159.341 Programming Languages, Algorithms & Concurrency

Lambda Calculus
(Part 2)

Daniel Playne
d.p.playne@massey.ac.nz

λ Calculus Reminder

So far we have covered the basics of λ-calculus:

- Abstraction and the basis of λ calculus
- Introduction to λ calculus syntax
- Normal order β reduction
- Syntax for function definitions
- α conversion for removing name-clashes
- η reduction for simplifying expressions

Developing a Functional Language

Lambda calculus is the ***model of computation*** that underlies functional languages.

To see how this can actually be developed into a useful functional language, we will look at the following features.

- Conditionals
- Integers
- Arithmetic operators

Conditionals

We will see how additional layers can be developed to create some higher-level (and easier to read) notation.

We will construct conditional expressions with truth values `true` and `false` and develop boolean operations `and`, `not` and `or`.

First we need to look at some functions that will help us along the way `select_first` and `select_second`.

Conditionals

Consider the function:

def `select_first = λfirst.λsecond.first`

When this is applied to two expressions, `select_first` will return the first.

For example:

```
((select_first exp1) exp2) ==
((λfirst.λsecond.first exp1) exp2) =>
(λsecond.exp1 exp2) =>
exp1
```

Conditionals

Consider the function:

def `select_second = λfirst.λsecond.second`

When this is applied to two expressions, `select_second` will return the second.

For example:

```
((select_second exp1) exp2) ==
((λfirst.λsecond.second exp1) exp2) =>
(λsecond.second exp2) =>
exp2
```

Conditionals

We can now use these functions to build up conditional expressions similar to the ternary expressions used in C:

```
condition ? expression1 : expression2
```

If the condition evaluates to true, then the first expression is selected and if the condition is false then the second expression is selected.

Conditionals

A similar expression can be constructed in λ calculus.

def `cond` = λ`e1.`λ`e2.`λ`c.((c e1) e2)`

Consider this function `cond` applied to two arbitrary expressions `exp1` and `exp2`:

```
((cond exp1) exp2) ==
((λe1.λe2.λc.((c e1) e2) exp1) exp2) =>
(λe2.λc.((c exp1) e2) exp2) =>
λc.((c exp1) exp2)
```

Conditionals

If this function is applied to `select_first`:

```
λc.((c exp1) exp2) select_first)  =>
((select_first exp1) exp2) => ... =>
exp1
```

or `select_second`:

```
(λc.((c exp1) exp2) select_second) =>
((select_second exp1) exp2) => ... =>
exp2
```

Conditionals

We can use the `cond` function to implement conditionals and the following definitions:

```
def cond = λe1.λe2.λc.((c e1) e2)
def true = select_first = λfirst.λsecond.first
def false = select_second = λfirst.λsecond.second
```

(Probably not the representation of true/false that you expected)

Conditionals

The `NOT` operator is a unary operator of the form `NOT operand`. It should also return the opposite of the boolean argument given to it - `not true = false` and `not false = true`.

Using our previous definitions, this function can be described by the following definition:

def `not =` λ`x.((cond false) true) x)`

Conditionals

This can be simplified:

```
(((cond false) true) x) ==
(((λe1.λe2.λc.((c e1) e2) false) true) x) =>
((λe2.λc.((c false) e2) true) x) =>
(λc.((c false) true) x) =>
((x false) true)
```

Thus we use:

def `not = λx.((x false) true)`

Conditionals

For example `(not true)`:

```
(not true) ==
(λx.((x false) true) true) =>
((true false) true) ==
((λfirst.λsecond.first false) true) =>
(λsecond.false true) =>
false
```

Conditionals

Work through the similar evaluation of `(not false)`.

Conditionals

Work through the similar evaluation of `(not false)`.

```
(not false) ==
(λx.((x false) true) false) =>
((false false) true) ==
((λfirst.λsecond.second false) true) =>
(λsecond.second true) =>
true
```

Conditionals

The `or` operator can also be constructed by using selectors. This function should result in `true` if either of the arguments are `true`.

If the first operand is `true` then select `true`, otherwise select the second operand.

def `or` = λ`x.`λ`y.(((cond true) y) x)`

Like the operator `not`, this can be simplified.

Conditionals

```
(((cond true) y) x) ==
(((λe1.λe2.λc.((c e1) e2) true) y) x) =>
((λe2.λc.((c true) e2) y) x) =>
(λc.((c true) y) x) =>
((x true) y)
```

Thus we have:

def `or = λx.λy.((x true) y)`

Conditionals

Evaluation of `((or true) false)`:

```
((or true) false) ==
((λx.λy.((x true) y) true) false)
(λy.((true true) y) false)
((true true) false)
((λfirst.λsecond.first true) false)
(λsecond.true false)
true
```

Conditionals

Try to write (and simplify) the `and` operator:

Conditionals

Try to write (and simplify) the and operator:

def and = λx.λy.(((cond y) false) x)

```
(((cond y) false) x) ==
(((λe1.λe2.λc.((c e1) e2) y) false) x) =>
((λe2.λc.((c y) e2) false) x) =>
(λc.((c y) false) x) =>
((x y) false)
```

Thus:

def and = λx.λy.((x y) false)

Conditionals

Evaluation of `((and true) false)`:

```
((and true) false) ==
((λx.λy.((x y) false) true) false) =>
(λy.((true y) false) false) =>
((true false) false) ==
((λfirst.λsecond.first false) false) =>
(λsecond.false false) =>
false
```

Developing a Functional Language

We have looked at the simple rules of λ calculus and how we can define conditionals and logical operators.

There are a number of other features necessary to fully develop a functional language from λ calculus functions. Most important are integers, arithmetic operations and recursion.

There is also a significant amount of syntax simplification used for functional languages, but it should be unambiguous and able to be directly translated back into pure λ calculus.

Developing a Functional Language

One obvious requirement for a functional language is to support numbers and arithmetic.

Natural numbers (non-negative integers) can be defined a ***successors*** of ***zero***. 1 is the successor of 0, 2 is the successor of, the successor of 0, three is the successor of, the successor of, the successor of 0 (etc).

```
def one = (succ zero)
def two = (succ one)
def three = (succ two)
```

Developing a Functional Language

This allows natural numbers to be defined as that many successors of zero. Thus does require a function `zero` and `succ`. One way of representing these is to use the following definitions:

```
def zero = identity
def succ = λn.λs.((s false) n)
```

This definition builds a pair function with `false` first and the number second.

Developing a Functional Language

For example:

```
one ==
(succ zero) ==
(λn.λs.((s false) n) zero) =>
λs.((s false) zero)
```

```
two ==
(succ one) ==
(λn.λs.((s false) n) one) =>
λs.((s false) one) ==
λs.((s false) λs.((s false) zero))
```

Developing a Functional Language

Similarly a predecessor function `pred` can be constructed such that:

```
(pred one)   => ... => zero
(pred two)   => ... => one
(pred three) => ... => two
```

For our number representation, this can be constructed by removing a layer of nesting from a given number.

```
λs.((s false) number)
```

Developing a Functional Language

A first attempt at defining the predecessor function (we will call this `pred1`) could be:

def `pred1 = λn.(n select_second)`

In the general application this would give:

```
(pred1 λs.((s false) number)) ==
(λn.(n select_second) λs.((s false) number)) =>
(λs.((s false) number) select_second) =>
((select_second false) number) == ... =>
number
```

Developing a Functional Language

What is the problem with this function definition?

Developing a Functional Language

What is the problem with this function definition?

What about `zero`?

```
(pred1 zero) ==
(λn.(n select_second) zero) =>
(zero select_second) ==
(λx.x select_second) =>
select_second ==
false
```

Which does not represent a number.

Developing a Functional Language

Instead we could define that the predecessor of `zero` is `zero` and use the definition:

def `pred` = λ`n.(((cond zero) (pred1 n)) (iszero n))`

Which can be simplified to:

def `pred = (((iszero n) zero) (pred1 n))`

def `iszero` = λ`n.(n true)`

Summary

- Conditionals - `true` and `false`
- Logical Operators - `not`, `and`, `or`
- Defining Integers - `succ`, `pred`

Chapters 2 & 3
An Introduction to Functional Programming Through Lambda Calculus
Greg Michaelson