Lecture Notes on
A Functional Language

15-312: Foundations of Programming Languages
Frank Pfenning

Lecture 5
September 9, 2003

We now introduce MinML, a small fragment of ML that serves to illustrates key points in its design and key techniques for verifying its properties. The treatment here is somewhat cursory; see [Ch. 8] for additional material. Roughly speaking, MinML arises from the arithmetic expression language by adding booleans, functions, and recursion. 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.

$$
\begin{array}{llll}
\text{Types} & \tau & ::= & \texttt{int} \mid \texttt{bool} \mid \texttt{arrow}(\tau_1, \tau_2) \\
\text{Integers} & n & ::= & \ldots \mid -1 \mid 0 \mid 1 \mid \ldots \\
\text{Primops} & o & ::= & \texttt{plus} \mid \texttt{minus} \mid \texttt{times} \mid \texttt{negate} \\
 & & \mid & \texttt{equals} \mid \texttt{lessthan} \\
 & & & \\
\text{Expressions} & e & ::= & \texttt{num}(n) \mid o(e_1, \ldots, e_n) \\
 & & \mid & \texttt{true} \mid \texttt{false} \mid \texttt{if}(e, e_1, e_2) \\
 & & \mid & \texttt{let}(e_1, x.e_2) \\
 & & \mid & \texttt{fn}(\tau, x.e) \mid \texttt{apply}(e_1, e_2) \\
 & & \mid & \texttt{rec}(\tau, x.e) \\
 & & \mid & x
\end{array}
$$

Our typing judgment that sorts out the well-formed expressions has the form $\Gamma \vdash e : \tau$, where a context Γ has the form $\cdot, x_1{:}\tau_1, \ldots, x_n{:}\tau_n$. It is a hypothetical judgment as explained in the previous lecture. Our assumption that all variables x_i declared in a context must be distinct is still in force, which means that the rule

$$\frac{x{:}\tau \in \Gamma}{\Gamma \vdash x : \tau}\ \textit{VarTyp}$$

is unambiguous since there can be at most one declaration for x in Γ.

We have already discussed arithmetic expressions; booleans constitute a similar basic type. Unlike languages such as C, integers and booleans are strictly separate types, avoiding some common confusions and errors. Below are the typing rules related to booleans.

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

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

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

Perhaps the only noteworthy point here is that the two branches of a conditional must have the same type. This is because we cannot know at type-checking time which branch will be taken at run-time. We are therefore conservative, asserting only that the result of the conditional will definitely have type τ if each branch has type τ. Later in this class, we will see a type system that can be more accurately analyze conditionals so that, for example, `if true then 1 else false` could be given a type (which is impossible here).

A more important extension from our first language of arithmetic expressions is the addition of functions. In mathematics we are used to describe functions in the form $f(x) = e$, for example $f(x) = x^2 + 1$. In a functional language we want a notation for the function f itself. The abstract (mathematical) notation for this concept is λ-abstraction, written $f = \lambda x.e$. The above example would be written as $f = \lambda x.x^2 + 1$.

In the concrete syntax of MinML we express $\lambda x{:}\tau.e$ as `fn x:t => e`; in our abstract syntax it is written as $\texttt{fn}(\tau, x.e)$. This is an illustration of the unfortunate situation that we generally have to deal with at least three ways of expressing the same concepts. One is the mathematical notation, one is the concrete syntax, and one is the abstract syntax. In research papers, one mostly uses mathematical notation or pseudo-concrete syntax that really stands for abstract syntax but is easier to read. Inevitably, we will also start sliding between levels of discourse which is acceptable as long as we always know what we *really* mean.

Returning to functions, the typing rules are rather straightforward.

$$\frac{\Gamma, x{:}\tau_1 \vdash e : \tau_2}{\Gamma \vdash \mathtt{fn}(\tau, x.e) : \mathtt{arrow}(\tau_1, \tau_2)}\ \textit{FnTyp}$$

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

Keep in mind that in the rule *FnTyp*, the variable x must not already be declared in Γ. We can always rename x in $\mathtt{fn}(\tau, x.e)$ to satisfy this condition, because we treat abstract syntax as α-equivalence classes, that is, modulo variable renaming.

Functions defined with the language given so far are rather limited. For example, there is no way to define the exponential function from multiplication and addition, because there is no way to express recursion implicit in the definition

$$\begin{array}{lcll} 2^0 & = & 1 & \\ 2^n & = & 2 \times 2^{n-1} & \text{for } n > 0. \end{array}$$

In order to express this, we introduce a general recursion construct $\mathtt{rec}(\tau, x.e)$. The function above would be expressed as

$$\begin{array}{l} \mathtt{rec}(\mathtt{arrow}(\mathtt{int}, \mathtt{int}), p.\mathtt{fn}(\mathtt{int}, n. \\ \quad \mathtt{if}(\mathtt{equals}(n, \mathtt{num}(0)), \\ \qquad \mathtt{num}(1), \\ \qquad \mathtt{times}(\mathtt{num}(2), \mathtt{apply}(p, \mathtt{minus}(x, \mathtt{num}(1))))))). \end{array}$$

or in concrete syntax as

```
rec p:int -> int => fn n:int =>
  if n = 0
     then 1
     else 2 * p (x - 1)
```

In general, an expression $\mathtt{rec}(\tau, x.e)$ should be *unfolded* by substituting the whole expression for x in e, $\{\mathtt{rec}(\tau, x.e)/x\}e$. You should convince yourself on the example above that this yields the correct behavior—if you have difficulties, you may need to consult the formal operational semantics defined later in this lecture. As for the typing rule: the whole expression must have the same type as x, so that the substitution $\{\texttt{rec}(\tau, x.e)/x\}$ makes sense. The same type τ must also be the type of e, because the value of e is returned as the value of the recursive expression.

$$\frac{\Gamma, x{:}\tau \vdash e : \tau}{\Gamma \vdash \texttt{rec}(\tau, x.e) : \tau}\ \mathit{RecTyp}$$

In MinML, most useful recursions have the form

$$\texttt{rec}(\texttt{arrow}(\tau_1, \tau_2), f.\texttt{fn}(\tau_1, x.e)),$$

because most other recursive expressions will not terminate (try, for example, $\texttt{rec}(\texttt{int}, x.x)$). We therefore introduce a new form of concrete syntax, `fun f(x:`τ_1`):`τ_2 `=>` e, as "syntactic sugar". During parsing it is expanded into $\texttt{rec}(\texttt{arrow}(\tau_1, \tau_2), f.\texttt{fn}(\tau_1, x.e))$. This means that a `fun`-expression does not have first-class status. For example, we do not give any typing or evaluation rules since we type-check and evaluate the result of the syntactic expansion, not the original form.

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

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

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

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

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

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

$$\frac{\Gamma, x{:}\tau_1 \vdash e : \tau_2}{\Gamma \vdash \texttt{fn}(\tau, x.e) : \texttt{arrow}(\tau_1, \tau_2)}\ \textit{FnTyp}$$

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

$$\frac{\Gamma, x{:}\tau \vdash e : \tau}{\Gamma \vdash \texttt{rec}(\tau, x.e) : \tau}\ \textit{RecTyp}$$

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:

$$e_1 \mapsto e_2 \mapsto \cdots \mapsto v$$

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.

$$\frac{}{\texttt{num}(k)\ \mathsf{value}}$$

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

$$\frac{e_1 \mapsto e_1'}{\texttt{equals}(e_1, e_2) \mapsto \texttt{equals}(e_1', e_2)} \qquad \frac{v_1\ \mathsf{value} \quad e_2 \mapsto e_2'}{\texttt{equals}(v_1, e_2) \mapsto \texttt{equals}(v_1, e_2')}$$

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

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

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.

$$\frac{}{\texttt{true}\ \mathsf{value}} \qquad \frac{}{\texttt{false}\ \mathsf{value}}$$

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.

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

$$\frac{}{\texttt{if}(\texttt{true}, e_1, e_2) \mapsto e_1} \qquad \frac{}{\texttt{if}(\texttt{false}, e_1, e_2) \mapsto e_2}$$

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

$$\frac{e_1 \mapsto e_1'}{\texttt{let}(e_1, x.e_2) \mapsto \texttt{let}(e_1', x.e_2)}$$

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

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 function by itself is a value.

$$\frac{}{\texttt{fn}(\tau, x.e)\ \mathsf{value}}$$

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.

$$\frac{e_1 \mapsto e_1'}{\texttt{apply}(e_1, e_2) \mapsto \texttt{apply}(e_1', e_2)} \qquad \frac{v_1\ \mathsf{value} \quad e_2 \mapsto e_2'}{\texttt{apply}(v_1, e_2) \mapsto \texttt{apply}(v_1, e_2')}$$

$$\frac{v_2\ \mathsf{value}}{\texttt{apply}(\texttt{fn}(\tau_2, x.e), v_2) \mapsto \{v_2/x\}e}$$

Recursion A recursive expression is evaluated simply by unfolding it.

$$\frac{}{\texttt{rec}(\tau, x.e) \mapsto \{\texttt{rec}(\tau, x.e)/x\}e}$$

A recursive expression is never a value, but in a typical use of the form

$$\texttt{rec}(\texttt{arrow}(\tau_1, \tau_2), f.\texttt{fn}(\tau_1, x.e))$$

we can make only one step before reaching a value, because unfolding the `rec` exposes an `fn`-abstraction which is always a value. In [Ch. 8], the recursive expression $\texttt{fun}(\tau_1, \tau_2, f.x.e)$ which corresponds to the above is directly a value. This is appropriate in the case of MinML, but would lead to difficulties in a more general setting later in the course where we study recursively defined lists, trees, and other data structures.

As an alternative to the above semantics, let us also consider a judgment that directly relates an expression to its value (if it has one). We use substitution instead of environments for simplicity, so the judgment has the form $e \Downarrow v$ where we assume that $\cdot \vdash e : \tau$.

Integers This is quite simple and as for arithmetic expressions; we only show the rules for the primitive equality operator.

$$\frac{}{\mathtt{num}(k) \Downarrow \mathtt{num}(k)}$$

$$\frac{e_1 \Downarrow \mathtt{num}(k_1) \quad e_2 \Downarrow \mathtt{num}(k_2) \quad (k_1 = k_2)}{\mathtt{equals}(e_1, e_2) \Downarrow \mathtt{true}}$$

$$\frac{e_1 \Downarrow \mathtt{num}(k_1) \quad e_2 \Downarrow \mathtt{num}(k_2) \quad (k_1 \neq k_2)}{\mathtt{equals}(e_1, e_2) \Downarrow \mathtt{false}}$$

Booleans Here, the decision on which branch of a conditional to evaluate is based on the return value of the condition.

$$\frac{}{\mathtt{true} \Downarrow \mathtt{true}} \qquad \frac{}{\mathtt{false} \Downarrow \mathtt{false}}$$

$$\frac{e \Downarrow \mathtt{true} \quad e_1 \Downarrow v_1}{\mathtt{if}(e, e_1, e_2) \Downarrow v_1} \qquad \frac{e \Downarrow \mathtt{false} \quad e_2 \Downarrow v_2}{\mathtt{if}(e, e_1, e_2) \Downarrow v_2}$$

Definitions This remains unchanged from the arithmetic expression language.

$$\frac{e_1 \Downarrow v_1 \quad \{v_1/x\}e_2 \Downarrow v_2}{\mathtt{let}(e_1, x.e_2) \Downarrow v_2}$$

Functions Here we need to remember that (a) `fn`-expressions are values and (b) our language is call-by-value. The left-to-right strategy of evaluation is not directly visible in this formulation.

$$\frac{}{\mathtt{fn}(\tau, x.e) \Downarrow \mathtt{fn}(\tau, x.e)}$$

$$\frac{e_1 \Downarrow \mathtt{fn}(\tau_2, x.e_1') \quad e_2 \Downarrow v_2 \quad \{v_2/x\}e_1' \Downarrow v}{\mathtt{apply}(e_1, e_2) \Downarrow v}$$

Recursion Finally, for recursion, we just unfold the recursion one step and continue with evaluation.

$$\frac{\{\mathtt{rec}(\tau, x.e)/x\}e \Downarrow v}{\mathtt{rec}(\tau, x.e) \Downarrow v}$$

Which of the theorems regarding an operational semantics still make sense in this setting? First, progress is difficult to formulation because we either have a derivation of $e \Downarrow v$ or we do not—it is difficult to say what would constitute a step of evaluation. However, we can state that evaluation should always result in a value. Preservation and determinism still make sense in the following form. We should also now

1. (Evaluation) If $\cdot \vdash e : \tau$ and $e \Downarrow v$ then v value.
2. (Preservation) If $\cdot \vdash e : \tau$ and $e \Downarrow v$, then $\cdot \vdash v : \tau$
3. (Determinism) If $\cdot \vdash e : \tau$ and $e \Downarrow v'$ and $e \Downarrow v''$ then $v' = v''$.

Lecture 4 Addendum: Equivalence of Substitution and Environment Semantics

This material was covered in Fall'02. As a continuation of Lecture 4 we showed the equivalence of the *substitution semantics* and *environment semantics* for the arithmetic expression language. This is an instructive example of the kind of proof we are doing in this class.

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

$$x_1 {\Downarrow} v_1, \ldots, x_n {\Downarrow} v_n \vdash e \Downarrow v.$$

Recall that this is a hypothetical judgment with assumptions $x_i {\Downarrow} v_i$. We call $x_1 {\Downarrow} v_1, \ldots, x_n {\Downarrow} v_n$ 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 `o` (such as `plus` and `times`) and their mathematical counterparts f_o. For simplicity, we just write binary operators here.

$$\frac{x {\Downarrow} v \in \eta}{\eta \vdash x \Downarrow v}\; e.var \qquad \frac{}{\eta \vdash \mathtt{num}(k) \Downarrow \mathtt{num}(k)}\; e.num$$

$$\frac{\eta \vdash e_1 \Downarrow \mathtt{num}(k_1) \quad \eta \vdash e_2 \Downarrow \mathtt{num}(k_2) \quad (f_o(k_1, k_2) = k)}{\eta \vdash \mathtt{o}(e_1, e_2) \Downarrow \mathtt{num}(k)}\; e.o$$

$$\frac{\eta \vdash e_1 \Downarrow v_1 \quad \eta, x {\Downarrow} v_1 \vdash e_2 \Downarrow v_2}{\eta \vdash \mathtt{let}(e_1, x.e_2) \Downarrow v_2}\; e.let \quad (x \text{ not declared in } \eta)$$

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

$$\textit{No rule for variables } x \qquad \frac{}{\mathtt{num}(k) \Downarrow \mathtt{num}(k)}\ s.num$$

$$\frac{e_1 \Downarrow \mathtt{num}(k_1) \quad e_2 \Downarrow \mathtt{num}(k_2) \quad (f_o(k_1, k_2) = k)}{\mathtt{o}(e_1, e_2) \Downarrow \mathtt{num}(k)}\ s.o$$

$$\frac{e_1 \Downarrow v_1 \quad \{v_1/x\}e_2 \Downarrow v_2}{\mathtt{let}(e_1, x.e_2) \Downarrow v_2}\ s.let$$

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]

$$(x_1{\Downarrow}v_1, \ldots, x_n{\Downarrow}v_n)^* = (v_1/x_1, \ldots, v_n/x_n)$$

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 $\mathtt{num}(k)$ for some k, so $v \Downarrow v$ for any value v by rule $s.num$.

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

$x{\Downarrow}v \in \eta$	Condition of $e.var$
$v/x \in \eta^*$	By definition of η^*
$\{\eta^*\}x = v$	By definition of substitution
$v \Downarrow v$	By definition of v and rule $s.num$

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

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

Case: (Rule $e.o$) Then $e = \mathtt{o}(e_1, e_2)$.

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

Case: (Rule $e.let$) Then $e = \mathtt{let}(e_1, x.e_2)$ 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
$\mathtt{let}(\{\eta^*\}e_1, x.\{\eta^*\}e_2)$	By rule $s.let$
$\{\eta^*\}\mathtt{let}(e_1, x.e_2)$	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

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

These properties hold under the assumption that all the x_i are distinct and that all $v_1, v_2, \ldots, v_n$ 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 tricker 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 $e' = \mathtt{num}(k)$ 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** $e' = \mathbf{num}(k)$**)** In that case $v = \mathtt{num}(k)$, so we can use rule $e.num$.

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

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

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

$e' = \mathtt{let}(e_1', x.e_2')$ 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.
$\eta \vdash \mathtt{let}(e_1', x.e_2') \Downarrow v_2$	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 $e = \{\cdot\}e$. ■