In programming language lingo, the substitution cache is usually called the environment. Henceforth, we will use that terminology instead.

1 Implementing Environments

In the previous lectures, we saw one way of implementing environments, which is a datatype representing a stack:

```
(define-datatype SubCache SubCache?
  [mtSub]
  [aSub (name symbol?) (value FWA-value?) (sc SubCache?)])
```

Let’s take a slightly different view of environments. Environments are really just a mapping from identifiers to values. But they are a particular kind of mapping: at every instant, when we look up the value of an identifier in an environment, we want to get at most one value for it. That is, environments are just functions! In a language like Scheme, we can implement those directly. First, let’s define a predicate for our new representation of environments:

```
(define (env? x)
  (procedure? x))
```

This predicate is not exact, but it’ll suffice for our purposes.

Using this representation, we have a different way of implementing aSub:

```
;; aSub: symbol FWA-value env → env
(define (aSub bound-name bound-value env)
  (lambda (want-name)
    (cond
     [(symbol=? want-name bound-name) bound-value]
     [else (lookup want-name env)])))
```

The function aSub must return an environment, and since we’ve chosen to represent environments by Scheme functions (lambda), the function aSub must return a function. This explains the lambda.

An environment is a function that consumes one argument, a want-name, the name we’re trying to look up. It checks whether the name that name is the one bound by the current procedure. If it is, it returns the bound value, otherwise it continues the lookup process. How does that work?

```
;; lookup : symbol env → FWA-value
(define (lookup name env)
  (env name))
```

An environment is just a procedure expecting an identifier’s name, so to look up a name, we simply apply it to the name.

The implementation of the initial value, mtSub, is simply a function that consumes an identifier name and halts in error:

```
;; mtSub : () → env
(define (mtSub)
  (lambda (name)
    (error 'lookup "no binding for identifier")))
```
These changes do affect the definition of FWA-value:

```
(define-datatype FWA-value FWA-value?
  [numV (n number?)])
[closureV (param symbol?)
  (body FWAE?)
  (cache env?)])
```

However, the core interpreter itself remains unchanged:

```
;; interp : FWAE env → FWA-value
;; evaluates FWAE expressions by reducing them to their corresponding values
(define (interp expr env)
  (cases FWAE expr
    [num (n) (numV n)]
    [add (l r) (numV+ (interp l env) (interp r env))]
    [sub (l r) (numV- (interp l env) (interp r env))]
    [with (bound-id named-expr bound-body)
      (interp bound-body
        (aSub bound-id
          (interp named-expr env)
          env))]
    [id (v) (lookup v env)]
    [fun (param body)
      (closureV param body env)]
    [app (fun-expr arg-expr)
      (local ([define fun-val (interp fun-expr env)]
        [define arg-val (interp arg-expr env)])
        (cases FWA-value fun-val
          [closureV (cl-param cl-body cl-env)
            (interp cl-body
              (aSub cl-param arg-val cl-env))]
          [else (error 'interp "can only apply functions")]))])
```

2 A New Representation for FWAE Procedures

Let’s consider our representation of numbers. We made the decision that FWAE numbers be represented as Scheme numbers. Scheme numbers handle overflow automatically by growing as large as necessary. If we want to have FWAE numbers overflow in a different way (by performing modular arithmetic, say, as Java’s numbers behave), we might need to provide our own implementation of arithmetic that captures our desired overflow modes, and use this to implement FWAE arithmetic.

This class, however, is not about representing numbers in a computer, so we won’t be conducting such an exercise. The important point relevant this course is that by writing an interpreter ourselves, we get the power to make these kinds of choices. A related choice, which is relevant to this course, is the representation of functions.

What other representations are available for functions ($\text{fun}$) in FWAE? Currently, our interpreter uses a datatype. We might also use strings or vectors; vectors would gain little over a datatype, and it’s not quite clear how to use a string. One Scheme type that ought to be useful, however, is Scheme’s own procedure mechanism, $\text{lambda}$. Let’s consider how that might work.

First, we need to change our representation of function values. We will continue to use a datatype, but only to serve as a wrapper of the actual function representation (just like the $\text{numV}$ clause only wraps the actual number). That is,

```
(define-datatype FWA-value FWA-value?
  [numV (n number?)])
[closureV (p procedure?)])
```
We will need to modify the *fun* clause of the interpreter. When we implemented environments with procedures, we embedded a variant of the original lookup code in the redefined *aSub*. Here we do a similar thing: We want FWAE function application to be implemented with Scheme procedure application, so we embed the original *app* code inside the Scheme procedure representing a FWAE function.

\[
\text{fun (param body)}
\]
\[
(\text{closureV (lambda (arg-val)}
\]
\[
(\text{interp body}
\]
\[
(aSub param arg-val env)))\)]

That is, we construct a *closureV* that wraps a real Scheme closure. That closure takes a single value, which is the value of the argument expression (what we defined to be the “actual” parameter some lectures ago). It then interprets the body in an extended environment that binds the parameter to the argument’s value.

These changes give rise to two important questions:

1. Which environment do we extend when evaluating the body? The one that was active at the time of procedure definition *env*, thereby preserving static scope. How do we know that’s the one we’ll get? *Because Scheme is statically scoped*. That is, Scheme’s *lambda* does the hard work of making sure we get the “right” environment.

2. Doesn’t the body get interpreted when we define the function? No, it doesn’t. It only gets evaluated when something—hopefully the application clause of the interpreter—extracts the Scheme procedure from the *closureV* value and applies it to a value.

In turn, the application case becomes

\[
\text{app (fun-expr arg-expr)}
\]
\[
(\text{local ([define fun-val (interp fun-expr env)])}
\]
\[
([\text{define arg-val (interp arg-expr env)})
\]
\[
(\cases \text{FWA-value fun-val}
\]
\[
(closureV (p)
\]
\[
(p arg-val))
\]
\[
[\text{else (error \textquote{interp *can only apply functions*})})])\)]

That is, having reduced the function and argument positions to values, the interpreter extracts the Scheme procedure that represents the function, and applies it to the argument value.

Therefore, the entire interpreter becomes:

\[
\text{;; interp : FWAE env \rightarrow FWA-value}
\]
\[
\text{;; evaluates FWAE expressions by reducing them to their corresponding values}
\]
\[
(\text{define (interp expr env)}
\]
\[
(\cases \text{FWAE expr}
\]
\[
[num (n) (numV n)]
\]
\[
[add (l r) (numV+ (interp l env) (interp r env))]
\]
\[
[sub (l r) (numV- (interp l env) (interp r env))]
\]
\[
[\text{with (bound-id named-expr bound-body)}
\]
\[
(\text{interp bound-body}
\]
\[
(aSub bound-id
\]
\[
(\text{interp named-expr env)}
\]
\[
(env))\)]
\]
\[
[id (v) (lookup v env)]
\]
\[
[fun (param body)
\]
\[
(\text{closureV (lambda (arg-val)}
\]
\[
(\text{interp body}
\]
\[
(aSub param arg-val env)))\)]
\]
\[
[app (fun-expr arg-expr)
\]
\[
(\text{local ([define fun-val (interp fun-expr env)])}
\]
\[
([\text{define arg-val (interp arg-expr env)})
\]
\[
(\cases \text{FWA-value fun-val}
\]

3
In short, a `fun` expression now evaluates to a Scheme procedure that takes a FWAE value as its argument. Function application in FWAE is now just procedure application in Scheme.

3 Meta Interpreters

It wasn’t that difficult to represent FWAE functions with Scheme procedures. There’s a reason for this: FWAE and Scheme have the same semantics.

Definition 1 (meta interpreter) A meta interpreter is an interpreter that uses the language features beyond rudimentary data structures of the interpreting language to represent features of the interpreted language.

Definition 2 (syntactic interpreter) A syntactic interpreter is an interpreter that uses only rudimentary data structures of the interpreting language to represent features of the interpreted language.

Our first FWAE substitution-based interpreter was very nearly a pure syntactic interpreter: the only language feature it borrowed from Scheme was numbers. Our current interpreter is a meta interpreter—we use Scheme closures to implement FWAE closures, Scheme procedure application for FWAE function application, Scheme numbers for FWAE numbers, and Scheme arithmetic for FWAE arithmetic.

With a good match between the interpreted language and the interpreting language, writing a meta interpreter can be very easy. With a bad match, though, it can be very hard. With a syntactic interpreter, implementing each semantic feature will be somewhat hard, but in return you don’t necessarily have to worry about getting a good language match. In particular, if there is a particularly strong mismatch between interpreting and interpreted language, it may take less effort to write a syntactic interpreter than a meta interpreter. (Consider, for instance, if we had wanted FWAE to employ dynamic rather than static scope! How much work would we have to do to implement this using the Scheme procedure representation of FWAE functions?)

In fact, ignoring the switch from parens to curls, our current interpreter can be classified as something more specific than a meta interpreter:

Definition 3 (meta-circular interpreter) A meta-circular interpreter is a meta interpreter in which the interpreting and interpreted language are the same.

Besides educational purposes (and generating head-splitting programming tasks), are there any uses for meta-circular interpreters? Yes. We saw that a meta interpreter is good for testing your understanding of a language. A meta-circular interpreter is good for finding bugs in a language that you write. By writing a meta-circular interpreter in a language that you designed, you are likely to find problems or inconsistencies that you hadn’t considered before. In fact, one definition of a truly powerful language is that makes it easy to write a meta-circular interpreter in it.

For instance, if you define a new scripting language, no doubt you will get some of its features fairly right, such as those to parse data files or communicate over a network. But will you get the domain-independent parts—procedures, scope, etc.—right also? And how can you be sure? One good way is to try and write a meta, then meta-circular interpreter using the language. You will probably soon discover all sorts of deficiencies in the core language. The failure to apply this simple but effective experiment is probably responsible for the amazingly messy state of many scripting languages (Tcl, Perl, JavaScript, Python, etc.) for so long; only now are they getting powerful enough to actually support effective meta-circular interpreters.