# Lecture Notes on Message-Passing Concurrency

15-814: Types and Programming Languages
Frank Pfenning

Lecture 20
Tuesday, November 16, 2021

## 1 Introduction

In this lecture we first write another programming using shared memory concurrency, namely the commonly used *mapreduce*, and identify the parallelism inherent in it. We observe that this program also has a natural message-passing interpretation, so we consider an alternative dynamics for our concurrent language based on sending and receiving messages along channels instead of writing to and reading from memory cells.

## 2 Simple Functions[1]

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.

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

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 $\lambda$-expression, which means that as a process we write another continuation to memory.

[1] This section not covered in lecture

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

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.

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

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:

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

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

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

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

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 $h$.

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

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

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

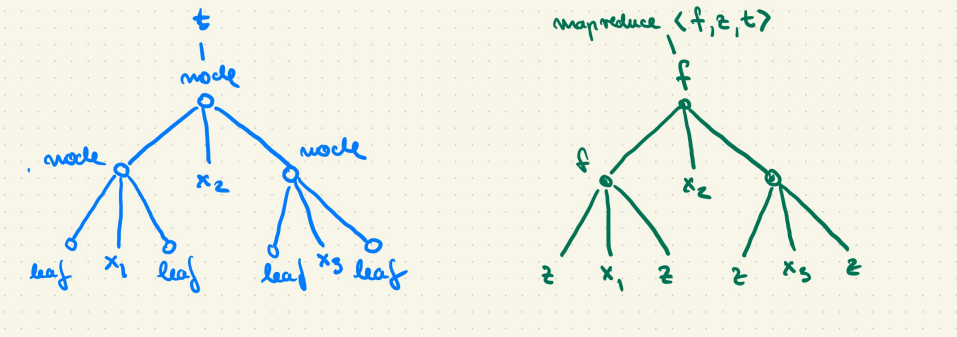
Neither of these processes has much intrinsic concurrency, but the arguments, for example, to $f$ and $g$ are addresses, and the value to be stored at these addresses may not yet have been written. We can see that neither $x$ nor $y$ are read by these functions, just passed through. As mentioned previously,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

## 3 Map/Reduce

As an 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 using shrubs[2]. 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.

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

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 *mapreduce* $K_{\mathit{mapreduce}}$ and define $K_{\mathit{mapreduce}}$. We have put the type quantifiers on $\alpha$ and $\beta$ 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 $\alpha$ and $\beta$.

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

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

[2] which we actually covered in lecture

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

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 $y$.

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

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.

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

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.

How much parallelism do we have here? Certainly, the two recursive calls to *mapreduce* can proceed in parallel. With fork/join parallelism we would then have to synchronize and form the triple from the results before calling $f$. With the futures-based parallelism [**?**] in our language we don't have wait to form the pair, which means that the computation of $f$ can proceed in parallel with the two recursive calls. Depending on the nature of $f$, this could mean some asymptotic improvement of the parallel complexity of an algorithm [**?**].

## 4 Recovering Sequentiality[3]

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.

[3] Bonus material not covered in lecture

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.

## 5 Message-Passing Concurrency

The map/reduce example, and also the bit-flipping example from Lecture 19 suggest also a *message-passing* interpretation. For example, in the bit-flipping example each of the two processes receives a stream of bits and sends a stream of bits. In the map/reduce example each node in the tree receives results from the recursive calls, combines them, and passes them up the tree. That raises the question on how message-passing is distinguished from our form of write-once shared memory (also called futures).

Recall that in our language (ignoring polymorphism) we have

| | | | | |
|---|---|---|---|---|
| Small values | $V$ | ::= | $\langle\rangle \mid \langle a_1, a_2\rangle \mid j \cdot a \mid \mathsf{fold}\ a$ | |
| Continuations | $K$ | ::= | $(\langle\rangle \Rightarrow P) \mid (\langle x_1, x_2\rangle \Rightarrow P) \mid (i \cdot x_i \Rightarrow P_i)_{i\in I} \mid (\mathsf{fold}\ x \Rightarrow P)$ | |
| Processes | $P$ | ::= | $x \leftarrow P\ ;\ Q \mid x \leftarrow y$ | |
| | | | $\mid x^W.V \mid \mathsf{case}\ x^R\ K$ | $(1, \times, +, \mu)$ |
| | | | $\mid x^R.V \mid \mathsf{case}\ x^W\ K$ | $(\rightarrow, \&)$ |
| Dynamic Objects | $\mathcal{C}$ | ::= | $!\mathsf{cell}\ a\ V$ | $(1, \times, +, \mu)$ |
| | | $\mid$ | $!\mathsf{cell}\ a\ K$ | $(\rightarrow, \&)$ |
| | | $\mid$ | $\mathsf{proc}\ a\ P$ | |
| | | $\mid$ | $(\cdot)$ | |
| | | $\mid$ | $(\mathcal{C}_1, \mathcal{C}_2)$ | |

We see a clear reflection of the duality between values and continuation and positive $(1, \times, +, \mu)$ and negative $(\rightarrow, \&)$ types. Here, it is reasonable to have continuations in cells the code for the continuation may be represented by an address in memory to jump to to execute it. Because these defined functions are shared between the threads in the same address space, this does not present an issue (even though, ultimately, avoiding substitution in favor of building environments and closures does).

For message-passing concurrency we would like to never pass a continuation. Because it contains references to other process definitions, and those again to other definitions, etc. it is well-known problem in the implementation of languages with message-passing concurrency that we may have to

"send the world", which is not a reasonable operation. So instead of sending a function and applying it to arguments where it is received, we keep in place and send it its arguments. In this manner, we can keep all messages to be *small values* $V$. And at runtime we have *communication channels* instead of (abstract) addresses. Under this interpretation, in a first pass, we have the following set of semantic objects:

proc $P$ — process $P$ (omitting the destination)

msg $a\ V$ — message $V$ on channel $a$

srv $a\ K$ — service $K$ waiting to receive a message on channel $a$

It turns out some of these should be persistent, but let's first consider the fundamental rule of interaction:

$$\mathsf{msg}\ a\ V, \mathsf{srv}\ a\ K \ \mapsto\ \mathsf{proc}\ (V \triangleright K)$$

where $V \triangleright K = P$ passes a value to a continuation and returns a process (see Lecture 18.9). In addition we can hypothesize the following simple rules (postponing the case of a process $a \leftarrow c$ for now):

$$\begin{array}{lcll} \mathsf{proc}\ (x \leftarrow P\ ;\ Q) & \mapsto & \mathsf{proc}\ [a/x]P, \mathsf{proc}\ [a/x]Q & \text{(a fresh)} \\ \mathsf{proc}\ (a.V) & \mapsto & \mathsf{msg}\ a\ V & \\ \mathsf{proc}\ (\mathsf{case}\ a\ K) & \mapsto & \mathsf{srv}\ a\ K & \end{array}$$

However, we need to differentiate these further depending on the polarity of the type. When a process is typed as

$$\underbrace{x_1 : \tau_1, \ldots, x_n : \tau_n}_{\text{client of channels } x_i} \ \vdash P :: \ \underbrace{(y : \sigma)}_{\text{providing channel } y}$$

we see some asymmetry in the definition. Just like previously an address may have many readers but only one writer, now a channel has one provider but potentially many clients. This means a message *sent* by $P$ along $y$ may need to be received by multiple clients and therefore must be *persistent*. This is the case for *positive* types $\sigma$. Conversely, a message send by $P$ along some $x_i$ has a unique recipient (the provider of $x_i$) and should therefore be ephemeral and be consumed when it is received. This is the case for *negative* types $\tau_i$. If we annotate channels with $()^+$ if they are of positive type and

$()^-$ if they are of negative type, we then obtain the following rules (omitting the cases for $a \leftarrow c$ for the moment)

$$\begin{array}{lcll}
\mathsf{proc}\ (x \leftarrow P\ ;\ Q) & \mapsto & \mathsf{proc}\ [a/x]P, \mathsf{proc}\ [a/x]Q & \text{(a fresh)} \\
\mathsf{proc}\ (a^+.V) & \mapsto & !\mathsf{msg}\ a^+\ V & \\
\mathsf{proc}\ (\mathsf{case}\ a^+\ K) & \mapsto & \mathsf{srv}\ a^-\ K & \\
\mathsf{proc}\ (a^-.V) & \mapsto & \mathsf{msg}\ a^-\ V & \\
\mathsf{proc}\ (\mathsf{case}\ a^-\ K) & \mapsto & !\mathsf{srv}\ a^-\ K & \\
!\mathsf{msg}\ a^+\ V, \mathsf{srv}\ a^+\ K & \mapsto & \mathsf{proc}\ (V \triangleright K) & \\
!\mathsf{srv}\ a^-\ K, \mathsf{msg}\ a^-\ V & \mapsto & \mathsf{proc}\ (V \triangleright K) &
\end{array}$$

Under shared memory interpretation, the process of $\mathsf{proc}\ a\ (a^W \leftarrow c^R)$ just copies the content of cell $c$ to $a$. When this contents is a small value $V$ the corresponding construct is a service to *forward* a message from channel $c$ to $a$. When this contents is a continuation $K$ the corresponding construct is a message that causes the service on channel $a$ to be replicated on channel $c$.

$$\begin{array}{lcl}
\mathsf{proc}\ (a^+ \leftarrow c^+) & \mapsto & \mathsf{srv}\ c^+\ a^+ \\
!\mathsf{msg}\ c^+\ V, \mathsf{srv}\ c^+\ a^+ & \mapsto & !\mathsf{msg}\ a^+\ V \\
\mathsf{proc}\ (a^- \leftarrow c^-) & \mapsto & \mathsf{msg}\ a^-\ c^- \\
!\mathsf{srv}\ a^-\ K, \mathsf{msg}\ a^-\ c^- & \mapsto & !\mathsf{srv}\ c^-\ K
\end{array}$$

For most constructs it is now relatively straightforward to establish a bisimulation relation $R$. We annotate the channels with read/write mode for shared memory and their polarity for message-passing.

$$\begin{array}{lcl}
\mathsf{proc}\ d\ P & R & \mathsf{proc}\ P \\
!\mathsf{cell}\ c\ V & R & !\mathsf{msg}\ c^+\ V \\
!\mathsf{cell}\ c\ K & R & !\mathsf{srv}\ c^-\ K \\
\mathsf{proc}\ d\ (c^R.V) & R & \mathsf{msg}\ c^-\ V \\
\mathsf{proc}\ d\ (\mathsf{case}\ c^R\ K) & R & \mathsf{srv}\ c^+\ K \\
\mathsf{proc}\ d\ (d^W \leftarrow c^W) & R & \mathsf{srv}\ c^+\ d^+ \\
\mathsf{proc}\ d\ (d^W \leftarrow c^W) & R & \mathsf{msg}\ d^-\ c^-
\end{array}$$

In the last two cases the translation depends on the polarity of the type for $d$ and $c$ (which must be the same). This relation induces a clear and simple correspondence on the configurations.

Then the relation between the steps is very simple: every step on the shared memory side corresponds to one or two steps on the message passing

side. The extra step is induced by the read operations, because in messsage-passing these reads take an extra step to become either a service (positive types) or a message (negative types). Similarly, an extra step is required for the copying operations because the copying process must become a message or a service.

Conversely, some steps on the message-passing side are the identity on the shared memory side, following the same reasoning as above.

Shared memory configurations are *final* when they consist only of cells. Correspondingly, message-passing configurations are *final* when they consist only of persistent messages and services. On final configuration, our intuitive notion of observability derived from our functional language also coincide, namely, we can observe cells with small values on the shared memory side and (positive and hence persistent) messages on the message-passing side.

We conclude that the same source language can be given coherent interpretations either using shared memory or message-passing in the semantics.

## 6 Example: A Nor Gate

Because we did not have to change our source language, just the dynamics, the same programs we wrote for shared memory will still be type-correct—they just have different behavior. As a new example we consider a *nor* gate and an *or* gate built from it, but with a slightly different representation of bit streams from before. We specify:

$\mathit{bit} = (\mathbf{b0} : 1) + (\mathbf{b1} : 1)$
$\mathit{bits} = \mathit{bit} \times \mathit{bits}$
$\mathit{bits2} = (\mathit{bit} \times \mathit{bit}) \times \mathit{bits2}$

In our functional language, *bits* and *bits2* would be empty types; here in the concurrent message-passing setting we don't worry about this. Also, we interpret the types *equirecursively* in order to avoid unnecessary fold messages.

We specify a process client to channels $x_i : \tau_i$ and providing $y : \sigma$ with $x_1 : \tau_1, \ldots, x_n : \tau_n \vdash p :: (y : \sigma)$. We define it with the notation $y \leftarrow p\, x_1 \ldots x_n = P$, recorded in a global signature $\Sigma$. We invoke $p$ with the process expression $b \leftarrow p\, a_1 \ldots a_n$ with the typing rule

$$\frac{x_1 : \tau_1, \ldots, x_n : \tau_n \vdash p :: (y : \sigma) \in \Sigma \quad a_i : \tau_i \in \Gamma \text{ (for all } 1 \leq i \leq n)}{\Gamma \vdash b \leftarrow p\, a_1 \ldots a_n :: (b : \sigma)}\ \mathsf{call}$$

Dynamically, we just substitute the concrete channels for the variables in the process definition. We abbreviate sequence of channels and variables with $\overline{a}$ and $\overline{x}$.

$$\mathsf{proc}\ (b \leftarrow p\,\overline{a}) \ \mapsto\ \mathsf{proc}\ ([\overline{a}/\overline{x}, b/y]P) \quad \text{for } y \leftarrow p\,\overline{x} = P \in \Sigma$$

A *nor* of two bits now becomes

$$\begin{array}{l}
\mathit{bit} = (\mathbf{b0} : 1) + (\mathbf{b1} : 1) \\
x : \mathit{bit}, y : \mathit{bit} \vdash \mathit{nor} :: (z : \mathit{bit}) \\
z \leftarrow \mathit{nor}\, x\, y = \mathsf{case}\ x\ (\ \mathbf{b0} \cdot x' \Rightarrow \mathsf{case}\ y\ (\ \mathbf{b0} \cdot y' \Rightarrow z' \leftarrow z'.\langle\,\rangle\ ;\ z.(\mathbf{b1} \cdot z') \\
\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad \mid \mathbf{b1} \cdot x' \Rightarrow z' \leftarrow z'.\langle\,\rangle\ ;\ z.(\mathbf{b0} \cdot z')\ ) \\
\qquad\qquad\qquad\qquad\qquad\qquad \mid \mathbf{b1} \cdot x' \Rightarrow \mathsf{case}\ y\ (\ \mathbf{b0} \cdot y' \Rightarrow z' \leftarrow z'.\langle\,\rangle\ ;\ z.(\mathbf{b0} \cdot z') \\
\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad \mid \mathbf{b1} \cdot x' \Rightarrow z' \leftarrow z'.\langle\,\rangle\ ;\ z.(\mathbf{b0} \cdot z')\ )\ )
\end{array}$$

Here, the continuation channels $x'$ and $y'$ are an artifact of our representation. They must carry the value $\langle\,\rangle$ of type 1. We could use either one of them when we send a message along the output channel $z$ but we chose to generate a fresh channel $z'$ in order to maintain symmetry. It will also help is in the next lecture when we consider (clock-based) timing of communication.

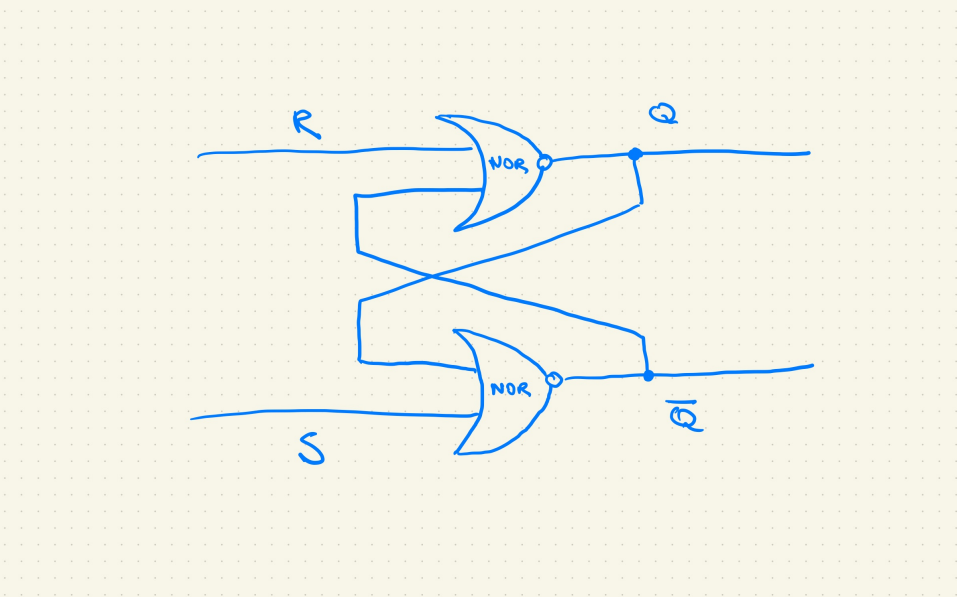
In order to transform a stream of pairs of bits into a stream of outputs we call the *nor* gate on each pair of inputs.

$$\begin{array}{l}
\mathit{bit} = (\mathbf{b0} : 1) + (\mathbf{b1} : 1) \\
x : \mathit{bit}, y : \mathit{bit} \vdash \mathit{nor} :: (z : \mathit{bit}) \\
\\
\mathit{bits} = \mathit{bit} \times \mathit{bits} \\
\mathit{bits2} = (\mathit{bit} \times \mathit{bit}) \times \mathit{bits2} \\
\\
\mathit{in} : \mathit{bits2} \vdash \mathit{nors} :: (\mathit{out} : \mathit{bits}) \\
\mathit{out} \leftarrow \mathit{nors}\ \mathit{in} = \\
\quad \mathsf{case}\ \mathit{in}\ (\ \langle\langle x, y\rangle, \mathit{in}'\rangle \Rightarrow \mathit{out}' \leftarrow \mathit{nors}\ \mathit{in}'\ ; \\
\qquad\qquad\qquad\qquad\qquad\quad z' \leftarrow \mathit{nor}\ x\ y\ ; \\
\qquad\qquad\qquad\qquad\qquad\quad \mathit{out}.\langle z', \mathit{out}'\rangle\ )
\end{array}$$

We have taken a small liberty where we match directly against the pair $\langle x, y\rangle$ instead of using two nested case expressions.

A more complicated example is a *latch* which can serve as a form of

storage for a bit. It feeds outputs back into inputs.




We assume (in a way that is certainly not accurate with respect to hardware) that after inputs are available are one clock cycle we obtain the output (say, of a nor gate). Therefore, to represent a latch we assume we have some initial values of the *outputs* $q, \overline{q}$ and generate new values for them in one cycle.

$$
\begin{array}{l}
q : \mathit{bit}, \overline{q} : \mathit{bit}, \mathit{in} : \mathit{bits2} \vdash \mathit{latch} :: (\mathit{out} : \mathit{bits2}) \\
\mathit{out} \leftarrow \mathit{latch}\ q\ \overline{q}\ \mathit{in} = \\
\quad \mathsf{case}\ \mathit{in}\ (\langle\langle r, s\rangle, \mathit{in}'\rangle \Rightarrow q' \leftarrow \mathit{nor}\ r\ \overline{q}\ ; \\
\qquad\qquad\qquad\qquad\qquad\quad \overline{q}' \leftarrow \mathit{nor}\ s\ q\ ; \\
\qquad\qquad\qquad\qquad\qquad\quad \mathit{out}' \leftarrow \mathit{latch}\ q'\ \overline{q}'\ \mathit{out} \\
\qquad\qquad\qquad\qquad\qquad\quad \mathit{out}.\langle\langle q', \overline{q}'\rangle, \mathit{out}'\rangle\ )
\end{array}
$$

Now we can simulate the behavior of the latch and make some interesting observations. Note that $q$ and $\overline{q}$ are supposed to carry complementary values. Then if $r$ and $s$ are both **b0** then $q$ and $\overline{q}$ retain their value. If we *reset* with a **b1** on the $R$ channel and **b0** on the $S$ channel, then $q$ becomes **b0** and $\overline{q}$ becomes **b1**. Conversely, if we *set* with b1 on $S$ and **b0** on $R$ then $q$ becomes **b1** and $\overline{q}$ becomes **b0**. However, there is a delay of one cycle for the set or reset to take effect. Below is a simulation with each line representing one

time step.

| $t$ | $Q$ | $\overline{Q}$ | $R$ | $S$ | $Q'$ | $\overline{Q}'$ | note |
|---|---|---|---|---|---|---|---|
| 0 | 1 | 0 | 0 | 0 | 1 | 0 | initial |
| 1 | 1 | 0 | 0 | 0 | 1 | 0 | stable |
| 2 | 1 | 0 | 1 | 0 | 0 | 0 | reset; intermediate state |
| 3 | 0 | 0 | 1 | 0 | 0 | 1 | stabilized |
| 4 | 0 | 1 | 0 | 0 | 0 | 1 | stable |
| 5 | 0 | 1 | 0 | 0 | 0 | 1 | stable |
| 6 | 0 | 1 | 0 | 1 | 0 | 0 | set; intermediate state |
| 7 | 0 | 0 | 0 | 1 | 1 | 0 | stabilized |
| 8 | 0 | 1 | 0 | 0 | 1 | 0 | stable |

Note that a state where both $R$ and $S$ are **b1** is disallowed. You might want to play through what happens in this case. Also note that the set and reset bits on $R$ and $S$ need to be in effect for at least two cycles and go back to **b0** before the next set or reset signal arrives.

For more information, see SR NOR latch, for example, on the Wikipedia page flip-flop.

## 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:

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

(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 4 is *eager* in the sense that in $x \Leftarrow P\ ;\ Q$, $P$ completes by writing to $x$ before $Q$ starts.

A lazy version, $x \leftarrow P\ ;\ Q$ 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 $x \leftharpoonup P \mathbin{;} Q$ 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.