Implementing First-Class Functions by Caching Substitution

sk and dtucker

2002-09-16

In our previous notes, we introduced first-class functions into the language, and described one implementation strategy: the use of substitution. We then commented that substitution can be mighty inefficient, so we’d like to use a caching strategy instead. We had also presented an implementation that uses the cache.

Let’s consider what the interpreter does with the following program:

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

Trace through the interpreter to determine its value. You ought to get 9.

Is this the correct answer? Well, it’s certainly possible that this is correct—after all, it’s what the interpreter returns, and this could well be the interpreter for some language.

But we do have a better way of answering this question. Recall that the interpreter was using the cache to conduct substitution more efficiently. As with any cache, we hope that its application only improves performance—not change the program’s behavior! Thus, our “reference implementation” is the one that performs explicit substitution. If we want to know what the value of the program really “is”, we need to return to that implementation.

What does the substitution-based interpreter return for this program? It says the answer is 7. So we have to regard our caching interpreter as being buggy.

While the caching interpreter is clearly buggy relative to substitution, which it was supposed to represent, let’s think for a moment about what we, as the human programmer, would want this program to evaluate to. It produces 7 because x gets its value from the with clause in whose scope the function is defined. The value 9 comes from giving x its value through the with clause in whose scope the function is used. A priori, is one “better” than the other? Before we tackle that, let’s introduce some terminology to make it easier to refer to these two behaviors.

Definition 1 (Static Scope) In a language with static scope, each identifier gets its value from the scope of its definition, not its use.

Definition 2 (Dynamic Scope) In a language with dynamic scope, each identifier gets its value from the scope of its use, not its definition.

1 Some Perspective on Scope

To address which is better, let’s step outside Scheme and interpreters for a moment and consider the following fragment of Java code. 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)
           ...

1
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. Today, most languages are statically scoped, but they sometimes take their time getting there: it’s shocking to see just how many initial versions of languages (typically “scripting” languages) get scope horribly mixed up. Even some very popular scripting languages still suffer from this.

2 Fixing the Interpreter

Let’s return to our interpreter. Our choice of static over dynamic scope has the benefit of confirming that the substituting interpreter did the right thing, so all we need do is make the caching interpreter be a correct transformation of it. In fact, it’s clear that we need to focus our attention on two lines: that for function creation and that for function application. These currently read:

\[
\begin{align*}
&\text{fun} \quad (\text{bound-id bound-body}) \\
&\text{expr} \\
&\text{app} \quad (\text{fun-expr arg-expr}) \\
&\text{local} \quad (\text{define fun-val } (\text{interp fun-expr sc})) \\
&\text{interp} \quad (\text{fun-body fun-val}) \\
&\text{aSub} \quad (\text{fun-arg fun-val}) \\
&\text{interp} \quad (\text{arg-expr sc}) \\
&\text{sc}
\end{align*}
\]

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

{with {x 3}
 {fun {y}
  (+ x y)}}

DrScheme prints

\#(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. 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. To do so, we define a new datatype for the interpreter’s return value that attaches the definition-time substitution cache to every function value:

\[\text{(define-datatype FWA-value FWA-value?)}\]

Accordingly, we change the rule for \text{fun} in the interpreter to

\[\begin{align*}
&\text{fun} \quad (\text{param body}) \\
&\text{(closureV param body sc)}
\end{align*}\]
We call this a closure because it “closes” the function body over the substitutions that are waiting to occur.

Now, when we are ready to apply the function, we have to make sure that its pending substitutions aren’t forgotten. Indeed, we aren’t interested in the substitutions at the location of invocation, because those are exactly what led us to dynamic scope instead of static scope. So we use the substitutions of the invocation location to convert the function and argument into values, hope that the function location yielded a closure, then proceed with the body but also the substitution cache from the closure.

```scheme
[app (fun-expr arg-expr)
 (local ([define fun-val (interp fun-expr sc)])
  [define arg-val (interp arg-expr sc)])
 (cases FWA-value fun-val
  [closureV (cl-param cl-body cl-cache) (interp cl-body
    (aSub cl-param
      arg-val
      cl-cache))]
  [else (error "interp can only apply functions")]))])
```

That is, having evaluated `fun-expr` to yield `fun-val`, we obtain 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 (stored) substitution cache:

```scheme
(interp cl-body
 (aSub cl-param
   arg-val
   cl-cache))
```

In the small example above, for instance, `cl-cache` would have recorded that `x` needs to be replaced with 3, so when the body actually refers to `x`, the substitution finally happens.

**Problem**

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