Supplementary Notes on Environments

15-312: Foundations of Programming Languages
Frank Pfenning

Lecture 26
December 3, 2002

In the final two lectures of this course we go into slightly lower-level issues regarding the implementation of functional languages. As we will see, related issues arise in object-oriented 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 always bind 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 order to concentrate on the essential issues with environments, we give here only a big-step operational semantics relating an expression to its final value. See [Ch. 11.2] for a version of the C-machine that maintains environments. We first give a big-step operational semantics using substitution. We concentrate on pairs and (non-recursive) functions; other constructs can be added but distract from the main issues.

$$\frac{e_1 \Downarrow v_1 \quad e_2 \Downarrow v_2}{\mathsf{pair}(e_1, e_2) \Downarrow \mathsf{pair}(v_1, v_2)}$$

$$\frac{e \Downarrow \mathsf{pair}(v_1, v_2)}{\mathsf{fst}(e) \Downarrow v_1} \qquad \frac{e \Downarrow \mathsf{pair}(v_1, v_2)}{\mathsf{snd}(e) \Downarrow v_2}$$

$$\frac{}{\lambda x.e \Downarrow \lambda x.e} \qquad \frac{e_1 \Downarrow \lambda x.e_3 \quad e_2 \Downarrow v_2 \quad \{v_2/x\}e_3 \Downarrow v}{\mathsf{apply}(e_1, e_2) \Downarrow v}$$

Next we try to add environments, being careful not to carry out substitutions, but just adding binding to the environments. The final two rules are actually incorrect, as we explain shortly.

$$\frac{\eta \vdash e_1 \Downarrow v_1 \quad \eta \vdash e_2 \Downarrow v_2}{\eta \vdash \mathsf{pair}(e_1, e_2) \Downarrow \mathsf{pair}(v_1, v_2)}$$

$$\frac{\eta \vdash e \Downarrow \mathsf{pair}(v_1, v_2)}{\eta \vdash \mathsf{fst}(e) \Downarrow v_1} \qquad \frac{\eta \vdash e \Downarrow \mathsf{pair}(v_1, v_2)}{\eta \vdash \mathsf{snd}(e) \Downarrow v_2}$$

$$\frac{}{\eta_1, x{=}v, \eta_2 \vdash x \Downarrow v}$$

$$\frac{}{\eta \vdash \lambda x.e \Downarrow \lambda x.e}\ ?? \qquad \frac{\eta \vdash e_1 \Downarrow \lambda x.e_3 \quad \eta \vdash e_2 \Downarrow v_2 \quad \eta, x{=}v_2 \vdash e_3 \Downarrow v}{\eta \vdash \mathsf{apply}(e_1, e_2) \Downarrow v}\ ??$$

If we now try to prove type preservation in the following form

If $\eta : \Gamma$ *and* $\Gamma \vdash e : \tau$ *and* $\eta \vdash e \Downarrow v$ *then* $\cdot \vdash v : \tau$

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

we find that it is violated in the rule for λ-abstraction, since the value $\lambda x.e$ may have free variables referring to η. If we try to fix this problem by proving instead

If $\eta : \Gamma$ *and* $\Gamma \vdash e : \tau$ *and* $\eta \vdash e \Downarrow v$ *then* $\Gamma \vdash v : \tau$

the rule for λ-abstraction now works correctly, but the rule for application has a problem. This is because we eventually obtain from the induction hypothesis and multiple steps of reasoning that $\Gamma, x{:}\tau_2 \vdash v : \tau$, but we need that $\Gamma \vdash v : \tau$.

So neither of the two ideas works, and type preservation would be violated. In order to restore is, we need to pair up a value with its environment forming a *closure*. There are many strategies to make this efficient. For example, we could restrict the environment to those variables occurring free in the value, but we do not consider such refinements here. This means we have a new form of value, only used in the operational semantics, but not in the source expression.

$$\text{Expressions} \quad e \quad ::= \quad \ldots \mid \langle\!\langle \eta; \lambda x.e \rangle\!\rangle$$

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.

$$\frac{}{\langle\!\langle \eta; \lambda x.e \rangle\!\rangle \ \mathsf{value}} \qquad \frac{\eta : \Gamma \quad \Gamma \vdash \lambda x.e : \tau}{\cdot \vdash \langle\!\langle \eta; \lambda x.e \rangle\!\rangle : \tau}$$

We now modify the incorrect rules by building and destructing closures instead.

$$\frac{}{\eta \vdash \lambda x.e \Downarrow \langle\!\langle \eta; \lambda x.e \rangle\!\rangle}$$

$$\frac{\eta \vdash e_1 \Downarrow \langle\!\langle \eta'; \lambda x.e_3 \rangle\!\rangle \quad \eta \vdash e_2 \Downarrow v_2 \quad \eta', x{=}v_2 \vdash e_3 \Downarrow v}{\eta \vdash \mathsf{apply}(e_1, e_2) \Downarrow v}$$

Now it is easy to prove by induction over the structure of the evaluation that type preservation holds in the following form.

Theorem 1 (Type preservation with environments)
Assume $\eta : \Gamma$ *and* $\Gamma \vdash e : \tau$. *If* $\eta \vdash e \Downarrow v$ *then* $\cdot \vdash v : \tau$.

Proof: By induction on the derivation of $\eta \vdash e \Downarrow v$, applying inversion on the typing derivation in each case. We show the three critical cases.

Case: $\eta_1, x{=}v, \eta_2 \vdash x \Downarrow v$.

$\Gamma \vdash x : \tau$	Given
$\Gamma = \Gamma_1, x{:}\tau, \Gamma_2$	By inversion
$\eta : \Gamma$	Given
$(\eta_1, x{=}v, \eta_2) : (\Gamma_1, x{:}\tau, \Gamma_2)$	Defns. of η and Γ
$\cdot \vdash v : \tau$	By inversion

Case: $\eta \vdash \lambda x.e \Downarrow \langle\!\langle \eta; \lambda x.e \rangle\!\rangle$.

$\Gamma \vdash \lambda x.e : \tau$	Given
$\eta : \Gamma$	Given
$\cdot \vdash \langle\!\langle \eta; \lambda x.e \rangle\!\rangle : \tau$	By rule

Case: $\eta \vdash \mathsf{apply}(e_1, e_2) \Downarrow v$.

$\eta \vdash e_1 \Downarrow \langle\!\langle \eta'; \lambda x.e_3 \rangle\!\rangle$	Subderivation
$\eta \vdash e_2 \Downarrow v_2$	Subderivation
$\eta', x{=}v_2 \vdash e_3 \Downarrow v$	Subderivation
$\Gamma \vdash \mathsf{apply}(e_1, e_2) : \tau$	Given
$\Gamma \vdash e_1 : \tau_2 \to \tau$ and	
$\Gamma \vdash e_2 : \tau_2$ for some τ_2	By inversion
$\eta : \Gamma$	Given
$\cdot \vdash \langle\!\langle \eta'; \lambda x.e_3 \rangle\!\rangle : \tau_2 \to \tau$	By i.h.
$\eta' : \Gamma'$ and	
$\Gamma' \vdash \lambda x.e_3 : \tau_2 \to \tau$ for some Γ'	By inversion
$\Gamma', x{:}\tau_2 \vdash e_3 : \tau$	By inversion
$\cdot \vdash v_2 : \tau_2$	By i.h.
$(\eta', x{=}v_2) : (\Gamma', x{:}\tau_2)$	By rule
$\cdot \vdash v : \tau$	By i.h.

■

A big-step semantics is unsuitable for proving a progress theorem, so we will not do so here (see [Ch. 11.2]).

Type preservation tells us that the environment semantics we gave is sensible, but actually want to know more, namely that is is in an appropriate sense equivalent to the substitution semantics we gave earlier. This is another instance of a bisimulation theorem. It will be an instance of *weak* bisimulation because we do not care about the intermediate states of evaluation. In other words, we can only observe the final value returned by a computation. Even this we have to refine, as discussed in lecture 22.

There are three steps in proving a bisimulation theorem that shows the observational equivalence of two forms of operational semantics

1. Define the bisimulation relation.
2. Show that it is an observational equivalence.
3. Prove that it is a bisimulation.

We now go through these steps on the substitution and environment semantics.

Defining the bisimulation. Intuitively, the bisimulation substitutes out the environment. The main complication is that it must do this recursively, because the values in an environment can again contain closures and environments. The bisimulation decomposes into two judgments: one to relate expressions in an environment to closure-free expression, and one to relate values to values.

$\eta \vdash e \Longleftrightarrow e'$ e in environment η is related to e'
$v \longleftrightarrow v'$ v is related to v'

In order to describe the typing properties of the translation, we need to generalize environment to contain bindings $x{=}x'$. Typing for environments is then generalized as follows

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

$$\frac{\Gamma' \vdash \eta : \Gamma \quad \Gamma' \vdash x' : \tau}{\Gamma' \vdash (\eta, x{=}x') : (\Gamma, x{:}\tau)}$$

The typings are presupposed to be related as follows: if $\eta \vdash e \Longleftrightarrow e'$ then $\Gamma' \vdash \eta : \Gamma$ and $\Gamma \vdash e : \tau$ and $\Gamma' \vdash e' : \tau$ for some Γ and Γ'.

$$\frac{(x{=}v \in \eta) \quad v \longleftrightarrow v'}{\eta \vdash x \Longleftrightarrow v'} \qquad \frac{(x{=}x' \in \eta)}{\eta \vdash x \Longleftrightarrow x'}$$

$$\frac{\eta \vdash e_1 \Longleftrightarrow e_1' \quad \eta \vdash e_2 \Longleftrightarrow e_2'}{\eta \vdash \mathsf{apply}(e_1, e_2) \Longleftrightarrow \mathsf{apply}(e_1', e_2')}$$

$$\frac{\eta, x{=}x' \vdash e \Longleftrightarrow e'}{\eta \vdash \lambda x.e \Longleftrightarrow \lambda x'.e'}$$

$$\frac{\eta' \vdash \lambda x.e \Longleftrightarrow v'}{\langle\langle \eta'; \lambda x.e \rangle\rangle \longleftrightarrow v'}$$

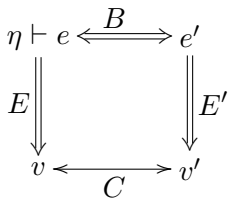
Observational Equivalence. Since we have simplified our language to just contain functions, the observational equivalance is trivialized. However, if we add, for example, integers and primitive operations, then we would have the rules

$$\frac{\eta \vdash e_1 \Longleftrightarrow e_1' \quad \eta \vdash e_2 \Longleftrightarrow e_2'}{\eta \vdash o(e_1, e_2) \Longleftrightarrow o(e_1', e_2')}$$

$$\frac{}{\mathsf{int}(n) \longleftrightarrow \mathsf{int}(n)}$$

and it is indeed the case that $\longleftrightarrow$ coincides with equality on the observable type int.

Proving the bisimulation. Bisimulation in this case can be discussed using the following diagram.



We need to show two properties:

1. If B and E are given, then v', E', and C exist, and
2. if B and E' are given, then v, E, and C exist.

Fortunately, neither of these direction is difficult. We do, however, need a substitution property.

Theorem 2 (Substitution for bisimulation)
If $v \longleftrightarrow v'$ *and* $\eta_1, x{=}x', \eta_2 \vdash e \Longleftrightarrow e'$ *then* $\eta_1, x{=}v, \eta_2 \vdash e \Longleftrightarrow \{v'/x'\}e'$.

Proof: By induction on the derivation of $\eta_1, x{=}x', \eta_2 \vdash e \Longleftrightarrow e'$. ■

For a more general language, we also need the easy property that $e \Downarrow e$ iff e value in the substitution semantics.

Theorem 3 (Simulation$_1$)
If $\eta \vdash e \Longleftrightarrow e'$ *and* $\eta \vdash e \Downarrow v$ *then there exists a* v' *such that* $e' \Downarrow v'$ *and* $v \longleftrightarrow v'$.

Proof: By induction on the derivation of $\eta \vdash e \Downarrow v$ and inversion on $\eta \vdash e \Longleftrightarrow e'$ in each case. ■

Theorem 4 (Simulation$_2$)
If $\eta \vdash e \Longleftrightarrow e'$ and $e' \Downarrow v'$ then there exists a v such that $\eta \vdash e \Downarrow v$ and $v \longleftrightarrow v'$.

Proof: By induction on the derivation of $e' \Downarrow v'$ and inversion on $\eta \vdash e \Longleftrightarrow e'$ in each case. ■

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) * {}`. We can repair the situation by using existential types. Since SML does not have first-class existential type, we just use the syntax `pack[t](e)` for $\mathsf{pack}(\tau, e)$. Then the example above can be written as

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

which now has type

$$\exists t.(t \to \mathsf{int} \to \mathsf{int}) \times t.$$

An application of a function before the translation is now translated to use `open`. For example,

```
let val x = 1
    val y = 2
    val f = pack [{x:int}]
            (fn env => fn w => (#x env) + w + 1,
             {x = x})
in open f as 'a => g => (#1 g) (#2 g) 3  end
```

where we wrote `open e1 as t => g => e2` for $\mathsf{open}(e_1, t.g.e_2)$.

The above discussion can be summarized by *closures have existential type*. When we apply this idea to objects in our encoding, we find that objects in a functional language are also become closures and therefore have existential type. Reconsider the simple example of a counter.

```
Counter = {get:1 -> int, inc:1 -> 1};
c : Counter =
    let x = ref 1
     in
        {get = λ_:1. !x,
         inc = λ_:1. x := !x+1}
    end;
```

After closure conversion, we obtain

```
c : Counter =
    let x = ref 1
     in
        pack [{x:int}]
        (λr.
         {get = λ_:1. !r.x,
         inc = λ_:1. r.x := !r.x+1},
         {x=x})
    end;
```

which has type

$$\exists t.(t \to \{\mathsf{get} : 1 \to \mathsf{int}, \mathsf{inc} : 1 \to 1) \times t$$

Here, the existential type hides the private fields of the record. This justifies the slogan that objects have existential type.