Harvard School of Engineering and Applied Sciences — CS 152: Programming Languages

Concurrency, Type and effect system, Message passing Section and Practice Problems

Apr 14–15, 2016

---

1 Concurrency

(a) Consider the following program.

$$(3+7) \;||\; ((\lambda x.\, x+1)\; 2+5)$$

Show an execution sequence for this program (i.e., give a sequence of expressions such that $e_0 \longrightarrow e_1 \longrightarrow ... \longrightarrow e_n$ where $e_0 = (3+7) \;||\; ((\lambda x.\, x+1)\; 2+5)$ and e_n is a value.

Answer: *This evaluation evaluates the left expression all the way to a value, and then the right.*

$$\begin{aligned}(3+7) \;||\; ((\lambda x.\, x+1)\; 2+5) &\longrightarrow 10 \;||\; ((\lambda x.\, x+1)\; 2+5)\\ &\longrightarrow 10 \;||\; ((\lambda x.\, x+1)\; 7)\\ &\longrightarrow 10 \;||\; 7+1\\ &\longrightarrow 10 \;||\; 8\\ &\longrightarrow (10, 8)\end{aligned}$$

Now give a different execution sequence for this program.

Answer: *This evaluation evaluates the right expression all the way to a value, and then the left.*

$$\begin{aligned}(3+7) \;||\; ((\lambda x.\, x+1)\; 2+5) &\longrightarrow 3+7 \;||\; ((\lambda x.\, x+1)\; 7)\\ &\longrightarrow 3+7 \;||\; 7+1\\ &\longrightarrow 3+7 \;||\; 8\\ &\longrightarrow 10 \;||\; 8\\ &\longrightarrow (10, 8)\end{aligned}$$

How many different execution sequences of this program are there?

Answer: *The left expression just needs a single step to evaluate to a value, the right expression takes 3 steps. There are 4 different possible interleavings (i.e., for $i \in 0..3$ there is an interleaving where the left expression takes its single step after the right expression has evaluated i steps).*

(b) Consider the following program.

```
let foo = ref 2 in
let y = (foo :=!foo+!foo || foo := 1) in
!foo
```

What are the possible final values of the program?

Answer: *The left expression (*foo :=!foo+!foo*) reads* foo *twice, and then updates it, and the right expression (*foo := 1*) just updates* foo.

There are four possible interleavings, which we show below, as well as the final value.

!foo *(=2)* !foo *(=2)* foo := ... *(=4)*	
	foo := 1

Final value: 1

!foo *(=2)* !foo *(=2)*	
	foo := 1
foo := ... *(=4)*	

Final value: 4

!foo *(=2)*	
	foo := 1
!foo *(=1)*	
foo := ... *(=3)*	

Final value: 3

	foo := 1
!foo *(=1)*	
!foo *(=1)*	
foo := ... *(=2)*	

Final value: 2

2 Type and effect system

Recall the type and effect system to ensure determinacy, covered in Lecture 17.

(a) Consider the program (from class) of a bank balance, where the bank balance is in the region A.

$$\mathsf{let\ bal} = \mathsf{ref}_A\ 0\ \mathsf{in}\ (\mathsf{let}\ y = (\mathsf{bal} := !\mathsf{bal} + 25\ ||\ \mathsf{bal} := !\mathsf{bal} + 50)\ \mathsf{in}\ !\mathsf{bal})$$

Try to produce a typing derivation for this program (using the type-and-effect typing rules from lecture). Where do the typing rules fail? Why?

Answer: *The typing derivation fails when we try to type the concurrent expression* bal :=!bal + 25 || bal := !bal + 50*. Specifically, the read and write effect for both the left and write expression is* $\{A\}$*, the set containing the region* A*. Thus, the write set of* bal :=!bal + 25 *intersects with the write set of* bal :=!bal + 50*, and so we can't satisfy the premise.*

(b) Write a program that allocates two locations (in different regions) and reads and writes from both of them. Moreover, make sure that your program is well-typed according to the type-and-effect system.

Is your program deterministic?

Answer: *Here's one. It is well-typed and deterministic! (Indeed, all well-typed programs are deterministic.)*

let a = $\mathsf{ref}_A\ 0$ *in*
let b = $\mathsf{ref}_B\ 0$ *in*
let y = (a :=!a + 25 || b :=!b + 50) *in*
!a+!b

3 Synchronous sends and receives

(a) To make sure you understand the operational semantics of sends and receives, show the execution of the following program. The initial configuration consists of a single thread, and your answer should show the sequence of configurations that program execution will step through.

$$\mathsf{let}\ c = \mathsf{newchan}_{\mathbf{int}}\ \mathsf{in}$$
$$\mathsf{spawn}\ (1 + \mathsf{recv\ from}\ c);$$
$$\mathsf{send}\ 6\ \mathsf{to}\ c$$

Answer: *Note that in the answer below we write* $\boldsymbol{c}$ *to represent the channel value assigned to the program variable* c*. Note that initially we start off with a configuration with a single thread.*

$$
\begin{aligned}
&\langle \textit{let } c = \textit{newchan}_{\textit{int}} \textit{ in spawn } (1 + \textit{recv from } c); \textit{send } 6 \textit{ to } c\rangle \\
\longrightarrow &\langle \textit{let } c = \boldsymbol{c} \textit{ in spawn } (1 + \textit{recv from } c); \textit{send } 6 \textit{ to } c\rangle \\
\longrightarrow &\langle \textit{spawn } (1 + \textit{recv from } \boldsymbol{c}); \textit{send } 6 \textit{ to } \boldsymbol{c}\rangle \\
\longrightarrow &\langle (); \textit{send } 6 \textit{ to } \boldsymbol{c},\ \ 1 + \textit{recv from } \boldsymbol{c}\rangle \\
\longrightarrow &\langle \textit{send } 6 \textit{ to } \boldsymbol{c},\ \ 1 + \textit{recv from } \boldsymbol{c}\rangle \\
\longrightarrow &\langle (),\ \ 1 + 6\rangle \\
\longrightarrow &\langle (),\ \ 7\rangle
\end{aligned}
$$

(b) Consider the following program. What are the possible final values of the main (i.e., initial) thread?

$$
\begin{aligned}
&\text{let } c = \text{newchan}_{\textbf{int}} \text{ in} \\
&\text{let } d = \text{newchan}_{\textbf{int}} \text{ in} \\
&\text{let } g = \text{newchan}_{\textbf{int chan}} \text{ in} \\
&\text{spawn send } c \text{ to } g; \\
&\text{spawn send } d \text{ to } g; \\
&\text{spawn send } 3 \text{ to } c; \\
&\text{spawn send } 4 \text{ to } c; \\
&\text{spawn send } 5 \text{ to } d; \\
&\text{spawn send } 6 \text{ to } d; \\
&(\text{recv from } (\text{recv from } g)) + \text{recv from } c
\end{aligned}
$$

Answer: *The way to think about this program is there is a single receive on channel* g*, but two values sent to channel* g*: the channel* c *and the channel* d*. Let* x *be the channel received over* g*. We then receive a value from* x *and a value from* c*.*

There are two values sent over c *(3 and 4) and two values sent over* d *(5 and 6).*

If x*, the value received over* g*, is the channel* c*, then we receive both values sent over* c *and add them. Although the order in which they are received is non-deterministic, adding them together will always produce 7.*

If x*, the value received over* g*, is the channel* d*, then we will receive one of the two value sent over* d *and receive one of the two values sent over* c*, and add them together. Thus we may get any of* $3 + 5 = 8$*,* $3 + 6 = 9$*,* $4 + 5 = 9$*, or* $4 + 6 = 10$*.*

Thus the possible final values of the program are: 7, 8, 9, and 10.

(c) Write a program that creates a channel c, and then sends the Fibonacci sequence over the channel (i.e., it sends 0, 1, 1, 2, 3, 5, 8, ...). Your program will spawn a thread to compute and send the Fibonacci sequence, and the other thread will consume the values produced. (You may assume you have a recursive operator $\mu x.\ e$.)

Answer: *We will need a recursive function to produce the Fibonacci sequence and send it over channel* c*. We will define that function, and then spawn it. We will have another recursive function that simply receives values over* c*.*

let $c =$ *newchan*$_{\textbf{int}}$ *in*
let $fib = \mu f.\ \lambda n : \textbf{int}, m : \textbf{int}.$ *send* n *to* $c;\ f\ m\ (n+m)$ *in*
spawn $fib\ 0\ 1;$
$\mu x.$ *recv from* $c; x$

4 First-class synchronization

(a) Write a program that creates a channel c, and then sends the Fibonacci sequence over the channel (i.e., it sends 0, 1, 1, 2, 3, 5, 8, ...), but does not block waiting for the sequence to be consumed. Spawn one thread to compute and send the Fibonacci sequence, but make the other thread consume (i.e., receive) just a single value from the channel. What are the possible values received by the other thread? (You may assume you have a recursive operator $\mu x.\ e$.)

Answer: *We adapt our answer from above. Note that the recursive function fib does a non-blocking send instead of a blocking send.*

The initial thread receives a single value over channel c. Due to the non-determinism in the evaluation of the concurrent program, the receive over c may evaluate to any *value sent over c, i.e., to any number in the Fibonacci sequence.*

let $c =$ *newchan*$_{\textbf{int}}$ *in*
let $fib = \mu f.\ \lambda n : \textbf{int}, m : \textbf{int}.$ *sendEvt* n *to* $c;\ f\ m\ (n+m)$ *in*
spawn $fib\ 0\ 1;$
recv from c

(b) Consider the following program, which uses events. Using the translation given in class, translate it to the language with blocking sends and receives without events.

let $c =$ newchan$_{\textbf{int}}$ in
spawn (sendEvt 8 to c; send 9 to c);
let $r =$ recvEvt from c in
(recv from c) + sync r

Answer: *Here is the direct translation. Note that only the* *sendEvt* 8 *to* c, *recvEvt from* c, *and* *sync* r *are*

different.

```
let c = newchan_int in
spawn (
    let v = 8 in
    let c' = c in
    let d = newchan_unit chan in
    spawn (let x = send v to c' in let y = recv from d in send x to y);
    d;
  send 9 to c
);
let r = (
    let c' = c in
    let d = newchan_int chan in
    spawn (let x = recv from c' in let y = recv from d in send x to y);
    d
) in
(recv from c)+
  (
    let d = r in
    let c' = newchan_int in
    send c' to d;
    recv from c'
  )
```

Let's clean it up a bit by simplifying some of the translated expressions, and also by noting that the send event is discarded.

```
let c = newchan_int in
spawn (
    spawn (send 8 to c);
  send 9 to c
);
let r = (
    let d = newchan_int chan in
    spawn (let x = recv from c in let y = recv from d in send x to y);
    d
) in
(recv from c)+
  (
    let c' = newchan_int in
    send c' to r;
    recv from c'
  )
```

The key thing to note about the translation is that the translation of the sync operation creates a new channel c' which is sends over r, and then waits to receive a value over c'. Meanwhile, the translation of the recvEvt from *c ensured that r was bound to a channel such that a thread waited to receive a value x from channel c, then waited to receive the channel c' sent over r, and then sent x (=8) over c'.*