Lecture Notes on Reasoning about Computation

15-816: Substructural Logics
Frank Pfenning

Lecture 6
September 15, 2016

In this lecture we will begin with a summary of the correspondence between proofs and programs from last lectures and then establish some key properties of the programming language that we derived from subsingleton logic. As usual, we will go back and forth between computational and logical interpretations. The properties we show are the usual preservation and progress, where the first shows that types are preserved during computation, and the second shows that computation can make progress unless it attempts to communicate with the "outside world". Both of these properties emerge very naturally from the cut reduction properties we checked for subsingleton logic.

1 Concurrent Subsingleton Programs

Types	A	$::=$	$\oplus\{l_i : A_i\}_{i\in I}$	internal choice	
		$\mid$	$\&\{l_i : A_i\}_{i\in I}$	external choice	
		$\mid$	$\mathbf{1}$	termination	
Processes	P, Q	$::=$	$\leftrightarrow$	forward	id
		$\mid$	$(P \mid Q)$	compose	cut
		$\mid$	$\mathsf{R}.l_k\ ;\ P$	send label right	$\oplus R_k$
		$\mid$	$\mathsf{caseL}(l_i \Rightarrow Q_i)_{i\in I}$	receive label left	$\oplus L$
		$\mid$	$\mathsf{caseR}(l_i \Rightarrow P_i)_{i\in I}$	receive label right	$\&R$
		$\mid$	$\mathsf{L}.l_k\ ;\ Q$	send label left	$\&L_k$
		$\mid$	closeR	close and notify right	$\mathbf{1}R$
		$\mid$	$\mathsf{waitL}\ ;\ Q$	wait on close left	$\mathbf{1}L$

We also allow mutually recursive type definitions $\alpha = A$ which must be *contractive*, that is, A must be of the form $\oplus\{\ldots\}$, $\&\{\ldots\}$, or $\mathbf{1}$. We treat a type name as equal to its definition and will therefore silently replace it. The usual manner of making this more explicit is to use types of the form $\mu\alpha.A$, but we forego this exercise here.

Similarly, we allow mutually recursive process definitions of variables X as processes P in the form $\omega \vdash X = P : A$. Collectively, these constitute the program $\mathcal{P}$. We fix a global program $\mathcal{P}$ so that the typing judgment, formally, is $\omega \vdash_{\mathcal{P}} P : A$ where we assume that $\omega \vdash_{\mathcal{P}} Q : A$ for every definition $\omega \vdash X = Q : A$ in $\mathcal{P}$. Since $\mathcal{P}$ does not change in any typing

derivation, we omit this subscript in the rules.

$$\frac{}{A \vdash \leftrightarrow : A}\ \mathsf{id}_A \qquad \frac{\omega \vdash P : A \quad A \vdash Q : C}{\omega \vdash (P \mid Q) : C}\ \mathsf{cut}_A$$

$$\frac{\omega \vdash P : A_k \quad (k \in I)}{\omega \vdash (\mathsf{R}.l_k\ ;\ P) : \oplus\{l_i : A_i\}_{i \in I}}\ \oplus R_k \qquad \frac{A_i \vdash Q_i : C \quad (\text{for all } i \in I)}{\oplus\{l_i : A_i\}_{i \in I} \vdash \mathsf{caseL}(l_i \Rightarrow Q_i)_{i \in I} : C}\ \oplus L$$

$$\frac{\omega \vdash P_i : A_i \quad (\text{for all } i \in I)}{\omega \vdash \mathsf{caseR}(l_i \Rightarrow P_i)_{i \in I} : \&\{l_i : A_i\}_{i \in I}}\ \& R \qquad \frac{A_k \vdash Q : C \quad (k \in I)}{\&\{l_i : A_i\}_{i \in I} \vdash (\mathsf{L}.l_k\ ;\ Q) : C}\ \& L_k$$

$$\frac{}{\cdot \vdash \mathsf{closeR} : \mathbf{1}}\ \mathbf{1}R \qquad \frac{\cdot \vdash Q : C}{\mathbf{1} \vdash \mathsf{waitL}\ ;\ Q : C}\ \mathbf{1}L$$

$$\frac{(\omega \vdash X = P : A) \in \mathcal{P}}{\omega \vdash X : A}\ X$$

For the synchronous operational semantics presented via ordered inference, we use ephemeral propositions $\mathsf{proc}(P)$ which expresses the current state of an executing process P. We also import the process definitions $X = P$ as persistent propositions $\underline{\mathsf{def}}(X, P)$.

$$\frac{\mathsf{proc}(\leftrightarrow)}{\cdot}\ \mathsf{fwd} \qquad \frac{\mathsf{proc}(P \mid Q)}{\mathsf{proc}(P) \quad \mathsf{proc}(Q)}\ \mathsf{cmp}$$

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

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

$$\frac{\mathsf{proc}(\mathsf{closeR}) \quad \mathsf{proc}(\mathsf{waitL}\ ;\ Q)}{\mathsf{proc}(Q)}\ \mathbf{1}C$$

$$\frac{\mathsf{proc}(X) \quad \underline{\mathsf{def}}(X, P)}{\mathsf{proc}(P)}\ \mathsf{def}$$

A process configuration $\mathcal{C}$ consists of an ordered collection of $\mathsf{proc}(P)$ propositions. The typechecking judgment for configurations, $\omega \vdash \mathcal{C} : \omega'$ is

defined by the following rules that work through $\mathcal{C}$ from right to left.

$$\frac{}{\omega \vdash (\cdot) : \omega} \qquad \frac{\omega \vdash \mathcal{C} : \omega' \quad \omega' \vdash P : A}{\omega \vdash (\mathcal{C}\ \mathsf{proc}(P)) : A}$$

2 Type Preservation

We have already indicated the deeper reason type preservation holds in the last lecture: cut reduction (which is how computation mostly proceeds) preserves the endsequent of the proof and thereby the type of the process. Here, we go through the proof more rigorously.

Even though typing was defined from left to right, if we have a well-typed configuration any subconfiguration is also well-typed.

Lemma 1 (Configuration Typing)

1. *(*Split*) If* $\omega_L \vdash \mathcal{C}_L\ \mathcal{C}_R : \omega_R$ *then* $\omega_L \vdash \mathcal{C}_L : \omega_M$ *and* $\omega_M \vdash \mathcal{C}_R : \omega_R$ *for some* ω_M.
2. *(*Concatenation*) If* $\omega_L \vdash \mathcal{C}_L : \omega_M$ *and* $\omega_M \vdash \mathcal{C}_R : \omega_R$*, then* $\omega_L \vdash \mathcal{C}_L\ \mathcal{C}_R : \omega_R$.
3. *(*Singleton*)* $\omega \vdash \mathsf{proc}(P) : A$ *iff* $\omega \vdash P : A$

Proof: Split follows by induction on the structure of $\mathcal{C}_R$, concatenation by induction on the typing of $\mathcal{C}_R$, and singleton follows by inversion in one direction and by constructing the derivation in the other direction. □

Theorem 2 (Type Preservation) *If* $\omega \vdash \mathcal{C} : A$ *and* $\mathcal{C} \longrightarrow \mathcal{C}'$ *by one step of ordered inference, then* $\omega \vdash \mathcal{C}' : A$.

Proof: By split (Lemma 1), we have $\mathcal{C} = (\mathcal{C}_L\ \mathcal{C}_M\ \mathcal{C}_R)$ where $\omega_L \vdash \mathcal{C}_M : A_M$ is the premise of one of the computation rules. By concatenation (Lemma 1), we have preservation if we can show that the conclusion $\mathcal{C}'_M$ of the computation rule again has type $\omega_L \vdash \mathcal{C}'_M : A_M$.

We show only one case of this proof, since all others proceed analogously.

Case: $\mathcal{C}_M = (\mathsf{proc}(\mathsf{R}.l_k\ ;\ P)\ \mathsf{proc}(\mathsf{caseL}(l_i \Rightarrow Q_i)_{i \in I}))$.

$\omega_L \vdash \mathsf{proc}(\mathsf{R}.l_k\ ;\ P) : B$
and $B \vdash \mathsf{proc}(\mathsf{caseL}(l_i \Rightarrow Q_i)_{i \in I}) : A_M$ by inversion on typing of $\mathcal{C}_M$
$B = \oplus\{l_i : B_i\}_{i \in I}$ and $B_i \vdash Q_i : A_M$ for all $i \in I$
by inversion on typing of caseL
$k \in I$ and $\omega_L \vdash P : B_k$ by inversion on typing of $\mathsf{R}.l_k$
$B_k \vdash Q_k : A_M$ using $i = k$
$\omega_L \vdash (\mathsf{proc}(P)\ \mathsf{proc}(Q_k)) : A_M$ by concatenation

□

3 Progress

Progress means that a configuration will either try to communicate with the "outside world" at its endpoints, or it will be able to make a transition. It does not get stuck in some unexpected way. Just as preservation came down to the fact that cut reduction preserves the endsequent, progress comes down to the fact that every right rule for a connective matched with any left rule will be able to reduce. This was essentially our test whether the interpretation of the logical connectives is meaningful. Again, this intuitive argument is couched in an inductive proof.

Theorem 3 (Progress) *If $\omega \vdash \mathcal{C} : A$ then*

1. *either $\mathcal{C}$ can make a transition (by ordered inference),*
2. *or $\mathcal{C} = (\cdot)$ is empty,*
3. *or $\mathcal{C}$ attempts to communicate to the left (*caseL, $\mathsf{L}.l_k$*, or* waitL*),*
4. *or $\mathcal{C}$ attempts to communicate to the right (*caseR, $\mathsf{R}.l_k$*, or* closeR*).*

Proof: Perhaps surprisingly, the proof is by a simply structural induction on the typing of $\mathcal{C}$. Note that the four cases are not mutually exclusive. For example, the rightmost process in a configuration may want to communicate to the right while another part of the configuration transitions. This is the nature of concurrent computation.

Case:

$$\frac{}{A \vdash (\cdot) : A}$$

The $\mathcal{C} = (\cdot)$ and part 2 applies.

Case:

$$\frac{\omega \vdash \mathcal{C}' : \omega' \quad \omega' \vdash \mathsf{proc}(P) : A}{\omega \vdash (\mathcal{C}'\ \mathsf{proc}(P)) : A}$$

where $\mathcal{C} = (\mathcal{C}'\ \mathsf{proc}(P))$. First, if P is a forward ($\leftrightarrow$), composition ($P_1 \mid P_2$), or a defined name (X), it can make a transition and therefore also $\mathcal{C}$. Moreover, if P communicates to the right, then so does $\mathcal{C}$ and part 4 applies. So we can exclude these cases from consideration below and we may assume that P attempts to communicate to the left (caseL, L.l_k, or waitL).

From the induction hypothesis on the first premise, we know we can distinguish the following subcases.

Subcase: $\mathcal{C}'$ can make a transition. Then so can $\mathcal{C}$.

Subcase: $\mathcal{C}' = (\cdot)$. Then $\mathcal{C} = \mathsf{proc}(P)$. Since P communicates to the left, so does $\mathcal{C}$ and part 3 applies.

Subcase: $\mathcal{C}'$ attempts to communicate to the left. Then so does $\mathcal{C} = (\mathcal{C}'\ \mathsf{proc}(P))$.

Subcase: $\mathcal{C}'$ attempts to communicate to the right, that is, its rightmost process P' has the form caseR, R.l_k or closeR. We already know that P communicates to the left, which is one of caseL, L.l_k, or waitL. Now we apply inversion on the typing of P' and P taking advantage of the fact that the mediating type $\omega' = B$ on the right of P' and left of P must be the same. It emerges from this analysis that one of the remaining two-premise rules ($\oplus C$, $\& C$, or $\mathbf{1}C$) must be applicable. This is because once we fix one side and therefore the interface type B, the rule on the other side must be one of the cases we considered when checking cut reduction.

□

4 Example: Bit Strings Revisited

In this section we revisit the implementation of bit strings and increment from Lecture 2. Recall the ordered logic program for incrementing a bit string, now using \$ as the terminator. Recall that numbers are written with

propositions b0 for 0 and b1 for 1 and are shown with the least significant bit to the right.

$$\frac{\mathsf{b0\ inc}}{\mathsf{b1}}\ \mathsf{inc0} \qquad \frac{\mathsf{b1\ inc}}{\mathsf{inc\ b0}}\ \mathsf{inc1} \qquad \frac{\$\ \mathsf{inc}}{\$\ \mathsf{b1}}\ \mathsf{inc}\$$$

This is not quite a finite state transducer, since we may not read the whole input, but it is still straightforward to represent this as a proof in subsingleton logic. We recommend you try before reading on.

$$
\begin{array}{l}
\mathit{bits} = \oplus\{\mathsf{b0} : \mathit{bits}, \mathsf{b1} : \mathit{bits}, \$: \mathbf{1}\} \\
\mathit{bits} \vdash \mathsf{inc} : \mathit{bits} \\
\mathsf{inc} = \mathsf{caseL}\ (\mathsf{b0} \Rightarrow \mathsf{R.b1}\ ;\ \leftrightarrow \\
\qquad\qquad\quad \mid \mathsf{b1} \Rightarrow \mathsf{R.b0}\ ;\ \mathsf{inc} \\
\qquad\qquad\quad \mid \$ \Rightarrow \mathsf{R.b1}\ ;\ \mathsf{R.\$}\ ;\ \leftrightarrow)
\end{array}
$$

We now experiment with ways we can use logical tools to reason about this program. First, how can we define a type of *std* which corresponds to a bit string in *standard form*, that is, without leading 0 bits? Again, it is an excellent way to test your understanding to construct such a definition before moving on.

The key idea is that we have two mutually recursive types, *std* for any standard bit string and *pos* for a positive one, that is, not consisting only of $\mathsf{b0}$ and \$.

$$\begin{aligned} \mathit{std} = \oplus\{ & \mathsf{b0} : \mathit{pos}, \\ & \mathsf{b1} : \mathit{std}, \\ & \$: \mathbf{1}\} \end{aligned}$$

$$\begin{aligned} \mathit{pos} = \oplus\{ & \mathsf{b0} : \mathit{pos}, \\ & \mathsf{b1} : \mathit{std}\} \end{aligned}$$

Now the increment process should have type

$$\mathit{std} \vdash \mathsf{inc} : \mathit{pos}$$

which expresses that if we receive any number in standard form from the left we will send a positive number in standard form to the right.

Before we proceed with checking the definition, one observation: if $\omega \vdash P : \mathit{std}$ then also $\omega \vdash P : \mathit{bits}$ no matter what ω and P are. This is easy to see, since *std* represents a sequence of 0's and 1's followed by \$ while *std* is just a restricted form of such a sequence. We say *std* is a *subtype* of *bits*, written $\mathit{std} \leq \mathit{bits}$. It turns out that subtyping is decidable [GH05] with an interesting algorithm we intend to return to in a later lecture. Here we just note that

$$\mathit{pos} \leq \mathit{std} \quad \text{and} \quad \mathit{std} \leq \mathit{bits}$$

We further note that in the presence of subtyping we can relax the identity rule to

$$\frac{A \leq A'}{A \vdash \leftrightarrow : A'}\ \mathsf{id}^{\leq}$$

Again, I strongly recommend writing out the typing derivation of the program yourself to check your understanding.

$$std = \oplus\{\mathsf{b0} : pos, \mathsf{b1} : std, \$: \mathbf{1}\}$$
$$pos = \oplus\{\mathsf{b0} : pos, \mathsf{b1} : std\}$$

$$std \vdash \mathsf{inc} : pos$$
$$\begin{array}{l}\mathsf{inc} = \mathsf{caseL}\ (\mathsf{b0} \Rightarrow \mathsf{R.b1}\ ;\ \leftrightarrow \\ \qquad\qquad\quad \mid \mathsf{b1} \Rightarrow \mathsf{R.b0}\ ;\ \mathsf{inc} \\ \qquad\qquad\quad \mid \$ \Rightarrow \mathsf{R.b1}\ ;\ \mathsf{R.\$}\ ;\ \leftrightarrow)\end{array}$$

$$\dfrac{\dfrac{\dfrac{pos \leq std}{pos \vdash \leftrightarrow : std}\ \mathsf{id}^{\leq}}{pos \vdash \mathsf{R.b1}\ ;\ \leftrightarrow : pos}\ \oplus R \quad \dfrac{std \vdash \mathsf{inc} : pos}{std \vdash \mathsf{R.b0}\ ;\ \mathsf{inc} : pos}\ \oplus R \quad \dfrac{\dfrac{\dfrac{}{\mathbf{1} \vdash \leftrightarrow : \mathbf{1}}\ \mathsf{id}}{\mathbf{1} \vdash \mathsf{R.\$}\ ;\ \leftrightarrow : std}\ \oplus R}{\mathbf{1} \vdash \mathsf{R.b1}\ ;\ \mathsf{R.\$}\ ;\ \leftrightarrow : pos}\ \oplus R}{std \vdash \mathsf{caseL}(\mathsf{b0} \Rightarrow \mathsf{R.b1}\ ;\ \leftrightarrow \mid \mathsf{b1} \Rightarrow \mathsf{R.b0}\ ;\ \mathsf{inc} \mid \$ \Rightarrow \mathsf{R.b1}\ ;\ \mathsf{R.\$}\ ;\ \leftrightarrow) : pos}\ \oplus L$$

When we arrive at process names X such as inc, we accept a globally asserted type. This is fine, as long as we make sure that we check all definitions and mirrors the same approach for recursively defined functions in functional languages.

5 Implementing Turing Machines

We have already established that Turing machines can be easily implemented using ordered inference, and that finite state transducers can be implemented using ordered inference as well as ordered proofs. Can we also implement Turing machines, following the same ideas? The answer is yes; our development roughly follows [DP16]. Here is a summary of the representation of Turing machines using ordered inference. The initial configuration is represented by the ordered context

$$\$\ q_0\ \triangleright\ a_1\ \ldots\ a_n\ \$$$

and the final configuration as

$$\$\ b_1\ \ldots\ b_k\ \$$$

and we go from the first to the last by a process of ordered inference using the following rules:

For $\delta(q,a) = (q', a', R)$:
$$\frac{q\ \triangleright\ a}{a'\ q'\ \triangleright}\ \mathsf{LRMR} \qquad \frac{a\ \triangleleft\ q}{a'\ q'\ \triangleright}\ \mathsf{LLMR}$$

For $\delta(q,a) = (q', a', L)$:
$$\frac{q\ \triangleright\ a}{\triangleleft\ q'\ a'}\ \mathsf{LRML} \qquad \frac{a\ \triangleleft\ q}{\triangleleft\ q'\ a'}\ \mathsf{LLML}$$

To extend the tape:
$$\frac{\triangleright\ \$}{\triangleright\ \sqcup\ \$}\ \mathsf{ER} \qquad \frac{\$\ \triangleleft}{\$\ \sqcup\ \triangleleft}\ \mathsf{EL}$$

To halt in final state:
$$\frac{\triangleleft\ q_f}{\cdot}\ \mathsf{FL} \qquad \frac{q_f\ \triangleright}{\cdot}\ \mathsf{FR}$$

We will represent the tape head together with the direction symbol as a process, so that for every state q_k in the machine we have two definitions, ${}^{<}Q_k$ and $Q_k{}^{>}$. What should their types be? A process ${}^{<}Q_k$ will have to receive a tape symbol or endmarker from the left, so its type should be

$$\begin{array}{l} \mathit{tape} = \oplus_{i\in\Sigma}\{a_i : \mathit{tape}, \$: \mathbf{1}\} \\ \mathit{tape} \vdash {}^{<}Q_i : ? \end{array}$$

On the other hand, a right-looking process $Q_k{}^{>}$ will have to receive a tape symbol or endmarker from the right, so its type should be

$$\begin{array}{l} \mathit{epat} = \&_{i\in\Sigma}\{a_i : \mathit{epat}, \$: \mathbf{1}\} \\ ? \vdash Q_i{}^{>} : \mathit{epat} \end{array}$$

It seems advisable for both kinds of processes to have the same type so we can transition easily between them, which gives us

$$\begin{array}{l} \mathit{tape} \vdash {}^{<}Q_i : \mathit{epat} \\ \mathit{tape} \vdash Q_i{}^{>} : \mathit{epat} \end{array}$$

Now consider one case, looking left and moving left:

$$\frac{a \ \lhd \ q}{\lhd \ q' \ a'} \ \mathsf{LLML}$$

The code to effect this change should recognize an a from the left, then output an a' to the right, and then transition to ${}^{<}Q'$.

$$\begin{array}{l} \mathit{tape} \vdash {}^{<}Q : \mathit{epat} \\ {}^{<}Q = \mathsf{caseL}(a \Rightarrow \mathsf{R}.a' \ ; \ {}^{<}Q' \mid \ldots) \end{array}$$

However, there is a serious problem with this code. Can you spot it?

The problem is that it does not type-check! The type that governs Q's communication on the right is $\mathit{epat} = \&\{\ldots\}$ which prescribes *receiving* a symbol from the right rather than sending one! At first, this looks like it is very difficult to repair, but by employing cut we can overcome the problem.

The idea is to spawn a new process on the right whose sole job is to send the symbol a' to the left! Recalling the definition of *epat*, we have

$$\begin{array}{l}\mathit{epat} = \&_{i\in\Sigma}\{a_i : \mathit{epat}, \$: \mathbf{1}\} \\ \mathit{epat} \vdash \mathsf{L}.a' \; ; \leftrightarrow : \mathit{epat}\end{array}$$

We then rewrite the code snippet above as

$$\begin{array}{ll}\mathit{tape} \vdash {}^{<}Q : \mathit{epat} & \\ {}^{<}Q = \mathsf{caseL}(a \Rightarrow \mathsf{R}.a' \; ; {}^{<}Q' \mid \ldots) & \text{ill-typed!} \\ {}^{<}Q = \mathsf{caseL}(a \Rightarrow ({}^{<}Q' \mid (\mathsf{L}.a' \; ; \leftrightarrow)) \mid \ldots) & \text{well-typed!}\end{array}$$

The new cut is well-typed:

$$\dfrac{\mathit{tape} \vdash {}^{<}Q' : \mathit{epat} \quad \dfrac{\dfrac{}{\mathit{epat} \vdash \leftrightarrow : \mathit{epat}}\,\mathsf{id}}{\mathit{epat} \vdash (\mathsf{L}.a' \; ; \leftrightarrow) : \mathit{epat}}\,\&L_{a'}}{\mathit{tape} \vdash ({}^{<}Q' \mid (\mathsf{L}.a' \; ; \leftrightarrow)) : \mathit{epat}}\,\mathsf{cut}$$

With this idea we can easily fill in the four symmetric cases:

$$\dfrac{q \;\triangleright\; a}{a' \; q' \;\triangleright}\,\mathsf{LRMR} \qquad \dfrac{a \;\triangleleft\; q}{a' \; q' \;\triangleright}\,\mathsf{LLMR}$$

$$\dfrac{q \;\triangleright\; a}{\triangleleft \; q' \; a'}\,\mathsf{LRML} \qquad \dfrac{a \;\triangleleft\; q}{\triangleleft \; q' \; a'}\,\mathsf{LLML}$$

with the following snippets

$$\begin{array}{ll}\mathsf{LRMR} & Q^{>} = \mathsf{caseR}(\ldots \mid a \Rightarrow ((\mathsf{R}.a' \; ; \leftrightarrow) \mid Q'^{>}) \mid \ldots) \\ \mathsf{LLMR} & {}^{<}Q = \mathsf{caseL}(\ldots \mid a \Rightarrow ((\mathsf{R}.a' \; ; \leftrightarrow) \mid Q'^{>}) \mid \ldots) \\ \mathsf{LRML} & Q^{>} = \mathsf{caseR}(\ldots \mid a \Rightarrow ({}^{<}Q' \mid (\mathsf{L}.a' \; ; \leftrightarrow)) \mid \ldots) \\ \mathsf{LLML} & {}^{<}Q = \mathsf{caseL}(\ldots \mid a \Rightarrow ({}^{<}Q' \mid (\mathsf{L}.a' \; ; \leftrightarrow)) \mid \ldots)\end{array}$$

The same idea can be used to implement extension of the tape on the left and right. We have slightly rewritten the rules to account for the state q,

because the directional markers $\triangleleft$ and $\triangleright$ are not represented as processes.

$$\frac{q \triangleright \$}{q \triangleright \sqcup \$}\ \mathsf{ER} \qquad \frac{\$ \triangleleft q}{\$ \sqcup \triangleleft q}\ \mathsf{EL}$$

The rule snippets for this are part of Exercise 4

Termination of the machine can now no longer be forwarding (the way it was for the transducer), since the types on the left and right, *tape* and *epat*, respectively, are different. Instead, we could finish with an idling process, or we could traverse the tape to the left end or right end. Going to the right end makes sense if we want to pass on the result of the computation as a string. Going to the left end makes sense if we want to compose Turing machines and start the next machine once the current one has finished (see Exercises 4 and 5)

Exercises

Exercise 1 Show the following cases in the proof of preservation (Theorem 2)

1. Termination (rule $\mathbf{1}C$)
2. Composition (rule cmp)
3. Forwarding (rule fwd)

Exercise 2 Rewrite the code for inc in Section 4 so that forwarding ($\leftrightarrow$) is only used at type $\mathbf{1}$ by adding a second process copy that represents the identity function and calling it in the right place. Then

1. Show the typing derivation for $\mathit{std} \vdash \mathsf{inc} : \mathit{pos}$. Which type(s) do you need for copy?
2. Show the typing derivation(s) for copy.

In the above, forwarding is used only at type $\mathbf{1}$, so we should not need subtyping.

Exercise 3 Define type *even* and *odd* for bit strings (not necessarily in standard form) that represent even and odd binary numbers, respectively. Where they exist, provide the typing derivations for or explain why typing might fail. You may use subtyping if it turns out to be convenient.

$$\mathit{even} \vdash \mathsf{inc} : \mathit{odd}$$
$$\mathit{odd} \vdash \mathsf{inc} : \mathit{even}$$

Exercise 4 Complete the encoding of Turing machines in Section 5.

1. Give the code snippets for the ER and EL rules that extend the tape on the right and left end, respectively.
2. Provide the correct types and code for a final state Q_f that traverses the tape to reach the right endmarker and then terminates so that the final configuration Z behaves like the string

   $$\$\ a_1\ \cdots\ a_n$$

   and has type $\mathbf{1} \vdash Z : \mathsf{tape}$. For this I believe it may be necessary to change the type *epat* in a way that does not affect the remainder of the program but allows us to reach Z.

Exercise 5 In order to compose Turing machines so that the second one runs on the result of the first one, we need the final state of the first machine to traverse the tape to its left end and then transition to the initial state of the next machine.

Develop this as an alternative to Exercise 4.2.

References

[DP16] Henry DeYoung and Frank Pfenning. Substructural proofs as automata. In *14th Asian Symposium on Programming Languages and Systems*, Hanoi, Vietnam, November 2016. Springer LNCS. To appear.

[GH05] Simon J. Gay and Malcolm Hole. Subtyping for session types in the π-calculus. *Acta Informatica*, 42(2–3):191–225, 2005.