Chapter 5

Deferring Substitution

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

```
  {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 this computational redundancy? We should create and use a *repository of deferred substitutions*. Concretely, here's the idea. Initially, we have no substitutions to perform, so the repository is empty. Every time we encounter a substitution (in the form of a `with` or application), we augment the repository 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which is that any identifier the interpreter encounters is of necessity free, for had it been bound, it would have been replaced by substitution.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 substitute them with by consulting the repository.

Exercise 5.0.2 *Can the complexity of substitution be worse than $O(n^2)$?*

5.1 The Substitution Repository

Let's provide a data definition for the repository:

```
(define-type DefrdSub
  [mtSub]
  [aSub (name symbol?) (value number?) (ds DefrdSub?)])
```

where DefrdSub stands for "deferred substitutions". A DefrdSub has two forms: it's either empty (mtSub[1]) or non-empty (represented by an aSub structure). The latter contains a reference to the rest of the repository in its third field.

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

```
;; interp : F1WAE listof(fundef) DefrdSub → number
```

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

```
;; lookup : symbol DefrdSub → F1WAE

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

With that introduction, we can now present the interpreter:

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

[1] "Empty sub"—get it?

```
(interp (fundef-body the-fun-def)
        fun-defs
        (aSub (fundef-arg-name the-fun-def)
              (interp arg-expr fun-defs ds)
              ds)))]))
```

Three clauses have changed: those for `with`, identifiers and applications. Applications must look up the value of an identifier in the repository. The rule for `with` evaluates the body in a repository that extends the current one (*ds*) 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 repository 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.

5.2 Deferring Substitution Correctly

Consider the evaluation of the expression

```
{with {n 5}
  {f 10}}
```

in the following list of function definitions:

```
(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

```
(define (f p) n)
```

followed by the application

```
(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 repository to conduct substitution more efficiently. We hope that that's all it does—that is, it must not also change the meaning of a program! 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 interpreter with deferred substitutions as buggy.

While this new 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, the scope of an identifier's binding is a syntactically delimited region.*

A typical region would be the body of a function or other binding construct. In contrast:

Definition 11 (Dynamic Scope) *In a language with* dynamic *scope, the scope of an identifier's binding is the entire remainder of the execution during which that binding is in effect.*

That is, in a language with dynamic scope, if a function `g` binds identifier `n` and then invokes `f`, then `f` can refer to `n`—and so can every other function invoked by `g` until it completes its execution—even though `f` has no locally visible binding for `n`.

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 bind a value for `n`, then that particular application of `f` would result in an error, even though a previous application of `f` in the very same program's execution completed successfully! 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 software system, especially with multiple developers and complex flows of control. We will therefore *regard dynamic scope as an error and reject its use* in the remainder of this text.

5.3 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 new interpeter be a correct reimplementation of it. We only need to focus our attention on one rule, that for function application. This currently reads:

```
[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 ds)
                     ds)))]
```

When the interpreter evaluates the body of the function definition, what deferred substitutions does it recognize? It recognizes all those already are in *ds*, 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:

```
[app (fun-name arg-expr)
     (local ([define the-fun-def (lookup-fundef fun-name fun-defs)])
       (interp (subst (fundef-body the-fun-def)
                      (fundef-arg-name the-fun-def)
                      (num (interp arg-expr fun-defs)))
               fun-defs))]
```

This performs only one substitution on the function's body: the formal parameter for its value. The code demonstrates that none of the substitutions applied to the *calling* function are in effect in the body of the *called* function. Therefore, at the point of invoking a function, our new interpeter 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 invoking context.

How do we fix our implementation? We clearly need to create a substitution for the formal parameter (which we obtain using the expression (*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,

```
[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 ds)
                     (mtSub))))]
```

That is, we use the empty repository to initiate evaluation of a function's body, extending it immediately with the formal parameter but no more. The difference between using *ds* and (mtSub) in the position of the box succintly captures the implementation distinction between dynamic and static scope, respectively—though the *consequences* of that distinction are far more profound that this small code change might suggest.

Exercise 5.3.1 *How come we never seem to "undo" additions to the repository? Doesn't this run the risk that one substitution might override another in a way that destroys static scoping?*

Exercise 5.3.2 *Why is the last sc in the interpretation of* with *also not replaced with (*mtSub*)? 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.*

Exercise 5.3.3 *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 substitutionrepository. 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 Exercise 5.3.1!*

```
(define-type F1WAE
  [num (n number?)]
  [add (lhs F1WAE?) (rhs F1WAE?)]
  [with (name symbol?) (named-expr F1WAE?) (body F1WAE?)]
  [id (name symbol?)]
  [app (fun-name symbol?) (arg F1WAE?)])

(define-type FunDef
  [fundef (fun-name symbol?)
          (arg-name symbol?)
          (body F1WAE?)])

(define-type DefrdSub
  [mtSub]
  [aSub (name symbol?) (value number?) (ds DefrdSub?)])

;; lookup-fundef : symbol listof(fundef) ⟶ fundef
(define (lookup-fundef fun-name fundefs)
  (cond
    [(empty? fundefs) (error fun-name "function not found")]
    [else (if (symbol=? fun-name (fundef-fun-name (first fundefs)))
              (first fundefs)
              (lookup-fundef fun-name (rest fundefs)))]))

;; lookup : symbol DefrdSub → F1WAE
(define (lookup name ds)
  (type-case DefrdSub ds
    [mtSub () (error 'lookup "no binding for identifier")]
    [aSub (bound-name bound-value rest-ds)
          (if (symbol=? bound-name name)
              bound-value
              (lookup name rest-ds))]))
```

Figure 5.1: Functions with Deferred Substitutions: Support Code

```
;; interp : F1WAE listof(fundef) DefrdSub → number
(define (interp expr fun-defs ds)
  (type-case F1WAE expr
    [num (n) n]
    [add (l r) (+ (interp l fun-defs ds) (interp r fun-defs ds))]
    [with (bound-id named-expr bound-body)
          (interp bound-body
                  fun-defs
                  (aSub bound-id
                        (interp named-expr
                                fun-defs
                                ds)
                        ds))]
    [id (v) (lookup v ds)]
    [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 ds)
                         (mtSub))))]))
```

Figure 5.2: Functions with Deferred Substitutions: Interpreter