Unit 1g: Computation

1 Computation

The theory of *computation* deals with how efficiently problems can be solved on a *model of computation*, using an algorithm – asks "What are the fundamental capabilities and limitations of computers? "

A model of computation is the definition of the set of allowable operations used in computation and their respective costs – it is essentially a *theoretical "computer"*. Models of computation were considered long before practical computers were common, notably:

- *Turing Machine*, a hypothetical machine devised by the mathematician Alan Turing in the 1930s – computation by performing instructions
- *Lambda Calculus*, developed by another mathematician, Alonzo Church, also in the 1930s – computation by applying functions

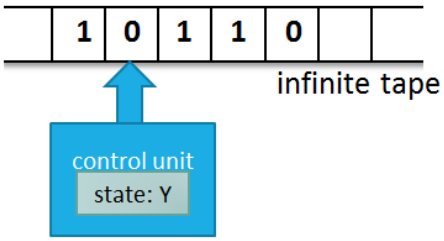
Both can simulate ANY computer algorithm, no matter how complicated it is: they are *equivalent* – Church-Turing thesis.

2 Turing Machine

The "Machine" consists of an infinitely-long tape which acts like the memory in a typical real computer. The tape consists of squares that can be written with symbols. It also has a control unit that can be in one of a finite number of states. At any one time, the machine has a head which is positioned over one of the squares on the tape. With this head, the machine can perform some very basic operations:

- read the symbol on the square under the head
- edit the symbol by writing a new symbol or erasing it
- move the tape left or right by one square

- change the state of the machine



Turing reasoned that any computation that could be performed by a human involved writing down intermediate results, reading them back and carrying out actions that depend only on what has been read and the current state of things.

Note that a Turing Machine is not a real device; it's just a way of describing a set of *allowable operations* but see www.youtube.com/watch?v=E3keLeMwfHY#t=33 for an example of a 'real' Turing machine.

2.1 Turing Machine computation - example

How would a Turing machine add 1 to a number?

First, how do we *represent* the number? - could be *binary* with symbols 1 and 0 on the tape.

Then, what is the algorithm?

- Start from right and *move to left* one digit at a time (that is, move the tape to the right past the head)
- If 1 change to 0
- If 0 change to 1 and stop – we have result.

example:

01011 (decimal 11)
01010
01000
01100 (stop – now have decimal 12)

Does this always work? Try some other examples, e.g. 01100, 01111. Why do we need to have the leading 0 when representing the number?

Subtraction is similar, though slightly more complicated. We can build up more complex computations – e.g. adding two numbers by repeatedly adding 1 to the first and subtracting 1 from the second until the second is 0.

3 Lambda calculus

Lambda (λ) calculus is the theoretical foundation for functional programming, and its influence can be seen in aspects of practical functional languages.

"Calculus" sounds very mathematical, but lambda calculus is quite simple - does not have any complicated formulae or operations – but it is very abstract which can be difficult to appreciate. All it ever does is take a collection of letters or symbols (an *expression*) and does some substitutions.

3.1 Example

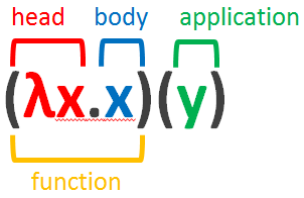
(λx.x)(y)

This contains:

- variables – **x** and **y** in this example
- parentheses **( )** - indicate that some part of an expression belongs together
- the Greek letter **λ** – indicates that what follows is a *function*

The *names* of the variables are not important; the only thing that matters is that when two variables have the same name, they are the same. **(λa.a)(b)** is exactly the same expression. They are not like variables as you understand them in programming, they do not hold any information.

3.2 Lambda expression example



The **Function** consists of a **head** (λ and an expression containing *bound variable*(s) – just x in this example), a *dot* and a **body -** a bound variable is somewhat like a function parameter in programming.

The **Body** is an expression which can consist of variables, parentheses and other expressions.

We can have an expression after the function, called an **Application**, this is a bit like the parameter value(s) in programming (but remember there are no actual values in lambda calculus).

The *whole line is an expression* – an expression can be built up from other expressions, including functions.

3.3 Resolving a lambda function

What does a function calculate? Nothing, really. It is just an expression, the only thing we can do with it is to resolve it. Resolution works by taking the variable mentioned in the *head*, and replacing all of its occurrences within the *body* with the *application*.

(λx.x)(y)

= y

We replace x in the body with y; the function is now resolved so remove **λx**.

This function, applied to **y** gives **y** – in other words it gives you back what you started with. This is a special function called the *identity function*.

3.4 Lambda calculus computation - example

Let's look at the same example as we did for the Turing machine – adding 1 to a number. The first problem is that we don't have numbers in lambda calculus, just expressions with functions and (meaningless) variables – we need to represent numbers as functions – seems strange, but why not? There are many ways of representing numbers apart from the decimal system we are used to, e.g. Roman numerals.

Let's define zero as:

0 = λsz.z

(remember names don't mean anything, so this is the same as **λ ab.b** or **λ fg.g**).

And the next few natural numbers as (you'll see why shortly):

1 = λsz.s(z)
2 = λsz.s(s(z))
3 = λsz.s(s(s(z)))
...

So this (rather weird number) system starts with a "0" function and nests an expression **s(..)** round the body once more for each successive number.

Our "add 1" computation needs to get us from one number to the next; the only mechanism we have is applying functions, so we need to define an "add 1" function that if you apply to our "0" expression and resolve will give our "1" expression, and so on. A function that works is:

λ abc.b(abc)

Applying this to our zero expression

(λ abc.b(abc)) (λ sz.z)

and resolving, gives our "1" expression **λ sz.s(z)**.

Similarly, applying the same function to "1" gives "2", and so on. We can add a number to another by applying the "add 1" function the required number of times.

Let's look at this process in some more detail:

(λabc.b(abc)) ("0")

= (λ**a**bc.b(**a**bc)) **(λ sz.z)**

First replace the first bound variable **a** with expression **λ sz.z** (the "0" function)

= λbc.b((λ **s**z.z) **b**c)

This time replace the first bound variable with **b,** the first expression in the application. However, the bound variable **s** is not in the body, so there is nothing to replace with **b**, so b "disappears". Note that parentheses matter, it makes us resolve the function inside them before continuing.

= λbc.b((λ **z**.**z**) **c**)

This time replace bound variable **z** with **c**; the function is now fully resolved so remove **λ**.

= λbc.b(c)

We're done, but can choose to change variable names to emphasise that the result is the same as "1" function defined earlier:

= λsz.s(z) = "1"

3.5 Lambda calculus summary

So where has this brief look at Lambda calculus got us? We have a weird way of representing numbers (known as *Church numerals*) and a rather masochistic way of adding them (and there are similar functions for other operations).

The point is to show that, in principle, *any computation* that can be carried out in a series of steps can also be carried out by defining functions and evaluating expressions that apply them. This is the foundation of functional programming.

There are variants of the lambda calculus

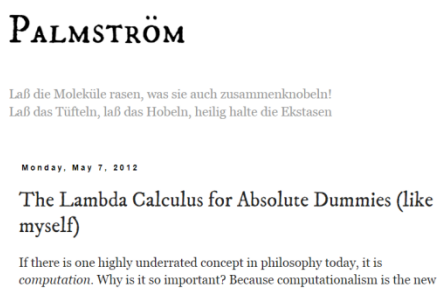
- Pure/untyped – this is what we have looked at here
- Applied – includes constant values and predefined functions
- Typed – allows variable types to be specified

You can do a lot more with lambda calculus, but we'll leave it there and get back to some practical programming (and you won't be asked to do any lambda calculus in the exam)!

3.6 Lambda calculus Further Exploration

Lambda Calculus - Computerphile

https://www.youtube.com/watch?v=eis11j_iGMs



PALMSTRÖM

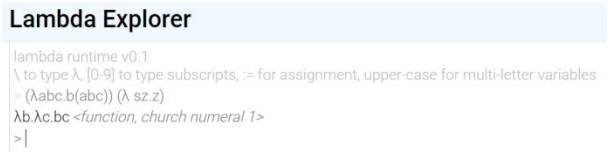
Laß die Moleküle rasen, was sie auch zusammenknobeln!
Laß das Tüfteln, laß das Hobeln, heilig halte die Ekstasen

Monday, May 7, 2012

The Lambda Calculus for Absolute Dummies (like myself)

If there is one highly underrated concept in philosophy today, it is *computation*. Why is it so important? Because computationalism is the new

http://palmstroem.blogspot.co.uk/2012/05/lambda-calculus-for-absolute-dummies.html



http://www.lambda-explorer.net/

4 Lambda expressions

The influence of lambda calculus is obvious in many modern programming languages in the form of *lambda expressions* (fairly recently "retro-fitted" to Java with Java 8's new functional capabilities).

A Lambda expression is an expression that defines a function, for example, in Scala:

```
x => x % 2 == 0
```

This is analogous to a function in lambda calculus:

- **x** is the *bound variable* and => is equivalent to the dot
- **x % 2 == 0** is the *body* – a Boolean expression that evaluates to *true* if x divides exactly by 2

This is often applied by passing the expression as a parameter to another function:

```
scala> val numbers = List(1,2,3,4,5,6)
numbers: List[Int] = List(1, 2, 3, 4, 5, 6)
scala> numbers.filter(x => x % 2 == 0)
res0: List[Int] = List(2, 4, 6)
```

The *List* collection type has a *filter* method that applies the function defined by the lambda to each element of the list in turn; includes in the result only those elements for which expression evaluates to *true*.

5 Additional reading

http://palmstroem.blogspot.co.uk/2012/05/lambda-calculus-for-absolute-dummies.html

http://www.oracle.com/technetwork/articles/java/ma14-java-se-8-streams-2177646.html