Lecture Notes on Linear Types

15-814: Types and Programming Languages
Frank Pfenning

Lecture 23
Thursday, November 21, 2019

1 Introduction

We have already seen linear types for expressions in the last lecture. We didn't prove preservation and progress, and they are still satisfied, but not satisfying: we haven't changed any of the dynamics of programs! Linear types, so far, "don't buy us anything".

In this lecture we assign linear types to processes, so that the translation of a linearly typed functional expression becomes a linearly typed process. Then we show that executing a linearly typed process does not require a garbage collector since we can eagerly deallocate cells when they are read. In other words, the right level of abstraction to benefit from linear typing is at a level where memory is made explicit.

Linear typing, though, is too restrictive so what we actually want is a language that combines linear with nonlinear typing. In this combination, linearly typed cells are ephemeral, while other cells remain persistent as in our original semantics for processes. We probably will not have time to cover such a language in this course, but refer you to a recent draft paper [PP19].

2 Linear Typing of Processes

Our judgment is

$$\Gamma \,;\, \Delta \vdash P :: (z : \sigma)$$

where Γ contains recursion variables/destinations and Δ contains linear variables/destinations. The destination z in the succedent is written to exactly once (as before), but it will also be read exactly once. Therefore, the rule for spawn/allocate is

$$\frac{\Gamma \,;\, \Delta \vdash P :: (x : \tau) \quad \Gamma \,;\, \Delta, x : \tau \vdash Q :: (z : \sigma)}{\Gamma \,;\, \Delta, \Delta' \vdash x \leftarrow P \,;\, Q :: (z : \sigma)}\ \mathsf{spawn}$$

In the computation rule, we just create a fresh cell as before.

$$\mathsf{proc}\ d\ (x \leftarrow P \,;\, Q) \mapsto \mathsf{proc}\ c\ ([c/x]P), \mathsf{cell}\ c\ \_, \mathsf{proc}\ d\ ([c/x]Q) \qquad (c\ \text{fresh})$$

We have two rules for variables: one that reads from an ephemeral (linear) cell and one that reads from a persistent (recursive) cell.

$$\frac{}{\Gamma \,;\, y : \tau \vdash x^W \leftarrow y^R :: (x : \tau)}\ \mathsf{move} \qquad \frac{f : \tau \in \Gamma}{\Gamma \,;\, \cdot \vdash x^W \leftarrow f^R :: (x : \tau)}\ \mathsf{copy}$$

Computationally, this first rule *moves* while the second one *copies*.

$$\mathsf{cell}\ c\ W, \mathsf{proc}\ d\ (d \leftarrow c), \mathsf{cell}\ d\ \_ \mapsto \mathsf{cell}\ d\ W \qquad \text{(move)}$$
$$!\mathsf{cell}\ f\ W, \mathsf{proc}\ d\ (d \leftarrow f), \mathsf{cell}\ d\ \_ \mapsto \mathsf{cell}\ d\ W \qquad \text{(copy)}$$

There should also be rules to *write* to a persistent cell, but we take a small shortcut here and restrict ourselves to top-level recursion between closed functions. Such functions can be compiled and become "read-only memory" at the time of translation, before the program is executed.

Eager Linear Pairs. As an example for linear typing, we use pairs. In general, we write linear typing rules as *left rules* (if the type constructor appears in the antecedent) and *right rules* (if the type constructor appears in the succedent). Note that left rules always read from memory, while right rules always write to memory.

$$\frac{}{\Gamma \,;\, x_1 : \tau_1, x_2 : \tau_2 \vdash z^W.\langle x_1, x_2 \rangle :: (z : \tau_1 \otimes \tau_2)}\ {\otimes}R^0$$

$$\frac{\Gamma \,;\, \Delta, x_1 : \tau_1, x_2 : \tau_2 \vdash P :: (z : \sigma)}{\Gamma \,;\, \Delta, x : \tau_1 \otimes \tau_2 \vdash \mathsf{case}\ x^R\ (\langle x_1, x_2 \rangle \Rightarrow P) :: (z : \sigma)}\ {\otimes}L$$

Operationally, the case rule reads from memory and passes it to the continuation. These rules are general for all positive types. The only difference from before is that the cell that is read is ephemeral and therefore "deallocated".

$\mathsf{proc}\ d\ (d^W.V), \mathsf{cell}\ d\ \_ \mapsto \mathsf{cell}\ d\ V$ (write/pos)
$\mathsf{cell}\ c\ V, \mathsf{proc}\ d\ (\mathsf{case}\ c^R\ K) \mapsto \mathsf{proc}\ d\ (V \triangleright K)$ (read/pos)

where

$$\begin{array}{llll} \text{Values} & V & ::= & \langle d_1, d_2 \rangle \mid \ldots \\ \text{Conts} & K & ::= & (\langle x_1, x_2 \rangle \Rightarrow P) \mid \ldots \end{array}$$

with

$$\langle d_1, d_2 \rangle \triangleright (\langle x_1, x_2 \rangle \Rightarrow P) = [d_1/x_1, d_2/x_2]P$$

Linear Sums. They follow the pattern of the eager pairs, since they are a positive type.

$$\frac{j \in I}{\Gamma\ ;\ y : \tau_j \vdash x^W.j(y) :: (x : \oplus_{i \in I}(i : \tau_i))}\ \oplus R^0$$

$$\frac{(\text{for all } i \in I) \quad \Gamma\ ;\ \Delta, y : \tau_i \vdash P_i :: (z : \sigma)}{\Gamma\ ;\ \Delta, x : \oplus_{i \in I}(i : \tau_i) \vdash \mathsf{case}\ x^R\ (i(y) \Rightarrow P_i)_{i \in I} :: (z : \sigma)}\ \oplus L$$

where

$$j(d) \triangleright (i(y) \Rightarrow P_i)_{i \in I} = [d/y]P_j$$

Linear functions. Since functions are a negative type, the case constructs writes a continuation to memory.

$$\frac{\Gamma\ ;\ \Delta, y : \tau \vdash P :: (z : \sigma)}{\Gamma\ ;\ \Delta \vdash \mathsf{case}\ x^W\ (\langle y, z \rangle \Rightarrow P) :: (x : \tau \multimap \sigma)}\ \multimap R$$

$$\frac{}{\Gamma\ ;\ x : \tau \multimap \sigma, y : \tau \vdash x^R.\langle y, z \rangle :: (z : \sigma)}\ \multimap L^0$$

This time, we have to provide a second set of rules since the roles of values and continuations are flipped.

$\mathsf{proc}\ d\ (\mathsf{case}\ d^W\ K), \mathsf{cell}\ d\ \_ \mapsto \mathsf{cell}\ d\ K$ (write/neg)
$\mathsf{cell}\ c\ K, \mathsf{proc}\ d\ (d^R.V) \mapsto \mathsf{proc}\ d\ (V \triangleright K)$ (read/neg)

where the reduction $\langle d_1, d_2 \rangle \triangleright (\langle y, z \rangle \Rightarrow P)$ has already been defined.

Recursion. We assume all functions can be mutually recursive and are defined at the top level and have no other free variables. Then we translate each definition

$$\mathit{func} = \lambda x.\, e$$

as

$$!\mathsf{cell}\, \mathit{func}\ (\langle x, z\rangle \Rightarrow [\![e]\!]\, z)$$

where

$$[\![\mathit{func}]\!]\, d = (d^W \leftarrow \mathit{func}^R)$$

Slightly more generally, if we want to allow mutually recursive definitions for arbitrary negative types constructed at the top level, we would translate each definition

$$f = e$$

to

$$!\mathsf{cell}\ f\ K \qquad \text{for } [\![e]\!]\, d_0 = \mathsf{case}\ d_0\ K$$

3 Example: Bit Flipping Revisited

With the treatment of recursion from the end of the previous section, the (linear) bit flipping program becomes:

$$\begin{array}{l} \mathit{flip\_proc} = (\langle x, z\rangle \Rightarrow \mathsf{case}\ x^R\ (\,\mathtt{B0}(y) \Rightarrow d_1 \leftarrow (d_2 \leftarrow (d_2^W \leftarrow \mathit{flip}^R)\ ;\\ \qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad\quad d_3 \leftarrow (d_3^W \leftarrow y^R)\ ;\\ \qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad\quad d_2^R.\langle d_3, d_1\rangle)\\ \qquad\qquad\qquad\qquad\qquad\qquad\qquad\quad z^W.\mathtt{B1}(d_1)\\ \qquad\qquad\qquad\qquad\quad \mid \mathtt{B1}(y) \Rightarrow \ldots\\ \qquad\qquad\qquad\qquad\quad \mid \mathtt{E}(y) \Rightarrow z^W.\mathtt{E}(y)\,)\,) \end{array}$$

where the initial state of running the program contains

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

This is now entirely linearly typed, except that the destination flip must be available to check its contents (in a recursion context Γ).

When looking at this program we observe some obvious opportunities for optimization. For example, we notice the pattern

$$x \leftarrow (x^W \leftarrow c^R)\ ;\ Q$$

which seems equivalent to

$$[c/x]Q$$

In order to see whether this might be a valid optimization, let's assume cell c has been filled so our configuration contains

$\mathsf{cell}\ c\ W, \mathsf{proc}\ d\ (x \leftarrow (x^W \leftarrow c^R)\ ;\ Q), \mathsf{proc}\ d\ \_$

Stepping from this initial configuration (and ignoring the empty cell d):

$$\begin{array}{ll} & \mathsf{cell}\ c\ W, \mathsf{proc}\ d\ (x \leftarrow (x^W \leftarrow c^R)\ ;\ Q) \\ \mapsto & \mathsf{cell}\ c\ W, \mathsf{proc}\ d_1\ (d_1^W \leftarrow c^R), \mathsf{cell}\ d_1\ \_, \mathsf{proc}\ d\ ([d_1/x]Q) \\ \mapsto & \mathsf{cell}\ d_1\ W, \mathsf{proc}\ d\ ([d_1/x]Q) \end{array}$$

On the other hand, the relevant part of the initial configuration after optimization is just

$$\mathsf{cell}\ c\ W, \mathsf{proc}\ d\ ([c/x]Q)$$

These two configurations differ only in the name of the cell, d_1 in the original configuration and c in the optimized one. In this dynamics we have the principle of *equivariance*, which means that two configurations that differ only in the names of their cells are indistinguishable. This is a generalization of the principle of α-conversion which states that two expressions that differ only in the names of their bound variables are indistinguishable. This important principle imposes some restriction on language extensions. For example, we cannot allow casts from destinations to natural numbers because that might allow us to distinguish the two programs above and we wouldn't be allowed to optimize.

Along similar lines, we briefly return to the dynamics where all cells are persistent. The same optimization above should still be valid. Before optimization we obtain the state

$$!\mathsf{cell}\ c\ W, !\mathsf{cell}\ d_1\ W, \mathsf{proc}\ d\ ([d_1/x]Q)$$

and after optimization

$$!\mathsf{cell}\ c\ W, \mathsf{proc}\ d\ ([c/x]Q)$$

Because cells are not ephemeral and processes may not be linear, $[d_1/x]Q$ (before optimization) might contain both c and d_1, while after optimization they are conflated to just c.

Now even an apparently harmless language extension such as allowing comparison of addresses (or "pointer comparison" in a function language) would destroy equivariance! This is because Q could contain a comparison $c == x$ which returns *false* in the unoptimized program (since $c \neq d_1$), but *true* in the optimized program (since $[c/x](c == x) = (c == c)$). Consider that a library may have the flip program originally implemented as

$$\begin{array}{llll}
\mathit{flip\_proc} = (\langle x, z\rangle \Rightarrow \mathsf{case}\ x^R\ (& \mathtt{B0}(y) \Rightarrow & d_1 \leftarrow (& d_2 \leftarrow (d_2^W \leftarrow \mathit{flip}^R)\ ; \\
 & & & d_3 \leftarrow (d_3^W \leftarrow y^R)\ ; \\
 & & & d_2^R.\langle d_3, d_1\rangle) \\
 & & z^W.\mathtt{B1}(d_1) & \\
 & \mid \mathtt{B1}(y) \Rightarrow \ldots & & \\
 & \mid \mathtt{E}(\_) \Rightarrow & d_1 \leftarrow d_1^W.\langle\,\rangle & \\
 & & z^W.\mathtt{E}(d_1)\,)\,) & \qquad \text{\% change here!}
\end{array}$$

Now the programmer might decide do avoid building a "new" $\langle\,\rangle$ on the right-hand side and replace the last line with

$$\mid \mathtt{E}(y) \Rightarrow z^W.\mathtt{E}(y)\,)\,)$$

This change in the library could now break the application code because two different addresses (the address of $\langle\,\rangle$ at the end of the *input* sequence of bits and the address of $\langle\,\rangle$ at the end of the *output* sequence of bits) are now equal since we no longer allocate a new one. This breaks some form of data abstraction and complicates reasoning about programs.

In ocaml there is a such an address-comparison operator == which, unfortunately, violates equivariance. As a library writer you cannot know what uses of the operator a client might make so you cannot safely replace the first version of *flip* with the second. This violates basic principles of modularity and data abstraction and therefore I consider it a design mistake (which is not present in Standard ML).

This is distinct from the comparison of mutable references. For mutable references we cannot apply a similar optimization since the identity (but not the actual address) of mutable references is important. When reasoning about programs with mutable references we need to be aware which references may alias each other and which not.

With this simple optimization we can simplify our *flip* process as

$$\begin{array}{lll}
\mathit{flip\_proc} = (\langle x, z\rangle \Rightarrow \mathsf{case}\ x^R\ (& \mathtt{B0}(y) \Rightarrow & d_1 \leftarrow \mathit{flip}^R.\langle y, d_1\rangle \\
 & & z^W.\mathtt{B1}(d_1) \\
 & \mid \mathtt{B1}(y) \Rightarrow & d_1 \leftarrow \mathit{flip}^R.\langle y, d_1\rangle \\
 & & z^W.\mathtt{B0}(d_1) \\
 & \mid \mathtt{E}(y) \Rightarrow & z^W.\mathtt{E}(y)\,)\,)
\end{array}$$

4 Look Ma, No Garbage!

With linear typing, cells are deallocated as they are used. For example, the flip program started with a state such as

$\mathsf{!cell}\ \mathit{flip}\ \mathit{flip\_proc}$,
$\mathsf{cell}\ c_4\ \langle\,\rangle, \mathsf{cell}\ c_3\ (\mathtt{E}\cdot c_4), \mathsf{cell}\ c_2\ (\mathsf{fold}\ c_3)$,
$\mathsf{cell}\ c_1\ (\mathtt{B0}\cdot c_2), \mathsf{cell}\ c_0\ (\mathsf{fold}\ c_1)$,
$\mathsf{proc}\ d_0\ (\mathit{flip}^R.\langle c_0, d_0\rangle)$

will end with a state

$\mathsf{!cell}\ \mathit{flip}\ \mathit{flip\_proc}$,
$\mathsf{cell}\ c_4\ \langle\,\rangle, \mathsf{cell}\ d_3\ (\mathtt{E}\cdot c_4), \mathsf{cell}\ d_2\ (\mathsf{fold}\ d_3)$,
$\mathsf{cell}\ d_1\ (\mathtt{B1}\cdot d_2), \mathsf{cell}\ d_0\ (\mathsf{fold}\ d_1)$

We have executed here the version that does not explicitly copy the unit element to a new cell. Note that all cells, except for *flip*, are reachable from d_0, the initial destination of the call.

In general, if we started with an empty configuration (again, excepting only the recursive functions), as would be the case for the translation of

$$[\![\mathit{flip}\ (\mathsf{fold}\ (\mathtt{B0}\cdot(\mathsf{fold}\ (\mathtt{E}\cdot\langle\,\rangle))))]\!]\ d_0$$

all cells in the resulting state would be reachable from d_0 as shown in this example.

In order to prove such a result we need to make the typing of configurations explicit and then examine the change in configurations during computation. We have:

$$\text{Configurations}\quad \mathcal{C} \quad ::= \quad \cdot \mid \mathcal{C}, \mathsf{proc}\ d\ P, \mathsf{cell}\ d\ \_ \mid \mathcal{C}, \mathsf{cell}\ c\ W \mid \mathcal{C}, \mathsf{!cell}\ f\ K$$

where the persistent cells are only for closed, top-level functions. We imagine they are defined in a global context In the typing judgment, the antecedents are always persistent because, as in the previous lecture, we are only interested in typing closed configurations (the analogue of closed expressions).

$$\Gamma\ ;\ \cdot \vdash \mathcal{C} :: \Delta$$

The point here is that $\mathcal{C}$ will write or has written to all the destinations in Δ.

$$\frac{}{\Gamma\ ;\ \cdot \vdash (\cdot) :: (\cdot)} \qquad \frac{\Gamma\ ;\ \cdot \vdash \mathcal{C} :: \Delta, \Delta' \quad \Gamma\ ;\ \Delta' \vdash P :: (d : \tau)}{\Gamma\ ;\ \cdot \vdash \mathcal{C}, \mathsf{proc}\ d\ P, \mathsf{cell}\ d\ \_ :: \Delta, d : \tau}$$

In the second rule, we divide all the ephemeral destinations into Δ' (read by P) and Δ (not read by P). Because cells are read only once, Δ' is consumed here and is not visible in the outside interface, which includes Δ and the cell d that's defined by the execution of P. The rule for ephemeral cells is analogous.

$$\frac{\Gamma \,;\, \cdot \vdash \mathcal{C} :: \Delta, \Delta' \quad \Gamma \,;\, \Delta' \vdash W : \tau}{\Gamma \,;\, \cdot \vdash \mathcal{C}, \mathsf{cell}\ d\ W :: \Delta, d : \tau}$$

For the persistent cells, we have a slightly different judgment

$$\Gamma \vdash \mathcal{C} :: \Gamma'$$

called at the top level with

$$\Gamma \vdash \mathcal{C} :: \Gamma$$

which means that every recursive function is (a) defined, and (b) can use any other function in mutual recursion.

$$\frac{}{\Gamma \vdash (\cdot) :: (\cdot)} \qquad \frac{\Gamma \vdash \mathcal{C} :: \Gamma' \quad \Gamma \,;\, \cdot \vdash P :: (f : \tau)}{\Gamma \vdash \mathcal{C}, !\mathsf{cell}\ f\ P :: \Gamma', f : \tau}$$

Now a whole configuration is well-typed if we can divide it into a persistent part $\mathcal{C}_1$ and an ephemeral part $\mathcal{C}_2$, and there is some typing Γ for the recursive variables that works for both parts (the definition as well as the use of the recursive variables).

$$\frac{\Gamma \vdash \mathcal{C}_1 :: \Gamma \quad \Gamma \,;\, \cdot \vdash \mathcal{C}_2 :: \Delta}{\vdash \mathcal{C}_1, \mathcal{C}_2 :: \Delta}$$

During the computation, only $\mathcal{C}_2$ will change.

5 Progress and Preservation

Progress is essentially unchanged from before.

Theorem 1 *If $\vdash \mathcal{C} :: \Delta$ then either $\mathcal{C}$ is final (consists only of cells) or $\mathcal{C} \mapsto \mathcal{C}'$ for some $\mathcal{C}'$.*

The preservation theorem is the interesting one. In case of linearly typed processes, the cells defined (or promised to be defined by a process) does not change throughout the computation!

Theorem 2 *If $\vdash \mathcal{C} :: \Delta$ and $\mathcal{C} \mapsto \mathcal{C}'$ then $\vdash \mathcal{C} :: \Delta$.*

Contrast this with the statement from Lecture 21 (Theorem 21.1) where the context can grow when the configuration takes a step. The proof here proceeds along the lines of the last lecture, with one interesting twist: when computation creates a fresh cell, it will always be together with a process that reads from it. Therefore, this cell does not show up in the interface on the right! Specifically:

$$\dfrac{\Gamma \;;\; \cdot \vdash \mathcal{C} :: \Delta, \Delta_1', \Delta_2' \qquad \dfrac{\Gamma \;;\; \Delta_1' \vdash P :: (x : \tau) \quad \Gamma \;;\; \Delta_2', x : \tau \vdash Q :: (d : \sigma)}{\Gamma \;;\; \Delta_1', \Delta_2' \vdash (x \leftarrow P \;;\; Q) :: (d : \sigma)}}{\Gamma \;;\; \cdot \vdash \mathcal{C}, \mathsf{proc}\ d\ (x \leftarrow P \;;\; Q), \mathsf{cell}\ d\ \_ :: \Delta, d : \sigma}$$

evolves to

$$\dfrac{\dfrac{\Gamma \;;\; \cdot \vdash \mathcal{C} :: \Delta, \Delta_1', \Delta_2' \quad \Gamma \;;\; \Delta_1' \vdash [c/x]P :: (c : \tau)}{\Gamma \;;\; \cdot \vdash \mathcal{C}, \mathsf{proc}\ c\ ([c/x]P), \mathsf{cell}\ c\ \_ :: \Delta, \Delta_2', c : \tau} \qquad \Gamma \;;\; \Delta_2', c : \tau \vdash [c/x]Q :: (d : \sigma)}{\Gamma \;;\; \cdot \vdash \mathcal{C}, \mathsf{proc}\ c\ ([c/x]P), \mathsf{cell}\ c\ \_, \mathsf{proc}\ d\ ([c/x]Q), \mathsf{cell}\ d\ \_ :: \Delta, d : \sigma}$$

The form of the preservation theorem now means that if we start, for example, with $\vdash \mathcal{C} :: (d_0 : 1)$ then any resulting final configuration $\mathcal{C} \mapsto^* \mathcal{C}'$ still has the same type. The persistent part is unchanged, and the linear part has type $\Gamma \;;\; \cdot \vdash \mathcal{C}_2 :: (d_0 : 1)$. Since there are only ephemeral cells in $\mathcal{C}_2$, it must be of the form $\mathcal{C}_2, \mathsf{cell}\ d_0\ W$ for some $\mathcal{C}_2$ and W. Since $d_0 : 1$, it follows by inversion that $W = \langle\,\rangle$. Moreover, $\Gamma \vdash \cdot \vdash \mathcal{C}_2 :: (\cdot)$. Again by inversion we find $\mathcal{C}_2 = (\cdot)$, so the whole configuration consists of just $\mathsf{cell}\ d_0\ \langle\,\rangle$.

Looking at the typing rules we can see that in general the context Δ acts like a frontier for an algorithm to traverse a tree with root d_0, the initial destination. It must eventually be empty which shows that every ephemeral cell is reachable and no garbage is created.

Exercises

Exercise 1 Prove that $\Gamma \;;\; \Delta \vdash e : \tau$ implies $\Gamma \;;\; \Delta \vdash [\![e]\!]\, d :: (d : \tau)$. You only need to show the cases relevant for functions ($\lambda x.\, e$, $e_1\, e_2$ and variables x).

Exercise 2 Write a linear function *inc* on the binary representation of natural numbers.

1. Provide the code as a functional expression.

2. Following the conventions of this lecture, show the result of the translation into a process expression. You may use the optimization we presented here. Concretely, define *inc_proc* and *inc* so that the program representation as a configuration would be !cell *inc inc_proc*.

3. Show the initial and final configuration of computation for incrementing the number 1 represented as $\mathsf{fold}\ (\mathtt{B1} \cdot (\mathsf{fold}\ (\mathtt{E} \cdot \langle\,\rangle)))$.

References

[PP19] Klaas Pruiksma and Frank Pfenning. Back to futures. Under submission, October 2019.