Lecture Notes on Linear Typechecking

15-417/817: HOT Compilation
Frank Pfenning

Lecture 2
Thursday, January 16, 2025

1 The Significance of Linearity

We first explore the strictures of linear typing a bit. Recall that each variable x and y_i in the judgment stands for an address of a cell at runtime.

$$\underbrace{y_1 : B_1, \ldots, y_n : B_n}_{\text{read from}} \vdash P :: \underbrace{(x : A)}_{\text{write to}}$$

Linearity guarantees that each variable y_i will be used, and variable x will be written to, assuming that the P terminates. The state of the execution consists of multiple semantics object cell $c\ v$ (where all the addresses c and distinct and v is a small value) and some processes proc P representing running processes. We reach a final configuration if it consists entirely of memory cells. Recall the rules:

$$\begin{array}{rlcll}
 & \mathsf{proc}\ (\mathbf{write}\ c\ v) & \longrightarrow & \mathsf{cell}\ c\ v & \\
\mathsf{cell}\ c\ v, & \mathsf{proc}\ (\mathbf{read}\ c\ K) & \longrightarrow & \mathsf{proc}\ (v \triangleright K) & \\
 & \mathsf{proc}\ (\mathbf{cut}\ (x : A)\ P(x)\ Q(x)) & \longrightarrow & \mathsf{proc}\ P(a), \mathsf{proc}\ Q(a) & (a\ \text{fresh}) \\
\mathsf{cell}\ b\ v, & \mathsf{proc}\ (\mathbf{id}\ a\ b) & \longrightarrow & \mathsf{cell}\ a\ v & \\
 & \mathsf{proc}\ (\mathbf{call}\ p\ a\ \overline{b}) & \longrightarrow & \mathsf{proc}\ Q(a, \overline{b}) & (\mathbf{proc}\ p\, x\, \overline{y} = Q(x, \overline{y}) \in \Sigma
\end{array}$$

$$\begin{array}{rcl@{\quad}c@{\quad}l}
(\,) & \triangleright & (\,) \Rightarrow Q & = & Q \\
(a, b) & \triangleright & (x, y) \Rightarrow Q(x, y) & = & Q(a, b) \\
k(a) & \triangleright & \{\ell(y_\ell) \Rightarrow Q_\ell(y_\ell)\}_{\ell \in L} & = & Q_k(a)
\end{array}$$

A step in multiset rewriting matches part of the configuration (which is unordered) against the left-hand side of the rule and replaces it by the right-hand side. In the rules for **read** and **id** the cell on the left-hand side therefore disappears from the configuration, which corresponds to deallocation of the cell. Because of linear typing, this means that at the end of the computation only cells reachable from the initial destination of the parameterless procedure will remain allocated in the configuration. In other words, we do not need a traditional garbage collector.

An interesting observation is that because all types are positive, we can actually *program* procedures that will deallocate or duplicate values. In the live-coded file lex02.sax we program deallocation and duplication functions for some types.

2 Two Problems with Typing

Let's look at just two of the rules and examine issues with implementing them.

$$\frac{}{y : A \vdash \mathbf{id}\ x\ y :: (x : A)}\ \mathsf{id} \qquad \frac{\Gamma \vdash P :: (x : A) \quad \Delta, x : A \vdash Q :: \delta}{\Gamma, \Delta \vdash \mathbf{cut}\ (x : A)\ P\ Q :: \delta}\ \mathsf{cut}$$

The first problem is visible in the identity rule: we are given $y : B \vdash \mathbf{id}\ x\ y :: (x : A)$ and we should accept the identity as well-typed if $B = A$. But when are two types equal? In many of today's functional languages, recursive type definitions are *generative*, so a type name is only equal to itself. But here we have equirecursive definitions, so a type name is *equal* to its definiens. In the presence of recursion, this creates the problem of deciding the equality of two types.

The second problem is exemplified in the cut rule. Each variable is supposed to be used exactly once, but some variables will appear in P and some on Q, so how do we "split" the context we have in the conclusion?

We'll address the two issues in turn.

3 Additive Algorithmic Typing

Assuming the problem of type equality can be solved, here is an approach towards designing an algorithm for linear type-checking. Because we don't know how to split the context, we just pass it to both premises of a cut. For commands that check, we return the context that was actually used in its type-checking. Then we combine the two and fail if (for example) a variable is used on both sides. This new judgment is $\Gamma \vdash P :: (x : A)\ /\ \Xi$, where Γ, P, x, and A are inputs, and Ξ is an output if the typing succeeds. Ξ contains the variables that are actually used in P. In other words:

If $\Gamma \vdash P :: (x : A)\ /\ \Xi$ *then* $\Xi \vdash P :: (x : A)$.

Conversely:

If $\Xi \vdash P :: (x : A)$ *then* $\Gamma, \Xi \vdash P :: (x : A)\ /\ \Xi$ *for all* Γ *disjoint from* Ξ.

Then a first version of the cut rule then might be

$$\frac{\Gamma \vdash P :: (x : A)\ /\ \Xi_1 \quad \Gamma, x : A \vdash Q :: \delta\ /\ \Xi_2}{\Gamma \vdash \mathbf{cut}\ (x : A)\ P\ Q :: \delta\ /\ \Xi_1\ ;\ \Xi_2}\ \mathsf{cut}??$$

where $\Xi_1\ ;\ \Xi_2$ merges two distinct contexts and checks that they are disjoint (see the definition below). However, this is not quite right. We also need to make sure that x is really used in the typing of Q, and that is doesn't escape its scope. So we get

$$\frac{\Gamma \vdash P :: (x : A)\ /\ \Xi_1 \quad \Gamma, x : A \vdash Q :: \delta\ /\ \Xi_2}{\Gamma \vdash \mathbf{cut}\ (x : A)\ P\ Q :: \delta\ /\ \Xi_1\ ;\ (\Xi_2 \setminus x)}\ \mathsf{cut}$$

Here, $\Xi_2 \setminus x$ removes $(x : A)$ from Ξ and also verifies such an x is actually in Ξ_2. This means,

We define the merge operation $\Xi_1\ ;\ \Xi_2$:

$$\begin{array}{lcll} (\Xi_1, x : A)\ ;\ \Xi_2 & = & (\Xi_1\ ;\ \Xi_2), x : A & \text{provided } x \notin \Xi_2 \\ \Xi_1\ ;\ (\Xi_2, x : A) & = & (\Xi_1\ ;\ \Xi_2), x : A & \text{provided } x \notin \Xi_1 \\ (\cdot)\ ;\ (\cdot) & = & (\cdot) & \end{array}$$

In other cases, the merge fails. And the subtraction operation:

$$\begin{array}{lcl} (\Xi, x : A) \setminus x & = & \Xi \\ (\Xi, y : B) \setminus x & = & (\Xi \setminus x), y : B \quad \text{provided } y \neq x \end{array}$$

In the remaining case, subtraction fails.

For the remaining rules, we just have to keep the defining property in mind: the output context Ξ contains the variables actually used. One note: in the rule for $\oplus L$, the subtraction $\Xi_\ell \setminus y_\ell$ must yield the same Ξ in each branch.

$$\frac{y : A \in \Gamma}{\Gamma \vdash \mathbf{id}\ x\ y :: (x : A) \mathbin{/} (y : A)}\ \mathsf{id} \qquad \frac{\Gamma \vdash P :: (x : A) \mathbin{/} \Xi_1 \quad \Gamma, x : A \vdash Q :: \delta \mathbin{/} \Xi_2}{\Gamma \vdash \mathbf{cut}\ (x : A)\ P\ Q :: \delta \mathbin{/} \Xi_1 \ ; (\Xi_2 \setminus x)}\ \mathsf{cut}$$

$$\frac{}{\Gamma \vdash \mathbf{write}\ x\ (\,) :: (x : \mathbf{1}) \mathbin{/} (\cdot)}\ \mathbf{1}X \qquad \frac{x : \mathbf{1} \in \Gamma \quad \Gamma \vdash P :: \delta \mathbin{/} \Xi}{\Gamma \vdash \mathbf{read}\ x\ (\,) \Rightarrow Q :: \delta \mathbin{/} \Xi \ ; (x : \mathbf{1})}\ \mathbf{1}L$$

$$\frac{x : A \in \Gamma, y : B \in \Gamma}{\Gamma \vdash \mathbf{write}\ z\ (x, y) :: z : A \otimes B \mathbin{/} (x : A) \ ; (y : B)}\ \otimes X \qquad \frac{z : A \otimes B \in \Gamma \quad \Gamma, x : A, y : B \vdash Q :: \delta \mathbin{/} \Xi}{\Gamma \vdash \mathbf{read}\ z\ (x, y) \Rightarrow Q :: \delta \mathbin{/} (\Xi \setminus x \setminus y) \ ; z : A \otimes B}\ \otimes R$$

$$\frac{(k \in L) \quad y : A_k \in \Gamma}{\Gamma \vdash \mathbf{write}\ x\ k(y) :: (x : \oplus\{\ell : A_\ell\}_{\ell \in L}) \mathbin{/} (y : A_k)}\ \oplus X$$

$$\frac{(x : \oplus\{\ell : A_\ell\}_{\ell \in L} \in \Gamma \quad \Gamma, y_\ell : A_\ell \vdash Q_\ell :: \delta \mathbin{/} \Xi_\ell \quad \Xi = \Xi_\ell \setminus y_\ell) \quad (\forall \ell \in L)}{\Gamma \vdash \mathbf{read}\ x\ \{\ell(y_\ell) \Rightarrow Q_\ell\}_{\ell \in L} :: \delta \mathbin{/} \Xi \ ; x : \oplus\{\ell : A_\ell\}_{\ell \in L}}\ \oplus L$$

$$\frac{(\overline{y : B} \vdash p :: (x : A)) \in \Sigma \quad \Gamma \vdash (\overline{b/y}) :: (\overline{y : B}) \mathbin{/} \Xi}{\Gamma \vdash \mathbf{call}\ p\ a\ \overline{b} :: (a : A) \mathbin{/} \Xi}\ \mathsf{call}$$

$$\frac{\Gamma \vdash \eta_1 :: \Delta_1 \mathbin{/} \Xi_1 \quad \Gamma \vdash \eta_2 :: \Delta_2 \mathbin{/} \Xi_2}{\Gamma \vdash (\eta_1, \eta_2) :: (\Delta_1, \Delta_2) \mathbin{/} (\Xi_1 \ ; \Xi_2)} \qquad \frac{}{\Gamma \vdash (\cdot) :: (\cdot) \mathbin{/} (\cdot)} \qquad \frac{y : A \in \Gamma}{\Gamma \vdash (y/x) :: (x : A) \mathbin{/} (y : A)}$$

Figure 1: Additive Algorithmic Typing

The idea of *subtractive typing* is to change the judgment to

$$\Gamma_I \vdash P :: (x : A) \mathbin{/} \Gamma_O$$

there Γ_I is the usable input, and Γ_O are the antecedents **not** used in P. The rule of cut then becomse

$$\frac{\Gamma_I \vdash P :: (x : A) \mathbin{/} \Gamma_M \quad \Gamma_M, x : A \vdash Q :: \delta \mathbin{/} \Gamma_O \quad x \notin \Gamma_O}{\Gamma_I \vdash \mathbf{cut}\ (x : A)\ P\ Q :: \delta \mathbin{/} \Gamma_O}\ \mathsf{cut}$$

The correctness theorems for subtractive typing are more complicated, although it may provide better error messages in some cases because errors are generally detects as soon as they arise. In other words, by passing Γ to both premises of a cut, we lose some information present in the subtractive version.

If $\Gamma_I \vdash P :: (x : A) \mathbin{/} \Gamma_O$ *then* $\Gamma_I - \Gamma_O \vdash P :: (x : A)$

4 Type Equality

Consider the type definitions

$$\begin{array}{lcl}\textbf{type}\ \mathsf{nat} & = & \oplus\{\underline{\mathsf{zero}} : \mathbf{1}, \underline{\mathsf{succ}} : \mathsf{nat}\} \\ \textbf{type}\ \mathsf{unary} & = & \oplus\{\underline{\mathsf{zero}} : \mathbf{1}, \underline{\mathsf{succ}} : \mathsf{unary}\}\end{array}$$

Then nat and unary are inhabited by exactly the same large values: $\underline{\mathsf{zero}}(\,)$, $\underline{\mathsf{succ}}(\underline{\mathsf{zero}}(\,))$, etc. Therefore, these two types should be considered equal. Because of the recursion in their definition, we use a *coinductive* definition of this equality, allowing *infinite* derivations.

$$\frac{}{\mathbf{1} = \mathbf{1}}\ \mathbf{1}S \qquad \frac{A_1 = B_1 \quad A_2 = B_2}{A_1 \otimes A_2 = B_1 \otimes B_2}\ \otimes S$$

$$\frac{L = K \quad A_\ell = B_\ell \quad (\forall \ell \in L)}{\oplus\{\ell : A_\ell\}_{\ell \in L} = \oplus\{k : B_k\}_{k \in K}}\ \oplus S$$

There are no explicit rules for names, because we can implicitly replace them by their definiens. And this is where the problem of infinite derivations can arise. For example, we can construct:

$$\frac{\dfrac{}{\mathbf{1} = \mathbf{1}}\ \mathbf{1}S \qquad \dfrac{\dfrac{}{\mathbf{1} = \mathbf{1}}\ \mathbf{1}S \qquad \begin{array}{c}\vdots \\ \mathsf{nat} = \mathsf{unary}\end{array}}{\mathsf{nat} = \mathsf{unary}}\ \oplus S}{\mathsf{nat} = \mathsf{unary}}\ \oplus S$$

So how can we *decide* type equality, since we cannot produce infinite derivations? The point is to produce finite representations of them, namely circular derivations. If we have n types occurring in a program, there are at most n^2 equalities that may occur on any branch. Therefore, we must eventually either fail, or succeed in closing off the dereivation with an axiom like $\mathbf{1}S$ or a loop.

This interpretation of the rules says: no matter how long we try to find a counterexample, there is none.

5 Subtyping

Our notion of equirecursive definition suggests an immediate generalization. For example, if we define

$$\textbf{type}\ \mathsf{pos} = \oplus\{\underline{\mathsf{succ}} : \mathsf{nat}\}$$

then every (closed, large) value of type pos will also have type nat. Therefore, it would be perfectly okay to allow

$$\frac{}{y : \mathsf{pos} \vdash \mathbf{id}\ x\ y :: (x : \mathsf{nat})}\ \mathsf{id}$$

However, the other direction would not work: we cannot move an small value of type nat into a call that requires its content to be of type pos. The rules for subtyping are just a small variation of the rules for type equality.

$$\frac{}{\mathbf{1} \leq \mathbf{1}}\ \mathbf{1}S \qquad \frac{A_1 \leq B_1 \quad A_2 \leq B_2}{A_1 \otimes A_2 \leq B_1 \otimes B_2}\ \otimes S$$

$$\frac{L \subseteq K \quad A_\ell \leq B_\ell \quad (\forall \ell \in L)}{\oplus\{\ell : A_\ell\}_{\ell \in L} \leq \oplus\{k : B_k\}_{k \in K}}\ \oplus S$$

But remember that the rules are *coinductive*, allowing infinite derivations.

We then modify each of the initial typing rules to allow subtyping wherever we checked two types to be equal before.

$$\frac{A' \leq A}{y : A' \vdash \textbf{id}\ x\ y :: (x : A)}\ \mathsf{id} \qquad \frac{\Gamma \vdash P :: (x : A) \quad \Delta, x : A \vdash Q :: \delta}{\Gamma, \Delta \vdash \textbf{cut}\ (x : A)\ P\ Q :: \delta}\ \mathsf{cut}$$

$$\frac{}{\cdot \vdash \textbf{write}\ x\ (\,) :: (x : \mathbf{1})}\ \mathbf{1}X \qquad \frac{\Gamma \vdash P :: \delta}{\Gamma, x : \mathbf{1} \vdash \textbf{read}\ x\ (\,) \Rightarrow Q :: \delta}\ \mathbf{1}L$$

$$\frac{A' \leq A \quad B' \leq B}{x : A', y : B' \vdash \textbf{write}\ z\ (x, y) :: z : A \otimes B}\ \otimes X \qquad \frac{\Gamma, x : A, y : B \vdash Q :: \delta}{\Gamma, z : A \otimes B \vdash \textbf{read}\ z\ (x, y) \Rightarrow Q :: \delta}\ \otimes R$$

$$\frac{(k \in L) \quad A'_k \leq A_k}{y : A'_k \vdash \textbf{write}\ x\ k(y) :: (x : \oplus\{\ell : A_\ell\}_{\ell \in L})}\ \oplus X \qquad \frac{\Gamma, y : A_\ell \vdash Q_\ell :: \delta \quad (\forall \ell \in L)}{\Gamma, x : \oplus\{\ell : A_\ell\}_{\ell \in L} \vdash \textbf{read}\ x\ \{\ell(y_\ell) \Rightarrow Q_\ell\}_{\ell \in L} :: \delta}\ \oplus L$$

$$\frac{(\overline{y' : B'} \vdash p :: (x' : A')) \in \Sigma \quad A' \leq A \quad \Gamma \vdash (\overline{y/y'}) :: (\overline{y' : B'})}{\Gamma \vdash \textbf{call}\ p\ x\ \overline{y} :: (x : A)}\ \mathsf{call}$$

$$\frac{\Gamma_1 \vdash \sigma_1 :: \Delta_1 \quad \Gamma_2 \vdash \sigma_2 :: \Delta_2}{\Gamma_1\ ;\ \Gamma_2 \vdash (\sigma_1, \sigma_2) :: (\Delta_1, \Delta_2)} \qquad \frac{}{\cdot \vdash (\cdot) :: (\cdot)} \qquad \frac{A \leq A'}{x : A \vdash (x/x') :: (x' : A')}$$

We leave it to you as part of Lab 1 to combine, somehow, the power of subtyping with the technique of additive algorithmic typing (or some other form of algorithmic typing).