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 $\tau \& \sigma$, 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

$P ::= x \leftarrow P ; Q$
$\mid x^W \leftarrow y^R$
$\mid x^W . \langle \rangle$
$\mid x^W . \langle y, z \rangle$
$\mid x^W . j(y)$
$\mid x^W . \text{fold}(y)$

allocate/spawn

$\mid \text{copy}$
$\mid \text{case } x^R (\langle \rangle \Rightarrow P)$ (1)
$\mid \text{case } x^R (\langle y, z \rangle \Rightarrow P)$ (×)
$\mid \text{case } x^R (i(y) \Rightarrow P)_{i \in I}$ (+)
$\mid \text{case } x^R (\text{fold}(y) \Rightarrow P)$ (ρ)

Cell Contents

$W ::= \langle \rangle \mid \langle d_1, d_2 \rangle \mid j(d) \mid \text{fold}(d)$

Configurations

$C ::= \cdot \mid C_1, C_2 \mid \text{proc } d \ P, \text{cell } d \_ \mid !\text{cell } c \ W$

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. We also
As the first negative type we consider function \( J \) proc \( \lambda x.e \) the translation of the function to destination is the original argument \( x \). 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 \( d \).

### 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.

\[ [[e_1 e_2] \ d = d_1 \leftarrow \ [e_1] \ d_1 ; \]
\[ \quad d_2 \leftarrow \ [e_2] \ d_2 ; \]

How should we complete this translation?

We know that after \([e_1] \ d_1\) has completed the cell \(e_1\) will contain a function of two arguments. The first argument is the original argument \(x\), which we find in \(d_2\) after \([e_2] \ d_2\) has completed. The second argument is the destination for the result of the functional application, which is \(d\). So we get:

\[ [[e_1 e_2] \ d = d_1 \leftarrow \ [e_1] \ d_1 ; \]
\[ \quad d_2 \leftarrow \ [e_2] \ d_2 ; \]
\[ \quad d_1'.(d_2, d) \]

This looks just like eager pairs, except that we read from \(d_1\) 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.

\[ [[\lambda x. \ e] \ d = \ case \ d^W \ (\langle x, y \rangle \Rightarrow [e] y) \]

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

\[
\begin{align*}
\text{proc } d \ (\text{case } d \ ((\langle x, y \rangle \Rightarrow P)), \ \text{cell } d\_ & \mapsto \text{!cell } d \ ((\langle x, y \rangle \Rightarrow P)) & \quad (\rightarrow R) \\
\text{!cell } c \ ((\langle x, y \rangle \Rightarrow P)), \ \text{proc } d \ (c.(c_1, c_2)) & \mapsto \text{proc } d \ ([c_1/x, c_2/y] P) & \quad (\rightarrow L^0)
\end{align*}
\]

As an example, we consider the expression \((\lambda x. \ e)\).

\[ [[(\lambda x. \ e)] \ d_0 = d_1 \leftarrow \ [\lambda x. \ e] \ d_1 ; \]
\[ \quad d_2 \leftarrow \ [\langle \rangle] \ d_2 ; \]
\[ \quad d_1'.(d_2, d_0) \]
\[ = d_1 \leftarrow \ \text{case } d^W_1 \ ((\langle x, y \rangle \Rightarrow [e] y) ; \]
\[ \quad d_2 \leftarrow d^W_2.\langle \rangle ; \]
\[ \quad d_1'.(d_2, d_0) \]
\[ = d_1 \leftarrow \ \text{case } d^W_1 \ ((\langle x, y \rangle \Rightarrow y^W \leftarrow x^R) ; \]
\[ \quad d_2 \leftarrow d^W_2.\langle \rangle ; \]
\[ \quad d_1'.(d_2, d_0) \]

Let’s execute the final process from with the initial destination \(d_0\).
\[
\text{proc } d_0 (d_1 \leftarrow \text{case } d_1^W (\ldots) ; d_2 \leftarrow d_2^R ; \ldots), \text{cell } d_0 \\
\Rightarrow \text{proc } d_1 \text{ (case } d_1^W ((x, y) \Rightarrow y^W \leftarrow x^R), \text{cell } d_1_\_)
\]

\[
\Rightarrow^2 \text{!cell } d_1 ((x, y) \Rightarrow y^W \leftarrow x^R),
\]

\[
\text{proc } d_0 (d_2 \leftarrow d_2^R, \text{cell } d_0_\_)
\]

In the final state 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 \((\langle y, z \rangle \Rightarrow P)\).

\[
\begin{align*}
\text{Processes } P &::= x \leftarrow P ; Q \\
&| x^W \leftarrow y^R \\
&| x^W.\langle \rangle \leftarrow y^R \\
&| x^W.(y, z) \leftarrow y^R \\
&| x^W.j(y) \leftarrow y^R \\
&| x^W.fold(y) \leftarrow y^R \\
&| x^R.(y, z) \leftarrow y^R \\
\end{align*}
\]

allocate/spawn

\[
\begin{align*}
&| \text{case } x^R (\langle \rangle \Rightarrow P) \\
&| \text{case } x^R (\langle y, z \rangle \Rightarrow P) \\
&| \text{case } x^R (i(y) \Rightarrow P_{i \in I}) \\
&| \text{case } x^R (\text{fold}(y) \Rightarrow P) \\
&| \text{case } x^R (\langle y, z \rangle \Rightarrow P) \\
\end{align*}
\]

\[
\begin{align*}
&\Rightarrow^2 \text{!cell } d_1 ((x, y) \Rightarrow y^W \leftarrow x^R),
\end{align*}
\]

\[
\text{proc } d_2 (\text{cell } d_2_\_)
\]

\[
\Rightarrow^2 \text{!cell } d_2 \langle \rangle
\]

\[
\text{!cell } d_0 \langle \rangle
\]
Negative Types

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. We call expressions following “case $x$” continuations $K$ (which are different from the continuations in the K Machine). A continuation will be actually implemented as a closure, that is, a pair 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 $(\lambda x. x) ()$ the continuation has the form $(\langle x, y \rangle \Rightarrow y^W \leftarrow x^R)$ which is a closed expression in that has no free variables. 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 $\tau \rightarrow (\sigma \rightarrow \rho) \cong (\tau \times \sigma) \rightarrow \rho$: an arrow on the right behaves like a product on the left.

One can ask if similarly symmetric constructors exists for $1$ and $+$ and the answer is yes. It turns out that lazy pairs are symmetric to sums and there is a type $\bot$ that is symmetric to $1$ (see Exercises 2 and 3). 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 4).

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 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

\[ \text{bits} \cong (b0 : \text{bits}) + (b1 : \text{bits}) + (e : 1) \]

(which also happen to be isomorphic to binary numbers). Here is a function $\text{flip} : \text{bits} \rightarrow \text{bit}$ that just flips every bit. We use our pattern matching syntax.
with constructors

\[
B_0 : \text{bits} \rightarrow \text{bits} \\
B_1 : \text{bits} \rightarrow \text{bits} \\
E : \text{bits}
\]

\(\text{flip} : \text{bits} \rightarrow \text{bits}\)

\(\text{flip} (B_0 y) = B_1 (\text{flip} y)\)

\(\text{flip} (B_1 y) = B_0 (\text{flip} y)\)

\(\text{flip} E = E\)

In preparation for translation to process form, we write it out in more explicit notation. For the sake of brevity, we skip the fold/unfold constructors, which can easily be added.

\[

\text{flip} = \text{flip} (B_0 y) = B_1 (\text{flip} y) \\
\text{flip} E = E

\]

Rather than addressing how fixed points are translated in their full generality, we merely derived a recursive specification for the process \([\text{flip}]\).

We didn’t do this in lecture, but we can apply some optimizations to see more easily how this process executes. First, since the definition of \(\text{flip}\) is closed, we assume its definition is stored in a fixed cell \(f_0\) (never mind that the store is then circular). Also, we can replace the left-hand side below with the right-hand side

\[
d_3 \leftarrow (d_3^W \leftarrow y^R) ; P \equiv [y/d_3]P
\]

to arrive at

\[

\text{cell} f_0 \langle x, z \rangle \Rightarrow \text{case} x^R (B_0 \cdot y \Rightarrow d_1 \leftarrow f_0^R \langle y^R, d_1 \rangle) ; \\
\text{z}^W.B1(d_1) \\\n\text{B1} \cdot y \Rightarrow \ldots \\
\text{E} \cdot y \Rightarrow \text{E} \cdot y )
\]

Now imagine we would like to compute \(\text{flip} (B_0(B_1(E)))\) The argument will be represent in memory by (again ignoring folds)
The call to \textit{flip} with destination \textit{d} would be

\begin{verbatim}
proc d0 (fR (.c0, d0)), cell d0 _
\end{verbatim}

When this process reads the contents \textit{c} and takes the B0 branch, it allocates a new destination \textit{d} for the result of the recursive call, but meanwhile it can already write B0(\textit{d}) into the destination \textit{d}. So we have

\begin{verbatim}
proc d1 (fR (.c1, d1))
\end{verbatim}

The significance in this state is that the process has already written part of the output (into destination \textit{d}) before having read all of the input. For example, if this where the inner flip of \textit{flip (flip (B0(B1(E)))}) then the outer flip could now read the destination \textit{d} and output B1(\textit{a}) while the inner flip had only read one bit of input.

So we see that under a concurrent interpretation the composition \textit{flip} \textit{..} \textit{flip} behaves like a pipeline with two processes, \textit{flip} \textit{..} \textit{flip} \textit{..} \textit{flip} behaves like a pipeline with three processes, etc.

Under a sequential interpretation, where \textit{x} \leftarrow \textit{P ; Q} waits until \textit{P} has written to destination \textit{x} before \textit{Q} starts executing, all recursive calls in \textit{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.

\textbf{Exercises}

\textbf{Exercise 1} A \textit{lazy record} is a generalization of a lazy pair where each alternative has a different label \textit{i}. For example, potentially infinite streams \textit{stream \alpha} of elements of some type \textit{\alpha} may be defined as

\begin{verbatim}
stream \alpha \equiv (hd : \alpha) \& (tl : stream \alpha)
\end{verbatim}
As an example of the general syntax $\langle i \mapsto e \rangle_{i \in I}$ for a lazy record with the fields in the finite index set $I$, we show how to define a stream of just 0s (omitting the standard definitions of `zero` and `succ`):

$$
nat \cong (z : 1) + (s : nat)$$

$$
zero : nat
succ : nat \to nat
zeros : stream nat
zeros = \text{fold} \langle hd \Rightarrow zero, tl \Rightarrow zeros \rangle
$$

In fully explicit form, the definition of `zeros` would be a fixed point:

$$
zeros = \text{fix} \ f. \text{fold} \langle hd \Rightarrow zero, tl \Rightarrow f \rangle
$$

but we prefer the first form where the recursion is implicit. This definition terminates because the record with field `hd` and `tl` is lazy. We select an element of a lazy record $e$ by writing $e \cdot j$ for a label $j$ (which is just the postfix version of the injection into a sum $j \cdot e$). As an example, the following function adds 1 to every elements of the given stream.

$$
succs : stream nat \to stream nat
succs = \lambda s. \langle hd \Rightarrow succ (((unfold s) \cdot hd), tl \Rightarrow succs (((unfold s) \cdot tl)) \rangle
ones = succs zeros
$$

Write functions satisfying the following specifications:

1. `up from : nat \to stream nat` where `up from n` generates the stream $n, n + 1, n + 2, \ldots$.  
2. `alt : \forall \alpha. stream \alpha \to stream \alpha \to stream \alpha` which alternates the elements from the two streams, starting with the first element of the first stream.  
3. `filter : \forall \alpha. (\alpha \to bool) \to stream \alpha \to stream \alpha` which returns the stream with just those elements of the input stream that satisfy the given predicate.  
4. `map : \forall \alpha. \forall \beta. (\alpha \to \beta) \to (stream \alpha \to stream \beta)` which returns a stream with the result of applying the given function to every element of the input stream.  
5. `diag : \forall \alpha. stream (stream \alpha) \to stream \alpha` which returns a stream consisting of the first element of the first stream, the second element of the second stream, the third element of the third stream, etc.
You may use earlier functions in the definition of later ones. To avoid some recomputation, you may use the syntactic sugar of let $x = e$ in $e'$ to stand for $(\lambda x. e') e$.

Your functions should be such that only as much of the output stream is computed as necessary to obtain a value of type stream $\alpha$ but not the components contained in the lazy record. For example, the definition of $suc$ below would be still terminating, but slightly too eager (for example, we may never access the element at the head of the resulting stream), while the second $suc''$ would not even be terminating any more.

\[
\begin{align*}
suc' &= \lambda s. \text{let } x = \text{suc } ((\text{unfold } s) \cdot \text{hd}) \\
& \quad \text{ in } \langle \text{hd }\Rightarrow x, t1 \Rightarrow suc' ((\text{unfold } s) \cdot t1) \rangle \\
suc'' &= \lambda s. \text{let } s' = suc'' ((\text{unfold } s) \cdot t1) \\
& \quad \text{ in } \langle \text{hd }\Rightarrow suc ((\text{unfold } s) \cdot \text{hd}), t1 \Rightarrow s' \rangle\
\end{align*}
\]

Exercise 2 For lazy records as introduced in Exercise 1 we introduce the following syntax in our language of expressions:

\[
\begin{align*}
\text{Types} &::= \ldots \cup \&_{i \in I}(i : \tau_i) \\
\text{Expressions} &::= \ldots \cup I \cdot e_i}_{i \in I} | e \cdot j
\end{align*}
\]

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 3 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 4 In our expression language the fold $e$ constructor for elements of recursive type is eager. Explore a new lazy ravel $e$ constructor, providing:

1. Typing rules for knit and a corresponding destructor (presumably an unravel or a 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. Transition rules for the new forms of processes.