Hints to PLC Assignment #5

Andreas Borchert

April 12, 2002

Question 1

Some were apparently confused by what $S$ really means and what semantic functions are in general. Perhaps a little Java notation helps. Let us define a Java class stub that is pretty close to $E = N + B$:

```java
public class Value {
    public Value(); // creates an undefined value
    public Value(int i); // creates an integer value
    public Value(boolean b); // creates a boolean value
    public boolean isDefined();
    public boolean isInteger();
    public boolean isBoolean();
    public int getIntVal();
    public int getBooleanVal();
}
```

See, an object of type $E$ can be either an integer, a boolean value, or $\perp$ (i.e. undefined). The operator $i \text{ in } E$ can be expressed as `new Value(i)`. Likewise you can inject Boolean values, i.e. `new Value(b)` is equivalent to $b \text{ in } E$. Note that instances of `Value` are never changed. All private instance variables can be assumed to be `final`. The notation $\eta \mid N$ is equivalent to $\eta.getIntVal()$.

Let us turn now to $S$, the symbol table that maps variable names to values, i.e. is of type $I \rightarrow E$:

```java
public class SymbolTable {
    public SymbolTable(); // default constructor, empty table
```
// [newval/varname] operator
public SymbolTable(SymbolTable old, String varname, Value newval);

// returns the value of the given variable name
public Value getValue(String varname);
}

Initially, you start with an empty symbol table which is constructed using the default constructor, i.e. SymbolTable s1 = new SymbolTable();. This initial symbol table returns undefined values for all possible variable names. If, for example, you have Value val = s1.getValue("i") then val.isDefined() will return false. Symbol tables cannot be modified but you can create a new symbol table that is based upon an old one with the exception of one variable name that gets a new value. In the mathematical notation we have the operator $\sigma_{x/i}$ which is equivalent to new SymbolTable($\sigma$, "i", x).

Given the Java notation, you can express the original semantic equation $E[E_1 + E_2] = \lambda \sigma. E[E_1][\sigma] + E[E_2][\sigma]$ on slide 108 in the following way in Java if you assume that the semantic function $E$ which resembles $E$ exists in several overloaded variants:

```java
class TupleES {
    public TupleES(Value e, SymbolTable s);
    public Value getValue();
}

class TupleES {
    public TupleES(Value e, SymbolTable s);
    public Value getValue();

    public static Value E(Expression e1, Expression e2, SymbolTable s) {
        return E(e1, s).getIntVal() + E(e2, s).getIntVal();
    }
}
```

BTW, you see the getIntVal method here. You can give it explicitly in the mathematical notation as well:

$$E[E_1 + E_2] = \lambda \sigma. E[E_1][\sigma] | N + E[E_2][\sigma] | N$$

Let us turn to question 1 of this assignment. Before, the semantic function $E$ just mapped an expression $Exp$ to a function $S \rightarrow E$ where $S$ represents the symbol table function $I \rightarrow E$ and $E$ the set of possible values. Now, that returned function returns a $E \times S$ tuple instead. In Java, we can define that as well:

```
class TupleES {
    public TupleES(Value e, SymbolTable s);
    public Value getValue();
}
```
public SymbolTable getSymbolTable();
}

In the mathematical notation, you can construct tuples with \langle x, \sigma \rangle which is equivalent to \text{new TupleES}(x, \sigma) in Java. In the mathematical notation you access the first element of a tuple using \downarrow 1 whereas in Java you would use getValue() instead.

It might help you to start to express the new semantic equation in Java first, i.e. you would need to flesh out following Java function:

```java
public static TupleES E(Expression e1, Expression e2, SymbolTable s) {
}
```

Note that all (overloaded) E functions return now TupleES instead of just Value. If you have this in Java notation, it should no longer be that difficult to translate it back to the mathematical notation which should be, in its outer form, close to
\[
E[E_1 + E_2] = \lambda \sigma. \langle \text{the value}, \text{the new symbol table} \rangle
\]

**Question 2**

The goal of this assignment is to show how privacy and a notation that is close to those of OO languages can be achieved through lexical closures. Here is a little example of a counter object that maintains an internal state and provides two mutators increment and decrement and one accessor value:

```scheme
(define (create-counter)
  (define value 0)
  (define (increment)
    (set! value (+ value 1)))
  (define (decrement)
    (set! value (- value 1)))
  (lambda (method)
    (case method
       ((increment) (increment))
       ((decrement) (decrement))
       ((value) value))))
```
And here is a sample session that demonstrates how it can be used:

```
doolin$ guile
  guile> (load "counter.scm")
  guile> (define c1 (create-counter))
  guile> (c1 'value)
    0
  guile> (c1 'increment)
  guile> (c1 'increment)
  guile> (c1 'value)
    2
  guile> (c1 'decrement)
  guile> (c1 'value)
    1
  guile> (define c2 (create-counter))
  guile> (c2 'value)
    0
  guile> (c2 'decrement)
  guile> (c2 'value)
    -1
  guile> (c1 'value)
    1
  guile> doolin$
  doolin$
```

Note that `increment` quotes the symbol `increment` which is necessary in a procedure call like `(c1 'increment)` to prevent the Scheme interpreter from looking up the current binding of `increment` (which most likely does not exist). However, in the `case` construct we must not quote the tags as they are not evaluated but just compared to.

There were some questions about the semantics of the `push` and `pop` operations. `push` has to store the current position and the current direction onto the stack. Consequently, `pop` restores both from the stack. The stack is best implemented on base of a list.

Another question was about the initial direction. I told originally in the writeup that the initial direction is upward, which would be a $\Delta = (0, 1)$. However, I found out later that at least in the diagrams on the slides 14 to 17, the initial direction was downward, i.e. $\Delta = (0, -1)$. Whatever you take, it will be accepted.