Supplementary Notes on Environments

15-312: Foundations of Programming Languages
Jonathan Aldrich
(portions due to Frank Pfenning and Daniel Spoonhower)

Lecture 9
September 23, 2003

In this lecture, we examine slightly lower-level issues regarding the implementation of functional languages.

This first observation about our semantic specifications is that most of them rely on *substitution* as a primitive operation. From the point of view of implementation, this is impractical, because a program would be copied many times. So we seek an alternative semantics in which substitutions are not carried out explicitly, but an association between variables and their values is maintained. Such a data structure is called an *environment*. Care has to be taken to ensure that the intended meaning of the program (as given by the specification with substitution) is not changed.

Because we are in a call-by-value language, environment η typically binds variables to values.

$$\text{Environments} \quad \eta \quad ::= \quad \cdot \mid \eta, x{=}v$$

The basic intuition regarding typing is that if $\Gamma \vdash e : \tau$, then e should be evaluated in an environment which supplies bindings of appropriate type for all the variables declared in Γ. We therefere formalize this as a judgment, writing $\eta : \Gamma$ if the bindings of variables to values in η match the context Γ. We make the general assumption that a variable x is bound only once in an environment, which corresponds to the assumption that a variable x is declared only once in a context. If necessary, we can rename bound variables in order to maintain this invariant.

$$\frac{}{\cdot : \cdot} \qquad \frac{\eta : \Gamma \quad \cdot \vdash v : \tau \quad v \ \mathsf{value}}{(\eta, x{=}v) : (\Gamma, x{:}\tau)}$$

Note that the values v bound in an environment are closed, that is, they contain no free variables. This means that expressions are evaluated in an environment, but the resulting values must be closed. This creates a difficulty when we come to the evaluation of function expressions. Relaxing this restriction, however, causes even more serious problems.[1]

In this lecture, we extend the C machine to keep track of the current environment mapping variables to values, resulting in the E machine. The states of the machine can be defined as follows:

States	s	$::=$	$k \mid \eta > e$	evaluate e under k with η
		$\mid$	$k \mid \eta < v$	return v to k with η
Stacks	k	$::=$	$\bullet$	empty stack
		$\mid$	$k \triangleright f$	stack k with top f
		$\mid$	$k \triangleright \eta$	stack k with environment η
Frames	f	$::=$	$\mathtt{o}(\square, e_2) \mid \mathtt{o}(v_1, \square)$	primops
		$\mid$	$\mathtt{pair}(\square, e_2) \mid \mathtt{pair}(v_1, \square)$	pairs
		$\mid$	$\mathtt{fst}(\square) \mid \mathtt{snd}(\square)$	projections
		$\mid$	$\mathtt{apply}(\square, e_2) \mid \mathtt{apply}(v_1, \square)$	applications
		$\mid$	$\mathtt{if}(\square, e_1, e_2)$	conditional

We have extended the states of the C machine to include the current environment η in addition to the stack k and expression e or value v. In addition to frames for evaluating inside function applications, pairs, etc., a frame can be used to store an environment that may need to be restored later.

We can proceed to give an operational semantics for the E machine. The rules are similar to those for the C machine, but instead of carrying out substitutions, we add bindings to the environment and look them up when necessary. We focus on pairs and (non-recursive) functions; the rules for integers and booleans are a straightforward extension of the rules in the C machine, and we will look at adding recursion later. The two function rules given below are actually incorrect, as we explain shortly.

[1]This is known in the Lisp community as the *upward funarg problem*.

$$
\begin{array}{lll}
k \mid \eta > \mathtt{pair}(e_1, e_2) & \mapsto_{\mathsf{e}} & k \triangleright \mathtt{pair}(\Box, e_2) \mid \eta > e_1 \\
k \triangleright \mathtt{pair}(\Box, e_2) \mid \eta < v_1 & \mapsto_{\mathsf{e}} & k \triangleright \mathtt{pair}(v_1, \Box) \mid \eta > e_2 \\
k \triangleright \mathtt{pair}(v_1, \Box) \mid \eta < v_2 & \mapsto_{\mathsf{e}} & k \mid \eta < \mathtt{pair}(v_1, v_2) \\
\\
k \mid \eta > \mathtt{fst}(e) & \mapsto_{\mathsf{e}} & k \triangleright \mathtt{fst}(\Box) \mid \eta > e \\
k \triangleright \mathtt{fst}(\Box) \mid \eta < \mathtt{pair}(v_1, v_2) & \mapsto_{\mathsf{e}} & k \mid \eta < v_1 \\
\\
k \mid \eta > \mathtt{snd}(e) & \mapsto_{\mathsf{e}} & k \triangleright \mathtt{snd}(\Box) \mid \eta > e \\
k \triangleright \mathtt{snd}(\Box) \mid \eta < \mathtt{pair}(v_1, v_2) & \mapsto_{\mathsf{e}} & k \mid \eta < v_2 \\
\\
k \mid \eta > x & \mapsto_{\mathsf{e}} & k \mid \eta < v \\
 & & \quad (\eta = (\eta_1, x = v, \eta_2)) \\
\\
k \mid \eta > \mathtt{apply}(e_1, e_2) & \mapsto_{\mathsf{e}} & k \triangleright \mathtt{apply}(\Box, e_2) \mid \eta > e_1 \\
k \triangleright \mathtt{apply}(\Box, e_2) \mid \eta < v_1 & \mapsto_{\mathsf{e}} & k \triangleright \mathtt{apply}(v_1, \Box) \mid \eta > e_2 \\
\\
k \mid \eta > \mathtt{fn}(\tau, x.e) & \mapsto_{\mathsf{e}} & k \mid \eta < \mathtt{fn}(\tau, x.e) \\
k \triangleright \mathtt{apply}(\mathtt{fn}(\tau, x.e), \Box) \mid \eta < v_2 & \mapsto_{\mathsf{e}} & k \triangleright \eta \mid \eta, x = v_2 > e \\
\\
k \triangleright \eta \mid \eta' < v & \mapsto_{\mathsf{e}} & k \mid \eta < v
\end{array}
$$

The rules for evaluating pairs states that the first and second arguments of the pair are evaluated in the same environment as the pair itself. Similar rules are defined for fst and snd. Since we do not substitute values for variables in expressions, we have an evaluation rule for variables. This rule simply looks up the value of the variable in the current environment and returns that value to the surrounding stack.

The search rules for function application are similar to the pair rules. A function evaluates to itself. The rule for function application stores the old environment to the stack, where it can be restored later after execution of the function. A new environment is built which is the old environment with an added binding of the variable x to the value v_2. When the function returns a value, the function's environment η' is discarded and the old environment η is restored for further evaluation of the body of the previous function on the stack.

Typing. In order to prove type soundness for the E machine, we need typing rules for machine configurations. We can define the well-formedness of machine states using the following rules:

$$\frac{\eta : \Gamma \quad \Gamma \vdash k : \tau\ \mathit{stack} \quad \Gamma \vdash e : \tau}{k \mid \eta > e\ \mathit{ok}}$$

$$\frac{\eta : \Gamma \quad \Gamma \vdash k : \tau\ \mathit{stack} \quad \cdot \vdash v : \tau \quad v\ \mathsf{value}}{k \mid \eta < v\ \mathit{ok}}$$

The first rule states that the environment η assigns values to variables according to the typing environment Γ. We use this Γ to check the type of the expression e, because e may have free variables that are bound to values in η. Finally, the stack k must be capable of accepting a value of type τ in the current hole $\Box$. The rule for returning a value is similar, except that the expression returned must be a value and it must be closed. We define stack typing with the following rules:

$$\frac{}{\Gamma \vdash \bullet : \tau\ \mathit{stack}}$$

$$\frac{\Gamma \vdash k : \tau'\ \mathit{stack} \quad \Gamma \vdash f : \tau\ \texttt{->}\ \tau'\ \mathit{frame}}{\Gamma \vdash k \triangleright f : \tau\ \mathit{stack}}$$

$$\frac{\eta' : \Gamma' \quad \Gamma' \vdash k : \tau\ \mathit{stack}}{\Gamma \vdash k \triangleright \eta' : \tau\ \mathit{stack}}$$

The rule for the empty stack states that it can take a value of any type in its hole. The rule for a stack with a frame on top states that if the rest of the stack can accept a value of type τ', then the frame had better accept a value of some type τ and pass it on to the rest of the stack as a value of type τ'. The overall stack type is then τ. Finally, the rule for an environment frame simply allows the remaining stack to be typed in an environment Γ' based on the bindings in η' rather than the original environment Γ. We define frame typing with rules like the following:

$$\frac{\Gamma \vdash e_2 : \tau_2}{\Gamma \vdash \texttt{apply}(\Box, e_2) : (\tau_2\ \texttt{->}\ \tau)\ \texttt{->}\ \tau\ \mathit{frame}}$$

$$\frac{\cdot \vdash v_1 : \tau_2\ \texttt{->}\ \tau}{\Gamma \vdash \texttt{apply}(v_1, \Box) : \tau_2\ \texttt{->}\ \tau\ \mathit{frame}}$$

For example, the first rule states that if expression e_2 has type τ_2, then the frame can accept a value of type τ_2 `->` τ and return a value of type τ to the enclosing frame. If we now try to prove type preservation in the following form

If s ok and $s \mapsto_{\mathsf{e}} s'$ then s' ok

we find that it is violated in the rule that transforms a function expression into a function value. The value $\mathtt{fn}(\tau, x.e)$ may have free variables referring to η, but the typing rule for returning a value states that values must be closed. If we relax this rule to allow values to refer to variables bound in the local environment, then the rule for returning a value to a previous stack frame will violate preservation. So the rules we have thus far are unsound.

Closures. In order to restore soundness, we need to pair up a value with its environment forming a *closure*. This means we have a new form of value, only used in the operational semantics, but not in the source expression.

$$\text{Expressions} \quad e \quad ::= \quad \ldots \mid \langle\!\langle \eta; \mathtt{fn}(\tau, x.e) \rangle\!\rangle$$

There are no evaluation rules for closures (they are values), and the typing rules have to "guess" an context that matches the environment. Note that we always type values in the empty environment.

$$\frac{}{\langle\!\langle \eta; \mathtt{fn}(\tau, x.e) \rangle\!\rangle \ \mathsf{value}} \qquad \frac{\eta : \Gamma \quad \Gamma \vdash \mathtt{fn}(\tau, x.e) : \tau}{\cdot \vdash \langle\!\langle \eta; \mathtt{fn}(\tau, x.e) \rangle\!\rangle : \tau}$$

Note that function expressions like $\mathtt{fn}(\tau, x.e)$ are no longer values–only function closures are values. We now modify the incorrect rules by building and destructing closures instead.

$$\begin{array}{lll} k \mid \eta > \mathtt{fn}(\tau, x.e) & \mapsto_{\mathsf{e}} & k \mid \eta < \langle\!\langle \eta'; \mathtt{fn}(\tau, x.e) \rangle\!\rangle \\ & & \quad (\eta' = \{y = \eta(y) \mid y \ in \ FV(x.e)\}) \\ k \triangleright \mathtt{apply}(\langle\!\langle \eta'; \mathtt{fn}(\tau, x.e) \rangle\!\rangle, \Box) \mid \eta < v_2 & \mapsto_{\mathsf{e}} & k \triangleright \eta \mid \eta', x = v_2 > e \end{array}$$

The first rule builds a closure which captures the bindings of the free variables of $x.e$ in the current environment. The second rule evaluates the function body with the environment stored in the closure (augmented with $x = v_2$) rather than with the calling environment.

Now it is easy to prove type preservation.

Theorem 1 (Type preservation in the E machine)
If s ok and $s \mapsto_{\mathsf{e}} s'$ then s' ok.

Proof: By induction on the derivation of $s \mapsto_{\mathsf{e}} s'$. ■

We can also state a progress theorem.

Theorem 2 (Progress in the E machine)
If $s\ ok$ then either $s = \bullet \mid \cdot < v$ and v value, or there exists s' such that $s \mapsto_{\mathsf{e}} s'$.

Proof: By induction on the derivation of $s\ ok$. ■

Recursion. Adding recursion to the E machine requires some care. The obvious rule to use would be the analog of the C machine rule:

$$k \mid \eta > \mathtt{rec}(\tau, x.e) \quad \mapsto_{\mathsf{e}} \quad k \rhd \eta \mid \eta, x = \mathtt{rec}(\tau, x.e) > e$$

This rule has a couple of problems, however. First, we are binding an expression, not a value, to x. This is a different kind of binding than we have been using before, and it makes sense to distinguish it from other bindings. For example, our existing rule for replacing variables with values won't work, because now we're going to potentially replace a variable with an expression that needs to be further evaluated.

The second problem is that the expression $\mathtt{rec}(\tau, x.e)$ is not closed; e could have free variables that are bound in η. This will create soundness problems in the same way that function values were unsound until we converted functions into closures. Thus, we need to use the same solution here: instead of binding x to the expression $\mathtt{rec}(\tau, x.e)$, we will bind it to a suspension that pairs the expression with its bound variables. The rules are as follows:

$$\begin{array}{lll} k \mid \eta > \mathtt{rec}(\tau, x.e) & \mapsto_{\mathsf{e}} & k \rhd \eta \mid \eta, x \overset{*}{=} \langle\!\langle \eta'; \mathtt{rec}(\tau, x.e)\rangle\!\rangle > e \\ & & \quad (\eta' = \{y \overset{[*]}{=} \eta(y) \mid y \ in\ FV(x.e)\}) \\ k \mid \eta > x & \mapsto_{\mathsf{e}} & k \rhd \eta \mid \eta' > e \\ & & \quad (\eta = (\eta_1, x \overset{*}{=} \langle\!\langle \eta'; e\rangle\!\rangle, \eta_2)) \end{array}$$

The suspended computation binding $x \overset{*}{=} \langle\!\langle \eta'; e\rangle\!\rangle$ is quite general. For example, if we add lazy expressions to the language, we can package up the lazy computation in a suspension that keeps track of its free variables. Later, when the program demands evaluation of the suspended lazy expression, we can extract the original environment from the suspension and proceed with evaluating the expression.

Closure conversion. There is a compile-time analogue to the closures that are generated in our operational semantics at run-time. This is the so-called *closure conversion*. To see the need for that, consider the simple program (shown in SML syntax)

```
let val x = 1
    val y = 2
in fn w => x + w + 1 end
```

How do we compile the function `fn w => x + w + 1`? The difficulty here is the reference to variable `x` defined in the ambient environment.

The solution is to close the code by abstracting over an environment, and pairing it up with the environment. This way we obtain

```
let val x = 1
    val y = 2
in (fn env => fn w => (#x env) + w + 1, {x = x}) end
```

If this transformation is carried out systematically, all functions are closed and can be compiled to a piece of each. Each of them expect and environment as an additional argument. This environment contains only the bindings of variables that actually occur free in the body of the function. An application of the function now also applies the function to the environment. For example,

```
let val x = 1
    val y = 2
    val f = fn w => x + w + 1
in f 3 end
```

is translated to

```
let val x = 1
    val y = 2
    val f = (fn env => fn w => (#x env) + w + 1,
             {x = x})
in (#1 f) (#2 f) 3  end
```

The problem with this transformation is that its target is generally not well typed. This is because functions with different sets of free variables will sometimes have different type. For example, the code

```
let val x = 1
 in if true then fn w => w + x
    else fn w => w + 2
end : int -> int
```

becomes

```
let val x = 1
 in if true
    then (fn env => fn w => w + (#x env), {x = x})
    else (fn env => fn w => w + 2, {})
end
```

which is not well-typed because the two branches of the conditional have different type: the first has type `({x:int} -> int -> int) * {x:int}` and the second has type `({} -> int -> int) * {}`.

There are a number of possible solutions to this problem. We could give up typing in the compiled code, but it turns out that preserving typing is very useful for checking that the compiler operates properly (as well as ensuring that the compiled program is well behaved!) We could use a list instead of a record to represent closures, so that the type of a closure does not include the list of free variables, but this is more inefficient for looking up bindings and interferes with garbage collection (more on this later). The best solution uses existential types, which will be described in a future lecture.