2013W1-lecture18

October 22, 2013

Contents

1 Question of the Day

In classical comutation, a bit can be either 0 or 1.

In quantum computation, a qubit can be in a superposition of the states 0 and 1. (As with classical computation, these labels are arbitrary; unlike with classical computation, alternate labelings can be quite meaningfully different.)

Worse yet, a system of multiple qubits cannot necessarily be described by simply independently describing its individual qubits. (For example, two qubits can be (but don't have to be!) entangled in such a way that measuring one of the qubits will give the value 0 with probability one-half or the value 1 with probability one-half, but measuring the other one afterward will **always** give the same value you get from measuring the first.)

So, how do you program these things?

```
 SOLUTION
So far, we don't really know!  Work on programming them is going in
multiple directions simultaneously, however.  (As usual, Wikipedia is
a good place to start:
http://en.wikipedia.org/wiki/Quantum_programming.)

People have introduced basic machines to model quantum systems.  These
include three models closely corresponding to common classical models
of computation:
+ Quantum circuits (analogous to classical circuits, which you'd learn
  about in CPSC 121)
+ Quantam Turing Machines (analogous to classical Turing Machines, a
  super-simple abstract computer model, which you'd learn about in
  CPSC 421)
+ Quantum lambda-calculi (analagous to lambda calculus, a fundamental
  description of computation using nothing but function definition and
  application.. which we'll learn about today!).

These models are necessary to understand what we're describing when we
write high-level quantum programs and to prove properties of the
programs (e.g., that we won't accidentally perform operations that
cause "decoherence"---collapsing of quantum states).

There's also much work on high-level languages for programming.  I'll
pick on QCL, the language I know the most about (which isn't much!).

Here's some choice quotes from Bernhard Omer's QCL thesis
(http://tph.tuwien.ac.at/~oemer/doc/structquprog.pdf): "The purpose of
programming languages is therefore twofold, as they allow the
expression of a computation's semantics in an abstract manner, as well
as the automated generation of a sequence of elementary operations to
control the computing device."
```

In other words, a programming language provides an interface to the programmer that enables abstraction and an interface to the underlying system that enables execution. (We talk about a few other purposes, including analysis!, which can be pretty important in quantum computation as well!)

He then specifically highlights the importance of subroutines (functions) in providing:

+ Functional Abstraction (i.e., functions with parameters let us generalize frequently used code into a single reusable definition)
+ Hierarchical Program Structure (i.e., allowing functions to call other functions enables programmers to define abstractions atop their own abstractions)
+ Recursion (i.e., allowing functions to call themselves enables programmers to express powerful control structures and algorithms)
+ Private Scopes and Local Variables (i.e., static scoping allows us to hide and protect implementation details to make software clearer and easier to change)

So, let's get functional!

2 Logistics

2.1 Quiz on Wednesday!

2.2 Programming Assignment #4

It's out; the milestone is due Sunday.

Get started, but be sure to read the chapter on state (Chapter 8) first!

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

Finishing file:2013W1-lecture16.org!

4 First-Class Functions

4.1 Expression Closures: Almost Function Values!

We said our functions had four key purposes:

- delaying evaluation of the function body,
- allowing evaluation of the function body from (possibly distant) locations in the program,
- blocking out bindings from the function's dynamic context, and
- "allowing in" a dynamic value via the parameter(s).

We now have "expression closures", which are very much like values representing functions.

If we can get to a full value representation, we can define a function anywhere and evaluate it to one of these values. We could also then pass function values as parameters, return them as results from functions, and generally manipulate them in very powerful ways! For example, we'd be able to write and use the `map` function.

Which of our four purposes do our expression closure values support?

```
 SOLUTION
They definitely support delaying evaluation of the expression (which
can be the function body).  That's the whole reason we introduced
them: for lazy evaluation!

Similarly, they clearly allow evaluating the expression from possibly
distant locations in the program.

They also block out the dynamic scope where they're evaluated,
replacing it with the static scope where the expression initially
appeared in the program.

They do not, however, allow in dynamic values(s) via the parameter(s).
```

4.2 Getting to Closures: Function Values

We're only missing binding parameter names to values supplied by dynamic context. What do we need to add to closures to allow this?

Do they need to store the parameter names? Do they need to store the parameters' values?

```
 SOLUTION
We do need to store the parameter names.  Those are specified where
the function is defined, but we need to keep them around until the
function is applied.  Since closures are function values, they must
store everything about the function definition that we need to keep
around until the function is applied.

We do not need to store the parameters' values.  We don't know those
values until the moment the function is applied, and we know just what
to do with them when the function is applied: the same thing we've
```

```
already been doing!  Bind the parameter names to their values in the
environment (the static environment in which the function was defined)
and use that extended environment to evaluate the function body.
```

4.3 Closures!

So, closures are function values. They include three parts: the parameter name(s), the function body, and the environment in which the function was defined. They're called "closures" because of the way they "close over" the environment in which the function was defined and keep it around for when the function is later applied.

Here's a data type we could use:

```
[closureV (param : symbol) (body : ExprC) (env : Env)]
```

Now we can define functions using syntax like:

```
(define double (lambda (x) (+ x x)))
```

Or using syntactic sugar for this same style like:

```
(define (double x) (+ x x))
```

In our core syntax, that might look like:

```
(LetC 'double (FuncC 'x (PlusC (IdC 'x) (IdC 'x))) ...)
```

where the `...` gets replaced by the rest of the program.

Let's fill in the `FuncC` and `AppC` cases of the interpreter now that we have this. Note: we're going **back to eager evaluation**. We just exploited lazy evaluation as a way to motivate our desire to close over environments!

```
(define (interp [expr : ExprC] [env : Env]) : Value
  (type-case ExprC expr
    ...
    [FuncC (param body-expr)
       ...]

    [AppC (fun-expr arg-expr)
       ...])
```

```
 SOLUTION
(define (interp [expr : ExprC] [env : Env]) : Value
  (type-case ExprC expr
    ...
    [FuncC (param body-expr)
      ; A closure needs the parameter (given in the FuncC),
      ; the body (given in the FuncC), and the environment in
      ; which the function was defined.  That environment is
      ; the current environment right NOW: the one passed to
      ; interp.
      ;
      ; Shouldn't we be recursively evaluating body-expr?
      ; No!! One of the main points of function definitions is
      ; that they defer evaluation of their bodies!
      (closureV param body-expr env)]

    [AppC (fun-expr arg-expr)
      ; A function application gets a function value from its
      ; first expression and applies the function inside to the
      ; value of its argument.  Here's where we finally get to
      ; USE the env we tucked away wherever we defined the function.
      (local [(define fun (interp fun-expr env))
              (define arg (interp arg-expr env))]
        (if (closureV? fun)
          ; The environment is the one in which the closure was
          ; defined, but as always, we also need to bind the parameter.
          (interp (closureV-body fun)
                  (extend-env (bind (closureV-param fun) arg)
                              (closureV-env fun)))
          (error 'interp "Attempt to apply a non-function.")))]))
```

4.4 Done: Functions Anywhere

That's it! Closures (function values) are just a tiny step from lazy evaluation with environments.

It turns out that now that we have first-class functions, we can perform any computation that any "real" language can perform. As a couple of examples:

1. Here's a desugaring of `let` to function definition and application:

`(let (id bind-expr) body)` becomes `((lambda (id) body) bind-expr)`

2. Below is code to give us cons cells! I assume we have multi-argument function definition and application (but we can desugar those to single-argument function definition and application!).

```
; To implement a cons cell, we need to store the first and rest fields.
; If only we had some way to remember something until later!  Oh, wait..
; we do: closures!  A closure can remember values for us by toting its
; envorinment around.
;
; So, our strategy for cons is to get the first and rest fields and then
; produce a closure that will remember those.  We'll have the closure
; accept a "selector" function that is given the two fields and can
; select which one is wanted.
;
; I'm going to add a third field that represents whether this cons cell
; is really the empty list.  That's false for cons.
(let (cons (lambda (fst rst) (lambda (sel) (sel fst rst false))))
  ; Now, we need the first and rest functions.  They just need to take
  ; cons cell and then give it an appropriate selector.  first will
  ; give a selector that gives back the fst field and ignores rst.
  ; rest will do the opposite.
  (let (first (lambda (cons-cell) (cons-cell (lambda (fst rst emp) fst))))
    (let (rest (lambda (cons-cell) (cons-cell (lambda (fst rst emp) rst))))
      ; We still need an empty list and the empty? function.  Those are
      ; pretty easy; they just focus on that third field that tells us
      ; whether a cons cell is empty.
      ;
      ; (Although they beg the question: how do we define true, false,
      ; and if using just functions?  We can do it.. but we'll wait until
      ; later to discuss it.)
      (let (empty (lambda (sel) (sel false false true)))
        (let (empty? (lambda (cons-cell) (cons-cell (lambda (fst rst emp) emp))))
        ; Now, I'll make a list and get the second element back.
          (first (rest (cons 1 (cons 2 (cons 3 empty)))))))))
```

5 What have we learned today?

- From QotD: How programming languages are venturing into a whole new world of computation.. and how important what we've already learned, particularly functions, is in making that new world work.
- From discussion of function closures:
  - Implement (or analyse the code implementing) anonymous function definition and function application in a language with first-class functions.
  - Explain via an example how the power to "close over environments" turns out to provide tremendous power in programming languages.
    - In this case, the example we used was that we can get let-bindings and lists (cons cells) using nothing more than functions and function application once we have closures. The let-bindings really just take advantage of the fact that we can declare functions anywhere and that they bind parameters to values. The cons cells, however, take advantage of the fact that closures let us remember values for later use!
    - In other words: closures are (in PL theory) the fundamental data structure from which all other data structures can be built. Closures are awesome! Don't you wish you were lambda bound?

6 Some Practice Exercises

Recording practice exercises we mention over time.

- Implement a new syntactic form `count-bindings`. It takes a single argument—a symbol—and evaluates to the number of bindings for that symbol presently in the environment (so, more than 1 if there's both a binding and at least one shadowed binding).
- Create multiple-argument functions and applications in a surface language by desugaring them to one-argument functions and applications in the core.
- Assume you have a language with multiple-argument function definitions and applications as well as lists (cons, first, rest, and empty?)

in the core. Implement a new core form `apply` that takes a function expression and an argument list expression, checks that they evaluate to the appropriate types (function value and list), and then applies the function to the list of arguments. `apply` should give an arity (i.e., number of arguments) error if the length of the list of arguments does not match the length of the list of parameters for the function.

- Now, desugar as much of this as you can to a core that has only single-argument function definition and application. Discuss why the arity check, in particular, is **very** hard to manage in the desugaring. (Hint: when you apply a "two-argument function" to just one argument in this desugaring, the natural result is to get a function back waiting for the second argument. Is such a value illegal or unreasonable as a "normal value" in a language with first-class functions?)