Lecture Notes on Linear Types

15-814: Types and Programming Languages
Frank Pfenning

Lecture 23
Thursday, November 19, 2020

1 Introduction

When we make memory explicit we have to face the problem of garbage collection, that is, freeing memory when it is no longer needed. Ideal would be to deallocate memory at the time we read from it (the last time). One particular interesting class of memory cells are those that have exactly one reader or a *unique reference*. In that case, we can deallocate when it is read. The usual rule (written here without persistent objects)

$$\mathsf{cell}\ c\ V, \mathsf{proc}\ d\ (\mathsf{case}\ c^R\ K) \mapsto \mathsf{cell}\ c\ V, \mathsf{proc}\ d\ (V \triangleright K)$$

would then become

$$\mathsf{cell}\ c\ V, \mathsf{proc}\ d\ (\mathsf{case}\ c^R\ K) \mapsto \mathsf{proc}\ d\ (V \triangleright K)$$

where we model deallocation of the cell c by not repeating it on the right-hand side of the rule.

This is not as infrequent as it might seem at first. For example, in our bit negation pipeline from Lecture 20 all the intermediate cells have a single reader, namely the second process in the pipeline.

Another class of examples comes from temporary cells in the translation of functional expressions.

$$\begin{aligned}[\![e_1\ e_2]\!]\ d = {} & x_1 \leftarrow [\![e_1]\!]\ x_1\ ; \\ & x_2 \leftarrow [\![e_2]\!]\ x_2\ ; \\ & x_1^R.\langle x_2, d\rangle\end{aligned}$$

The destination x_1 will be written by the translation $[\![e_1]\!]\, x_1$ and is then read by the last line. But it could not be used beyond that because it can not occur elsewhere in the program since x_1 is fresh and not passed to anywhere.

The situation is different for x_2. Even though it is freshly allocated here it is passed on to the function stored in x_1 so it "escapes its lexical scope" and we cannot deallocate it here.

Methodologically, we might now examine various constructs to see which destinations we may be able to "deallocate" by not copying them from the left-hand sides of transition rule to the right. But this is complicated, so first we examine what would be required so that we would *never* have to copy cells that are being read from (excluding mutable cells from consideration for the moment, for simplicity). Essentially, can we delineate a subsect of the language so that every cell will not only be *written to* once, but also *read from* once. Of course, as you might expect in this course after all we have been through together, this is expressed as a type system! Every memory cell will have not only a unique provider (to write it) but also a unique client (to read from it). We call a type system that enforces this property *linear*, after Girard's *linear logic* [Gir87].

2 Linear Expressions

Even though our ultimate goal is in the runtime system, we start with functional expressions. We say a *function* is linear in one of its arguments if it uses that argument exactly once. The notion of "usage" here is a dynamic one; it doesn't mean that the variable *occurs* exactly once, as we will see.

$$\lambda x.\, x \qquad \text{(linear)}$$

This is linear in x and therefore the whole expression is linear.

$$\lambda x.\, \lambda y.\, x \qquad \text{(not linear)}$$

This expression is linear in x but not linear in y and therefore not linear. It's not linear in y because y is not used, but linearity requires a single use. Related to linearity is the is the notion of *affine*. A function is *affine* in a variable if it is used *at most once*. So the function above is *affine* but not *linear*. The notion of affine has recently received a lot of attention because the Rust programming language treats memory references as affine.

$$\lambda x.\, \langle x, x \rangle \qquad \text{(not linear)}$$

This expression is not linear because x is used twice and hence more than once. Functions that use their argument *at least once* are called *strict*. The notion of strictness is important because it is useful in the optimization of call-by-need languages such as Haskell. If we have a function application $e_1\, e_2$ and we can tell that e_1 denotes a *strict* function we can safely evaluate e_2 rather than waiting until e_1 might need its argument.

$$\lambda x.\, \mathsf{if}\ x\ \mathit{false}\ x \qquad \text{(not linear)}$$

This function is not linear in x. It uses x the first time to decide the condition, and then again when x is false. However, if x is false this returns x which is *false*, so extensionally equal would be

$$\lambda x.\, \mathsf{if}\ x\ \mathit{false}\ \mathit{false} \qquad \text{(linear)}$$

which is linear. These two examples show that linearity is an intensional property of expressions (how do they compute) and not an extensional property (what do they compute).

$$\lambda x.\, \lambda y.\, \mathsf{if}\ x\ y\ (\mathit{not}\ y) \qquad \text{(linear)}$$

This function is linear: x is used once as subject of the conditional. The variable y occurs twice, but whenever this expression is executed it is used exactly once: if y is true then in the first branch, and if y is false then in the second branch.

$$\lambda x.\, (\lambda y.\langle\,\rangle)\, x \qquad \text{(not linear)}$$

It shouldn't be suprising by now that this is not linear, since y is not linear in $\langle\,\rangle$. But, moreover, the whole expression is not linear in x, even though x occurs exactly once. That's because x occurs in a position where it will be dropped. On the other hand:

$$\lambda x.\, (\lambda y.\, \langle y, \langle\,\rangle\rangle)\, x \qquad \text{(linear)}$$

3 Linear Typing of Expressions

With these examples, we now work through the inference rules for expressions and classify those that are linear. We use a different notation for functions, eager pairs, sums, etc. since the connectives are subtly different from the regular ones. Our judgment has the form

$$\Delta \Vdash e : \tau$$

where Δ is a context of variables, each of which must be used once in e. We have seen the $\Vdash$ notation once before, in Lecture 12 where we used it to type patterns in which no variables could be repeated. The use of Δ instead of Γ is just stylistic, to help remind ourselves that all the variables should be linear. We use here the names of the inference rules derived from *linear logic* where introductions rules (for constructors) use I while elimination rules (for destructors) use E.

Linear Functions $\tau \multimap \sigma$. A function is linear just if its parameter is used linearly in its body.

$$\frac{\Delta, x : \tau \Vdash e : \sigma}{\Delta \Vdash \lambda x.\, e : \tau \multimap \sigma} \multimap I$$

When applying a function we have to divide up the variables among those that occur in the function (Δ_1) and those that occur in the argument (Δ_2).

$$\frac{\Delta_1 \Vdash e_1 : \tau_2 \multimap \tau_1 \quad \Delta_2 \Vdash e_2 : \tau_2}{\Delta_1, \Delta_2 \Vdash e_1\, e_2 : \tau_1} \multimap E$$

Our usual presupposition regarding contexts kicks in and we implicit require the $\mathrm{dom}(\Delta_1) \cap \mathrm{dom}(\Delta_2) = \emptyset$. The ordering of the variables in Δ is irrelevant here. If we wanted to maintain them (say, because there are type variables present) then we would use a *merge* operator between the two contexts instead. Nevertheless, in the direction we usually read the rules it would be a *split* operator.

When we look up variables, there cannot be other variables in the context because they would not be used and therefore not be linear.

$$\frac{}{x : \tau \Vdash x : \tau}\ \mathsf{hyp}$$

Eager Linear Pairs $\tau \otimes \sigma$. Eager linear pairs are written as $\tau \otimes \sigma$. The rules are straightforwardly patterned after previous rules, keeping in mind that for the destructor (case), the variables standing for the components of the pair must be linear.

$$\frac{\Delta_1 \Vdash e_1 : \tau_1 \quad \Delta_2 \Vdash e_2 : \tau_2}{\Delta_1, \Delta_2 \Vdash \langle e_1, e_2 \rangle : \tau_1 \otimes \tau_2} \otimes I \qquad \frac{\Delta \Vdash e : \tau_1 \otimes \tau_2 \quad \Delta', x_1 : \tau_1, x_2 : \tau_2 \Vdash e' : \tau'}{\Delta, \Delta' \Vdash \mathsf{case}\ e\ (\langle x_1, x_2 \rangle \Rightarrow e') : \tau'} \otimes E$$

The nullary version of pairs, the unit is written as 1 and the rules are the nullary version of the binary rules above (see Section 4).

Linear Sums $\tau \oplus \sigma$. Actually, we will show the labeled, variadic version $\oplus_{i \in I}(i : \tau_i)$. In the constructor rule $\oplus I$, there is not much to consider.

$$\frac{\Delta \Vdash e : \tau_j}{\Delta \Vdash j \cdot e : \oplus_{i \in I}(i : \tau_i)} \oplus I$$

For the destructor (case) we need to consider the same as for the conditional in the last section: only one branch of the case will be taken, so all branches must be checked with the same linear context.

$$\frac{\Delta \Vdash e : \oplus_{i \in I}(i : \tau_i) \quad (\text{for all } i \in I) \quad \Delta', x_i : \tau_i \Vdash e'_i : \tau'}{\Delta, \Delta' \Vdash \mathsf{case}\ e\ (i \cdot x_i \Rightarrow e'_i)_{i \in I} : \tau'} \oplus E$$

Recursion. The remaining type constructors follow similar patterns so we omit the details (see Section 4 for a listing). Recursion, however, is interesting. The computation rule for fixed points is

$$\mathsf{fix}\ f.\, e \mapsto [\mathsf{fix}\ f.\, e/f]e$$

This already departed from the pattern of the other rules. For one, we substitute an expression ($\mathsf{fix}\ f.\, e$) for a variable f in an expression e, while all the other rules just substitute values for variables. For another, it is not attached to a particular type constructor and can always be applied.

There are several sources of operational "nonlinearity" in this rule. First, even if f occurs only once in e, it is replaced by another expression ($\mathsf{fix}\ f.\, e$) containing e, thereby duplicating e. Also, when we define a recursive function we would like to make multiple recursive calls and still consider the function linear.

For example, the function that takes a bit string (usually considered just a binary number) and flips every bit should be linear: each bit of the input string is read and a corresponding bit written to the output.

$$\mathit{bits} = \rho \mathit{bits}.\, (\mathtt{b0} : \mathit{bits}) \oplus (\mathtt{b1} : \mathit{bits}) \oplus (\mathtt{e} : 1)$$

$$\mathit{flip} : \mathit{bits} \multimap \mathit{bit}$$

$$\begin{aligned} \mathit{flip} = \lambda x.\, \mathsf{case}\ (\mathsf{unfold}\ x)\ (&\ \mathtt{b0} \cdot y \Rightarrow \mathsf{fold}\ (\mathtt{b1} \cdot \mathit{flip}\ y) \\ \mid &\ \mathtt{b1} \cdot y \Rightarrow \mathsf{fold}\ (\mathtt{b0} \cdot \mathit{flip}\ y) \\ \mid &\ \mathtt{e} \cdot y \Rightarrow \mathsf{fold}\ (\mathtt{e} \cdot y)\ ) \end{aligned}$$

Note that there is no recursive call to *flip* in the third branch and yet we should consider the function linear. In order to formally represent this,

we have to nonlinear variables to the context, which can be propagated to multiple premises of a rule and may be left over in rules with no premises. Moreover, since the body of the recursively defined expression is duplicated when it is unwound, it may not depend on any linear variables.

$$\frac{\Gamma_{\mathsf{U}}, f_{\mathsf{U}} : \tau \Vdash e : \tau}{\Gamma_{\mathsf{U}} \Vdash \mathsf{fix}\ f.\, e : \tau}\ \mathsf{rec}$$

Here, the subscript U means the variable is *unrestricted* (that is, non necessarily linear), and Γ_{U} stands for a context where all variables are unrestricted. The rules for variables, for example, then would become

$$\frac{}{\Gamma_{\mathsf{U}}, x : \tau \Vdash x : \tau}\ \mathsf{hyp} \qquad \frac{}{\Gamma_{\mathsf{U}}, x_{\mathsf{U}} : \tau \Vdash x : \tau}\ \mathsf{hyp}_{\mathsf{U}}$$

With these rules (and the straightforward ones for *fold* and *unfold*) the *flip* function can indeed be checked as linear.

This example is also remarkable because a tiny change in the last branch of the conditional

$$\begin{array}{l} \mathit{flip} : \mathit{bits} \multimap \mathit{bit} \\ \mathit{flip} = \lambda x.\, \mathsf{case}\ (\mathsf{unfold}\ x)\ (\ \mathtt{b0} \cdot y \Rightarrow \mathsf{fold}\ (\mathtt{b1} \cdot \mathit{flip}\ y) \\ \qquad\qquad\qquad\qquad \mid \mathtt{b1} \cdot y \Rightarrow \mathsf{fold}\ (\mathtt{b0} \cdot \mathit{flip}\ y) \\ \qquad\qquad\qquad\qquad \mid \mathtt{e} \cdot y \Rightarrow \mathsf{fold}\ (\mathtt{e} \cdot \langle\rangle)\ ) \qquad\qquad \%\ \text{bug here!} \end{array}$$

makes this function now nonlinear: y is not used. Besides the code shown earlier, we can also fix the problem by *using* $y : 1$.

$$\begin{array}{l} \mathit{flip} : \mathit{bits} \multimap \mathit{bit} \\ \mathit{flip} = \lambda x.\, \mathsf{case}\ (\mathsf{unfold}\ x)\ (\ \mathtt{b0} \cdot y \Rightarrow \mathsf{fold}\ (\mathtt{b1} \cdot \mathit{flip}\ y) \\ \qquad\qquad\qquad\qquad \mid \mathtt{b1} \cdot y \Rightarrow \mathsf{fold}\ (\mathtt{b0} \cdot \mathit{flip}\ y) \\ \qquad\qquad\qquad\qquad \mid \mathtt{e} \cdot y \Rightarrow \mathsf{case}\ y\ (\langle\rangle \Rightarrow \mathsf{fold}\ (\mathtt{e} \cdot \langle\rangle))\ ) \end{array}$$

4 Linear Rule Summary

The syntax for the language of expression does not change, but the language of types is new.

$$\text{Linear types} \quad \tau \quad ::= \quad \tau_1 \multimap \tau_2 \mid \tau_1 \otimes \tau_2 \mid 1 \mid \oplus_{i \in I}(i : \tau_i) \mid \rho\alpha.\, \tau \mid \alpha$$

The definition of values and the rules for evaluation remain the same as for our nonlinear functional language.

We name the propositional rules I (for introduction, representing a constructor for a type) and E (for elimination, representing a destructor for a type). Missing here are the unrestricted variables that would be needed for recursion.

$$\frac{}{x : \tau \Vdash x : \tau}\ \mathsf{hyp}$$

$$\frac{\Delta, x : \tau \Vdash e : \sigma}{\Delta \Vdash \lambda x.\, e : \tau \multimap \sigma}\ \multimap I \qquad \frac{\Delta_1 \Vdash e_1 : \tau_2 \multimap \tau_1 \quad \Delta_2 \Vdash e_2 : \tau_2}{\Delta_1, \Delta_2 \Vdash e_1\, e_2 : \tau_1}\ \multimap E$$

$$\frac{\Delta_1 \Vdash e_1 : \tau_1 \quad \Delta_2 \Vdash e_2 : \tau_2}{\Delta_1, \Delta_2 \Vdash \langle e_1, e_2 \rangle : \tau_1 \otimes \tau_2}\ \otimes I \qquad \frac{\Delta \Vdash e : \tau_1 \otimes \tau_2 \quad \Delta', x_1 : \tau_1, x_2 : \tau_2 \Vdash e' : \tau'}{\Delta, \Delta' \Vdash \mathsf{case}\ e\ (\langle x_1, x_2 \rangle \Rightarrow e') : \tau'}\ \otimes E$$

$$\frac{}{\cdot \Vdash \langle\,\rangle : 1}\ 1I \qquad \frac{\Delta \Vdash e : 1 \quad \Delta' \Vdash e' : \tau'}{\Delta, \Delta' \Vdash \mathsf{case}\ e\ (\langle\,\rangle \Rightarrow e') : \tau'}\ 1E$$

$$\frac{(j \in I) \quad \Delta \Vdash e : \tau_j}{\Delta \Vdash j \cdot e : \oplus_{i \in I}(i : \tau_i)}\ \oplus I$$

$$\frac{\Delta \Vdash e : \oplus_{i \in I}(i : \tau_i) \quad (\text{for all } i \in I) \quad \Delta', x_i : \tau_i \Vdash e_i' : \tau'}{\Delta, \Delta' \Vdash \mathsf{case}\ e\ (i \cdot x_i \Rightarrow e_i')_{i \in I} : \tau'}\ \oplus E$$

$$\frac{\Delta \Vdash e_i : \tau_i \quad (\text{for all } i \in I)}{\Delta \Vdash \langle\!|\, i \Rightarrow e_i \,|\!\rangle_{i \in I} : \&_{i \in I}(i : \tau_i)}\ \& I \qquad \frac{\Delta \Vdash e : \&_{i \in I}(i : \tau_i) \quad (j \in I)}{\Delta \Vdash e.j : \tau_j}\ \& E$$

$$\frac{\Delta \Vdash e : [\rho\alpha.\, \tau/\alpha]\tau}{\Delta \Vdash \mathsf{fold}\ e : \rho\alpha.\, \tau}\ \rho I \qquad \frac{\Delta \Vdash e : \rho\alpha.\, \tau}{\Delta \Vdash \mathsf{unfold}\ e : [\rho\alpha.\, \tau/\alpha]\tau}\ \rho E$$

5 Linear Typing of Processes

We didn't prove preservation and progress for linear types. While they are still satisfied, they are 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 [PP20]. Here, we just present purely linear typing.

Our judgment is

$$\Delta \Vdash P :: (z : \sigma)$$

where Δ contains linear variables. 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

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

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

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

The rule for variables: one that reads from an ephemeral (linear) cell and deallocates it.

$$\frac{}{y : \tau \Vdash x^W \leftarrow y^R :: (x : \tau)}\ \mathsf{move}$$

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

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

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.

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

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

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".

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

where

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

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.

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

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

where

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

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

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

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

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

$\mathsf{proc}\ d\ (\mathsf{case}\ d^W\ K) \mapsto \mathsf{cell}\ d\ K$ (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.

The summary of all the rules for linear processes can be found in the linear rule sheet.

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

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

Under this view, functions become like *constants* that are visible throughout the program, similarly to the specific treatment we have given fixed points.

6 Example: Bit Flipping Revisited

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

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

where the initial state of running the program contains

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

This is now entirely linearly typed, except for the references to *flip*.

7 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

$!\mathsf{cell}\ \mathit{flip}\ \mathit{flip}_K,$
$\mathsf{cell}\ c_4\ \langle\,\rangle, \mathsf{cell}\ c_3\ (\mathbf{e} \cdot c_4), \mathsf{cell}\ c_2\ (\mathsf{fold}\ c_3),$
$\mathsf{cell}\ c_1\ (\mathbf{b0} \cdot c_2), \mathsf{cell}\ c_0\ (\mathsf{fold}\ c_1),$
$\mathsf{proc}\ d_0\ (\mathit{flip}^R.\langle c_0, d_0\rangle)$

will end with a state

$!\mathsf{cell}\ \mathit{flip}\ \mathit{flip}_K,$
$\mathsf{cell}\ c_4\ \langle\,\rangle, \mathsf{cell}\ d_3\ (\mathbf{e} \cdot c_4), \mathsf{cell}\ d_2\ (\mathsf{fold}\ d_3),$
$\mathsf{cell}\ d_1\ (\mathbf{b1} \cdot d_2), \mathsf{cell}\ d_0\ (\mathsf{fold}\ d_1)$

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

$$[\![\mathit{flip}\ (\mathsf{fold}\ (\mathbf{b0} \cdot (\mathsf{fold}\ (\mathbf{e} \cdot \langle\,\rangle))))]\!]\, d_0$$

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:

$$\text{Configurations}\quad \mathcal{C} \quad ::= \quad \cdot \mid \mathcal{C}_1, \mathcal{C}_2 \mid \mathsf{proc}\ d\ P \mid \mathsf{cell}\ c\ W$$

where we omit the persistent cells for closed, top-level functions. We imagine they are defined in a global context. The linear typing judgment then has the form

$$\Delta \Vdash \mathcal{C} :: \Delta'$$

for a configuration that writes to Δ' and reads from Δ. As before, any addresses in Δ not read by a process in $\mathcal{C}$ are passed on to Δ' to be read by a

process further on the right.

$$\frac{}{\Delta \Vdash (\cdot) :: \Delta}\ \mathsf{tp/empty} \qquad \frac{\Delta \Vdash P :: (d : \tau)}{\Delta', \Delta \Vdash \mathsf{proc}\ d\ P :: (\Delta', d : \tau)}\ \mathsf{tp/proc}$$

$$\frac{\Delta \Vdash c^W.V :: (c : \tau)}{\Delta', \Delta \Vdash \mathsf{cell}\ c\ V :: (\Delta', c : \tau)}\ \mathsf{tp/cell/val} \qquad \frac{\Delta \Vdash \mathsf{case}\ c^W\ K :: (c : \tau)}{\Delta', \Delta \Vdash \mathsf{cell}\ c\ K :: (\Delta', c : \tau)}\ \mathsf{tp/cell/cont}$$

$$\frac{\Delta \Vdash \mathcal{C}_1 :: \Delta_1 \quad \Delta_1 \Vdash \mathcal{C}_2 :: \Delta_2}{\Delta \Vdash (\mathcal{C}_1, \mathcal{C}_2) :: \Delta_2}\ \mathsf{tp/join}$$

8 Progress and Preservation

Progress is essentially unchanged from before.

Theorem 1 *If $\cdot \Vdash \mathcal{C} :: \Delta$ 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 $\Delta \Vdash \mathcal{C} :: \Delta'$ and $\mathcal{C} \mapsto \mathcal{C}'$ then $\Delta \Vdash \mathcal{C} :: \Delta'$.*

Contrast this with the previous statement of preservation where the output context may grow when a new cell is allocated.

The form of the preservation theorem now means that if we start, for example, with $\cdot \Vdash \mathcal{C} :: (d_0 : 1)$ then any resulting final configuration $\mathcal{C} \mapsto^* \mathcal{F}$ still has the same type. Since there are only ephemeral cells in $\mathcal{F}$, it must be of the form $\mathcal{F}', \mathsf{cell}\ d_0\ W$ for some $\mathcal{F}'$ and W. Since $d_0 : 1$, it follows by inversion that $W = \langle\,\rangle$. Moreover, $\cdot \Vdash \mathcal{F}' :: (\cdot)$. Again by inversion we find $\mathcal{F}' = (\cdot)$, 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 Write a linear increment function on natural numbers in binary representation.

Exercise 2 Recall the definition of a purely positive type, updated to reflect the notation for linear types.

$$\tau^+ ::= 1 \mid \tau_1^+ \otimes \tau_2^+ \mid \oplus_{i \in I}(i : \tau_i^+) \mid \rho\alpha^+.\, \tau^+ \mid \alpha^+$$

Even in the purely linear language, it is possible to *copy* a value of purely linear type. Define a family of functions

$$\mathit{copy}_{\tau^+} : \tau^+ \multimap (\tau^+ \otimes \tau^+)$$

such that $\mathit{copy}_{\tau^+}\ v \mapsto^* \langle v, v \rangle$ for every $v : \tau^+$. You do not need to prove this property, just give the definitions of the *copy* functions. Your definitions may be mutually recursive.

Exercise 3 A type isomorphism is *linear* if the functions *Forth* and *Back* are both linear. For each of the following pairs of types provide linear functions witnessing an isomorphism if they exist, or indicate no linear isomorphism exists. You may assume all functions terminate and use either extensional or logical equality as the basis for your judgment.

1. $\tau \multimap (\sigma \multimap \rho)$ and $\sigma \multimap (\tau \multimap \rho)$
2. $\tau \multimap (\sigma \multimap \rho)$ and $(\tau \otimes \sigma) \multimap \rho$
3. $\tau \multimap (\sigma \otimes \rho)$ and $(\tau \multimap \sigma) \otimes (\tau \multimap \rho)$
4. $(\tau \oplus \sigma) \multimap \rho$ and $(\tau \multimap \rho) \otimes (\sigma \multimap \rho)$
5. $(1 \oplus 1) \multimap \tau$ and $\tau \otimes \tau$

Exercise 4 Write out the following theorems, updated to the purely linear language (where only recursively defined variables are nonlinear). We change neither the definition of value nor the rules for stepping from our previous language that does not employ linearity.

1. Canonical forms for types $\multimap$, $\otimes$, 1, $\oplus$, and ρ. No proofs are needed.
2. The substitution properties, in a form sufficient needed for preservation. No proofs are needed.
3. The preservation property for evaluation of closed linear expressions. Show the proof cases for linear functions.
4. The progress property for closed linear expressions. Show the proof cases for linear functions.

5. Where do these properties and their proofs differ when compared to our language that does not enforce linearity?

Exercise 5 Prove that $\Delta \Vdash e : \tau$ implies $\Delta \Vdash [\![e]\!]\, d :: (d : \tau)$. You only need to show the cases relevant for functions ($\lambda x.\, e$, $e_1\ e_2$ and variables x).

Exercise 6 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_K and *inc* so that the program representation as a configuration would be !cell *inc* inc_K.
3. Show the initial and final configuration of computation for incrementing the number 1 represented as $\mathsf{fold}\ (\mathbf{b1} \cdot (\mathsf{fold}\ (\mathbf{e} \cdot \langle\rangle)))$.

References

[Gir87] Jean-Yves Girard. Linear logic. *Theoretical Computer Science*, 50:1–102, 1987.

[PP20] Klaas Pruiksma and Frank Pfenning. Back to futures. *CoRR*, abs/2002.04607, February 2020.