COMP4630: \(\lambda\)-Calculus

1. Basics

Michael Norrish

Michael.Norrish@data61.csiro.au

Data61, CSIRO

Semester 2, 2019
Outline

Introduction

Lambda Calculus Terms
   Alpha Equivalence
   Substitution

Dynamics
   Beta Reduction
   Eta Reduction
   Normal Forms
   Evaluation Strategies

Conclusion
The Lambda Calculus

Alonzo Church (1903–1995)

The $\lambda$-calculus is a fundamental computational model.

It also has a number of connections to logic.

The next five lectures (including this one) will concentrate on the untyped calculus.

The typed calculus has the most elegant connections to logic, so we will focus on computation.

But there is also an interesting equational logic to consider.
Course Plan

By me:

1. Basics (today)
2. Equational Logic
   - connection to $\xrightarrow{\beta}$
   - examples
   - inconsistent assumptions
   - soundness & completeness
3. Soundness via Church-Rosser
4. Standardisation
5. Computation
   - Church Numerals
   - Encoding Other Types
   - Encoding the Recursive Functions

And then, five lectures on the typed calculus by Jeremy Dawson.

What are Lambda Terms?

Lambda terms make up the world’s simplest programming language.

A lambda term is either
- a variable \( v \); or
- the application of term \( M \) to term \( N \), written \( MN \); or
- the abstraction of variable \( v \) in term \( M \), written \((\lambda v. M)\).

Examples:
- \( fx \) — \( f \), a “function”, applied to an argument \( x \)
- \((fx)y\) — a function applied to two arguments
- \(f(gx)\) — two function calls
- \((\lambda v. v)\) — the identity function
- \((\lambda u. (\lambda v. u z))x\) — abstraction and application
Term Shorthands

Applications are left-associative.
So, you can write $MN\ P$, and don’t have to write $(M\ N)\ P$
(You do have to write $M\ (N\ P)$ with the parentheses.)

You can “chain” binders.
Instead of $(\lambda u.\ (\lambda v.\ M))$, just write $(\lambda u v.\ M)$

Thus:

$$S = (\lambda a b c.\ a\ c\ (b\ c))$$

instead of

$$S = (\lambda a.\ (\lambda b.\ (\lambda c.\ (a\ c)\ (b\ c))))$$

Think of abstract syntax trees if necessary.
Abstractions are (Anonymous) Functions

In secondary school mathematics, you learn to write things like

\[ f(x) = x^2 + 2x + 1 \]

meaning that \( f \) is a function that takes a parameter \( x \) and returns a value derived from that \( x \).

When you learnt to program, you might have learnt to write

```c
int f(int x)
{
    return x*x + 2*x + 1;
}
```

instead.
Abstractions are (Anonymous) Functions

In $\lambda$-calculus (with arithmetic added), you might write

$$f = (\lambda x. x^2 + 2x + 1)$$

In other words,

$$(\lambda x. x^2 + 2x + 1)$$

is a function that takes a value $x$ as a parameter, and calculates a “return value”.

This $\lambda$-term is a function without a name. We can decide to give it the name $f$ (or $g$, or nothing) later.
Consider: \((\lambda v. u \ v)\)

The variable \(v\) is \textit{bound}. It’s the name of a parameter.

The variable \(u\) is \textit{free}.

It’s \textit{not} the name of an enclosing parameter.

The same notions are apparent in other languages:

```plaintext
int f(int v)
{
    return u + v;
}
```
Bound Names Can be Renamed

These two functions are the same:

```c
int f(int v) {
    return u + v;
}
```

```c
int f(int x) {
    return u + x;
}
```

But this one is different:

```c
int f(int u) {
    return u + u;
}
```

Clearly, not all bound variable renaming is OK.
Two \( \lambda \)-terms \( M \) and \( N \) are alpha-equivalent (written \( M \equiv N \)) if they

“are the same up to renaming of bound variables”

Proof rules

\[
\begin{align*}
  v \equiv v & \quad \frac{M_1 \equiv M_2 \quad N_1 \equiv N_2}{M_1 \ N_1 \equiv M_2 \ N_2} \\
  \text{\( v \) not free in } M & \quad \frac{N \equiv (u \ v) \cdot M}{(\lambda u. \ M) \equiv (\lambda v. \ N)}
\end{align*}
\]

where \( (u \ v) \cdot M = \text{“swap } u \text{ and } v \text{ everywhere they appear in } M \)”
Rule for \((\lambda v. M)\) again:

\[
\begin{align*}
\text{\(v\) not free in \(M\)} & \Rightarrow \text{\(N \equiv (u v) \cdot M\)} \\
\text{\((\lambda u. M) \equiv (\lambda v. N)\)}
\end{align*}
\]

where \((u v) \cdot M = \text{“swap } u \text{ and } v \text{ everywhere they appear in } M\)”

So,

\[
(\lambda v. v u)\ x \equiv (\lambda w. w u)\ x \not\equiv (\lambda u. u u)\ x
\]
Rule for \((\lambda v. M)\) again:

\[
\begin{align*}
\text{\(v\) not free in \(M\)} &\quad \text{\(N \equiv (uv) \cdot M\)} \\
(\lambda u. M) &\equiv (\lambda v. N)
\end{align*}
\]

where \((uv) \cdot M = \text{“swap \(u\) and \(v\) everywhere they appear in \(M\)”}\)

So,

\[
\begin{align*}
(\lambda v. v u) \ x &\equiv (\lambda w. w u) \ x \neq (\lambda u. u u) \ x \\
(\lambda u. (\lambda v. u v) \ u) &\equiv (\lambda v. (\lambda u. v u) \ v) \neq (\lambda u. (\lambda u. u v) \ u)
\end{align*}
\]
Alpha Equivalence—Examples

Rule for \( (\lambda v. M) \) again:

\[
\begin{align*}
\text{\( v \) not free in \( M \)} & \quad N \equiv (uv) \cdot M \\
(\lambda u. M) & \equiv (\lambda v. N)
\end{align*}
\]

where \( (uv) \cdot M = \text{“swap } u \text{ and } v \text{ everywhere they appear in } M \” \)

So,

\[
\begin{align*}
(\lambda v. v u) x & \equiv (\lambda w. w u) x \neq (\lambda u. u u) x \\
(\lambda u. (\lambda v. u v) u) & \equiv (\lambda v. (\lambda u. v u) v) \neq (\lambda u. (\lambda u. u v) u) \\
(\lambda x. (\lambda y. f y) x) & \equiv (\lambda y. (\lambda y. f y) y) \neq (\lambda f. (\lambda y. f y) f)
\end{align*}
\]
Substitution

Substitution is a ternary operation:

\[ M[v := N] \]

means “replace free occurrences of \( v \) in \( M \) with \( N \)”

(making sure that free variables in \( N \) are not captured by binders in \( M \))

(alpha-convert \( M \) as necessary)

Conditions:

- **Freeness**: \( (\lambda v. M)[v := N] \equiv (\lambda v. M) \)
  
  (as \( v \) is not free in \( (\lambda v. M) \))

- **Capture-avoiding**: \( (\lambda v. u \ v)[u := v] \neq (\lambda v. v \ v) \) (!!!)
Substitution

Substitution is a ternary operation:

\[ M[v := N] \]

means “replace free occurrences of \( v \) in \( M \) with \( N \)”

(making sure that free variables in \( N \) are not captured by binders in \( M \))

(alpha-convert \( M \) as necessary)

Conditions:

- **Freeness:** \((\lambda v. M)[v := N] \equiv (\lambda v. M)\)
  (as \( v \) is not free in \( (\lambda v. M) \))

- **Capture-avoiding:**

\[
(\lambda v. u v)[u := v] \not\equiv (\lambda v. v v) \quad (!!!) \\
(\lambda v. u v)[u := v] \equiv (\lambda w. u w)[u := v] \\
\equiv (\lambda w. v w)
\]
Substitution: Please Take Care!

Substitution is tricky!

You have seen it before (in handling f.o.l. quantifiers)

Use the Barendregt Variable Convention:
rename bound variables so they don’t overlap with free variables.
Substitution Drives Computation

A General Programming Language Axiom:

When you apply a function to an argument, the result is as if you substituted the actual argument for the formal parameter and then performed the specified computation.

For example, how do we figure out what $f(3, 5)$ will return?

```c
int f(int x, int y)
{
    return 2*x + y;
}
```
Beta Reduction: $\lambda$-Terms Doing Things

Substitution of “actuals for formals” is the essence of beta-reduction:

$$(\lambda v. M) N \rightarrow_\beta M[v := N]$$

Reduction can occur anywhere within a term.

Examples:

$$(\lambda v. v) (\lambda x. x v) \rightarrow_\beta (\lambda x. x v)$$
Beta Reduction: λ-Terms Doing Things

Substitution of “actuals for formals” is the essence of beta-reduction:

\[(\lambda v. M) N \rightarrow_\beta M[v := N]\]

Reduction can occur anywhere within a term.

Examples:

\[(\lambda v. v) (\lambda x. x v) \rightarrow_\beta (\lambda x. x v)\]
\[(\lambda u. (\lambda v. v z) (u z)) \rightarrow_\beta (\lambda u. (u z) z)\]
Beta Reduction: \( \lambda \)-Terms Doing Things

Substitution of “actuals for formals” is the essence of beta-reduction:

\[
(\lambda v. M) \, N \rightarrow_\beta M[v := N]
\]

Reduction can occur anywhere within a term.

Examples:

\[
(\lambda v. v) \, (\lambda x. x \, v) \rightarrow_\beta (\lambda x. x \, v)
\]

\[
(\lambda u. (\lambda v. v \, z) \, (u \, z)) \rightarrow_\beta (\lambda u. (u \, z) \, z)
\]

\[
(\lambda u. (\lambda v. v \, u) \, (u \, z)) \, v \, z \rightarrow_\beta (\lambda w. w \, v) \, (v \, z) \, z
\]

\[
\rightarrow_\beta (v \, z) \, v \, z
\]
Beta Reduction: λ-Terms Doing Things

Substitution of “actuals for formals” is the essence of beta-reduction:

\[(\lambda v. M) N \rightarrow^\beta M[v := N]\]

Reduction can occur anywhere within a term.

Examples:

\[
\begin{align*}
(\lambda v. v) (\lambda x. x v) & \rightarrow^\beta (\lambda x. x v) \\
(\lambda u. (\lambda v. v z) (u z)) & \rightarrow^\beta (\lambda u. (u z) z) \\
(\lambda u. (\lambda v. v u) (u z)) v z & \rightarrow^\beta (\lambda w. w v) (v z) z \\
& \quad \rightarrow^\beta (v z) v z \\
(\lambda x. x x) (\lambda y. y y) & \rightarrow^\beta (\lambda y. y y) (\lambda y. y y) \\
& \equiv (\lambda x. x x) (\lambda y. y y)
\end{align*}
\]
Beta Reduction Rules and Notation

\[(\lambda v. M) \; N \rightarrow_\beta M[v := N]\]

\[M \rightarrow_\beta M'\]

\[M \; N \rightarrow_\beta M' \; N\]

\[N \rightarrow_\beta N'\]

\[M \; N \rightarrow_\beta M \; N'\]

\[M \rightarrow_\beta M'\]

\[(\lambda v. M) \rightarrow_\beta (\lambda v. M')\]

Write \(M \rightarrow^*_\beta N\) to mean \(M\) can take zero or more \(\beta\)-reduction steps and evolve to \(N\).

Write \(M \rightarrow^+_\beta N\) to mean \(M\) can take one or more \(\beta\)-reduction steps and evolve to \(N\).
Beta Reduction Does It All!

Lambda Terms + Beta Reduction = All Computation

Any computation can be encoded in what we have just seen.

We will look at computational aspects of the $\lambda$-calculus in detail in later lectures.

In particular, it is easy to encode
- Numbers
- Recursion

And that’s all you need.
We can also add another computational rule, $\eta$-reduction:

$$
\frac{\nu \text{ not free in } M}{(\lambda \nu. M \nu) \rightarrow^\eta M}
$$

(Can perform $\eta$-reduction anywhere within a term, as with $\beta$.)

We will later see how $\eta$ ties into extensionality.

Eta-reduction and eta-expansion (!) also play a role in the typed $\lambda$-calculus.
Can Combine Beta and Eta

For example:

\[(\lambda f. f \ x) \ (\lambda y. g \ y) \]\n
\[\rightarrow^\beta \eta (\lambda f. f \ x) \ g\]

\[\rightarrow^\beta \eta g \ x\]
Can Combine Beta and Eta

For example:

\[(\lambda f. f\ x) (\lambda y. g\ y)\]

\[\rightarrow_{\beta\eta} (\lambda f. f\ x)\ g\]

\[\rightarrow_{\beta\eta} g\ x\]

Or:

\[(\lambda f. f\ x) (\lambda y. g\ y)\]

\[\rightarrow_{\beta\eta} (\lambda y. g\ y)\ x\]

\[\rightarrow_{\beta\eta} g\ x\] (Eta and Beta coincide here)
A normal form is a term that cannot be further reduced.

For example, \((\lambda x. g x)\) is a \(\beta\)-normal form, but not an \(\eta\)-normal form.

It is reasonable to think of normal forms as values towards which we want evaluation to proceed.
Normal Form Questions

It is natural to ask:

1. Does term $M$ have a normal form?
2. Does $M$ have more than one normal form?

And, we can ask these questions of all terms too.
Lambda Term Evaluation is Non-Deterministic

Recall:

\[
\begin{align*}
M \rightarrow^\beta & \ M' \\
M \ N \rightarrow^\beta & \ M' \ N \\
N \rightarrow^\beta & \ N'
\end{align*}
\]

Remember also:

\[
\begin{align*}
(\lambda x. \ x \ x) \ (\lambda y. \ y \ y) \rightarrow^\beta & \ (\lambda y. \ y \ y) \ (\lambda y. \ y \ y)
\end{align*}
\]

Call \((\lambda x. \ x \ x) \ (\lambda y. \ y \ y)\), the self-looping term, \(\Omega\).
Infinite Loops (Depending on Evaluation Order)

Consider what \( (\lambda x. y) \Omega \) might do.

- If we keep evaluating the argument (\( \Omega \)), we never stop:
  \[
  (\lambda x. y) \Omega \to_{\beta} (\lambda x. y) \Omega \to_{\beta} (\lambda x. y) \Omega \to_{\beta} \ldots
  \]

- If we apply the top-level function, substituting \( \Omega \) for \( x \) we get:
  \[
  (\lambda x. y) \Omega \to_{\beta} y
  \]

Done!

This term \( (\lambda x. y) \Omega \) has one normal form \( y \), but not all evaluation choices reach it.
How Do Other Languages Do This?

```c
int global = 3;

unsigned f(void)
{
    unsigned x = 0;
    while (1) { x++; }
    return x;
}

int g(unsigned i) { return global; }

int main(void) { return g(f()); }
```

What happens in C?
Evaluation Strategy #1: Applicative Order

Evaluate everything from the bottom up.

- I.e., in \((\lambda v. M) N\) work will start with \(M\), passing to \(N\) and performing the top-level \(\beta\)-reduction last

A function’s arguments (and the function itself) will be evaluated before the argument is passed to the function.

Also known as strict evaluation.

(Used in Fortran, C, Pascal, Ada, SML, Java† . . .)

Causes \((\lambda x. y) \Omega\) to go into an infinite loop.
Evaluation Strategy #2: Normal Order

Evaluate top-down, left-to-right.

- With \((\lambda v. M) N\), start by performing the \(\beta\)-reduction, producing \(M[v := N]\)
- Find the top-most, left-most \(\beta\)-redex and reduce it.
- Keep going

This strategy is behind the lazy evaluation of languages like Haskell.

Normal order evaluation will always terminate with a normal form if a term has one. (Proof in Lecture 4)
Strategy Rules

Write $\mathcal{M} \rightarrow_n \mathcal{N}$ for “$\mathcal{M}$ normal order reduces to $\mathcal{N}$”. Then:

$$ (\lambda v. \mathcal{M}) \mathcal{N} \rightarrow_n \mathcal{M}[v := \mathcal{N}]$$

$$ \mathcal{M} \rightarrow_n \mathcal{M}' $$

$$ (\lambda v. \mathcal{M}) \rightarrow_n (\lambda v. \mathcal{M}') $$

$$ \mathcal{M} \rightarrow_n \mathcal{M}' \quad \text{M not an abstraction} $$

$$ \mathcal{M} \mathcal{N} \rightarrow_n \mathcal{M}' \mathcal{N} $$

$$ \mathcal{N} \rightarrow_n \mathcal{N}' \quad \text{M not an abstraction} $$

$$ \mathcal{M} \text{ in } \beta\text{-nf} $$

$$ \mathcal{M} \mathcal{N} \rightarrow_n \mathcal{M} \mathcal{N}' $$

Rules for applicative order evaluation will be an assignment question.
Summary

Lambda Terms:
- Variables, applications, abstractions
- Bound names, free names, alpha equivalence
- Substitution

Dynamic Behaviour:
- Beta reduction: computation through substitution
- Eta reduction
- Normal forms, and strategies for achieving them