Lecture Notes on Concurrent Programming

15-814: Types and Programming Languages
Frank Pfenning

Lecture 21
Thursday, November 12, 2020

1 Introduction

In this lecture we explore concurrent programming in our language of processes through two different examples: a *pipeline* (as started in the last lecture) and *fork/join* parallelism with *map/reduce*. Before we get to these, a small example to get used to the representation of functions in the concurrent language.

2 Simple Functions

We want to define a process

$$\mathit{curry} : ((\tau \times \sigma) \to \rho) \to (\tau \to (\sigma \to \rho))$$

Its implementation will immediately write a continuation to memory.

$$[\![\mathit{curry}]\!]\, d = \mathsf{case}\ d^W\ (\langle f, g\rangle \Rightarrow \boxed{})$$

So the real essence of this function is in the continuation

$K_{\mathit{curry}} = (\langle f, g\rangle \Rightarrow P)$

where P reads from $f : (\tau \times \sigma) \to \rho$ and writes to $g : \tau \to (\sigma \to \rho)$. The result is immediately a λ-expression, which means that as a process we write another continuation to memory.

$K_{\mathit{curry}} = (\langle f, g\rangle \Rightarrow \mathsf{case}\ g^W\ (\langle x, h\rangle \Rightarrow \boxed{}))$

Here $x : \tau$, the argument to g. Again, we write a function, this time one that takes $y : \sigma$ and a destination $r : \rho$ for the final result.

$$K_{curry} = (\langle f, g\rangle \Rightarrow \mathsf{case}\ g^W\ (\langle x, h\rangle \Rightarrow \mathsf{case}\ h^W\ (\langle y, r\rangle \Rightarrow \boxed{})))$$

At this point we have x and y in hand, so we can pair them up and pass the pair to f. But, wait! We cannot actually construct a pair and pass it. Instead, we need to allocate a cell to hold the pair $\langle x, y\rangle$ and pass its address p to g. In addition, we also have to pass an address as the destination of f, but that is just r. That is:

$$\begin{aligned} K_{curry} = \langle f, g\rangle \Rightarrow \mathsf{case}\ g^W\ (\langle x, h\rangle \Rightarrow \mathsf{case}\ h^W\ (\langle y, r\rangle \Rightarrow \\ p \leftarrow p^W.\langle x, y\rangle\ ; \\ f^R.\langle p, r\rangle)) \end{aligned}$$

Similarly, we start for a function in the other direction:

$$K_{uncurry} : (\tau \to (\sigma \to \rho)) \to ((\tau \times \sigma) \to \rho)$$

$$K_{uncurry} = \langle g, f\rangle \Rightarrow \mathsf{case}\ f^w\ (\langle p, r\rangle \Rightarrow \boxed{})$$

Now we have $p : \tau \times \sigma$ and the destination $r : \rho$. We read out the component from the cell at address p.

$$K_{uncurry} = \langle g, f\rangle \Rightarrow \mathsf{case}\ f^w\ (\langle p, r\rangle \Rightarrow \mathsf{case}\ p^R\ (\langle x, y\rangle \Rightarrow \boxed{}))$$

Now we need to pass $x : \tau$ to g, but we also need a destination. The one we have ($r : \rho$) does not work, so we need to allocate a new one, call it h.

$$\begin{aligned} K_{uncurry} = \langle g, f\rangle \Rightarrow \mathsf{case}\ f^w\ (\langle p, r\rangle \Rightarrow \mathsf{case}\ p^R\ (\langle x, y\rangle \Rightarrow \\ h \leftarrow g^R.\langle x, h\rangle\ ; \\ \boxed{})) \end{aligned}$$

At this point we can just read the function at $h : \sigma \to \rho$ and pass it $y : \sigma$ and the destination $r : \rho$.

$$\begin{aligned} K_{uncurry} = \langle g, f\rangle \Rightarrow \mathsf{case}\ f^w\ (\langle p, r\rangle \Rightarrow \mathsf{case}\ p^R\ (\langle x, y\rangle \Rightarrow \\ h \leftarrow g^R.\langle x, h\rangle\ ; \\ h^R.\langle y, r\rangle)) \end{aligned}$$

Neither of these processes has much intrinsic concurrency, but the arguments, for example, to f and g are addresses, and the value to be stored at these addresses may not yet have been written. We can see that neither x nor y are read by these functions, just passed through. As mentioned previously, this is the characteristic of *futures*.

3 A Bit-Flipping Pipeline

More interesting from the concurrency point of view is the bit-flipping pipeline. Recall from the last lecture the type of sequences of bits

$$\mathit{bits} = \rho\alpha.\, (\mathbf{b0} : \alpha) + (\mathbf{b1} : \alpha) + (\mathbf{e} : 1)$$

We start by writing an ordinary function $\mathit{flip} : \mathit{bits} \to \mathit{bits}$ that flips the input bits from 0 to 1 and vice versa.

$$\begin{array}{l}\mathit{flip} : \mathit{bits} \to \mathit{bits} \\ \mathit{flip} = \mathsf{fix}\, \mathit{flip}.\, \lambda x.\, \mathsf{case}\ x\ (\ \mathsf{fold}\ (\mathbf{b0} \cdot x') \Rightarrow \mathsf{fold}\ (\mathbf{b1} \cdot (\mathit{flip}\ x')) \\ \qquad\qquad\qquad\qquad\qquad\quad \mid \mathsf{fold}\ (\mathbf{b1} \cdot x') \Rightarrow \mathsf{fold}\ (\mathbf{b0} \cdot (\mathit{flip}\ x')) \\ \qquad\qquad\qquad\qquad\qquad\quad \mid \mathsf{fold}\ (\mathbf{e} \cdot u) \Rightarrow \mathsf{fold}\ (\mathbf{b1} \cdot \mathsf{fold}\ (\mathbf{e} \cdot u))\ )\end{array}$$

In the remainder of this section we make a small syntactic simplification which makes the code much shorter without loss of information content: we skip reading and writing the *fold* messages. We can think of the types as "silently unfolded", or postulate an elaboration pass over the program that inserts suitable fold constructors and fold patterns. The *flip* function then would look like

$$\begin{array}{l}\mathit{flip} : \mathit{bits} \to \mathit{bits} \\ \mathit{flip} = \mathsf{fix}\, \mathit{flip}.\, \lambda x.\, \mathsf{case}\ x\ (\ \mathbf{b0} \cdot x' \Rightarrow \mathbf{b1} \cdot (\mathit{flip}\ x') \\ \qquad\qquad\qquad\qquad\qquad\quad \mid \mathbf{b1} \cdot x' \Rightarrow \mathbf{b0} \cdot (\mathit{flip}\ x') \\ \qquad\qquad\qquad\qquad\qquad\quad \mid \mathbf{e} \cdot u \Rightarrow \mathbf{b1} \cdot (\mathbf{e} \cdot u)\ )\end{array}$$

Instead of translating this function we write it directly to a process. For this purpose we have to decide how to handle recursion. There seem to be two solutions:

1. We add a process $\mathsf{fix}\ \ f.P$ which transitions to $[\mathsf{fix}\ \ f.P/f]P$. This is entirely straightforward but requires process substitution in the dynamics.

2. We allow recursively defined processes

   $\mathsf{!cell}\ \mathit{flip}\ K_{\mathit{flip}}$

   where K_{flip} refers back to its own cell with address *flip* to encode a recursive call.

We choose the latter option, for variety, even though it would require more complicated typing rules for configurations.

It remains to define K_{flip}.

$$\begin{aligned} K_{\mathit{flip}} = \langle x, y\rangle \Rightarrow \mathsf{case}\ x\ (\ & \mathbf{b0}\cdot x' \Rightarrow \boxed{} \\ \mid\ & \mathbf{b1}\cdot x' \Rightarrow \boxed{} \\ \mid\ & \mathbf{e}\cdot u \Rightarrow \boxed{}\ ) \end{aligned}$$

In the first branch, we have to allocate a fresh cell y' for the output and make a recursive call to fill it. We can also write $\mathbf{b1}\cdot y'$ to y.

$$\begin{aligned} K_{\mathit{flip}} = \langle x, y\rangle \Rightarrow \mathsf{case}\ x\ (\ & \mathbf{b0}\cdot x' \Rightarrow y' \leftarrow \mathit{flip}^R.\langle x', y'\rangle\ ; \\ & \qquad\qquad\ \ y^W.(\mathbf{b1}\cdot y') \\ \mid\ & \mathbf{b1}\cdot x' \Rightarrow \boxed{} \\ \mid\ & \mathbf{e}\cdot u \Rightarrow \boxed{}\ ) \end{aligned}$$

The branch for $\mathbf{b1}$ is symmetric to the first one.

$$\begin{aligned} K_{\mathit{flip}} = \langle x, y\rangle \Rightarrow \mathsf{case}\ x\ (\ & \mathbf{b0}\cdot x' \Rightarrow y' \leftarrow \mathit{flip}^R.\langle x', y'\rangle\ ; \\ & \qquad\qquad\ \ y^W.(\mathbf{b1}\cdot y') \\ \mid\ & \mathbf{b1}\cdot x' \Rightarrow y' \leftarrow \mathit{flip}^R.\langle x', y'\rangle\ ; \\ & \qquad\qquad\ \ y^W.(\mathbf{b0}\cdot y') \\ \mid\ & \mathbf{e}\cdot u \Rightarrow \boxed{}\ ) \end{aligned}$$

In the last case, we allocate a new cell to hold $\mathbf{e}\cdot u$ and share $u : 1$ between the input and the output. Alternatively, we could avoid the inner allocation and just share x itself, or we could copy u also.

$$\begin{aligned} K_{\mathit{flip}} = \langle x, y\rangle \Rightarrow \mathsf{case}\ x\ (\ & \mathbf{b0}\cdot x' \Rightarrow y' \leftarrow \mathit{flip}^R.\langle x', y'\rangle\ ; \\ & \qquad\qquad\ \ y^W.(\mathbf{b1}\cdot y') \\ \mid\ & \mathbf{b1}\cdot x' \Rightarrow y' \leftarrow \mathit{flip}^R.\langle x', y'\rangle\ ; \\ & \qquad\qquad\ \ y^W.(\mathbf{b0}\cdot y') \\ \mid\ & \mathbf{e}\cdot u \Rightarrow\ \ y' \leftarrow y'^W.(\mathbf{e}\cdot u) \\ & \qquad\qquad\ \ y^W.(\mathbf{b1}\cdot y')\ ) \end{aligned}$$

As shown in the last lecture, we can compose two *flip* processes into a pipeline as follows:

$$\begin{aligned} K_{\mathit{flip2}} = \langle x, z\rangle \Rightarrow\ & \\ & y \leftarrow \mathit{flip}^R.\langle x, y\rangle \\ & \mathit{flip}^R.\langle y, z\rangle \end{aligned}$$

You may look back at the diagrams to visualize how the two processes work together, effectively communicating via the shared location y, which then becomes y', y'', etc. as the computation progresses and recursive calls are mad in both of them.

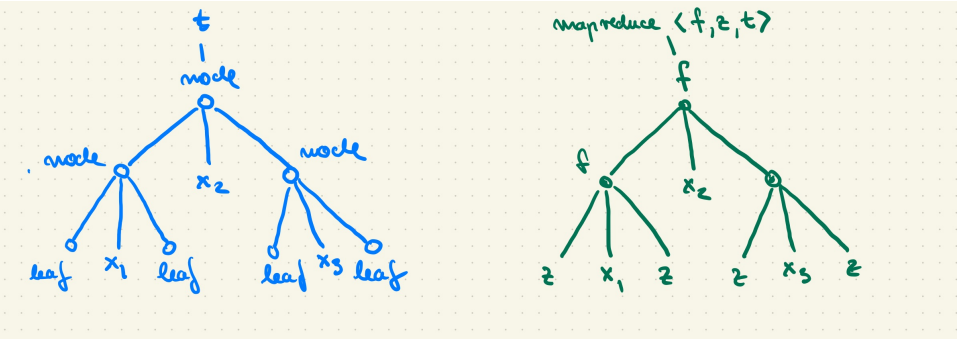
Under a sequential interpretation, where $x \leftarrow P \,;\, Q$ waits until P has written to destination x before Q starts executing, all recursive calls in *flip* would have to be finished before the first bit of output is written. When we compose two, the inner one has to finish entirely, writing out the whole sequence of bits before the outer one can start. This is the behavior of the functional $\lambda x.\, \mathit{flip}\,(\mathit{flip}\,x)$ where the intermediate destination y and the final destination z remain unnamed.

4 Map/Reduce

As a second example with significant concurrency we consider the popular *mapreduce*. We use a function f to *map* over a tree, reducing it to a value. In many applications the tree may not be explicit, but emerge dynamically from the way the data are distributed. As a consequence we require our function f to be associative and have a unit z, which may stand in for the absence of data. See Exercise 3 for a version where trees are represented differently. We define tree as a family of types, indexed by the type of element, even though we have not formally introduced this into our language.

$$\mathit{tree}\ \alpha = \rho t.\, (\mathbf{node} : t \times \alpha \times t) + (\mathbf{leaf} : 1)$$

We can picture the action of *mapreduce* as an *iteration* over this kind of tree. We supply a function f to "replace" every node and constant z to stand in for every leaf, as pictured in green in the image below.



We can read off the type

$\mathit{mapreduce} : [\forall\alpha.\,\forall\beta.]\,(\beta \times \alpha \times \beta \to \beta) \times \beta \times \mathit{tree}\ \alpha \to \beta$

As before, we imagine a cell !cell $\mathit{mapreduce}\ K_{\mathit{mapreduce}}$ and define $K_{\mathit{mapreduce}}$. We have put the type quantifiers on α and β in brackets because we haven't explicitly considered how to handle these in our concurrent language. Instead, we think of *mapreduce* as a family of functions indexed by α and β.

$K_{\mathit{mapreduce}} = \langle\langle f, z, t\rangle, y\rangle \Rightarrow \boxed{}$

Here we have $f : \beta \times \alpha \times \beta \to \beta$, $z : \beta$, and $t : \mathit{tree}\ \alpha$, with the destination $y : \beta$. We have taken a small shortcut here by using pattern matching: in fully official syntax, the right-hand side would start as

$K_{\mathit{mapreduce}} = \langle p, y\rangle \Rightarrow \mathsf{case}\ p\ (\langle f, q\rangle \Rightarrow \mathsf{case}\ q\ (\langle z, t\rangle \Rightarrow \boxed{}))$

but this is more verbose and more difficult to read. Back to the previous version. We start with a case analysis over t: is it a leaf or a node? If it is a leaf, we just copy z to the destination y.

$$
\begin{aligned}
K_{\mathit{mapreduce}} = \langle\langle f, z, t\rangle, y\rangle \Rightarrow\ & \\
&\mathsf{case}\ t^R\ (\ \mathbf{leaf} \cdot \langle\,\rangle \Rightarrow y^W \leftarrow z^R \\
&\quad \mid \mathbf{node} \cdot \langle l, x, r\rangle \Rightarrow \boxed{}\ )
\end{aligned}
$$

Here, l is the address of the left subtree, x is the element at the node, and r is the address of the right subtree. Now we need to make two recursive calls, on the left and right subtrees. In order to make these calls we need to allocate two new cells y_1 and y_2 to receive the values of these calls and pass them as destinations.

$$
\begin{aligned}
K_{\mathit{mapreduce}} = \langle\langle f, z, t\rangle, y\rangle \Rightarrow\ & \\
&\mathsf{case}\ t^R\ (\ \mathbf{leaf} \cdot \langle\,\rangle \Rightarrow y^W \leftarrow z^R \\
&\quad \mid \mathbf{node} \cdot \langle l, x, r\rangle \Rightarrow \\
&\qquad y_1 \leftarrow \mathit{mapreduce}^R.\langle\langle f, z, l\rangle, y_1\rangle\ ; \\
&\qquad y_2 \leftarrow \mathit{mapreduce}^R.\langle\langle f, z, r\rangle, y_2\rangle\ ; \\
&\qquad \boxed{}\ )
\end{aligned}
$$

Note that these two recursive calls proceed concurrently. Finally, we have to invoke the function f on the results from these recursive calls and x, and pass the result to y.

$$
\begin{array}{l}
K_{\mathit{mapreduce}} = \langle\langle f, z, t\rangle, y\rangle \Rightarrow \\
\qquad \mathsf{case}\ t^R\ (\ \mathbf{leaf} \cdot \langle\,\rangle \Rightarrow y^W \leftarrow z^R \\
\qquad\qquad \mid \mathbf{node} \cdot \langle l, x, r\rangle \Rightarrow \\
\qquad\qquad\quad y_1 \leftarrow \mathit{mapreduce}^R.\langle\langle f, z, l\rangle, y_1\rangle\ ; \\
\qquad\qquad\quad y_2 \leftarrow \mathit{mapreduce}^R.\langle\langle f, z, r\rangle, y_2\rangle\ ; \\
\qquad\qquad\quad f^R.\langle\langle y_1, x, y_2\rangle, y\rangle\ )
\end{array}
$$

Again, we have used a short-hand here. In official syntax we have to allocate pairs to hold the first argument to f, so the last line would expand to:

$$
\begin{array}{l}
p_1 \leftarrow p_1^W.\langle x, y_2\rangle\ ; \\
p_2 \leftarrow p_2^W.\langle y_1, p_1\rangle\ ; \\
f^R.\langle p_2, y\rangle
\end{array}
$$

In any case, we can see that no synchronization on y_1 or y_2 occurs until the function f actually needs their values.

5 Recovering Sequentiality

Originally, we thought of our concurrent process language as the result of translating our expression language LAMBDA. However, the result of the translation behaves significantly differently from the source due the pervasive concurrency.

We could just say that we schedule the different processes for taking step in a way that exactly mimics left-to-right sequential execution. Or we can manipulate the translation to enforce sequentiality. Since only cut (an allocate followed by a spawn) creates a new thread of control, this is our main lever to work with. For example, we could have a *sequential cut* $x \Leftarrow P; Q$ which runs P to completion before starting Q. Its semantics might be:

$$\mathsf{proc}\ d\ (x \Leftarrow P; Q) \mapsto \mathsf{proc}\ c\ ([c/x]P), \mathsf{susp}\ c\ d\ ([c/x]Q)$$

with a new semantics object wait with the rule

$$!\mathsf{cell}\ c\ W, \mathsf{susp}\ c\ d\ Q \mapsto \mathsf{proc}\ d\ Q$$

where the new semantic object $\mathsf{susp}\ c\ d\ Q$ represents a suspected process, waiting for the cell c to be written to. Since writing to c is the last action of a process with destination c, this will prevent Q from computing until P has finished. Moreover, Q will never have to synchronize on c because it is guaranteed to have already been written to.

It is then easy to prove, by induction on transition sequences, that there is at most one (unsuspended) process in a configuration if we start with just one process. Also, the concurrent semantics can *simulate* the sequential one by always making particular choices, but not the other way around.

In a language with both sequential and concurrent cut we can work mostly sequentially and occasionally spawn a process to run concurrently. This is the idea behind *futures* where the expression future e immediately returns a destination d that the evaluation of e eventually writes to. Attempts to read the future will block until the value has been written.

Exercises

Exercise 1 Consider the translation

$$[\![\mathsf{fix}\, f.\, e]\!]\, d = \mathsf{case}\ d^W\ (\langle x, y\rangle \Rightarrow [d/f][\![e]\!]\, y)$$

in which d is written to but also (potentially) read from in the translation of $[d/f][\![e]\!]\, y$. Execution of this process may therefore create circular references in the configuration.

(i) Give an example where the translation behaves *incorrectly* with respect to the dynamics of the expression $\mathsf{fix}\, f.\, e$ in LAMBDA.

(ii) Give an example where circular references arise but behave *correctly* with respect to the dynamics in the source.

(iii) From your examples, conjecture a restriction of the general translation so the result behaves correctly.

(iv) Devise new typing rules for processes and configurations such that (a) the translation above is well-typed, as a process, and (b) the typing of configurations is preserved by transitions, and (c) the progress theorem continues to be true. You do not need to prove these properties, but it may be helpful to sketch the proof to yourself to make sure your rules are correct.

Exercise 2 When translating functional fixed point expression to recursively defined processes, we need to account for the fact that processes may be invoked in multiple places with different destinations. We there introduce the notation $x.\, P$ for a process with variable destination x and $(x.\, P)(d)$ for its instantiation to a particular destination. We then translate:

$[\![\mathsf{fix}\, f.\, e]\!]\, d = (x.\, \mathsf{rec}\, f.\, [\![e]\!]\, x)(d)$
where $[\![f]\!]\, c = f(c)$ for every occurrence of f in e.

We also extend the dynamics with the rule

$$\mathsf{proc}\ d\ ((x.\, \mathsf{rec}\, f.\, P)(d)) \mapsto \mathsf{proc}\ d\ ([(x.\, \mathsf{rec}\, f.\, P)/f][d/x]P)$$

(i) Give typing rules for the new forms of processes.

(ii) Provide an implementation of the *flip* process using this representation of recursion.

(iii) Illustrate the key transition steps in the computation of *flip*, showing the plausibility of this translation.

Exercise 3 Consider the type of tree where the information is kept only in the leaves:

$$\mathit{shrub}\ \alpha = \rho t.\, (\mathbf{branch} : t \times t) + (\mathbf{bud} : \alpha)$$

(i) Write a version of *mapreduce* that operates on shrubs and exhibits analogous concurrent behavior. You may use similar shortcuts to the ones we used in our implementation.

(ii) Write processes *forth* and *back* to translate between trees and shrubs while preserving the elements. Do they form an isomorphism? If not, do you see a simple modification to restore an isomorphism?

Exercise 4 The sequential execution in Section 5 is *eager* in the sense that in $x \Leftarrow P\ ;\ Q$, P completes by writing to x before Q starts.

A lazy version, $x \leftharpoonup P\ ;\ Q$ would immediately start Q and suspend P until Q (or some process spawned by it) would try to read from x. We would still like it to be sequential in the sense that at most one process can take a step at any time.

Devise a semantics for $x \leftharpoonup P\ ;\ Q$ that exhibits the desired lazy behavior while remaining sequential. You may introduce new semantic objects or apply some transformation to P and/or Q, but you should strive for the simplest, most elegant solution to keep the dynamics simple.