

Supplementary Notes on A Functional Language

15-312: Foundations of Programming Languages
Frank Pfenning

Lecture 5
September 10, 2002

In this lecture we first show the equivalence of the two styles of operational semantics: a *substitution semantics* and an *environment semantics*. We then proceed to extend our expression language to include booleans and functions.

We first recall the environment semantics, presented here as a particular form of evaluation semantics [Ch. 7.2]. The basic judgment is

$$x_1 \Downarrow v_1, \dots, x_n \Downarrow v_n \vdash e \Downarrow v.$$

Recall that this is a hypothetical judgment with assumptions $x_i \Downarrow v_i$. We call $x_1 \Downarrow v_1, \dots, x_n \Downarrow v_n$ an *environment* and denote an environment by η . It is important that all variables x_i in an environment are distinct so that the value of a variable is uniquely determined. Here we assume some primitive operators \circ (such as `plus` and `times`) and their mathematical counterparts f_\circ . For simplicity, we just write binary operators here.

$$\frac{x \Downarrow v \in \eta}{\eta \vdash x \Downarrow v} \text{ e.var} \qquad \frac{}{\eta \vdash \text{num}(k) \Downarrow \text{num}(k)} \text{ e.num}$$
$$\frac{\eta \vdash e_1 \Downarrow \text{num}(k_1) \quad \eta \vdash e_2 \Downarrow \text{num}(k_2) \quad (f_\circ(k_1, k_2) = k)}{\eta \vdash \circ(e_1, e_2) \Downarrow \text{num}(k)} \text{ e.o}$$
$$\frac{\eta \vdash e_1 \Downarrow v_1 \quad \eta, x \Downarrow v_1 \vdash e_2 \Downarrow v_2}{\eta \vdash \text{let}(e_1, x.e_2) \Downarrow v_2} \text{ e.let} \quad (x \text{ not declared in } \eta)$$

The alternative semantics uses substitution instead of environments. For this judgment we evaluate only closed terms, so no hypothetical judg-

ment is needed.

$$\begin{array}{c}
 \text{No rule for variables } x \quad \frac{}{\text{num}(k) \Downarrow \text{num}(k)} \text{ s.num} \\
 \\
 \frac{e_1 \Downarrow \text{num}(k_1) \quad e_2 \Downarrow \text{num}(k_2) \quad (f_o(k_1, k_2) = k)}{\text{o}(e_1, e_2) \Downarrow \text{num}(k)} \text{ s.o} \\
 \\
 \frac{e_1 \Downarrow v_1 \quad \{v_1/x\}e_2 \Downarrow v_2}{\text{let}(e_1, x.e_2) \Downarrow v_2} \text{ s.let}
 \end{array}$$

We show each direction of the translation between the two systems separately. In the first direction we assume $\cdot \vdash e \Downarrow v$ and we want to show $e \Downarrow v$. A direct proof by induction is suspect, because the environment will in general not be empty in the derivation of $\cdot \vdash e \Downarrow v$. In particular, the second premise of *e.let* adds a new assumption, which prevents us from using the induction hypothesis.

In order to generalize the induction hypothesis, we need to figure out what corresponds to $\eta \vdash e \Downarrow v$ in the substitution semantics. From the definition of the semantics we can see that an environment is a “postponed” substitution: rather than carrying out the substitution for each variable as we encounter it, we look up the variable at the end when we see it. Formalizing this intuition is the key to the proof. We define the translation from an environment to a simultaneous substitution [Ch. 5.3]

$$(x_1 \Downarrow v_1, \dots, x_n \Downarrow v_n)^* = (v_1/x_1, \dots, v_n/x_n)$$

Then we generalize to account for environments.

Lemma 1

If $\eta \vdash e \Downarrow v$ then $\{\eta^*\}e \Downarrow v$.

Proof: By rule induction on the given derivation. Recall that values v always have the form $\text{num}(k)$ for some k , so $v \Downarrow v$ for any value v by rule *s.num*.

Case: (Rule *e.var*) Then $e = x$.

$$\begin{array}{ll}
 x \Downarrow v \in \eta & \text{Condition of } e.\text{var} \\
 v/x \in \eta^* & \text{By definition of } \eta^* \\
 \{\eta^*\}x = v & \text{By definition of substitution} \\
 v \Downarrow v & \text{By definition of } v \text{ and rule } s.\text{num}
 \end{array}$$

Case: (Rule $e.num$) Then $e = \text{num}(k) = v$.

$\text{num}(k) \Downarrow \text{num}(k)$ By rule $s.num$

Case: (Rule $e.o$) Then $e = o(e_1, e_2)$.

$\eta \vdash e_1 \Downarrow \text{num}(k_1)$	Subderivation
$\eta \vdash e_2 \Downarrow \text{num}(k_2)$	Subderivation
$f_o(k_1, k_2) = k$	Given condition
$\{\eta^*\}e_1 \Downarrow \text{num}(k_1)$	By i.h.
$\{\eta^*\}e_2 \Downarrow \text{num}(k_2)$	By i.h.
$o(\{\eta^*\}e_1, \{\eta^*\}e_2) \Downarrow \text{num}(k)$	By rule $s.o$
$\{\eta^*\}o(e_1, e_2) \Downarrow \text{num}(k)$	By definition of substitution

Case: (Rule $e.let$) Then $e = \text{let}(e_1, x.e_2)$ and $v = v_2$.

$\eta \vdash e_1 \Downarrow v_1$	Subderivation
$\eta, x \Downarrow v_1 \vdash e_2 \Downarrow v_2$	Subderivation
$\{\eta^*\}e_1 \Downarrow v_1$	By i.h.
$(\eta, x \Downarrow v_1)^* = (\eta^*, v_1/x)$	By definition of $(\)^*$
$\{\eta^*, v_1/x\}e_2 \Downarrow v_2$	By i.h.
$\{v_1/x\}(\{\eta^*\}e_2) \Downarrow v_2$	By properties of simultaneous substitution
$\text{let}(\{\eta^*\}e_1, x.\{\eta^*\}e_2)$	By rule $s.let$
$\{\eta^*\}\text{let}(e_1, x.e_2)$	By definition of substitution

■

In the last case we need two properties that connects simultaneous substitution and the “single” substitution $\{v_1/s\}$. They are (a) that the order of the definition of variables in a simultaneous substitution does not matter, and (b) that

$$\{v_1/x_1\}(\{v_2/x_2, \dots, v_n/x_n\}e) = \{v_1/x_1, v_2/x_2, \dots, v_n/x_n\}e.$$

These properties hold under the assumption that all the x_i are distinct and that all v_1, v_2, \dots, v_n are closed, which is known in our case.

In lecture we proceeded slightly differently. Although the essential idea we were converging on was the same, we were getting to a lemma which asserted that $\eta \vdash e \Downarrow v$ then $\cdot \vdash \{\eta^*\}e \Downarrow v$ with a derivation of equal length. The above proof is somewhat more economical.

The other direction is quite a bit trickier to generalize correctly.

Lemma 2

If $e \Downarrow v$ and $e = \{\eta^*\}e'$ then $\eta \vdash e' \Downarrow v$.

Proof: The proof is by rule induction on the derivation of $e \Downarrow v$

Case: (Rule $s.num$) Then we have to consider two subcases, depending on whether $e' = x$ for some variable x , or $e' = \text{num}(k)$ for some k .

Subcase: (Rule $s.num$ and $e' = x$) Then $x \Downarrow v \in \eta$ in order for $e = \{\eta^*\}x \Downarrow v$ and hence $\eta \vdash x \Downarrow v$ by rule $e.var$.

Subcase: (Rule $s.num$ and $e' = \text{num}(k)$) In that case $v = \text{num}(k)$, so we can use rule $e.num$.

Case: (Rule $s.o$) Then $e = o(e_1, e_2) = \{\eta^*\}e'$.

$e' = o(e'_1, e'_2)$ with	
$e_1 = \{\eta^*\}e'_1$ and $e_2 = \{\eta^*\}e'_2$	By definition of substitution
$e_1 \Downarrow \text{num}(k_1)$	Subderivation
$e_2 \Downarrow \text{num}(k_2)$	Subderivation
$f_o(k_1, k_2) = k$	Given condition
$\eta \vdash e'_1 \Downarrow \text{num}(k_1)$	By i.h.
$\eta \vdash e'_2 \Downarrow \text{num}(k_2)$	By i.h.
$\eta \vdash o(e'_1, e'_2) \Downarrow k$	By rule $e.o$

Case: (Rule $s.let$) Then $e = \text{let}(e_1, x.e_2) = \{\eta^*\}e'$ and $v = v_2$.

$e' = \text{let}(e'_1, x.e'_2)$ with	
$e_1 = \{\eta^*\}e'_1$ and $e_2 = \{\eta^*\}e'_2$ and	
x not defined in η	By definition of substitution
$e_1 \Downarrow v_1$	Subderivation
$\eta \vdash e'_1 \Downarrow v_1$	By i.h.
$\{v_1/x\}e_2 \Downarrow v_2$	Subderivation
$\{v_1/x\}e_2 = \{v_1/x\}(\{\eta^*\}e'_2) = \{\eta^*, v_1/x\}e'_2$	Property of substitution
$\{(\eta, x \Downarrow v_1)^*\}e'_2 \Downarrow v_2$	By definition of $(\)^*$
$\eta, x \Downarrow v_1 \vdash e'_2 \Downarrow v_2$	By i.h.
$\eta \vdash \text{let}(e'_1, x.e'_2) \Downarrow v_2$	By rule $e.let$

■

Now we can prove our main theorem.

Theorem 3 (Equivalence of Environment and Substitution Semantics)

(i) If $\cdot \vdash e \Downarrow v$ then $e \Downarrow v$

(ii) If $e \Downarrow v$ then $\cdot \vdash e \Downarrow v$.

Proof: Part (i) follows immediately from the first lemma with $\eta = \cdot$, the empty environment.

Part (ii) follows from the second lemma by using the empty environment for η and e for e' , which is correct since $e = \{\cdot\}e$. ■

We now proceed with the introduction of MinML. The treatment here is somewhat cursory; see [Ch. 8] for additional material. Roughly speaking, MinML arises from the arithmetic expression language by adding booleans and recursive functions. These recursive functions are (almost) first-class in the sense that they can occur anywhere in an expression, rather than just at the top-level as in other languages such as C. This has profound consequences for the required implementation techniques (to which we will return later), but it does not affect typing in an essential way.

First, we give the grammar for the higher-order abstract syntax. For the concrete syntax, please refer to Assignment 2.

Types	$\tau ::= \text{int} \mid \text{bool} \mid \text{arrow}(\tau_1, \tau_2)$
Integers	$n ::= \dots \mid -1 \mid 0 \mid 1 \mid \dots$
Primops	$o ::= \text{plus} \mid \text{minus} \mid \text{times} \mid \text{negate}$ $\quad \mid \text{equals} \mid \text{lessthan}$
Expressions	$e ::= \text{num}(n) \mid o(e_1, \dots, e_n)$ $\quad \mid \text{true} \mid \text{false} \mid \text{if}(e, e_1, e_2)$ $\quad \mid \text{let}(e_1, x.e_2)$ $\quad \mid \text{fun}(\tau_1, \tau_2, f.x.e) \mid \text{apply}(e_1, e_2)$ $\quad \mid x$

Note that, unlike ML, the `fun`-expression binds both f (the function) and x (the argument). It does not define f in the rest of the program, only in the function body e in order to allow a recursive call. For example, the concrete syntax function

```
fun p(x:int):int is if x = 0
                    then 1
                    else 2 * p(x-1) fi end
```

is represented by

```
fun(int, int, p.x.if(equals(x, num(0))
                    num(1),
                    times(num(2), apply(p, minus(x, num(1)).))))
```

This is a naive implementation of $p(x) = 2^x$ for $x \geq 0$. If $x < 0$, it will simply not terminate.

Below are the typing rules for the language. We show only the case of one operator—the others are analogous.

$$\frac{x:\tau \in \Gamma}{\Gamma \vdash x : \tau} \text{VarTyp} \qquad \frac{}{\Gamma \vdash \text{num}(n) : \text{int}} \text{NumTyp}$$

$$\frac{\Gamma \vdash e_1 : \text{int} \quad \Gamma \vdash e_2 : \text{int}}{\Gamma \vdash \text{equals}(e_1, e_2) : \text{bool}} \text{EqualsTyp}$$

$$\frac{}{\Gamma \vdash \text{true} : \text{bool}} \text{TrueTyp} \qquad \frac{}{\Gamma \vdash \text{false} : \text{bool}} \text{FalseTyp}$$

$$\frac{\Gamma \vdash e : \text{bool} \quad \Gamma \vdash e_1 : \tau \quad \Gamma \vdash e_2 : \tau}{\Gamma \vdash \text{if}(e, e_1, e_2) : \tau} \text{IfTyp}$$

$$\frac{\Gamma \vdash e_1 : \tau_1 \quad \Gamma, x:\tau_1 \vdash e_2 : \tau_2}{\Gamma \vdash \text{let}(e_1, x.e_2) : \tau_2} \text{LetTyp}$$

$$\frac{\Gamma, f:\tau_1 \rightarrow \tau_2, x:\tau_1 \vdash e : \tau_2}{\Gamma \vdash \text{fun}(\tau_1, \tau_2, f.x.e) : \text{arrow}(\tau_1, \tau_2)} \text{FunTyp}$$

$$\frac{\Gamma \vdash e_1 : \text{arrow}(\tau_2, \tau) \quad \Gamma \vdash e_2 : \tau_2}{\Gamma \vdash \text{apply}(e_1, e_2) : \tau} \text{AppTyp}$$

We specify the operational semantics as a *structured operational semantics* also called a *small-step semantics*. The reason for this style of specification is that the evaluation semantics (also called big-step semantics) we used so far makes it difficult to talk about non-termination and the individual steps during evaluation, because it is slightly too abstract.

So we define two basic judgments

- (i) $e \mapsto e'$ which expresses that e steps to e' , and
- (ii) e value which expresses that e is a value (written v)

The idea is that, given a closed, well-typed expression e_1 , computation proceeds step-by-step until it reaches a value:

$$e_1 \mapsto e_2 \mapsto \dots \mapsto v$$

where v value. We will eventually prove the following three important properties, which guide us in the design of the rules

1. (Progress) If $\cdot \vdash e : \tau$ then either
 - (i) $e \mapsto e'$ for some e' , or
 - (ii) e value
2. (Preservation) If $\cdot \vdash e : \tau$ and $e \mapsto e'$ then $\cdot \vdash e' : \tau$
3. (Determinism) If $\cdot \vdash e : \tau$ and $e \mapsto e'$ and $e \mapsto e''$ then $e = e''$.

Note that for all three properties we are only interested in closed, well-typed expressions.

When presenting the operational semantics, we proceed type by type.

Integers This is straightforward. First, integers themselves are values.

$$\frac{}{\text{num}(k) \text{ value}}$$

Second, we evaluate the arguments to a primitive operation from left to right, and apply the operation once all arguments have been evaluated.

$$\frac{e_1 \mapsto e'_1}{\text{equals}(e_1, e_2) \mapsto \text{equals}(e'_1, e_2)} \quad \frac{v_1 \text{ value} \quad e_2 \mapsto e'_2}{\text{equals}(v_1, e_2) \mapsto \text{equals}(v_1, e'_2)}$$

$$\frac{(k_1 = k_2)}{\text{equals}(\text{num}(k_1), \text{num}(k_2)) \mapsto \text{true}}$$

$$\frac{(k_1 \neq k_2)}{\text{equals}(\text{num}(k_1), \text{num}(k_2)) \mapsto \text{false}}$$

We refer to the first two as *search rules*, since they traverse the expression to “search” for the subterm where the actual computation step takes place. The latter two are *reduction rules*.

Booleans First, `true` and `false` are values.

$$\overline{\text{true value}} \quad \overline{\text{false value}}$$

For if-then-else we have only one search rule for the condition, since we never evaluate in the branches before we know which one to take.

$$\frac{e \mapsto e'}{\text{if}(e, e_1, e_2) \mapsto \text{if}(e', e_1, e_2)}$$

$$\overline{\text{if}(\text{true}, e_1, e_2) \mapsto e_1} \quad \overline{\text{if}(\text{false}, e_1, e_2) \mapsto e_2}$$

Definitions We proceed as in the expression language with the substitution semantics. There are no new values, and only one search rule.

$$\frac{e_1 \mapsto e'_1}{\text{let}(e_1, x.e_2) \mapsto \text{let}(e'_1, x.e_2)}$$

$$\frac{v_1 \text{ value}}{\text{let}(v_1, x.e_2) \mapsto \{v_1/x\}e_2}$$

Functions It is often claimed that functions are “first-class”, but this is not quite true, since we cannot observe the structure of functions in the same way we can observe booleans or integers. Therefore, there is no need to evaluate the body of a function, and in fact we could not since it is not closed and we would get stuck when encountering the function parameter. So, any (recursive) function by itself is a value.

$$\overline{\text{fun}(\tau_1, \tau_2, f.x.e) \text{ value}}$$

Applications are evaluated from left-to-right, until both the function and its argument are values. This means the language is a *call-by-value* language with a *left-to-right* evaluation order.

$$\frac{e_1 \mapsto e'_1}{\text{apply}(e_1, e_2) \mapsto \text{apply}(e'_1, e_2)} \quad \frac{v_1 \text{ value} \quad e_2 \mapsto e'_2}{\text{apply}(v_1, e_2) \mapsto \text{apply}(v_1, e'_2)}$$

$$\frac{(v_1 = \text{fun}(\tau_1, \tau_2, f.x.e)) \quad v_2 \text{ value}}{\text{apply}(v_1, v_2) \mapsto \{v_1/f\}\{v_2/x\}e}$$