Types and Programming Languages (15-814), Fall 2018
Assignment 9: Call-by-Need and Session Types

Contact: 15-814 Course Staff

Due **Thursday**, December 6, 2018, 11:59pm

This assignment is due by 11:59pm on the above date and it must be submitted electronically as a PDF file on Canvas. Please use the attached template to typeset your assignment and make sure to include your full name and Andrew ID. As before, problems marked "WB" are subject to the whiteboard policy; all other problems must be done individually.

Task 0 (0 points)**.** How long did you spend on this assignment? Please list the questions that you discussed with classmates using the whiteboard policy.

Task 1 (5 points)**.** Please take a few minutes to complete the Faculty Course Evaluation and TA evaluations! You can complete the FCEs at `https://www.cmu.edu/hub/fce/`. We are particularly interested in suggestions for improving the course for future iterations. You can complete the two parts of the TA evaluation here:

- `https://www.ugrad.cs.cmu.edu/ta/F18/feedback/`
- `https://www.ugrad.cs.cmu.edu/ta/F18/freeform/`

Once you have completed all three evaluations, you can claim your five points by affirming that you have done so.

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, $(\lambda x.\, \langle\rangle)\ (\mathbf{fix}\, y.\, y)$ will not terminate, even though it β-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, $(\lambda x.\langle x, x\rangle)\ e$ will evaluate e twice because it steps to $\langle e, e\rangle$.

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:

$$\mathsf{eval}\ (e_1\ e_2)\ d \quad \mapsto \quad \mathsf{eval}\ e_1\ d_1, \mathsf{cont}\ d_1\ (\_\ e_2)\ d \qquad (d_1 \text{ fresh}) \tag{1.1}$$

$$\mathsf{!cell}\ d_1\ c_1, \mathsf{cont}\ d_1\ (\_\ e_2)\ d \quad \mapsto \quad \mathsf{eval}\ e_2\ d_2, \mathsf{cont}\ d_2\ (d_1\ \_)\ d \qquad (d_2 \text{ fresh}) \tag{1.2}$$

$$\mathsf{!cell}\ d_1\ (\lambda x.\, e_1'), \mathsf{!cell}\ d_2\ c_2, \mathsf{cont}\ d_2\ (d_1\ \_)\ d \quad \mapsto \quad \mathsf{eval}\ ([d_2/x]e_1')\ d \tag{1.3}$$

$$\mathsf{!cell}\ d_1\ c, \mathsf{eval}\ d_1\ d \quad \mapsto \quad \mathsf{!cell}\ d\ c \tag{1.4}$$

$$\mathsf{eval}\ (\lambda x.\, e)\ d \quad \mapsto \quad \mathsf{!cell}\ d\ (\lambda x.\, e) \tag{1.5}$$

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. $\mathsf{eval}\ ((\lambda x.\, \langle\,\rangle)\ (\mathbf{fix}\, y.\, y))\ d_0$ terminates, and
2. $\mathsf{eval}\ \ ((\lambda x.\, \langle x, x\rangle)\, e)\ \ d_0$ evaluates e only once, where you should assume evaluation of e terminates.

2 Programming with Session Types

2.1 String Processing

Given some alphabet Σ, we can encode strings as processes of type

$$\mathbf{str} = \oplus\{\sigma : \mathbf{str}, \$: \mathbf{1}\}_{\sigma\in\Sigma}.$$

In particular, we saw that binary numbers were strings over $\Sigma = \{0, 1\}$. Assume throughout that $\Sigma = \{a, b\}$. This means that

$$\mathbf{str} = \oplus\{a : \mathbf{str}, b : \mathbf{str}, \$: \mathbf{1}\}.$$

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

$\cdot \Vdash \ulcorner aabbb \urcorner :: (s : \mathbf{str})$
$s \leftarrow \ulcorner aabbb \urcorner = s.a; s.a; s.b; s.b; s.b; s.\$; \mathsf{close}\ s$

In verbatim syntax we might write

```
. |- example :: (s : str)
s <- example = s.a ; s.a ; s.b ; s.b ; s.b ; s.$ ; 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 $\mathbf{bool} = \oplus\{\mathbf{true} : \mathbf{1}, \mathbf{false} : \mathbf{1}\}$. The following process may be of use to you in Task 5.

Task 4 (5 points, WB)**.** Implement a process $s : \mathbf{str} \Vdash \mathsf{flush} :: (o : \mathbf{1})$ 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 $\mathsf{case}\ c\ \{\ l_1 \Rightarrow e_1 \mid l_2 \Rightarrow e_2 \mid \_ \Rightarrow e_3\}$ 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 $s_1 : \mathbf{str}, s_2 : \mathbf{str} \Vdash \mathsf{eq} :: (r : \mathbf{bool})$. It should send the message $\mathbf{true}$ 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 $\mathbf{false}$ and terminate.

2.1.2 String reversal

Next, we will implement a string reversal process

$$s : \mathbf{str} \Vdash \mathsf{rev} :: (r : \mathbf{str})$$

that provides the reversal of s on r.

Task 6 (15 points, WB)**.** Implement the string reversal process

$$s : \mathbf{str} \Vdash \mathsf{rev} :: (r : \mathbf{str}).$$

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**:

$$\cdot \Vdash s_1 \leftarrow \ulcorner ab \urcorner; s_2 \leftarrow \ulcorner ba \urcorner; r \leftarrow \mathsf{rev} \leftarrow s_2; e \leftarrow \mathsf{eq} \leftarrow s_1, r :: (e : \mathbf{bool}).$$

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

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

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

$\cdot \Vdash \ulcorner + \times 1\ 1\ 0 \urcorner :: (e : \mathbf{exp})$
$e \leftarrow \ulcorner + \times 1\ 1\ 0 \urcorner = e.+; e.\times; e.1; e.1; e.0; e.\$; \mathsf{close}\ e$

In verbatim syntax, we might write

```
. |- example :: (e : exp)
e <- example = e.+ ; e.* ; e.1 ; e.1 ; e.0 ; e.$ ; close e
```

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

Our calculator process eval will have the type $e' : \mathbf{exp} \Vdash \mathsf{eval} :: (e : \mathbf{exp})$. We need to include an error case to cope with malformed arithmetical expressions. Consider for example the expression "$\times+$", which induces the process $\cdot \Vdash e.\times; e.+; e.\$; \mathsf{close}\ e$. 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

$$e : \mathbf{exp} \Vdash \mathsf{flush} :: (o : \mathbf{1})$$

that discards all of the messages on e. Using this, implement a process

$$e' : \mathbf{exp} \Vdash \mathsf{abort} :: (e : \mathbf{exp})$$

that aborts the computation by sending the label E along e and terminating.

Task 8 (10 points, WB)**.** Implement the processes

$e' : \mathbf{exp} \Vdash \mathsf{plus} :: (e : \mathbf{exp})$
$e' : \mathbf{exp} \Vdash \mathsf{times} :: (e : \mathbf{exp})$

The process plus should behave as follows: if the next two labels on e' are integers, plus should output their sum on e and then forward. Otherwise, plus should abort by sending the error label E and terminating. The process times should behave similarly, except by implementing multiplication.

Task 9 (10 points, WB)**.** Implement the process $e' : \mathbf{exp} \Vdash \mathsf{eval} :: (e : \mathbf{exp})$. Whenever it encounters a malformed expression, it should abort with an error. The process

$$\cdot \Vdash e' \leftarrow \ulcorner +1 \times 1\ 1 \urcorner ; e \leftarrow \mathsf{eval} \leftarrow e' :: (e : \mathbf{exp})$$

should be indistinguishable from the process $\ulcorner 0 \urcorner$ 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 <- y`
Close channel x	
close x	`close x`
Wait for x and continue as P	
wait x ; P	`wait x ; P`
Case on x	
case x { $l \Rightarrow P \mid \_ \Rightarrow Q$}	`case x { l => P | _ => 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 |- P :: (c : C)`
Named process definition	
$c \leftarrow \mathsf{foo} \leftarrow a, b = P$	`c <- foo <- a, b = P`
Spawn named process	
$c \leftarrow \mathsf{bar} \leftarrow a, b$; Q	`c <- bar <- a, b ; Q`