Lecture Notes on Closures

15-417/817: HOT Compilation
Frank Pfenning

Lecture 8
February 6, 2025

1 Introduction

In the last lecture we have seen negative types, inhabited by functions $A \to B$ and lazy records (or *objects*) $\&\{\ell : A_\ell\}_{\ell \in L}$. It was in a way straightforward to extend our dynamics and in fact in Sax there was a beautiful symmetry between the negative and positive types.

Unfortunately, values of negative type include *expressions*, as in $\lambda x.\, e$ or $\{\ell \Rightarrow e_\ell\}_{\ell \in L}$. The dynamics then substitutes into the expressions in ND, or the corresponding commands in Sax. While this is quite reasonable for an abstract specification, once we compile to lower-level languages, we have to find another way to implement these.

There seem to be two primary approaches. One called *defunctionalization* [Reynolds, 1972] is a whole program transformation the eliminates higher-order functions in favor of data types. This is used effectively in the MLton [Weeks, 2006] compiler for Standard ML, but requires also a global control-flow analysis to be efficient [Cejtin et al., 2000]. Other compilers support separate compilation and use a more local technique called *closure conversion* [Steele, 1978]. Both of these can be *typed* [Minamide et al., 1996] and thus fit within the general theme of this course using statically typed intermediate languages.

In this lecture we introduce the basic idea of closures, not yet stepping to full closure conversion.

2 Evaluation with Environments

Substitutions abound in our formulation of the dynamics, but let's focus on functions. Instead of $e(x)$ and $e(v_2)$ we write e and $[v_2/x]e$ to emphasize the substitution operation as a primitive.

$$\frac{}{\lambda x.\, e \hookrightarrow \lambda x.\, e} \qquad \frac{e_1 \hookrightarrow \lambda x.\, e \quad e_2 \hookrightarrow v_2 \quad [v_2/x]e \hookrightarrow v}{e_1\, e_2 \hookrightarrow v}$$

This semantics allows us to start with a closed expression

$$\cdot \vdash e : A$$

and evaluate it all the way to a (closed) values

$$e \hookrightarrow v \qquad \text{with} \qquad \cdot \vdash v : A$$

In order to avoid substitution into expressions, we could try to evaluate *open expression* in an *environment* that assigns values to its free variables. That is, assuming

$$\Gamma \vdash e : A \qquad \text{and} \qquad \eta : \Gamma$$

we evaluate

$$\eta \vdash e \hookrightarrow v$$

Here, $\eta : \Gamma$ means that η maps every variable $x : A$ in Γ to a (closed) values $v : A$. That is:

$$\frac{}{(\cdot) \vdash (\cdot)} \qquad \frac{\eta : \Gamma \quad \cdot \vdash v : A}{(\eta, x \mapsto v) : (\Gamma, x : A)}$$

One could also imagine that v is not closed in $\eta \vdash e \hookrightarrow v$, but this is not sustainable for several reasons. One is that the environments form a stack during evaluation, so the environment in effect while evaluating, say $\lambda x.\, e$ will cease to exist.

In order to solve this problem, we have to pair up the current environment with the value. That is, we have to *close* the expression. We also have to look up variables in the environment.

$$\frac{}{\eta \vdash \lambda x\, e \hookrightarrow \langle \eta, \lambda x.\, e \rangle} \qquad \frac{x \mapsto v \in \eta}{\eta \vdash x \hookrightarrow v}$$

Actually, it would be sufficient to restrict η to the free variables occurring in $\lambda x\, e$, something we'll take a look at later.

A closure such as $\langle \eta, \lambda x\, e \rangle$ is a *runtime artifact* and does not appear in the source. Here is then our new language of large values:

$$\begin{array}{llll}
\text{Large Values} & v & ::= & (v_1, v_2) & (A \otimes B) \\
 & & \mid & (\,) & (\mathbf{1}) \\
 & & \mid & k(v) & (\oplus\{\ell : A_\ell\}_{\ell \in L}) \\
 & & \mid & \langle \eta, \lambda x.\, e \rangle & (A \to B) \\
 & & \mid & \langle \eta, \{\ell \Rightarrow e_\ell\}_{\ell \in L} \rangle & (\&\{\ell : A_\ell\}_{\ell \in L})
\end{array}$$

These new values are typed with the $v : A$ judgment for closed values:

$$\frac{\eta : \Gamma \quad \Gamma \vdash e : A}{\langle \eta, e \rangle : A}$$

where e could be a λ-abstraction or a lazy record. Note that from the perspective of type-checking this is not great, since the context Γ appears in both premises but not the conclusion. Fortunately, they do not appear in the source, only at runtime, so we don't have to be concerned with actually checking their type. We'll reexamine this when we come to closure conversion.

Keeping all this in mind, we can now rewrite the rule for evaluation of function application.

$$\frac{\eta \vdash e_1 \hookrightarrow \langle \eta', \lambda x.\, e \rangle \quad \eta \vdash e_2 \hookrightarrow v_2 \quad \eta', x \mapsto v_2 \vdash e \hookrightarrow v}{\eta \vdash e_1\, e_2 \hookrightarrow v}$$

We see that we "resurrect" the environment η' from the closure, forgetting about the current environment η. This is forced even if we only consider typing, because the free variables of $\lambda x.\, e$ are bound in η', not in η. It is also clear that we need to add $x \mapsto v_2$, because x is free in the body e and should be have the value v_2.

Lazy records are actually a little bit more straightforward.

$$\frac{}{\eta \vdash \{\ell \Rightarrow e_\ell\}_{\ell \in L} \hookrightarrow \langle \eta, \{\ell \Rightarrow e_\ell\}_{\ell \in L}\rangle} \qquad \frac{\eta \vdash e \hookrightarrow \langle \eta', \{\ell \Rightarrow e_\ell\}_{\ell \in L}\rangle \quad (k \in L) \quad \eta' \vdash e_k \hookrightarrow v}{e.k \hookrightarrow v}$$

One fundamental change in this approach we have to get used to is that values are typed differently from expressions. Also, we should remember that the structure of closures as values of negative type or not directly observable.

Another rule where environments play a role is in called top-level functions F (that is metavariables). We need to evaluate the substitution σ that matches the context Δ in the definition of F, which will result in an environment. This will be installed for the evaluation of the body of the definition, supplanting the current environment η.

$$\frac{F[\Delta] = e \quad \eta \vdash \sigma \hookrightarrow \eta' \quad \eta' \vdash e \hookrightarrow v}{\eta \vdash F\,[\sigma] : A \hookrightarrow v}$$

$$\frac{\eta \vdash e \hookrightarrow v}{\eta \vdash (\sigma, x \mapsto e) \hookrightarrow (\eta', x \mapsto v)} \qquad \frac{}{\eta \vdash (\cdot) \hookrightarrow (\cdot)}$$

The pattern matching rules also change, but they do not need to build closures because the control flow and the scoping go hand in hand. For example:

$$\frac{\eta \vdash e_1 \hookrightarrow v_1 \quad \eta \vdash e_2 \hookrightarrow v_2}{\eta \vdash (e_1, e_2) \hookrightarrow (v_1, v_2)} \qquad \frac{\eta \vdash e \hookrightarrow (v, w) \quad \eta, x \mapsto v, y \mapsto w \vdash e' \hookrightarrow v}{\eta \vdash \textbf{match}\ e\ ((x, y) \Rightarrow e') \hookrightarrow v}$$

We don't bother writing out the remaining rules for unit and sums because they are straightforward.

3 Changes in Sax

Because closures are only a runtime artifact, the compilation $[\![e]\!]\, d$ does not need to change. But for Sax commands, the same problem arises as for ND expressions: we do not want to substitute into commands.

A major rewrite of the rules would maintain a global stack and commit to the sequential execution of programs. A less drastic change is to maintain environments locally, with each expression. We show the latter and reserve the former for a potential future lectures.

The commands now never contain addresses, only variables.

$$
\begin{array}{llcll}
\text{Commands} & P, Q & ::= & \textbf{write}\ x\ v & (A \otimes B, \mathbf{1}, \oplus\{\ell : A_\ell\}) \\
 & & \mid & \textbf{write}\ x\ K & (A \to B, \&\{\ell : A_\ell\}) \\
 & & \mid & \textbf{read}\ x\ K & (A \otimes B, \mathbf{1}, \oplus\{\ell : A_\ell\}) \\
 & & \mid & \textbf{read}\ x\ v & (A \to B, \&\{\ell : A_\ell\}) \\
 & & \mid & \textbf{cut}\ (x : A)\ P\ Q & \\
 & & \mid & \textbf{id}\ x\ y & \\
 & & \mid & \textbf{call}\ F\ x\ y_1\ \ldots\ y_n & \\
\text{Storables} & S & ::= & v \mid \langle \eta, K \rangle & \\
\text{Small values} & v & ::= & (a, b) & (\otimes, \to) \\
 & & \mid & (\,) & (\mathbf{1}, [\bot]) \\
 & & \mid & k(a) & (\oplus, \&) \\
\text{Continuations} & K & ::= & (x, y) \Rightarrow P & (\otimes, \to) \\
 & & \mid & (\,) \Rightarrow P & (\mathbf{1}, [\bot]) \\
 & & \mid & \{\ell(x_\ell) \Rightarrow P_\ell\}_{\ell \in L} & (\oplus, \&)
\end{array}
$$

The process objects now carry an environment (written $\mathsf{proc}\ \eta\ P$), where all the free variables in P are bound to addresses in η. Storables are still closed, where small values ((a, b), $(\,)$, $k(a)$) contain addresses, while for negative types, we use closures $\langle \eta, K \rangle$. We have to be careful to look up variables in the appropriate environment where they contain variables rather than addresses.

$$
\begin{array}{lcl}
\mathsf{proc}\ \eta\ (\textbf{write}\ x\ v) & \longrightarrow & \mathsf{cell}\ \eta(x)\ (\eta(v)) \\
\mathsf{proc}\ \eta\ (\textbf{write}\ x\ K) & \longrightarrow & \mathsf{cell}\ \eta(x)\ \langle \eta, K \rangle \\
\mathsf{cell}\ \eta(x)\ v, \mathsf{proc}\ \eta\ (\textbf{read}\ x\ K) & \longrightarrow & (\eta \vdash v \rhd K) \\
\mathsf{cell}\ \eta(x)\ \langle \eta', K \rangle, \mathsf{proc}\ \eta\ (\textbf{read}\ x\ v) & \longrightarrow & (\eta' \vdash \eta(v) \rhd K) \\
\mathsf{proc}\ \eta\ (\textbf{cut}\ (x : A)\ P(x)\ Q(x)) & \longrightarrow & \mathsf{proc}\ (\eta, x \mapsto a)\ P(x), \mathsf{proc}\ (\eta, x \mapsto a)\ Q(x) \quad (a\ \text{fresh}) \\
\mathsf{cell}\ \eta(y)\ S, \textbf{proc}\ \eta\ (\textbf{id}\ x\ y) & \longrightarrow & \mathsf{cell}\ \eta(x)\ S \\
\textbf{proc}\ \eta\ (\textbf{call}\ F\ x\ \overline{y}) & \longrightarrow & \mathsf{proc}\ (u \mapsto \eta(x), \overline{w \mapsto \eta(y)})\ P \quad \text{where } F(u, \overline{w}) = P
\end{array}
$$

and

$$
\begin{array}{lclclcll}
\eta & \vdash & (a, b) & \rhd & (x, y) \Rightarrow P(x, y) & = & \mathsf{proc}\ (\eta, x \mapsto a, y \mapsto b)\ P(x, y) & \\
\eta & \vdash & (\,) & \rhd & (\,) \Rightarrow P & = & \mathsf{proc}\ \eta\ P & \\
\eta & \vdash & k(a) & \rhd & \{\ell(x_\ell) \Rightarrow P_\ell(x_\ell)\}_{\ell \in L} & = & \mathsf{proc}\ (\eta, x_k \mapsto a)\ P_k(x_\ell) & (k \in L)
\end{array}
$$

References

Henry Cejtin, Suresh Jagannathan, and Stephen Weeks. Flow-directed closure conversion for typed languages. In Gert Smolka, editor, *9th European Symposium on Programming (ESOP 2000)*, pages 56–71, Berlin, Germany, March 2000. Springer LNCS 1782.

Yasuhiko Minamide, J. Gregory Morrisett, and Robert Harper. Typed closure conversion. In *23rd Symposium on Principles of Programming Languages (POPL 1996)*, pages 271–283. ACM, January 1996.

John C. Reynolds. Definitional interpreters for higher-order programming languages. In *Proceedings of the ACM Annual Conference*, pages 717–740, Boston, Massachusetts, August 1972. ACM Press. Reprinted in *Higher-Order and Symbolic Computation*, 11(4), pp.363–397, 1998.

Guy L. Steele, Jr. Rabbit: a compiler for Scheme. Masters thesis, MIT, 1978.

Stephen Weeks. Whole-program compilation in MLton. In Andrew Kennedy and Franҫis Pottier, editors, *Proceedings of the Workshop on ML*, Portland, Oregon, September 2006. ACM. Slides available at `http://www.mlton.org/References.attachments/060916-mlton.pdf`.