19
Semantics: The environment model

The addition of mutability – which enables impure programming paradigms like imperative and procedural programming, with its potential for efficiencies in both time and space, and enables lazy and object-oriented programming as well – comes at a cost. Leibniz's law no longer applies. One and the same expression in the same context can evaluate to different values, making reasoning about programs more difficult.

That complexity ramifies in providing explicit semantics for the language as well. The simple substitution semantics of Chapter 13 is no longer sufficient. For that reason, and looking forward to the implementation of an interpreter for a larger fragment of OCaml (Chapter A), we revisit the formal substitution semantics from Chapter 13, modifying and augmenting it to provide a rigorous semantics for references and assignment, showing where the additional complexity arises and clarifying issues such as scope, side effects, and order of evaluation.

19.1 Review of substitution semantics

Recall from Section 13.6 the abstract syntax of a simple functional language with arithmetic:

⟨*binop*⟩ ::= `+` | `-` | `*` | `/`
⟨*var*⟩ ::= `x` | `y` | `z` | ⋯
⟨*expr*⟩ ::= ⟨*integer*⟩
| ⟨*var*⟩
| ⟨$expr_1$⟩ ⟨*binop*⟩ ⟨$expr_2$⟩
| `let` ⟨*var*⟩ `=` ⟨$expr_{def}$⟩ `in` ⟨$expr_{body}$⟩
| `fun` ⟨*var*⟩ `->` ⟨$expr_{body}$⟩
| ⟨$expr_{fun}$⟩ ⟨$expr_{arg}$⟩

The semantics for this language was provided through the apparatus of evaluation rules, which defined derivations for judgements of the form

$$P \Downarrow v$$

where P is an expression and v is its value (a simplified expression that means the same and that cannot be further evaluated).

The substitution semantics is sufficient for this simple language because it is a pure functional programming language. But binding constructs like `let`, `let rec`, and `fun` are awkward to implement, and extending the language to handle references, mutability, and imperative programming is quite challenging if not impossible. For that reason, we start by modifying the substitution semantics to make use of an ENVIRONMENT that stores a mapping from variables to their values. In the next two sections, we develop the environment semantics for the language of Chapter 13 in two variants: a dynamic environment semantics and a lexical environment semantics. We then augment the environment semantics with a model of a mutable store to allow for reference values and their assignment.

19.2 Environment semantics

In an environment semantics, instead of substituting for variables the value that they specify, we directly model a mapping between variables and their values, which we call an ENVIRONMENT. We use the following notation for mappings in the semantics. A mapping from elements, say, x, y, z, to elements a, b, c, respectively, will be notated as $\{x \mapsto a; y \mapsto b; z \mapsto c\}$. The notation purposefully evokes the OCaml record notation, since a record also provides a kind of mapping from a finite set of elements (labels) to associated values. It also evokes, through the use of the $\mapsto$ symbol, the idea of substitution, as these mappings will replace substitutions in the environment semantics.

Indeed, the environments that give their name to environment semantics are just such mappings – from variables to their values. We'll conventionally use the symbol E and its primed versions (E', E'', ...) as metavariables standing for environments. The empty environment will be notated $\{\}$, and the environment E augmented so as to add the mapping of the variable x to the value v will be notated $E\{x \mapsto v\}$. To look up what value an environment E maps a variable x to, we use Euler's function application notation: $E(x)$.

Having introduced the necessary notation, we turn to modifying the substitution semantics to use environments instead.

19.2.1 Dynamic environment semantics

Recall that the substitution semantics is given through a series of rules defining judgements of how expressions evaluate to values. (Reviewing Figure 13.5 may refresh your memory.)

In an environment semantics, expressions aren't evaluated in isolation. Rather, they are evaluated in the context of an environment that specifies which variables have which values. Instead of defining rules for P evaluating to v (written as the judgement $P \Downarrow v$), we define rules for P evaluating to v in an environment E (written as the judgement $E \vdash P \Downarrow v$). The rule for evaluating numbers, for instance, becomes

$$E \vdash \overline{n} \Downarrow \overline{n} \qquad (R_{int})$$

stating that "in environment E a numeral $\overline{n}$ evaluates to itself (independent of the environment)", and the rule for addition provides the environment as context for evaluating the subexpressions:

$$\begin{array}{l} E \vdash P \texttt{ + } Q \Downarrow \\ \quad \left| \begin{array}{l} E \vdash P \Downarrow \overline{m} \\ E \vdash Q \Downarrow \overline{n} \end{array} \right. \\ \quad \Downarrow \overline{m+n} \end{array} \qquad (R_{+})$$

Glossing again, the rule says "to evaluate an expression of the form P `+` Q in an environment E, first evaluate P in the environment E to an integer value $\overline{m}$ and Q in the environment E to an integer value $\overline{n}$. The value of the full expression is then the integer literal representing the sum of m and n."

To construct a derivation for a whole expression using these rules, we start in the empty environment $\{\}$. For instance, a derivation for the expression `3 + 5` would be

$$\begin{array}{l} \{\} \vdash \texttt{3 + 5} \Downarrow \\ \quad \left| \begin{array}{l} \{\} \vdash \texttt{3} \Downarrow \texttt{3} \\ \{\} \vdash \texttt{5} \Downarrow \texttt{5} \end{array} \right. \\ \quad \Downarrow \texttt{8} \end{array}$$

So far, not much is different from the substitution semantics. The differences show up in the handling of binding constructs like `let`. Recall the R_{let} rule for `let` binding in the substitution semantics.

$$\begin{array}{l} \texttt{let } x \texttt{ = } D \texttt{ in } B \Downarrow \\ \quad \left| \begin{array}{l} D \Downarrow v_D \\ B[x \mapsto v_D] \Downarrow v_B \end{array} \right. \\ \quad \Downarrow v_B \end{array} \qquad (R_{let})$$

This rule specifies that an expression of the form `let` x `=` D `in` B evaluates to the value v_B, whenever the definition expression D evaluates to v_D and the body expression B *after substituting* v_B *for the variable* x evaluates to v_B.

The corresponding environment semantics rule doesn't substitute into B. It evaluates B directly, but it does so in an environment augmented with a new binding of x to its value v_D:

$$E \vdash \texttt{let } x \texttt{ = } D \texttt{ in } B \Downarrow \quad \begin{array}{|l} E \vdash D \Downarrow v_D \\ E\{x \mapsto v_D\} \vdash B \Downarrow v_B \end{array} \quad \Downarrow v_B \qquad (R_{let})$$

According to this rule, "to evaluate an expression of the form `let` x `=` D `in` B in an environment E, first evaluate D in E resulting in a value v_D and then evaluate the body B in an environment that is like E except that the variable x is mapped to the value v_D. The result of this latter evaluation, v_B, is the value of the `let` expression as a whole."

In the substitution semantics, we will have substituted away all of the bound variables in a closed expression, so no rule is needed for evaluating variables themselves. But in the environment semantics, since no substitution occurs, we'll need to be able to evaluate expressions that are just variables. Presumably, those variables will have values in the prevailing environment; we'll just look them up.

$$E \vdash x \Downarrow E(x) \qquad (R_{var})$$

A gloss for this rule is "evaluating a variable x in an environment E yields the value of x in E."

Putting all these rules together, we can derive a value for the expression `let x = 3 in x + x`:

$$\{\} \vdash \texttt{let x = 3 in x + x} \Downarrow \quad \begin{array}{|l} \{\} \vdash \texttt{3} \Downarrow \texttt{3} \\ \{\texttt{x} \mapsto \texttt{3}\} \vdash \texttt{x + x} \Downarrow \quad \begin{array}{|l} \{\texttt{x} \mapsto \texttt{3}\} \vdash \texttt{x} \Downarrow \texttt{3} \\ \{\texttt{x} \mapsto \texttt{3}\} \vdash \texttt{x} \Downarrow \texttt{3} \end{array} \quad \Downarrow \texttt{6} \end{array} \quad \Downarrow \texttt{6}$$

The derivation makes clear how the environment semantics differs from the substitution semantics. Rather than replacing a bound variable with its value, we add the bound variable with its value to the environment; when an occurrence of the variable is reached, we simply look up its value in the environment.

Exercise 184

Construct the derivation for the expression

```
let x = 3 in
let y = 5 in
x + y ;;
```

Exercise 185

Construct the derivation for the expression

```
let x = 3 in
let x = 5 in
x + x ;;
```

Continuing the translation of the substitution semantics directly into an environment semantics, we turn to functions and their application. Maintaining functions as values is reflected in this simple rule:

$$E \vdash \texttt{fun}\ x\ \texttt{->}\ P \Downarrow \texttt{fun}\ x\ \texttt{->}\ P \qquad (R_{fun})$$

and the application of a function to its argument again adds the argument's value to the environment used in evaluating the body of the function:

$$\begin{array}{l} E \vdash P\ \ Q \Downarrow \\ \quad \left|\begin{array}{l} E \vdash P \Downarrow \texttt{fun}\ x\ \texttt{->}\ B \\ E \vdash Q \Downarrow v_Q \\ E\{x \mapsto v_Q\} \vdash B \Downarrow v_B \end{array}\right. \\ \quad \Downarrow v_B \end{array} \qquad (R_{app})$$

Exercise 186

Provide glosses for these two rules.

We can try the example from Section 13.6:

```
(fun x -> x + x) (3 * 4)
```

whose evaluation to 24 is captured by the following derivation:

$$\begin{array}{l} \{\} \vdash \texttt{(fun x -> x + x) (3 * 4)} \\ \quad \Downarrow \\ \quad \left|\begin{array}{l} \{\} \vdash \texttt{(fun x -> x + x)} \Downarrow \texttt{(fun x -> x + x)} \\ \{\} \vdash \texttt{3 * 4} \Downarrow \\ \qquad \left|\begin{array}{l} \{\} \vdash \texttt{3} \Downarrow \texttt{3} \\ \{\} \vdash \texttt{4} \Downarrow \texttt{4} \end{array}\right. \\ \qquad \Downarrow \texttt{12} \\ \{\texttt{x} \mapsto \texttt{12}\} \vdash \texttt{x + x} \Downarrow \\ \qquad \left|\begin{array}{l} \{\texttt{x} \mapsto \texttt{12}\} \vdash \texttt{x} \Downarrow \texttt{12} \\ \{\texttt{x} \mapsto \texttt{12}\} \vdash \texttt{x} \Downarrow \texttt{12} \end{array}\right. \\ \qquad \Downarrow \texttt{24} \end{array}\right. \\ \quad \Downarrow \texttt{24} \end{array}$$

The full set of dynamic environment semantics rules so far is presented in Figure 19.1.

Figure 19.1: Dynamic environment semantics rules for evaluating expressions, for a functional language with naming and arithmetic.

$$E \vdash \overline{n} \Downarrow \overline{n} \qquad (R_{int})$$

$$E \vdash x \Downarrow E(x) \qquad (R_{var})$$

$$E \vdash \texttt{fun}\ x\ \texttt{->}\ P \Downarrow \texttt{fun}\ x\ \texttt{->}\ P \qquad (R_{fun})$$

$$\frac{E \vdash P \Downarrow \overline{m} \quad E \vdash Q \Downarrow \overline{n}}{E \vdash P\ \texttt{+}\ Q \Downarrow\ \Downarrow \overline{m+n}} \qquad (R_{+})$$

(and similarly for other binary operators)

$$\frac{E \vdash D \Downarrow v_D \quad E\{x \mapsto v_D\} \vdash B \Downarrow v_B}{E \vdash \texttt{let}\ x\ \texttt{=}\ D\ \texttt{in}\ B \Downarrow\ \Downarrow v_B} \qquad (R_{let})$$

$$\frac{E \vdash P \Downarrow \texttt{fun}\ x\ \texttt{->}\ B \quad E \vdash Q \Downarrow v_Q \quad E\{x \mapsto v_Q\} \vdash B \Downarrow v_B}{E \vdash P\ \ Q \Downarrow\ \Downarrow v_B} \qquad (R_{app})$$

Problems with the dynamic semantics The environment semantics captured in these rules (Figure 19.1) seems like it should generate the same evaluations as the substitution semantics (Figure 13.5). After all, the only difference would seem to be that instead of the binding constructs (`let` and `fun`) substituting a value for the variables they bind, they place the value in the environment, to be retrieved when the variables they bind need them. But there are subtle differences, hidden in the decision as to which variable occurrences see which values.

Recall (Section 5.5) that in OCaml the connection between occurrences of variables and the binding constructs they are bound by is

determined by the *lexical structure* of the code. For instance, in the expression

```
# let x = 1 in
# let f = fun y -> x + y in
# let x = 2 in
# f 3 ;;
Line 3, characters 4-5:
3 | let x = 2 in
        ^
Warning 26 [unused-var]: unused variable x.
- : int = 4
```

the highlighted occurrence of the variable x is bound by the outer `let` x, not the inner. For that reason, the result of evaluating the expression is 4, and not 5. The substitution semantics reflects this fact, as seen in the derivation

```
let x = 1 in let f = fun y -> x + y in let x = 2 in f 3
    ⇓
    | 1⇓1
    | let f = fun y -> 1 + y in let x = 2 in f 3
    |    ⇓
    |    | fun y -> 1 + y⇓fun y -> 1 + y
    |    | let x = 2 in (fun y -> 1 + y) 3
    |    |    ⇓
    |    |    | 2⇓2
    |    |    | (fun y -> 1 + y) 3
    |    |    |    ⇓
    |    |    |    | fun y -> 1 + y⇓fun y -> 1 + y
    |    |    |    | 3⇓3
    |    |    |    | 1 + 3⇓
    |    |    |    |       | 1⇓1
    |    |    |    |       | 3⇓3
    |    |    |    |       ⇓4
    |    |    |    ⇓4
    |    |    ⇓4
    |    ⇓4
    ⇓4
```

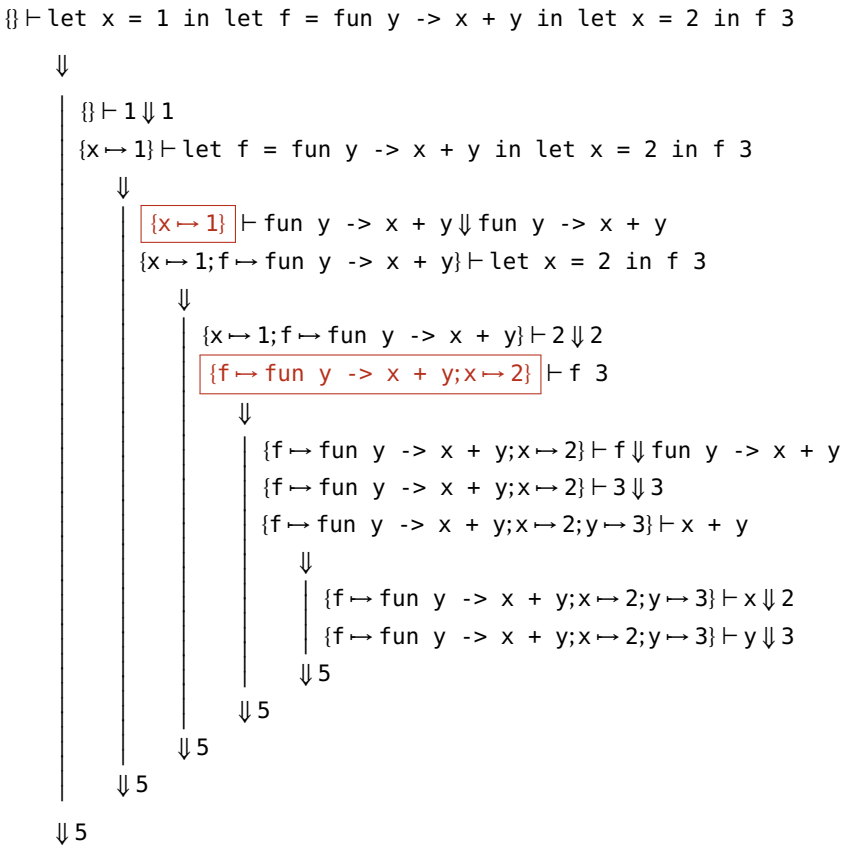
But the environment semantics evaluates this expression to 5.

Exercise 187

Before proceeding, see if you can construct the derivation for this expression according to the environment semantics rules. Do you see where the difference lies?

According to the environment semantics developed so far, a deriva-

tion for this expression proceeds as



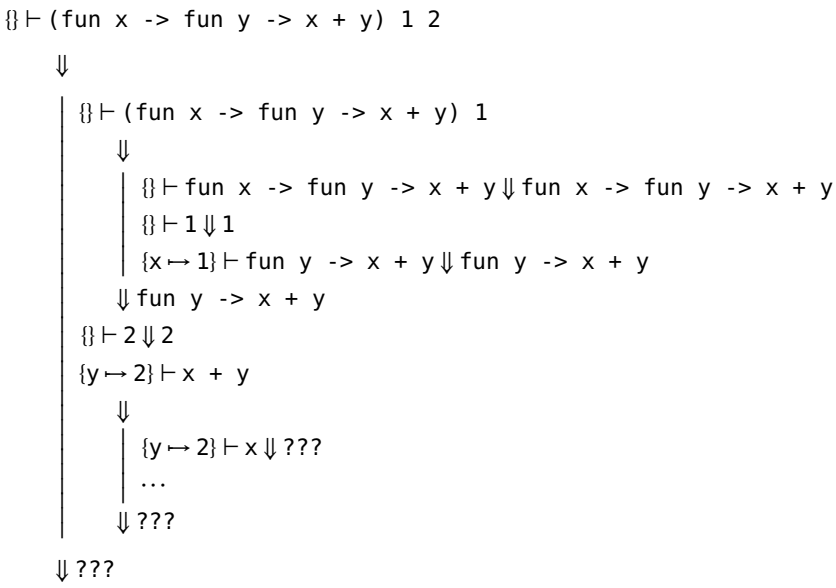
The crucial difference comes when augmenting the environment during application of the function `fun y -> x + y` to its argument. Examine closely the two highlighted environments in the derivation above. The first is the environment in force when the function is *defined*, the LEXICAL ENVIRONMENT of the function. The second is the environment in force when the function is *applied*, its DYNAMIC ENVIRONMENT. The environment semantics presented so far augments the dynamic environment with the new binding induced by the application. It manifests a DYNAMIC ENVIRONMENT SEMANTICS. But for consistency with the substitution semantics (which substitutes occurrences of a bound variable when the binding construct is defined, not applied), we should use the lexical environment, thereby manifesting a LEXICAL ENVIRONMENT SEMANTICS.

In Section 19.2.2, We'll develop a lexical environment semantics that cleaves more faithfully to the lexical scope of the substitution semantics, but first, we note some other divergences between dynamic and lexical semantics.

Consider this simple application of a curried function:

```
(fun x -> fun y -> x + y) 1 2
```

The substitution semantics rules specify that this expression evaluates to 3. But the dynamic semantics misbehaves:



We can start the derivation, but the dynamic environment available when we come to evaluate the x in the function body contains no binding for x! (If only we had been evaluating the body in its lexical environment.) In a dynamic semantics, currying – so central to many functional idioms – becomes impossible.

On the other hand, under a dynamic semantics, recursion needs no special treatment. By using the dynamic environment in evaluating the definiendum of a `let`, the definition of the bound variable is already available. We revisit the derivation for factorial from Section 13.7, but

this time using the dynamic environment semantics:

```
{} ⊢ let f = fun n -> if n = 0 then 1 else n * f (n - 1) in f 2
  ⇓
  | {} ⊢ fun n -> if n = 0 then 1 else n * f (n - 1)
  |     ⇓ fun n -> if n = 0 then 1 else n * f (n - 1)
  | {f ↦ fun n -> if n = 0 then 1 else n * f (n - 1)} ⊢ f 2
  |    ⇓
  |    | {f ↦ fun n -> if n = 0 then 1 else n * f (n - 1)} ⊢ f
  |    |     ⇓ fun n -> if n = 0 then 1 else n * f (n - 1)
  |    | {f ↦ fun n -> if n = 0 then 1 else n * f (n - 1)} ⊢ 2
  |    |     ⇓ 2
  |    | {f ↦ if n = 0 then 1 else n * f (n - 1);n ↦ 2} ⊢ if n = 0 then 1 else n * f (n - 1)
  |    |    ⇓
  |    |    | ...
  |    |    ⇓ 2
  |    ⇓ 2
  ⇓ 2
```

Notice how the body of the function, with its free occurrence of the variable `f`, is evaluated in an environment in which `f` is bound to the function itself. By using the dynamic environment semantics rules, we get recursion "for free". Consequently, the dynamic semantics rule for the `let rec` construction can simply mimic the `let` construction:

$$E \vdash \texttt{let rec } x \texttt{ = } D \texttt{ in } B \Downarrow \quad \begin{array}{|l} E \vdash D \Downarrow v_D \\ E\{x \mapsto v_D\} \vdash B \Downarrow v_B \end{array} \quad \Downarrow v_B \qquad (R_{letrec})$$

To truly reflect the intended semantics of expressions in an environment semantics, we need to find a way of using the lexical environment for functions instead of the dynamic environment; we need a *lexical environment semantics*.

19.2.2 Lexical environment semantics

To modify the rules to provide a lexical rather than dynamic environment semantics, we must provide some way of capturing the lexical environment when functions are defined. The technique is to have functions evaluate not to themselves (awaiting the dynamic environment for interpretation of the variables within them), but rather to have them evaluate to a "package" containing the function and its lexical (defining) environment. This package is called a CLOSURE.[1]

[1] The term comes from the terminology of open versus closed expressions. Open expressions have free variables in them; closed expressions have none. By capturing the defining environment, we essentially use it to close the free variables in the function. The closure thus turns what would otherwise be an open expression into a closed expression.

We'll notate the closure that packages together a function P and its environment E as $[E \vdash P]$. In evaluating a function, then, we merely construct such a closure, capturing the function's defining environment.

$$E \vdash \texttt{fun}\ x\ \texttt{->}\ P \Downarrow [E \vdash \texttt{fun}\ x\ \texttt{->}\ P] \qquad (R_{fun})$$

We make use of closures constructed in this way when the function is applied:

$$\frac{\begin{array}{l} E_d \vdash P \Downarrow [E_l \vdash \texttt{fun}\ x\ \texttt{->}\ B] \\ E_d \vdash Q \Downarrow v_Q \\ E_l\{x \mapsto v_Q\} \vdash B \Downarrow v_B \end{array}}{E_d \vdash P\ \ Q \Downarrow v_B} \qquad (R_{app})$$

Rather than augmenting the dynamic environment E_d in evaluating the body, we augment the lexical environment E_l extracted from the closure.

The lexical environment semantics properly reflects the intended semantics for several of the problematic examples in Section 19.2.1, as demonstrated in the following exercises. However, the handling of recursion still requires some further work, which we'll return to in Section 19.4.

Exercise 188

Carry out the derivation using the lexical environment semantics for the expression

```
let x = 1 in
let f = fun y -> x + y in
let x = 2 in
f 3 ;;
```

What value does it evaluate to under the lexical environment semantics?

Exercise 189

Carry out the derivation using the lexical environment semantics for the expression

```
(fun x -> fun y -> x + y) 1 2 ;;
```

Problem 190

In problem 155, you evaluated several expressions as OCaml would, with lexical scoping. Which of those expressions would evaluate to a different value using dynamic scoping?

19.3 Conditionals and booleans

In Section 13.5, exercises asked you to develop abstract syntax and substitution semantics rules for booleans and conditionals. In this section, we call for similar rules for environment semantics (applicable to dynamic or lexical variants).

Exercise 191

Adjust the substitution semantics rules for booleans from Exercise 135 to construct environment semantics rules for the constructs.

Exercise 192

Adjust the substitution semantics rules for conditional expressions (`if` ⟨⟩ `then` ⟨⟩ `else` ⟨⟩) from Exercise 136 to construct environment semantics rules for the construct.

19.4 Recursion

The dynamic environment semantics already allows for recursion – in fact, too much recursion – because of its dynamic nature. Think about an ill-formed "almost-recursive" function, like

```
let f = fun x -> if x = 0 then 1 else f (x - 1) in f 1 ;;
```

It's ill-formed because the lack of a `rec` keyword means that the `f` in the definition part ought to be unbound. But it works just fine in the dynamic environment semantics. When `f 1` is evaluated in the dynamic environment in which `f` is bound to `fun x -> if x = 0 then 1 else f (x - 1)`, all of the subexpressions of the definiens, including the occurrence of `f` itself, will be evaluated in an augmentation of that environment, so the "recursive" occurrence of `f` will obtain a value. (It is perhaps for this reason that the earliest implementations of functional programming languages, the original versions of LISP, used a dynamic semantics.)

The lexical semantics, of course, does not benefit from this fortuitous accident of definition. The lexical environment in force when `f` is defined is empty, and thus, when the body of `f` is evaluated, it is the empty environment that is augmented with the argument `x` bound to `1`. There is no binding for the recursively invoked `f`, and the derivation cannot be completed – consistent, by the way, with how OCaml behaves:

```
# let f = fun x -> if x = 0 then 1 else f (x - 1) in f 1 ;;
Line 1, characters 38-39:
1 | let f = fun x -> if x = 0 then 1 else f (x - 1) in f 1 ;;
                                          ^

Error: Unbound value f
Hint: If this is a recursive definition,
you should add the 'rec' keyword on line 1
```

To allow for recursion in the lexical environment semantics, we should add a special rule for `let rec` then. A `let rec` expression is built from three parts: a variable name (x), a definition expression (D), and a body (B). To evaluate it, we ought to first evaluate the definition part D, but using what environment? Any functions inside the definition part will see this environment as their lexical environment, to

be captured in a closure. We'll thus want to make a value for x available in that environment. But what will we use for the value of x in the environment? We can't merely map x to the definition D, with its free occurence of x; that just postpones the problem.

In one sense, it doesn't matter what value we use for x in evaluating the definition D, because in evaluating D, we won't (or at least better not) make use of x directly, as for instance in

```
# let rec x = x + 1 in x ;;
Line 1, characters 12-17:
1 | let rec x = x + 1 in x ;;
                ^^^^^
Error: This kind of expression is not allowed as right-hand side of
  `let rec'
```

That wouldn't be a well-founded recursion. Instead, the occurrences of x in D will have to be in contexts where they are not evaluated. Canonically, that would be within an unapplied function, like the factorial example

```
# let rec f = fun n -> if n = 0 then 1 else n * f (n - 1) in f 2 ;;
- : int = 2
```

Because of this requirement for well-founding of the recursion, whatever value we use for x, we'll be able to evaluate the definition to some value, call it v_D. That value, however, may involve closures that capture the binding for x, and we'll need to look up the value for x later in evaluating the body. Thus, the environment used in evaluating the body best have a binding for x to v_D.

These considerations call for the following approach to handling the semantics of `let rec` in an environment E. We start by forming an environment E' that extends E with a binding for x, but a binding that is *mutable*, so it can be changed later. Initially, x can be bound to some recognizable and otherwise ungenerable value, say, `Unassigned`. We evaluate the definition D in environment E' to a value v_D, which may capture E' (or extensions of it) in closures. We then *change* the value stored for x in E' to v_D, and evaluate the body B in E' (thus modified). By mutating the value bound to x, any closures that have captured E' will see this new value for x as well, so that (recursive) lookups of x in the body will see the evaluated v_D as well.

Because this approach relies on mutation, our notation for environment semantics isn't up to the task of formalizing this idea, and doing so is beyond the scope of this discussion, so we'll leave it at that for now. But once mutability is incorporated into the semantics – that was the whole point in moving to an environment semantics, remember – we'll revisit the issue and give appropriate rules for `let rec`.

Even without formal rules for `let rec`, you'll see in Chapter A how

this approach can be implemented in an interpreter for a language with a `let rec` construction.

Figure 19.2: Lexical environment semantics rules for evaluating expressions, for a functional language with naming and arithmetic.

$$E \vdash \overline{n} \Downarrow \overline{n} \qquad (R_{int})$$

$$E \vdash x \Downarrow E(x) \qquad (R_{var})$$

$$E \vdash \texttt{fun}\ x\ \texttt{->}\ P \Downarrow [E \vdash \texttt{fun}\ x\ \texttt{->}\ P] \qquad (R_{fun})$$

$$\frac{\begin{array}{l} E \vdash P \Downarrow \overline{m} \\ E \vdash Q \Downarrow \overline{n} \end{array}}{E \vdash P\ \texttt{+}\ Q \Downarrow \overline{m+n}} \qquad (R_{+})$$

(and similarly for other binary operators)

$$\frac{\begin{array}{l} E \vdash D \Downarrow v_D \\ E\{x \mapsto v_D\} \vdash B \Downarrow v_B \end{array}}{E \vdash \texttt{let}\ x\ \texttt{=}\ D\ \texttt{in}\ B \Downarrow v_B} \qquad (R_{let})$$

$$\frac{\begin{array}{l} E_d \vdash P \Downarrow [E_l \vdash \texttt{fun}\ x\ \texttt{->}\ B] \\ E_d \vdash Q \Downarrow v_Q \\ E_l\{x \mapsto v_Q\} \vdash B \Downarrow v_B \end{array}}{E_d \vdash P\ \ Q \Downarrow v_B} \qquad (R_{app})$$

19.5 *Implementing environment semantics*

In Section 13.4.2, we presented an implementation of the substitution semantics in the form of a function `eval : expr -> expr`. Modifying it to follow the environment semantics requires just a few simple changes. First, evaluation is relative to an environment, so the `eval` function should take an additional argument, of type, say `env`. Second, under the lexical environment semantics, expressions evaluate to values that include more than just the pertinent subset of expressions. In particular, expressions may evaluate to closures, so that an extended

notion of value, codified in a type `value` is needed. In summary, the type of `eval` should be `expr -> env -> value`.

The new `env` type, a simple mapping from variables to the values they are bound to, can be implemented as an association list

```
type env = (varid * value ref) list
```

and the `value` type can include expression values and closures in a simple variant type

```
type value =
  | Val of expr
  | Closure of (expr * env)
```

(The `env` data structure maps variables to mutable `value refs` to allow for the mutation required in implementing `let rec` as described in Section 19.4.) The carrying out of this exercise is the subject of Chapter A.

19.6 Semantics of mutable storage

In this section, we further extend the lexical environment semantics to allow for imperative programming with references and assignment. (As a byproduct, we'll have the infrastructure to probide a formal semantics rule for `let rec`.) To do so, we'll start by augmenting the syntax of the language, and then adjust the environment semantic rules so that the context of evaluation includes not only an environment, but also a model for the mutable storage that references require.

We'll start with adding to the syntax a unit value `()` and operators (`ref`, `!`, and `:=`) to manipulate reference values:

⟨*binop*⟩ ::= `+` | `-` | `*` | `/`
⟨*var*⟩ ::= `x` | `y` | `z` | ⋯
⟨*expr*⟩ ::= ⟨*integer*⟩
| ⟨*var*⟩
| ⟨$expr_1$⟩ ⟨*binop*⟩ ⟨$expr_2$⟩
| `let` ⟨*var*⟩ `=` ⟨$expr_{def}$⟩ `in` ⟨$expr_{body}$⟩
| `fun` ⟨*var*⟩ `->` ⟨$expr_{body}$⟩
| ⟨$expr_{fun}$⟩ ⟨$expr_{arg}$⟩
| `()`
| `ref` ⟨*expr*⟩
| `!` ⟨*expr*⟩
| ⟨*var*⟩ `:=` ⟨*expr*⟩

The plan for handling references is to add a new kind of value, a LOCATION, which is an index or pointer into an abstract model of memory that we will call the STORE. A store S will be a finite mapping

(like the environment) from locations to values. So a reference to a value v will be a location l such that the store S maps l to v. Evaluation will need to be relative to a store in addition to an environment, so evaluation judgements will look like $E, S \vdash P \Downarrow \cdots$.

Because the store can change as a side effect of evaluation (that's the whole point of mutability), the result of evaluation can't simply be a value. We'll need access to the modified store as well. So the right-hand side of the evaluation arrow $\Downarrow$ will provide both a value and a store. Our final form for evaluation judgements is thus

$$E, S \vdash P \Downarrow v_P, S' \quad .$$

(See Figure 19.3 for a breakdown of such a judgement.)

A semantic rule for references reflects these ideas:

$$E, S \vdash \texttt{ref}\ P \Downarrow$$
$$\left| \; E, S \vdash P \Downarrow v_P, S' \right. \qquad (R_{ref})$$
$$\Downarrow l, S'\{l \mapsto v_P\} \qquad \text{(where } l \text{ is a new location)}$$

According to this rule, "to evaluate an expression of the form `ref` P in an environment E and store S, we evaluate P in that environment and store, yielding a value v_P for P and a new store S' (as there may have been side effects to S in the evaluation). The value for the reference is a new location l, and as side effect, a new store that is S' augmented so that l maps to the value v_P."

To dereference such a reference, as in an expression of the form `!` P, P will need to be evaluated to a location, and the value at that location retrieved.

Exercise 193

Write a semantic rule for dereferencing references.

Finally, and most centrally to the idea of mutable storage, is *assignment* to a reference. Evaluating an assignment of the form P `:=` Q involves evaluating P to a location l and evaluating Q to a value v_Q, and updating the store so that l maps to v_Q. Along the way, the various subevaluations may themselves have side effects leading to updated stores, which must be dealt with. For instance, starting with an environment E and store S, evaluating P may result in an updated store S'. That updated store would then be the store with respect to which Q would be evaluated, leading to a possibly updated store S''. It is this final store that would be augmented with the new assignment. A rule

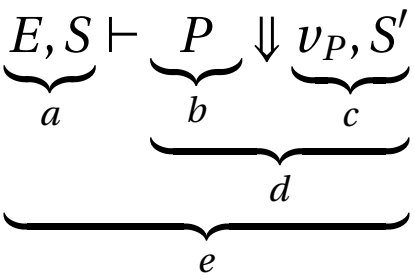


Figure 19.3: Anatomy of an evaluation judgement. (*a*) The context of evaluation, including an environment E and a store S. (*b*) The expression to be evaluated. (*c*) The result of the evaluation, a value and a store mutated by side effect. (*d*) The evaluation of P to its result. (*e*) The judgement as a whole. "In the environment E and store S, expression P evaluates to value v_P with modified store S'."

specifying this semantics is

$$
\begin{array}{l}
E, S \vdash P \texttt{ := } Q \Downarrow \\
\qquad \left| \begin{array}{l} E, S \vdash P \Downarrow l, S' \\ E, S' \vdash Q \Downarrow v_Q, S'' \end{array} \right. \\
\qquad \Downarrow (), S''\{l \mapsto v_Q\}
\end{array}
\qquad (R_{assign})
$$

The important point of the rule is the update to the store. But like all evaluation rules, a value must be returned for the expression as a whole. Here, we've simply returned the unit value `()`.

Exercise 194

In the presence of side effects, sequencing (with `;`) becomes important. Write an evaluation rule for sequencing.

To complete the semantics of mutable state, the remaining rules must be modified to use and update stores appropriately. Figure 19.4 provides a full set of rules.

As an example of the deployment of these semantic rules, we consider the expression

```
let x = ref 3 in
x := 42;
!x
```

Here is the derivation in full.

$\{\}, \{\} \vdash$ `let x = ref 3 in x := 42; !x`

$\Downarrow$

- $\{\}, \{\} \vdash$ `ref 3` $\Downarrow$
  - $\{\}, \{\} \vdash 3 \Downarrow 3, \{\}$
  - $\Downarrow l_1, \{l_1 \mapsto 3\}$
- $\{\mathsf{x} \mapsto l_1\}, \{l_1 \mapsto 3\} \vdash$ `x := 42; !x`
  - $\Downarrow$
    - $\{\mathsf{x} \mapsto l_1\}, \{l_1 \mapsto 3\} \vdash$ `x := 42`
      - $\Downarrow$
        - $\{\mathsf{x} \mapsto l_1\}, \{l_1 \mapsto 3\} \vdash \mathsf{x} \Downarrow l_1, \{l_1 \mapsto 3\}$
        - $\{\mathsf{x} \mapsto l_1\}, \{l_1 \mapsto 3\} \vdash 42 \Downarrow 42, \{l_1 \mapsto 3\}$
      - $\Downarrow (), \{l_1 \mapsto 42\}$
    - $\{\mathsf{x} \mapsto l_1\}, \{l_1 \mapsto 42\} \vdash$ `!x`
      - $\Downarrow$
        - $\{\mathsf{x} \mapsto l_1\}, \{l_1 \mapsto 42\} \vdash \mathsf{x} \Downarrow l_1, \{l_1 \mapsto 42\}$
      - $\Downarrow 42, \{l_1 \mapsto 42\}$
  - $\Downarrow 42, \{l_1 \mapsto 42\}$

$\Downarrow 42, \{l_1 \mapsto 42\}$

Figure 19.4: Lexical environment semantics rules for evaluating expressions, for a functional language with naming, arithmetic, and mutable storage.

$$E, S \vdash \overline{n} \Downarrow \overline{n}, S \qquad (R_{int})$$

$$E, S \vdash x \Downarrow E(x), S \qquad (R_{var})$$

$$E, S \vdash \texttt{fun } x \texttt{ -> } P \Downarrow [E \vdash \texttt{fun } x \texttt{ -> } P], S \qquad (R_{fun})$$

$$\begin{array}{l} E, S \vdash P \texttt{ + } Q \Downarrow \\ \qquad \left| \begin{array}{l} E, S \vdash P \Downarrow \overline{m}, S' \\ E, S' \vdash Q \Downarrow \overline{n}, S'' \end{array} \right. \\ \qquad \Downarrow \overline{m+n}, S'' \end{array} \qquad (R_{+})$$

(and similarly for other binary operators)

$$\begin{array}{l} E, S \vdash \texttt{let } x \texttt{ = } D \texttt{ in } B \Downarrow \\ \qquad \left| \begin{array}{l} E, S \vdash D \Downarrow v_D, S' \\ E\{x \mapsto v_D\}, S' \vdash B \Downarrow v_B, S'' \end{array} \right. \\ \qquad \Downarrow v_B, S'' \end{array} \qquad (R_{let})$$

$$\begin{array}{l} E_d, S \vdash P \;\; Q \Downarrow \\ \qquad \left| \begin{array}{l} E_d, S \vdash P \Downarrow [E_l \vdash \texttt{fun } x \texttt{ -> } B], S' \\ E_d, S' \vdash Q \Downarrow v_Q, S'' \\ E_l\{x \mapsto v_Q\}, S'' \vdash B \Downarrow v_B, S''' \end{array} \right. \\ \qquad \Downarrow v_B, S''' \end{array} \qquad (R_{app})$$

$$
\begin{array}{l}
E, S \vdash \texttt{ref}\ P \Downarrow \\
\qquad \left| \begin{array}{l} E, S \vdash P \Downarrow v_P, S' \end{array} \right. \qquad (R_{ref}) \\
\qquad \Downarrow l, S'\{l \mapsto v_P\} \qquad \text{(where } l \text{ is a new location)}
\end{array}
$$

$$
\begin{array}{l}
E, S \vdash \texttt{!}\ P \Downarrow \\
\qquad \left| \begin{array}{l} E, S \vdash P \Downarrow l, S' \end{array} \right. \qquad (R_{deref}) \\
\qquad \Downarrow S'(l), S'
\end{array}
$$

$$
\begin{array}{l}
E, S \vdash P\ \texttt{:=}\ Q \Downarrow \\
\qquad \left| \begin{array}{l} E, S \vdash P \Downarrow l, S' \\ E, S' \vdash Q \Downarrow v_Q, S'' \end{array} \right. \qquad (R_{assign}) \\
\qquad \Downarrow (), S''\{l \mapsto v_Q\}
\end{array}
$$

$$
\begin{array}{l}
E, S \vdash P\ \texttt{;}\ Q \Downarrow \\
\qquad \left| \begin{array}{l} E, S \vdash P \Downarrow (), S' \\ E, S' \vdash Q \Downarrow v_Q, S'' \end{array} \right. \qquad (R_{seq}) \\
\qquad \Downarrow v_Q, S''
\end{array}
$$

Figure 19.4: (continued) Lexical environment semantics rules for evaluating expressions, for a functional language with naming, arithmetic, and mutable storage.

19.6.1 *Lexical environment semantics of recursion*

The extended language with references and assignment is sufficient to provide a semantics for the recursive `let rec` construct. A simple way to observe this is to reconstruct a `let rec` expression of the form

```
let rec x = D in B
```

as syntactic sugar for an expression that caches the recursion out using just the trick described in Section 19.4: first assigning to x a mutable reference to a special unassigned value, then evaluating the definition D, replacing the value of x with the evaluated D, and finally, evaluating B in that environment. We can carry out that recipe with the following expression, which we can think of as the desugared `let rec`:

```
let x = ref unassigned in
x := D[x ↦ !x];
B[x ↦ !x]
```

(Since we've changed x to a reference type, we need to replace occurrences of x in D and B with `!`x to retrieve the referenced value.)

One way to verify that this approach works is to test it out in OCaml itself. Take this application of the factorial function, for instance:

```
# let rec f = fun n -> if n = 0 then 1
#                      else n * f (n - 1) in
# f 4 ;;
- : int = 24
```

Desugaring it as above, we get

```
# let unassigned = fun _ -> failwith "unassigned" ;;
val unassigned : 'a -> 'b = <fun>

# let f = ref unassigned in
# (f := fun n -> if n = 0 then 1
#               else n * !f (n - 1));
# !f 4 ;;
- : int = 24
```

(To serve as the "unassigned" value, we define `unassigned` to simply raise an exception.)

This expression, note, makes use of only the language constructs provided in the semantics in the previous section. That semantics, with its lexical environment and mutable store, thus has enough expressivity for capturing the approach to recursion described informally in Section 19.4. In fact, we could even provide a semantic rule for `let rec` by carrying through the semantics for the desugared expression. This leads to the following `let rec` semantic rule for the `let rec` construction:

$$E, S \vdash \texttt{let rec } x \texttt{ = } D \texttt{ in } B \Downarrow$$
$$\left|\begin{array}{l} E\{x \mapsto l\}, S\{l \mapsto \texttt{unassigned}\} \vdash D[x \mapsto \texttt{!}x] \Downarrow v_D, S' \\ E\{x \mapsto l\}, S'\{l \mapsto v_D\} \vdash B[x \mapsto \texttt{!}x] \Downarrow v_B, S'' \end{array}\right.$$
$$\Downarrow v_B, S'' \qquad (R_{letrec})$$

Problem 195

For the formally inclined, prove that the semantic rule for `let rec` above is equivalent to the syntactic sugar approach.

19.7 Supplementary material

- Lab 18: Environment semantics
- Lab 19: Synthesis: Cellular automata
- Lab 20: Synthesis: Digital halftoning