Lecture Notes on Negative Types

15-814: Types and Programming Languages
Frank Pfenning

Lecture 20
Tuesday, November 10, 2020

1 Introduction

We continue the investigation of shared memory concurrency by adding *negative types*. In our language so far they are functions $\tau \to \sigma$, lazy pairs $\tau \mathbin{\&} \sigma$, and universal types $\forall\alpha.\,\tau$.

2 Review of Positives

We review the types so far, with a twist: we annotate every address that we write to with a superscript^W and every address we read from with a superscript^R.

Processes	P	::=	$x \leftarrow P\,;\,Q$		allocate/spawn
		$\mid$	$x^W \leftarrow y^R$		copy
		$\mid$	$x^W.\langle\,\rangle$	$\mid$ case x^R $(\langle\,\rangle \Rightarrow P)$	(1)
		$\mid$	$x^W.\langle y, z\rangle$	$\mid$ case x^R $(\langle y, z\rangle \Rightarrow P)$	$(\times)$
		$\mid$	$x^W.(j \cdot y)$	$\mid$ case x^R $(i \cdot y \Rightarrow P_i)_{i\in I}$	$(+)$
		$\mid$	x^W.fold y	$\mid$ case x^R (fold $y \Rightarrow P$)	(ρ)
Small Values	V	::=	$\langle\,\rangle \mid \langle a_1, a_2\rangle \mid j \cdot a \mid$ fold a		
Configurations	$\mathcal{C}$	::=	$\cdot \mid \mathcal{C}_1, \mathcal{C}_2 \mid$ proc $d\ P \mid$!cell $c\ V$		

The configurations are unordered and we think of "," as an associative and commutative operator with unit ".". Since we have changed our notation a few times, we summarize the translation and the transition rules.

$$[\![x]\!]\, d = d^W \leftarrow x^R$$

$$[\![\langle\,\rangle]\!]\, d = d^W.\langle\,\rangle$$
$$[\![\mathsf{case}\ e\ (\langle\,\rangle \Rightarrow e')]\!]\, d = x \leftarrow [\![e]\!]\, x\ ;\ \mathsf{case}\ x^R\ (\langle\,\rangle \Rightarrow [\![e']\!]\, d)$$

$$[\![\langle e_1, e_2\rangle]\!]\, d = x_1 \leftarrow [\![e_1]\!]\, x_1\ ;\ x_2 \leftarrow [\![e_2]\!]\, x_2\ ;\ d^W.\langle x_1, x_2\rangle$$
$$[\![\mathsf{case}\ e\ (\langle x_1, x_2\rangle \Rightarrow e')]\!]\, d = x \leftarrow [\![e]\!]\, x\ ;\ \mathsf{case}\ x^R\ (\langle x_1, x_2\rangle \Rightarrow [\![e']\!]\, d)$$

$$[\![j \cdot e]\!]\, d = x \leftarrow [\![e]\!]\, x\ ;\ d^W.(j \cdot x)$$
$$[\![\mathsf{case}\ e\ (i \cdot x \Rightarrow e_i)_{i\in I}]\!]\, d = x \leftarrow [\![e]\!]\, x\ ;\ \mathsf{case}\ x^R\ (i \cdot x \Rightarrow [\![e_i]\!]\, d)_{i\in I}$$

$$[\![\mathsf{fold}\ e]\!]\, d = x \leftarrow [\![e]\!]\, x\ ;\ d^W.(\mathsf{fold}\ x)$$
$$[\![\mathsf{case}\ e\ (\mathsf{fold}\ y \Rightarrow e')]\!]\, d = x \leftarrow [\![e]\!]\, x\ ;\ \mathsf{case}\ x^R\ (\mathsf{fold}\ y \Rightarrow [\![e']\!]\, d)$$

To show the computation rules for *configurations* we refactor the specifications, separating out *continuations* K.

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	::=	$x \leftarrow P\ ;\ Q$	$(allocate/spawn)$
		$\mid$	$c \leftarrow d$	$(copy)$
		$\mid$	$d^W.V$	$(write)$
		$\mid$	$\mathsf{case}\ c^R\ K$	$(read/match)$

We only have four transition rules for configurations, in addition to explaining how values are matched against continuations.

$$\begin{array}{lllll}
 & \mathsf{proc}\ d\ (x \leftarrow P\ ;\ Q) & \mapsto & \mathsf{proc}\ c\ ([c/x]P), \mathsf{proc}\ d\ ([c/x]Q) & (c\ \text{fresh}) \\
!\mathsf{cell}\ c\ V, & \mathsf{proc}\ d\ (d \leftarrow c) & \mapsto & !\mathsf{cell}\ d\ V & \\
 & \mathsf{proc}\ d\ (d.V) & \mapsto & !\mathsf{cell}\ d\ V & \\
!\mathsf{cell}\ c\ V, & \mathsf{proc}\ d\ (\mathsf{case}\ c\ K) & \mapsto & \mathsf{proc}\ d\ (V \triangleright K) &
\end{array}$$

$$\begin{array}{rcll} \langle\,\rangle & \triangleright & (\langle\,\rangle \Rightarrow P) & = & P \\ \langle c_1, c_2\rangle & \triangleright & (\langle x_1, x_2\rangle \Rightarrow P) & = & [c_1/x_1, c_2/x_2]P \\ k \cdot c & \triangleright & (i \cdot x_i \Rightarrow P_i)_{i\in I} & = & [c/x_k]P_k \\ \mathsf{fold}\ c & \triangleright & (\mathsf{fold}\ x \Rightarrow P) & = & [c/x]P \end{array}$$

3 Functions

As the first negative type we consider function $\tau \to \sigma$. How do we translate an abstraction $\lambda x.\, e$? The translation must actually take *two* arguments: one is the original argument x, the other is the destination where the result of the functional call should be written to. And the process $[\![\lambda x.\, e]\!]\, d$ must write the translation of the function to destination d.

Before we settle on the syntax for this, consider how to translate function application.

$$\begin{array}{l} [\![e_1\, e_2]\!]\, d = x_1 \leftarrow [\![e_1]\!]\, x_1\ ; \\ \qquad\qquad\quad x_2 \leftarrow [\![e_2]\!]\, x_2\ ; \\ \qquad\qquad\quad \boxed{\qquad\qquad} \end{array}$$

How should we complete this translation?

We know that after $[\![e_1]\!]\, x_1$ has completed the cell x_1 will contain *a function of two arguments*. The *first* argument is the original argument, which we find in x_2 after $[\![e_2]\!]\, x_2$ has completed. The *second* argument is the destination for the result of the function application, which is d. So we get:

$$\begin{array}{l} [\![e_1\, e_2]\!]\, d = x_1 \leftarrow [\![e_1]\!]\, x_1\ ; \\ \qquad\qquad\quad x_2 \leftarrow [\![e_2]\!]\, x_2\ ; \\ \qquad\qquad\quad x_1^R.\langle x_2, d\rangle \end{array}$$

This looks just like eager pairs, except that we *read* from x_1 instead of writing to it. To retain the analogy, we write the translation of a function using case, but *writing* the (single) branch of the case expression to memory.

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

The transition rules for these new constructs just formalize the explanation.

$$\begin{array}{ll} \mathsf{proc}\ d\ (\mathsf{case}\ d^W\ (\langle x, y\rangle \Rightarrow P)) \mapsto \mathsf{!cell}\ d\ (\langle x, y\rangle \Rightarrow P) & (\to R) \\ \mathsf{!cell}\ c\ (\langle x, y\rangle \Rightarrow P), \mathsf{proc}\ d\ (c^R.\langle c_1, d\rangle) \mapsto \mathsf{proc}\ d\ ([c_1/x, d/y]P) & (\to L^0) \end{array}$$

As an example, we consider the expression $(\lambda x.\, x)\,\langle\,\rangle$.

$$
\begin{aligned}
[\![(\lambda x.\, x)\ \langle\,\rangle]\!]\, d_0 &= x_1 \leftarrow [\![\lambda x.\, x]\!]\, x_1\ ; \\
&\quad\ x_2 \leftarrow [\![\langle\,\rangle]\!]\, x_2\ ; \\
&\quad\ x_1^R.\langle x_2, d_0\rangle \\
&= x_1 \leftarrow \mathsf{case}\ x_1^W\ (\langle x, y\rangle \Rightarrow [\![x]\!]\, y)\ ; \\
&\quad\ x_2 \leftarrow x_2^W.\langle\,\rangle\ ; \\
&\quad\ x_1^R.\langle x_2, d_0\rangle \\
&= x_1 \leftarrow \mathsf{case}\ x_1^W\ (\langle x, y\rangle \Rightarrow y^W \leftarrow x^R)\ ; \\
&\quad\ x_2 \leftarrow x_2^W.\langle\,\rangle\ ; \\
&\quad\ x_1^R.\langle x_2, d_0\rangle
\end{aligned}
$$

Let's execute the final process from with the initial destination d_0.

$$
\begin{aligned}
&\quad \mathsf{proc}\ d_0\ (x_1 \leftarrow \mathsf{case}\ x_1^W\ (\ldots)\ ;\ x_2 \leftarrow x_2^W.\langle\,\rangle\ ;\ \ldots) \\
&\mapsto \quad \mathsf{proc}\ d_1\ (\mathsf{case}\ d_1^W\ (\langle x, y\rangle \Rightarrow y^W \leftarrow x^R)), \\
&\quad\ \ \mathsf{proc}\ d_0\ (x_2 \leftarrow x_2^W.\langle\,\rangle\ ;\ d_1^R.\langle x_2, d_0\rangle) \\
&\mapsto^2 \quad \mathsf{!cell}\ d_1\ (\langle x, y\rangle \Rightarrow y^W \leftarrow x^R), \\
&\quad\ \ \mathsf{proc}\ d_2\ (d_2^W.\langle\,\rangle), \\
&\quad\ \ \mathsf{proc}\ d_0\ (d_1^R.\langle d_2, d_0\rangle) \\
&\mapsto^2 \quad \mathsf{!cell}\ d_1\ (\langle x, y\rangle \Rightarrow y^W \leftarrow x^R), \\
&\quad\ \ \mathsf{!cell}\ d_2\ \langle\,\rangle, \\
&\quad\ \ \mathsf{proc}\ d_0\ (d_0^W \leftarrow d_2^R) \qquad\qquad \text{(from } [d_2/x, d_0/y](y^W \leftarrow x^R)) \\
&\mapsto \quad \mathsf{!cell}\ d_1\ (\langle x, y\rangle \Rightarrow y^W \leftarrow x^R), \\
&\quad\ \ \mathsf{!cell}\ d_2\ \langle\,\rangle \\
&\quad\ \ \mathsf{!cell}\ d_0\ \langle\,\rangle
\end{aligned}
$$

In the final configuration we have cell d_0 holding the final result $\langle\,\rangle$, which is indeed the result of evaluating $(\lambda x.\, x)\ \langle\,\rangle$. We also have some newly allocated intermediate destinations d_1 and d_2 that are preserved, but could be garbage collected if we only retain the cells that are reachable from the initial destination d_0 which now holds the final value.

4 Store Revisited

In our table of process expression, two things stand out. One is that functions are exactly like pairs, except that the role of reads and writes are reversed.

The other is that a cell may now contain something of the form $(\langle y, z\rangle \Rightarrow P)$.

Processes	P	$::=$	$x \leftarrow P\,;Q$		allocate/spawn
		$\mid$	$x^W \leftarrow y^R$		copy
		$\mid$	$x^W.\langle\,\rangle$	$\mid \mathsf{case}\ x^R\ (\langle\,\rangle \Rightarrow P)$	(1)
		$\mid$	$x^W.\langle y, z\rangle$	$\mid \mathsf{case}\ x^R\ (\langle y, z\rangle \Rightarrow P)$	$(\times)$
		$\mid$	$x^W.(j \cdot y)$	$\mid \mathsf{case}\ x^R\ (i \cdot y \Rightarrow P_i)_{i\in I}$	$(+)$
		$\mid$	$x^W.\mathsf{fold}(y)$	$\mid \mathsf{case}\ x^R\ (\mathsf{fold}(y) \Rightarrow P)$	(ρ)
		$\mid$	$x^R.\langle y, z\rangle$	$\mid \mathsf{case}\ x^W\ (\langle y, z\rangle \Rightarrow P)$	$(\rightarrow)$

We can refactor this into a more uniform presentation, even though not all of the syntactically legal forms have corresponding types in the current language.

Processes	P	$::=$	$x \leftarrow P\,;Q$		allocate/spawn
		$\mid$	$x^w \leftarrow y^R$		copy
		$\mid$	$x^W.V$	$\mid \mathsf{case}\ x^R\ K$	$(1, \times, +, \rho)$
		$\mid$	$x^R.V$	$\mid \mathsf{case}\ x^W\ K$	$(\rightarrow)$

Small values	V	$::=$	$\langle\,\rangle \mid \langle a_1, a_2\rangle \mid i \cdot a \mid \mathsf{fold}\ a$
Continuations	K	$::=$	$(\langle\,\rangle \Rightarrow P) \mid (\langle x_1, x_2\rangle \Rightarrow P) \mid (i \cdot x_i \Rightarrow P_i)_{i\in I} \mid (\mathsf{fold}\ x \Rightarrow P)$
Cell contents	W	$::=$	$V \mid K$
Configurations	$\mathcal{C}$	$::=$	$\cdot \mid \mathcal{C}_1, \mathcal{C}_2 \mid \mathsf{proc}\ d\ P \mid !\mathsf{cell}\ c\ W$

There is now a legitimate concern that the contents of cells in memory is no longer "small", because a program P could be of arbitrary size. At a lower level of abstraction, continuations would probably be implemented as *closures*, that is, a pairs consisting of an environment and the address of code to be executed. The translation to get us to this form is called *closure conversion*, which we might discuss in a future lecture. For now, we are content with the observation that, yes, we are violating a basic principle of fixed-size storage and that it can be mitigated (but is not completely solved) through the introduction of closures.

In our example of $(\lambda x.\, x)\ \langle\,\rangle$ the continuation has the form $(\langle x, y\rangle \Rightarrow y^W \leftarrow x^R)$ which is a closed process. This can be directly compiled to a function that takes two addresses x and y and writes the contents of x into y. So at least in this special case the contents of the cell d_1 could simply be the address of this piece of code.

The symmetry between eager pairs (positive) and functions (negative) stems from the property that in logic we have $A \vdash B \supset C$ if and only if $A \times B \vdash C$ (where $\times$ is a particular form of conjunction). Or, we can chalk it up to the isomorphism $\tau \to (\sigma \to \rho) \cong (\tau \times \sigma) \to \rho$: an arrow on the right behaves like a product on the left.

One can ask if similarly symmetric constructors exists for 1, $+$, and ρ and the answer is yes. It turns out that lazy records are symmetric to sums and there is a type $\bot$ that is symmetric to 1 (see Exercises 1 and 2). There may even be a lazy analogue of recursive types that exhibits the same kind of symmetry and maybe useful to model so-called corecursive types (see Exercise 3).

We postpone discussion on the typing of process expression, cells, and configurations until the next lecture when we consider analogues of the progress and preservation theorems.

5 Typing

Before writing an example, it may be helpful to revisit the typing in its factored form. We separate out the positives, since the typing for the negatives is not quite as uniform as one might expect.

To type the contents of cells directly, we have the judgment $\Gamma \vdash V : \tau$ for positive τ.

$$\frac{}{\Gamma \vdash \langle\,\rangle : 1}\ \mathsf{val/unit} \qquad \frac{y : \tau \in \Gamma \qquad z : \sigma \in \Gamma}{\Gamma \vdash \langle y, z\rangle : \tau \times \sigma}\ \mathsf{val/prod}$$

$$\frac{(j \in I) \quad y : \tau_j \in \Gamma}{\Gamma \vdash j \cdot y : \sum_{i \in I}(i : \tau_i)}\ \mathsf{val/sum} \qquad \frac{y : [\rho\alpha.\,\tau/\alpha]\tau \in \Gamma}{\Gamma \vdash \mathsf{fold}\ y : \rho\alpha.\,\tau}\ \mathsf{val/fold}$$

Process typing for the positives is now unified, but we still separate out the negative with some special-purpose rules. For positive types τ we also have a judgment to verify that a value $V : \tau$ is matched against a suitable continuation, $\Gamma \vdash \tau \triangleright K :: (z : \sigma)$.

$$\frac{\Gamma \vdash V : \tau}{\Gamma \vdash x^W.V :: (x : \tau)}\ \mathsf{write/pos} \qquad \frac{x : \tau \in \Gamma \qquad \Gamma \vdash \tau \triangleright K :: (z : \sigma)}{\Gamma \vdash \mathsf{case}\ x^R\ K :: (z : \sigma)}\ \mathsf{read/pos}$$

$$\frac{\Gamma \vdash P :: (z : \sigma)}{\Gamma \vdash 1 \triangleright (\langle\,\rangle \Rightarrow P) :: (z : \sigma)}\ \mathsf{m/unit} \qquad \frac{\Gamma, x_1 : \tau_1, x_2 : \tau_2 \vdash P :: (z : \sigma)}{\Gamma \vdash \tau_1 \times \tau_2 \triangleright (\langle x_1, x_2\rangle \Rightarrow P) : (z : \sigma)}\ \mathsf{m/prod}$$

$$\frac{(\text{for all } i \in I) \quad \Gamma, y : \tau_i \vdash P_i :: (z : \sigma)}{\Gamma \vdash \sum_{i\in I}(i : \tau_i) \triangleright (i(y) \Rightarrow P_i) :: (z : \sigma)}\ \mathsf{m/sum} \qquad \frac{\Gamma, y : [\rho\alpha.\,\tau/\alpha]\tau \vdash P :: (z : \sigma)}{\Gamma \vdash \rho\alpha.\,\tau \triangleright (\mathsf{fold}(y) \Rightarrow P) :: (z : \sigma)}\ \mathsf{m/rho}$$

For the negative types (here only functions), we have somewhat more specific rules. They arise, because for the type $\tau \to \sigma$ the types τ and σ are on different sides of the turnstile.

$$\frac{x : \tau \to \sigma \in \Gamma \quad y : \tau \in \Gamma}{\Gamma \vdash x^R.\langle y, z\rangle :: (z : \sigma)}\ \mathsf{read/arrow} \qquad \frac{\Gamma, y : \tau \vdash P :: (z : \sigma)}{\Gamma \vdash \mathsf{case}\ x^W\ (\langle y, z\rangle \Rightarrow P) :: (x : \tau \to \sigma)}\ \mathsf{write/arrow}$$

6 Example: A Pipeline

As a simple example for concurrency in this language we consider setting up a (very small) pipeline. We consider a sequence of bits

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

(which also happen to be isomorphic to binary numbers). We assume there is a process $\mathit{flip} : \mathit{bits} \to \mathit{bits}$ that just flips every bit. We will write this during the next lecture; for this lecture the goal is to compose two such processes in a pipeline.

Assume there is a cell

$$!\mathsf{cell}\ \mathit{flip}\ K_{\mathit{flip}} : \mathit{bits} \to \mathit{bits}$$

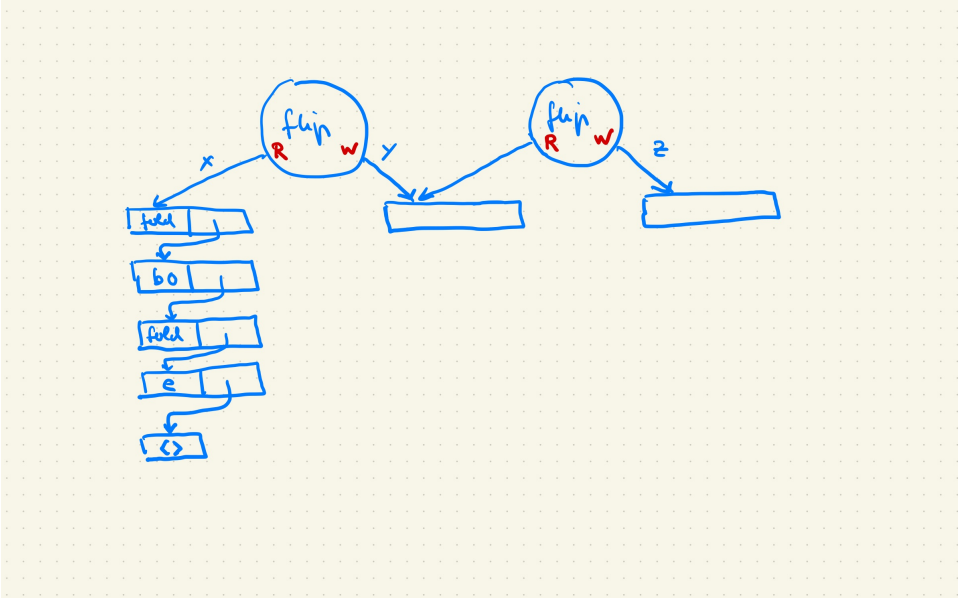
This means that $K_{\mathit{flip}} = (\langle x, y\rangle \Rightarrow P)$ where $x : \mathit{bits}$ is address of the argument and $y : \mathit{bits}$ is the destination for the result.

Then we can compose two of these as

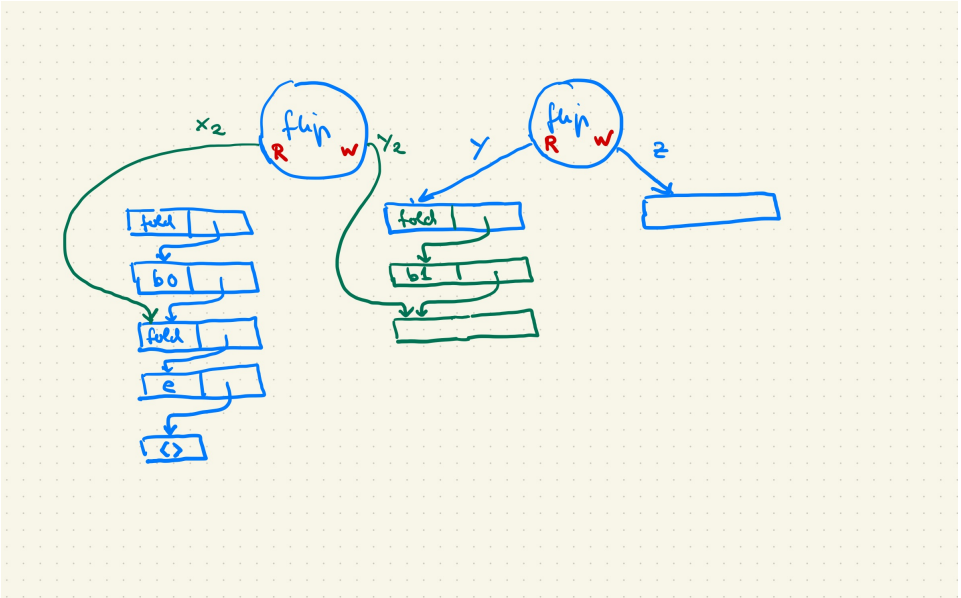
$$\begin{aligned} K_{\mathit{flip2}} = \langle x, z\rangle \Rightarrow\ & \\ & y \leftarrow \mathit{flip}^R.\langle x, y\rangle \\ & \mathit{flip}^R.\langle y, z\rangle \end{aligned}$$

In the picture below we see the two *flip* processes running, after the code for *kflip2* has executed but neither of these has taken any action yet. The process on the left reads from x and writes to y while the process on the right reads from y and writes to z. The destinations y and z have been allocated but

have not yet been written to. Cell x contains the sample input, which is the memory representation of fold $(\mathbf{b0} \cdot \text{fold}\ (\mathbf{e}\ \cdot \langle\rangle))$.

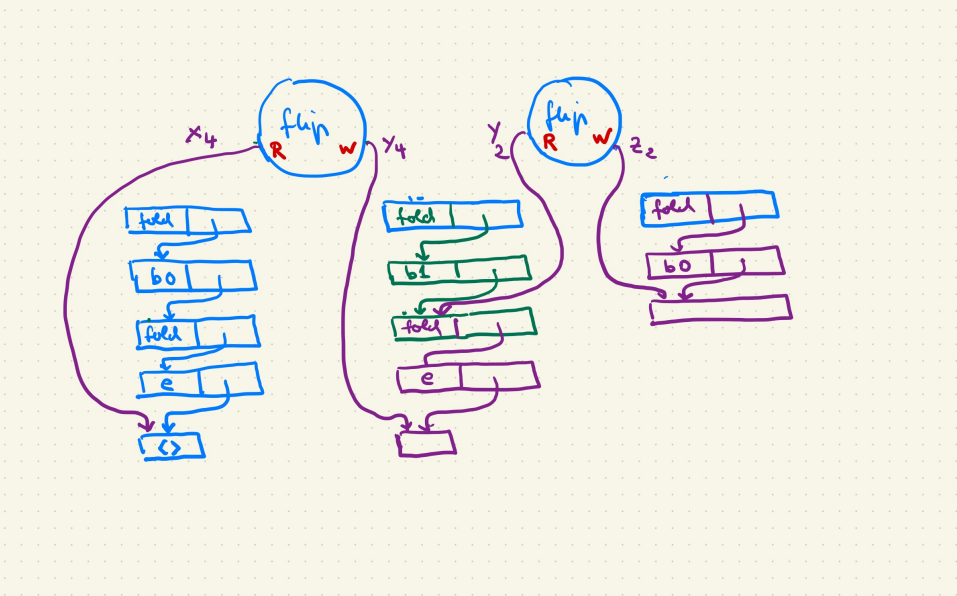


The left process now reads along x and allocates and writes along y. After it runs for a few steps, we might reach the following situation:



The green part here is the new part compared to the previous configuration. It should be clear how each of the two processes translates into a proc object, while each filled cell corresponds to a cell object. The empty cells are addresses that have been allocated, but not yet written to, so they are not explicit in the configuration.

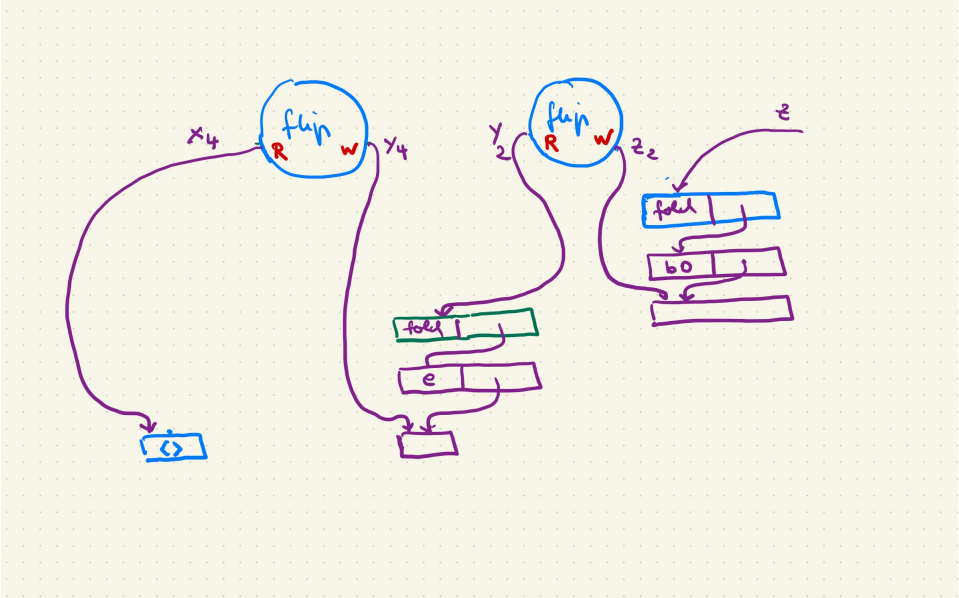
The two processes also run in parallel, which is how they form a pipeline. For example, after a few more steps we might reach the configuration (with the purple part being new):



The right process here lags behind the left one, which is possible since the semantics here is not synchronous. A cell can be read as soon as it is filled, but it may not be read immediately while other computations take place.

If we knew that the left process was the only reader along x (and any cells reachable from it) we could "garbage-collect" the cells that are no longer accessible and the situation would look as follows (assuming here

some process not shown could read the output z).



In the next lecture we will write the code for *flip* that can exhibit the shown behavior. In a future lecture we will consider a type system that tracks if cells have unique readers which will allow the eager deallocation of cells that have been read.

Exercises

Exercise 1 For lazy records (as a generalization of lazy pairs) we introduce the following syntax in our language of expressions:

$$\begin{array}{lll} \text{Types} & ::= & \ldots \mid \&_{i \in I}(i : \tau_i) \\ \text{Expressions} & ::= & \ldots \mid \langle\!| i \Rightarrow e_i |\!\rangle_{i \in I} \mid e \cdot j \end{array}$$

1. Give the typing rules and the dynamics (stepping rules) for the new constructs.

2. Extend the translation $[\![e]\!]\, d$ to encompass the new constructs. Your process syntax should expose the duality between eager sums and lazy records.

3. Extend the transition rules of the store-based dynamics to the new constructs. The translated form may permit more parallelism than the

original expression evaluation, but when scheduled sequentially they should have the same behavior (which you do not need to prove).

4. Show the typing rules for the new process constructs.

Exercise 2 Explore what the rules and meaning of $\bot$ as the formal dual of 1 in the process language should be, including whichever of the following you find make sense. If something does not make sense somehow, please explain.

1. Write out the new forms of process expressions.
2. Provide the store-based dynamics for the new process expressions.
3. Show the typing rules for the new process expressions.
4. Reverse-engineer new functional expressions in our original language so they translate to your new process expression. Show the rules for typing and stepping the new constructs.
5. Summarize and discuss what you found.

Exercise 3 In our expression language the fold e constructor for elements of recursive type is eager. Explore a new *lazy* ravel e constructor which has type $\delta\alpha.\,\tau$, providing:

1. Typing rules for ravel and a corresponding destructor (presumably an unravel or case construct).
2. Stepping rules for the new forms of expressions.
3. A translation from the new forms of expressions to processes, extending the language of processes as needed
4. Typing rules for the new forms of processes.
5. Transition rules for the new forms of processes.