1 Implementing Call-by-Need

Our machine so far (K, S, and $S_η$) implemented a call-by-value evaluation strategy for functions applications. In call-by-value, whenever we evaluate $e_1 e_2$, we first evaluate $e_1$ to, say, $\lambda x. e'_1$, then $e_2$ to $v_2$ and then proceed by evaluating $[v_2/x]e'_1$. This might sometimes do unnecessary work. For example, $(\lambda x. \langle \rangle)(\text{fix } y. y)$ will not terminate, even though it $\beta$-reduces to $\langle \rangle$. 
Another strategy is call-by-name. In call-by-name, whenever we evaluate \( e_1 e_2 \) we first evaluate \( e_1 \) to, say, \( \lambda x. e'_1 \) and then \([e_2/x]e'_1\). This might also sometimes do unnecessary work. For example, \((\lambda x.(x,x)) e\) will evaluate \( e \) twice because it steps to \( (e,e) \).

A strategy designed to avoid both issues is call-by-need. In call-by-need, whenever we evaluate \( e_1 e_2 \) we start as before by evaluating \( e_1 \), yielding, say, \( \lambda x. e'_1 \). Now we do not yet evaluate \( e_2 \) but substitute a reference \( d \) to \( e_2 \) for \( x \) and evaluate \( e_1 \). When we encounter \( d \) during evaluation the first time we evaluate \( e_2 \) to a value, say, \( v_2 \). We then remember the value \( v_2 \) so that any further reference to \( d \) just returns \( v_2 \) without re-evaluating \( e_2 \).

Your task will be to specify a version of the S machine that implements call-by-need. For your reference, we previously had the following rules for call-by-value functions and for locations in the S machine:

\[
\begin{align}
\text{eval } (e_1 e_2) d & \mapsto \text{eval } e_1 d_1, \text{cont } d_1 (\_ e_2) d \quad (d_1 \text{ fresh}) \quad (1.1) \\
!\text{cell } d_1 c_1, \text{cont } d_1 (\_ e_2) d & \mapsto \text{eval } e_2 d_2, \text{cont } d_2 (d_1 \_ ) d \quad (d_2 \text{ fresh}) \quad (1.2) \\
!\text{cell } d_1 (\lambda x. e'_1), !\text{cell } d_2 c_2, \text{cont } d_2 (d_1 \_ ) d & \mapsto \text{eval } ([d_2/x]e'_1) d \quad (1.3) \\
!\text{cell } d_1 c, \text{eval } d_1 d & \mapsto !\text{cell } d c \quad (1.4) \\
\text{eval } (\lambda x. e) d & \mapsto !\text{cell } d (\lambda x. e) \quad (1.5)
\end{align}
\]

**Task 2** (20 points, WB). Implement call-by-need evaluation of function application on the S machine, replacing the rules above. If you keep any of the rules 1.1 to 1.5, state which ones.

Feel free to define a new kind of semantic object besides eval, cont, and !cell to track references to function arguments, but consider carefully if it should be ephemeral or persistent.

**Task 3** (5 points, WB). Verify by showing the key steps of computation in each case that

1. eval \((\lambda x. (\_)) (\text{fix } y. y)) d_0\) terminates, and

2. eval \((\lambda x. (x,x)) e) d_0\) evaluates \( e \) only once, where you should assume evaluation of \( e \) terminates.

## 2 Programming with Session Types

### 2.1 String Processing

Given some alphabet \( \Sigma \), we can encode strings as processes of type

\[
\text{str} = \oplus \{ \sigma : \text{str}, \mathbb{S} : 1 \}_{\sigma \in \Sigma}.
\]
In particular, we saw that binary numbers were strings over \( \Sigma = \{0, 1\} \). Assume throughout that \( \Sigma = \{a, b\} \). This means that

\[
\text{str} = \oplus \{a : \text{str}, b : \text{str}, \$ : 1\}.
\]

We can then encode literal strings in a manner analogous to how we encoded binary numbers. For example,

\[
\cdot \vdash \text{"aabbb"} :: (s : \text{str})
\]
\[
s \leftarrow \text{"aabbb"} = s.a; s.a; s.b; s.b; s.b; s.\$; \text{close } s
\]

In verbatim syntax we might write

\[
\cdot \vdash \text{example} :: (s : \text{str})
\]
\[
s \leftarrow \text{example} = s.a; s.a; s.b; s.b; s.b; s.\$; \text{close } s
\]

You may use verbatim syntax in this style in your answers.

### 2.1.1 String equality

To help you warm up, we will begin by implementing string equality. Let the type of Booleans be \( \text{bool} = \oplus \{\text{true} : 1, \text{false} : 1\} \). The following process may be of use to you in Task 5.

**Task 4** (5 points, WB). Implement a process \( s : \text{str} \vdash \text{flush} :: (o : 1) \) that discards all of the messages on \( s \).

**Note.** You may use the shorthand “\( _- \)” to match all other possible labels in a case statement. For example, you may write case \( c \{ l_1 \Rightarrow e_1 \mid l_2 \Rightarrow e_2 \mid _- \Rightarrow e_3 \} \) to transition to process \( e_3 \) whenever \( c \) provides a label other than \( l_1 \) and \( l_2 \).

**Task 5** (10 points, WB). Implement a process \( s_1 : \text{str}, s_2 : \text{str} \vdash \text{eq} :: (r : \text{bool}) \). It should send the message \( \text{true} \) over \( r \) and terminate if and only if the strings \( s_1 \) and \( s_2 \) produce the same sequence of characters (labels); otherwise, it should send \( \text{false} \) and terminate.

### 2.1.2 String reversal

Next, we will implement a string reversal process

\[
s : \text{str} \vdash \text{rev} :: (r : \text{str})
\]

that provides the reversal of \( s \) on \( r \).
Task 6 (15 points, WB). Implement the string reversal process

\[ s : \text{str} \vdash \text{rev} :: (r : \text{str}). \]

You may define auxiliary processes as you see fit. You do not need to be concerned about efficiency.

As an example, the output on the channel \( e \) of the following process should be true:

\[ \cdot \vdash s_1 \leftarrow \left[ ab \right]; s_2 \leftarrow \left[ ba \right]; r \leftarrow \text{rev} \leftarrow s_2; e \leftarrow \text{eq} \leftarrow s_1, r :: (e : \text{bool}). \]

### 2.2 A Polish Notation Calculator

In grade school, you likely learned the infix notation for arithmetic operators. An alternative notation is the prefix notation, sometimes called the Polish notation. In this notation, the arithmetic operators come before their operands. For example, we parse \( + \times 1 + 2 \ 3 \ 4 \) as \( +(\times(1, +(2, 3)), 4) \), corresponding to the infix expression \( 1 \times (2 + 3) + 4 \). In order to avoid the complexity of arbitrary numbers, we write a calculator for arithmetic modulo 2. In other words, there are only two values 0 and 1 with their expected modular interpretation. For example, \( 1 \times 1 = 1 \) and \( 1 + 1 = 0 \).

We will implement a simple calculator for modular arithmetic operations in Polish notation involving \(+, \times, \) and integers 0 and 1. We let the type of expressions \( \exp \) be

\[ \exp = \oplus \{ 0 : \exp, 1 : \exp, + : \exp, \times : \exp, E : 1, $ : 1 \}, \]

where \( E \) stands for an error. We encode expressions in the obvious way, e.g.,

\[ \cdot \vdash \Gamma + \times 1 \ 1 \ 0 \vdash :: (e : \exp) \]
\[ e \leftarrow \Gamma + \times 1 \ 1 \ 0 \leftarrow = e.+; e.\times; e.1; e.1; e.0; e.$; \text{close } e \]

In verbatim syntax, we might write

\[ . \mid - \text{example} :: (e : \exp) \]
\[ e \leftarrow \text{example} = e.+ ; e.\times ; e.1 ; e.1 ; e.0 ; e.$ ; \text{close } e \]

You may use verbatim syntax in this style in your answers.

Our calculator process \( \text{eval} \) will have the type \( e' : \exp \vdash \text{eval} :: (e : \exp) \). We need to include an error case to cope with malformed arithmetical expressions. Consider for example the expression \( \times+ \), which induces the process

\[ \cdot \vdash e.\times; e.+; e.$; \text{close } e \]. Because we cannot evaluate it, we should abort with an error, which we do by sending the label \( E \) and closing the channel.
Task 7 (5 points, WB). Implement a process
\[
e : \text{exp} \vdash \text{flush} :: (o : 1)
\]
that discards all of the messages on \(e\). Using this, implement a process
\[
e' : \text{exp} \vdash \text{abort} :: (e : \text{exp})
\]
that aborts the computation by sending the label \(E\) along \(e\) and terminating.

Task 8 (10 points, WB). Implement the processes
\[
\begin{align*}
e' : \text{exp} & \vdash \text{plus} :: (e : \text{exp}) \\
e' : \text{exp} & \vdash \text{times} :: (e : \text{exp})
\end{align*}
\]
The process \(\text{plus}\) should behave as follows: if the next two labels on \(e'\) are integers, \(\text{plus}\) should output their sum on \(e\) and then forward. Otherwise, \(\text{plus}\) should abort by sending the error label \(E\) and terminating. The process \(\text{times}\) should behave similarly, except by implementing multiplication.

Task 9 (10 points, WB). Implement the process \(e' : \text{exp} \vdash \text{eval} :: (e : \text{exp})\). Whenever it encounters a malformed expression, it should abort with an error. The process
\[
\cdot \vdash e' \leftarrow \Gamma + 1 \times 1 \cdot ; e \leftarrow \text{eval} \leftarrow e' :: (e : \text{exp})
\]
should be indistinguishable from the process \(\Gamma 0\) to any process using \(e\).

A Verbatim Syntax for Session Types

We have provided a table of suggested verbatim syntax:

- **Forwarding**
  \[
  x \leftarrow y
  \]
  \[
  x \leftarrow y
  \]
- **Close channel**
  \[
  \text{close } x
  \]
  \[
  \text{close } x
  \]
- **Wait for** \(x\) and continue as \(P\)
  \[
  \text{wait } x ; P
  \]
  \[
  \text{wait } x ; P
  \]
- **Case on** \(x\)
  \[
  \text{case } x \{ l \Rightarrow P | \_ \Rightarrow Q \}
  \]
  \[
  \text{case } x \{ l \Rightarrow P | \_ \Rightarrow Q \}
  \]
- **Send** \(l\) over \(x\), continue as \(P\)
  \[
  x.l ; P
  \]
  \[
  x.l ; P
  \]
- **Typing judgment**
  \[
  a : A, b : B \vdash P :: (c : C)
  \]
  \[
  a : A, b : B \vdash P :: (c : C)
  \]
- **Named process definition**
  \[
  c \leftarrow \text{foo} \leftarrow a, b = P
  \]
  \[
  c \leftarrow \text{foo} \leftarrow a, b = P
  \]
- **Spawn named process**
  \[
  c \leftarrow \text{bar} \leftarrow a, b ; Q
  \]
  \[
  c \leftarrow \text{bar} \leftarrow a, b ; Q
  \]