MASSEY UNIVERSITY
TE KUNENGA KI PŪREHUROA
UNIVERSITY OF NEW ZEALAND

159.341 Programming Languages, Algorithms & Concurrency

Lambda Calculus (Part 3)

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

Reminder

Conditionals

```
def true  = λfirst.λsecond.first
def false = λfirst.λsecond.second
def cond  = λe1.λe2.λc.((c e1) e2)
def not   = λx.((x false) true)
def or    = λx.λy.((x true) y)
def and   = λx.λy.((x y) false)
```

Reminder

In the previous lecture we defined numbers through the use of successor and predecessor functions.

```
def zero    = identity
def iszero  = λn.(n select_first)
def succ    = λn.λs.((s false) n)
def pred1   = λn.(n select_second)
def pred    = λn.(((cond zero) (pred1 n)) (iszero n))
```

Developing a Functional Language

Before defining arithmetic functions we will define some ways for simplifying the λ expressions. Even simple expressions involve a lot of brackets which are both tedious and introduce a lot of errors due to mismatched brackets (for humans).

The application of a function `<function>` to N arguments will be simplified from:

```
(...((function arg1) arg2) ...  argN)
```

to:

```
function arg1 arg2 ...  argN
```

Developing a Functional Language

Functions will be applied to the nearest argument to the right first. If an argument represents a function application then it must be surrounded in brackets.

The other way we can simplify the syntax is to drop the λ and move the bound variable to the other side of the = sign.

def `name` = λ`arg.expression`

becomes:

def `name arg = expression`

Developing a Functional Language

This syntax allows us to re-write our functions in the following form:

```
def identity x            = x
def self_apply s          = s s
def apply func arg        = func arg
def cond e1 e2 c          = c e1 e2
def true first second     = first
def false first second    = second
def not x                 = x false true
def and x y               = x y false
def or x y                = x true y
```

Developing a Functional Language

We could also introduce some new syntax for conditional statements - instead of writing:

```
cond c exp1 exp2
```

we can write:

```
if c then exp1 else exp2
```

Arithmetic Functions

Using our simplified syntax we could define arithmetic functions using `succ` and `pred`.

```
def add x y = if (iszero y) then x else (add (succ x) (pred y))
```

Functions like this actually require some extra work to deal with functions that contain themselves in their own definition (recursive functions).

Recursive Functions

The problem with a function like:

```
def add x y = if (iszero y) then x else (add (succ x) (pred y))
```

Is that according to our name replacement rules, we should technically replace the definition of `add` with its definition.

Recursive Functions

```
add x y
```

```
if (iszero y)
then x
else (add (succ x) (pred y))
```

```
if (iszero y)
then x
else (if (iszero (pred y))
      then (succ x)
      else (add (succ (succ x)) (pred (pred y))))
```

etc...

Recursive Functions

Recursive functions are a vital part of functional languages and ensuring that they are supported by λ-calculus is important and relies on something called the ***paradoxical combinator*** or ***fixed-point combinator***.

Unfortunately this is somewhat involved and not really necessary for this course.

However, if you are interested you could review "An Introduction to Functional Programming Through Lambda Calculus" - Greg Michaelson, Sections 4.3-4.7.

Developing a Functional Language

These examples only show a fraction of the work necessary to construct a complete functional language from λ-calculus.

Many other features are still required - lists, floating-point numbers, input/output, a type system, etc.

Hopefully though, you can see the idea of how we could build programming languages from λ-calculus as well as where some of the functional programming ideas come from.

Functional Languages

Some key features of functional languages:

- Referential Transparency
- Recursion
- Lists
- Types
- High-Order Functions
- Lazy Evaluation

Referential Transparency

Referential transparency makes the language and programs easier to read and understand..

When a programmer writes an imperative function that interacts with the state of the program, they must ensure they understand how that state affects the rest of the program.

In functional languages, the programmer needs to understand the parameters and the result of a function in order to understand it.

Recursion

Recursion plays an important role in functional languages, it takes the place of iteration that is more commonly used in imperative languages. The calculation of factorial is a common example:

```
fac ::  num -> num
fac x = 1, if x = 0
fac x = x * fac(x-1), otherwise
```

The function `fac` involves two separate equations, each with their own condition. The system will use the first one (in order in the code) whose condition is true.

Lists

Lists are a common data structure in most functional languages. Lists are different from arrays as they have an unspecified (unknown) length.

Lists contain zero or more elements of the same type:

`[1,4,9,16]`	`list of four integers`
`[1..10]`	`list of ten integers`
`[‘u’, ‘f’, ‘o’]`	`list of three characters`
`[]`	`empty list`
`[[2,4,6],[0,3,6,9]]`	`list of two lists of integers`

List Operations

Functional languages may provide many list operations but the three simple ones are:

- `head (hd)` - selects the first element
- `tail (tl)` - selects the rest (a list)
- `cons (:)` - inserts an element

```
hd[10,20,30]      results in 10
tl[10,20,30]      results in [20,30]
5:[]              results in [5]
3:[4,5,6]         results in [3,4,5,6]
```

Types

The type systems used in functional languages are often very different from those in imperative languages.

Many functional languages use either no (***weak***) typing, ***dynamic*** or ***implicit strong*** typing and may provide the flexibility of polymorphic functions.

When everything is represented by λ-expressions, it is easier to understand how a language could have no type system (everything is effectively the same type).

Higher-Order Functions

Most imperative languages treat variables and functions ***differently*** (another example of non-orthogonality).

A function that takes another function as a argument or returns one as a result is called a ***higher-order function***.

Once again, when everything in the language is represented by λ-expressions, this is a very natural idea.

Higher-Order Functions

An example of a higher-order function is `map` which takes a function and a list as input and applies the function to each element in the list.

```
map triple [1,2,3,4] =>
[triple 1, triple 2, triple 3, triple 4] =>
[3, 6, 9, 12]
```

Higher-order functions are a powerful and useful tool. Support for similar ideas has been added to many imperative languages.

Eager Evaluation

When evaluating a function, the simplest way is to evaluate all of the arguments and then invoke the function. This is the approach that imperative languages use.

```
mult (fac 3) (fac 4)
```

The function applications `(fac 3)` and `(fac 4)` would be done first, reducing the expression to `mult 6 24`.

This approach is known as ***applicative order reduction*** or ***eager evaluation***. Expressions are evaluated from the inner-most and working towards the outer-most.

Lazy Evaluation

Functional languages often take the approach of evaluating the outermost expressions and working inwards.

`mult (fac 3) (fac 4)` is reduced to `(fac 3) * (fac 4)`.

This is possible because referential transparency ensures that the result of evaluating the arguments does not depend on when they are called.

This approach is known as ***lazy evaluation*** or ***normal order reduction***.

Example Functional Languages

We will now briefly consider some important functional languages.

- Lisp
- Haskell

Lisp

List ***P***rocessing language or ***Lisp*** was the first functional programming languages - designed in 1958 by John McCarthy. This is the second-oldest programming language still in common usage today (Fortran was developed in 1957).

The syntax of Lisp is similar to λ-calculus and is often criticised due to the excessive use of parentheses.

Consider the definition of factorial as an example:

```
(define (fac x)
   (cond ((= x 0) 1) (t (* x (fac (- x 1))))))
```

`t` is the Lisp for true

Lisp

Lisp supports many of the major features of functional languages - recursion, higher-order functions, memory allocation and takes care of automatic garbage collection.

Lisp does not have any type-system and allows lists of any elements - makes it hard to detect type errors at compile time.

While the syntax of Lisp is often hard to read due to the number of parentheses, it does mean the syntax is very uniform which makes it easier to manipulate a Lisp program as data.

Lisp

Despite its age, Lisp is still a relatively popular language - currently ranked 29th on the Tiobe Index (March 2023) - ahead of languages such as: Lua, Prolog, Julia, Scala, Groovy, TypeScript.

Lisp is also the direct ancestor of many modern functional languages such as Common Lisp and Scheme. These descendants typically introduce a type system and features such as lazy evaluation and pattern matching.

While Lisp took ideas from λ-calculus it does not actually derive from it. Many variants have been retrofitted to make them closer in spirit.

Haskell

Haskell is a statically scoped and strongly typed functional programming language. It uses the same type inferencing method as ML (Meta-Language) which means that type declarations are not always required.

```
fac ::  Int -> Int     (optional)
fac 0 = 1
fac n = n * fac (n-1)
```

Haskell uses lazy evaluation, list comprehension, pattern matching, type classes and type polymorphism.

Haskell

As a functional language, general functions in Haskell do not have any side effects (referential transparency).

However, Haskell does also support constructs called ***monads*** that (among other things) can be used to support input/output.

This does introduce an imperative aspect to Haskell but, if used correctly, the imperative component can be minimised and will not detract from the functional nature of Haskell.

Haskell

Haskell is used in some major projects such as:

Cryptol - a project used for specifying cryptographic algorithms.
`www.cryptol.net`

Sigma - the Facebook anti-spam system is now implemented using Haskell - it handles more than one million requests per second.

`https://code.facebook.com/posts/745068642270222/fighting-spam-with-haskell/`

Summary

- Developing λ-calculus into a functional language
- Functional Language Features
- Lisp
- Haskell