2013W1-lecture19

October 25, 2013

Contents

1 Question of the Day

What does the following C program do?

```
#include <stdio.h>

int * get_local_addr() {
  int x;
  return &x;
}

int set_addr(int * addr) {
  int y = 0;
  *addr = 888888888;
  return y;
}

int main(int argc, char * argv[]) {
  int * a = get_local_addr();
  printf("%d\n", set_addr(a));
}
```

```
  SOLUTION
 The answer is "it depends".

 In C, variables are bound to memory locations, and those memory
 locations contain values.  However, C also provides operators =&= and
 =*= that allow us to compute the address a variable is bound to (=&=)
 and find or change the value at a given address (=*=).

 Because *values* are not statically scoped like variable bindings are,
 that means that the memory location of a variable can escape the
 static scope in which the variable is available.  The variable can be
 changed from outside of its scope.

 In C++, memory locations can also be reused.  In a practical language,
 we *must* be able to reuse memory locations.  Else, we can run out
 even if we never use too much memory at the same time in our program.

 Unfortunately, C++ allows the memory for a particular instance of a
```

```
local variable to be reused as soon as the function call that created
that instance finishes executing, *even if* the address of that
variable is still around!

In our code, first get the address of a local variable.  Then, we set
whatever the value at that address to =888888888=.  It so happens on
my computer with my compiler and this code that that sets the local
variable I created in the new function to =888888888=, but we can
certainly imagine much more damaging situations!

(For example, try changing =int y= to =char y= or =double y=, changing
the return types and the =%d= appropriately.  Did each do what you
thought they would?)
```

2 Logistics

2.1 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!

2.2 Neat PL research talk TOMORROW: 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.

2.3 Quiz!

3 From Expression Closures to Function Closures

PROBABLY DEFERRED to when we discuss recursion, since we already hit the big points.

Quick review of file:2013W1-lecture18.org

4 Mutation and State: A Brand New Flow (sort of)

(Note: we'll leave the "Church" part of "Church or State" for later. That's Alonzo Church and, particularly, his "Church numerals", that allow us to encode numbers using only function definition and application.)

4.1 Three Flows through the Program

Let's look at a program that uses `map` again:

```
(define (map f lox)
  (if (empty? lox)
      empty
      (cons (f (first lox))
            (map f (rest lox)))))

(define (label-size mid lon)
  (map (lambda (n)
         (cond [(< n mid) 'small]
          [(= n mid) 'medium]
          [(> n mid) 'large]))
   lon))
```

We've discussed three "flows" through our programs so far:

1. **Lexical**: The lexical "flow" defines the scope of variables in our programs. In our program, I know `label-size`'s parameters `mid` and `lon` are not bound in map, even though `label-size` calls map!

   If I didn't know that, I'd have to fear that any function called from one of my functions could change my own local variables. Even without side effects, this can be a problem, because other functions may rely on knowing the names and uses of my local variables, making changing the code for my function much more difficult and error-prone later on.

2. **Function-Call**: On the other hand, the flow of control in our program follows function calls, and we want it to do so! Otherwise, we couldn't call `map` from `label-size`, and `map` couldn't then call "back into" the `lambda` defined inside `label-size`. Instead, we'd have a language where the flow of control always follows the "shape" of the program—like our rather unimpressive "calculator" languages from the start of the term!

   Unfortunately, this flow is so fundamental to functions that it's hard to see. A more illustrative example is exceptions. In Java, for example, I might call two functions one after another:

   ```
   public static int mymethod() {
     int x = function1();
     return function2(x);
   }
   ```

   I expect to receive exceptions raised in either of the functions I call (or in functions they call) in `mymethod`. If `function1` raises an `IOException` but I know how to handle it in `mymethod`, I should be able to do so! That's because exceptions travel back up the function call chain, going backward along the flow we're discussing.

   On the other hand, if `function1` raises an `IOException`, I do **not** expect that exception to go on to `function2`. It either gets handled in the chain of function calls leading to `function1`—in this case, `function1`, mymethod, whatever function called my-method, whatever called that function, and so on—or it doesn't get handled at all.

Together these two ideas define our function-call flow.

3. **Persistent**: Our third flow through the program is persistent flow. We haven't discussed this one much, which might make it feel new, but it isn't really. Both our map example and our Java example illustrate it.

   In the map example, consider the symbol small, which is one of our values. It's produced within the lexical scope (the body) of label-size, yet it clearly passes through map as map produces a list of values small, medium, and big. So, the value doesn't just follow lexical scope.

   However, values also don't just follow function-call scope. We might write a piece of code to use label-size like:

```
(local [(define result (label-size 0 (list -5 5 0 1)))]
  (display result))
```

   Our value small makes it out of label-size but then goes back into the display function, yet display never called label-size, and label-size never called display. Instead, the value just persisted (continued) through the progress of the program after it was created.

   Similarly, function1 above computed a value and returned it to my-method. Clearly, that value isn't following lexical scope, because the local scope inside function1 is somewhere totally separate from the local scope inside my-method.

   But the value also didn't stop at my-method or flow back to the function that called it. Instead, we passed it along to function2.

   In fact, depending on how we write our code, a value we compute at any point in the run of our program can make it to any later point in our program. Values **can** persist throughout the life of our program!

4.2 State Changes Do Not Follow Lexical Flow

Hopefully you already figured out that I think state changes follow persistent flow, but let's show that it's true.

It's hard to even describe what we might mean by state changes only following lexical flow. That's because something like set! doesn't even have a nested "body" where this lexical flow can go. It only has the variable we're setting and the expression that gives the value for that variable:

```
(set! variable value-expr)
```

If we tried to give it a body:

```
(set! variable value-expr body-expr)
```

We'd end up with something that looks much more like a `let` than a `set!`!

4.3 State Changes Do Not Follow Function-Call Flow

Let's show that state changes do not follow function-call flow.

Consider the following code, assuming that `+` evaluates its actual parameters from left-to-right:

```
(let ((f (lambda (x) ...))
      (g (lambda (y) ...))
      (b (box 0)))
  (+ (f b) (g b)))
```

By the time we start evaluating the function `g`, the call to the function `f` has finished. So, the call to `g` is not part of the function-call chain that starts at `f`.

However, state changes in `f` can still be seen in `g`.

Write bodies for `f` and `g` so that `f` makes a state change that flows into `g`.

Reminder:

- `box` constructs a data structure that contains the value of its argument.
- `set-box!` evaluates its first argument to a box value and then replaces the box's contents with the value of its second argument.
- `unbox` evaluates to the contents of the box value that it gets from evaluating its argument.

```
 SOLUTION
Essentially anything that changes the value in b's box in f and then
accesses that value in g works:

(let ((f (lambda (x) (set-box! x 1)))
      (g (lambda (y) (unbox x)))
      (b (box 0)))
  (+ (f b) (g b)))
```

```
This evaluates to 2---not 1---because the change to x inside the
function f is visible inside the call to g.
```

4.4 How Can We Implement the Flow for State Changes?

Why reinvent the wheel? We already said values follow persistent flow. How do they do that?

How does the `1` in this code get from inside of `f` to inside of `g` in this code? In our interpreter?

```
(let (g (lambda (y) (+ y y)))
  (let (f (lambda (x) 1))
    (g (f 0))))
```

```
 SOLUTION
In the code, f RETURNS the value 1, which is then PASSED to g as a
parameter.  Interestingly, our interpreter uses the same solution to
let values flow where they will.

Look at the app case of our interpreter.  Here's a shortened version
of what I used to implement the Chapter 7 interpreter:

  [appC (fun-exp arg-exp)
    (local [(define fv (interp fun-exp env))       ;; Get the closure (fun value)
            (define av (interp arg-exp env))]      ;; Get the arg value
      (interp (closV-body fv)                      ;; Call the function!
              (extend-env (bind (closV-arg fv) av)
                          (closV-env fv))))]

At some point, we're evaluating (g (f 0)); so, fun-exp is g and
arg-exp is (f 0).

On the line marked "Get the arg value", we make a recursive call to
interp to evaluate (f 0).

In that recursive call, fun-exp is f and arg-exp is 0.  On the line
marked "Call the function!", we recursively call interp to evaluate
```

```
the body of f, which is just 1.  That call RETURNS the value 1.

So, values flow OUT of function calls through the return value of
interp.

That 1 is then returned (flows out) from the call to interp that
evaluated (f 0).

We're back to the line marked "Get the arg value" in the call to
interp where we're interpreting (g (f 0)), but now we have the value 1
in av.

How does g get access to the value 1?  Our interpreter evaluates the
body of g on the line marked "Call the function!".  It includes the
value 1 in the environment data structure it PASSES to the that
recursive call.

In other words, values can follow persistent flow because they're
PASSED IN AS ARGUMENTS to interp and also RETURNED AS RESULTS.
```

4.5 Implementing State Changes in our Interpreter

We'll call the data structure we use to record state changes the *store*. We need to **pass it as an argument** to `interp` and **return it as a result** from `interp` so that it can follow persistent flow.

We will pass in a store representing the state of the world just before the call to `interp` evaluates.

In turn, `interp` will return a store representing the state of the world just after the call to `interp` finishes evaluating.

4.5.1 Trivial Cases

We might know there are no changes to the store in a call to `interp`, in which case, we just give back the same store.

Fill in our `numC` case below.

(**WARNING**: For lecture, I use tuples to return two values from `interp`, which is what you might do in C++ with the templated `pair` class. To work with tuples in `plai-typed`, use `(values ... ...)` to create a tuple and `(define-values (name1 name2) ...)` in a `local`'s definitions to pull one apart. Both the ParselTongue core implementation and the textbook instead create a data structure to do the same thing, which is what you'd do in Java. There's no meaningful semantic difference.)

```
(define (interp [expr : ExprC] [env : Env] [store : Store]) : (Value * Store)
  (type-case ExprC expr
    ...
    [numC (n)                     ;; Fill me in!



            ]
    ...))
```

```
  SOLUTION
 (define (interp [expr : ExprC] [env : Env] [store : Store]) : (Value * Store)
   (type-case ExprC expr
     ...
     ;; The value is just the number n.
     ;; The new store is the same as the old one.
     [numC (n) (values (numV n) store)]
     ...))
```

4.5.2 "Threading the Store" through Recursive Calls

There might be no obvious changes in a call, but the call might make recursive calls that could themselves make changes.

Consider the beginC case in our interpreter (the equivalent of seqC in ParselTongue).

Changes to the store caused by evaluating the first expression had better be reflected in the store given to second expression! Otherwise, what was the point of the first expression at all?

Fill in the beginC case:

```
(define (interp [expr : ExprC] [env : Env] [store : Store]) : (Value * Store)
  (type-case ExprC expr
    ...
    [beginC (e1 e2)                     ;; Fill me in!
      (local [(define-value (e1-result e1-store) (interp e1 env store))]
          ]
    ...))

  SOLUTION
 (define (interp [expr : ExprC] [env : Env] [store : Store]) : (Value * Store)
   (type-case ExprC expr
     ...
     [beginC (e1 e2)
       (local [(define-values (e1-value e1-store) (interp e1 env store))]
         ;; We've now evaluated the first expression.
         ;; However, the variable store may be out of date!
         ;; Good thing we have e1-store:
         (interp e2 env e1-store))]
     ...))

 Notice how e1-store represents the state of the world just AFTER the
 first expression is evaluated and just BEFORE the second is evaluated.
 Also notice that it's NOT the store we return.  What store do we
 return?
```

4.5.3 Accessing and Changing the Store

Some syntactic constructs actually directly access or change the store. Since we're just using boxes, we'll focus on the most interesting box form: `setboxC`.

`setboxC` should evaluate its first expression to yield a box value. Then, it should evaluate its second expression to yield a value to put in the box. Finally, it should give back a store representing the state of the world in which the box's new contents are the new value.

Be careful! `setboxC` not only changes the store directly, it also makes two recursive calls to `interp`!

```
(define (interp [expr : ExprC] [env : Env] [store : Store]) : (Value * Store)
  (type-case ExprC expr
    ...
    [setboxC (be ve)                      ;; Fill me in!
```

```
          ]
   ...))
```

SOLUTION

```
(define (interp [expr : ExprC] [env : Env] [store : Store]) : (Value * Store)
  (type-case ExprC expr
    ...
    [setboxC (be ve)                      ;; Fill me in!
      (local [(define-values (box-val be-store) (interp be env store))]
        (local [(define-values (val ve-store) (interp ve env be-store))]
              ;; Why use be-store to evaluate ve?
              (define-values (val ve-store) (interp ve env be-store))]
         ;; Why update ve-store rather than store or be-store?
         (values val (update-store (cell (boxV-loc box-val) val)
                                   ve-store)))]
    ...))
```

SOLUTION

By the way, this is one case where I personally recommend:

Don't take advantage of the fact that locals let you do multiple definitions per line. Do just one definition and then nest another local. Why?

Because then you can call EVERY store variable "store". Why?

Well, each new declaration of store will shadow the old declarations. Why?

(This is starting to sound like a conversation with a two-year-old.)

(Why?)

Because then static scoping and shadowing ensure you never make the mistake of using an out-of-date store!

4.5.4 Closing Comment: Is This How State Is Implemented?

Of course not!!

What kind of crazy language would pretend to allow state changes but implement them in a pure functional style by secretly threading a value representing the current store through its computations?

(Cough cough, Haskell. Splutter, state monad.)

However, this does give us a clearer sense of how state changes flow through our program and what the **semantics** of state change are.

Remember how we said that dynamic scoping (which follows the simpler, more constrained function-call flow) is usually BAD for software engineering?

Discuss with your neighbor one good reason to use state change and one good reason to avoid using it.

```
 SOLUTION
Here's some thoughts.

State change often does a great job of representing the world,
particularly if you're representing something about the world that
authentically changes with time.  It's also a "cheap" way of
"communicating" between distant parts of your program.

But all that comes at a BIG cost.

Mutation or state change makes it HARD to reason about how programs
work.  In fact, compiler analyses will often try to effectively "break
up" a variable according to how its possible values move around (e.g.,
see Data-Flow Analysis) in order to isolate the impact of mutation.

Parallelism is especially impacted by mutation.  In the most efficient
parallel computations, individual processes solve their own problems
with no communication between them.  However, each state change in one
process that is (or, in some cases, *could be*) visible to another
mucks up that independent processing with communication and
synchronization.
```

5 What have we learned today?

- From QotD: An industrial programming language (in fact most!) has the same two-step lookup that we're modelling with our language with mutation: variables bound to "store" locations and values in the store locations. However, the fact that the store flows through the program by a different route than either variable scope (static) or the dynamic call structure of a program (corresponding to "dynamic scope") can cause big problems!

- State Change as a new-ish Flow ("Persistent" vs. "Lexical" or "Function-Call")
  - Justify with examples that the flow of state changes in a program does not match either of the flows we have so far discussed: lexical (i.e., the flow that static scoping follows) or function-call (i.e., the flow that dynamic scoping follows).
  - Implement this flow **without using mutation in the implementation** using store-threading style: passing a store representing the state before an operation as an extra parameter to that operation and returning a store representing the state after an operation as an extra result from that operation.
- Unnecessary State Change as a "Bad Thing"
  - Explain from a software engineering standpoint why (unnecessary or excessive) use of mutation can make software more difficult to reason about and change.

6 Some Practice Exercises

Recording practice exercises we mention over time.

- Imagine we wanted to introduce something sort of halfway between `let` and `set!`. We'll call it `sleet`. In a language that otherwise has static scoping, `(sleet id value-expr body-expr)` introduces a new **dynamic** binding for `id`. If a static binding for `id` already exists in the current scope, that produces an error. If a dynamic binding already exists, this `sleet` shadows it. (We'll assume we **cannot** use `set!` on `sleet`-bound variables.) An `id` reference will always be to a static binding if one exists; otherwise, it will find dynamic (`sleet`) bindings.
  - Use a core language with `sleet` to desugar `(deffun name funbody body)`, and discuss why it's easier than with `let`.

  - Imagine implementing `sleet` by passing **two** environments into `interp`, `static-env` and `dynamic-env`. Implement the `appC`, `funcC`, and `idC` cases for the interpreter, being sure to manage the two separate environments correctly. (Do closures need one environment or two??)
  - Describe any concerns you have about `sleet` from a software engineering perspective.

- Implement a new syntactic form `incr!` that takes a variable and adds one to its value.

- Implement a new syntactic form `time-maximum` that evaluates to the largest value a variable has ever contained. Discuss whether we could incorporate this into a mainstream language. (Could we do it with desugaring rather than changing the underlying language?)

- 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?)