Today's Lecture Notes for cs173

sk and dbtucker

September 15, 2000

Topic: Lambda.

Adding functions to our language

We want to have functions in our language, so we extend the BNF:

$$
\begin{array}{lcl}
L & ::= & \langle \text{number} \rangle \\
 & | & \langle \text{id} \rangle \\
 & | & \texttt{(}\langle L \rangle\ \texttt{+}\ \langle L \rangle\texttt{)} \\
 & | & \texttt{(}\langle L \rangle\ \texttt{*}\ \langle L \rangle\texttt{)} \\
 & | & \texttt{(fun (}\langle \text{id} \rangle\texttt{)}\ \langle L \rangle\texttt{)} \\
 & | & \texttt{(}\langle L \rangle\ \texttt{(}\langle L \rangle\texttt{))}
\end{array}
$$

The second-to-last line represents a function of one parameter and a body, and the last line represents the application of a function to an argument. (Note that all our functions take a single argument.)

Now that we have functions, we don't need the let form anymore. To see this, consider the expression:

```
(let (x 3) (x + 4))
```

is equivalent to:

```
((fun (x) (x + 4)) (3))
```

In both cases, the variable `x` is bound to `3` in the body `(x + 4)`. (In fact, Scheme does exactly this conversion—when it sees `(let ((x v)) body)`, it changes the form to `((lambda (x) body) v)`.)

We need to extend the abstract datatype as well; we call it `AFunExp`:

```
(define-datatype AFunExp AFunExp?
  [numE (n number?)]
  [varE (v symbol?)]
  [addE (lhs AFunExp?)
        (rhs AFunExp?)]
  [multE (lhs AFunExp?)
         (rhs AFunExp?)]
```

```
  [funE (param symbol?)
        (body AFunExp?)]
  [appE (fun AFunExp?)
        (arg AFunExp?)])
```

Note that expressions like: `(3 (4))` and `(5 + (fun (x) x))` are legal syntactically—they can be parsed into our datatype representation. However, these expressions are clearly nonsense, so we will assume that our input does not contain such forms.

Where do the expressions `varE`, `funE`, and `appE` fall in terms of values and computations? `appE` is a computation, since it has to evaluate the function applied to its argument. `funE` is a value, since we don't do anything with it when we evaluate it. Here's what we have:

- values: `numE`, `funE`
- computations: `addE`, `multE`, `appE`

`varE` doesn't fall nicely into either category since it's just a placeholder for values. On one hand, we not computing anything for `varE`, just substituting into it. On the other hand, it's certainly not a value that a program can return.

A substitution interpreter

Now we can write an interpreter which uses substitution. We use the `subst` function written previously, and assume the functions `numE+` and `numE*`.

```
;; interp : AFunExp -> AFunExp
(define (interp a)
  (cases AFunExp a
    [numE (n) a]
    [varE (v) (error "not closed")]
    [addE (le re)
      (numE+ (interp le) (interp re))]
    [multE (le re)
      (numE* (interp le) (interp re))]
    [funE (param body) a]
    [appE (fe ae)
      (apply-fun (interp fe) (interp ae))]))

;; apply-fun : AFunExp * AFunExp -> AFunExp
(define (apply-fun fv av)
  (cases AFunExp fv
    [funE (param body)
      (interp (subst body
                     param
                     av))]
    [else (error "can only apply functions")]))
```

Three notes on this interpreter:

1. Note that when `interp` comes across a `funE`, it just returns the function as is.

2. The interesting case is `appE`. The function `apply-fun` takes a function value and an argument value, substitutes the argument for the parameter in the body of the function, and then calls `interp` on the new expression. Since the argument is evaluated before it is substituted, the evaluation is eager.

3. It only makes sense to apply a function value to an argument, so trying to apply anything else is an error (the final line of `apply-fun`).

An environment-passing interpreter

For the environment-passing interpreter, it will be useful to have a separate datatype for values:

```
(define-datatype Value Value?
  [numV (n number?)]
  [funV (param symbol?)
        (body AFunExp?)])
```

We assume we have functions `numV+` and `numV*` written over this datatype.

Now let's take a crack at the interpreter, using the functions for delayed substitution we developed before:

```
;; interp : AFunExp * DSub -> Value
(define (interp a d)
  (cases AFunExp a
    [numE (n) (numV n)]
    [varE (v)
      (get-sub v d)]
    [addE (le re)
      (numV+ (interp le d) (interp re d))]
    [multE (le re)
      (numV* (interp le d) (interp re d))]
    [funE (param body)
      (funV param body)]
    [appE (fe ae)
      (apply-fun (interp fe d) (interp ae d) d)]))

;; apply-fun : Value * Value * DSub -> Value
(define (apply-fun fv av d)
  (cases Value fv
    [funV (param body)
```

```
      (interp body
              (new-sub param
                       av
                       d))]
    [else (error "can only apply functions")]))
```

This appears to be right, but we haven't taken into account how the delayed substitution works. For example, consider this expression (we use `let`s just for clarity):

```
(let (x 3)
  (let (f (fun (y) (x + y)))
    (let (x 4)
      (f (5)))))
```

What are the delayed substitutions at each step?

0. Before evaluation, $\texttt{d} = \{\}$.
1. After the first let, $\texttt{d} = \{\texttt{x} \mapsto \texttt{3}\}$.
2. After the second, $\texttt{d} = \{\texttt{x} \mapsto \texttt{3},\ \texttt{f} \mapsto \texttt{(fun (y) (x + y))}\}$.
3. After the third, $\texttt{d} = \{\texttt{x} \mapsto \texttt{4},\ \texttt{f} \mapsto \texttt{(fun (y) (x + y))}\}$.
4. The body of `f`—that is, `(x + y)`—is then evaluated under $\texttt{d} = \{\texttt{x} \mapsto \texttt{4},\ \texttt{y} \mapsto \texttt{5},\ \texttt{f} \mapsto \texttt{(fun (y) (x + y))}\}$.

Thus, the expression evaluates to 9. The reason is that the variable `x` in the function was bound to the second `x` in the program (4) instead of the first `x` (3).

Is 9 the correct answer? Well, it depends on what we mean by correct. In most languages, we would expect the `x` in `f` to be bound by the nearest enclosing definition of `x`—the first one—and 8 would be the answer. This is called *static binding*, because the variable is bound in the environment where the function is defined. The substitution intepreter would also compute 8, so in some sense this is correct.

On the other hand, older versions of Lisp used *dynamic binding*, as the above interpreter does. Under these semantics, a variable is bound in the environment where the function is applied.

We would like our environment-passing interpreter to use static binding. To do this, we need to "remember" the current delayed substitutions (i.e. enviroment) whenever we evaluate a function. Thus, we modify the `Value` datatype to carry around the environment:

```
(define-datatype Value Value?
  [numV (n number?)]
  [funV (param symbol?)
        (body AFunExp?)
        (env DSub?)])
```

Now we rewrite `interp` and `apply-fun` to store and recall these saved substitutions:

```
;; interp : AFunExp * DSub -> Value
(define (interp a d)
  (cases AFunExp a
    [numE (n) (numV n)]
    [varE (v)
      (get-sub v d)]
    [addE (le re)
      (numV+ (interp le d) (interp re d))]
    [multE (le re)
      (numV* (interp le d) (interp re d))]
    [funE (param body)
      (funV param body d)]
    [appE (fe ae)
      (apply-fun (interp fe d) (interp ae d))]))

;; apply-fun : Value * Value -> Value
(define (apply-fun fv av)
  (cases Value fv
    [funV (param body env)
      (interp body
              (new-sub param
                       av
                       env))]
    [else (error "can only apply functions")]))
```

We made three modifications:

1. For the `funE` case in `interp`, we now create a `funV` which stores the current environment.

2. In `apply-fun`, the body is interpreted in the environment stored in `funV` plus the binding of the function parameter.

3. Note that `apply-fun` no longer needs a environment parameter.

When evaluating functions, we create a package consisting of the function definition and the current environment. This package is called a *closure*.