The Lambda-calculus

There are many good and concise tutorials to the lambda calculus, for example Graham Hutton's.

Lambda calculus was the result of Alonzo Church's attempt to formalise the concept of computability using a simple notion of *functions*.

One of the ways we can understand functions is as a mapping from some inputs to some outputs. For example the function *add 1 to the input* in mathematics is written as

$`x \mapsto x + 1`$

Functions can have several variables, for example the function computing *the average of its inputs*:

$`(x, y) \mapsto (x + y) / 2`$

Functions can also take other other functions as arguments, for example the *argmax* function which returns the value of the argument for which the the result is the maximum (*argmax*):

$`f \mapsto x \mathrm{\,such\, that\,}\forall x'.f(x') \leq f(x)`$

Operations such as *computing the derivative* ($`f \mapsto \frac{df}`

`{dx}` $) or *integration* ($ `f \mapsto \int_a^b f(x) \mathrm dx` $) can be thought of as functions taking functions as input and producing functions as output.

All these needed to be formalised uniformly, and in a way that was *clearly computable*, that is executable in a mechanical way.

Church's brilliant idea was to define a simple function calculus with only variables, function definition, and function application:

- Function application: `f x` or `f (x)`.
- Function definition: `\x -> F`, where `F` is another "lambda term", i.e. an expression formed of variables, function application, and function definition.

> ***Note:*** *the original syntax for function defintion is $* `\lambda x. F` *$, hence the term "lambda calculus". We will use directly the Agda syntax, with* `\` *instead of $* `\lambda` *$ and* `->` *instead of ".". One of Church's PHd students, Dana Scott, who was a major contributors to the mathematical foundations of programming language visited Birmingham a few years ago and told us the story of how the lambda calculus got its name (the name has no significance).*

<iframe width="560" height="315" src="https://www.youtube.com/embed/juXwu0Nqc3I" frameborder="0" allow="accelerometer; autoplay; encrypted-media; gyroscope; picture-in-picture" allowfullscreen></iframe>

Example: the two functions above are

```
\x -> x + 1
\x -> \y -> x + y
```

The second one can be equivalently written more compactly as

```
\x y -> x + y
```

The way functions are used is just like in mathematics. The expression `(\x -> ...x...) E` becomes `...E...`. That is all (with the technical caveat that sometimes local variables need to have their names changed so that they do not clash with variables with the same name but different scope).

For example:

```
(\x -> x + 1) 7
  = 7 + 1
  = 8

(\x y -> x + y) 7 8
  = (\y -> 7 + y) 8
  = 7 + 8
  = 15
```

```
(\x -> x + 1) ((\x -> x + 1) 2)
  = (\x -> x + 1) (2 + 1)
  = (\x -> x + 1) 3
  = 3 + 1
  = 4
```

Let us now look at some “higher-order” functions, i.e. functions that take functions as arguments. For example the function which applies its function-argument twice:
`(\f x -> f (f x))`.

Let us use this function on our earlier example:

```
(\f x -> f (f x)) (\x -> x + 1)
  = (\f x -> f (f x)) (\y -> y + 1)
  = \x -> (\y -> y + 1) ((\y -> y + 1) x)
  = \x -> (\y -> y + 1) (x + 1)
  = \x -> (x + 1) + 1
  = \x -> x + 2
```

Unsurprisingly, applying the function `\x -> x + 1` twice is the same as `\x -> x + 2`.

Proof terms

The lambda calculus allows us to eliminate `where` clauses altogether from proof terms. Consider for example the proof term (which is also a

program) for `LEM`:

```
LEM : {P : Prop} -> P or (P -> Falsity)
LEM {P} = DNE g where
  g : (P or (P -> Falsity) -> Falsity) -> Falsity
  g z = z (orIr v) where
    v : P → Falsity
    v x = z (orIl x)
```

Note that in the last clause we are defining _the function which takes `x` as an argument and returns `x (orIL x)` as a result, which is simply `\x -> z (orIl x)`. If we substitute `v` for its definition we get:

```
LEM : {P : Prop} -> P or (P -> Falsity)
LEM {P} = DNE g where
  g : (P or (P -> Falsity) -> Falsity) -> Falsity
  g z = z (orIr (\x -> z (orIl x)))
```

Similarly, we can inline the definition of the function `g` using a lambda term:

```
LEM : {P : Prop} -> P or (P -> Falsity)
LEM = DNE (\z -> z (orIr (\x -> z (orIl x))))
```

Using lambda forms we can replace any definition with `where` clauses with a one-line definitions (this doesn't necessarily make them more

readable, on the contrary).

Lambda calculus in practical programming

The lambda calculus is the basis of all functional languages such as Haskell, OCaml, Scala, etc. However, lambda calculus is now a part of many non-functional languages such as Java, C++, Python, etc. The reason is that defining *anonymous functions* is sometimes a convenience, just like anonymous classes are a convenience, most often in the definition of *callback functions*.

Lambdas in Java

In Java using lambdas avoids the use of anonymous classes or the "adaptor pattern". Consider for example (numeric) integration:

```
public static double integrate(DoubleFunction df, double a
, double b, int n)
```

To call it without lambdas we must use the adaptor pattern:

```
import com.softmoore.math.DoubleFunction;
public class DoubleFunctionSineAdapter implements DoubleFu
nction
  {
    public double f(double x)
```

```
      {
        return Math.sin(x);
      }
  }

...
DoubleFunctionSineAdapter sine = new DoubleFunctionSineAda
pter();
double result = Simpson.integrate(sine, 0, Math.PI, 30);
```

Anonymous classes are slightly more concise:

```
DoubleFunction sineAdapter = new DoubleFunction()
  {
    public double f(double x)
      {
        return Math.sin(x);
      }
  };
...
double result = Simpson.integrate(sineAdapter, 0, Math.PI,
 30);
```

whereas with lambdas we simply write:

```
DoubleFunction sine = (double x) -> Math.sin(x);
double result = Simpson.integrate(sine, 0, Math.PI, 30);
```

Church encodings

What is special about this calculus is that despite its utter simplicity a lot of mathematics can be encoded in it, for example boolean algebra and integer arithmetic. We don't need any built-in constants.

Let us check out the encoding of the Booleans in the lambda calculus. It may seem weird to encode *values* as *functions* but bear with it.

```
true  = \a b -> a
false = \a b -> b
and   = \p q -> p q p
or    = \p q -> p p q
```

Let us check some of the equations, such as `and true false = false`:

```
and true false
  = (\p q -> p q p) true false
  = true false true
```

This may seem crazy, but remember, `true` and `false` are encoded *as functions*!

```
  = (\a b -> a) false true
  = false
```

Similar calculuations will confirm that all desirable properties hold. This is remarkably clever!

Negation

One way to define negation is:

```
not = \p a b -> p b a
```

In this case

```
not true = (\p a b -> p b a) true
  = \a b -> true b a
  = \a b -> (\x y -> x) b a
  = \a b -> b
  = false
```

> *Note: For negation, the defintion might fail if we don't always evaluate the argument before the function.*

Integers

Check out the Wikipedia page to see how integers can be encoded into the lambda calculus.

The Y combinator

The functions definable by the lambda calculus are *partial*. Consider the function

```
omega = (\x -> x x)(\x -> x x)
  = (\x -> x x)(\x -> x x)
  = ...
```

or

```
omega' = (\x -> x x x)(\x -> x x x)
 = (\x -> x x x)(\x -> x x x)(\x -> x x x)
 = (\x -> x x x)(\x -> x x x)(\x -> x x x)(\x -> x x x)
 = (\x -> x x x)(\x -> x x x)(\x -> x x x)(\x -> x x x)(\x
 -> x x x)
 = ...
```

Because the lambda calculus is equivalent to the Turing machine, we cannot computationally determine whether a lambda term terminates evaluating or not.

One particularly interesting such term is the so-called *Y combinator* which encodes *recursion*:

```
Y = \f -> (\x -> f (x x)) (\x -> f (x x))
```

For any function `g` we have that

```
Y g = (\x -> g (x x)) (\x -> g (x x))
 = g (((\x -> g (x x)))((\x -> g (x x))))
 = g (Y g)
 ...
 = g (g (Y g))
 ...
 = g (g (g (Y g)))
 ...
```

Lambdas and recursion in Agda.

Just like DNE violates the Curry-Howard correspondence from the logical side, so do terms like `omega` and `Y` from the functional side. The programming language Agda, which is based on the CH correspondence, bans such terms, which lead to errors.

Recursive functions can also be defined without lambdas:

```
unsafe a = unsafe a
```

noting that for any argument we have that

```
unsafe 0 = unsafe 0 = unsafe 0 = ...
```

Such functions are disallowed. In general any function that Agda cannot determine that it is terminating is disallowed. The reason they are

disallowed is that non-terminating functions can have *any type*.

We can temporarily suspend Agda's termination checker and, indeed:

```
{-# TERMINATING #-}
unsafe : {A : Set} -> A -> Void
unsafe a = unsafe a
```

Through the CH correspondence `unsafe` is a proof that `A -> Falsity`, making the system inconsistent.

Exercises

- Revisit the proofs from Assignment 1 and write them as lambda terms.
- Using the Church encoding check that indeed `plus one one` is equal to `two`.