Functional Programming

Christopher D. Clack

FUNCTIONAL PROGRAMMING

Lecture 3

The LAMBDA CALCULUS
continued

FUNCTIONAL PROGRAMMING
The LAMBDA CALCULUS

This lecture provides more information about the λ Calculus, including the encoding of natural numbers (Church numerals), further detail about critical aspects of substitution and free variable capture, a comparison of two reduction strategies and a discussion of different kinds of normal form. After the summary, a 10-minute challenge is included – you should be ready with a pen and paper!

CONTENTS

FUNCTIONAL PROGRAMMING
The LAMBDA CALCULUS

Review of the λ Calculus

This slide provides a quick reminder of some of the definitions provided in the previous lecture:

- The formal definition of λ terms
- A simple BNF syntax
- An extended syntax with constants and infix operators
- The α, β, η and δ reduction rules

REVIEW of the λ CALCULUS

- The set of λ terms, called **Λ**, is defined as follows:

  $x \quad \in \Lambda$ *variable*
  $M \quad \in \Lambda \Longrightarrow (\lambda x\ M) \in \Lambda$ *abstraction*
  $M, N \in \Lambda \Longrightarrow (M\ N) \in \Lambda$ *application*

- BNF syntax: e :: x | e e | λx.e
- Extended syntax: e ::= c | o | x | e e | e o e | λx. e | '(' e ')'
- **α Reduction:** $\lambda x.E \longrightarrow \lambda y.E[y/x]$
- **β reduction:** $(\lambda x.E)\ z \longrightarrow E[z/x]$
- **η reduction:** $\lambda x.(E\ x) \longrightarrow E$ if x is not free in E
- **δ Rules:** Each operator (such as +, ×) has its own δ rule

FUNCTIONAL PROGRAMMING
The LAMBDA CALCULUS

ENCODING NUMBERS IN THE λ CALCULUS

- In the basic λK Calculus there are no constants
- Numbers are represented by an *encoding*: Church Numerals
- 0 is encoded as λf.λx.x
  1 is encoded as λf.λx.(f x)
  2 is encoded as λf.λx.(f (f x)
  and this extends to all whole numbers: e.g. the number 4 is encoded as
  λf.λx.(f (f (f (f x))))
- We can write λ-calculus functions to perform arithmetic on these representations. For example, a function abstraction that takes an encoding of the number 1 and an encoding of the number 2 and returns an encoding of the number 3

Encoding numbers in the λ Calculus

In the basic λK Calculus there are no constants

So how then can we represent numbers in the λK Calculus?

The answer is there is an *encoding* for numbers ("Church Numerals"). Remember that the λK Calculus is a low-level foundation for programming and is not concerned about whether such encodings are efficient or easy to use

The encoding for Natural numbers is based on function application – specifically, numbers are represented by how many times a function is applied to an argument

Thus:
0 is encoded as λf.λx.x
1 is encoded as λf.λx.(f x)
2 is encoded as λf.λx.(f (f x))
and this extends to all whole numbers:
e.g. the number 4 is encoded as
λf.λx.(f (f (f (f x))))

What makes this an *effective* encoding is that we can write λ-calculus functions to perform arithmetic on these representations. For example, it is possible to write a function abstraction that will take an encoding of the number 1 and an encoding of the number 2 and performs addition to return an encoding of the number 3

FUNCTIONAL PROGRAMMING
The LAMBDA CALCULUS

REVISITING SUBSTITUTION

Consider β reduction of $(\lambda x.((\lambda x.(x + 3))\ (x + 4)))\ 5$

Which of the two occurrences of "x" should be substituted?

Recall the β reduction rule: $(\lambda x.E)z \longrightarrow E[z/x]$

Recall the definition of substitution:

$x[N/x] \equiv N$

$y[N/x] \equiv y, \quad \text{if } x \not\equiv y$

$(\lambda y.M)[N/x] \equiv \lambda y.(M[N/x]), \text{ if } \quad x \not\equiv y$ and $y \notin FV(N)$

$(M_1\ M_2)[N/x] \equiv (M_1[N/x])(M_2[N/x])$

Applying the rules gives:

$(\lambda x.((\lambda x.(x + 3))\ (x + 4)))\ 5$

$\longrightarrow\ ((\lambda x.(x + 3))\ (x + 4))\ [5\ /x] \equiv ((\lambda x.(x + 3))[5/x])\ \ ((x + 4)[5/x])$

$\equiv (\lambda x.(x + 3))\ (5 + 4)$

Only the 2nd x is substituted

Revisiting substitution

β-reduction of the following term might cause concern due to the nested use of the variable "x" (which of the "x" should be substituted, and which not?)

$(\lambda x.((\lambda x.(x + 3))\ (x + 4)))\ 5$

Previously we defined E[N/x] as follows:

$x[N/x] \equiv N$

$y[N/x] \equiv y, \quad \text{if } x \not\equiv y$

$(\lambda y.M)[N/x] \equiv \lambda y.(M[N/x]), \text{ if } \quad x \not\equiv y$ and $y \notin FV(N)$

$(M_1\ M_2)[N/x] \equiv (M_1[N/x])(M_2[N/x])$

This can be extended to cope with syntax extensions such as constants and infix operators

But how does substitution really work?

The β reduction rule says that $(\lambda x.E)M \longrightarrow E[M/x]$, so a single β reduction of the example above should give:

$((\lambda x.(x + 3))\ (x + 4))\ [5\ /x]$

In this case E is of the form $(M_1\ M_2)$ with $M_1 = (\lambda x.(x + 3))$ and $M_2 = (x + 4)$. So we can apply the rules to give:

$((\lambda x.(x + 3))[5/x])\ ((x+4)[5/x])$

The rightmost term is easy (it gives (5 + 4)). The leftmost term fails the constraint $x \not\equiv y$ and no substitution occurs. Thus, overall we have:

$(\lambda x.((\lambda x.(x + 3))\ (x + 4)))\ 5$

$\longrightarrow\ (\lambda x.(x + 3))\ (5+4)$

Only the 2nd x is substituted

FUNCTIONAL PROGRAMMING
The LAMBDA CALCULUS

FREE VARIABLE CAPTURE

For β-reduction, identifying free variables in E is not enough!

Consider β-reduction of the following term:

$((\lambda x.(\lambda y.(y\ x)))\ M)\ N \longrightarrow (\lambda y.(y\ M))\ N \longrightarrow (N\ M)$

But if M=y and N=x then we have

$((\lambda x.(\lambda y.(y\ x)))\ y)\ x \longrightarrow (\lambda y.(y\ y))\ x \longrightarrow (x\ x)$?

Also consider the following term where the second "a" (underlined) is free:

$(\lambda f.(\lambda a.(f\ a)))\ (\lambda f.(f\ \underline{a})) \longrightarrow (\lambda a.((\lambda f\ .(f\ a))\ a))$

Following β-reduction the free occurrence of "a" has been "captured" by λ*a*, which is not the behaviour we want

These are two examples of "free variable capture"

Free variable capture

For β-reduction, identifying free variables in *E* is not enough!

Consider β-reduction of the following term:

$((\lambda x.(\lambda y.(y\ x)))\ M)\ N$
$\longrightarrow (\lambda y.(y\ M))\ N$
$\longrightarrow (N\ M)$

But if M=y and N=x then we have

$((\lambda x.(\lambda y.(y\ x)))\ y)\ x$
$\longrightarrow (\lambda y.(y\ y))\ x$
$\longrightarrow (x\ x)$

This is not the behaviour we want!

Next consider the following term where the first *a* is bound and the second *a* (in the argument) is free:

$(\lambda f.(\lambda a.(f\ a)))\ \ (\lambda f.(f\ a))$

This β-reduces to the following, where *both* copies of the name *a* are now bound (the second *a*, previously free, has been "captured" and is now bound by the λ*a*, which is **not** the behaviour we want) :

$(\lambda a.((\lambda f\ .(f\ a))\ a))$

These are two examples of "free variable capture" caused by name clashes

β-reduction needs to be more sophisticated in the way that it operates, and on the next slide we shall see how this can be done

FUNCTIONAL PROGRAMMING
The LAMBDA CALCULUS

AVOIDING FREE VARIABLE CAPTURE

During β-reduction of (λ*x*.*E*) *M*, if *M* is an expression that contains any free variables that are bound inside *E*, then each such free variable must be α-converted inside *E* **before** performing the β-reduction substitution.

Both examples on the previous slide now behave correctly:

((λx.(λy.(y x))) y) x
⟶ ((λx.(λz.(z x))) y) x (α reduction)
⟶ (λz.(z y)) x (β reduction)
⟶ (x y) (β reduction)

(λ*f* .(λ*a*.(*f a*))) (λ*f* .(*f a*))
⟶ (λ*f* .(λ*b*.(*f b*))) (λ*f* .(*f a*)) (α reduction)
⟶ (λ*b*.((λ*f* .(*f a*)) *b*)) (β reduction)

Avoiding free variable capture

To avoid free variable capture during β reduction, we use the following rule:

During β-reduction of (λ*x*.*E*) *M*, if *M* is an expression that contains any free variables that are bound inside *E*, then each such free variable must have its name changed by α-reduction inside *E* **before** performing the β-reduction substitution

If we use this rule then both examples on the previous slide behave correctly:

((λx.(λy.(y x))) y) x
⟶ ((λx.(λz.(z x))) y) x (α reduction)
⟶ (λz.(z y)) x (β reduction)
⟶ (x y) (β reduction)

(λ*f* .(λ*a*.(*f a*))) (λ*f* .(*f a*))
⟶ (λ*f* .(λ*b*.(*f b*))) (λ*f* .(*f a*)) (α reduction)
⟶ (λ*b*.((λ*f* .(*f a*)) *b*)) (β reduction)

Question: for β reduction of (λx.E) M where M has free variables that are bound inside E, why do you think it is important to α reduce inside E rather than inside M?

FUNCTIONAL PROGRAMMING
The LAMBDA CALCULUS

COMPARING REDUCTION STRATEGIES

Normal Order (NO) reduction (leftmost-outermost-first) is similar to "call by reference" passing of function arguments and is guaranteed to terminate if termination is possible. Simple implementations can be slow.

Applicative Order (AO) reduction (leftmost-innermost-first) is similar to "call by value" passing of function arguments and often faster than NO but may not always terminate

$(\lambda x.(x+x))\ (3+5) \longrightarrow ((3+5)+(3+5)) \longrightarrow (8 + (3+5))$
$\longrightarrow\ (8 + 8) \longrightarrow\ 16$ (NO)

$\longrightarrow (\lambda x.(x+x))\ 8 \longrightarrow\ (8 + 8) \longrightarrow\ 16$ (AO)

$(\lambda x.3)\ ((\lambda x.(x\ x))\ (\lambda x.(x\ x)))$ $\longrightarrow 3$ (NO)
$\longrightarrow \infty$ LOOP (AO)

The ∞ loop occurs because reduction of $((\lambda x.(x\ x))\ (\lambda x.(x\ x)))$ loops forever

Comparing reduction strategies

Two popular reduction strategies used by functional languages are Normal Order and Applicative Order reduction

Normal Order (NO) reduction (leftmost-outermost-first) is a "normalizing" strategy that is guaranteed to terminate if termination is possible

Normal order is similar to "call by reference" passing of function arguments, though simple implementations can suffer from duplication and therefore be slow

Applicative Order (AO) reduction (leftmost-innermost-first) may not terminate for some terms where Normal Order would terminate. It is similar to "call by value" passing of function arguments and in simple implementations it is often faster than Normal Order reduction

$(\lambda x.(x+x))\ (3+5) \longrightarrow ((3+5)+(3+5))$
$\longrightarrow (8 + (3+5))$
$\longrightarrow\ (8 + 8) \longrightarrow\ 16$ (NO)

$\longrightarrow (\lambda x.(x+x))\ 8$
$\longrightarrow\ (8 + 8) \longrightarrow\ 16$ (AO)

$(\lambda x.3)\ ((\lambda x.(x\ x))\ (\lambda x.(x\ x)))$
$\longrightarrow 3$ (NO)
$\longrightarrow \infty$ LOOP (AO)

The infinite loop occurs because reduction of $((\lambda x.(x\ x))\ (\lambda x.(x\ x)))$ loops forever (just as it is possible to write infinite loops in other programming languages):

FUNCTIONAL PROGRAMMING
The LAMBDA CALCULUS

DIFFERENT KINDS OF NORMAL FORM

- A term is in **Normal Form (NF)** if it doesn't have a β redex
  x is in NF
  M N is in NF if M, N are in NF and M is not an abstraction
  (λx.E) is in NF if E is in NF

- A term *M* is in Head Normal Form (HNF) if it is of the form $M \equiv \lambda x_1 \ldots x_n \,.\, (x\ N_1 \ldots N_m)$ *where* $n,m \geq 0$ *and x is a variable ("head" variable). HNF is not unique*
  x is in HNF
  x $N_1 \ldots N_m$ *is in HNF*
  (λx.E) is in HNF if E is in HNF

- A term M is in Weak Head Normal Form (WHNF) if it is either (i) $M \equiv \lambda x_1 \ldots x_n \,.\, (x\ N_1 \ldots N_m)$ *where* $n,m \geq 0$ *and x is a variable or (ii)* $M \equiv \lambda x.N$. *WHNF is not unique*
  x is in WHNF
  x N is in WHNF
  (λx.E) is in WHNF

Different kinds of Normal Form

To avoid wasted computation, practical implementations of functional languages almost never reduce to full Normal Form

Weak Head Normal Form (WHNF) and Head Normal Form (HNF) are successive stopping points on the journey to full Normal Form

Assuming the unextended λK Calculus:

- A term is in **Normal Form (NF)** if it does not contain a β redex. NF is unique

  x is in NF
  M N is in NF if M, N are in NF and M is not an abstraction
  (λx.E) is in NF if E is in NF

- A term M is in **Head Normal Form (HNF)** if it is of the form $M \equiv \lambda x_1 \ldots x_n \,.\, (x\ N_1 \ldots N_m)$ *where* $n,m \geq 0$ *and x is a variable ("head" variable). HNF is not unique*

  x is in HNF
  x $N_1 \ldots N_m$ *is in HNF*
  (λx.E) is in HNF if E is in HNF

- A term M is in **Weak Head Normal Form (WHNF)** if it is either (i) $M \equiv \lambda x_1 \ldots x_n \,.\, (x\ N_1 \ldots N_m)$ *where* $n,m \geq 0$ *and x is a variable or (ii)* $M \equiv \lambda x.N$. *WHNF is not unique*

  x is in WHNF
  x N is in WHNF
  (λx.E) is in WHNF

FUNCTIONAL PROGRAMMING
The LAMBDA CALCULUS

EXAMPLE EVALUATIONS

$$\lambda x.(((\lambda y.(\lambda p.\lambda q.(q\ p)))\ 4)\ (5 + 6))$$

Evaluate to WHNF (not unique):

$$\lambda x.(((\lambda y.(\lambda p.\lambda q.(q\ p)))\ 4)\ (5 + 6))$$

Evaluate further to HNF (not unique):

$$\lambda x\ .(\lambda q.(q\ (5 + 6)))$$

Then to NF (unique and cannot be evaluated any further):

$$\lambda x.(\lambda q.(q\ 11))$$

Example evaluations

Start with the expression:

$\lambda x.(((\lambda y.(\lambda p.\lambda q.(q\ p)))\ 4)\ (5 + 6))$

This expression can be evaluated to WHNF (not unique):

$\lambda x.(((\lambda y.(\lambda p.\lambda q.(q\ p)))\ 4)\ (5 + 6))$

It can be evaluated further to HNF (not unique):

$\lambda x\ .(\lambda q.(q\ (5 + 6)))$

And can be evaluated even further to NF (which is unique and cannot be evaluated any further):

$\lambda x.(\lambda q.(q\ 11))$

FUNCTIONAL PROGRAMMING
The LAMBDA CALCULUS

In summary, this lecture has given more information about the λ Calculus, including the encoding of natural numbers (Church numerals), further detail about critical aspects of substitution and free variable capture, a comparison of two reduction strategies and a discussion of different kinds of normal form. On the next slide there is a 10-minute coding challenge

SUMMARY

- Review of the λ Calculus
- Encoding numbers in the λ Calculus
- Substitution and Free Variable Capture
- Comparing reduction strategies
- Different kinds of Normal Form
- Summary
- Challenge (the next slide)

FUNCTIONAL PROGRAMMING
The LAMBDA CALCULUS

CHALLENGE

Time's up!

Challenge — can you write a λ-calculus function for addition of two Church numerals, given the encoding of numbers repeated below? Get a pen and paper now!

0 is encoded as λf.λx.x
1 is encoded as λf.λx.(f x)
2 is encoded as λf.λx.(f (f x))
3 is encoded as λf.λx.(f (f (f x)))

Hints:

1. A function abstraction to add two numbers must take two arguments – call them a and b

   λa.(λb.M)

2. When your function is applied to the encodings for numbers 1 and 2 it should return the encoding for the number 3, so: ((λa.(λb.M)) (λf.λx.(f x))) (λf.λx.(f (f x))) →

   λf.λx.(f (f (f x)))

Solving this challenge will start to get you thinking like a functional programmer, and you need to engage with the mental struggle. If you solve it in 2 minutes or under, it was too easy for you and you will have learnt very little. If you haven't solved it in 10 minutes we will discuss it in the group Q&A (but you will have learnt something during the struggle). As long as you grapple with the problem you will learn from the experience

TRY IT NOW (10 minute timer on the left)

FUNCTIONAL PROGRAMMING
The LAMBDA CALCULUS