwd\_ D} \\ \ulcorner x \leftarrow P_x\ ;\ Q_x \urcorner^d & = & \texttt{spawn\_}\,\ulcorner P_x \urcorner^x\ \texttt{(\textbackslash x.}\ \ulcorner Q_x \urcorner^d\ \texttt{x)} \end{array}$$

from which we can read off the following linear types

```
fwd : ch -o exp.
spawn : exp -o (ch -o exp) -o exp.
```

In summary, for the constructs implementing forward, spawn, $A \otimes B$, and $\mathbf{1}$, we have the following representation function where we assume that channels $c$ in the programming notation are translated to variables $C$ with the same name in the logical framework:

$$
\begin{array}{lcll}
\ulcorner c \leftarrow d \urcorner^{c} & = & \texttt{fwd\_ D} & \\
\ulcorner x \leftarrow P_x \,;\, Q_x \urcorner^{c} & = & \texttt{spawn\_ P (\textbackslash x. Q x)} & \text{where } \ulcorner P_x \urcorner^{x} = \texttt{P}, \ulcorner Q_x \urcorner^{c} = \texttt{Q} \\
\ulcorner \mathsf{send}\ c\ w\ P \urcorner^{c} & = & \texttt{send\_ W P} & \text{where } \ulcorner P \urcorner^{c} = \texttt{P} \\
\ulcorner y \leftarrow \mathsf{recv}\ c \,;\, Q_y \urcorner^{d} & = & \texttt{recv C (\textbackslash y. \textbackslash c. Q y c)} & \text{where } \ulcorner Q_y \urcorner^{d} = \texttt{Q} \\
\ulcorner \mathsf{close}\ c \urcorner^{c} & = & \texttt{close\_} & \\
\ulcorner \mathsf{wait}\ c \,;\, Q \urcorner^{d} & = & \texttt{wait C Q} & \text{where } \ulcorner Q \urcorner^{d} = \texttt{Q}
\end{array}
$$

This gives rise to the following *linear* types for the process constructors:

```
ch : type.
exp : type.

fwd_ : ch -o exp.
spawn_ : exp -o (ch -o exp) -o exp.

send_ : ch -o exp -o exp.
recv : ch -o (ch -o ch -o exp) -o exp.

close_ : exp.
wait : ch -o exp -o exp.
```

## 2 Intrinsic Typing

At this point we achieved a partial victory: only process expressions which treat channels linearly will be well-typed in the framework, providing a modicum of correctness checking. However, many meaningless process expressions can still be represented. For example,

$$x \leftarrow \mathsf{close}\ x \,;\, y \leftarrow \mathsf{recv}\ x \,;\, \mathsf{wait}\ x \,;\, d \leftarrow y$$

does not make much sense since close $x$ is matched up with a recv $x$ instead of a wait. And yet, its translation

```
example1 : exp =
spawn_ (close_) (\x. recv x (\y. \x. wait x (fwd_ y))).
```

type-checks perfectly.

Of course, the problem is that the translation *enforces linearity* but does *not enforce types*. If we want to achieve this additional amount of precision, we need to *index* both channels and expressions with their session types. Fortunately, this can be done quite easily. We don't even need to change the representation function, just make their CLF types more precise. To start with we define (on our small fragment so far):

```
tp : type.
tensor : tp -> tp -> tp.
one : tp.

ch : tp -> type.
exp : tp -> type.
```

Processes `proc P C` define a process of type $A$ that offers a channel of type $A$, so we have

```
proc : exp A -> proc A -> type.
```

This declaration is schematic over $A$ and Celf type reconstruction will determine $A$ wherever it sees `proc P C` from `P` and `C`.

For the language constructs, we just need to read the type indices off the typing rules. We show a few examples, starting with identity.

$$\frac{}{d{:}A \vdash c \leftarrow d :: (c : A)}\ \textsf{id}$$

Recall that $\ulcorner c \leftarrow d \urcorner^{c} =$ `fwd_ D` which means that

```
fwd : ch A -o exp A.
```

Here, the first `A` comes from the type of the channel $d$, while the expression comes from the type of the (implicit) channel $c$.

Similarly

$$\frac{\Delta \vdash P_x :: (x : A) \quad \Delta', x{:}A \vdash Q_x :: (c : C)}{\Delta, \Delta' \vdash x \leftarrow P_x \;;\; Q_x :: (c : C)}\ \textsf{cut}$$

Since

$$\ulcorner x \leftarrow P_x \;;\; Q_x \urcorner^{c} = \texttt{spawn\_}\ \ulcorner P_x \urcorner^{x}\ \texttt{(\textbackslash x.}\ \ulcorner Q_x \urcorner^{c}\ \ \texttt{x)}$$

we obtain

```
spawn_ : exp A -o (ch A -o exp C) -o exp C.
```

As a last detailed example, let's look at receiving by the client.

$$\frac{\Delta, y{:}A, x{:}B \vdash Q_y :: (c : C)}{\Delta, x{:}A \otimes B \vdash y \leftarrow \mathsf{recv}\ x\ ;\ Q_y :: (c : C)}\ \otimes L$$

with

$$\ulcorner y \leftarrow \mathsf{recv}\ x\ ;\ Q_y \urcorner^c = \texttt{recv X (\textbackslash y. \textbackslash x.}\ \ulcorner Q_y \urcorner^c\ \texttt{y x)}$$

```
recv : ch (tensor A B) -o (ch A -o ch B -o exp C) -o exp C.
```

Summarizing all the types so far, we have

```
tp : type.
tensor : tp -> tp -> tp.
one : tp.

ch : tp -> type.
exp : tp -> type.

fwd_ : ch A -o exp A.
spawn_ : exp A -o (ch A -o exp C) -o exp C.

send_ : ch A -o exp B -o exp (tensor A B).
recv : ch (tensor A B) -o (ch A -o ch B -o exp C) -o exp C.

close_ : exp one.
wait : ch one -o exp C -o exp C.

proc : exp A -> ch A -> type.
```

Now our previous example

```
example1 : exp =
spawn_ (close_) (\x. recv x (\y. \x. wait x (fwd_ y))).
```

will no longer type-check, but fail with the (slightly edited, replacing two variables with underscores) message

```
Type-checking failed in declaration of example1 on line 17:
Unification failed: Constants tensor and one differ
Object 1 has type:
ch one
but expected:
ch (tensor _ _)
```

## 3 Choice

Choice, whether external or internal is not significant, reveals a new challenge. We decide to only implement binary choice just to keep the encoding as straightforward as possible. We use internal choice as our example. We have the following constructs

$$\begin{array}{l} c.\pi_1 \ ; P \\ c.\pi_2 \ ; P \\ \mathsf{case}\ c\ (\pi_1 \Rightarrow Q_1 \mid \pi_2 \Rightarrow Q_2) \end{array}$$

Since $A \oplus B$ is also positive, the sending constructs use the provider channel and therefore have an implicit argument. We use `select1_` and `select2_` as our concrete names for the two sending constructs.

$$\begin{array}{lcl} \ulcorner c.\pi_1 \ ; P \urcorner^c & = & \texttt{select1\_}\ \ulcorner P \urcorner^c \\ \ulcorner c.\pi_2 \ ; P \urcorner^c & = & \texttt{select2\_}\ \ulcorner P \urcorner^c \\ \ulcorner \mathsf{case}\ c\ (\pi_1 \Rightarrow Q_1 \mid \pi_2 \Rightarrow Q_2) \urcorner^d & = & \texttt{case C (\textbackslash c.}\ \ulcorner Q_1 \urcorner^d\ \texttt{c) (\textbackslash c.}\ \ulcorner Q_2 \urcorner^d\ \texttt{c)} \quad ?? \end{array}$$

The problem here is the last line. Let's examine the typing rule.

$$\frac{\Delta, c{:}A \vdash Q_1 :: (d : D) \quad \Delta, c{:}B \vdash Q_2 :: (d : D)}{\Delta, c{:}A \oplus B \vdash \mathsf{case}\ c\ (\pi_1 \Rightarrow Q_1 \mid \pi_2 \Rightarrow Q_2) :: (d : D)} \oplus L$$

Since $\oplus$ is an additive connective, all channels are propagated into both branches of the case construct. In the first attempted encoding above, however, the context will be split between $\ulcorner Q_1 \urcorner^d$ and $\ulcorner Q_2 \urcorner^d$. So we need to exploit that the framework also has an additive connective, namely external choice $A^- \mathbin{\&} B^-$, with the proof term being a pair $\langle M, N \rangle$.

$$\begin{array}{lcl} \ulcorner c.\pi_1 \ ; P \urcorner^c & = & \texttt{select1\_}\ \ulcorner P \urcorner^c \\ \ulcorner c.\pi_2 \ ; P \urcorner^c & = & \texttt{select2\_}\ \ulcorner P \urcorner^c \\ \ulcorner \mathsf{case}\ c\ (\pi_1 \Rightarrow Q_1 \mid \pi_2 \Rightarrow Q_2) \urcorner^d & = & \texttt{case C <(\textbackslash c.}\ \ulcorner Q_1 \urcorner^d\ \texttt{c),(\textbackslash c.}\ \ulcorner Q_2 \urcorner^d\ \texttt{c)>} \end{array}$$

which yields the types

```
select1_ : exp A -o exp (plus A B).
select2_ : exp B -o exp (plus A B).
case : ch (plus A B) -o (ch A -o exp C) & (ch B -o exp C) -o exp C.
```

## 4 Operational Semantics: Forward and Spawn

Ideally, at the highest level of abstraction, we would like

```
c/fwd : proc (fwd_ D) C -o { C = D }.
```

that is, `C` and `D` are globally identified. Unfortunately, we omitted equality as a type constructor in CLF, so we need a different idea. The the simplest seems to be to actually synchronize with the provider of `D` and relabel it to become the provider of `C`. Since our calculus is linear, there will be exactly one provider of `D`, so this does not create any ambiguity or race conditions.

```
c/fwd : proc P D * proc (fwd_ D) C -o { proc P C }.
```

The forwarding process itself of course terminates in this step. We can see how the decision to leave the providing channel implicit helps here: if `P` referred to the channel `D` multiple times as the concrete process syntax does, then we would actually have to substitute `C` for `D` throughout `P`, which is a somewhat complex operation. In the rule above, all implicit references are now to `C`, where they previously referenced `D`.

For process spawn we need to create a fresh channel and start a new process. As usual, we use existential quantification in the framework to create a fresh parameter.

```
c/spawn : proc (spawn_ P (\x. Q x)) C
           -o { Exists a. proc P a * proc (Q a) C }.
```

We see that `P` provides along the new channel `a`, while `(Q a)` is a client of `a`.

## 5 Operational Semantics: Communication

The synchronous semantics is now a straightforward transcription of our inference rule, taking care of creating and using continuation channels. As we did for the description of the operational semantics we use the notation `c'` for the continuation channel of `c`.

```
c/tensor : proc (send_ W P) C * proc (recv C (\x. \c'. Q x c')) D
        -o { Exists c'. proc P c' * proc (Q W c') D }.
```

In the right-hand side of this rule, we write `Q W c'` to substitute `W` for `x` and the continuation channel `c'` for the bound variable `c'`. This takes advantage of $\beta$-reduction at the framework level to implement the name-for-name substitution at the process level.

For **1**, there will be no continuation channel so we just synchronize the `close_` with the matching `wait`.

```
c/one : proc (close_) C * proc (wait C Q) D
        -o { proc Q D }.
```

There are two rules for $A \oplus B$, depending on whether the first or the second branch is selected.

```
c/plus1 : proc (select1_ P) C * proc (case C <(\c'. Q1 c'),(\c'. Q2 c')>) D
         -o { Exists c'. proc P c' * proc (Q1 c') D }.
c/plus2 : proc (select2_ P) C * proc (case C <(\c'. Q1 c'),(\c'. Q2 c')>) D
         -o { Exists c'. proc P c' * proc (Q2 c') D }.
```

Note that even though no types are mentioned, these rules are type-checked, so both linearity and session-typing must at least be consistent. Of course, we can still make mistakes that will pass this test. For example, if we replace `Q2` by `Q1` in the `c/plus2` rule, CLF type reconstruction will still succeed by giving both branches the same type. We would need formal metatheory to catch such an error, but there is currently no tool to support such an activity.

We can try our semantics on some simple example: spawning a process of type **1** which just closes, in parallel with one that just waits for that to finish.

```
#query * 1 * 1
Pi c0. proc (spawn_ (close_) (\c1. wait c1 (close_))) c0 -o {proc P c0}.
```

The Celf directive `#query * 1 * 1` means that we run without bound, expecting 1 solution, looking for arbitrarily many, and running the query only once. We write `Pi c0.` to create a fresh initial channel `c0`. On the right hand side of this negative type, we have `proc P c0` which will test if the final configuration (that is, the one where we have quiescence) consists of a single process offering along `c0`. It will show this process, which we expect to be `close_`. We get:

```
Solution: \!c0. \X1. {
    let {[!a, [X2, X3]]} = c/spawn X1 in
    let {X4} = c/one [X2, X3] in X4}
 #P = close_
Query ok.
```

The resulting proof term is a representation of the computation of the process expression above, that is, a sequent of configurations. Let's write in the state of the configuration at each line as a comment with the types. We mark persistent variables with an exclamation mark `!`.

```
Solution: \!c0. \X1. {                  % !c0:ch one, X1:proc (spawn_ ...)
    let {[!a, [X2, X3]]} = c/spawn X1 in % !c0:ch one, !a:ch one,
                                        %    X2:proc (close_) a, X3:proc (wait a ...) c0
    let {X4} = c/one [X2, X3]           % !c0:ch one, !a:ch one, X4:proc (close_) c0
    in X4}
 #P = close_
Query ok.
```

We see that while expressions are typed linearly, channels are *not* linear in configurations. This is not directly possibly, since channels appear in index positions of processes that provide or use them. However, the process that provides each channel is treated as a linear resource.

Let's write one more example, which represents a kind of negation, where we think of $\mathsf{bool} = \mathbf{1} \oplus \mathbf{1}$. The for $c : \mathsf{bool}$ we think of $c.\pi_1 \; ; \mathsf{close}\; c$ as true and $c.\pi_2 \; ; \mathsf{close}\; c$ as false. The following program spawns a process which behaves like false, which is then negated by its client.

$c_1 \leftarrow (c_1.\pi_2 \; ; \mathsf{close}\; c_1) \; ;$
$\mathsf{case}\; c_1 \;\; (\pi_1 \Rightarrow c_0.\pi_2 \; ; \mathsf{close}\; c_0$
$\qquad\quad \mid \pi_2 \Rightarrow c_0.\pi_1 \; ; \mathsf{close}\; c_0)$

In the CLF encoding:

```
Pi c0. proc (spawn_ (select2_ close_)
             (\c1. case c1 <(\c2. wait c2 (select2_ close_)) ,
                            (\c3. wait c3 (select1_ close_))>)) c0
```

As expected, this will execute in 3 steps: one spawn, one select, and one close, and end up as a process wishing to send $\pi_1$ and then closing the channel $c_0$.

```
Solution: \!c0. \X1. {
    let {[!a, [X2, X3]]} = c/spawn X1 in
    let {[!c', [X4, X5]]} = c/plus2 [X2, X3] in
    let {X6} = c/one [X4, X5] in X6}
 #P = select1_ close_
Query ok.
```

The signature and queries from this section can be found with the course materials[1]. The completion with the remaining connectives $A \multimap B$ and $A \mathbin{\&} B$ is also available online[2].

## 6 An Asynchronous Semantics

The representation and typing of all channels and process expressions remains the same when we want to give an asynchronous semantics, but we have two propositions $\mathsf{proc}(c, P)$ and $\mathsf{msg}(c, P)$ to define the operational semantics. Only certain kinds of messages are permitted, but we will not formalize this within the judgments.

First, spawn does not change, but a forwarding process interacts now with a message rather than a process. Because our language fragment has only positive connectives so far ($A \otimes B$, $\mathbf{1}$, $A \oplus B$) all messages come *from* the provider of `D`, so the rule can essentially stay the same. If we add negative propositions, forwarding may also need to interact with messages coming along `C` (see Exercise 2).

```
proc : exp A -> ch A -> type.
msg : exp A -> ch A -> type.

c/fwd : msg P D * proc (fwd_ D) C
         -o { msg P C }.
c/spawn : proc (spawn_ P (\x. Q x)) C
        -o { Exists a. proc P a * proc (Q a) C }.
```

The simple send now decomposes into two. As a reminder, we show the formulation using linear inference.

$$\frac{\mathsf{proc}(c, \mathsf{send}\ c\ w\ ;\ P)}{\mathsf{proc}(c', P) \quad \mathsf{msg}(c, \mathsf{send}\ c\ w\ ;\ c \leftarrow c')}\ \otimes C_s^{c'}$$

$$\frac{\mathsf{msg}(c, \mathsf{send}\ c\ w\ ;\ c \leftarrow c') \quad \mathsf{proc}(d, y \leftarrow \mathsf{recv}\ c\ ;\ Q_y)}{\mathsf{proc}(d, [c'/c]Q_w)}\ \otimes C_r$$

In CLF syntax, these become

[1] http://www.cs.cmu.edu/~fp/courses/15816-f16/misc/session/session-sync.clf

[2] http://www.cs.cmu.edu/~fp/courses/15816-f16/misc/session/session-complete-sync.clf

```
s/tensor : proc (send_ W P) C
        -o { Exists c'. proc P c' * msg (send_ W (fwd_ c')) C }.
r/tensor : msg (send_ W (fwd_ C')) C * proc (recv C (\x. \c'. Q x c')) D
        -o { proc (Q W C') D }.
```

We see that the decision to parameterize by a continuation channel works out well. Note that send creates the continuation channel, which is then received together with the channel $w$.

The remainder of the rules are divided analogously into send and receive rules.

```
s/one : proc (close_) C
        -o { msg (close_) C }.
r/one : msg (close_) C * proc (wait C Q) D
        -o { proc Q D }.

s/plus1 : proc (select1_ P) C
        -o { Exists c'. proc P c' * msg (select1_ (fwd_ c')) C }.
r/plus1 : msg (select1_ (fwd_ C')) C * proc (case C <(\c'. Q1 c'),(\c'. Q2 c')>) D
        -o { proc (Q1 C') D }.

s/plus2 : proc (select2_ P) C
        -o { Exists c'. proc P c' * msg (select2_ (fwd_ c')) C }.
r/plus2 : msg (select2_ (fwd_ C')) C * proc (case C <(\c'. Q1 c'),(\c'. Q2 c')>) D
        -o { proc (Q2 C') D }.
```

Now, for example, the second query (slightly modified to send true instead of false) ends in a configuration with no remaining process but two messages, transmitting $\pi_2$ followed by a close. We capture this when we examine the final configuration by expecting it to consist of two messages, one with a new destination (we couldn't predict what it is called, so we quantify over it) and a second one with the original destination.

```
#query * 1 * 1
Pi c0. proc (spawn_ (select1_ close_)
            (\c1. case c1 <(\c2. wait c2 (select2_ close_)) ,
                           (\c3. wait c3 (select1_ close_))>)) c0
-o { Exists c1. msg P c1 * msg (Q c1) c0 }.
```

Indeed, we obtain the expected answer and a proof term representing the computation.

```
Solution: \!c0. \X1. {
    let {[!a, [X2, X3]]} = c/spawn X1 in
    let {[!c', [X4, X5]]} = s/plus1 X2 in
    let {X6} = s/one X4 in
    let {X7} = r/plus1 [X5, X3] in
    let {X8} = r/one [X6, X7] in
    let {[!c'_1, [X9, X10]]} = s/plus2 X8 in
    let {X11} = s/one X9 in [!c'_1, [X11, X10]]}
 #P = close_
 #Q = \X1. select2_ (fwd_ X1)
Query ok.
```

Due to the increased parallelism afforded by asynchronous communication, some steps here could be carried out in parallel. For example, the sending of close and the receiving of $\pi_1$ are independent and could happen in either order. We can see that because there is no dependencies between these two lines: `X6` does not occur in `r/plus1 [X5, X3]` (nor does `X7` occur in `s/one X4`).

```
    let {X6} = s/one X4 in
    let {X7} = r/plus1 [X5, X3] in
```

By *true concurrency*, the computation where these two lines are swapped are indistinguishable from the given one.

The signature and queries from this section can be found with the course materials[3]. The completion with the remaining connectives $A \multimap B$ and $A \mathbin{\&} B$ is also available[4].

## 7 A Cost Semantics Tracking Total Work

We now want to instrument our semantics to compute the *total work* performed by a concurrent computation. We define this here as the total number of communication steps that take place, and for simplicity we restrict ourselves to the synchronous semantics (see Exercise 4).

For every process, we keep track of the total work that it has performed. We count here the number of *send* operations. Since every message that is sent is also received, counting the number of receives just doubles this cost.

[3]http://www.cs.cmu.edu/~fp/courses/15816-f16/misc/session/session-async.clf

[4]http://www.cs.cmu.edu/~fp/courses/15816-f16/misc/session/session-complete-async.clf

Our basic predicate for linear inference is now $\mathsf{proc}(c, w, P)$, where $w$ tracks the amount of work performed by this process so far. We begin with the rules for tensor, which is straightfoward.

$$\frac{\mathsf{proc}(c, w, \mathsf{send}\ c\ e\ ;\ P) \quad \mathsf{proc}(d, w', y \leftarrow \mathsf{recv}\ c\ ;\ Q_y)}{\mathsf{proc}(c, w+1, P) \quad \mathsf{proc}(d, w', Q_e)}\ \otimes C$$

Spawning makes sure the new process starts at work 0.

$$\frac{\mathsf{proc}(d, w, x \leftarrow P_x\ ;\ Q_x)}{\mathsf{proc}(c, 0, P_c) \quad \mathsf{proc}(d, w, Q_c)}\ \mathsf{spawn}^c$$

Forwarding is interesting, because the work performed by the forwarding process must be accounted for. So we have to add it into the process that it notifies of the forwarding.

$$\frac{\mathsf{proc}(d, w, P) \quad \mathsf{proc}(c, w', c \leftarrow d)}{\mathsf{proc}(c, w+w', P)}\ \mathsf{fwd}$$

If we decided to count forwarding as communication, we would send the cost of the resulting process to $w + w' + 1$. Similar reasoning applies to process of type $\mathbf{1}$.

$$\frac{\mathsf{proc}(c, w, \mathsf{close}\ c) \quad \mathsf{proc}(d, w', \mathsf{wait}\ c\ ;\ Q)}{\mathsf{proc}(c, w+w'+1, Q)}\ \mathbf{1}C$$

The remaining rules and their transcription into Celf follows the pattern of what we have done before.

For a work semantics for asynchronously communicating processes, see Exercise 4.

## 8 A Cost Semantics for Span

The *span* of a concurrent computation can be defined in different ways. We can say that the span consists of the number of communication steps where everything that can happen in parallel, does. Another way to define it is by looking at the dependency graph induced by the truly concurrent semantics and define it as the longest path from the root (where the computation starts) to the leaf (where the computation ends).

We specify this through a predicate $\mathsf{proc}(c, t, P)$, where $t$ is *the earliest time* that this process could be at the given stage of the computation. Again, we only count explicit communication steps as costing time, but more complex measures are certainly possible.

We begin again with sending a channel along a channel. The earliest the two processes can synchronize is at time $\mathsf{max}(t, t')$, and then we have to add 1 to be able to continue.

$$\frac{\mathsf{proc}(c, t, \mathsf{send}\ c\ e\ ;\ P) \quad \mathsf{proc}(d, t', y \leftarrow \mathsf{recv}\ c\ ;\ Q_y)}{\mathsf{proc}(c, \mathsf{max}(t, t') + 1, P) \quad \mathsf{proc}(d, \mathsf{max}(t, t') + 1, Q_e)}\ {\otimes}C$$

Spawning makes sure the new process starts with the the clock of the parent process, because that is the earliest time it could have been spawned. If we like, we could also count the spawn itself; here we do not.

$$\frac{\mathsf{proc}(d, t, x \leftarrow P_x\ ;\ Q_x)}{\mathsf{proc}(c, t, P_c) \quad \mathsf{proc}(d, t, Q_c)}\ \mathsf{spawn}^c$$

Forwarding can take place at the earliest that either process could have gotten to this point.

$$\frac{\mathsf{proc}(d, t, P) \quad \mathsf{proc}(c, t', c \leftarrow d)}{\mathsf{proc}(c, \mathsf{max}(t, t'), P)}\ \mathsf{fwd}$$

If we decided to count forwarding as communication, we would set the cost of the resulting process to $w + w' + 1$. Similar reasoning applies to process of type $\mathbf{1}$.

$$\frac{\mathsf{proc}(c, t, \mathsf{close}\ c) \quad \mathsf{proc}(d, t', \mathsf{wait}\ c\ ;\ Q)}{\mathsf{proc}(c, \mathsf{max}(t, t') + 1, Q)}\ \mathbf{1}C$$

One reason we are counting communication steps that advance the type, but not spawns or forwards is that this allows us to use the type as a guide for the number of communications that must happen, even if we do not necessarily know their timing.

Again, transcription into CLF does not pose any particular challenges, except perhaps implementing the arithmetic.

## Exercises

**Exercise 1** Our encoding takes advantage of the asymmetric nature of intuitionistic sequents so we can leave the offering channel implicit. Revise this implementation so that the offering process is abstracted over the offering channel, which would give the type

```
proc : (ch A -o proc A) -> ch A -> type.
```

Of course, the encoding of process expressions has to change accordingly write. Encode this approach in Celf, rewrite the examples in the new syntax, and compare.

**Exercise 2** In the case of the asynchronous semantics, the simple rule

```
c/fwd : msg P D * proc (fwd_ D) C -o {msg P C}.
```

is no longer sufficient to implement forwarding. Exhibit a concrete, well-typed process that will get stuck with only this rule and extend the implementation of forwarding that it works for all the connectives.

**Exercise 3** Write a cost semantics that counts the total number of processes that will be created during an execution.

**Exercise 4** Give a cost semantics counting total work for *asynchronous* communication. As before, only count sending of messages (not receipt) and exclude forward and spawn.

**Exercise 5** Give a cost semantics for span for *asynchronous* communication. As before, only count the sending of message (not receipt) and exclude forward and spawn.

**Exercise 6** Extend the representation of SILL with recursively defined types and recursively defined processes so that you can encode programs such as the queue or stack. Discuss some of the options and obstacles, and implement your extension, with example, in Celf.

## References

[BBMS16] Peter Brottveit Bock, Alessandro Bruni, Agata Murawska, and Carsten Schürmann. Representing session types. Unpublished manuscript, presented at the seminar in honor of Dale Miller's 60th birthday, December 2016.

[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.

[Pfe94] Frank Pfenning. Structural cut elimination in linear logic. Technical Report CMU-CS-94-222, Department of Computer Science, Carnegie Mellon University, December 1994.

[Ree09] Jason C. Reed. *A Hybrid Logical Framework*. PhD thesis, Carnegie Mellon University, September 2009. Available as Technical Report CMU-CS-09-155.

[SN11] Anders Schack-Nielsen. *Implementing Substructural Logical Frameworks*. PhD thesis, IT University of Copenhagen, January 2011.

[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.