1 How Efficient is Substitution?

Let’s examine the process of interpreting the following small program. Consider the following sequence of evaluation steps:

\[
\begin{align*}
\{\text{with } \{x 3\} \\
\text{with } \{y 4\} \\
\{\text{with } \{z 5\} \\
\{+ x \{+ y z\}\}\}\} \\
= \{\text{with } \{y 4\} \\
\{\text{with } \{z 5\} \\
\{+ 3 \{+ y z\}\}\}\} \\
= \{\text{with } \{z 5\} \\
\{+ 3 \{+ 4 z\}\}\} \\
= \{+ 3 \{+ 4 5\}\}
\end{align*}
\]

at which point it reduces to an arithmetic problem. To reduce it, though, the interpreter had to apply substitution three times: once for each with. This is slow! How slow? Well, if the program has length \(n\) (measured in abstract syntax tree nodes), then each substitution sweeps the rest of the program once, making the complexity of this interpreter at least \(O(n^2)\).\(^1\) That seems rather wasteful; surely we can do better.

How do we avoid redundancy in computations? The classic computer science technique is to use caching. Caches are not only low-level units of hardware; anything that stores results of prior or potential computations can be called a cache. In this case, we want a cache of substitutions.

Concretely, here’s the idea. Initially, we have no substitutions to perform, so the cache is empty. Every time we encounter a substitution (in the form of a with or application), we augment the cache with one more entry, recording the identifier’s name and the value (if eager) or expression (if lazy) it should eventually be substituted with. We continue to evaluate without actually performing the substitution.

This strategy breaks a key invariant we had established earlier, which is that any identifier the interpreter encounters is of necessity free, for had it been bound, it would have been replaced by substitution. Because we’re no longer using substitution, we are indeed likely to encounter identifiers during interpretation. How do we handle them? We must replace them with their bound values by consulting the substitution cache.

2 The Substitution Cache

Let’s provide a data definition for a substitution cache:

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

\[
\text{[mtSub]}
\]

\[
\text{[aSub (name symbol?) (value FWAE?) (sc SubCache?)]}
\]

where SubCache stands for a “substitution cache”. A SubCache has two forms: it’s either empty (mtSub\(^2\)) or non-empty (represented by an aSub structure). The latter contains a reference to the rest of the cache in its third field.

\(^1\)Puzzle: Can it be worse than that? Prove it either way.

\(^2\)“Empty sub”—get it!
The interpreter obviously needs to consume a substitution cache in addition to the expression to interpret. Therefore, its contract becomes

;; interp : FWAE SubCache → FWAE

It will need a helper function that looks up the value of identifiers in the cache. Its code is:

;; lookup : symbol SubCache → FWAE

\[
\text{(define (lookup name sc)} \\
\text{(cases SubCache sc)} \\
\hspace{1cm} [\text{mtSub () (error 'lookup "no binding for identifier")}] \\
\hspace{1cm} [\text{aSub (bound-name bound-value rest-sc)}] \\
\hspace{1cm} \hspace{1cm} (\text{if (symbol=? bound-name name)} \\
\hspace{1cm} \hspace{1cm} \hspace{1cm} bound-value \\
\hspace{1cm} \hspace{1cm} \hspace{1cm} (lookup name rest-sc))])
\]

With that fanfare, we can now present the interpreter:

\[
\text{(define (interp expr sc)} \\
\text{(cases FWAE expr)} \\
\hspace{1cm} [\text{num (n) expr]} \\
\hspace{1cm} \hspace{1cm} [\text{add (l r) (num+ (interp l sc) (interp r sc))}] \\
\hspace{1cm} \hspace{1cm} [\text{sub (l r) (num- (interp l sc) (interp r sc))}] \\
\hspace{1cm} \hspace{1cm} [\text{with (bound-id named-expr bound-body)} \\
\hspace{1cm} \hspace{1cm} \hspace{1cm} (interp bound-body \\
\hspace{1cm} \hspace{1cm} \hspace{1cm} (aSub bound-id \\
\hspace{1cm} \hspace{1cm} \hspace{1cm} \hspace{1cm} (interp named-expr sc) \\
\hspace{1cm} \hspace{1cm} \hspace{1cm} \hspace{1cm} \hspace{1cm} sc)))] \\
\hspace{1cm} \hspace{1cm} [\text{id (v) (lookup v sc)}] \\
\hspace{1cm} \hspace{1cm} [\text{fun (bound-id bound-body)} \\
\hspace{1cm} \hspace{1cm} \hspace{1cm} \hspace{1cm} \hspace{1cm} expr] \\
\hspace{1cm} \hspace{1cm} [\text{app (fun-expr arg-expr)}] \\
\hspace{1cm} \hspace{1cm} \hspace{1cm} \hspace{1cm} \hspace{1cm} (\text{local ([define fun-val (interp fun-expr sc)])}) \\
\hspace{1cm} \hspace{1cm} \hspace{1cm} \hspace{1cm} \hspace{1cm} (interp (fun-body fun-val) \\
\hspace{1cm} \hspace{1cm} \hspace{1cm} \hspace{1cm} \hspace{1cm} (aSub (fun-arg fun-val) \\
\hspace{1cm} \hspace{1cm} \hspace{1cm} \hspace{1cm} \hspace{1cm} \hspace{1cm} (interp arg-expr sc) \\
\hspace{1cm} \hspace{1cm} \hspace{1cm} \hspace{1cm} \hspace{1cm} \hspace{1cm} \hspace{1cm} sc)))]
\]

To make sure this is correct, we recommend that you first study its behavior on programs that have no identifiers—i.e., verify that the arithmetic rules do the right thing—and only then proceed to the rules that involve identifiers.

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

\[
\{\text{with \{x 3\}} \\
\hspace{1cm} \{\text{with \{f \{fun \{y\} (+ x y)\} \{} \\
\hspace{1cm} \hspace{1cm} \{\text{with \{x 5\}} \\
\hspace{1cm} \hspace{1cm} \hspace{1cm} \{f 4\})\}\} \\
\}
\]

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 \( \text{with} \) clause in whose scope the function is defined. The value 9 comes from giving \( x \) its value through the \( \text{with} \) 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.*

### 3 Some Perspective on Scope

The example above 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 illustrate its value.

#### 3.1 Differentiation

First, let’s look at the differentiation example we saw in the Scheme lectures. To wit, the program was

\[
\begin{align*}
&\text{(define } H 0.0001) \\
&(\text{define } (d/dx f) \\
&\quad (\text{lambda } (x) \\
&\quad \quad (/ \ (- \ (f \ (+ \ x \ H)) \ (f \ x)) \ H)))
\end{align*}
\]

In this example, the free variable in the inner function—the \((\text{lambda } (x) \cdots)\)—is \( f \). 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)
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
```

3
3.2 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.

```java
// 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

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

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

## 4 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 interpeter be a correct reimplementation of it. In fact, it’s clear that we need to focus our attention on two rules: those for function creation and function application. These currently read:

```
[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-arg fun-val)
      (interp arg-expr sc) sc)))]
```

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. We therefore define a new datatype for the interpreter’s return
value that attaches the definition-time substitution cache to every function value:

\[
(\text{define-datatype} \ FWA-\text{value} \ FWA-\text{value}?)
\begin{align*}
&[\text{numV} (n \text{ number}?) ] \\
&[\text{closureV} (\text{param} \text{ symbol}?)
\begin{align*}
&(\text{body} \ FWAE?) \\
&(\text{cache} \text{ SubCache}?) \]
\end{align*}
\]

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

\[
[\text{fun} (\text{param} \text{ body})
\begin{align*}
&(\text{closureV} \text{ param} \text{ body sc})
\end{align*}
\]

We call this a \textit{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 evaluating the body
according to the substitution cache from the closure.

\[
[\text{app} \ (\text{fun-expr} \ \text{arg-expr})
\begin{align*}
&(\text{local} \ ((\text{define} \text{ fun-val} (\text{interp} \text{ fun-expr} \text{ sc})))
\begin{align*}
&(\text{define} \text{ arg-val} (\text{interp} \text{ arg-expr sc})))
\end{align*}
\begin{align*}
&(\text{cases} \ FWA-\text{value} \text{ fun-val}
\begin{align*}
&(\text{closureV} (\text{cl-param cl-body cl-cache}) (\text{interp} \text{ cl-body}
\begin{align*}
&(\text{aSub} \text{ cl-param}
\begin{align*}
&(\text{arg-val}
\begin{align*}
&(\text{cl-cache})
\end{align*}
\end{align*}
\end{align*}
\end{align*}
\end{align*}
\end{align*}
\begin{align*}
&(\text{else} \ (\text{error} \ "\text{interp} \ "\text{can only apply functions})))))
\end{align*}
\end{align*}
\end{align*}
\end{align*}
\end{align*}
\end{align*}
\]

That is, having evaluated \texttt{fun-expr} to yield \texttt{fun-val}, we obtain not only the actual function body from \texttt{fun-val}’s closure
record but also the substitution cache stored within it. Crucially, while we evaluate \texttt{arg-expr} in \texttt{sc}, the substitution
cache active at the invocation location, we evaluate the function’s body in its “remembered” substitution cache:

\[
(\text{interp} \text{ cl-body}
\begin{align*}
&(\text{aSub} \text{ cl-param}
\begin{align*}
&(\text{arg-val}
\begin{align*}
&(\text{cl-cache})
\end{align*}
\end{align*}
\end{align*}
\end{align*}
\]

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

Puzzles

1. Define a caching interpreter for a \textit{lazy} language with first-class functions.

2. How come we never seem to \textit{undo} additions to the substitution cache? Doesn’t this run the risk that one
substitution might override another in a way that destroys static scoping?

3. Why did we not update the code for the \textit{with} clause? Given that \textit{with} is just a shorthand for creating and
applying a closure, shouldn’t the changes we made to closure creation and function application have an effect
on \textit{with} too?

4. Our implementation of \texttt{lookup} can take time linear in the size of the program to find some identifiers. Therefore,
it’s not clear we have really solved the time-complexity problem that motivated the use of a substitution cache.
We could address this by using a better data structure and algorithm for \texttt{lookup}: a hash table, say. What changes
do we need to make if we use a hash table? (\textbf{Hint:} This is closely tied to puzzle 2 above!)