First-Class Functions

Shriram Krishnamurthi

2003-09-10

We began our previous lecture under the guise of avoiding redundancy, but as the notes demonstrated, that was really just a cover for discussing substitution. Much of the time, simply being able to name an expression isn’t enough: the expression’s value is going to depend on the context of its use. That means the expression needs to be parameterized; that is, it must be a function.

Dissecting a with expression is a useful exercise in helping us design functions. Consider the program

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

In this program, the expression `{+ x 3}` is parameterized over the value of `x`. In that sense, it’s just like a function definition: in mathematical notation, we might write

\[ f(x) = x + 3 \]

Actually, that’s not quite right: in the math equation above, we gave the function a name, `f`, whereas there is no identifier named `f` anywhere in the WAE program above. We’ll return to that point in a moment.

Anyway, having named and defined `f`, what do we do with it? The WAE program introduces `x` and then immediately binds it to `5`. The way we bind a function’s argument to a value is to apply it. Thus, it is as if we wrote

\[ f(x) = x + 3; f(5) \]

Given that \( f \) has no use other than being applied to `5`, it seems rather pointless to give it a name at all. Indeed, using the `lambda` notation we have learned for defining anonymous functions, we can write this better:

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

is the same as

\[ (\lambda(x)x + 3)(5) \]

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.

1 About Functions

We are ready to add functions; we will augment the base language WAE with functions to define FWAE. We will assume that, as in Scheme, functions can occur anywhere a value can occur: that is, functions are first class. There is a rudimentary taxonomy whose names you will find handy:

first order Functions cannot be returned as values by other functions. Typically, this means they also cannot be stored in data structures.

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.
2 Enriching the Language with Functions

To add functions to WAE, we must define their concrete and abstract syntax. First, we must determine how many new kinds of terms to add. For simplicity, we’ll assume all functions take exactly one argument. Here’s an example program:

```
{{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.

Consider this example:

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

This evaluates to 30. The program defines a function, binds it to double, then uses that name twice in slightly different contexts (i.e., they instantiate the parameter differently). The x in {fun {x} ...} is called the formal parameter while the 10 and 5 passed into x are called actual parameters.

Studying these examples and the preceding prose, it’s clear that we need to introduce two new kinds of expressions: function definitions and function applications. We extend the concrete syntax to include these two new expressions:

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

We also augment the abstract syntax’s datatype accordingly.

```
(define-datatype FWAE FWAE?
  [num (n number?)]
  [add (lhs FWAE?) (rhs FWAE?)]
  [sub (lhs FWAE?) (rhs FWAE?)]
  [id (name symbol?)]
  [with (name symbol?) (named-expr FWAE?) (body FWAE?)]
  [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 has a function alone? 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?). Rather, it’s just a function: they are also values!

We could design an elaborate representation for function values, but for now, we’ll remain modest. Just as the result of evaluating a number is an instance of the corresponding num structure, so we will let the result of a function be the corresponding fun structure. (In a lecture or so, we will get more sophisticated than this.) Thus, the result of evaluating the program above might be

```
#{(struct:fun x #{(struct:id x)})}
```
or some such internal representation.

Now we’re ready to write the interpreter. We’ll call it interp rather than calc, since our language has grown up past basic arithmetic. The rules present in the WAE interpreter remain the same, so we can focus on the two new rules. We do need to pick a type for the value that interp returns. We use FWAE, with the caveat that only two kinds of FWAE terms can appear in the output: numbers and functions.

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

(define (interp expr)
  (cases 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
                (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-arg fun-val)
          (interp fun-expr)))))])

(We made a slight change to the rules for add and sub: they use num+ and num− since interp now returns an FWAE. Also, the functions fun-arg and fun-body select the fields of a fun expression. We define these auxiliary functions in the Appendix.)

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 function position consists of the application of the identity function to a function that adds five to its argument. If the interpreter fails to evaluate the function position, what goes wrong?

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. (Later in the course, we will spend about a month simply addressing the question of what to do if they do have 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 interp 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.
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.

4 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 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}} {double {double 5}}}

\= {{fun {x} {+ x x}} {double {double 5}}}

\= {{fun {x} {+ x x}} {+ 5 5}}

\= {+ {+ 5 5} {+ 5 5}}

\= {+ 10 (+ 5 5)}

\= {+ 10 10}

\= 20

Not much has changed in our comparison between eagerness and laziness, but things might in the near future.
Appendix

The auxiliary functions $\text{num}+$ and $\text{num} -$ operate on numbers. We define them as follows:

\[
\text{fun-arg} : \text{FWAE} \ [\text{fun}] \rightarrow \text{symbol}
\]
\[
(\text{define} \ (\text{fun-arg} \ \text{expr})
(\text{cases} \ \text{FWAE} \ \text{expr})
  \ [\text{fun} \ (\text{name} \ \text{body}) \ \text{name}]
  \ [\text{else} \ (\text{error} \ '\text{fun-arg} \ "\text{not a fun}"))]])
\]

\[
\text{fun-body} : \text{FWAE} \ [\text{fun}] \rightarrow \text{FWAE}
\]
\[
(\text{define} \ (\text{fun-body} \ \text{expr})
(\text{cases} \ \text{FWAE} \ \text{expr})
  \ [\text{fun} \ (\text{name} \ \text{body}) \ \text{body}]
  \ [\text{else} \ (\text{error} \ '\text{fun-body} \ "\text{not a fun}"))]])
\]

\[
\text{num-n} : \text{FWAE} \ [\text{num}] \rightarrow \text{number}
\]
\[
(\text{define} \ (\text{num-n} \ \text{expr})
(\text{cases} \ \text{FWAE} \ \text{expr})
  \ [\text{num} \ (\text{n}) \ \text{n}]
  \ [\text{else} \ (\text{error} \ '\text{num-n} \ "\text{not a number}"))]])
\]

\[
\text{num+} : \text{num} \ \text{num} -> \text{num}
\]
\[
(\text{define} \ (\text{num+} \ n1 \ n2)
  (\text{num} \ (+ \ (\text{num-n} \ n1) \ (\text{num-n} \ n2))))
\]

\[
\text{num-} : \text{num} \ \text{num} \rightarrow \text{num}
\]
\[
(\text{define} \ (\text{num-} \ n1 \ n2)
  (\text{num} \ (- \ (\text{num-n} \ n1) \ (\text{num-n} \ n2))))
\]