2013W1-lecture16

October 20, 2013

Contents

1 Question of the Day

What's the difference between a variable and a zero-argument function?

```
 SOLUTION
It depends on the language.  In Prolog and Haskell, for example, the
two are the same.  In Racket and C++, however, there are some
important differences that go beyond syntax.

For example, consider these two:
```

```
(define counter-expr
  (local [(define value 0)]
    (begin
      (set! value (add1 value))
      value)))

(define counter-fun
  (local [(define value 0)]
    (lambda ()
      (begin
        (set! value (add1 value))
        value))))
```

2 Logistics

2.1 Short (sort of) quiz!

As usual: submit corrections by Tue at 8PM (as `ca3` on handin).

Unlike usual, you get credit for 4 of the 6 questions as thanks for filling out the midterm course evaluation :)

2.2 Programming Assignment #4

We're releasing it soon! You'll be implementing the core of ParselTongue.

2.3 Neat PL research talk upcoming: 24 Oct, 3:30-4:30PM, X836

One bonus point for attending and posting a brief summary of the talk from the CPSC 311 perspective.

Two bonus points for attending and posting a thoughtful, thorough discussion of the talk.

Three bonus points for being the speaker. :)

TypeScript is a programming language whose goal is to support development of large JavaScript programs. TypeScript is a superset of the current JavaScript standard (ECMAScript 5) that adds an optional static type system to JavaScript. TypeScript exists only to support high-level thinking about JavaScript programs; it has no impact on runtime behavior. Because

of this, TypeScript is an example of "types for tooling" vs. the more traditional idea of "types for runtime safety." TypeScript has a novel design for type inference; the goal of the design is to provide maximum convenience (few annotations required) and transparency (chains of inference are clear and local). The TypeScript compiler, incremental static analysis tools, and specification are open source (see typescriptlang.org). Several million lines of TypeScript are part of shipping Microsoft products. Since the community preview release in October, 2012, several 100K+ line TypeScript projects have grown up outside of Microsoft and the TypeScript community has created a site, at github.com, that holds over 100 community-maintained TypeScript descriptions of popular JavaScript frameworks such as jQuery.

Steve Lucco is a Technical Fellow at Microsoft, where he is responsible for Microsoft's web development tools and runtimes. He led the development of Microsoft's Chakra JavaScript engine, which powers Internet Explorer. Currently, Chakra is 30% faster than Chrome V8 on SunSpider, the most widely cited JavaScript benchmark. He started the TypeScript team and contributes to the design and implementation of TypeScript.

3 Lazy Evaluation with Environments

We have been using eager evaluation semantics. We're switching to lazy evaluation. (Our secret goal, however, is to discover the concept of a "closure".)

3.1 Changing Environments to Support Lazy Evaluation

Challenge #1 is to make this test case pass:

```
(test (run '(useless x)
           (list (parse-fun-def '(define (useless ignore) 2))))
      2)
```

In lazy evaluation, we bind the formal parameter to the *unevaluated* actual parameter expression.

DISCUSSION: What will we need to change to put that expression in our environment?

```
  SOLUTION
There's a few approaches, but somehow we need to keep expressions
around in the environment.  We'll do that by altering our notion of
"value" from just =number= to a user-defined type that can be a number
```

```
or an expression.  That will cause a /lot/ of cascading changes, since
we produce values in our interpreter!
```

3.2 Not Being Lazy Forever

DISCUSSION: What do we get back from this program in our current interpreter?

```
(run '(identity (+ 1 2)) (list (parse-fun-def '(define (identity x) x))))
```

```
 SOLUTION
We get (exprV (plusC (numC 1) (numC 2)))

Is that what we want?  If not, how shall we fix it?
```

In a lazy language, a "strictness point" is a point in the program where we want so-far-deferred evaluation to actually occur so we get actual values out rather than deferred expressions. Let's define a helper function `strict` inside `interp`. It should take an expression and evaluate it (stubbornly, over and over if necessary!) until it gets back a "real" value and not another deferred expression.

Unless the people running our programs want to see as their result "a program that, if run, would give you the value you asked for", it seems like we should have a strictness point in `run`!

Is there anywhere else we need a strictness point? Let's step through the cases of `interp` and see if an `exprV` would ever cause trouble.

```
 SOLUTION
An exprV would cause trouble if we tried to use the primitive +, /,
or * operators on it or if (for the divC case) we tried to check
whether the exprV was equal to zero.

In all of these cases, we're going to need a strictness point so we
can get the actual VALUE we want to deal with.
```

3.3 Oops, Lost Static Scoping

DISCUSSION: What do we get back from this program in our current interpreter?

```
(run '(triple 5) (list (parse-fun-def '(define (triple y) (+ y (double y))))
                       (parse-fun-def '(define (double x) (+ x x)))))
```

```
 SOLUTION
An unbound identifier error on y.  Why?

When we call triple, its formal parameter y is bound to the
unevaluated expression (numc 5).  The difference between the
expression (numc 5) and the value (numv 5) seems almost academic.

Indeed, when we look-up y for the first addend, we construct a
perfectly sensible expression ready to add 5 to the second addend.

However, we then bind double's formal parameter x to the
unevaluated expression (idC 'y).  The difference between that and
(numV 5) no longer looks academic.  How do we know y's value is 5?
We don't unless we remember y's value.

Unfortunately, there's nothing in the expression (idC 'y) to represent
the environment in which that expression was created.
```

3.4 Grabbing ("Closing Over") an Expression's Environment

We need the environment in which the expression was "meant" to be evaluated. Can we get that?

Sure! Look back at our *eager* language. We actually *do* evaluate the expression right away in that language. All we need is whatever that language supplied to its call to `interp`:

```
(define arg-value (interp a env fds))]
```

We already have `a` (the argument expression itself). `env` is the environment in which the expression should be evaluated, when it comes time to evaluate it. (And, we actually don't need the list of function definitions `fds`, because that's the same throughout our program.)

A *closure* is a data structure that "closes over" its static context, the environment in which it was created. The reason to do that is so that we can "jump back" to that environment at some later point.

3.5 What More Do We Need for Functions?

With closures, we can "jump back to" a static point in the program and evaluate a deferred piece of code. That sounds a lot like evaluating a function body!

In fact, we're really only missing one piece in our "expression closures" to be able to implement functions. What is it?

```
 SOLUTION
We're just missing parameters.  In our case, for single-argument
functions, we just need the name of the formal parameter.

We also need a syntax to apply functions, but we already have that!
```

Next time, let's modify our interpreter to go back to eager semantics but allow anonymous functions!

4 What have we learned today?

- From QotD: Preparation for the note below about expression closures being much like function closures.
- Lazy Evaluation w/Environments
  - Justify changing the value type for our interpreter and environment from `number` to a richer type that supports expressions.
  - Identify the changes necessary to accomplish this.
  - Define a "strictness point": a point in the program where we want so-far-deferred evaluation to actually occur so we get actual values out rather than deferred expressions.
  - Justify the need for a strictness point at the top level of the interpreter.
  - Justify the need for a strictness point anywhere that we rely on specific properties of the value of an expression in order to continue interpretation, including addition and other numeric operations.
  - Justify the need for deferred expressions in the environment to "close over" (grab and carry along with them) the environment in which they originally appeared (by explanation and by giving an example that behaves incorrectly without doing this).
- Closures
  - Define a closure: a data structure that wraps up an expression (i.e., a piece of code) and its static context.

- Explain how closures enable us to preserve static scoping even when we defer evaluation of an expression until some (statically unknown) other point(s) in the program.
- Justify the contents of a closure by comparing it to the arguments to a call to `interp`. (This is in addition to the justification above for preserving static scoping!)
- Explain the difference between an expression closure and a function closure:
  * Expression closures are only used for lazy evaluation.
  * Function closures (typically) list at least one formal parameter. Expression closures are effectively functions of zero parameters.
  * Not a point that I'll test you on unless and until we return to lazy evaluation, but: Expression closures don't count as "real values" at strictness points, but function closures (even with zero arguments) *maybe* should.