Chapter 5

Caching Substitution

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

```markdown
{with {x 3}
  {with {y 4}
    {with {z 5}
      {+ x {+ y z}}}}}
= {with {y 4}
  {with {z 5}
    {+ 3 {+ y z}}}}
= {with {z 5}
  {+ 3 {+ 4 z}}}
= {+ 3 {+ 4 5}}
```

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 size \( n \) (measured in abstract syntax tree nodes), then each substitution sweeps the rest of the program once, making the complexity of this interpreter at least \( O(n^2) \). That seems rather wasteful; surely we can do better.

How do we avoid redundancy in computations? The classic computer science technique is to employ a cache. The term “cache” doesn’t only apply to 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which is that any identifier the interpreter encounters is of necessity free, for had it been bound, it would have been replaced by substitution.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 will encounter bound identifiers during interpretation. How do we handle them? We must replace them with their bindings by consulting the substitution cache.

**Problem 5.0.1** Can the complexity of substitution be worse than \( O(n^2) \)?


## 5.1 The Substitution Cache

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

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

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

The interpreter obviously needs to consume a substitution cache in addition to the expression to interpret. Therefore, its contract becomes

```scheme
;; interp : F1WAE listoff(fundef) SubCache → number
```

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

```scheme
;; lookup : symbol SubCache → FWAE

(define (lookup name sc)
  (type-case SubCache sc
    [mtSub () (error 'lookup "no binding for identifier")]
    [aSub (bound-name bound-value rest-sc)
      (if (symbol=? bound-name name)
          bound-value
          (lookup name rest-sc))]))
```

With that introduction, we can now present the interpreter:

```scheme
(define (interp expr fun-defs sc)
  (type-case F1WAE expr
    [num (n) n]
    [add (l r) (+ (interp l fun-defs sc) (interp r fun-defs sc))]
    [sub (l r) (− (interp l fun-defs sc) (interp r fun-defs sc))]
    [with (bound-id named-expr bound-body)
      (interp bound-body
        fun-defs
        (aSub bound-id
          (interp named-expr
            fun-defs
            sc)
          sc))]
    [id (v) (lookup v sc)]
    [app (fun-name arg-expr)
      ...])
```

1“Empty sub”—get it?
Three clauses have changed: those for with, identifiers and applications. Applications must look up the value of an identifier in the substitution cache. The rule for with evaluates the body in a substitution cache that extends the current one (sc) with a binding for the with-bound identifier to its interpreted value. The rule for an application similarly evaluates the body of the function with the substitution cache extended with the formal argument bound to the value of the actual argument.

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.

Consider the evaluation of the expression

\{with \{n 5\} \{f 10\}\}

in the following list of function definitions:

\(\text{(list (fundef 'f 'p (id 'n)))}\)

That is, \(f\) consumes an argument \(p\) and returns the value bound to \(n\). This corresponds to the Scheme definition

\(\text{(define (f p) n)}\)

followed by the application

\(\text{(local ((define n 5)) (f 10))}\)

What result does Scheme produce?

Our interpreter produces the value 5. 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 program has an unbound identifier (specifically, \(n\)). 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 the value 5 because the identifier \(n\) gets the value it was bound to by the with expression, that
is, from the scope in which the function \( f \) is used. Is this a reasonable way for the language to behave? A priori, is one interpretation better than the other? Before we tackle that, let’s introduce some terminology to make it easier to refer to these two behaviors.

**Definition 10 (Static Scope)** In a language with static scope, each identifier gets its value from the scope of its definition, not its use.

**Definition 11 (Dynamic Scope)** In a language with dynamic scope, each identifier gets its value from the scope of its use, not its definition.

Armed with this terminology, we claim that dynamic scope is entirely unreasonable. The problem is that we simply cannot determine what the value of a program will be without knowing everything about its execution history. If the function \( f \) were invoked by some other sequence of functions that did not have a value for \( n \), then the application of \( f \) would result in an error! In other words, simply by looking at the source text of \( f \), it would be impossible to determine one of the most rudimentary properties of a program: whether or not a given identifier was bound. You can only imagine the mayhem this would cause in a large modern software system. Furthermore, simply looking at the definition of \( f \), it is clear that there is no good reason a programmer might want to write such a program. We will therefore regard dynamic scope as an implementation error and reject its use in the remainder of this text.

### 5.2 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 reimplementation of it. We only need to focus our attention on one rule, that for function application. This currently reads:

\[
\text{app (fun-name arg-expr)}
\begin{align*}
\text{(local \[\text{define the-fun-def (lookup-fundef fun-name fun-defs)]})} \\
\text{(interp (fundef-body the-fun-def) fun-defs aSub (fundef-arg-name the-fun-def) interp arg-expr fun-defs sc sc))}
\end{align*}
\]

When the interpreter begins to evaluate the body of the function definition, how many substitutions does it apply? It applies as many as there already are in \( sc \), with one more for the function’s formal parameter to be replaced with the value of the actual parameter. But how many substitutions should be in effect in the function’s body? In our substitution-based interpreter, we implemented application as follows:

\[
\text{app (fun-name arg-expr)}
\begin{align*}
\text{(local \[\text{define the-fun-def (lookup-fundef fun-name fun-defs)]})} \\
\text{(interp (subst (fundef-body the-fun-def) (fundef-arg-name the-fun-def))}
\end{align*}
\]
Here, there is only one substitution performed on the function’s body, namely the formal parameter for its value. In particular, none of the substitutions applied to the calling function are in effect in the body of the called function (read the code carefully to convince yourself of this). This indicates that, at the point of invoking a function, the interpreter must “forget” about the current substitutions. Put differently, at the beginning of every function’s body, there is only one bound identifier—the function’s formal parameter— independent of the context from which it was called.

Given this, how do we fix our implementation? We clearly need to create a substitution for the formal parameter \((\text{fundef-arg-name the-fun-def})\). But the remaining substitutions must be empty, so as to not pick up the bindings of the calling context. Thus,

\[
\begin{align*}
\text{app} & \ (\text{fun-name arg-expr}) \\
& \ \text{local} \ ((\text{define the-fun-def (lookup-fundef fun-name fun-defs)})) \\
& \ \ (\text{interp (fundef-body the-fun-def)} \\
& \ \ \text{fun-defs} \\
& \ \ (\text{aSub (fundef-arg-name the-fun-def)}} \\
& \ \ \ (\text{interp arg-expr fun-defs sc}) \\
& \ \ (m\text{tSub})))
\end{align*}
\]

That is, we use the empty substitution cache to initiate evaluation of a function’s body. The difference between using \(sc\) and \((m\text{tSub})\) in the position of the box succinctly captures the implementation distinction between dynamic and static scope, respectively—though the consequences of that distinction are far more profound than this small code change might suggest.

Problem 5.2.1  How come we never seem to “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?

Problem 5.2.2  Why is the last sc in the interpretation of with also not replaced with \((m\text{tSub})\)? What would happen if we were to effect this replacement? Write a program that illustrates the difference, and argue whether the replacement is sound or not.

Problem 5.2.3  Our implementation of 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 lookup: a hash table, say. What changes do we need to make if we use a hash table?  
Hint: This is tied closely to Problem 5.2.1!
(define-type F1WAE
  [num (n number?)]
  [add (lhs F1WAE?) (rhs F1WAE?)]
  [sub (lhs F1WAE?) (rhs F1WAE?)]
  [with (name symbol?) (named-expr F1WAE?) (body F1WAE?)]
  [id (name symbol?)]
  [app (fun-name symbol?) (arg F1WAE?)])

;; interp : F1WAE listof(fundef) SubCache → number

(define (interp expr fun-defs sc)
  (type-case F1WAE expr
    [num (n) n]
    [add (l r) (+ (interp l fun-defs sc) (interp r fun-defs sc))]
    [sub (l r) (− (interp l fun-defs sc) (interp r fun-defs sc))]
    [with (bound-id named-expr bound-body)
      (interp bound-body
        fun-defs
        (aSub bound-id
          (interp named-expr
            fun-defs
            sc)
          sc))]
    [id (v) (lookup v sc)]
    [app (fun-name arg-expr)
      (local ((define the-fun-def (lookup-fundef fun-name fun-defs)))
        (interp (fundef-body the-fun-def)
          fun-defs
          (aSub (fundef-arg-name the-fun-def)
            (interp arg-expr fun-defs sc)
            (mtSub))))])

Figure 5.1: First-Order Functions with Cached Substitutions