Meeting 21: Mutation

Announcements

Lab 5
March-April 2014

<table>
<thead>
<tr>
<th>Su</th>
<th>Mo</th>
<th>Tu</th>
<th>We</th>
<th>Th</th>
<th>Fr</th>
<th>Sa</th>
</tr>
</thead>
<tbody>
<tr>
<td>23</td>
<td>24</td>
<td>25: Spring Break</td>
<td>26</td>
<td>27: Spring Break</td>
<td>28</td>
<td>29</td>
</tr>
<tr>
<td>30</td>
<td>31</td>
<td>1</td>
<td>2</td>
<td>3</td>
<td>4</td>
<td>5</td>
</tr>
<tr>
<td>6: Lab 5 Due</td>
<td>7</td>
<td>8: Quiz 5</td>
<td>9</td>
<td>10</td>
<td>11</td>
<td>12</td>
</tr>
</tbody>
</table>

Leaderboard
http://csci3155.cs.colorado.edu/leaderboard.php

Quiz 4
Mean: 17.6, Stdev: 2.6
Median: 18, 3rd Quartile: 19.25, Max: 20

Lab 5 Feature List

1. null
2. var declarations, var assignment, field assignment
3. functions with var mode
4. functions with ref mode
5. functions with name mode (subst calling avoidCapture)
Lab 5 Implementation Notes

What is DoWith[W,R]?
It is a data structure that encapsulates a function of type \( W \rightarrow (W,R) \). The \( W \) is the "with" type that is the input-output type of the function, while \( R \) is the "result" type of the function. Note that the \( W \) type parameter is used in those two places in the function type.

For us, we care only about \( \text{DoWith}[\text{Mem}, R] \) for some result type \( R \). And most of the time, we care about \( \text{DoWith}[\text{Mem}, \text{Expr}] \) (where \( R \) is \( \text{Expr} \)).

Why is it useful to encapsulate \( W \rightarrow (W,R) \)?
The purpose of encapsulating a function of this type is that we can define operations on it to handle "boilerplate". In this case, the \( W \) type (or in our case the \( \text{Mem} \) type) should be "hidden" and simply threaded through. That's what the \( \text{DoWith}[W,R] \) abstraction provides.

Let's focus on map.

```scala
sealed class DoWith[W,R](doer: W => (W,R)) {
  def map[B](f: R => B): DoWith[W,B] = new DoWith[W,B]({
    (w: W) => {
      val (wp, r) = doer(w)
      (wp, f(r))
    }
  })
}
```

The `map` method allows us to manipulate the result \( R \) part while leaving the \( W \) part alone. This is precisely the boilerplate that we factor out.

The `flatMap` method is analogous to `map` but allows the callback \( f \) to return a `DoWith`.

We then define 3 other method as conveniences for working with `DoWith` values.
The **doget** helper encapsulates a function that just returns a pair with its input \( w \) in both positions. Conceptually, this exposes the \( W \) input value because it puts it into the second position of the pair.

The **doreturn** helper encapsulates a function that always returns its input \( w \) paired with the given result \( r \).

The **domodify** helper encapsulates a function that applies a callback \( f \) to the \( W \) part.

Note that the code shown above is the direct construction of a **DoWith** versus the commented code that does the same thing using just **doget** and then applying **map**. It might be a bit easier to read the directly constructed versions, but they are the same.

### Creating DoWith\[Mem,Expr\] in step

The intent is that **doget**, **doreturn**, **domodify**, **map**, and **flatMap** are all you need when you implement **step**. You never need to create a **DoWith** explicitly with **new** (in fact, I advise against it).

### A Search rule

Consider the **Unary** case.

```scala
case Unary(uop, e1) =>
  step(e1) map { e1p => Unary(uop, e1p) }
  /* for (e1p <- step(e1)) yield Unary(uop, e1p) */
```

We recursively call **step**, which gives us a **DoWith\[Mem,Expr\]**. We want to leave alone the **Mem** to **Mem** part, but we need to change the result **Expr** encapsulated there. To do so, we use a **map**. Note that this code never needs to explicitly mention the **Mem** part. The pass-through is exactly what we want.

The for-comprehension desugars to the **map** call. My suggestion is to not use the for-comprehensions unless you’ve become totally comfortable with what the desugaring is.

### DoVar
Here's the signature for `Mem.alloc`.

```scala
object Mem {
  def alloc(v: Expr): DoWith[Mem, A]
}
```

So it returns a `DoWith[Mem, A]`, the "result" component being the freshly allocated address.

Thus, to implement `DoVar`, we need to grab that address via `map` and return with the appropriate call to `substitute`.

```scala
case Decl(MVar, x, v1, e2) if isValue(v1) =>
  Mem.alloc(v1) map { a => substitute(e2, Unary(Deref, a), x) }
/* for (a <- Mem.alloc(v1)) yield substitute(e2, Unary(Deref, a), x) */
```