13
Semantics: The substitution model

We've introduced a broad swath of OCaml, describing both the syntax of different constructions and their use in constructing programs. But *why* the expressions of OCaml actually have the meanings they have has been dealt with only informally.

Semantics is about what expressions *mean*. As described so far, asking what an OCaml expression means is tantamount to asking what it evaluates to, what value it "means the same" as. Before getting into the details, however, it bears considering why a *formal, rigorous, precise* semantics of a programming language is even useful. Why not stick to the informal discussion of what the constructs of a programming language do? After all, such informal discussions, written in a natural language (like English), seem to work just fine for reference manuals and training videos.

There are three reasons that formalizing a semantics with mathematical rigor is beneficial.

Mental hygiene Programming is used to communicate our computational intentions to others. But what *exactly* is being communicated? Without a precise meaning to the expressions of the programming language, there is room for miscommunication from program author to reader.

Interpreters Computers generate computation by interpreting the expressions of the programming language. Developers of interpreters (or compilers) for a programming language implement their understanding of the meaning of the constructs of the programming language. Without a precise meaning to the expressions of the programming language, two interpreters might generate different computations for the same expression, even though both were written in good faith efforts to manifest the interpreter developers' understandings of the language.

Metaprogramming Programs that operate over expressions of the

programming language – such as programs to verify correctness of a program or transform it for efficiency or analyze it for errors – must use a precise notion of the meanings of those expressions.

For these reasons, we introduce in this chapter a technique for giving a semantics to some small subsets of OCaml. We continue this exercise in Chapter 19. The final project described in Chapter A – the implementation of a small subset of OCaml – relies heavily on the discussion in these two chapters.

As noted, we'll cash out the meaning of an expression by generating a simpler expression that "means the same". In essence, this is the notion of evaluation that we've seen before. In this chapter we'll introduce a first method for providing a *rigorous* semantics of a programming language, based on the substitution of subexpressions, substituting for particular expressions expressions that "mean the same" but that are simpler.

The underlying conception of substitution as the basis for semantics dates from 1677 in Gottfried Leibniz's statement of the *identity of indiscernibles*:

> That *A* is the same as *B* signifies that the one can be substituted for the other, *salva veritate*, in any proposition whatever.

Salva veritate – preserving the truth. Leibniz claims that substituting one expression with another that means the same thing preserves the truth of expressions.

We'll see later (Chapters 15 and 16) that a naive interpretation of Leibniz's law isn't sustainable. In particular, in the presence of state and state change, the province of imperative programming, the law seems to fail. But for the portion of OCaml we've seen so far, Leibniz's statement works quite well.

Following Leibniz's view, in this chapter we provide a semantics for a language that can be viewed as a (simple and untyped) subset of OCaml, with constructs like arithmetic and boolean operators, conditionals, functions (including recursive functions), and local naming.

We provide these semantic notions in two ways: as formal rule systems that define the evaluation relation, and as computer programs to evaluate expressions to their values.

The particular method of providing formal semantics that we introduce in this chapter is called *large-step operational semantics* and is based on the NATURAL SEMANTICS method of computer scientist Gilles Kahn (Figure 13.2).

The semantics we provide is FORMAL in the sense that the semantic rules rely only on manipulations based on the *forms* of the notations

Figure 13.1: Gottfried Wilhelm Leibniz (1646–1716), German philosopher, (co-)inventor of the differential and integral calculus, and philosopher. His law of the identity of indiscernibles underlies substitution semantics.

Figure 13.2: Gilles Kahn (1946–2006), French computer scientist, developer of the natural semantics approach to programming language semantics. Kahn was president of the French research institute INRIA, where OCaml was developed.

we introduce. The semantics we provide is an OPERATIONAL SEMANTICS because we provide a formal specification of what programs *evaluate to*, rather than what they *denote*.[1] The semantics we provide is a LARGE-STEP semantics because it characterizes directly the relation between expressions and what they (eventually, after perhaps many individual small steps) evaluate to, rather than characterizing the relation between expressions and what they lead to after each individual small step. (That would be a SMALL-STEP SEMANTICS.) Notationally, we characterize this relation between an expression P and the value v it evaluates to with an evaluation JUDGEMENT notated $P \Downarrow v$, which can be read as "the expression P evaluates to the value v".

[1] The primary alternative method of providing a formal semantics is DENOTATIONAL SEMANTICS, which addresses exactly this issue of what expressions denote.

13.1 Semantics of arithmetic expressions

Recall the language of arithmetic expressions from Section 11.4. We start by augmenting that language with a local naming construct, the `let` ⟨⟩ `in` ⟨⟩ . We'll express the abstract syntax of the language using the following BNF:

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

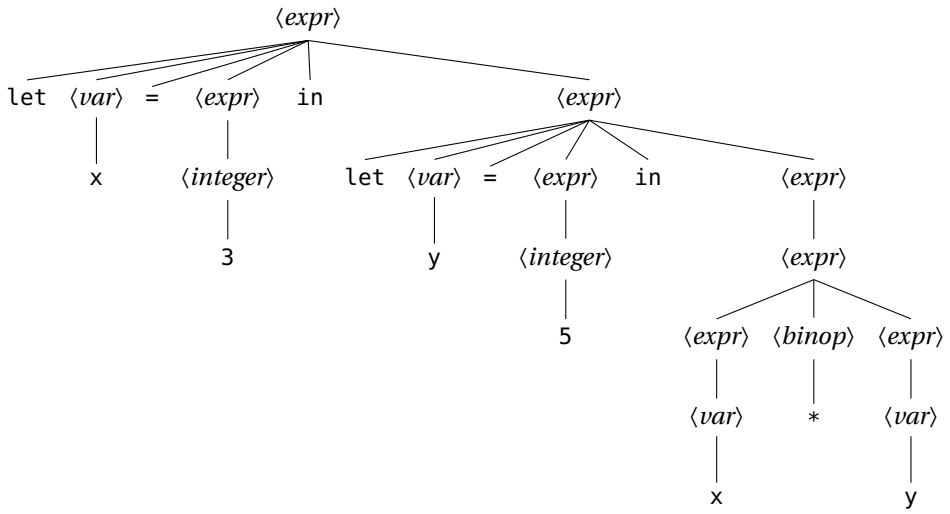
Exercise 123

For brevity, we left off unary operators. Extend the grammar to add unary operators (negation, say).

With this grammar, we can express the abstract syntax of the concrete expression

```
let x = 3 in
let y = 5 in
x * y
```

as the tree



What rules shall we use for evaluating the expressions of the language? Recall that we write a judgement $P \Downarrow v$ to mean that the expression P evaluates to the value v. The VALUES, the results of evaluation, are those expressions that evaluate to themselves. By convention, we'll use italic capitals like P, Q, etc. to stand for arbitrary expressions, and v (possibly subscripted) to stand for expressions that are values. You should think of P and v as expressions structured as per the abstract syntax of the language – it is the abstract, structured expressions that have well-defined meanings by the rules we'll provide – though we notate them using the concrete syntax of OCaml, since we need some linear notation for specifying them.

Certain cases are especially simple. Numeric literal expressions like `3` or `5` are already as simplified as they can be. They evaluate to themselves; they are values. We could enumerate a plethora of judgements that express this self-evaluation, like

$$\texttt{1} \Downarrow \texttt{1}$$
$$\texttt{2} \Downarrow \texttt{2}$$
$$\texttt{3} \Downarrow \texttt{3}$$
$$\texttt{4} \Downarrow \texttt{4}$$
$$\texttt{5} \Downarrow \texttt{5}$$
$$\ldots$$

but we'd need an awful lot of them. Instead, we'll just use a schematic rule for capturing permissible judgements:

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

Here, we use a schematic variable n to stand for any integer, and use the notation $\overline{n}$ for the OCaml numeral expression that encodes the number n.

Using this schematic rule notation we can provide general rules for evaluating other arithmetic expressions. To evaluate an expression of the form P `+` Q, where P and Q are two subexpressions, we first need to know what values P and Q evaluate to; since they will be numeric values, we can take them to be $\overline{m}$ and $\overline{n}$, respectively. Then the value that P `+` Q evaluates to will be $\overline{m+n}$. We'll write the rule as follows:

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

In this rule notation, the first line is intended to indicate that we are evaluating P `+` Q, the blank space to the right of the $\Downarrow$ indicating that some further evaluation judgements are required. Those are the two indented judgements provided to the right of the long vertical bar between the two occurrences of $\Downarrow$. The final line provides the value that the original expression evaluates to.

Thus, this rule can be glossed as "To evaluate an expression of the form P `+` Q, first evaluate P to an integer value $\overline{m}$ and Q to an integer value $\overline{n}$. The value of the full expression is then the integer literal representing the sum of m and n." The two subderivations for $P \Downarrow \overline{m}$ and $Q \Downarrow \overline{n}$ are derived independently, and not in any particular order.

Using these two rules, we can now show a particular evaluation, like that of the expression `3 + 5`:[2]

[2] Wait, where did that `8` come from exactly? Since $\texttt{3} \equiv \overline{3}$ and $\texttt{5} \equiv \overline{5}$, the rule R_{int} gives the result as $\overline{3+5} \equiv \overline{8} \equiv \texttt{8}$.

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

or the evaluation of `3 + 5 + 7`:

$$\begin{array}{l} \texttt{3 + 5 + 7} \Downarrow \\ \qquad \left|\begin{array}{l} \texttt{3 + 5} \Downarrow \\ \qquad \left|\begin{array}{l} \texttt{3} \Downarrow \texttt{3} \\ \texttt{5} \Downarrow \texttt{5} \end{array}\right. \\ \qquad \Downarrow \texttt{8} \\ \texttt{7} \Downarrow \texttt{7} \end{array}\right. \\ \qquad \Downarrow \texttt{15} \end{array}$$

Exercise 124

Why is the proof for the value of `3 + 5 + 7` not structured as

```
3 + 5 + 7⇓
        |  3⇓3
        |  5 + 7⇓
        |        |  5⇓5
        |        |  7⇓7
        |        ⇓12
        ⇓15    ?
```

We should have similar rules for other arithmetic operators. Here's a possible rule for division:

$$\dfrac{\begin{array}{l} P \Downarrow \overline{m} \\ Q \Downarrow \overline{n} \end{array}}{P \ / \ Q \Downarrow \overline{\lfloor m/n \rfloor}} \qquad (R_/)$$

In this rule, we've used some standard mathematical notation in the final result: / for numeric division and $\lfloor\ \rfloor$ for truncating a real number to an integer.

These rules for addition and division may look trivial, but they are not. The division rule specifies that the / operator in OCaml when applied to two numerals specifies the integer portion of their ratio. The language being specified might have been otherwise.[3] The language might have used a different operator (like //) for integer division,

$$\dfrac{\begin{array}{l} P \Downarrow \overline{m} \\ Q \Downarrow \overline{n} \end{array}}{P \ // \ Q \Downarrow \overline{\lfloor m/n \rfloor}}$$

(as happens to be used in Python 3 for instance). The example should make clear the distinction between the OBJECT LANGUAGE whose semantics is being defined and the METALANGUAGE being used to define it.

Similarly, the rule could have defined the result differently, say

$$\dfrac{\begin{array}{l} P \Downarrow \overline{m} \\ Q \Downarrow \overline{n} \end{array}}{P \ / \ Q \Downarrow \overline{\lceil m/n \rceil}} \ ,$$

which specifies that the result of the division is the integer resulting from rounding up, rather than down.

Nonetheless, there is not *too* much work being done by these rules, and if that were all there were to defining a semantics, there would be

[3] What may be mind-boggling here is the role of the mathematical notation used in the result part of the rule. How is it that we can make use of notations like $\lfloor m/n \rfloor$ in defining the semantics of the / operator? Doesn't appeal to that kind of mathematical notation beg the question? Or at least call for its own semantics? Yes, it does, but since we have to write down the semantics of constructs *somehow* or other, we use commonly accepted mathematical notation applied in the context of natural language (in the case at hand, English). You may think that this merely postpones the problem of giving OCaml semantics by reducing it to the problem of giving semantics for mathematical notation and English. You would be right, and the problem is further exacerbated when the semantics makes use of mathematical notation that is not so familiar, for instance, the substitution notation to be introduced shortly. But we have to start somewhere.

little reason to go to the trouble. Things get more interesting, however, when additional constructs such as local naming are considered, which we turn to next.

Exercise 125

Write evaluation rules for the other binary operators and the unary operators you added in Exercise 123.

13.2 *Semantics of local naming*

The ⟨*expr*⟩ language defined in the grammar above includes a local naming construct, whose concrete syntax is expressed with `let` ⟨⟩ `in` ⟨⟩ . What is the semantics of such an expression? It is here that substitution starts to play a critical role. We will take the meaning of this local naming construct to work by *substituting the value of the definition for occurrences of the variable in the body*. More precisely, we use the following evaluation rule:

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

We've introduced a new notation – $Q[x \mapsto P]$ – for substituting the expression P for occurrences of the variable x in the expression Q. For instance,

$$\texttt{(x * x)}[\texttt{x} \mapsto \texttt{5}] = \texttt{5 * 5}$$

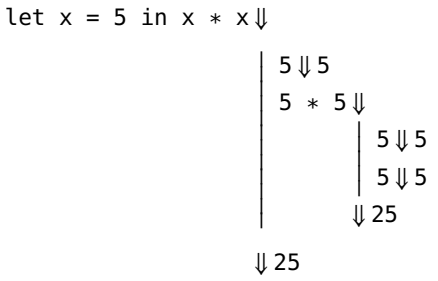
that is, substituting `5` for `x` in the expression `x * x` yields `5 * 5`. (It doesn't yield `25` though. That would require a further evaluation, which is what the part of the rule $B[x \mapsto v_D] \Downarrow v_B$ does.)

The evaluation rule R_{let} can be glossed as follows: “To evaluate an expression of the form `let` x `=` D `in` B, first evaluate the expression D to a value v_D and evaluate the result of substituting v_D for occurrences of x in the expression B to a value v_B. The value of the full expression is then v_B.”

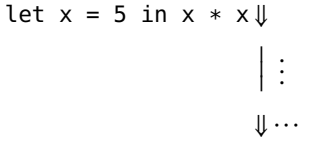
Using this rule (and the others), we can now show

$$\texttt{let x = 5 in x * x} \Downarrow \texttt{25}$$

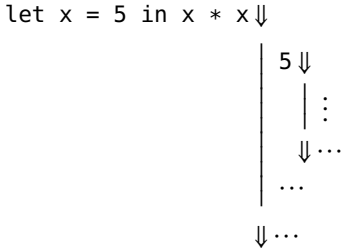
as per the following derivation:



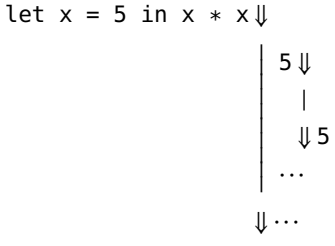
Let's put this first derivation together step by step so the steps are clear. We want a derivation that demonstrates what `let x = 5 in x * x` evaluates to. It will be of the form



This pattern matches rule R_{let}, where `x` plays the role of the schematic variable x, `5` plays the role of the schematic expression D, and `x * x` plays the role of B. We will plug these into the two subderivations required. First is the subderivation evaluating D (that is, `5`):



This subderivation can be completed using the R_{int} rule, which requires no subderivations itself.



Thus, the result of this subderivation, v_D is `5`.

Second is the subderivation for evaluating $B[x \mapsto v_D]$ to its value v_B.

Now

$$\begin{aligned} B[x \mapsto v_D] &= (\texttt{x * x})[\texttt{x} \mapsto \texttt{5}] \\ &= \texttt{x}[\texttt{x} \mapsto \texttt{5}] \ \texttt{*} \ \texttt{x}[\texttt{x} \mapsto \texttt{5}] \\ &= \texttt{5 * 5} \end{aligned}$$

(We'll define this substitution operation carefully in Section 13.3.) So the second subderivation must evaluate the expression `5 * 5`:

$$\dfrac{\dfrac{\dfrac{}{\texttt{5} \Downarrow \texttt{5}} \quad \dfrac{\dfrac{}{\texttt{5} \Downarrow \texttt{5}} \quad \dfrac{}{\texttt{5} \Downarrow \texttt{5}}}{\texttt{5 * 5} \Downarrow \texttt{25}}}{\texttt{let x = 5 in x * x} \Downarrow \texttt{25}}}{\texttt{let x = 5 in x * x} \Downarrow \texttt{25}}$$

This second subderivation matches a rule R_* analogous to R_+. (You would have written it in Exercise 125.) Here, `5` plays the role of both P and Q:

$$\dfrac{\dfrac{\dfrac{}{\texttt{5} \Downarrow \texttt{5}} \quad \dfrac{\dfrac{}{\texttt{5} \Downarrow \overline{m}} \quad \dfrac{}{\texttt{5} \Downarrow \overline{n}}}{\texttt{5 * 5} \Downarrow \overline{m \cdot n}}}{\texttt{let x = 5 in x * x} \Downarrow \overline{m \cdot n}}}{\texttt{let x = 5 in x * x} \Downarrow \overline{m \cdot n}}$$

Now, the subderivations of the `5 * 5` subderivation both evaluate to `5`. We use the R_{int} rule twice, with `5` for both $\overline{m}$ and $\overline{n}$, so m and n are both 5, and $\overline{m \cdot n}$ is 25. The result for the original expression as a whole is therefore also 25.

$$\dfrac{\dfrac{\dfrac{}{\texttt{5} \Downarrow \texttt{5}} \quad \dfrac{\dfrac{}{\texttt{5} \Downarrow \texttt{5}} \quad \dfrac{}{\texttt{5} \Downarrow \texttt{5}}}{\texttt{5 * 5} \Downarrow \texttt{25}}}{\texttt{let x = 5 in x * x} \Downarrow \texttt{25}}}{\texttt{let x = 5 in x * x} \Downarrow \texttt{25}}$$

Exercise 126

Carry out derivations for the following expressions:

1. `let x = 3 in let y = 5 in x * y`

2. `let x = 3 in let y = x in x * y`

3. `let x = 3 in let x = 5 in x * y`

4. `let x = 3 in let x = x in x * x`

5. `let x = 3 in let x = y in x * x`

Are the values for these expressions according to the semantics consistent with how OCaml evaluates them?

13.3 Defining substitution

Because of the central place of substitution in providing the semantics of the language, this approach to semantics is referred to as a SUBSTITUTION SEMANTICS.

Some care is needed in precisely defining this substitution operation. A start (which we'll see in Section 13.3.2 isn't fully correct) is given by the following recursive equational definition:[4]

$$\begin{aligned}
\overline{m}[x \mapsto Q] &= \overline{m} \\
x[x \mapsto Q] &= Q \\
y[x \mapsto Q] &= y \quad \text{where } x \not\equiv y \\
(P \texttt{ + } R)[x \mapsto Q] &= P[x \mapsto Q] \texttt{ + } R[x \mapsto Q] \\
&\quad \text{and similarly for other binary operators} \\
(\texttt{let } y \texttt{ = } D \texttt{ in } B)[x \mapsto Q] &= \texttt{let } y \texttt{ = } D[x \mapsto Q] \texttt{ in } B[x \mapsto Q]
\end{aligned}$$

[4] The $\equiv$ operator here is intended to indicate syntactic identity, that is, that its arguments are the same (syntactic) expression. Thus, $x \not\equiv y$ specifies that the two variables notated x and y are not two occurrences of the same variable.

Exercise 127

Verify using this definition for substitution the derivation above showing that $(\texttt{x * x})[\texttt{x} \mapsto \texttt{5}] = \texttt{5 * 5}$.

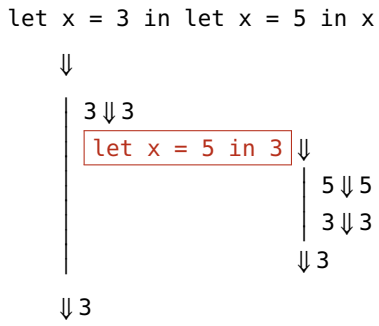
13.3.1 A problem with variable scope

You may have noticed in Exercise 126 that some care must be taken when substituting. Consider the following case:

```
let x = 3 in let x = 5 in x
```

Intuitively, given the scope rules of OCaml described informally in Section 5.5, this expression should evaluate to `5`, since the final occurrence of `x` is bound by the inner `let` (defined to be `5`), not the outer one.

However, if we're not careful, we'll get a derivation like this:



The highlighted expression is supposed to be the result of replacing `x` with its value `3` in the body of the definition `let x = 5 in x`, that is,

$$\texttt{(let x = 5 in x)}[\texttt{x} \mapsto \texttt{3}] \quad .$$

Using the equational definition given above, we have

$$\begin{aligned} &\texttt{(let x = 5 in x)}[\texttt{x} \mapsto \texttt{3}] \\ &\quad = \texttt{let x = 5}[\texttt{x} \mapsto \texttt{3}]\ \texttt{in x}[\texttt{x} \mapsto \texttt{3}] \\ &\quad = \texttt{let x = 5 in x}[\texttt{x} \mapsto \texttt{3}] \\ &\quad = \texttt{let x = 5 in 3} \quad . \end{aligned}$$

13.3.2 Free and bound occurrences of variables

It appears we must be very careful in how we define this substitution operation $P[x \mapsto Q]$. In particular, we don't want to replace *every* occurrence of the token x in P, only the *free* occurrences. The variable being introduced in a `let` should definitely not be replaced, nor should any occurrences of x within the body of a `let` that also introduces x.

A binding construct (a `let` or a `fun`) is said to BIND the variable that it introduces. A variable occurrence is said to be BOUND if it falls within the scope of a construct that binds that variable. Thus, in the expressions `fun x -> x + y` or `let x = 3 in x + y`, the highlighted occurrences of `x` are bound occurrences, bound by the `fun` or `let`, respectively, in the expressions.

A variable occurrence is said to be FREE if it is not bound. Thus, in the expressions `fun x -> x + y` or `let x = 3 in x + y`, the occurrences of `y` are free occurrences.

Exercise 128

In the following expressions, draw a line connecting each bound variable to the binding construct that binds it. Then circle all of the free occurrences of variables.

1. `x`
2. `x + y`

3. `let x = 3 in x`
4. `let f = f 3 in x + y`
5. `(fun x -> x + x) x`
6. `fun x -> let x = y in x + 3`

We can define the set[5] of FREE VARIABLES in an expression P, notated $FV(P)$, through the recursive definition in Figure 13.3. By way of example, the definition says that the free variables in the expression `fun y -> f (x + y)` are just `f` and `x`, as shown in the following derivation:

$$\begin{aligned} FV(\texttt{fun y -> f (x + y)}) &= FV(\texttt{f (x + y)}) - \{\texttt{y}\} \\ &= FV(\texttt{f}) \cup FV(\texttt{x + y}) - \{\texttt{y}\} \\ &= \{\texttt{f}\} \cup FV(\texttt{x}) \cup FV(\texttt{y}) - \{\texttt{y}\} \\ &= \{\texttt{f}\} \cup \{\texttt{x}\} \cup \{\texttt{y}\} - \{\texttt{y}\} \\ &= \{\texttt{f},\texttt{x},\texttt{y}\} - \{\texttt{y}\} \\ &= \{\texttt{f},\texttt{x}\} \end{aligned}$$

[5] For a review of the set notations that we use, see Section B.5.

Exercise 129

Use the definition of FV to derive the set of free variables in the expressions below. Circle all of the free occurrences of the variables.

1. `let x = 3 in let y = x in f x y`
2. `let x = x in let y = x in f x y`
3. `let x = y in let y = x in f x y`
4. `let x = fun y -> x in x`

Exercise 130

The definition of FV in Figure 13.3 is incomplete, in that it doesn't specify the free variables in a `let rec` expression. Add appropriate rules for this construct of the language, being careful to note that in an expression like `let rec x = fun y -> x in x`, the variable `x` is not free. (Compare with Exercise 129(4).)

13.3.3 Handling variable scope properly

Now that we have formalized the idea of free and bound variables, it may be clearer what is going wrong in the previous substitution example. The substitution rule for substituting into a `let` expression

$$(\texttt{let } y \texttt{ = } D \texttt{ in } B)[x \mapsto Q] = \texttt{let } y \texttt{ = } D[x \mapsto Q] \texttt{ in } B[x \mapsto Q]$$

shouldn't apply when x and y are *the same* variable. In such a case, the occurrences of x in D or B are not free occurrences, but are bound by the `let`. We modify the definition of substitution accordingly:

$$\overline{m}[x \mapsto Q] = \overline{m}$$

$$x[x \mapsto Q] = Q$$

$$y[x \mapsto Q] = y \qquad \text{where } x \not\equiv y$$

$$(P \texttt{ + } R)[x \mapsto Q] = P[x \mapsto Q] \texttt{ + } R[x \mapsto Q] \qquad \text{and similarly for other binary operators}$$

$$(\texttt{let } y \texttt{ = } D \texttt{ in } B)[x \mapsto Q] = \texttt{let } y \texttt{ = } D[x \mapsto Q] \texttt{ in } B[x \mapsto Q] \qquad \text{where } x \not\equiv y$$

$$(\texttt{let } x \texttt{ = } D \texttt{ in } B)[x \mapsto Q] = \texttt{let } x \texttt{ = } D[x \mapsto Q] \texttt{ in } B$$

Exercise 131

Use the definition of the substitution operation above to give the expressions (in concrete syntax) specified by the following substitutions:

1. `(x + x)[x↦3]`
2. `(x + x)[y↦3]`
3. `(x * x)[x↦3 + 4]`
4. `(let x = y in y + x)[y↦z]`
5. `(let x = y in y + x)[x↦z]`

Exercise 132

Use the semantic rules developed so far (see Figure 13.5) to reduce the following expressions to their values. Show the derivations.

1.
```
let x = 3 * 4 in
x + x
```
2.
```
let y = let x = 5
        in x + 1
in y + 2
```

13.4 Implementing a substitution semantics

Given a grammar and appropriate semantic evaluation rules and definitions for substitution, it turns out to be quite simple to implement the corresponding semantics, as a function that evaluates expressions to their values.

The grammar defining the abstract syntax of the language (repeated here for reference)

$$\begin{array}{rcl}
\langle binop \rangle & ::= & \texttt{+} \mid \texttt{-} \mid \texttt{*} \mid \texttt{/} \\
\langle var \rangle & ::= & \texttt{x} \mid \texttt{y} \mid \texttt{z} \mid \cdots \\
\langle expr \rangle & ::= & \langle integer \rangle \\
& \mid & \langle var \rangle \\
& \mid & \langle expr_1 \rangle \ \langle binop \rangle \ \langle expr_2 \rangle \\
& \mid & \texttt{let } \langle var \rangle \texttt{ = } \langle expr_{def} \rangle \texttt{ in } \langle expr_{body} \rangle
\end{array}$$

can be implemented, as we have done before (Section 11.4), with an algebraic type definition

```
# type binop = Plus | Divide ;;
type binop = Plus | Divide

# type varspec = string ;;
type varspec = string

# type expr =
#   | Int of int
#   | Var of varspec
#   | Binop of binop * expr * expr
#   | Let of varspec * expr * expr ;;
type expr =
    Int of int
  | Var of varspec
  | Binop of binop * expr * expr
  | Let of varspec * expr * expr
```

The `varspec` type specifies strings as a means to differentiate distinct variables. The `binop` type enumerates the various binary operators. (For brevity, in this example, we've only included two binary operators, for addition and division.) The `expr` type provides the alternative methods for building expressions recursively.

Then, the abstract syntax for the concrete expression

```
let x = 3 in
let y = 5 in
x / y
```

is captured by the OCaml expression

```
# Let ("x", Int 3,
#      Let ("y", Int 5,
#           Binop (Divide, Var "x", Var "y"))) ;;
- : expr =
Let ("x", Int 3, Let ("y", Int 5, Binop (Divide, Var "x", Var
  "y")))
```

Exercise 133

Augment the type definitions to allow for other binary operations (subtraction and multiplication, say) and for unary operations (negation).

13.4.1 Implementing substitution

With a representation of expressions in hand, we can proceed to implement various useful functions over the expressions. Rather than provide implementations, we leave them as exercises.

Exercise 134

Write a function `subst : expr -> varspec -> expr -> expr` that performs substitution, that is, `subst p x q` returns the expression that is the result of substituting `q` for the variable `x` in the expression `p`. For example,

```
# subst (Binop (Plus, Var "x", Var "y")) "x" (Int 3) ;;
- : expr = Binop (Plus, Int 3, Var "y")
# subst (Binop (Plus, Var "x", Var "y")) "y" (Int 3) ;;
- : expr = Binop (Plus, Var "x", Int 3)
# subst (Binop (Plus, Var "x", Var "y")) "z" (Int 3) ;;
- : expr = Binop (Plus, Var "x", Var "y")
```

13.4.2 Implementing evaluation

Now the semantics of the language – the evaluation of expressions to their values – can be implemented as a recursive function `eval : expr -> expr`, which follows the evaluation rules just introduced. The type of the function indicates that the header line should be

```
let rec eval (exp : expr) : expr = ...
```

The computation proceeds based on the structure of `exp`, which might be any of the structures introducing the semantic rules. Consequently, we match on these structures:

```
let rec eval (exp : expr) : expr =
  match exp with
  | Int n -> ...
  | Var x -> ...
  | Binop (Plus, e1, e2) -> ...
  | Binop (Divide, e1, e2) -> ...
  | Let (var, def, body) -> ...
```

The computation for each of the cases mimics the computations in the evaluation rules exactly. Integers, for instance, are self-evaluating.

```
let rec eval (exp : expr) : expr =
  match exp with
  | Int n -> Int n
  | Var x -> ...
  | Binop (Plus, e1, e2) -> ...
  | Binop (Divide, e1, e2) -> ...
  | Let (var, def, body) -> ...
```

The second pattern concerns what should be done for evaluating *free* variables in expressions. (Presumably, any *bound* variables were substituted away by virtue of the final pattern-match.) We have provided no evaluation rule for free variables, and for good reason. Expressions with free variables, called OPEN EXPRESSIONS, don't have a value in and of themselves. Consequently, we can simply report an error upon evaluation of a free variable. We introduce an exception for this purpose.

```
let rec eval (exp : expr) : expr =
  match exp with
  | Int n -> Int n
  | Var x -> raise (UnboundVariable x)
  | Binop (Plus, e1, e2) -> ...
  | Binop (Divide, e1, e2) -> ...
  | Let (var, def, body) -> ...
```

The binary operator rules work by recursively evaluating the operands and applying an appropriate computation to the results.

```
let rec eval (exp : expr) : expr =
  match exp with
  | Int n -> Int n
  | Var x -> raise (UnboundVariable x)
  | Binop (Plus, e1, e2) ->
      let Int m = eval e1 in
      let Int n = eval e2 in
      Int (m + n)
  | Binop (Divide, e1, e2) ->
      let Int m = eval e1 in
      let Int n = eval e2 in
      Int (m / n)
  | Let (var, def, body) -> ...
```

Finally, the naming rule R_{let} performs substitution of the value of the definition in the body, and evaluates the result. We appeal to the function `subst` from Exercise 134.

```
# exception UnboundVariable of string ;;
exception UnboundVariable of string

# let rec eval (exp : expr) : expr =
#   match exp with
#   | Int n -> Int n                    (* R_int *)
#   | Var x -> raise (UnboundVariable x)
#   | Binop (Plus, e1, e2) ->           (* R_+ *)
#       let Int m = eval e1 in
#       let Int n = eval e2 in
#       Int (m + n)
#   | Binop (Divide, e1, e2) ->         (* R_/ *)
#       let Int m = eval e1 in
#       let Int n = eval e2 in
#       Int (m / n)
#   | Let (var, def, body) ->           (* R_let *)
#       let def' = eval def in
#       eval (subst body var def') ;;
Lines 7-8, characters 0-11:
7 | let Int n = eval e2 in
8 | Int (m + n)
Warning 8 [partial-match]: this pattern-matching is not exhaustive.
Here is an example of a case that is not matched:
(Var _|Binop (_, _, _)|Let (_, _, _))
Lines 6-8, characters 0-11:
6 | let Int m = eval e1 in
7 | let Int n = eval e2 in
8 | Int (m + n)
Warning 8 [partial-match]: this pattern-matching is not exhaustive.
Here is an example of a case that is not matched:
(Var _|Binop (_, _, _)|Let (_, _, _))
Lines 11-12, characters 0-11:
11 | let Int n = eval e2 in
```

```
12 | Int (m / n)
Warning 8 [partial-match]: this pattern-matching is not exhaustive.
Here is an example of a case that is not matched:
(Var _|Binop (_, _, _)|Let (_, _, _))
Lines 10-12, characters 0-11:
10 | let Int m = eval e1 in
11 | let Int n = eval e2 in
12 | Int (m / n)
Warning 8 [partial-match]: this pattern-matching is not exhaustive.
Here is an example of a case that is not matched:
(Var _|Binop (_, _, _)|Let (_, _, _))
val eval : expr -> expr = <fun>
```

Two problems jump out: First, violating the edict of intention, we've not provided information about what to do in cases where the arguments to an integer operator evaluate to something other than integers. These show up as "pattern-matching not exhaustive" warnings. Second, violating the edict of irredundancy, the code for binary operators is quite redundant. We'll solve both problems simultaneously by factoring out the redundancy into a function for evaluating binary operator expressions. We'll introduce another exception for reporting ill-formed expressions.

```
# exception UnboundVariable of string ;;
exception UnboundVariable of string
# exception IllFormed of string ;;
exception IllFormed of string

# let binopeval (op : binop) (v1 : expr) (v2 : expr)
#             : expr =
#   match op, v1, v2 with
#   | Plus, Int x1, Int x2 -> Int (x1 + x2)
#   | Plus, _, _ ->
#       raise (IllFormed "can't add non-integers")
#   | Divide, Int x1, Int x2 -> Int (x1 / x2)
#   | Divide, _, _ ->
#       raise (IllFormed "can't divide non-integers") ;;
val binopeval : binop -> expr -> expr -> expr = <fun>

# let rec eval (e : expr) : expr =
#   match e with
#   | Int _ -> e
#   | Var x -> raise (UnboundVariable x)
#   | Binop (op, e1, e2) ->
#       binopeval op (eval e1) (eval e2)
#   | Let (x, def, body) ->
#       eval (subst body x (eval def)) ;;
val eval : expr -> expr = <fun>
```

This function allows evaluating expressions in the language reflecting the semantics of those expressions.

```
# eval (Binop (Plus, Int 5, Int 10)) ;;
- : expr = Int 15
```

```
# eval (Let ("x", Int 3,
#           Let ("y", Int 5,
#                Binop (Divide, Var "x", Var "y")))) ;;
- : expr = Int 0
```

13.5 Problem section: Semantics of booleans and conditionals

Exercise 135

Augment the abstract syntax of the language to introduce boolean literals `true` and `false`. Add substitution semantics rules for the new constructs. Adjust the definitions of `subst` and `eval` to handle these new literals.

Exercise 136

Augment the abstract syntax of the language to add conditional expressions (`if` ⟨⟩ `then` ⟨⟩ `else` ⟨⟩). Add substitution semantics rules for the new construct. Adjust the definitions of `subst` and `eval` to handle conditionals.

13.6 Semantics of function application

We can extend our language further, by introducing (anonymous) functions and their application. We augment the language with two rules for function expressions and function applications as follows:

$$
\begin{array}{rcl}
\langle binop \rangle & ::= & \texttt{+} \mid \texttt{-} \mid \texttt{*} \mid \texttt{/} \\
\langle var \rangle & ::= & \texttt{x} \mid \texttt{y} \mid \texttt{z} \mid \cdots \\
\langle expr \rangle & ::= & \langle integer \rangle \\
& \mid & \langle var \rangle \\
& \mid & \langle expr_1 \rangle \ \langle binop \rangle \ \langle expr_2 \rangle \\
& \mid & \texttt{let} \ \langle var \rangle \ \texttt{=} \ \langle expr_{def} \rangle \ \texttt{in} \ \langle expr_{body} \rangle \\
& \mid & \texttt{fun} \ \langle var \rangle \ \texttt{->} \ \langle expr_{body} \rangle \\
& \mid & \langle expr_{fun} \rangle \ \langle expr_{arg} \rangle
\end{array}
$$

To complete the semantics for this language, we simply have to add rules for the evaluation of functions and applications.

The case of functions is especially simple. Functions are pending computations; they don't take effect until they are applied. So we can take functions to be values, that is, they self-evaluate.

$$\texttt{fun}\ x\ \texttt{->}\ B \Downarrow \texttt{fun}\ x\ \texttt{->}\ B \qquad (R_{fun})$$

All the work happens upon application. To evaluate an application, we must evaluate the function part to get the function to be applied and evaluate the argument part to get the argument's value, and then evaluate the body of the function, after substituting in the argument

for the variable bound by the function.

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

Exercise 137

Give glosses for these two rules R_{fun} and R_{app}, as was done for the previous rules R_+ and R_{let}.

Let's try an example:

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

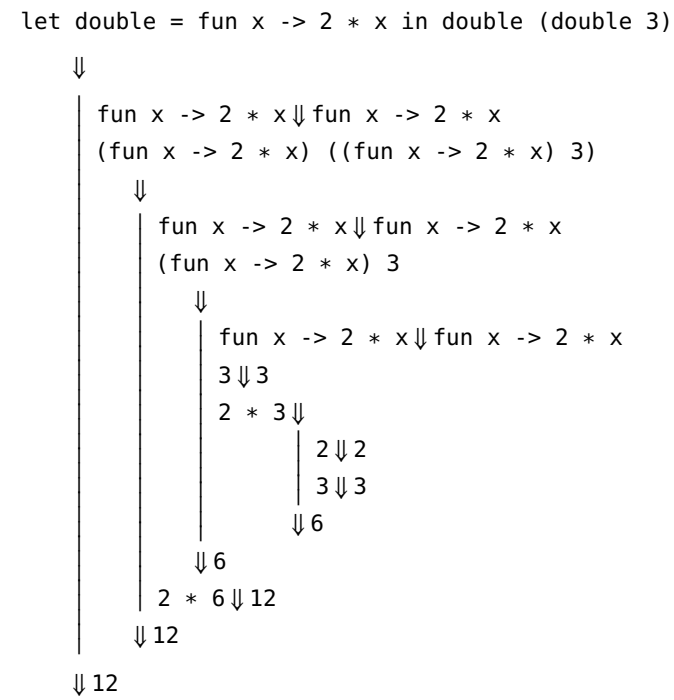
Intuitively, this should evaluate to 24. The derivation proceeds as follows:

```
(fun x -> x + x) (3 * 4)
    ⇓
    | (fun x -> x + x) ⇓ (fun x -> x + x)
    | 3 * 4 ⇓
    |       | 3 ⇓ 3
    |       | 4 ⇓ 4
    |       ⇓ 12
    | 12 + 12 ⇓
    |          | 12 ⇓ 12
    |          | 12 ⇓ 12
    |          ⇓ 24
    ⇓ 24
```

The combination of local naming and anonymous functions gives us the ability to give names to functions:

```
let double = fun x -> 2 * x in
double (double 3)
```

The derivations start getting a bit complicated:



Exercise 138

Carry out similar derivations for the following expressions:

1. `(fun x -> x + 2) 3`

2. ```
   let f = fun x -> x in
   f (f 5)
   ```

3. ```
   let square = fun x -> x * x in
   let y = 3 in
   square y
   ```

4. ```
   let id = fun x -> x in
   let square = fun x -> x * x in
   let y = 3 in
   id square y
   ```

13.6.1 More on capturing free variables

There is still a problem in our definition of substitution. Consider the following expression: `let f = fun x -> y in (fun y -> f 3) 1`. Intuitively speaking, this expression seems ill-formed; it defines a function `f` that makes use of an unbound variable `y` in its body. But using the definitions that we have given so far, we would have the
   ```

following derivation:

```
let f = fun x -> y in (fun y -> f 3) 1
    ⇓
    │ fun x -> y ⇓ fun x -> y
    │ (fun y -> (fun x -> y) 3) 1
    │    ⇓
    │    │ (fun y -> (fun x -> y) 3) ⇓ (fun y -> (fun x -> y) 3)
    │    │ 1 ⇓ 1
    │    │ (fun x -> 1) 3 ⇓
    │    │                │ fun x -> 1 ⇓ fun x -> 1
    │    │                │ 1 ⇓ 1
    │    │                │ 1 ⇓ 1
    │    │                ⇓ 1
    │    ⇓ 1
    ⇓ 1
```

The problem happens in the highlighted expression, where according to the R_{let} rule we should be evaluating `((fun y -> f 3) 1)`$[$`f` $\mapsto$ `fun x -> y`$]$, which according to our current understanding of substitution should be the highlighted `(fun y -> (fun x -> y) 3) 1`.

We're sneaking the `y` in `fun x -> y` inside the scope of the `fun y`. That's not kosher. And the OCaml interpreter seems to agree:

```
# let f = fun x -> y in (fun y -> f 3) 1 ;;
Line 1, characters 17-18:
1 | let f = fun x -> y in (fun y -> f 3) 1 ;;
                     ^
Error: Unbound value y
```

We need to change the definition of substitution to make sure that such VARIABLE CAPTURE doesn't occur. The following rules for substituting inside a function work by replacing the bound variable `y` with a new freshly minted variable, say `z`, that doesn't occur elsewhere, renaming all occurrences of `y` accordingly.

$$\texttt{(fun } x \texttt{ -> } P\texttt{)}[x \mapsto Q] = \texttt{fun } x \texttt{ -> } P$$

$$\texttt{(fun } y \texttt{ -> } P\texttt{)}[x \mapsto Q] = \texttt{fun } y \texttt{ -> } P[x \mapsto Q] \quad \text{where } x \not\equiv y \text{ and } y \notin FV(Q)$$

$$\texttt{(fun } y \texttt{ -> } P\texttt{)}[x \mapsto Q] = \texttt{fun } z \texttt{ -> } P[y \mapsto z][x \mapsto Q] \quad \text{where } x \not\equiv y \text{ and } y \in FV(Q) \text{ and } z \text{ is a fresh variable}$$

Exercise 139

Carry out the derivation for

```
let f = fun x -> y in (fun y -> f 3) 1
```

as above but with this updated definition of substitution. What happens at the step highlighted above?

Exercise 140

What should the corresponding rule or rules defining substitution on `let` $\cdots$ `in` $\cdots$ expressions be? That is, how should the following rule be completed? You'll want to think about how this construct reduces to function application in determining your answer.

$$(\texttt{let } y \texttt{ = } Q \texttt{ in } R)[x \mapsto P] = \cdots$$

Try to work out your answer before checking it with the full definition of substitution in Figure 13.4.

Exercise 141

Use the definition of the substitution operation above to determine the results of the following substitutions:

1. `(fun x -> x + x)`$[\texttt{x} \mapsto \texttt{3}]$
2. `(fun x -> y + x)`$[\texttt{x} \mapsto \texttt{3}]$
3. `(let x = y * y in x + x)`$[\texttt{x} \mapsto \texttt{3}]$
4. `(let x = y * y in x + x)`$[\texttt{y} \mapsto \texttt{3}]$

The implementation of substitution should be updated to handle this issue of avoiding the capture of free variables. The next two exercises do so.

Exercise 142

Write a function `free_vars : expr -> varspec Set.t` that returns a set of `varspecs` corresponding to the free variables in the expression as per Figure 13.3. (Recall the discussion of the OCaml library module `Set` in Section 12.8.)

Exercise 143

Revise the definition of the function `subst` from Section 13.4.1 to eliminate the problem of variable capture by implementing the set of rules given in Figure 13.4.

$$FV(\overline{m}) = \varnothing \qquad \text{(integers)} \tag{13.1}$$

$$FV(x) = \{x\} \qquad \text{(variables)} \tag{13.2}$$

$$FV(P \texttt{ + } Q) = FV(P) \cup FV(Q) \qquad \text{(and similarly for other binary operators)} \tag{13.3}$$

$$FV(P\ Q) = FV(P) \cup FV(Q) \qquad \text{(applications)} \tag{13.4}$$

$$FV(\texttt{fun } x \texttt{ -> } P) = FV(P) - \{x\} \qquad \text{(functions)} \tag{13.5}$$

$$FV(\texttt{let } x \texttt{ = } P \texttt{ in } Q) = (FV(Q) - \{x\}) \cup FV(P) \qquad \text{(binding)} \tag{13.6}$$

Figure 13.3: Definition of FV, the set of free variables in expressions for a functional language with naming and arithmetic.

$$\overline{m}[x \mapsto P] = \overline{m} \tag{13.7}$$

$$x[x \mapsto P] = P \tag{13.8}$$

$$y[x \mapsto P] = y \qquad \text{where } x \not\equiv y \tag{13.9}$$

$$(Q \texttt{ + } R)[x \mapsto P] = Q[x \mapsto P] \texttt{ + } R[x \mapsto P] \tag{13.10}$$

and similarly for other binary operators

$$Q\,R[x \mapsto P] = Q[x \mapsto P]\,R[x \mapsto P] \tag{13.11}$$

$$(\texttt{fun } x \texttt{ -> } Q)[x \mapsto P] = \texttt{fun } x \texttt{ -> } Q \tag{13.12}$$

$$(\texttt{fun } y \texttt{ -> } Q)[x \mapsto P] = \texttt{fun } y \texttt{ -> } Q[x \mapsto P] \tag{13.13}$$

where $x \not\equiv y$ and $y \notin FV(P)$

$$(\texttt{fun } y \texttt{ -> } Q)[x \mapsto P] = \texttt{fun } z \texttt{ -> } Q[y \mapsto z][x \mapsto P] \tag{13.14}$$

where $x \not\equiv y$ and $y \in FV(P)$ and z is a fresh variable

$$(\texttt{let } x \texttt{ = } Q \texttt{ in } R)[x \mapsto P] = \texttt{let } x \texttt{ = } Q[x \mapsto P] \texttt{ in } R \tag{13.15}$$

$$(\texttt{let } y \texttt{ = } Q \texttt{ in } R)[x \mapsto P] = \texttt{let } y \texttt{ = } Q[x \mapsto P] \texttt{ in } R[x \mapsto P] \tag{13.16}$$

where $x \not\equiv y$ and $y \notin FV(P)$

$$(\texttt{let } y \texttt{ = } Q \texttt{ in } R)[x \mapsto P] = \texttt{let } z \texttt{ = } Q[x \mapsto P] \texttt{ in } R[y \mapsto z][x \mapsto P] \tag{13.17}$$

where $x \not\equiv y$ and $y \in FV(P)$ and z is a fresh variable

Figure 13.4: Definition of substitution of expressions for variables in expressions for a functional language with naming and arithmetic.

13.7 Substitution semantics of recursion

You may observe that the rule for evaluating `let` ⟨⟩ `in` ⟨⟩ expressions doesn't allow for recursion. For instance, the Fibonacci example proceeds as follows:

```
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)
  | (fun n -> if n = 0 then 1 else n * f (n - 1)) 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)
    | 2 ⇓ 2
    | if 2 = 0 then 1 else 2 * f (2 - 1) ⇓ ???
    ⇓ ???
  ⇓ ???
```

The highlighted expression, `if 2 = 0 then 1 else 2 * f (2 - 1)`, eventually leads to an attempt to apply the unbound variable `f` to its argument `1`.

Occurrences of the name definiendum in the body are properly replaced with the definiens, but occurrences in the definiens itself are not. But what should those recursive occurrences of `f` be replaced *by*? It doesn't suffice simply to replace them with the definiens, as that still has a free occurrence of the definiendum. Rather, we'll replace them with their own recursive `let` construction, thereby allowing later occurrences to be handled as well. In the factorial example, we'll replace the free occurrence of `f` in the definiens by `let rec f = fun n -> if n = 0 then 1 else n * f (n - 1) in f`, that is, an expression that evaluates to whatever `f` evaluates to *in the context of the recursive definition itself.*

Thus the substitution semantics rule for `let rec`, subtly different from the `let` rule, will be as follows:

`let rec` x `=` D `in` $B \Downarrow$

| $D \Downarrow v_D$
| $B[x \mapsto v_D[x \mapsto$ `let rec` x `=` v_D `in` $x]] \Downarrow v_B$

$\Downarrow v_B$

(R_{letrec})

Continuing the factorial example above, we would substitute for `f` in the third line the expression `let rec f = fun n -> if n = 0 then 1 else n * f (n - 1) in f`, forming

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

Proceeding further, the final line becomes

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

which will (eventually) evaluate to 2.

Exercise 144

Thanklessly continue this derivation until it converges on the final result for the factorial of 2, viz., 2. Then thank your lucky stars that we have computers to do this kind of rote repetitive task for us.

We'll provide an alternative approach to semantics of recursion when we introduce environment semantics in Chapter 19.

❧

We defined a set of formal rules providing the meanings of OCaml expressions via simplifying substitutions of equals for equals, resulting finally in the values that most simply encapsulate the meanings of complex expressions.

An interpreter for a programming language (the object language) written in the same programming language (as metalanguage) – a METACIRCULAR INTERPRETER – provides another way of getting at the semantics of a language. In fact, the first semantics for the programming language LISP was given as a metacircular interpreter.

In both cases, we see the advantage of having a language with a small core, sprinkled liberally with syntactic sugar, since only the core need be given the formal treatment through rules or metacircular interpretation. The syntactic sugar can be translated out. For instance, although the metacircular interpreter that we started developing here does not handle the more compact function definition notation seen in

```
let f x = x + 1
```

this expression can be taken as syntactic sugar for (that is, a variant concrete syntax for the abstract syntax of) the expression

```
let f = fun x -> x + 1
```

which we already have defined formal rules to handle. In the case of a metacircular interpreter, we can imagine that the parser for the former expression will simply provide the abstract syntax of the latter.

Our exploration of rigorous semantics for programs will continue once the substitution approach starts to falter in the presence of state change and imperative programming.

Figure 13.5: Substitution semantics rules for evaluating expressions, for a functional language with naming and arithmetic.

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

$$\texttt{fun}\ x\ \texttt{->}\ B \Downarrow \texttt{fun}\ x\ \texttt{->}\ B \qquad (R_{fun})$$

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

$$\frac{P \Downarrow \overline{m} \quad Q \Downarrow \overline{n}}{P\ \texttt{/}\ Q \Downarrow \overline{\lfloor m/n \rfloor}} \qquad (R_{/})$$

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

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

$$\frac{D \Downarrow v_D \quad B[x \mapsto v_D[x \mapsto \texttt{let rec}\ x\ \texttt{=}\ v_D\ \texttt{in}\ x]] \Downarrow v_B}{\texttt{let rec}\ x\ \texttt{=}\ D\ \texttt{in}\ B \Downarrow v_B} \qquad (R_{letrec})$$

13.8 Supplementary material

- Lab 9: Substitution semantics