159.341 Programming Languages, Algorithms & Concurrency

Lambda Calculus (Part 1)

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

Origins of Functional Programming

λ-calculus is a simple but powerful system based on function abstraction and application developed by Alonzo Church as a ***model of computation***.

It is easy to always think of computation as synonymous with the computer hardware we are familiar with (or at least the ***Random Access Machine***).

We will briefly explore some of the fundamentals of λ-calculus and see how we can develop familiar programming constructs with an entirely different basis.

Origins of Functional Programming

λ-calculus is based on function abstraction and application.

Function abstraction generalises expressions through the introduction of names.

Function application evaluates generalised expressions by giving names particular values

λ-calculus

λ-calculus is a system for manipulating λ expressions.

A λ expression may be:

- A ***name*** that identifies an abstraction
- A ***function*** to introduce an abstraction
- A ***function application*** that specialises an abstraction

To make sense of this, we will look at some simple examples.

Identity Function

Consider the following function (note that the choice of `x` is completely arbitrary).

$$\lambda \texttt{x}.\texttt{x}$$

This is the identity function, it's result is whatever argument it is applied to. It has a ***bound*** variable `x` and its body expression is the name `x`.

Whenever this is applied as a function, the bound variable `x` will be replaced by the argument expression in the body expression `x` to give the original expression.

Self-Application Function

Consider the (somewhat odd) function:

$$\lambda \texttt{s.(s s)}$$

This function takes an argument and applies it to itself. The bound variable `s` and the body expression `(s s)`.

Now that we have two different expressions, we can see what happens if we apply them to each other.

Self-Application Function

Application of the identity function to the self-application function:

$$(\lambda x.x\ \lambda s.(s\ s))$$

The function expression is $\lambda x.x$ and the argument expression is $\lambda s.(s\ s)$. When the function is evaluated, the bound variable of the function expression `x` is replaced by the argument $\lambda s.(s\ s)$ in the function expression body `x`.

$(\lambda x.x\ \lambda s.(s\ s))$
=> $\lambda s.(s\ s)$

Self-Application Function

Application of the self-application function to the identity function.

$$(\lambda s.(s\ s)\ \lambda x.x)$$

The function expression is $\lambda s.(s\ s)$ and the argument expression is $\lambda x.x$. The function expression bound variable s is replaced with the argument expression $\lambda x.x$ in the function body $(s\ s)$.

$(\lambda s.(s\ s)\ \lambda x.x)$
$\Rightarrow (\lambda x.x\ \lambda x.x)$
$\Rightarrow \lambda x.x$

Self-Application Function

What happens when you apply the self-application function to itself?

$(\lambda s.(s\ s)\ \lambda s.(s\ s))$

Self-Application Function

What happens when you apply the self-application function to itself?

`(λs.(s s) λs.(s s))`

Replace the bound variable `s` in the function body `(s s)` with the argument expression `λs.(s s)` which gives:

`(λs.(s s) λs.(s s))`

and so on and so on...

Function application function

Consider the function:

λ`func.`λ`arg.(func arg)`

This function has a bound variable `func` and a body expression that is another function λ`arg.(func arg)` which has its own bound variable `arg` and a function application `(func arg)`.

When used, the whole function will return a second function which then applies the first function's argument to the second function's argument. For example:

`((`λ`func.`λ`arg.(func arg)` λ`x.x)` λ`s.(s s))`

Function application function

((λfunc.λarg.(func arg) λx.x) λs.(s s))

The bound variable func is replaced by the argument λx.x.

(λarg.(λx.x arg) λs.(s s))

The bound variable arg is replaced by the argument λs.(s s).

(λx.x λs.(s s))

The bound variable x is replaced by the argument λs.(s s).

λs.(s s)

Additional Syntax

As the λ expressions become more complex and elaborate, they quickly become very hard to work with.

To simplify working with λ expressions and start to construct a higher-level functional language, additional syntax can be introduced.

This introduction of higher-level layers is called ***syntactic sugaring*** as the underlying meaning remains the same.

Additional Syntax

It quickly becomes tedious to write out the functions over and over again so instead we introduction syntax to define functions.

def `name` = **function**

We can use the name-function association to name the functions we used in the previous examples.

def `identity` = λ`x.x`
def `self_apply` = λ`s.(s s)`
def `apply` = λ`func.`λ`arg.(func arg)`

Additional Syntax

Strictly speaking, every name in an expression should be replaced by its definition before the expression is evaluated, however we will only replace a name with its associated function when it is the function expression. We also introduce the notation:

(**name argument**) == (**function argument**)

To indicate that **name** has been replaced by its associated function **function**.

β reduction

Formally, the replacement of a bound variable with an argument in a function body is called a β ***(beta) reduction***. Rather than going through each β reduction step-by-step we will use the notation:

(**function argument**) => **expression**

To represent that **expression** is the result of applying the function **function** to **argument**.

Functions from functions

Let us define a function with the same effect as the identity function and apply it to the `identify` function.

def `identity2 = λx.((apply identity) x)`

```
(identity2 identity)                                  ==
(λx.((apply identity) x) identity)                    =>
((apply identity) identity)                           ==
((λfunc.λarg.(func arg) identity) identity)           =>
(λarg.(identity arg) identity)                        =>
(identity identity)                                   ==
...                                                   =>
identity
```

Free & Bound Variables

So far we have considered only functions which have distinct names in their expressions. When arguments are substituted in place of the bound variables of the function we have no problems.

However, it is possible for the bound variables in different functions to have the same name. Consider the following:

`(λf.(f λf.f) λs.(s s))`

Here the bound variable `f` should be replaced by `λs.(s s)` in `(f λf.f)` but not in the body of `λf.f`.

Free & Bound Variables

To clarify this situation we must be more specific about how bound variables relate to the function bodies.

For a function λ`name.body`, the bound variable `name` may correspond to occurrences of `name` inside `body` and nowhere else.

In other words, the ***scope*** of the bound variable `name` is `body`. Note that we say ***may*** correspond, this is because the reuse of name may alter a bound variable's scope.

Free & Bound Variables

This introduces the idea of ***bound*** and ***free*** variables. A variable is said to be bound to the occurrences in the body of a function (for which it is the bound variable) if no other functions within the body introduce the same bound variable. Otherwise it is said to be free.

In the expression λ`x.x` the variable `x` is bound. But in the expression `x` the variable `x` is free.

In the expression λ`f.(f` λ`x.x)`, the variable `f` is bound but in the expression `(f` λ`x.x)` the variable `f` is free.

Free & Bound Variables

Thus in general, for a function λ`name.body`, `name` refers to the same variable through `body` except for where another function has `name` as its bound variable.

Formally, all the free occurrences of `name` in `body` are references to the same bound variable `name` introduced by the original function. The variable `name` is in scope in `body` wherever it may occur as a free variable.

For example, in the body of λ`f.(f` λ`f.f))` which is `(f` λ`f.f)` the first `f` is free, so it refers to the original bound variable. All subsequent `f`s are bound and are distinct from the original bound variable.

Free & Bound Variables

This allows us to provide a more formal definition of β reduction. The β reduction of an application:

`(λname.body argument)`

Means to replace all the free occurrences of `name` in `body` with `argument`. This ensures that only the occurrences of `name` that actually correspond to the bound variable are replaced.

Free & Bound Variables

For example, consider the application:

```
(λf.(f λf.f) λs.(s s))
```

The first occurrence of `f` in the body `(f λf.f)` is free so it gets replaced.

```
(λf.(f λf.f) λs.(s s)) =>
(λs.(s s) λf.f) =>
(λf.f λf.f) =>
λf.f
```

Name Clashes & α Conversion

We have restricted the use of names in expressions to the bodies of functions. This may be restated as the requirement that there are no free variables in a λ expression.

Without this restriction, names become objects in their own right. While this eases data representation, it also makes reduction more complicated.

Given:
def `apply` = λ`func.`λ`arg.(func arg)`

Consider:
`((apply arg) identity)`

Name Clashes & α Conversion

This would evaluate as:

```
((apply arg) identity) ==
((λfunc.λarg.(func arg) arg) identity) =>
((λarg.(arg arg) identity) =>
(identity identity)
```

Which is not what was meant. The argument `arg` has been inserted within the scope of the bound variable `arg` and creates a new occurrence of that bound variable.

Name Clashes & α Conversion

Name clashes arise when a β reduction places an expression with a free variable into the scope of a bound variable with the same name.

A consistent method of renaming known as α ***(alpha) conversion*** can be used to remove the naming clash.

In a function λ`name.body`, the name `name` and all free occurrences of `name` inside `body` can be replaced by a new name `name1` provided that `name1` is not the name of a free variable in λ`name.body`.

Name Clashes & α Conversion

```
((apply arg) identity) ==
((λfunc.λarg.(func arg) arg) identity)  (α)
((λfunc.λarg1.(func arg1) arg) identity) ==>
((λarg1.(arg arg1) identity) =>
(arg identity)
```

Where the (α) denotes that we have performed an α conversion to rename the `arg` to `arg1`. Applying α conversion can remove name conflicts and ensure the meaning of the λ-expressions is unchanged.

η Reduction

Consider the expression:

λ`name.(exp name)`

This is similar to the function application function, it is equivalent to `expression` because the application of this expression to an arbitrary argument `argument` is the same as applying `expression` to `argument`.

η Reduction

For example:

`(λname.(exp name) arg) => (exp arg)`

The simplification of `λname.(exp name)` to `exp` is called η (eta) reduction.

It can be used to simplify expressions and remove an unnecessary function abstraction.

Summary

- Origins of Functional Programming
- Abstraction
- Introduction to Lambda Calculus
- Identity & Self-Application
- β-reduction
- Free & Bound Variables

Chapters 1 & 2
An Introduction to Functional Programming Through Lambda Calculus
Greg Michaelson