Lecture Notes on
Law and Order

15-816: Substructural Logics
Frank Pfenning

Lecture 8
September 22, 2016

In this lecture we first complete a set of connectives for ordered logic and their operational interpretation. Then we will write some programs exploiting the gained expressive power.

1 Multiplicative Connectives

We call A / B, $B \setminus A$, $A \bullet B$ and $A \circ B$ the *multiplicative connectives*, following Girard's nomenclature for linear logic [Gir87]. This class of connectives also includes $\mathbf{1}$, which have already defined.

Under. This is entirely symmetric to / from the last lecture, so we just state the rules here.

$$\frac{(y{:}B)\ \Omega \vdash P_y :: (x{:}A)}{\Omega \vdash (y \leftarrow \mathsf{recv}\ x\ ;\ P_y) :: (x{:}B \setminus A)}\ \setminus R \qquad \frac{\Omega_L\ (x{:}A)\ \Omega_R \vdash Q :: (z{:}C)}{\Omega_L\ (w{:}B)\ (x{:}B \setminus A)\ \Omega_R \vdash (\mathsf{send}\ x\ w\ ;\ Q) :: (z{:}C)}\ \setminus L^*$$

The following computation rule implements the cut reduction of $\setminus R$ and $\setminus L^*$.

$$\frac{\mathsf{proc}(x, y \leftarrow \mathsf{recv}\ x\ ;\ P_y) \quad \mathsf{proc}(z, \mathsf{send}\ x\ w\ ;\ Q)}{\mathsf{proc}(x, P_w) \quad \mathsf{proc}(z, Q)}\ \setminus C$$

Note that the operational reading here is *identical* for A / B; the difference is entirely in the restrictions about where $w{:}B$ or $y{:}B$ are to be found. This indicates that order in this formulation of the operational semantics is not essential from the computational point of view, but imposes some restrictions on the programs one can write. These restrictions should express some intuitively understandable program properties which will hopefully emerge when we start to use these connectives.

Fuse. The natural right rule for fuse has two premises.

$$\frac{\Omega_1 \vdash A \quad \Omega_2 \vdash B}{\Omega_1\ \Omega_2 \vdash A \bullet B}\ \bullet R$$

In order to avoid spawning a new process in this rule, we have to find an equivalent one-premise version somehow analogous to the restricted left rules for $A \,/\, B$ and $B \setminus A$.

Actually, we could decide that either of the two premises should be the identity, just as the specialized form forms of $/L$ and $/R$ arise by forcing one of the premises to be an identity. So either $\Omega_1 = A$ or $\Omega_2 = B$. These considerations yield:

$$\frac{\Omega \vdash B}{A\ \Omega \vdash A \bullet B}\ \bullet R^* \qquad\qquad \frac{\Omega \vdash A}{\Omega\ B \vdash A \bullet B}\ \bullet R^\dagger$$

Rather arbitrarily we pick the first, which yields the following pair of right and left rules for $A \bullet B$

$$\frac{\Omega \vdash B}{A\ \Omega \vdash A \bullet B}\ \bullet R^* \qquad\qquad \frac{\Omega_L\ A\ B\ \Omega_R \vdash C}{\Omega_L\ (A \bullet B)\ \Omega_R \vdash C}\ \bullet L$$

Again, we can ask which of the rules carries information, and here it is $\bullet R^*$ which sends. Filling in channel names, we see that once again a channel is sent and received.

$$\frac{\Omega \vdash P :: (x{:}B)}{(w{:}A)\ \Omega \vdash (\mathsf{send}\ x\ w\ ;\ P) :: (x{:}A \bullet B)}\ \bullet R^* \quad \frac{\Omega_L\ (y{:}A)\ (x{:}B)\ \Omega_R \vdash Q_y :: (z{:}C)}{\Omega_L\ (x{:}A \bullet B)\ \Omega_R \vdash (y \leftarrow \mathsf{recv}\ x\ ;\ Q_y) :: (z{:}C)}\ \bullet L$$

Interestingly, we don't need any new program constructs, since $A \bullet B$, just like $A \,/\, B$ and $B \setminus A$, just send and receive channels. This is reflected in this simple computation rule where we now write Q_w for $[w/y]Q_y$:

$$\frac{\mathsf{proc}(x, \mathsf{send}\ x\ w\ ;\ P) \quad \mathsf{proc}(z, y \leftarrow \mathsf{recv}\ x\ ;\ Q_y)}{\mathsf{proc}(P) \quad \mathsf{proc}(Q_w)}\ \bullet C$$

Twist. $A \circ B$ is entirely symmetric to $A \bullet B$, so we just state the rules.

$$\frac{\Omega \vdash P :: (x{:}B)}{\Omega\ (w{:}A) \vdash (\mathsf{send}\ x\ w\ ;\ P) :: (x{:}A \circ B)}\ \circ R^* \qquad \frac{\Omega_L\ (x{:}B)\ (y{:}A)\ \Omega_R \vdash Q_y :: (z{:}C)}{\Omega_L\ (x{:}A \circ B)\ \Omega_R \vdash (y \leftarrow \mathsf{recv}\ x\ ;\ Q_y) :: (z{:}C)}\ \circ L$$

$$\frac{\mathsf{proc}(x, \mathsf{send}\ x\ w\ ;\ P) \quad \mathsf{proc}(z, y \leftarrow \mathsf{recv}\ x\ ;\ Q_y)}{\mathsf{proc}(P) \quad \mathsf{proc}(Q_w)}\ \circ C$$

2 Rule Summary

Here is a summary of the propositions and proofs, which double as types and programs, together with their rules.

$$\begin{array}{llll}
\text{Types} & A, B, C & ::= & \oplus\{l_i : A_i\}_{i\in I} \mid \&\{l_i : A_i\})_{i\in I} \mid \mathbf{1} \\
& & \mid & A \,/\, B \mid B \setminus A \mid A \bullet B \mid A \circ B \\
\\
\text{Processes} & P, Q & ::= & x \leftarrow y & \text{identity} \\
& & \mid & x \leftarrow P_x\ ;\ Q_x & \text{cut} \\
& & \mid & x.l_k\ ;\ P \mid \mathsf{case}\ x\ (l_i \Rightarrow Q_i)_{i\in I} & \oplus, \& \\
& & \mid & \mathsf{close}\ x \mid \mathsf{wait}\ x\ ;\ Q & \mathbf{1} \\
& & \mid & \mathsf{send}\ x\ y\ ;\ P \mid y \leftarrow \mathsf{recv}\ x\ ;\ Q_x & /, \setminus, \bullet, \circ
\end{array}$$

We can see that despite some complexity in the language of types, the process language is relatively parsimonious. This syntactic overloading of several constructs is acceptable because during type checking of processes we always track the types of all channels exactly. For this to be always possible, we need to annotate cut with the type of freshly introduced channel, as in $x{:}A \leftarrow P_x\ ;\ Q_x$. Because we are in the sequent calculus, the types of channels in the premises are always strict components of the types in the conclusion.

Judgmental Rules

$$\frac{\Omega \vdash P_x :: (x{:}A) \quad \Omega_L\ (x{:}A)\ \Omega_R \vdash Q_x :: (z{:}C)}{\Omega_L\ \Omega\ \Omega_R \vdash (x \leftarrow P_x\ ;\ Q_x) :: (z{:}C)}\ \mathsf{cut} \qquad \frac{}{y{:}A \vdash x \leftarrow y :: (x{:}A)}\ \mathsf{id}$$

Additive Connectives

$$\frac{\Omega \vdash P :: (x{:}A_k) \quad (k \in I)}{\Omega \vdash (x.l_k\ ;\ P) :: (x : \oplus\{l_i{:}A_i\}_{i\in I})}\ \oplus R_k \qquad \frac{\Omega_L\ (x{:}A_i)\ \Omega_R \vdash Q_i :: (z{:}C) \quad (\forall i \in I)}{\Omega_L\ (x{:}\oplus\{l_i{:}A_i\}_{i\in I})\ \Omega_R \vdash \mathsf{case}\ x\ (l_i \Rightarrow Q_i)_{i\in I} :: (z{:}C)}\ \oplus L$$

$$\frac{\Omega \vdash P_i :: (x{:}A_i) \quad (\forall i \in I)}{\Omega \vdash \mathsf{case}\ x\ (l_i \Rightarrow P_i)_{i\in I} :: (x{:}\&\{l_i{:}A_i\}_{i\in I}))}\ \& R \qquad \frac{\Omega_L\ (x{:}A_k)\ \Omega_R \vdash P :: (z{:}C) \quad (k \in I)}{\Omega_L\ (x : \&\{l_i{:}A_i\}_{i\in I})\ \Omega_R \vdash (x.l_k\ ;\ Q) :: (z{:}C)}\ \& L_k$$

Multiplicative Connectives

$$\frac{}{\cdot \vdash \mathsf{close}\ x :: (x{:}\mathbf{1})}\ \mathbf{1}R \qquad \frac{\Omega_L\ \Omega_R \vdash Q :: (z{:}C)}{\Omega_L\ (x{:}\mathbf{1})\ \Omega_R \vdash (\mathsf{wait}\ x\ ;\ Q) :: (z{:}C)}\ \mathbf{1}L$$

$$\frac{\Omega\ (y{:}B) \vdash P_y :: (x{:}A)}{\Omega \vdash (y \leftarrow \mathsf{recv}\ x\ ;\ P_y) :: (x{:}A\ /\ B)}\ /R \qquad \frac{\Omega_L\ (x{:}A)\ \Omega_R \vdash Q :: (z{:}C)}{\Omega_L\ (x{:}A\ /\ B)\ (w{:}B)\ \Omega_R \vdash (\mathsf{send}\ x\ w\ ;\ Q) :: (z{:}C)}\ /L^*$$

$$\frac{(y{:}B)\ \Omega \vdash P_y :: (x{:}A)}{\Omega \vdash (y \leftarrow \mathsf{recv}\ x\ ;\ P_y) :: (x{:}B \setminus A)}\ \setminus R \qquad \frac{\Omega_L\ (x{:}A)\ \Omega_R \vdash Q :: (z{:}C)}{\Omega_L\ (w{:}B)\ (x{:}B \setminus A)\ \Omega_R \vdash (\mathsf{send}\ x\ w\ ;\ Q) :: (z{:}C)}\ \setminus L^*$$

$$\frac{\Omega \vdash P :: (x{:}B)}{(w{:}A)\ \Omega \vdash (\mathsf{send}\ x\ w\ ;\ P) :: (x{:}A \bullet B)}\ \bullet R^* \qquad \frac{\Omega_L\ (y{:}A)\ (x{:}B)\ \Omega_R \vdash Q_y :: (z{:}C)}{\Omega_L\ (x{:}A \bullet B)\ \Omega_R \vdash (y \leftarrow \mathsf{recv}\ x\ ;\ Q_y) :: (z{:}C)}\ \bullet L$$

$$\frac{\Omega \vdash P :: (x{:}B)}{\Omega\ (w{:}A) \vdash (\mathsf{send}\ x\ w\ ;\ P) :: (x{:}A \circ B)}\ \circ R^* \qquad \frac{\Omega_L\ (x{:}B)\ (y{:}A)\ \Omega_R \vdash Q_y :: (z{:}C)}{\Omega_L\ (x{:}A \bullet B)\ \Omega_R \vdash (y \leftarrow \mathsf{recv}\ x\ ;\ Q_y) :: (z{:}C)}\ \circ L$$

For the operational semantics we have to remember that we are using *linear inference*, not ordered inference.

$$\frac{\mathsf{proc}(z, x \leftarrow P_x \;;\; Q_x)}{\mathsf{proc}(w, P_w) \quad \mathsf{proc}(z, Q_w)}\ \mathsf{cmp}^w \qquad \frac{\mathsf{proc}(x, x \leftarrow y)}{x = y}\ \mathsf{fwd}$$

$$\frac{\mathsf{proc}(x, x.l_k \;;\; P) \quad \mathsf{proc}(z, \mathsf{case}\ x\ (l_i \Rightarrow Q_i)_{i \in I})}{\mathsf{proc}(x, P) \quad \mathsf{proc}(z, Q_k)}\ \oplus C$$

$$\frac{\mathsf{proc}(x, \mathsf{case}\ x\ (l_i \Rightarrow P_i)_{i \in I}) \quad \mathsf{proc}(z, x.l_k \;;\; Q)}{\mathsf{proc}(x, Q) \quad \mathsf{proc}(z, P_k)}\ \& C$$

$$\frac{\mathsf{proc}(x, \mathsf{close}\ x) \quad \mathsf{proc}(z, \mathsf{wait}\ x \;;\; Q)}{\mathsf{proc}(z, Q)}\ \mathbf{1}C$$

$$\frac{\mathsf{proc}(x, y \leftarrow \mathsf{recv}\ x \;;\; P_y) \quad \mathsf{proc}(z, \mathsf{send}\ x\ w \;;\; Q)}{\mathsf{proc}(x, P_w) \quad \mathsf{proc}(z, Q)}\ /C, \backslash C$$

$$\frac{\mathsf{proc}(x, \mathsf{send}\ x\ w \;;\; P) \quad \mathsf{proc}(z, y \leftarrow \mathsf{recv}\ x \;;\; Q_y)}{\mathsf{proc}(P) \quad \mathsf{proc}(Q_w)}\ \bullet C, \circ C$$

3 Example: Lists

We are used to lists being a data structure; here it describes the behavior of a process which (essentially) sends a sequence of elements. Looking back at the rules, we see that this requires either fuse or twist, and we choose fuse. It therefore goes beyond the subsingleton fragment.

$$\mathsf{list}_A = \oplus\{\mathsf{cons} : A \bullet \mathsf{list}_A, \mathsf{nil} : \mathbf{1}\}$$

Lists are polymorphic in the sense the type of all elements in a list must be the same, but arbitrary, session type A. We indicate this with a subscript A on the list type, which therefore represents really a whole collection of types. We might decide to formally introduce first class type constructors and explicit polymorphism at a later time.

Our first program will be to append two lists. We will write it in stages.

```
$(l_1{:}\mathsf{list}_A)\ (l_2{:}\mathsf{list}_A) \vdash \mathit{append} : (l{:}\mathsf{list}_A)$
$\mathit{append} = \mathsf{case}\ l_1$  $(\mathsf{cons} \Rightarrow \ldots$        %   $(l_1{:}A \bullet \mathsf{list}_A)\ (l_2{:}\mathsf{list}_A) \vdash l{:}\mathsf{list}_A$
                    $\mid \mathsf{nil} \Rightarrow \ldots)$
```

We have written in the type of the first hole, indicated by the first ellipsis, that we are working to fill. From the type we can see that l_1 will send us a channel x of type A, and since it is $A \bullet \mathsf{list}_A$ we are required to add this to the left of l_1, which yields:

```
(l₁:list_A) (l₂:list_A) ⊢ append : (l:list_A)
append = case l₁  (cons ⇒ x ← recv l₁ ;
                           ...      %   (x:A) (l₁:list_A) (l₂:list_A) ⊢ l:list_A
                  | nil ⇒ ...)
```

At this point we know the result list will start with x, and fortunately we can find x at the left end of the context. So we can send it along x, after we indicate the result starts with an element by sending the label cons.

```
(l₁:list_A) (l₂:list_A) ⊢ append : (l:list_A)
append = case l₁  (cons ⇒ x ← recv l₁ ;
                          l.cons ; send l x ;
                           ...      %   (l₁:list_A) (l₂:list_A) ⊢ l:list_A
                  | nil ⇒ ...)
```

Now we are back to the original type and we can make a recursive call in this branch of the case expression.

```
(l₁:list_A) (l₂:list_A) ⊢ append : (l:list_A)
append = case l₁  (cons ⇒ x ← recv l₁ ;
                          l.cons ; send l x ;
                          append
                  | nil ⇒ ...       %   (l₁:1) (l₂:list_A) ⊢ l:list_A
                  )
```

We have filled in the type of l_1 in the second branch of the case expression. The code is now easy: we wait for l_1 to terminate and then implement l as l_2 by forwarding. This is possible since their types match.

```
(l₁:list_A) (l₂:list_A) ⊢ append : (l:list_A)
append = case l₁  (cons ⇒ x ← recv l₁ ;
                          l.cons ; send l x ;
                          append
                  | nil ⇒ wait l₁ ; l ← l₂)
```

It is very easy to imagine some syntactic sugar, where consecutive sends and receives of labels and channels are combined. For example:

$(l_1{:}\mathsf{list}_A)\ (l_2{:}\mathsf{list}_A) \vdash \mathit{append} : (l{:}\mathsf{list}_A)$
$\mathit{append} = \mathsf{case}\ l_1\ \ (\mathsf{cons}(x) \Rightarrow l.\mathsf{cons}(x)\ ;\ \mathit{append}$
$\qquad\qquad\qquad\quad \mid \mathsf{nil}() \Rightarrow l \leftarrow l_2)$

We refrain from such niceties because it obscures the structure of communication.

However, we will make one change. There is an oddity in our code in that the type declaration for *append* declares channel variables l_1, l_2 and l which appear free in the definition. Really, they should be somehow *bound* so that the definition is self-contained. Our notation for a process name X with type declaration

$$(x_1{:}A_1) \ldots (x_n{:}A_n) \vdash X :: (x{:}A)$$

is

$$x \leftarrow X \leftarrow x_1 \ldots x_n = P(x, x_1, \ldots, x_n)$$

thereby writing the provided channel x like a return value and the used channels channel and $x_1, \ldots, x_n$ like arguments to X. Note that $x, x_1, \ldots x_n$ are *bound* channel names occuring in the body P and can be renamed as convenient. In this form, which we will use from now on, we have

```
$(l_1{:}\mathsf{list}_A)\ (l_2{:}\mathsf{list}_A) \vdash \mathit{append} : (l{:}\mathsf{list}_A)$
$l \leftarrow \mathit{append} \leftarrow l_1\ l_2 =$
  $\mathsf{case}\ l_1\ \ (\mathsf{cons} \Rightarrow x \leftarrow \mathsf{recv}\ l_1\ ;$
                  $l.\mathsf{cons}\ ;\ \mathsf{send}\ l\ x\ ;$
                  $l \leftarrow \mathit{append} \leftarrow l_1\ l_2$
            $\mid \mathsf{nil} \Rightarrow \mathsf{wait}\ l_1\ ;\ l \leftarrow l_2)$
```

Note that the recursive calls also now specifies the offered channels l and the used channels l_1 and l_2.

In practice, is it convenient to fold in an appeal to a defined process name with a cut. More formally, we generalize definitions and the def rules for defined process names to be as follows:

$$\frac{\mathsf{proc}(y \leftarrow X \leftarrow y_1 \ldots y_n\ ;\ Q_y) \quad \underline{\mathsf{def}}(x \leftarrow X \leftarrow x_1 \ldots x_n = P_{x,x_1,\ldots,x_n})}{\mathsf{proc}(P_{w,y_1,\ldots,y_n}) \quad \mathsf{proc}(Q_w)}\ \mathsf{def}^w$$

At first it might look like there could be many processes of the type

$(l_1{:}\mathsf{list}_A)\ (l_2{:}\mathsf{list}_A) \vdash X : (l{:}\mathsf{list}_A)$

for example, taking alternating elements from l_1 and l_2, or appending l_2 to l_1. I believe that none of these would be well-typed, and essentially the only possible behavior is to append l_1 and l_2 or diverge at some point. This is pure conjecture, since at this point we have not developed the necessary theories of observational equivalence and parametricity that might allow us to prove such a result. Similarly, I would conjecture that $(t{:}\mathsf{list}_A) \vdash X :: (l{:}\mathsf{list}_A)$ would force X to be observationally equivalent to the identity and could not, for example, reverse t. Occasionally we will want to reverse lists, which means eventually we will have to leave the confines of ordered concurrency. But let's first see what we can write here.

Next we think about writing *processes nil* and *cons* that behave like the empty list, or add an element to a given list. We will write these in one installment.

```
$\cdot \vdash \mathit{nil} : (l{:}\mathsf{list}_A)$
$l \leftarrow \mathit{nil} =$
  $l.\mathsf{nil}$ ; $\mathsf{close}\ l$
```

```
$(x{:}A)\ (t : \mathsf{list}_A) \vdash \mathit{cons} : (l{:}\mathsf{list}_A)$
$l \leftarrow \mathit{cons} \leftarrow x\ t =$
  $l.\mathsf{cons}$ ; $\mathsf{send}\ l\ x$ ; $l \leftarrow t$
```

Note that there seems to be no possible implementation if we *reverse* the arguments to cons.

```
$(t : \mathsf{list}_A)\ (x{:}A) \vdash \mathit{cons}' : (l{:}\mathsf{list}_A)$
$l \leftarrow \mathit{cons}' \leftarrow x\ t =$
  $l.\mathsf{cons}$ ; $\mathsf{send}\ l\ x$    ??
```

The send is ill-typed since x is at the wrong end of the context for the type $l : A \bullet \mathsf{list}_A$. The ordering constraints impose a tight discpline on the use of channels. See also Exercise 2.

In the next lecture we will write some more programs along these lines.

Exercises

Exercise 1 Reconsider the proposition $\bot$ from Lecture 5 and write out logical rules as well as an operational semantics via rules of linear inference. Implicitly this means that all the other rules with a parametric succedent (usuall denoted by C or $z{:}C$) should be generalized to also allow an empty succedent (which you do not need to rewrite). Operationally, this corresponds to a detached process that has no client, but may use other processes.

Exercise 2 Can you write an intuititively meaningful process *cons*′ with

$$(t : \mathsf{list}_A)\ (x{:}A) \vdash \mathit{cons}' : (l{:}\mathsf{list}_A)$$

If so, show the definition and explain what it does. If not, explain why you think it might be impossible to write a process of this type.

Exercise 3 Define

$$\mathsf{tsil}_A = \oplus\{\mathsf{snoc} : A \circ \mathsf{tsil}_A, \mathsf{lin} : \mathbf{1}\}$$

Define processes *dneppa*, *lin*, and *snoc* that mirror *append*, *nil* and *cons*.

Exercise 4 Define

$$\mathsf{dlist}_A = \oplus\{\mathsf{cons} : A \bullet \mathsf{dlist}_A, \mathsf{snoc} : A \circ \mathsf{dlist}_A, \mathsf{nil} : \mathbf{1}\}$$

Explore the behavior of this type, and which kinds of operations can be defined over this type. Form some conjectures about which operations may be impossible.

Exercise 5 We would like to define a type $\%A$ which behaves exactly like A once it has synchronized with a client at the same type. No information except that the client has arrived at a matching point will be exchanged.

1. Give logical $\%R$ and $\%L$ rules.
2. Assign process expressions to these rules.
3. Provide the operational $\%C$ rule using linear inference.
4. Can you define $\%A$ in terms of other connectives in ordered logic with a corresponding operational behavior?

References

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