Lecture Notes on Session Types

15-814: Types and Programming Languages
Frank Pfenning

Lecture 22
November 27, 2018

1 Introduction

Some of the material in this lecture, specifically, the discussion of external choice and the implementation of a counter, are already provided in the notes for Lecture 21 on *Message-Passing Concurrency*. First, we have identity (forwarding) and cut (spawn), which work parametrically over all types:

$c \leftarrow d$	implement c by d and terminate
$x \leftarrow P\ ;\ Q$	spawn $[c/x]P$, providing a fresh c, with client $[c/x]Q$

Here is a summary table of the message-passing constructs in our process language so far, organized by type.

Type	**Provider**	**Client**	**Continuation Type**
$c : \oplus\{\ell : A_\ell\}_{\ell \in L}$	$(c.k\ ;\ P)$	$\mathbf{case}\ c\ \{\ell \Rightarrow Q_\ell\}_{\ell \in L}$	$c : A_k$
$c : \&\{\ell : A_\ell\}_{\ell \in L}$	$\mathbf{case}\ c\ \{\ell \Rightarrow P_\ell\}_{\ell \in L}$	$(c.k\ ;\ Q)$	$c : A_k$
$c : 1$	$\mathbf{close}\ c$	$\mathbf{wait}\ c\ ;\ Q$	*(none)*

Note that there is a complete symmetry here between internal and external choice, only the role of provider and client are swapped. Compare that to the functional world, where disjoint sums $\tau + \sigma$ and lazy pairs $\tau\ \&\ \sigma$ exhibit a number of differences. Partly, the additional simplicity gained is due to the sequent calculus as compared to natural deduction. In particular, in the sequent calculus all connectives have uniform right- and left-rules, while in natural deduction the elimination rules for positive connectives ($1, \tau \otimes \sigma, \tau + \sigma$) all use **case** constructs and are therefore much different from those for negative connectives ($\tau \rightarrow \sigma, \tau\ \&\ \sigma$). The other reason for the simplicity here is linearity.

2 Passing Channels

Even though the examples in this course do not use them, we can also ask what the message-passing counterparts to $\tau \to \sigma$ and $\tau \otimes \sigma$ are. The first one is easy to guess: $A \multimap B$ corresponds to receiving a channel d of type A and continuing with type B. Conversely, $A \otimes B$ corresponds to *sending* a channel d of type A. From this we can straightforwardly reconstruct the typing rules, but we refer the interested reader, for example, to Balzer et al. [BP17].

Type	Provider	Client	Continuation Type
$c : \oplus\{\ell : A_\ell\}_{\ell \in L}$	$(c.k\ ;\ P)$	$\mathbf{case}\ c\ \{\ell \Rightarrow Q_\ell\}_{\ell \in L}$	$c : A_k$
$c : \&\{\ell : A_\ell\}_{\ell \in L}$	$\mathbf{case}\ c\ \{\ell \Rightarrow P_\ell\}_{\ell \in L}$	$(c.k\ ;\ Q)$	$c : A_k$
$c : 1$	$\mathbf{close}\ c$	$\mathbf{wait}\ c\ ;\ Q$	*(none)*
$c : A \multimap B$	$x \leftarrow \mathbf{recv}\ c\ ;\ P$	$\mathbf{send}\ c\ d$	B
$c : A \otimes B$	$\mathbf{send}\ c\ d$	$x \leftarrow \mathbf{recv}\ c\ ;\ Q$	B

Again, we see a symmetry between $A \multimap B$ and $A \otimes B$, while in a functional language, functions $\tau \to \sigma$ and eager pairs $\tau \otimes \sigma$ are quite different.

3 Session Types in Concurrent C0

In the remainder of this lecture we demonstrate the robustness and practicality of these somewhat abstract ideas about message-passing concurrent computation by presenting a concrete instantiation of the ideas in Concurrent C0 [WPP16, BP17].

Instead of the notations of linear logic, Concurrent C0 uses more traditional notation of *session types* [Hon93]. Concurrent C0 is based on C0 [1], a type-safe and memory-safe subset of C0 augmented with contracts. C0 is used in the freshman-level introductory computer science class at CMU. Many of the syntax decision are motivated by consistency with C and should be viewed in this light.

First, session types are enclosed in angle brackets `< ... >` to make lexing and parsing them conservative over C0 (and C). Any *sending* interaction from the provider perspective is written as `!_` while a receiving interaction is written as `?_`.

A *choice*, whether it is *internal* ($\oplus$) or *external* ($\&$), must be declared with a name. This declaration is modeled after a `struct` declaration in C. So

$$\{\ell_1 : A_1, \ldots, \ell_n : A_n\}$$

[1] http://c0.typesafety.net

is written as

```
choice cname {
  < A1 > l1;
  ...
  < An > ln;
};
```

where `cname` is the name of this particular choice.

For example, to represent binary numbers

$$bin = \oplus\{\mathsf{b0} : bin, \mathsf{b1} : bin, \epsilon : 1\}$$

we would start by declaring the choice

```
choice bin {
  < ... > b0;
  < ... > b1;
  < ... > eps;
};
```

How do we fill in the continuation session types inside the angle brackets? The first two are straightforward: They are of type *bin*, which is the *internal choice* over `bin`.

```
choice bin {
  <!choice bin> b0;
  <!choice bin> b1;
  < ... >       eps;
};
```

In the case of epsilon (label `eps`) we close the channel without a continuation, which is written as the empty session type.

```
choice bin {
  <!choice bin> b0;
  <!choice bin> b1;
  < >           eps;
};
```

For good measure, we *define* the type `bin` to stand for the internal choice `!choice bin`:

```
typedef <!choice bin> bin;
```

4 Channels and Process Definitions

In Concurrent C0, names of channels are prefixed by '`$`' so they can be easily distinguished from ordinary variables. A process definition then has the general form

```
<A> $c pname (t1 x1, ..., tn xn) {
  ... process expression ...
}
```

where `A` is the session type of the channel c provided by the process name `pname`. Each of the arguments `xi` can be either a channel or an ordinary variable.

We start by defining the process that send the representation of the number 0.

```
bin $z zero () {
  $z.eps ; close($z);
}
```

We see that sending a label `l` along channel `$c` is written as `$c.l` and closing a channel `$c` is simply `close($c)`.

Next, we implement the successor process that receives a stream of binary digits representing n along a channel x it uses, and sends a stream of digits representing n along a channel y is provides. Recall from the last lecture:

$$x : \mathit{bin} \Vdash \mathit{succ} :: (y : \mathit{bin})$$

$$\begin{aligned} \mathit{succ} = \mathbf{case}\ x\ (\ & \mathsf{b0} \Rightarrow y.\mathsf{b1}\ ;\ y \leftarrow x \\ \mid\ & \mathsf{b1} \Rightarrow y.\mathsf{b0}\ ;\ \mathit{succ} \\ \mid\ & \epsilon \Rightarrow y.\mathsf{b1}\ ;\ y.\epsilon\ ; \\ & \qquad \mathbf{wait}\ x\ ;\ \mathbf{close}\ y\ ) \end{aligned}$$

Following the style of C, the **case** construct is written as a `switch` statement whose subject is a channel `$c`. We select the appropriate branch according to the label received along `$c`.

```
bin $y succ(bin $x) {
  switch ($x) {
    case b0: {
      $y.b1; $y = $x;
    }
```

```
    case b1: {
      $y.b0; $y = succ($x);
    }
    case eps: {
      $y.b1; wait($x); $y.eps; close($y);
    }
  }
}
```

Forwarding $y \leftarrow x$ is written as an assignment `$y = $x`.

5 Functions Using Channels

In Concurrent C0, functions that return values may also use channels. For example, here is a function to print a number in binary form, with the *most significant bit* first (the way we are used to seeing numbers).

```
void print_bin_rec(bin $x) {
  switch ($x) {
    case b0: {
      print_bin_rec($x);
      print("0");
      return;
    }
    case b1: {
      print_bin_rec($x);
      print("1");
      return;
    }
    case eps: {
      wait($x);
      return;
    }
  }
}

void print_bin(bin $x) {
  print_bin_rec($x);
  print(".\n");
  return;
```

```
}
```

Now we can implement a simple main function for testing purposes.

6 Functions Using Channels

In Concurrent C0, functions that return values may also use channels. For example, here is a function to print a number in binary form, with the *most significant bit* first (the way we are used to representing numbers).

```
void print_bin_rec(bin $x) {
  switch ($x) {
    case b0: {
      print_bin_rec($x);
      print("0");
      return;
    }
    case b1: {
      print_bin_rec($x);
      print("1");
      return;
    }
    case eps: {
      wait($x);
      return;
    }
  }
}

void print_bin(bin $x) {
  print_bin_rec($x);
  print(".\n");
  return;
}
```

The following simple main function should print `100.` and finish. Note that the call to `print_bin` is *sequential*, while the calls to `zero` and `succ` spawn new processes. We also see how each channel is created and then used, so that at the end of the functions all channels have been used.

```
int main() {
```

```
  bin $z = zero();
  bin $one = succ($z);
  bin $two = succ($one);
  bin $three = succ($two);
  bin $four = succ($three);
  print_bin($four);
  return 0;
}
```

7 Implementing a Counter Process

Recall that a counter has the interface

$ctr = \&\{\mathsf{inc} : ctr, \mathsf{val} : bin\}$

that is, it receives either a inc or val label. There are no new ideas required to represent this type. We just use external choice `?_` instead of internal choice `!_` where appropriate.

```
choice ctr {
  <?choice ctr> inc;
  <!choice bin> val;
};

typedef <?choice ctr> ctr;
```

Recall from the last lecture

$$x : bin \Vdash counter :: (c : ctr)$$

$$\begin{array}{rl} counter = \mathbf{case}\ c\ (\ \mathsf{inc} \Rightarrow & y \leftarrow succ \leftarrow x\ ; \\ & \quad c \leftarrow counter \leftarrow y \\ \mid\ \mathsf{val} \Rightarrow & c \leftarrow x\ ) \end{array}$$

This is easy to transliterate:

```
ctr $c counter(bin $x) {
  switch ($c) {
    case inc: {
      bin $y = succ($x);
      $c = counter($y);
    }
```

```
    case val: {
      $c = $x;
    }
  }
}
```

We now write a more complicated `main` function, using two loops. For each loop, we have to make sure that the type of any channel is *loop invariant*, since we do not know how many times we go around the loop.

```
int main() {
  bin $z = zero();
  bin $one = succ($z);
  bin $two = succ($one);
  bin $three = succ($two);
  bin $four = succ($three);
  for (int i = 0; i < 1000; i++) {
    $four = succ($four);
  }
  ctr $c = counter($four);      /* counter, initialized with 1004 */
  for (int i = 0; i < 2000; i++) {
    $c.inc;
  }
  $c.val;
  print_bin($c);                /* 3004 */
  return 0;
}
```

8 Lists and Stacks

As a final example, we program lists of binary numbers and stacks, where a stack is like an object holding a list. This example demostrates the passing of channels.

$list = \oplus\{\mathsf{nil} : 1, \mathsf{cons} : bin \otimes list\}$

$stack = \&\ \{\mathsf{push} : bin \multimap stack, \mathsf{pop} : response\}$
$response = \oplus\{\mathsf{none} : 1, \mathsf{some} : bin \otimes stack\}$

We say "list", but it is not represented in memory but a protocol by which individual elements are sent across a channel. Note that type *stack* and *response* are mutually recursive. In Concurrent C0:

```
choice list {
  < >                   nil;
  <!bin ; !choice list> cons;
};

typedef <!choice list> list;

choice stack {
  <?bin ; ?choice stack> push;
  <!choice response>     pop;
};
choice response {
  < >                    none;
  <!bin ; ?choice stack> some;
};
typedef <?choice stack> stack;
```

Then we have processes `Nil` and `Cons`, somewhat similar to `zero` and `succ`.

```
list $n Nil() {
  $n.nil; close($n);
}

list $k Cons(bin $x, list $l) {
  $k.cons; send($k,$x); $k = $l;
}
```

Finally, the process implementing a stack. It is the sole client of the list `$l`, which acts as a "local" storage.

```
stack $s stack_proc(list $l) {
  switch ($s) {
    case push: {
      bin $x = recv($s);
      list $k = Cons($x,$l);
      $s = stack_proc($k);
    }
    case pop: {
      switch ($l) {
        case nil: {
          wait($l);
          $s.none; close($s);
        }
        case cons: {
```

```
          bin $x = recv($l);
          $s.some; send($s, $x);
          $s = stack_proc($l);
        }
      }
    }
  }
}
```

In the updated `main` function we just push one element onto the stack, pop it off, and print it. We should now actually call pop again and wait for the stack process to terminate, but we ran out of time during lecture so we just raise an error. With this particular code we cannot reach the end of the `main` function, so we have to comment out the return since Concurrent C0 detects and flags unreachable code.

```
int main() {
  bin $z = zero();
  bin $one = succ($z);
  bin $two = succ($one);
  bin $three = succ($two);
  bin $four = succ($three);
  for (int i = 0; i < 1000; i++) {
    $four = succ($four);
  }
  ctr $c = counter($four);
  for (int i = 0; i < 2000; i++) {
    $c.inc;
  }
  $c.val;
  // print_bin($c);                /* 3004 */
  list $n = Nil();
  stack $s = stack_proc($n);
  $s.push; send($s, $c);
  $s.pop;
  switch ($s) {
    case none: {
      error("impossible");
    }
    case some: {
      bin $d = recv($s);
      print_bin($d);
      error("out of time");
    }
```

```
  }
  // return 0;
}
```

References

[BP17] Stephanie Balzer and Frank Pfenning. Manifest sharing with session types. In *International Conference on Functional Programming (ICFP)*, pages 37:1–37:29. ACM, September 2017.

[Hon93] Kohei Honda. Types for dyadic interaction. In *4th International Conference on Concurrency Theory*, CONCUR'93, pages 509–523. Springer LNCS 715, 1993.

[WPP16] Max Willsey, Rokhini Prabhu, and Frank Pfenning. Design and implementation of Concurrent C0. In *Fourth International Workshop on Linearity*, pages 73–82. EPTCS 238, June 2016.