Lecture Notes on
Ordered Proofs as Concurrent Programs

15-816: Substructural Logics
Frank Pfenning

Lecture 5
September 13, 2016

In this lecture we will first define *subsingleton logic* which is the subset of ordered logic where each judgment has at most one antecedent. Then we provide an *operational interpretation* of subsingleton logic in which *cut reduction* drives computation. This is in contrast to what we have been doing so far, where *logical inference*, that is, *proof construction* models computation.

The correspondence from this lecture is summarized in the following table.

Logic	**Programming**
Propositions	Session types
Ordered proofs	Concurrent programs
Cut reduction	Communication

This is an instance of a very general connection between proofs and programs studied in type theory. We can vary the logic and the computational interpration. The big upside of this form of correspondence is that it helps us design programming languages in concert with the logic for reasoning about its programs.

This analysis was pioneered by Curry [Cur34] who related proofs in axiomatic form with combinatory logic. Late, Howard [How69] made the discovery that the Church's simply typed λ-calculus was in bijective correspondence with intuitionistic natural deduction. The particular instance of this correspondence for subsingleton logic is a recent discovery by DeYoung and yours truly [DP16].

1 Subsingleton Logic

We can examine the rules for each of the connectives to see which of them are still meaningful if we restrict ourselves to at most one antecedent. For example,

$$\frac{\Omega\ y \vdash x}{\Omega \vdash x \mathbin{/} y}\ /R$$

would have a premise with two antecedents if the conclusion has only one. Restricting the conclusion to just one[1] would not work because identity expansion would fail: in subsingleton logic, we wouldn't be able to prove $x \mathbin{/} y \vdash x \mathbin{/} y$.

What remains from the connectives we have introduced so far is only $x \mathbin{\&} y$ and $\mathbf{1}$. But we haven't had a notion of *disjunction* yet, which is written as $x \oplus y$. It is a disjunction, which means that $x \oplus y$ is true if either x or y is true. So we have two right rules:

$$\frac{\Omega \vdash x}{\Omega \vdash x \oplus y}\ \oplus R_1 \qquad \frac{\Omega \vdash y}{\Omega \vdash x \oplus y}\ \oplus R_2$$

Knowing that x or y is true, but not which one, means that the left rule proceeds by cases.

$$\frac{\Omega_L\ x\ \Omega_R \vdash z \quad \Omega_L\ y\ \Omega_R \vdash z}{\Omega_L\ (x \oplus y)\ \Omega_R \vdash z}\ \oplus L$$

It is straightforward to check the identity expansion and cut reduction, as well as extend the proof of cut elimination accordingly. Disjunction, just like the alternative conjunction, make sense in subsingleton logic.

We summarize the rules of subsingleton logic, with two small notational changes: we write ω for zero or one antecedent, and we use letters A, B, C to denote propositions (rather than x, y, z).

[1] in response to a question in lecture

$$\frac{}{A \vdash A}\ \mathsf{id}_A \qquad \frac{\omega \vdash A \quad A \vdash C}{\omega \vdash C}\ \mathsf{cut}_A$$

$$\frac{\omega \vdash A}{\omega \vdash A \oplus B}\ \oplus_1 \quad \frac{\omega \vdash B}{\omega \vdash A \oplus B}\ \oplus_2 \qquad \frac{A \vdash C \quad B \vdash C}{A \oplus B \vdash C}\ \oplus L$$

$$\frac{\omega \vdash A \quad \omega \vdash B}{\omega \vdash A \,\&\, B}\ \&R \qquad \frac{A \vdash C}{A \,\&\, B \vdash C}\ \&L_1 \quad \frac{B \vdash C}{A \,\&\, B \vdash C}\ \&L_2$$

$$\frac{}{\cdot \vdash \mathbf{1}}\ \mathbf{1}R \qquad \frac{\cdot \vdash C}{\mathbf{1} \vdash C}\ \mathbf{1}L$$

We can generalize this slightly if we are prepared to accept an empty right-hand side. It will not be particularly useful for our purposes today, but the rules would be

$$\frac{A \vdash \cdot}{A \vdash \bot}\ \bot R \qquad \frac{}{\bot \vdash \cdot}\ \bot L$$

Important[2] is that we also have to generalize all the other left rules to permit an empty succedent. So in the presence of $\bot$, sequents would have the form $\omega \vdash \gamma$, with both sides either empty or a singleton.

2 Proofs as Programs

We write $\omega \vdash P : A$ with two alternative interpretations:

1. P is a proof of A with antecedent ω.
2. P is a process providing A and using ω.

Two processes are composed by cut, so that if

$$\omega \vdash P : A \qquad A \vdash Q : C$$

then P and Q can run next to each other and exchange messages. Which messages can be exchanged is dictated by the type (= proposition) A.

As an example, consider

$$\omega \vdash P : A \oplus B \qquad A \oplus B \vdash Q : C$$

[2] which I neglected to mention in lecture

The proof P of $A \oplus B$ will contain critical information, namely if A is true or if B is true. Since the proof Q of C must account for both possibilities, we see that P will eventually *send* some information (inl if A is true, or inr if B is true) and Q will *receive* it. In a synchronous communication model, the message exchange can ony take place if both sides are ready, which correspond to a principal case of cut where $A \oplus B$ is the principal formula in both inferences. Using this intuition to fill in proof terms for one of the cases we arrive at:

$$\frac{\omega \vdash P : A}{\omega \vdash (\mathsf{R}.\mathsf{inl}\ ;\ P) : A \oplus B}\ \oplus R_1 \qquad \frac{A \vdash Q_1 : C \quad B \vdash Q_2 : C}{A \oplus B \vdash (\mathsf{caseL}\ (\mathsf{inl} \Rightarrow Q_1 \mid \mathsf{inr} \Rightarrow Q_2)) : C}\ \oplus L$$

This reduces to the new cut at type A

$$\omega \vdash P : A \qquad A \vdash Q_1 : C$$

So we think of $(\mathsf{R}.\mathsf{inl}\ ;\ P)$ as sending the label inl to the right and then continuing as P, while $\mathsf{caseL}\ (\mathsf{inl} \Rightarrow Q_1 \mid \mathsf{inr} \Rightarrow Q_2)$ receives either inl or inr from the left and continues as Q_1 or Q_2, respectively. The types A that type the interface between two processes are called *session types* (see [HHN+14] for a survey). A strong logical foundation for session types in *linear logic* was discovered by Caires and your lecturer [CP10] and later extended by others [Wad12, CPT13, Ton15].

If $\oplus R_2$ was used in the first proof, then the new cut would be at type B. In either case, the communication corresponds exactly to a principal case in cut reduction.

Looking at computation more globally, processes are configured into a linear chain

$$P_1\ P_2\ P_3\ \cdots\ P_n$$

which we write as ordered propositions

$$\mathsf{proc}(P_1) \quad \mathsf{proc}(P_2) \quad \mathsf{proc}(P_3) \quad \mathsf{proc}(P_n)$$

since we would like to specify the rules of computation for this programming language using ordered inference. Such a configuration is well-typed if we have

$$(\omega_0 \vdash P_1 : A_1) \quad (A_1 \vdash P_2 : A_2) \quad (A_2 \vdash P_3 : A_3) \quad \ldots \quad (A_n \vdash P_n : A_n)$$

where for any two adjacent processes, the type A_i provided by P_i has to be the same as the one used by P_{i+1}.

We now go through the rules and connective of ordered logic and develop the operational interpretation of proofs.

Cut as Composition. Cut is straightforward, since it just corresponds to parallel composition.

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

with the computation rule

$$\frac{\mathsf{proc}(P \mid Q)}{\mathsf{proc}(P) \quad \mathsf{proc}(Q)}$$

Clearly, this rule preserves the typing invariant for a configuration if $(P \mid Q)$ is typed by the cut rule, since the type A provided by P is exactly the same as the one as used by Q, and the left and right interfaces ω and C, respectively, are preserved.

Identity as Forwarding. Cut creates two processes from one, while identity removes one process from the configuration, acting like a "forwarding" between the processes to its sides.

$$\frac{}{A \vdash \leftrightarrow : A}\ \mathsf{id}_A$$

The computation rule simply removes the process from the configuration.

$$\frac{\mathsf{proc}(\leftrightarrow)}{\cdot}$$

Again, this preserves the typing invariant for configurations since the processes to the left and right of $\mathsf{proc}(\leftrightarrow)$ have the same type A on the right and left sides, respectively.

Now we come to the logical connectives. We already foreshadowed the case for disjunction, but we first generalize it to be more amenable for programming without changing its logical meaning.

Disjunction as Internal Choice. We generalize disjunction to be an n-ary connective by writen $\oplus\{l_i : A_i\}_{i \in I}$ for some finite index set I and *labels* l_i. Now the binary disjunction is defined as $A \oplus B = \oplus\{\mathsf{inl} : A, \mathsf{inr} : B\}$. Disjunction is also called *internal choice* since the proof itself determines which of the alternatives is chosen.

The right rule will send the apropriate label while the left rule will receive it and branch on it.

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

Again, the computation rule just mirrors the cut reduction and therefore is easily seen to preserve configuration typing.

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

The interface type between the two adjacent processes transitions from $\oplus\{l_i : A_i\}_{i\in I}$ to A_k for some $k \in I$.

Alternative Conjunction as External Choice. Again, we generalize from $A\&B$ to $\&\{l_i : A_i\}_{i\in I}$ and defined $A\&B = \&\{\mathsf{inl} : A, \mathsf{inr} : B\}$. $A\&B$ is sometimes called *external choice* since its proof must account for both possibilities and the clients selects between them. Otherwise, it is the straightforward dual of $\oplus$, sending to the left and receiving from the right.

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

Again, the computation rule just mirrors the cut reduction and therefore is easily seen to preserve configuration typing.

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

Unit as Termination. The unit $\mathbf{1}$ just corresponds to termination. Since communication is synchronous, the paired process to the right just waits for the termination to occur.

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

The computation rule lets the waiting process proceed while the closing one disappears.

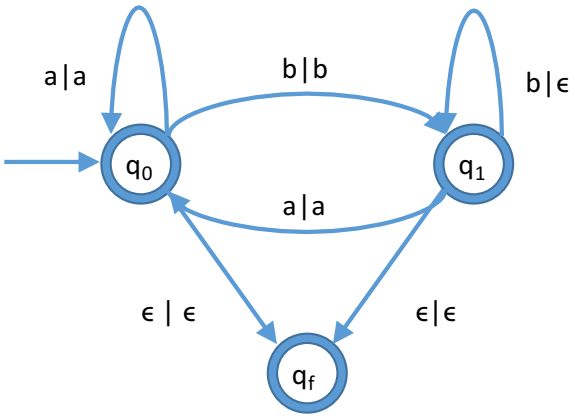
$$\frac{\mathsf{proc}(\mathsf{closeR}) \quad \mathsf{proc}(\mathsf{waitL}\ ;\ Q)}{\mathsf{proc}(Q)}$$

This preserves types since there is no left interface to the configuration before and after the step, and the right interface C is preserved. Again, this can simply be read off from the cut reduction.

This completes the introduction of the computational interpretation of subsingleton logic. The computational metatheory, namely the *progress* and *preservation* theorems will be discussed in the next lecture. We move on to write some programs!

3 Example: Subsequential Finite State Transducers

We have already introduced FSTs in Lecture 4, where we provided an implementation using the computation-as-ordered-inference paradigm. Here, we will use the computation-as-ordered-proof-reduction paradigm instead. We begin with the example of an FST that compresses runs of b into a single b.



We would like to represent the transducer as a process T that receives the input string, symbol by symbol, from the left and sends the output string, again symbol by symbol, to the right.

$$\mathit{string} \vdash T : \mathit{string}$$

The first problem is how to represent the type *string*. It is easy to represent symbols as *labels* and a choice between symbols a, b, and the endmarker \$ as an internal choice

$$\oplus\{a : A_a, b : A_b, \$: A_\$\} \vdash T : \mathit{string}$$

Clearly, T can proceed with a $\oplus L$ rule, which means it can branch on whether it receives an a, b, or a \$. After receiving an a, for example, what should the type on the left be? We can receive further symbols, so it should be again string. This leads us to

$$\mathit{string} = \oplus\{a : \mathit{string}, b : \mathit{string}, \$: A_\$\}$$

which is an example of a *recursive type* since *string* is defined in terms of itself. What should the remaining unspecified type $A_\$$ be? Once we receive the endmarker we can receive no further symbols (or anything else) from the left, we can only wait for the process to our left (that produced the string) to terminate. So $A_\$ = \mathbf{1}$ and we have

$$\mathit{string} = \oplus\{a : \mathit{string}, b : \mathit{string}, \$: \mathbf{1}\}$$

It is convenient in this setting to think of the *string* as being *equal* to the type on the right. If every type definition is *contractive* [GH05] in the sense that it starts with a type constructor ($\oplus, \&, \mathbf{1}$) then we do not need any explicit right or left rules since we can "silently" replace a type by its definition and apply the appropriate rule. This is the idea behind *equirecursive* treatment of recursive types. One should be worried that they could destroy all the good properties of the logic, but with some care this does not have to be the case. We will come back to this point in a future lecture.

Here is a simple program that produces the string $babb$:

$$\mathbf{1} \vdash \ulcorner babb \urcorner : \mathit{string}$$
$$\ulcorner babb \urcorner = \mathsf{R}.b\ ;\ \mathsf{R}.a\ ;\ \mathsf{R}.b\ ;\ \mathsf{R}.b\ ;\ \mathsf{R}.\$\ ;\ \leftrightarrow$$

To deal with recursive types, the program will have to be similarly recursive. At the level of proofs, this can be analyzed as circular proofs [FS13], fixed points [Bae12], or corecursive proofs [TCP14]. Again, we may come back to this point in a future lecture and just freely use recursion for now. Each state of the FST becames a process definition that captures how the FST behaves with the corresponding input. Output is handled simply by sending the appropriate label to the right, and the new state is handled by

invoking this state.

$$\begin{array}{lcl} Q_0 & = & \mathsf{caseL}\ (a \Rightarrow \mathsf{R}.a\ ;\ Q_0 \\ & & \quad \mid b \Rightarrow \mathsf{R}.b\ ;\ Q_1 \\ & & \quad \mid \$ \Rightarrow \mathsf{R}.\$\ ;\ Q_f) \\ \\ Q_1 & = & \mathsf{caseL}\ (a \Rightarrow \mathsf{R}.a\ ;\ Q_0 \\ & & \quad \mid b \Rightarrow Q_1 \\ & & \quad \mid \$ \Rightarrow \mathsf{R}.\$\ ;\ Q_f) \\ \\ Q_f & = & \leftrightarrow \end{array}$$

The type of the final state Q_f is a bit different, since we know input and output have completed by the time this state is reached. We have

$$\begin{array}{lcll} \mathit{string} & \vdash & Q_0 & : \ \mathit{string} \\ \mathit{string} & \vdash & Q_1 & : \ \mathit{string} \\ \mathbf{1} & \vdash & Q_f & : \ \mathbf{1} \end{array}$$

We also note an alternative definition for Q_f

$$Q_f \quad = \quad \mathsf{waitL}\ ;\ \mathsf{closeR}$$

These two definitions are equivalent in the sense that $(\mathsf{waitL}\ ;\ \mathsf{closeR})$ is the identity expansion of $\leftrightarrow\ :\ \mathbf{1}$. We will not go into detail, but this means that those two processes are *observationally equivalent* and can be used interchangeably [PCPT14].

At this point we could almost formulate a conjecture such as

$$\begin{array}{cc} \mathsf{proc}(\ulcorner w \urcorner) & \mathsf{proc}(Q_0) \\ \\ \vdots & \\ \\ \mathsf{proc}(\ulcorner v \urcorner) & \end{array}$$

where Q_0 is the process representing the initial state of the machine that transforms input w to output v. Before reading on, consider why this may not hold.

Yes: the problem is that when T attempts to send an output symbol to the right, there is no consumer so the process will actually block and the computation will come to a halt. There are at least two ways to solve this problem. One is to make communication *asynchronous* so that output (sending a label to the left of right) can always take place. This has two advantages: (1) it is more realistic from the implementation perspective, and (2) it increases the available parallelism. We will return to this option in a future lecture.

Another solution is to create a client that will accept the expected output string. This client is written in the form of a finite automaton, which we discuss in the next section.

4 Finite-State Automata

A (deterministic) finite-state automaton works almost exactly like a subsequential transducer, but it will output only either *acc* or *rej*, not a whole string. This is easy to model:

$$\mathit{answer} = \oplus\{\mathsf{acc} : \mathbf{1}, \mathsf{rej} : \mathbf{1}\}$$

It generalizes the grammar for strings by allowing two different endmarkers (instead of just \$), and has otherwise no symbols.

The we would write

$$\begin{array}{l}
\mathit{string} \vdash \mathit{reject} : \mathit{answer} \\
\mathit{reject} = \mathsf{caseL}\ (a \Rightarrow \mathit{reject} \mid b \Rightarrow \mathit{reject} \mid \$ \Rightarrow \mathsf{R.rej}\ ;\ \leftrightarrow) \\
\\
\mathit{string} \vdash \mathit{accept} : \mathit{answer} \\
\mathit{accept} = \mathsf{caseL}\ (a \Rightarrow \mathit{accept} \mid b \Rightarrow \mathit{accept} \mid \$ \Rightarrow \mathsf{R.acc}\ ;\ \leftrightarrow) \\
\\
\mathit{string} \vdash \llcorner \mathit{bab} \lrcorner : \mathit{answer} \\
\llcorner \mathit{bab} \lrcorner = \mathsf{caseL}\ (a \Rightarrow \mathit{reject} \\
\qquad\qquad\qquad \mid \$ \Rightarrow \mathsf{R.rej}\ ;\ \leftrightarrow \\
\qquad\qquad\qquad \mid b \Rightarrow \mathsf{caseL}\ (b \Rightarrow \mathit{reject} \\
\qquad\qquad\qquad\qquad\qquad \mid \$ \Rightarrow \mathsf{R.rej}\ ;\ \leftrightarrow \\
\qquad\qquad\qquad\qquad\qquad \mid a \Rightarrow \mathsf{caseL}\ (a \Rightarrow \mathit{reject} \\
\qquad\qquad\qquad\qquad\qquad\qquad\qquad \mid \$ \Rightarrow \mathsf{R.rej}\ ;\ \leftrightarrow \\
\qquad\qquad\qquad\qquad\qquad\qquad\qquad \mid b \Rightarrow \mathit{accept})))
\end{array}$$

We can then test our machine with

$$\begin{array}{c} \mathsf{proc}(\ulcorner bbab \urcorner) \quad \mathsf{proc}(Q_0) \quad \mathsf{proc}(\llcorner bab \lrcorner) \\ \vdots \\ \mathsf{proc}(\mathsf{R}.\mathit{acc} \; ; \leftrightarrow) \end{array}$$

Now we can state more generally that

$$\begin{array}{c} \mathsf{proc}(\ulcorner w \urcorner) \quad \mathsf{proc}(Q_0) \quad \mathsf{proc}(\llcorner v \lrcorner) \\ \vdots \\ \mathsf{proc}(\mathsf{R}.\mathit{acc} \; ; \leftrightarrow) \end{array}$$

if and only of the FST with initial state Q_0 transduces input w to output v. The proof of essentially this theorem is sketched in a recent paper [DP16].

The transitions of the transducer here are not exactly in one-to-one correspondence with the steps of proof construction, since the sequence of reading the input and writing the output are usually seen as a single step. Except for this potential minor difference regarding what counts as a step (depending on the precise formulation of the finite-state transducer), automata transitions are modeled precisely a logical inference steps.

In fact, the opposite is also true. If we have a *cut-free proof*

$$\mathit{string} \Vdash P : \mathit{string}$$

then P will behave like a finite state transducer. The proof is essentially by inversion: Since the proof is cut-free, P can proceed only by forwarding, receiving from the left, sending to the right, or recursing. From this we can easily construct an FST with the same behavior, again allowing for some minor discrepancies in how steps are counted.

Taken together, this means we have an isomormophism between proofs in subsingleton logic containing only $\oplus$ and $\mathbf{1}$ and inductively defined types and subsequential finite state transducers. A recent paper [DP16] slightly generalized FSTs so that they encompass finite-state automata as well by allowing multiple distinct endmarkers, as we have done for the representation of string acceptors.

As a last remark, we notice that composition of transducers is logically trivial, namely just cut. If we have

$$\mathit{string} \vdash T_1 : \mathit{string} \quad \text{and} \quad \mathit{string} \vdash T_2 : \mathit{string}$$

then

$$\mathit{string} \vdash (T_1 \mid T_2) : \mathit{string}$$

Here, the two transducers will run in parallel, similarly to our earlier modeling of transducers via ordered inference. T_1 will pass its output to T_2, which will in turn pass its output to a consumer on the right. We can also just perform cut elimination to obtain a cut-free T' equivalent to $(T_1 \mid T_2)$, but a word of caution: in the presence of corecursive (circular) proofs, the usual cut elimination algorithms has to work somewhat differently [FS13]. Nevertheless, it is an illustration how logical tools such as cut elimination can be used in programming languages, this time in program transformation.

Exercises

Exercise 1 Write a transducer over the alphabet a, b which produces ab for every occurrence of ab in the input and erases all other symbols.

1. Present it in the form of ordered inference rules.
2. Present it in the form of a well-typed program.

Exercise 2 Rewrite your parity-computing inference rules from Exercise L2.2 as a transducer, replacing eps with the endmarker $\$$.

1. Present the transducer in the form of ordered inference rules, for reference. You may freely change your solution of Exercise L2.2 in order to prepare it for part 2.
2. Rewrite it in the form of a well-typed ordered concurrent program.

Exercise 3 Rewrite the program below as a finite state transducer, expressed as a set of ordered inference rules. Describe the function on strings that Q_0 computes.

$$\begin{aligned} Q_0 = \mathsf{caseL}\ (&a \Rightarrow Q_1 \\ \mid\ &b \Rightarrow Q_2 \\ \mid\ &\$ \Rightarrow \mathsf{R}.\$\ ;\ \leftrightarrow) \end{aligned}$$

$$\begin{aligned} Q_1 = \mathsf{caseL}\ (&a \Rightarrow Q_1 \\ \mid\ &b \Rightarrow \mathsf{R}.b\ ;\ Q_2 \\ \mid\ &\$ \Rightarrow \mathsf{R}.\$\ ;\ \leftrightarrow) \end{aligned}$$

$$\begin{aligned} Q_2 = \mathsf{caseL}\ (&a \Rightarrow \mathsf{R}.a\ ;\ Q_1 \\ \mid\ &b \Rightarrow Q_2 \\ \mid\ &\$ \Rightarrow \mathsf{R}.\$\ ;\ \leftrightarrow) \end{aligned}$$

Exercise 4 Reconsider the transducers for compressing runs of b's, given here as a set of ordered inference rules. We present here the version without an explicit final state.

$$\frac{a\ q_0}{q_0\ a} \qquad \frac{b\ q_0}{q_1\ b} \qquad \frac{\$\ q_0}{\$}$$

$$\frac{a\ q_1}{q_0\ a} \qquad \frac{b\ q_1}{q_1} \qquad \frac{\$\ q_1}{\$}$$

In our encoding as a program Q_0 of type *string* $\vdash Q_0$: *string* we treated letters as messages and states as processes. No explicit representation of the final state is necessary with the rules above.

Define a dual encoding where symbols of the alphabet and endmarkers are represented processes and states as messages.

1. Define an appropriate type *state* so that *state* $\vdash P_a$: *state* where P_a is the process representation for the alphabet symbol a.

2. For each symbol a of the transducer alphabet, define the process P_a.

3. Give the type of the process $P_\$$ representing the endmarker \$. You may choose whether to represent a final state as an explicit message of some form or not.

4. Define the process $P_\$$ for the endmarker.

5. Define the initial configuration for the string $babb$ and initial state q_0. Then describe it in general for the machine under consideration here.

6. Define the final configuration for the given example string and initial state. Then describe it in general for the machine under consideration here.

7. Do you foresee any difficulties for encoding subsequential finite state transducers in general in this style? Note that FSTs read one symbol at a time but may output any number of symbols (including none) in one transition. Describe how this could be handled, or explain why a dual construction may only work for a restricted class of FSTs.

8. Consider how to compose transducers and compare to the composition in the original encoding given in lecture.

References

[Bae12] David Baelde. Least and greatest fixed points in linear logic. *ACM Transactions on Computational Logic*, 13(1), 2012.

[CP10] Luís Caires and Frank Pfenning. Session types as intuitionistic linear propositions. In *Proceedings of the 21st International Conference on Concurrency Theory (CONCUR 2010)*, pages 222–236, Paris, France, August 2010. Springer LNCS 6269.

[CPT13] Luís Caires, Frank Pfenning, and Bernardo Toninho. Linear logic propositions as session types. *Mathematical Structures in Computer Science*, 2013. To appear. Special Issue on Behavioural Types.

[Cur34] H. B. Curry. Functionality in combinatory logic. *Proceedings of the National Academy of Sciences, U.S.A.*, 20:584–590, 1934.

[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.

[FS13] Jérôme Fortier and Luigi Santocanale. Cuts for circular proofs: Semantics and cut elimination. In *22nd Conference on Computer Science Logic*, volume 23 of *LIPIcs*, pages 248–262, 2013.

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

[HHN+14] Kohei Honda, Raymond Hu, Rumyana Neykova, Tzu-Chun Chen, Romain Demangeon, Pierre-Malo Deniélou, and Nobuko Yoshida. Structuring communcation with session types. In *Concurrent Objects and Beyond (COB 2014)*, pages 105–127. Springer LNCS 8665, 2014.

[How69] W. A. Howard. The formulae-as-types notion of construction. Unpublished note. An annotated version appeared in: *To H.B. Curry: Essays on Combinatory Logic, Lambda Calculus and Formalism*, 479–490, Academic Press (1980), 1969.

[PCPT14] Jorge A. Pérez, Luís Caires, Frank Pfenning, and Bernardo Toninho. Linear logical relations and observational equivalences

for session-based concurrency. *Information and Computation*, 239:254–302, 2014.

[TCP14] Bernardo Toninho, Luís Caires, and Frank Pfenning. Corecursion and non-divergence in session-typed processes. In *Proceedings of the 9th International Symposium on Trustworthy Global Computing (TGC 2014)*, Rome, Italy, September 2014. To appear.

[Ton15] Bernardo Toninho. *A Logical Foundation for Session-based Concurrent Computation*. PhD thesis, Carnegie Mellon University and Universidade Nova de Lisboa, May 2015. Available as Technical Report CMU-CS-15-109.

[Wad12] Philip Wadler. Propositions as sessions. In *Proceedings of the 17th International Conference on Functional Programming*, ICFP 2012, pages 273–286, Copenhagen, Denmark, September 2012. ACM Press.