FUNCTIONAL PROGRAMMING (2)
PROF. SIMON PARSONS

- Last lecture I introduced functional programming using the language HOPE.
- This allows us to write functions like:

```
dec map : (alpha -> beta) # list(alpha)
                  -> list(beta);
--- map(f, nil) <= nil;
--- map(f, x :: l) <= f(x) :: map(f, l);
```

- This lecture will look at the mathematical underpinning of functional languages.
- But first we will look at some more typical things one can do with them.

- One important property of functional languages is the way they handle parameters.
- We are used to passing parameters using *call-by-value*.
- You may even have come across *call-by-reference*.
- Call by value is good for efficiency.
- It may, however, result in redundant computation.
- So we also consider *call-by-need* where arguments are passed unevaluated and evaluated when required.

- The advantage of call-by-need is that we waste no computation.
- The disadvantage is the expense of implementation.
- In general with functional languages we distinguish between *eager* and *lazy* evaluation.
- Eager evaluation does everything as soon as possible without worrying whether it is useful.
- Lazy evaluation only does things when absolutely necessary.
- They roughly correspond to call-by-value and call-by-need.

- However, there is more to eagerness and laziness than just these efficiency issues.
- Their effect permeates the whole of functional programming.
- In particular they determine what it is possible to do at the extreme limits of the language.
- (For example, handling infinite data structures.)
- Let's start by considering the limitations that *strictness* imposes on a language.

Strictness

- The function + is strict.
- It requires that both its arguments are known before it can be called.
- More precisely any function which requires at least one of its arguments have known value before it can be called is strict.
- + is strict in both arguments.
- Some functions work perfectly well without being strict.

- One such is:

```
--- f(x, y) <= if x < 10 then x else y
```

- This does not always need the value of `y`, but does need the value of `x`.
- It is thus strict in `x`.
- And non-strict in `y`.
- Thus when `x` < 10, we waste computation evaluating `y` since there is no need to know its value.

- As an extreme, consider:

  `f(4, <non-terminating expression>)`

- Here, an eager implementation would cause the program to fail, whereas a lazy one would give us 4.
- Of course, this does not help us when we have:

  `f(<non-terminating expression>, 4)`

- To see how laziness can eliminate redundant computation, consider the function:

```
dec reduce : (alpha # beta -> beta)
     # beta # list(alpha) -> beta
--- reduce(f, b, nil) <= b;
--- reduce(f, b, x::l) <= f(x, reduce(f, b, l));
```

- This function takes a function as an argument and applies it to reduce a list to a single element.
- b is what you use as an argument when you get to the end of the list.
- Thus `reduce(+, 0, L)` sums the elements of `L` and `reduce(*, 1, L)` computes their product.

- Writing `b` for:

  ```
  lambda(el, isthere) => if isthere then true
    else (1 = el)
  ```

  we can use `reduce(b, false, List)` to test if 1 is in `List`.
- Consider doing this for the list `[1, 3, 5, 7]`.
- For eager evaluation we would get:

```
reduce(b, false, [1, 3, 5, 7]
b(1, b(3, b(5, b(7, false))))
b(1, b(3, b(5, false)))
b(1, b(3, false))
b(1, false)
true
```

- If the implementation were lazy, we would get:

```
reduce(b, false, [1, 3, 5, 7]
b(1, b(3, b(5, b(7, false))))
true
```

since in this case the second argument never has to be evaluated.

Lambda calculus

- The lambda calculus is the calculus of anonymous functions.
- It provides a means of representing functions and a means of transforming them.
- Let's consider a very simple function:

```
--- double(x) <= 2 * x
```

- We write this in lambda notation as:

$$\lambda x. * 2\ x$$

- Dropping the name anonymises the function.
- Note that we use the prefix form of the * function.

- We read this lambda expression as follows.
- The λ we read as "The function of".
- The dot we read as "which returns".
- So, the whole thing is:

  The function of x which returns x times 2.
- Of course it is very similar to:

```
lambda x => 2 * x
```

- The x in the *lambda abstraction* is called the *bound variable*.
- This corresponds to the idea of a formal parameter.
- The bit of the lambda abstraction to the right of the dot is the *body*.
- The body can be any valid lambda expression, so it can be another lambda abstraction.

$$\lambda x.\lambda y. * (+x\ y)2$$

" The function of x which returns the function of y which returns the sum of x and y multiplied by 2."

- This is just the lambda calculus version of:

  ```
  lambda x => lambda y => (x + y) * 2
  ```

- All lambda calculus functions have just a single argument.
- So multi-argument functions become multiple applications of single-argument functions.
- This is known as "currying".
- Although we should write brackets between the different functions:

  $$(\lambda x.(\lambda y. * (+x\ y)2))$$

  by convention we don't.

- When we call a lambda function we place it in brackets before its argument.
- Thus calling:

  $$\lambda x. * 2\ x$$

  on the value 4 is done by writing:

  $$(\lambda x. * 2\ x)4$$

  and we call:

  $$(\lambda x.(\lambda y. * (+x\ y)2))$$

  with y as 2 and x as 3 by:

  $$(\lambda x.(\lambda y. * (+x\ y)2)\ 2)\ 3$$

- This is all there is to the syntax of the lambda calculus.
- The BNF is:

```
<exp> ::=λ<id>.<exp>|<id>
          |<exp><exp>|(<exp>)|<con>

<id>  ::= any identifier
<con> ::= constant
```

- There is a suprising amount that you can put together with such a simple syntax.

- The syntax shows us how to build valid lambda expressions.
- But how do we evaluate them?
- We have *conversion rules* which do this.
- The first rule is the simplest.
- Constants evaluate to constants.
- Other functions are reduced using the δ-rules.
- These allow us to replace function applications with their values.

- For example:

$$+1\ 3 \rightarrow_\delta 4$$

  we read this as "+ 1 3 *reduces to* 4".

- To do this we have to have the arguments of the function be themselves already reduced.
- So we cannot directly reduce:

$$*(+1\ 2)(-4\ 1)$$

- Instead we have to reduce each argument of the outer * first.

- So we have:

$$
\begin{aligned}
& *(+1\ 2)(-4\ 1) \\
\rightarrow_\delta\ & (+1\ 2)3 \\
\rightarrow_\delta\ & *3\ 3 \\
\rightarrow_\delta\ & 9
\end{aligned}
$$

- This reduction then evaluates simple functions.
- To evaluate lambda abstractions we need a *β-reduction*

- A β-reduction replaces the value of the bound variables with the values they are called with.
- Thus evaluating:

  $$(\lambda x. * \; x \; x)2$$

  we have:

  $$\begin{aligned}(\lambda x. * \; x \; x)2 &\rightarrow_\beta \; *2 \; 2 \\ &\rightarrow_\delta \; 4\end{aligned}$$

- This kind of reduction might end up having to be repeated.

- For example:

$$((\lambda x.\lambda y. + x\ y)7)8)$$

  will reduce as:

$$\begin{aligned}((\lambda x.\lambda y. + x\ y)7)8) &\rightarrow_\beta (\lambda y. + 7\ y)8 \\ &\rightarrow_\beta +7\ 8 \\ &\rightarrow_\delta 15\end{aligned}$$

- However, we have to be careful in doing this.

- Consider:

$$\lambda x.(\lambda x.x)(+1\ x)$$

- Here we have two distinct xs.
- There is the inner one, in the $(+1\ x)$, and the one referred to in the outer λx.
- Thus it would be wrong to do a β-reduction for the argument 1 as:

$$\begin{aligned}&(\lambda x.(\lambda x.x)(+1\ x))1\\ &\rightarrow\ (\lambda x.1)(+1\ 1)\\ &\rightarrow\ 1\end{aligned}$$

- We have to be careful not to substitute inside an abstraction if the bound variable has the same name as the variable being substituted.
- The easiest way to do avoid this is to do α-conversion.
- This is to rename the bound variables to make them have different names.
- The same thing as standardising variables in logic.
- Applying this to our previous example, gives:

$$\lambda x.(\lambda y.y)(+1\ x)$$

- There is one other form of reduction
- η-reduction allows the reduction:

$$\lambda x.E\ x \rightarrow_\eta E$$

  if x does not occur free in E since:

$$(\lambda x.E\ x)A$$

  is just

$$E\ A$$

- This is not widely used.
- (It is a compile-time feature rather than a run-time one.)

- Sometimes we have a choice in the order we apply reductions.
- This can have a big impact on the result.
- Consider:

$$(\lambda x.\lambda y.y)((\lambda z.z\ z)(\lambda z.z\ z))$$

- Here we end up either trying to reduce:

$$(\lambda z.z\ z)(\lambda z.z\ z)$$

or

$$(\lambda x.\lambda y.y)((\lambda z.z\ z)(\lambda z.z\ z))$$

- If we try to reduce the first one we get:

$$\begin{aligned}
&(\lambda z.z\ z)(\lambda z.z\ z)\\
&\rightarrow\ (\lambda z.z\ z)(\lambda z.z\ z)\\
&\rightarrow\ (\lambda z.z\ z)(\lambda z.z\ z)\\
&\vdots
\end{aligned}$$

  which does not terminate.

- IF we reduce the other we get:

$$\begin{aligned}
&(\lambda x.\lambda y.y)((\lambda z.z\ z)(\lambda z.z\ z))\\
&\rightarrow\ \lambda y.y
\end{aligned}$$

  which does terminate.

- So, order matters.

- We call the expression being reduced the *redex*.
- Then we have:
- The *leftmost* redex is the one whose λ is to the left of all other redexes in the expression.
- The rightmost is similarly defined.
- The *outermost* redex is a redex not contained in any other redex.
- The *innermost* redex is one which contains no other.
- We can then define two ways of reducing expressions.

- *Applicative order reduction* (AOR) says you should always reduce the leftmost innermost redex first.
- *Normative order reduction* (NOR) says you should always reduce the leftmost outermost redex first.
- So in our example,

  $$(\lambda z.z\ z)(\lambda z.z\ z)$$

  is the leftmost innermost, and

  $$(\lambda x.\lambda y.y)((\lambda z.z\ z)(\lambda z.z\ z))$$

  is the leftmost outermost.

- Thus AOR will try to evaluate:

  $$(\lambda z.z\ z)(\lambda z.z\ z)$$

  and so fail to terminate, while:

  $$(\lambda x.\lambda y.y)((\lambda z.z\ z)(\lambda z.z\ z))$$

  will be evaluated by normal order reduction and this will terminate.
- Do not infer from this that NOR is better than AOR.
- While NOR plays safe, and avoids evaluating any expressions until it has to, AOR is more efficient on conventional computers.
- Of course, AOR is related to eager evaluation, and NOR to lazy evaluation.

- A lambda expression is said to be in *normal form* when it can be reduced no more.
- Thus:

  $$\lambda y.y$$

  is the normal form of:

  $$(\lambda x.\lambda y.y)((\lambda z.z\ z)(\lambda z.z\ z))$$

- It turns out that there is something special about the normal form, and normal order reduction.
- Basically we can use NOR to get the same normal form expression whatever way we do the reduction.

- The Church-Rosser theorem has as a consequence:

  If an expression E can be reduced in two different ways to two normal forms, then these normal forms are the same up to alphabetical equivalence.

- The last bit means you can change variable names to make them identical.

- The standardization theorem says:

  If an expression E has a normal form then reducing the leftmost outermost redex at each stage in the reduction of E guarantees to reach that normal form (up to alphabetical equivalence).

- Thus the normal form is unique.
- (Which is exactly what we want— it would be a shame if different implementations of the same function could give us different results.)
- We also have the diamond property for reductions.

  If you can reduce E to $E1$ and $E2$ by applying any reduction operation several times, then by applying the same operation some more, you can reduce both $E1$ and $E2$ down to some expression N.