# Formal Foundations

Winter 2016-2017

# Table of Contents

# Purpose

- To introduce the Lambda Calculus
- CMPT340 will need only a light introduction
- The topic and the theory are quite deep (there is a lot that we won't study because we don't need it for CMPT 340)

# Objectives

- Using the lambda calculus, we will (hopefully) be able to appreciate some of Haskell's abilities more clearly:
  - Currying
  - Lazy evaluation
  - Type signatures

# Background

- The idea of "computation" can be formalized in (at least) 3 distinct ways:
  1. Turing machine
  2. First-order predicate calculus
  3. Lambda calculus
- These are equivalent in terms of computability: anything that is computable can be computed using any of these approaches.
- Languages in CMPT340 use two of these:
  1. Haskell is based on the Lambda calculus
  2. Prolog is based on first-oder predicate calculus

# Functions without names

- The basic idea behind Lambda Calculus is to express functions without requiring the function be given a name
- In normal mathematics, we define functions like:

$$f(x) = x^2 + 3$$

- In Lambda calculus, we write the same function as follows:

$$f = \lambda x.(x^2 + 3)$$

# Understanding the notation

$$\lambda x.(x^2 + 3)$$

"The function that maps $x$ to $x^2 + 3$."

- The symbol $\lambda$ is not a name, but an abstraction operator.
- The notation does not give the function a name.
- The function is described only by its input and output behaviour.

# Understanding the notation

$$\lambda x.(x^2 + 3)$$

"The function that maps $x$ to $x^2 + 3$."

- It denotes the fact that the expression is a function.
- The symbol $x$ follows the $\lambda$ and it is the name of the function's one parameter.
- The symbol . separates the parameter from the body of the function.
- The parentheses are used to indicate the extent of the body.

# Simple examples

- A function that adds 1 to its input:

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

- A function that multiplies its input by 7:

$$\lambda u.(u \times 7)$$

- A function that squares its input then adds 3:

$$\lambda z.(z^2 + 3)$$

# Simple assumption

- Note: we are assuming, for convenience, the use of numbers and simple arithmetic operators like $+$ and $\times$.
- These are convenient for our use, but not a necessary part of the theory.
- In other words, we could do everything with just functions, and no numerals or familiar operators. We get something with a lot more notation, but we lose nothing without it, in terms of what we can compute.
- This is an advanced topic, and we will say a little more about it later.

# Naming functions

- A $\lambda$-expr defines a function without giving it a name.
- We are allowed to name functions, but a name is not an essential part of the function.
- We use the equation operator $=$ to give names to values. E.g.:

$$\begin{aligned} f &= \lambda x.(x^2 + 3) \\ \mathit{three} &= 3 \end{aligned}$$

- The name on the left, the value on the right.
- A function is just a value!
- Equality is not assignment! You cannot change a binding!

# Examples: definitions

## Haskell

```
f x       = x*x + 3
incr x    = 1 + x
seven x   = 7
```

## Lambda Calculus

$$\begin{aligned} \mathrm{f} &= \lambda x.(x^2 + 3) \\ \mathrm{incr} &= \lambda x.(1 + x) \\ \mathrm{seven} &= \lambda x.(7) \end{aligned}$$

# Simple evaluation rules

- There are 2 basic rules that we use on $\lambda$-expressions:
  - Parameter renaming
  - Function application
- Because we are using basic arithmetic and numbers, we also include expression simplification. For example, $3 + 4$ can be simplified or evaluated, resulting in 7. This keeps things simple for our needs, but is not a necessary part of Lambda Calculus. We could do everything with just $\lambda$-expressions.

# Rule 1: Parameter renaming

- A parameter can be renamed:

$$\lambda x.(x^2 + 3) \rightsquigarrow \lambda y.(y^2 + 3)$$

- You have to avoid using a name that already appears in a $\lambda$-expr.
- E.g., Given the function $\lambda x.(x + y)$ , we cannot replace $x$ with $y$. because $y$ is already present in the function.
- This basic "rule" has a fancy name: in the lambda calculus it is called $\alpha$-conversion.

# Rule 2: Function application

- A $\lambda$-expr descibes a function, but we have to know how to use it ("function application")
- First, we will review the "normal mathematics" way to use a function.
- Recall the definition ("normal"): $f(x) = x^2 + 3$
- We can use the function by giving it an argument. We call this an application. In normal notation, we write $f(4)$. We say "Apply the function $f$ to the value (argument) 4."
- Obviously, $f(4) = 19$

# More closely

- We can simplify the expression $f(4)$ by the following (very tediously expressed) steps:

$$\begin{aligned} f(4) &= 4^2 + 3 \\ &= 16 + 3 \\ &= 19 \end{aligned}$$

  The first step is the interesting one: We used the definition of $f$ by:

  1. Taking the parameter $x$ and pairing it with the argument 4.
  2. rewriting the body of the function, substituting $x$ for 4.

- After that, it was just "normal arithmetic."

# Function application in Lambda calculus

- To "call a function" on an argument, we write the function and the argument side by side. E.g.,

$$(\lambda x.(x^2 + 3))\ \ 4$$

- This is called an "application" of the function. The function is applied to the argument 4. It is a legal expression because the thing on the left is a function, and the thing on the right is its argument.
- The extra set of parentheses indicates the extent of the function. You may use parentheses, brackets, angle brackets, whatever you like to delimit the function visually.

# Evaluating function applications

- When a function is side-by-side with an argument, you can simplify that expression with the following steps:
  1. Pair the left-most parameter with the left-most argument.
  2. Drop the $\lambda$ and the parameter name, and the dot (.)
  3. Write the body of the function replacing the parameter name with the argument it is paired with. E.g.

$$(\lambda x.(x^2 + 3))\ 4 \rightsquigarrow (4^2 + 3)$$

- This is a simplification rule (or a reduction) called "application". It has a fancy name in the Lambda Calculus, where it is known as a $\beta$-reduction.

# More examples of application

- A very simple one:

$$(\lambda x.(x + 1))\ 5 \quad \rightsquigarrow \quad (5 + 1)$$

- Another simple one:

$$(\lambda y.(7 \times y))\ 4 \quad \rightsquigarrow \quad (7 \times 4)$$

- Note: We are only showing how to perform the single application step. The resulting expression may or may not be in "simplest form."

# A slightly less simple example

- There are two ways to simplify the following expression:

$$(\lambda z.(z^2 + 3))\ (3 + 4)$$

- The first way is familiar; the second way is a little different.
- The answer does not depend on which one you use.
- However, it may be more efficient one way or the other.

# Strict: Arguments first

- The first way is the most familiar to you, since it is the way such expressions are calculated in most programming languages:

$$\begin{aligned}(\lambda z.(z^2+3))\,(3+4) &= (\lambda z.(z^2+3))\,7 \\ &\rightsquigarrow (7^2+3)\end{aligned}$$

- In this example, the expression $(3+4)$ was simplified first, then we used the application rule.
- When the argument expression is always evaluated first, it is informally called "strict evaluation" or "eager evaluation." Formally, this is referred to as the "applicative order" of evaluation.

# Lazy: Application first

- The second way is to use the application rule first, as follows:

$$(\lambda z.(z^2 + 3))\,(3 + 4) \quad \rightsquigarrow \quad (3 + 4)^2 + 3$$

  The expressions that result have the same value; only the order is different.

- When the application rule is always applied first, it is informally called "lazy evaluation" or "non-strict." Formally, this is referred to as the "normal order" of evaluation.

# The lazy evaluation difference

Consider the application:

$$(\lambda x.(7))\ (3 \times 4)$$

## Strict

$$\begin{aligned}(\lambda x.(7))\ (3 \times 4) &\rightsquigarrow (\lambda x.(7))\ 12 \\ &\rightsquigarrow 7\end{aligned}$$

Evaluated $(3 \times 4)$ for no reason.

## Lazy

$$(\lambda x.(7))\ (3 \times 4) \rightsquigarrow 7$$

Ignored $(3 \times 4)$ because $x$ was not needed.

# Summarizing lazy vs. strict

- Suppose we have an expression $(e_1\ \ e_2)$, where $e_1$ is a function (either a name of a function, or a $\lambda$-expression).
- We know that $e_1$ must be a function, because it is the leftmost sub-expression. $e_2$ is its argument.

## Eager/strict evaluation:

- $e_2$ is evaluated first.
- The application rule substitutes the value of $e_2$ into $e_1$.

## Lazy evaluation:

- The application rule is used first.
- The expression $e_2$ is substituted into $e_1$.

# The lazy evaluation advantage

- Haskell uses lazy evaluation by default.
- Lazy evaluation changes the practice of programming
- E.g., Use an infinite list to represent any amount of data.
- E.g., Use non-terminating recursion to model computations where the number of steps is unknown.
- Productive and useful abstractions to simplify complex algorithms.
- Caveat: Applications will limit actual resources used. But the burden of control is moved to a different (more flexible) location.

# A Simple example

- Here's an interesting $\lambda$-expression:

$$\lambda x.(x\ 3)$$

  The body of the expression is the application of $x$ to the argument 3.
- In other words, the function $\lambda x.(x\ 3)$ must be given a function as an argument. Otherwise, the body won't be a legal expression.

# How to evaluate with functions as arguments

- Let's see how to use this kind of function. The rule of application does not change; it's just the kind of values that have changed.

$$(\lambda x.(x\ 3))\ \ (\lambda z.(z+1)) \quad \rightsquigarrow \quad (\lambda z.(z+1))\ 3$$

- The two expressions side by side indicate a function application. The only difference from before is that both expressions are functions.
- We paired $x$ with the argument $\lambda z.(z+1)$, and rewrote the body of the function on the left.

# Why is this interesting?

$$(\lambda x.(x\ 3))\ \ (\lambda z.(z+1)) \quad \rightsquigarrow \quad (\lambda z.(z+1))\ 3$$

- There are many kinds of functions that can take other functions as arguments.
- It provides a much different way of building sophisticated programs.
- It is the essence of functional programming.

# Not all applications are legal in Haskell!

Legal:

$$\begin{aligned}(\lambda x.(x\ 3))\ (\lambda z.(z+1)) &\leadsto (\lambda z.(z+1))\ 3\\ &\leadsto (3+1)\\ &\leadsto 4\end{aligned}$$

Not legal:

$$\begin{aligned}(\lambda z.(z+1))\ (\lambda x.(x\ 3)) &\leadsto (\lambda x.(x\ 3)) + 1\\ &\leadsto \textit{Type error in Haskell}!\end{aligned}$$

# Type inference in Haskell

- Every expression has a type.
- Every sub-component of an expression has a type.
  - Assume that $1, 3$ are integers.
  - $3 + 1$ is also an integer, provided that $(+)$ maps 2 integers to an integer.
  - $z + 1$ is an integer, provided that $z$ is an integer.
  - $\lambda z.(z + 1)$ is a function that maps an integer to another integer.
  - $\lambda x.(x\ \ 3)$ is a function that maps a function to something, provided that $x$ is a function that maps integers to something.
- Haskell provides a system to express types.

# Type signatures in Haskell

- Haskell provides a system to express types.
  - Assume $1, 3 :: \mathit{Integer}$
  - $3 + 1 :: \mathit{Integer}$ provided that $(+)$ maps 2 integers to an integer.
  - $z + 1 :: \mathit{Integer}$ is an integer, provided that $z :: \mathit{Integer}$
  - $\lambda z.(z + 1) :: \mathit{Integer} \to \mathit{Integer}$
  - $\lambda x.(x\ \ 3) :: (\mathit{Integer} \to t) \to t$
- The last example uses a type variable – more later!

# Functions with more than 1 argument

- It is easy to define functions of 2 or more parameters using "normal" notation, e.g.

$$f(x, y) = x^2 + y^2$$

- In the Lambda calculus, we write this kind of function using a sequence of 2 $\lambda$s:

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

# Functions with more than 1 argument

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

- This technique (of using more than one $\lambda$) is called "currying"
- If we read this carefully we see that it is a function of 1 argument, $x$.
- The body of this function is itself a function: $\lambda y.(x^2 + y^2)$
- To reduce the clutter in the notation, we can write it this way:

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

# Evaluations involving curried functions I

- The rules we saw earlier work fine on these funny functions. using the application rule.
- The only difference will be the kind of result you get.
- E.g.,

$$(\lambda x.\lambda y.(x^2 + y^2))\ 9 \quad \rightsquigarrow \quad (\lambda y.(9^2 + y^2))$$

- Note that we used the application rule, pairing $x = 9$, and rewriting the body, which itself is a $\lambda$-expression.
- This is as far as our rules allow us to go
- Thus, the result of giving an argument to this function is another function. It's related to the original function (it's a special version, with $x = 9$)

# Another example

$$
\begin{aligned}
&(\lambda x.\lambda y.\lambda z.(x^2 + y^2 - 3 \times z))\ 7\ 5 \\
&\quad \rightsquigarrow \quad (\lambda y.\lambda z.(7^2 + y^2 - 3 \times z))\ 5 \\
&\quad \rightsquigarrow \quad (\lambda z.(7^2 + 5^2 - 3 \times z))
\end{aligned}
$$

Here the application rule was used twice:

1. First time, pairing $x = 7$, and rewriting the body of the first $\lambda$-expression
2. Second time, pairing $y = 5$, and rewriting the body of the $\lambda$-expression for $y$

We are left with a $\lambda$-expression, which means the final value is a function.

We always use the application rule with the left most parameter and the left most argument.

# Currying

- To summarize: currying is the technique of writing a function of 2 or more parameters as as sequence of functions of one parameter each.
- The application rule is applied to curried functions does not change: pair the left-most parameter with the left-most argument, and rewrite the body of the expression with the appropriate substitution.
- When we give only some of the possible arguments to a curried function, the result is always a curried function of the remaining parameters. The given arguments are paired left to right with the parameters (from left to right).

# Examples: definitions

Haskell:

```
incr x = 1 + x

blam x y = (x + y) * x

com u v w = u (v w)
```

See notes (next slide)!

Lambda Calculus

$$\begin{aligned} \text{incr} &= \lambda x.(1 + x) \\ \text{blam} &= \lambda x.\lambda y.((x + y) * x) \\ \text{com} &= \lambda u.\lambda v.\lambda w.(u\ (v\ w)) \end{aligned}$$

# Notes on example 1

- Don't be fooled by the simplicity. All of the following expressions are also correct:

$$\begin{aligned} \text{incr} &= \lambda y.(1 + y) \\ \text{incr} &= \lambda a.(1 + a) \\ \text{incr} &= \lambda xyzzy.(1 + xyzzy) \end{aligned}$$

- This is the renaming rule at work. These functions are all the identical, except for variable renaming.

# Notes on example 2

- Function application associates left. In other words:

$$x\ y\ z = (x\ y)\ z$$

  - The left most expression $x$ is a function and the remaining expressions are the arguments.
  - It also means that $(x\ y)$ is a function, whose argument is $z$.

# Notes on example 3

- On the RHS of 'com',
  - The name $u$ is "leftmost" in the expression $u\ (v\ w)$.
  - The name $v$ is "leftmost" of the expression $v\ w$.
- Thus, by visual "parsing" of the RHS, we know $u$ and $v$ must be functions.
- Of course, $u$ and $v$ are parameters to 'com', so we infer that 'com' takes 3 parameters (in turn), of which the first 2 are functions.

# Examples: simple evaluations

$$\begin{aligned} \text{incr}\ (2+1) &= (\lambda x.(1+x))\ \ (2+1) \\ &\rightsquigarrow 1+(2+1) \\ &\rightsquigarrow 4 \end{aligned}$$

- Replace the name 'incr' with its value
- Use the application rule before evaluating the argument
- Simplify using rules of arithmetic.
- Note: In our context, arithmetic is "primitive," and is always evaluated. We'll see that Haskell can sometimes avoid arithmetic.

# Examples: simple evaluations

$$\begin{aligned} \text{blam}\ (2+1) &= (\lambda x.\lambda y.(x+y)*x))\ \ (2+1) \\ &\rightsquigarrow \lambda y.(((2+1)+y)*(2+1)) \end{aligned}$$

- Repace the name 'blam' with its value.
- Use the application rule pairing $x$ with $(2+1)$.

# Examples: simple evaluations

$$
\begin{array}{rcl}
\text{incr } (\text{incr } 0) & = & (\lambda x.(1+x))\ \ (\text{incr } 0) \\
& \rightsquigarrow & 1 + (\text{incr } 0) \\
& = & 1 + ((\lambda x.(1+x))\ 0) \\
& \rightsquigarrow & 1 + (1+0) \\
& \rightsquigarrow & 2
\end{array}
$$

- The outer application was handled first.
- The arithmetic on line 2 can't happen until the second application is completed. Thus this application is "forced."
- Simplify last.

# Examples: more evaluations

$$
\begin{aligned}
\text{com incr (blam 3)} &= (\lambda u.\lambda v.\lambda w.(u\ (v\ w)))\ \text{incr (blam 3)} \\
&\rightsquigarrow (\lambda v.\lambda w.(\text{incr}\ (v\ w)))\ (\text{blam 3}) \\
&\rightsquigarrow \lambda w.(\text{incr}\ ((\text{blam 3})\ w))
\end{aligned}
$$

It's possible to do more steps, but Haskell would stop here, because the expression is a function.

# Notes on the examples

- The outer-most application rule is used first.
- In the previous example, the arguments were "consumed" from left to right.
  1. Pairing $u$ with 'incr' first
  2. Pairing $v$ with $(\mathrm{blam}\ 3)$ next
- I used the application rule from outside to inside.
- I stopped when there were no more "outer" applications to do.

# Haskell's evalautions

- Haskell will force outer evaluations involving arithmetic, e.g., $(3 + 4)$
- Haskell will force evaluation if the value is to be displayed (e.g., in the interpreter).
- There are other ways to force evaluation, but they are not covered in this lecture.
- If an evaluation is not forced, it will not happen.

# Exercises

1. You should be able to derive all of these:

$$\begin{aligned} \text{com incr incr} &= \lambda x.(1 + (1 + x)) \\ \text{com blam incr} &= \lambda x.\lambda v.(((1 + x) + v) * (1 + x)) \end{aligned}$$

2. You might be able to show that the following is not a valid expression. It's tricky.

$$\text{com blam blam}$$

# Lambdas in Haskell

| Lambda calculus | Haskell |
| --- | --- |
| $(\lambda x.(1 + x))$ | `(\ x -> (1 + x))` |
| $\lambda x.\lambda y.((x + y) * x)$ | `(\ x -> (\ y -> ( x + y) * x) )` |

Haskell uses `\` for $\lambda$ and `->` for . The parentheses indicate the extent of the expression.

# Type signatures for curried functions

Assuming integer numbers – this is not quite general enough, but more later!

```
incr :: Integer -> Integer
incr x = 1 + x

blam :: Integer -> Integer -> Integer
blam x y = (x + y) * x

fred :: Integer -> Integer -> Integer -> Integer
fred x y z = x*x + y*y - 3*z
```

There is a type for every argument, a type for the result, and a `->` between them all!

# Example: Gravitational Force

```
gravForce :: Float -> Float -> Float -> Float
gravForce m1 r m2
  | r == 0    = 0
  | otherwise = gravConstant*m1*m2/square r

weightOnEarth :: Float -> Float
weightOnEarth = gravForce earthMass earthRadius

sunGravity :: Float -> Float -> Float
sunGravity = gravForce sunMass
```

# Summary I

- The Lambda calculus uses the abstraction operator $\lambda$ to describe a function in terms of the input parameters, and the output expression (the body of the function).
- Using this notation, we have seen how to evaluate expressions using the application rule. (we also used basic arithmetic, for convenience; it is not a necessary part of the theory)
- Lazy evaluation is implemented when the application rule is always used as soon as possible, delaying any other simplification until the application rule cannot be used at all.
- Currying is the technique of writing a function of 2 or more parameters as as sequence of functions of one parameter each.
    - In the Lambda Calculus, this is represented by a sequence of nested $\lambda$-expressions.
    - In Haskell, we don't use the $\lambda$ explicitly, but the type signature of the function shows the sequence of functions.

# Pointers to other ideas I

- These notes give just enough background to the Lambda Calculus to introduce lazy evaluation and currying.
- The theory of Lambda Calculus is much deeper and richer.
- The Lambda Calculus needs $\alpha$-conversions, and $\beta$-reductions (there is a 3rd rule, called an $\eta$-reduction, too). Everything else is a matter of convenience.
- We assumed basic arithmetic, and numbers. Numbers can be represented by functions; we can define arithmetic on these functions too. Similarly, we can define functions to represent boolean values, or anything else. It's a lot more work, but it shows that the Lambda calculus does not depend on anything other than functions for its power.

# Pointers to other ideas II

- We looked only at very simple functions. There are much more sophisticated functions called combinators that really give Lambda calculus its power for computation. We didn't study these, because they are slightly off-topic.

# Working with Lambda Calculus outside of Haskell

- As a model of computation, we want to push evaluations farther than Haskell might.

# Boolean logic in Lambda Calculus

Given the following definitions:

$$\begin{aligned} \text{true} &= \lambda x.\lambda y.(x) && (1)\\ \text{false} &= \lambda x.\lambda y.(y) && (2)\\ \text{not} &= \lambda v.\lambda w.\lambda x.(v\ x\ w) && (3)\\ \text{or} &= \lambda v.\lambda w.(v\ v\ w) && (4)\\ \text{and} &= \lambda v.\lambda w.(v\ w\ v) && (5) \end{aligned}$$

The functions for 'and' and 'or' are expressed as prefix functions, rather than infix.

# Boolean logic in Lambda Calculus

Evaluate the following:

$$\text{not true} \tag{6}$$

$$\text{and false true} \tag{7}$$

You should be able to derive the expression 'false' for both using nothing but the application rule.
All of Boolean logic can be expressed as functions!

$$
\begin{aligned}
\text{not true} &= (\lambda v.\lambda w.\lambda x.(v\ x\ w))\ \ \text{true} \\
&= \lambda w.\lambda x.(\text{true}\ x\ w) \\
&= \lambda w.\lambda x.(\lambda a.\lambda b.(a)\ x\ w) \\
&= \lambda w.\lambda x.((\lambda b.(x))\ w) \\
&= \lambda w.\lambda x.(x) \\
&= \lambda x.\lambda y.(y) \\
&= \text{false}
\end{aligned}
$$

Note: Haskell would stop at line 2, if the LHS were forced. We continued on to show the equivalence. We did "inner applications" to complete the proof.

# Boolean logic in Lambda Calculus

- This representation shows that Lambda Calculus can represent "data" using functions only.
- There is no argument that doing so is efficient – it's not very efficient. But it is possible!
- Very important: Don't try to implement this directly in Haskell.
  - You can define every function I've shown using Haskell syntax
  - BUT: Equality between functions is undefined, so Haskell can never tell if a function is the one for 'true' or for 'false'
  - So the functions are correct, but not practical for computation in Haskell
  - Cool idea: Represent lambda calculus using datatypes instead of functions...