CS98-52 Lecture 8

Wednesday, November 13, 2019

Example

Quine (CS61A HW8, Q2)

> In most programming languages, there's some way to write a program that prints itself. We haven't really had a way to do this:
>
> ```
> >>> print(2)
> 2
> >>> print('print(2)')
> print(2)
> ```
>
> There's always one more `print` in our program than is printed out. We can fix this with semicolon:
>
> ```
> >>> s = 'print(s)'; eval(s)
> print(s)
> ```
>
> But we haven't quite fixed everything! There's an extra eval, and a semicolon, and the assignment statements, so your challenge will be to fix all those things in HW8.

Promises

Today in 61A Lecture, we discussed streams, which are lists that are lazily evaluated.

```
> (cons 1 (cons 2 nil))
(1 2)
> (cons-stream 1 (cons-stream 2 nil))
(1 . #[promise (not forced)])
```

This has multiple benefits. We could generate an infinite stream of 1s. But what is this promise thing? It is an expression that hasn't been evaluated.

You can create promises using either `cons-stream`, or you can create it directly. There is a special form called `delay`:

```
> (define t (delay (list 2 3 4)))
t
> (force t)
(2 3 4)
```

The `force` special form can also be used on streams:

```
> (define s (cons-stream 1 (cons-stream 2 nil)))
s
```

```
> (force (cdr s))
(2)
```

A promise is an expression, along with an environment in which to evaluate it.

Delaying an expression creates a promise to evaluate it later in the current environment. Forcing a promise returns its value in the environment in which it was defined.

```
> (define promise (let ((x 2)) (delay (list (+ x 1)) ) ))
promise
> (define x 5)
x
> (force promise)
(3)
```

What else works like this? That's right: a function.

```
> (define promise (let ((x 2)) (lambda () (list + x 1)) ) ))
promise
```

This means we can define `delay` and `expr` in terms of a function:

```
(define-macro (delay expr) `(lambda () ,expr))
(define (force promise) (promise))
```

A stream is a list, but the rest of the list is computed only when forced:

```
(define-macro (cons-stream a b) `(cons ,a (delay ,b)))
(define (cdr-stream s) (force (cdr s)))
```

Lazy evaluation is good because it means only parts of the code that matter will error. We saw this in `if`. What happens if we want to switch the `if` special form with an `unless` special form?

```
(define (unless p a c) (if p c a))
```

The problem is, this doesn't quite work the same.

```
> (unless false 2 1)
2
> (unless false 2 (/ 1 0))
Error
```

We could fix this with a macro, but a more extreme way to fix this issue is to change our programming language to interpret lazily, so that only the parts of the code we need are evaluated.

If the whole interpreter were lazy in this way, not only would we be able to define `unless` this way, but all other kinds of things. Most programming languages don't do this, and only one language in regular use uses lazy evaluation by default: Haskell, which is mostly used in the academic community.

But we could make Scheme work this way too!

Lazy Evaluation

Let's figure out what happens when a procedure is applied. If it's a built-in procedure like `+`:

```
(+ (* 2 3) 4)
```

Well, in this case, we do need to evaluate the arguments, and the primitive procedure is applied to them, otherwise nothing happens. However, if it's a user-defined procedure, then we maybe don't yet need to know. Thus, we delay the arguments.

Other parts of the language must change too. When an `if` expression is evaluated, then we need to evaluate the predicate, but we can delay any call expression evaluation in the consequent and alternative until we need it.

```
class Thunk:
    """An expression EXPR to be evaluated in the environment ENV"""
    def __init__(self,expr,env):
        self.expr = expr
        self.env = env
```

And now let's rewrite `scheme_eval`:

```
def lazy_eval(expr,env):
    if scheme_symbolp(expr):
        return env.lookup(expr)
    while self_evaluating(expr):
        return expr

    if not scheme_listp(expr):
        raise SchemeError('malformed list')

    first, rest = expr.first, expr.second

    if scheme_symbolp(first) and first in SPECIAL_FORMS:
        return SPECIAL_FORMS(first)(rest,env)
    else:
        procedure = scheme_eval(first,env)
        if isinstance(procedure,BuiltinProcedure):
            args = rest.map(lambda o: scheme_eval(o,env))
        else:
            args = rest.amp(lambda o: Thunk(o,env))
        return scheme_apply(procedure,args,env)
```

As it is, our evaluation procedure is so lazy it never actually does any work. It returns a `Thunk` object:

```
def scheme_eval(expr,env,can_delay=False):
    result = lazy_eval(expr,env)
    if not can_delay and isinstance(result,Thunk):
        return scheme_eval(result.expr,result.env)
```

```
    else:
        return result
```

Now we can do basic operations like before:

```
> (+ 1 2)
3
> (define (f x) (+ x 1))
f
> (f 3)
4
> (f (/ 1 0))
Error
> (if false (/ 1 0) 2)
2
> (define (unless p a c) (if p c a))
unless
> (unless false 2 (/ 1 0))
2
```

And if you were wondering when `can_delay` is implemented, then you need to rework your previous functions to figure out when `can_delay` is `True` and when it is `False`. For example, `do_if_form` will declare `can_delay` to be `False` for the predicate, and `True` for the consequent and the alternative.

And now we have one other thing to worry about:

```
> (cons 1 (cons (/ 1 0) nil))
Error
> (car (cons 1 (cons (/ 1 0) nil)))
Error
```

That second expression should return 1, because we don't care about the rest of the list! We can fix the definition of `cons`:

```
> (define (cons a d) (lambda (m) (if m a d)))
cons
> (define (car c) (c true))
car
> (define (cdr c) (c false))
cdr
```

The list no longer prints itself, but otherwise it works like the built-in implementation:

```
> (cons 1 (cons 2 nil))
(lambda (m) (if m a d))
> (car (cdr (cons 1 (cons 2 nil))))
2
> (car (cons 1 (cons (/ 1 0) nil)))
1
```