Chapter 6

First-Class Functions

We began Section 4 by observing the similarity between a `with` expression and a function definition applied immediately to a value. Specifically, we observed that

```
{with {x 5} {+ x 3}}
```

is essentially the same as

$$f(x) = x + 3; f(5)$$

Actually, that's not *quite* right: in the math equation above, we give the function a name, f, whereas there is no identifier named `f` anywhere in the WAE program. We can, however, rewrite the mathematical formulation as

$$f = \lambda(x).x + 3; f(5)$$

which we can then rewrite as

$$(\lambda(x)x + 3)(5)$$

to get rid of the unnecessary name (f).

That is, `with` effectively creates a new anonymous function and immediately applies it to a value. Because functions are useful in their own right, we may want to separate the act of function *declaration* or *definition* from *invocation* or *application* (indeed, we might want to apply the same function multiple times). That is what we will study now.

6.1 A Taxonomy of Functions

The translation of `with` into mathematical notation exploits two features of functions: the ability to create anonymous functions, and the ability to define functions anywhere in the program (in this case, in the function position of an application). Not every programming language offers one or both of these capabilities. There is, therefore, a standard taxonomy that governs these different features, which we can use when discussing what kind of functions a language provides:

first-order Functions are not values in the language. They can only be defined in a designated portion of the program, where they must be given names for use in the remainder of the program. The functions in F1WAE are of this nature, which explains the 1 in the name of the language.

higher-order Functions can return other functions as values.

first-class Functions are values with all the rights of other values. In particular, they can be supplied as the value of arguments to functions, returned by functions as answers, and stored in data structures.

We would like to extend F1WAE to have the full power of functions, to reflect the capability of Scheme. In fact, it will be easier to return to WAE and extend it with first-class functions.

6.2 Enriching the Language with Functions

To add functions to WAE, we must define their concrete and abstract syntax. First let's examine some concrete programs:

```
{{fun {x} {+ x 4}}
 5}
```

This program defines a function that adds `4` to its argument and immediately applies this function to `5`, resulting in the value `9`. This one

```
{with {double {fun {x} {+ x x}}}
  {+ {double 10}
     {double 5}}}
```

evaluates to `30`. The program defines a function, binds it to `double`, then uses that name twice in slightly different contexts (i.e., instantiates the formal parameter with different actual parameters).

From these examples, it should be clear that we must introduce two new kinds of expressions: function applications (as before), as well as (anonymous) function definitions. Here's the revised BNF corresponding to these examples:

```
<FWAE> ::= <num>
         | {+ <FWAE> <FWAE>}
         | {- <FWAE> <FWAE>}
         | {with {<id> <FWAE>} <FWAE>}
         | <id>
         | {fun {<id>} <FWAE>}
         | {<FWAE> <FWAE>}
```

Note that F1WAE did not have function definitions as part of the expression language, since the definitions were assumed to reside outside the expression being evaluated. In addition, notice that the function position of an application (the last BNF production) is now more general: instead of just the name of a function, programmers can write an arbitrary expression that must be evaluated to obtain the function to apply. The corresponding abstract syntax is:

(**define-type** FWAE
  [*num* (*n number?*)]
  [*add* (*lhs FWAE?*) (*rhs FWAE?*)]

[*sub* (*lhs FWAE?*) (*rhs FWAE?*)]
[*with* (*name symbol?*) (*named-expr FWAE?*) (*body FWAE?*)]
[*id* (*name symbol?*)]
[*fun* (*param symbol?*) (*body FWAE?*)]
[*app* (*fun-expr FWAE?*) (*arg-expr FWAE?*)])

To define our interpreter, we must think a little about what kinds of values it consumes and produces. Naturally, the interpreter consumes values of type FWAE. What does it produce? Clearly, a program that meets the WAE description must yield numbers. As we have seen above, some program that use functions and applications also evaluate to numbers. How about a program that consists solely of a function? That is, what is the value of the program

```
{fun {x} x}
```

? It clearly doesn't represent a number. It may be a function that, *when applied* to a numeric argument, produces a number, but it's not itself a number (if you think differently, you need to indicate which number it will be: 0? 1? 1729?). We instead realize from this that functions are also values that may be the result of a computation.

We could design an elaborate representation for function values, but for now, we'll remain modest. We'll let the function evaluate to its abstract syntax representation (i.e., a *fun* structure). (We will soon get more sophisticated than this.) For consistency, we'll also let numbers evaluate to *num* structures. Thus, the result of evaluating the program above would be

```
#(struct:fun x #(struct:id x))
```

Now we're ready to write the interpreter. We must pick a type for the value that *interp* returns. Since we've decided to represent function and number answers using the abstract syntax, it make sense to use FWAE, with the caveat that only two kinds of FWAE terms can appear in the output: numbers and functions. Our first interpreter will use explicit substitution, to offer a direct comparison with the corresponding WAE and F1WAE interpreters.

```
;; interp : FWAE → FWAE
;; evaluates FWAE expressions by reducing them to their corresponding values
;; return values are either num or fun

(define (interp expr)
  (type-case FWAE expr
    [num (n) expr]
    [add (l r) (num+ (interp l) (interp r))]
    [sub (l r) (num− (interp l) (interp r))]
    [with (bound-id named-expr bound-body)
          (interp (subst bound-body
                         bound-id
                         (interp named-expr)))]
    [id (v) (error 'interp "free identifier")]
```

```
[fun (bound-id bound-body)
     expr]
[app (fun-expr arg-expr)
      (local ([define fun-val (interp fun-expr)])
        (interp (subst (fun-body fun-val)
                       (fun-param fun-val)
                       (interp arg-expr))))])
```

(We made a small change to the rules for *add* and *sub*: they use *num+* and *num−* since *interp* now returns an FWAE. These auxilliary definitions are given in Section 6.7.)

The rule for a function says, simply, to return the function itself. (Notice the similarity to the rule for numbers!) That leaves only the rule for applications to study. This rule first evaluates the function position of an application. This is because that position may itself contain a complex expression that needs to be reduced to an actual function. For instance, in the expression

```
{{{fun {x} x}
  {fun {x} {+ x 5}}}
 3}
```

the outer function position consists of the application of the identity function to a function that adds five to its argument.

When evaluated, the function position had better reduce to a function value, not a number (or anything else). For now, we implicitly assume that the programs fed to the interpreter have no errors. (In Section X, we will expend a great deal of effort to identify programs that may contain such errors.) Given a function, we need to evaluate its body after having substituted the formal argument with its value. That's what the rest of the program does: evaluate the expression that will become the bound value, bind it using substitution, and then interpet the resulting expression. The last few lines are very similar to the code for `with`.

To understand this interpreter better, consider what it produces in response to evaluating the following term:

```
{with {x 3}
  {fun {y}
    {+ x y}}}
```

DrScheme prints the following:

```
#(struct:fun y #(struct:add #(struct:num 3) #(struct:id y)))
```

Notice that the `x` inside the function body has been replaced by `3` as a result of substitution, so the function has no references to `x` left in it.

Problem 6.2.1 *What induced the small change in the rules for add and sub? Explain, with an example, what would go wrong if we did not make this change.*

Problem 6.2.2 *Did you notice the small change in the interpretation of* `with`*?*

Problem 6.2.3 *What goes wrong if the interpreter fails to evaluate the function position (by invoking the interpreter on it)? Write a program and present the expected and actual results.*

6.3 Making `with` Redundant

Now that we have functions and function invocation as two distinct primitives, we can combine them to recover the behavior of `with` as a special case. Every time we encounter an expression of the form

```
{with {var named} body}
```

we can replace it with

```
{{fun {var} body}
 named}
```

and obtain the same effect. The result of this translation does not use `with`, so it can be evaluated by a more primitive interpreter: one for AE enriched with functions. A simple pre-processor that runs before the interpreter can perform this translation. We will assume the existence of such a pre-processor, and use the language FAE as our basis for subsequent exploration.

6.4 Implementing Functions using a Substitution Cache

As Section 5 described, our implementation will be more sprightly if we cache substitutions instead of performing them greedily. Thus, let us study how to adapt our interpreter.

First, we must include a definition of a substitution cache. The substitution cache associates identifiers with their values. Previously, the value had always been a number, but now our set of values is richer. We therefore use the following type, with the understanding that the value will always be a *num* or *fun*:

```
(define-type SubCache
  [mtSub]
  [aSub (name symbol?) (value FAE?) (sc SubCache?)])
```

Relative to this, the definition of *lookup* remains the same (only it now returns values of type FAE).

Our first attempt at a resulting interpreter is

```
(define (interp expr sc)
  (type-case FAE expr
    [num (n) expr]
    [add (l r) (num+ (interp l sc) (interp r sc))]
    [sub (l r) (num- (interp l sc) (interp r sc))]
    [id (v) (lookup v sc)]
    [fun (bound-id bound-body)
         expr]
    [app (fun-expr arg-expr)
         (local ([define fun-val (interp fun-expr sc)])
           (interp (fun-body fun-val)
                   (aSub (fun-param fun-val)
                         (interp arg-expr sc)
                         sc))))]))
```

When we run a battery of tests on this interpreter, we find that the expression

```
{with {x 3}
  {with {f {fun {y} {+ x y}}}
    {with {x 5}
      {f 4}}}}
```

evaluates to `9`. This should be surprising, because we seem to again have introduced dynamic scope! (Notice that the value of `x` depends on the context of the application of `f`, not its definition.)

To understand the problem better, let's return to this example, which we examined in the context of the substitution interpreter: the result of interpreting

```
{with {x 3}
  {fun {y}
    {+ x y}}}
```

in the substitution interpreter is

```
#(struct:fun y #(struct:add #(struct:num 3) #(struct:id y)))
```

That is, it had substituted the `x` with `3` in the procedure. But because we are deferring substitution, our representation for the procedure is just its text. As a result, the interpreter that employs a substitution cache instead evaluates the same term to

```
#(struct:fun y #(struct:add #(struct:id x) #(struct:id y)))
```

What happened to the substitution for its body?

The moral here is that, to properly defer substitution, the value of a function should be not only its text, but also the substitutions that were due to be performed on it. We therefore define a new datatype for the interpreter's return value that attaches the definition-time substitution cache to every function value:

```
(define-type FAE-Value
  [numV (n number?)]
  [closureV (param symbol?)
            (body FAE?)
            (sc SubCache?)])
```

Accordingly, we change the rule for *fun* in the interpreter to

```
[fun (bound-id bound-body)
     (closureV bound-id bound-body sc)]
```

We call this constructed value a *closure* because it "closes" the function body over the substitutions that are waiting to occur.

When the interpreter encounters a function application, it must ensure that the function's pending substitutions aren't forgotten. It must, however, ignore the substitutions pending at the location of the *invocation*, for that is precisely what led us to dynamic instead of static scope. It must instead use the substitutions

of the invocation location to convert the function and argument into values, hope that the function expression evaluated to a closure, then proceed with evaluating the body employing the substitution cache in the closure.

```
[app (fun-expr arg-expr)
     (local ([define fun-val (interp fun-expr sc)])
       (interp (closureV-body fun-val)
               (aSub (closureV-param fun-val)
                     (interp arg-expr sc)
                     (closureV-sc fun-val)))))]
```

That is, having evaluated *fun-expr* to yield *fun-val*, we obtain not only the actual function body from *fun-val*'s closure record but also the substitution cache stored within it. Crucially, while we evaluate *arg-expr* in *sc*, the substitution cache active at the invocation location, we evaluate the function's body *in its "remembered" substitution cache*. Once again, the content of this boxed expression determines the difference between static and dynamic scope. Figure 6.1 presents the complete interpreter.

Problem 6.4.1 *This interpreter does not check whether the function position evaluated to a closure value. Modify the interpreter to check and, if the expression fails to yield a closure, report an error.*

Problem 6.4.2 *Suppose we explicitly implemented* `with` *in the interpreter. Given that* `with` *is just a shorthand for creating and applying a closure, would the changes we made to closure creation and function application have an effect on* `with` *too?*

Problem 6.4.3 *Define a caching interpreter for a* lazy *language with first-class functions.*

6.5 Some Perspective on Scope

The example above that demonstrated the problem with our caching interpreter might not be a very convincing demonstration of the value of static scope. Indeed, you might be tempted to say, "If you know `x` is `3`, why not just use `3` instead of `x` inside the procedure declaration? That would avoid this problem entirely!" That's a legitimate response for that particular example, which was however meant only to *demonstrate the problem*, not to *motivate the need for the solution*. Let's now consider two examples that do the latter.

6.5.1 Differentiation

First, let's look at implementing (a simple version of) numeric differentiation in Scheme. The program is

```
(define H 0.0001)

(define (d/dx f)
  (lambda (x)
    (/ (- (f (+ x H)) (f x))
       H)))
```

In this example, in the algebraic expression, the identifier f is free relative to the inner function. However, we cannot do what we proposed earlier, namely to substitute the free variable with its value; this is because we don't know what values f will hold during execution, and in particular f will likely be bound to several different values over the course of the program's lifetime. If we run the inner procedure under dynamic scope, one of two things will happen: either the identifier f will not be bound to any value in the context of use, resulting in an unbound identifier error, or the procedure will use whatever f is bound to, which almost certainly will not correspond to the value supplied to *d/dx*. That is, in a hypothetical dynamically-scoped Scheme, you would get

$>$ (**define** *diff-of-square* (*d/dx* (**lambda** (*x*) ($*$ *x* *x*))))
$>$ (*diff-of-square* 10)
reference to undefined identifier: f
$>$ (**define** *f* 'greg)
$>$ (*diff-of-square* 10)
procedure application: expected procedure, given: greg; arguments were: 10.0001
$>$ (**define** *f* *sqrt*)
$>$ (*diff-of-square* 10)
0.15811348772487577

That is, f assumes whatever value it has at the point of *use*, ignoring the value given to it at the inner procedure's *definition*. In contrast, what we really get from Scheme is

$>$ (*diff-of-square* 10)
20.000099999890608 ;; approximately $10 \times 2 = 20$

6.5.2 Callbacks

Let's consider another example, this one from Java. This program implements a *callback*, which is a common programming pattern employed in programming GUIs. In this instance, the callback is an object installed in a button; when the user presses the button, the GUI system invokes the callback, which brings up a message box displaying the number of times the user has pressed the button. This powerful paradigm lets the designer of the GUI system provide a generic library of graphical objects independent of the behavior each client wants to associate with that object.

```
// GUI library code
public class JButton {
    public void whenPressed(ActionEvent e) {
        for (int i = 0; i < listeners.length; ++i)
            listeners[i].actionPerformed(e);
    }
}

// User customization
public class GUIApp {
    private int count = 0;
```

```
    public class ButtonCallback implements ActionListener {
        public void actionPerformed(ActionEvent e) {
            count = count + 1;
            JOptionPane.showMessageDialog(null,
                                          "Callback was invoked " +
                                          count + " times!");
        }
    }

    public Component createComponents() {
        JButton button = new JButton("Click me!");
        button.addActionListener(new ButtonCallback());
        return button;
    }
}
```

Stripped to its essence, the callback code is really no different from

```
;; GUI library code
(define (button callback)
  (local [(define (sleep-loop)
            (when button-pressed
              (begin
                (callback)
                (sleep-loop))))]
    (sleep-loop)))

;; User customization
(local [(define count 0)
        (define (my-callback)
          (begin
            (set! count (add1 count)) ;; increment counter
            (message-box
              (string-append "Callback was invoked "
                             (number→string count)
                             " times!"))))]
  (button my-callback))
```

That is, a callback is just a function passed to the GUI toolbox, which the toolbox invokes when it has an argument. But note that in the definition of *my-callback* (or `ButtonCallback`), the identifier *count* is not bound within the function (or object) itself. That is, it is *free* in the function. Therefore, whether it is scoped statically or dynamically makes a huge difference!

How do we want our callback to behave? Naturally, as the users of the GUI toolbox, we would be very upset if, the first time the user clicked on the button, the system halted with the message

```
error: identifier ‘count’ not bound
```

The bigger picture is this. As programmers, we hope that other people will use our functions, perhaps even in fantastic contexts that we cannot even imagine. Unfortunately, that means we can't possibly know what the values of identifiers will be at the location of use, or whether they will even be bound. If we must rely on the locus of use, we will produce highly fragile programs: they will be useful only in very limited contexts, and their behavior will be unpredictable everywhere else.

Static scoping avoids this fear. In a language with static scope, the programmer has full power over choosing from the definition and use scopes. By default, free identifiers get their values from the definition scope. If the programmer wants to rely on a value from the use scope, they simply make the corresponding identifier a parameter. This has the added advantage of making very explicit in the function's interface which values from the use scope it relies on.

Dynamic scoping is primarily interesting as a historical mistake: it was in the earliest versions of Lisp, and persisted for well over a decade. Scheme was created as an experimental language in part to experiment with static scope. This was such a good idea that eventually, even Common Lisp adopted static scope. Most modern languages are statically scoped, but sometimes they make the mistake of recapitulating this phylogeny. So-called "scripting" languages, in particular, often make the mistake of implementing dynamic scope (or the lesser mistake of just failing to create closures), and must go through multiple iterations before they eventually implement static scope correctly.

6.6 Eagerness and Laziness

Recall that a lazy evaluator was one that did not reduce the named-expression of a `with` to a value at the time of binding it to an identifier. What is the corresponding notion of laziness in the presence of functions? Let's look at an example: in a lazy evaluator,

```
{with {x {+ 3 3}}
  {+ x x}}
```

would first reduce to

```
{+ {+ 3 3} {+ 3 3}}
```

But based on what we've just said in section 6.3 about reducing `with` to procedure application, the treatment of procedure arguments should match that of the named expression in a `with`. Therefore, a lazy language with procedures is one that does not reduce its argument to a value until necessary in the body. The following sequence of reduction rules illustrates this:

```
  {{fun {x} {+ x x}}
   {+ 3 3}}
= {+ {+ 3 3} {+ 3 3}}
= {+ 6 {+ 3 3}}
= {+ 6 6}
= 12
```

which is just an example of the `with` translation described above; a slightly more complex example is

```
  {with {double {fun {x} {+ x x}}}
    {double {double 5}}}
= {{fun {x} {+ x x}}
   {{fun {x} {+ x x}}
    5}}
= {{fun {x} {+ x x}}
   {+ 5 5}}
= {+ {+ 5 5} {+ 5 5}}
= {+ 10 {+ 5 5}}
= {+ 10 10}
= 20
```

What do the corresponding reductions look like in an eager regime? Are there significant differences between the two?

6.7 Helper Functions

The auxiliary functions *num*+ and *num*− operate on *num*s (as opposed to *number*s). We define them as follows:

```
;; num+ : num num → num
(define (num+ n1 n2)
  (num (+ (num-n n1) (num-n n2))))

;; num- : num num → num
(define (num- n1 n2)
  (num (- (num-n n1) (num-n n2))))
```

```
(define-type FAE
  [num (n number?)]
  [add (lhs FAE?) (rhs FAE?)]
  [sub (lhs FAE?) (rhs FAE?)]
  [id (name symbol?)]
  [fun (param symbol?) (body FAE?)]
  [app (fun-expr FAE?) (arg-expr FAE?)])

;; num+ : numV numV ⟶ numV

(define (num+ n1 n2)
  (numV (+ (numV-n n1) (numV-n n2))))

;; num- : numV numV ⟶ numV

(define (num− n1 n2)
  (numV (− (numV-n n1) (numV-n n2))))

(define-type FAE-Value
  [numV (n number?)]
  [closureV (param symbol?)
            (body FAE?)
            (sc SubCache?)])

(define-type SubCache
  [mtSub]
  [aSub (name symbol?) (value FAE-Value?) (sc SubCache?)])

;; lookup : symbol SubCache → FAE-Value

;; interp : FAE SubCache → FAE-Value
(define (interp expr sc)
  (type-case FAE expr
    [num (n) (numV n)]
    [add (l r) (num+ (interp l sc) (interp r sc))]
    [sub (l r) (num− (interp l sc) (interp r sc))]
    [id (v) (lookup v sc)]
    [fun (bound-id bound-body)
         (closureV bound-id bound-body sc)]
    [app (fun-expr arg-expr)
         (local ([define fun-val (interp fun-expr sc)])
           (interp (closureV-body fun-val)
                   (aSub (closureV-param fun-val)
                         (interp arg-expr sc)
                         (closureV-sc fun-val))))]))
```

Figure 6.1: First-Class Functions with Cached Substitutions