Lecture Notes on
Ordered Programming

15-816: Substructural Logics
Frank Pfenning

Lecture 10
Thursday, September 29

We begin the lecture by continuing to program in ordered logic using the example of *list segments*, whose imperative versions have recently become popular in verification.

Then we return to the operational reading of the original, general rules for $\backslash L$, $/L$, $\bullet R$ and $\circ R$ in response to a question from an earlier lecture.

Finally we will sketch the proof of *progress* for ordered logic considered as a programming language.

1 List Segments

A list segment is the beginning of a list without its tail. It becomes a list once a tail is supplied. Functionally, it can be seen as function from a list to a list; here it will be a process that when given a list on its right will behave like a list.

$$\mathsf{seg}_A = \mathsf{list}_A \mathbin{/} \mathsf{list}_A$$

We discussed this in response to a question on how we might get direct access to the end of a queue, since our implementation of the $\mathtt{queue}_A$ interface actually had to pass any newly inserted element down a chain of processes. As we will see it doesn't exactly serve the same purpose, but first let's program.

We begin with the empty segment. If we append a tail t the empty segment becomes the list t. Our definition of seg_A is transparent, so that we will silently replace it by its definition as was our habit earlier in this course.

```
$\cdot \vdash \mathit{empty} :: (s : \mathsf{seg}_A)$
$s \leftarrow \mathit{empty} =$                $\%\quad \cdot \vdash s : \mathsf{list}_A \,/\, \mathsf{list}_A$
   $t \leftarrow \mathsf{recv}\ s$ ;              $\%\quad t{:}\mathsf{list}_A \vdash s{:}\mathsf{list}_A$
   $s \leftarrow t$
```

Concatenating two lists is straightforward, and the code more or less writes itself if we heed the types.

```
$(s_1{:}\mathsf{seg}_A)\ (s_2{:}\mathsf{seg}_A) \vdash \mathit{concat} :: (s : \mathsf{seg}_A)$
$s \leftarrow \mathit{concat} \leftarrow s_1\ s_2 =$
   $t \leftarrow \mathsf{recv}\ s$ ;              $\%\quad (s_1{:}\mathsf{list}_A \,/\, \mathsf{list}_A)\ (s_2{:}\mathsf{list}_A \,/\, \mathsf{list}_A)\ (t{:}\mathsf{list}_A) \vdash s : \mathsf{list}_A$
   $\mathsf{send}\ s_2\ t$ ;                 $\%\quad (s_1{:}\mathsf{list}_A \,/\, \mathsf{list}_A)\ (s_2{:}\mathsf{list}_A) \vdash s : \mathsf{list}_A$
   $\mathsf{send}\ s_1\ s_2$ ;                $\%\quad s_1{:}\mathsf{list}_A \vdash s : \mathsf{list}_A$
   $s \leftarrow s_1$
```

Next, we can *prepend* an element to a segment to obtain another segment, which means we add it to the *front* of the given segment.

```
$(x{:}A)\ (s'{:}\mathsf{seg}_A) \vdash \mathit{prepend} :: (s : \mathsf{seg}_A)$
$s \leftarrow \mathit{prepend} \leftarrow x\ s' =$
   $t \leftarrow \mathsf{recv}\ s$ ;              $\%\quad (x{:}A)\ (s'{:}\mathsf{list}_A \,/\, \mathsf{list}_A)\ (t{:}\mathsf{list}_A) \vdash s : \mathsf{list}_A$
   $\mathsf{send}\ s'\ t$ ;                  $\%\quad (x{:}A)\ (s'{:}\mathsf{list}_A) \vdash s : \mathsf{list}_A$
   $s'' \leftarrow \mathsf{cons} \leftarrow x\ s'$         $\%\quad s''{:}\mathsf{list}_A \vdash s : \mathsf{list}_A$
   $s \leftarrow s''$
```

For reasons of symmetry with the next case, we have not combined the last two lines into the simpler tail call $s \leftarrow \mathsf{cons} \leftarrow x\ s'$.

Appending an element to the end of a segment is similar. It will still come before (the absent) tail. This is right behavior, since a segment with x appended still accepts a tail to come *after* x. Note that we have to be careful to state the arguments to *postpend* in the right order.

```
$(s'{:}\mathsf{seg}_A)\ (x{:}A) \vdash \mathit{postpend} :: (s : \mathsf{seg}_A)$
$s \leftarrow \mathit{postpend} \leftarrow s'\ x =$
   $t \leftarrow \mathsf{recv}\ s$ ;              $\%\quad (s'{:}\mathsf{list}_A \,/\, \mathsf{list}_A)\ (x{:}A)\ (t{:}\mathsf{list}_A) \vdash s : \mathsf{list}_A$
   $t' \leftarrow \mathsf{cons} \leftarrow x\ t$          $\%\quad (s'{:}\mathsf{list}_A \,/\, \mathsf{list}_A)\ (t'{:}\mathsf{list}_A) \vdash s : \mathsf{list}_A$
   $\mathsf{send}\ s'\ t'$ ;                 $\%\quad s'{:}\mathsf{list}_A \vdash s : \mathsf{list}_A$
   $s \leftarrow s'$
```

What can we do with a segment? We can create an empty segment and then add elements to the left and right ends. In that way, it is almost like a double-ended queue. However, we cannot remove elements. Instead, at some point we need to convert the element to a list by appending an empty list, after which we can no longer (easily) access the tail (see Exercise 1).

```
$s{:}\mathsf{seg}_A \vdash \mathit{seg\_to\_list} :: (l : \mathsf{list}_A)$
$l \leftarrow \mathit{seg\_to\_list} \leftarrow s =$
   $n \leftarrow \mathit{nil}$ ;              % $(s{:}\mathsf{list}_A \,/\, \mathsf{list}_A)\ (n{:}\mathsf{list}_A) \vdash l : \mathsf{list}_A$
   send $s\ n$ ;              % $s{:}\mathsf{list}_A \vdash l : \mathsf{list}_A$
   $l \leftarrow s$
```

2 An Operational Reading of $\bullet R$

Recall that we restricted the general form $\bullet R$ to $\bullet R^*$:

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

In the presence of cut and identity, we can interderive these rules so the same sequents remain provable.

$$\frac{\dfrac{}{A \vdash A}\ \mathsf{id}_A \qquad \Omega_2 \vdash B}{A\ \Omega_2 \vdash A \bullet B}\ \bullet R \qquad\qquad \frac{\Omega_1 \vdash A \qquad \dfrac{\Omega_2 \vdash B}{A\ \Omega_2 \vdash A \bullet B}\ \bullet R^*}{\Omega_1\ \Omega_2 \vdash A \bullet B}\ \mathsf{cut}_A$$

But what is the right operational reading of the fully general rule, and how does it relate to the more restricted one? Let's take a look at the operational interpretation of the proof on the right, which is available to us in ordered programming since it only uses $\bullet R^*$. From this, we may be able to glean the right interpretation of $\bullet R$.

$$\frac{\Omega_1 \vdash Q_y :: (y : A) \qquad \dfrac{\Omega_2 \vdash P :: (x : B)}{(y{:}A)\ \Omega_2 \vdash (\mathsf{send}\ x\ y\ ;\ P) :: (x : A \bullet B)}\ \bullet R^*}{\Omega_1\ \Omega_2 \vdash (y \leftarrow Q_y\ ;\ \mathsf{send}\ x\ y\ ;\ P) :: (x : A \bullet B)}\ \mathsf{cut}_A$$

The program here will create a new channel w, spawn a new process Q_w, send w along x and continue as P. We invent a new notation for the general $\bullet R$ to accomplish the same interactions.

$$\frac{\Omega_1 \vdash Q :: (y : A) \quad \Omega_2 \vdash P :: (x : B)}{\Omega_1\ \Omega_2 \vdash (\mathsf{send}\ x\ (y \leftarrow Q_y)\ ;\ P) :: (x : A \bullet B)}\ \bullet R$$

In an *asynchronous* model of communication these two behave exactly the same. Under a *synchronous* interpretation, there is a small difference. Here is the generalized communication rule for $\bullet$ (which always creates a fresh channel in the conclusion):

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

We can see that Q_w does not start to execute until this synchronous communication actually takes place. However, when executing our derived form (on the top line)

$$\frac{\mathsf{proc}(x, y \leftarrow Q_y\ ;\ \mathsf{send}\ x\ y\ ;\ P)}{\mathsf{proc}(x, \mathsf{send}\ x\ w\ ;\ P) \quad \mathsf{proc}(w, Q_w)}\ \mathsf{cmp}^w$$

a new channel w will always be created and Q_w starts immediately, whether a client is ready to communicate along x or not. If there is a client ready it can then immediately step to the same configuration as the $\bullet C^w$ rule.

With appropriate small changes, the same construction can be used for $\circ R$, $/L$ and $\backslash L$ (see Exercise 4).

Why did we choose the simplified forms of these constructs? One reason is that all the logical rules just send or receive some information and continue, but do nothing else. The rules for internal ($\oplus$) and external ($\&$) choice send or receive a label, the rules for the unit ($\mathbf{1}$) send or receive a end-of-communication token, the rules for $/$, $\backslash$, $\bullet$, and $\circ$ send or receive a channel. Composition (cut) is solely responsible for spawning new processes and forwarding (id) terminates a process (as does $\mathbf{1}R$, for a different reason).

With the general rules, the multiplicative connectives ($/, \backslash, \bullet, \circ$) also spawn new processes, which complicates reasoning about their operational behavior. The big advantage, however, is that the rules are directly derived from the sequent calculus and therefore satisfy full cut elimination, which, as we have seen, is not the case for the restricted rules. But since we view them operationally, from the perspective of a programming language, we are more interested in progress and preservation properties. And these still hold, because cut reduction and identity expansion still hold.

3 Progress

We now would like to generalize the proof of progress from the subsingleton to the ordered case. This means the scaffolding around the key insight, namely that cut reduction holds for each connective, has to be generalized.

We begin by typing configurations. While configurations are *linear* rather than *ordered*, we still need type checking to proceed in some order. So we may need to "permute" the configuration so that the following rules apply. The typing derivation for a configuration then fixes some order. At the top level we use this for a configuration executing a single process offering along a single channel.

$$\models \mathsf{proc}(x, P) :: (x : A)$$

We need to generalize almost immediately if P spawns a new process. Then then have

$$\models \mathcal{C} :: (x : A)$$

that is, we have multiple running processes but, overall, we can interact with the configuration only along one channel. If P spans multiple processes, say $P_1, \ldots, P_n$, then we need to check all of them. While each of them offers along a single channel, collectively they offer along a whole sequence of channels, so we end up with the judgment

$$\models \mathcal{C} :: \Omega$$

allowing both multiple processes in the configuration and multiple channels that they provide, namely all the ones in Ω. This judgment is defined by the following two rules:

$$\frac{}{\models (\cdot) :: (\cdot)} \qquad \frac{\models \mathcal{C} :: \Omega\ \Omega' \quad \Omega' \vdash P :: (x : A)}{\models \mathcal{C}\ \mathsf{proc}(x, P) :: \Omega\ (x{:}A)}$$

This means that our particular arrangement of the configuration will have to list processes in dependency order, with a provider always preceding (looking left to right) its client. This corresponds to a pre-order traversal of the dependency tree where we traverse the subtrees from right to left.[1]

The key is now to come up with the correct induction hypothesis to prove progress. We introduce the in-line notation $\mathcal{C} \longrightarrow \mathcal{D}$ for the clumsier

$$\frac{\mathcal{C}}{\mathcal{D}}$$

We need one lemma, whose role will only become clear in the proof. It allows us to extract a typing derivation for the offering process for a channel $x{:}A$ that's provided by a configuration.

[1]In lecture, I had a slightly more general rule which also worked out, but was unnecessarily complicated (as suggested by several students).

Lemma 1 (Configuration Inversion)
If $\models \mathcal{C} :: \Omega_1\ (x{:}A)\ \Omega_2$ *then* $\mathcal{C} = (\mathcal{C}_1\ \mathsf{proc}(x, P)\ \mathcal{C}_2)$
with $\models \mathcal{C}_1 :: \Omega_1\ \Omega'$ *and* $\Omega' \vdash P :: (x : A)$ *for some* $\mathcal{C}_1$*,* P*,* $\mathcal{C}_2$*, and* Ω'*.*

Proof: By induction on the structure of the given typing derivation (see Exercise 5). □

Theorem 2 *If* $\models \mathcal{C} :: \Omega$ *then either*

(i) $\mathcal{C} \longrightarrow \mathcal{D}$ *for some* $\mathcal{D}$*, or*

(ii) all processes in $\mathcal{C}$ *are executing a right rule.*

Proof: By induction on the structure of the typing derivation $\models \mathcal{C} :: \Omega$.

Case: The empty configuration. Then *(i)* holds.

Case: $\mathcal{C} = \mathcal{C}'\ \mathsf{proc}(x, P)$ and

$$\models \mathcal{C}' :: \Omega'\ \Omega^* \qquad\qquad \Omega^* \vdash P :: (x{:}A)$$

We first consider some subcases for P.

Subcase: P ends in a cut. Then *(i)* holds because P can transition.

Subcase: P ends in an identity. Then *(i)* holds because P can transition.

Subcase: P is a defined name. Then *(i)* holds because P can transition.

In the remaining cases we can now assume that P is not a cut, identity, or name. In other words, it must end in a logical rule. We now appeal to the induction hypothesis and get two further subcases.

Subcase: $\mathcal{C}' \longrightarrow \mathcal{D}'$ for some $\mathcal{D}'$. Then also $\mathcal{C} \longrightarrow \mathcal{D}'\ \mathsf{proc}(x, P)$ so *(i)* holds.

Subcase: All processes in $\mathcal{C}'$ execute a right rule.

At this point we further descend in our tree of case distinctions: P executes either a right rule or a left rule.

Subcase: P executes a right rule. Then all processes in $\mathcal{C} = \mathcal{C}'\ \mathsf{proc}(x, P)$ execute a right rule and *(ii)* holds.

Subcase: P executes a left rule.

Sadly, we now have to make further distinctions as to which left rule P executes. We consider only one case; the others are analogous.

Subcase: P executes $\backslash L^*$. Then $P = (\mathsf{send}\ y\ w\ ;\ P')$, $\Omega^* = \Omega_L^*\ (w{:}B)\ (y : B \backslash C)\ \Omega_R^*$,

$$\frac{\Omega_L^*\ (y{:}C)\ \Omega_R^* \vdash P' :: (x : A)}{\Omega_L^*\ (w{:}B)\ (y : B \backslash C)\ \Omega_R^* \vdash \mathsf{send}\ y\ w\ ;\ P' :: (x : A)}\ \backslash L^*$$

and

$$\models \mathcal{C}' :: \Omega'\ \Omega_L^*\ (w{:}B)\ (y : B \backslash C)\ \Omega_R^*$$

By configuration inversion (Lemma 1) the derivation of the latter contains a typing derivation for the provider of $y : B \backslash C$

$$\Omega'' \vdash Q :: (y : B \backslash C)$$

for some Ω'' and Q. We also know in this subcase that Q executes a right rule. By inversion, this must be $\backslash R$, and we get $Q = (z \leftarrow \mathsf{recv}\ y\ ;\ Q'_z)$ and

$$\frac{(z{:}B)\ \Omega'' \vdash Q'_z :: (y : C)}{\Omega'' \vdash (z \leftarrow \mathsf{recv}\ y\ ;\ Q'_z) :: (y : B \backslash C)}\ \backslash R$$

Unraveling this, the original configuration has the form

$$\mathcal{C} = \mathcal{C}'_1\ \mathsf{proc}(y, z \leftarrow \mathsf{recv}\ y\ ;\ Q'_z)\ \mathcal{C}'_2\ \mathsf{proc}(x, \mathsf{send}\ y\ w\ ;\ P')$$

where $\mathcal{C}'_1$ and $\mathcal{C}'_2$ come from the appeal to configuration inversion. Hence we can finally infer

$$\frac{\mathcal{C}'_1\ \mathsf{proc}(y, z \leftarrow \mathsf{recv}\ y\ ;\ Q'_z)\ \mathcal{C}'_2\ \mathsf{proc}(x, \mathsf{send}\ y\ w\ ;\ P')}{\mathcal{C}'_1\ \mathsf{proc}(y, Q'_w)\ \mathcal{C}'_2\ \mathsf{proc}(x, P')}\ \backslash C$$

and clause *(i)* holds.

□

Exercises

Exercise 1 Write an ordered program to convert a list back into a segment.

$$l{:}\mathsf{list}_A \vdash \mathit{list\_to\_seg} :: (s : \mathsf{seg}_A)$$

Characterize work and span (= reaction time under maximal parallelism) of this operation when provided with a list of length n?

Exercise 2 We cannot write a direct analogue for the functional *map* from a functional language, because the function f that is mapped over a data structure is used potentially many times, violating linearity and order. In order to circumvent this problem, we define a *mapper* to be a process that can transform inputs of type A to outputs of B arbitrarily often.

$$\mathsf{mapper}_{AB} = \&\{\mathsf{next} : (B \bullet \mathsf{mapper}_{AB}) \mathbin{/} A, \mathsf{done} : \mathbf{1}\}$$

1. Define a process *map* with type

   $$(m{:}\mathsf{mapper}_{AB})\ (l{:}\mathsf{list}_A) \vdash \mathit{map} :: (k : \mathsf{list}_B)$$

   that applies mapper m to each element of list l to produce list k.

2. Define a mapper *map_id* such that

   $l{:}\mathsf{list}_A \vdash \mathit{identity} :: (k : \mathsf{list}_A)$
   $k \leftarrow \mathit{identity} \leftarrow l =$
   $\quad m \leftarrow \mathit{map\_id}\ ;$
   $\quad k \leftarrow \mathit{map} \leftarrow m\ l$

   does indeed behave like the identity. State both the type of *map_id* and its definition.

Exercise 3 We cannot write a direct process analogue of the function *fold* for the same reason as for *map* in Exercise 2.

1. Devise a type folder_{AB} that can reduce a list of type A to a result of type B. We suggest you study mapper_{AB} from Exercise 2 for hints on how to proceed.

2. Define a process *fold* with type

   $$(b{:}B)\ (m{:}\mathsf{folder}_{AB})\ (l{:}\mathsf{list}_A) \vdash \mathit{fold} :: (r : B)$$

   that folds m over the list l with initial value b to produce r. Feel free to change the order of the antecedents if another order turns out to be more convenient.

3. Define a folder *fold_id* and complete the following program such that

   $l{:}\mathsf{list}_A \vdash \mathit{identity} :: (k : \mathsf{list}_A)$
   $k \leftarrow \mathit{identity} \leftarrow l =$
   $\quad f \leftarrow \mathit{fold\_id}\ ;$
   $\quad \ldots$

   does indeed behave like the identity. State both the type of *fold_id* and its definition.

Exercise 4 Give a direct proof term assignment for the general $/L$ using the new form of process

$$\mathsf{send}\ x\ (y \leftarrow Q_y)\ ;\ P$$

from Section 2.

Exercise 5 Prove configuration inversion (Lemma 1).