
CS 421 – Spring 2007

Lecture Notes Set 36:

Data Abstraction

Elsa L. Gunter¹

Transcribed by: Pooja Mathur

Corresponding to Slides: 25-infinite-data (slides 24-42)

Made available: April 25, 2007

Revision 1.0

1 Change Log

1.0 Initial Release.

2 Infinite Data

2.1 Call by Name (lazy) (slides 24-26)

We ended last time, before we got to the side effects of this evaluation. So the side effects of call by name can be kinda nasty. Let's take a look to further understand why. If we had the following code (note it's in c, not OCaml)

```
int x = 5;
int square(int a) {return a * a}
```

If we tried to run `square(++x)` with call by value semantics, we would get $6 * 6 = 36$. However, what happens with call by name? We evaluate `++x` once and get 6...for the first `a`, but now the program thinks `x` is 6 so we do `++x` on 6 and get 7...so our program ends up telling us that 6 squared is 42... So, you need to be aware of effects like this when writing code.

Even with this problem, the most useful thing is that it terminates if possible. However, if there is a very expensive expression that does need to be evaluated, call by name is better.

2.2 Call by Need (slides 27-36)

The last evaluation method we will look at is call by need. This attempts to get the best of both worlds. Here you still don't evaluate your argument until it is needed..but once you do evaluate it, it stays saved.

So let's look at the same example we used before. let $f\ x = (x+1)*(x+1)$ and we call $f\ ((3*4)/6)$

```
f ((3*4)/6)
(<thunk>+1)*(<thunk>+1)
(2+1)*(<thunk>+1)
(3)*(<thunk>+1)
(3)*(2+1)
(3)*(3)
9
```

So the idea here is that we replace the arguments with 'thunks'. The thunks are suspensions of evaluating the expression. So we have two thunks in place. In the 3rd step, we need the value of the thunk, so then and only then do we evaluate the thunk. So we do the computation of $((3*4)/6)$ and get 2 (note that we do this work at the side and the

¹© 2007, Share and Enjoy

value is saved now). So then we finish the $2 + 1$ and now we have to evaluate the other side. But we already evaluate the thunk, so we don't waste any time and just get the 2 back and move on by finishing the evaluation.

So, what are the side-effects of this? Well, with thunks, you end up creating references...like pointers. And this these pointers, you can dereference them you use the `!` operator. So in the slides we have `let counter = ref 0`. That means we have a pointer named counter that is points to a memory location that holds the value 0. Then dereference the pointer by doing `!counter`. So to add 1 to counter you can't write `counter + 1`, you need to write `!counter + 1`.

3 Thunks

3.1 Implementation (slides 37-39)

To do this, we need three things, we a place to hold data/expression, a suspend to stop the evaluation of the data/expression until we need it, an then something to remove the suspend, so we can get the value.

So our type thunk will tell us if we have a value or a suspended expression.

```
# type a thunk_type = Value of a | Susp of (unit → a);;
```

Next we need the a function that takes an argument and creates a pointer to it. From there, we figure out if we are putting a value in the place the pointer is pointing to, or are we putting a suspend. If we are putting a suspend, then we need to lambda lift the expression so not to evaluate it yet and then that expression gets pointed to by the pointer. Note that we can have nothing to evaluate, so it could be possible to simply place a value in the thunk.

```
# let delay f =  
  let thunk = ref (Susp f) in  
  fun () → match (!thunk) with  
  | Value a → a  
  | Susp f → let result = f () in  
  (thunk := (Value result); result);;
```

The to evaluate the expression in the suspend, we need a function to force the evaluation. Since our lambda lifted expression is only held up by unit, we give the expression a unit and then it can be evaluated. So that would look like:

```
# let force f = f ();;
```

So to use this with our example... `let f x = (x+1)*(x+1)` and `f ((3*4)/6)`... we would instead write... `let f x = ((force x)+1)*((force x)+1)` and `f (delay (fun () → ((3*4)/6)))`. That is we lambda lift `((3*4)/6)` and then apply `f` to it. Applying `f` ends up forcing the argument that we just lambda lifted.

3.2 Thunks in OCaml (slide 40)

Luckily for us, we don't have to write this code ourselves. There is a lazy library in OCaml that does this. So instead of the above `let f x = ((force x)+1)*((force x)+1)` and `f (delay (fun () → ((3*4)/6)))` we could write `let f x = ((Lazy.force x)+1)*((Lazy.force x)+1)` and `f (lazy ((3*4)/6))`. So `lazy` is a constructor that takes an argument which can be a value or expression. It makes a thunk out of it. And then to evaluate it, you call `Lazy.force` on the pointer.

4 Infinite Data (slides 41-42)

What is great about thunks and the way it evaluates things is that it can actually work in infinite data sets. If you had an infinite list of ones (`let rec ones = 1::ones;;`) and you wanted to perform some function on every element but only wanted to return the first three elements... with call by value, you would have to perform the function on every element, and that would never end. With call by name, you set up the evaluation three times, and then evaluate it three times... So you get what you want, but you wasted time evaluating the same thing numerous times. With thunks, you perform the evaluation once, and then you can just send that value back three times and your done. So it's the best options.