COMP4630: $\lambda$-Calculus

5. Computability

Michael Norrish
Michael.Norrish@data61.csiro.au

Data61, CSIRO

Semester 2, 2019
Last Time

An involved proof.

Lesson #1: The $\lambda$-calculus is a plausible programming language
  ▷ there is an algorithm for turning $\lambda$-terms into values
  ▷ (when those terms have values at all)

Lesson #2: Evaluation Orders are a Design Question
  ▷ Do you want to guarantee as much termination as possible?
    ▷ Use normal order
  ▷ Or, do you want more speed, and unnecessary non-terminations?
    ▷ Use applicative order (like C, Java etc)
Today

What Makes a Computer?
  Recursive Functions

Church Numerals

Richer Types
  Lists

Conclusion
Introduction

Today’s Objective:

Show that the \( \lambda \)-calculus is capable of all possible computation.

This requires us to know what “all possible computation” means.

For this we consult the Church-Turing Thesis.
The Church-Turing Thesis

In simple terms, the Church-Turing thesis states that a function is algorithmically computable if and only if it is computable by a Turing machine.

—Wikipedia, 2012

All computational models invented so far have been shown equivalent to Turing Machines.

Two important examples:

- The Recursive Functions
- The $\lambda$-Calculus
Computational Equivalence

Two models $M_1$ and $M_2$ are computationally equivalent if

- they each compute the same functions; or
- each can simulate the other

We will assume that Turing Machines and the Recursive Functions are equivalent.

We’ll show that the $\lambda$-Calculus can simulate the Recursive Functions.

- That will mean that the $\lambda$-Calculus is at least as powerful as the Recursive Functions
- Showing the other direction (that the Recursive Functions can simulate the $\lambda$-Calculus) is very fiddly.
Recursive functions are functions that take natural number arguments and return natural number results.

Different recursive functions can take different numbers of arguments.

For example, factorial \((\mathbb{N} \rightarrow \mathbb{N})\) and addition \((\mathbb{N}^2 \rightarrow \mathbb{N})\) are both recursive functions.

Not all functions of this sort are recursive! (Consider cardinality)
Recursive Functions—Base Cases

The following are all recursive functions

- Zero: \( n \mapsto 0 \)  
  (written \( Z \))

- Successor: \( n \mapsto n + 1 \)  
  (written \(Suc\))

- Projection: \( \langle n_1, n_2, \ldots n_m \rangle \mapsto n_i \)  
  (written \( \pi^m_i \))
Combining Recursive Functions: Composition

If $f$ and $g$ are recursive functions, so too is $f \circ g$.

If $f$ expects $n$ arguments, can compose with $n$ different others:

$$f \circ \langle g_1, \ldots, g_n \rangle$$

For example,

$$+ \circ \langle \times, f \rangle$$

is a recursive function that behaves:

$$(+ \circ \langle \times, f \rangle)(x, y) = xy + f(x, y)$$
If $f : \mathbb{N}^n \to \mathbb{N}$ and $g : \mathbb{N}^{n+2} \to \mathbb{N}$ are recursive,

then $\Pr \langle f, g \rangle$ is recursive also, taking $n + 1$ arguments such that

$$\Pr \langle f, g \rangle(0, x_1, \ldots, x_n) = f(x_1, \ldots, x_n)$$

$$\Pr \langle f, g \rangle(m + 1, x_1, \ldots, x_n) = g(m, \Pr \langle f, g \rangle(m, x_1, \ldots, x_n), x_1, \ldots, x_n)$$
Primitive Recursion: Example

\[
\text{Pr}\langle f, g \rangle(0, x_1, \ldots, x_n) = f(x_1, \ldots, x_n)
\]

\[
\text{Pr}\langle f, g \rangle(m + 1, x_1, \ldots, x_n) = g(m, \text{Pr}\langle f, g \rangle(m, x_1, \ldots, x_n), x_1, \ldots, x_n)
\]

Let \( f : \mathbb{N} \to \mathbb{N} \) be \( \pi_1^1 \), the identity function (a projection). Let \( g : \mathbb{N}^3 \to \mathbb{N} \) be \( \text{Suc} \circ \pi_2^3 \).

Then

\[
\text{Pr}\langle \pi_1^1, \text{Suc} \circ \pi_2^3 \rangle(0, y) = \pi_1^1(y) = y
\]

\[
\text{Pr}\langle \pi_1^1, \text{Suc} \circ \pi_2^3 \rangle(x + 1, y)
\]

\[
= (\text{Suc} \circ \pi_2^3)(x, \text{Pr}\langle \pi_1^1, \text{Suc} \circ \pi_2^3 \rangle(x, y), y)
\]

\[
= \text{Suc}(\pi_2^3(x, \text{Pr}\langle \pi_1^1, \text{Suc} \circ \pi_2^3 \rangle(x, y), y))
\]

\[
= \text{Suc}(\text{Pr}\langle \pi_1^1, \text{Suc} \circ \pi_2^3 \rangle(x, y))
\]

In other words, \( \text{Pr}\langle \pi_1^1, \text{Suc} \circ \pi_2^3 \rangle \) is addition.
Using just these facilities, we can define

- Addition (as per previous slide)
- Predecessor
  - predecessor of 0 is 0
- Subtraction (repeat predecessor)
  - Write $m \div n$ to mean $m - n$ if $m \geq n$, 0 otherwise
- Multiplication (repeat addition)
- Exponentiation (repeat multiplication)
- Equality Tests
- ...
Combining Recursive Functions: Minimisation

Let \( f \) be a recursive function taking \( n > 1 \) arguments.

Then \( \mu f \) is also a recursive function, taking \( n - 1 \) arguments.

If there is an \( x \) where \( f(x, y_1, \ldots, y_{n-1}) = 0 \),
(and \( f(y, y_1, \ldots, y_{n-1}) \) is defined for all \( y < x \)) then
\( \mu f(y_1, \ldots, y_{n-1}) \) returns the least such \( x \).

Otherwise \( \mu f(y_1, \ldots, y_{n-1}) \) is undefined.

For example, if \( f(x, y) = (x^2 \div y) + (y \div x^2) \), then
\[
\begin{align*}
\mu f(9) & = 3 \\
\mu f(10) & = \bot \quad \text{(undefined)}
\end{align*}
\]
So, How Do We Do All This With the λ-Calculus?

Important things:

- Numbers
- Primitive Recursion
- Representation of Multiple Arguments (easy!)
- Minimisation (unbounded search)
Church Numerals

Represent number $n$ with the $\lambda$-term $(\lambda f \ x. \ f^n \ x)$.

Thus

<table>
<thead>
<tr>
<th>Number</th>
<th>Term</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>$(\lambda f \ x. \ x)$</td>
</tr>
<tr>
<td>1</td>
<td>$(\lambda f \ x. \ f \ x)$</td>
</tr>
<tr>
<td>2</td>
<td>$(\lambda f \ x. \ f \ (f \ x))$</td>
</tr>
<tr>
<td></td>
<td>...</td>
</tr>
<tr>
<td>7</td>
<td>$(\lambda f \ x. \ f \ (f \ (f \ (f \ (f \ (f \ (f \ x)))))))$</td>
</tr>
<tr>
<td></td>
<td>...</td>
</tr>
</tbody>
</table>

A number is represented by its own “recursion operator”:

*Give a number $n$ something to do $n$ times to a base case*
Operations on Church Numerals: Successor

\[ \text{Suc} = (\lambda n. (\lambda f x. f (n f x))) \]

In action:

\[
\text{Suc}(3) \equiv \text{Suc}(\lambda f x. f (f (f x))) \\
= (\lambda f x. f ((\lambda f x. f (f (f x))) f x)) \\
= (\lambda g y. g ((\lambda f x. f (f (f x))) g y)) \\
= (\lambda g y. g ((\lambda x. g (g (g x))) y)) \\
= (\lambda g y. g (g (g (g y)))) \\
= 4
\]
Operations on Church Numerals: Addition

Numbers embody recursive operations.

\[ + = (\lambda x \ y. \ x \ \text{Suc} \ y) \]

In action:

\[
+ (\lambda f \ x. \ f \ (f \ (f \ x))) \ y = (\lambda f \ x. \ f \ (f \ (f \ x))) \ \text{Suc} \ y \\
= (\lambda x. \ \text{Suc} \ (\text{Suc} \ (\text{Suc} \ x))) \ y \\
= \text{Suc} \ (\text{Suc} \ (\text{Suc} \ y))
\]
The famous Y combinator:

\[ Y = (\lambda f. (\lambda x. f (x \ x)) (\lambda x. f (x \ x))) \]

The important property of \( Y \):

\[ Y \ f \ = \ (\lambda x. f (x \ x)) (\lambda x. f (x \ x)) \]
\[ = \ f ((\lambda x. f (x \ x)) (\lambda x. f (x \ x))) \]
\[ = \ f (Y \ f) \]
Using the Y Combinator

Imagine wanting to implement the Collatz function:

\[
\begin{align*}
c(0) &= 1 \\
c(1) &= 1 \\
c(n) &= \begin{cases} 
  c(n \div 2) & \text{if } n \text{ is even} \\
  c(3n + 1) & \text{if } n \text{ is odd}
\end{cases}
\end{align*}
\]

When programming without pattern-matching:

\[
c(n) = \begin{cases} 
  \text{if } n < 2 \text{ then } 1 \\
  \text{else if } \text{even}(n) \text{ then } c(n \div 2) \\
  \text{else } c(3n + 1)
\end{cases}
\]
Using the Y Combinator

Want:

\[
c(n) = \begin{cases} 
  1 & \text{if } n < 2 \\
  c(n \div 2) & \text{if } \text{even}(n) \\
  c(3n + 1) & \text{else}
\end{cases}
\]

Let

\[
c_0 = (\lambda n. \text{if } (\leq n 2) 1 \ (\text{if } (\text{even } n) (f \ (\div n 2)) \ (f \ (+ \ (\times 3 n) 1))))
\]

Then (assuming we have \(<\), \(\text{even}\), \(\div\) and \(\text{if}\)):

\[
c \overset{\text{def}}{=} Y c_0 \\
= c_0 \ (Y \ c_0) \\
= c_0 \ c \\
= (\lambda n. \text{if } (\leq n 2) 1 \ (\text{if } (\text{even } n) (c \ (\div n 2)) \ (c \ (+ \ (\times 3 n) 1))))
\]
How Do We Implement Booleans?

Let

\[ \bot = (\lambda xy. y) \]
\[ \top = (\lambda xy. x) \]
\[ \text{if} = (\lambda bte. b t e) \]

Then

\[ \text{if} \top t e = \top t e \]
\[ = (\lambda y. t) e \]
\[ = t \]
\[ \text{if} \bot t e = \bot t e \]
\[ = (\lambda y. y) e \]
\[ = e \]
Booleans and Numbers

Implement the “is zero” test:

\[
isZero = (\lambda n. n (K \perp) T)
\]

Behaviour:

\[
isZero 0 = (\lambda n. n (K \perp) T) (\lambda f x. x) \\
= (\lambda f x. x) (K \perp) T \\
= T
\]

\[
isZero 3 = (\lambda n. n (K \perp) T) (\lambda f x. f (f (f x))) \\
= (\lambda f x. f (f (f x))) (K \perp) T \\
= K \perp (K \perp (K \perp T)) \\
= \perp
\]
How Do We Implement Pairs?

Let

\[
\begin{align*}
\text{Pair} & \quad = \quad (\lambda x \ y \ f. \ f \ x \ y) \\
\text{fst} & \quad = \quad (\lambda p. \ p \ (\lambda x \ y. \ x)) \\
\text{snd} & \quad = \quad (\lambda p. \ p \ (\lambda x \ y. \ y))
\end{align*}
\]

Then

\[
\begin{align*}
\text{fst} \ (\text{Pair} \ M \ N) & \quad = \quad \text{fst} \ (\lambda f. \ f \ M \ N) \\
& \quad = \quad (\lambda f. \ f \ M \ N) \ (\lambda x \ y. \ x) \\
& \quad = \quad (\lambda x \ y. \ x) \ M \ N \\
& \quad = \quad (\lambda y. \ M) \ N \\
& \quad = \quad M
\end{align*}
\]
How Do We Implement Lists?

Let

\[
\begin{align*}
\text{nil} & \quad = \quad (\lambda c \, n. \, n) \\
\text{cons} & \quad = \quad (\lambda h \, t \, c \, n. \, h \, (t \, c \, n))
\end{align*}
\]

Thus:

\[
\begin{align*}
\text{[2]} & \quad = \quad \text{cons} \, 2 \, \text{nil} \\
& \quad = \quad (\lambda c \, n. \, c \, 2 \, (\text{nil} \, c \, n)) \\
& \quad = \quad (\lambda c \, n. \, c \, 2 \, n) \\
\text{[3, 2]} & \quad = \quad \text{cons} \, 3 \, \text{[2]} \\
& \quad = \quad (\lambda c \, n. \, c \, 3 \, (\text{[2]} \, c \, n)) \\
& \quad = \quad (\lambda c \, n. \, c \, 3 \, (c \, 2 \, n))
\end{align*}
\]
More Lists

Let

\[ \text{Sum} = (\lambda \ell. \ell + 0) \]

Then

\[ \text{Sum} \ [3, 1, 2] = \text{Sum} (\lambda \mathrm{n}. \mathrm{c} \ 3 \ (\mathrm{c} \ 1 \ (\mathrm{c} \ 2 \ \mathrm{n}))) \]
\[ = (\lambda \mathrm{n}. \mathrm{c} \ 3 \ (\mathrm{c} \ 1 \ (\mathrm{c} \ 2 \ \mathrm{n}))) + 0 \]
\[ = (\lambda \mathrm{n}. + 3 \ (+ \ 1 \ (+ \ 2 \ \mathrm{n}))) \ 0 \]
\[ = + 3 \ (+ \ 1 \ (+ \ 2 \ 0)) \]
\[ = 6 \]
Summary

The λ-Calculus is “Turing-Complete”.

Church Numerals allow for direct encodings of primitive recursive functions such as addition and multiplication.

The Y combinator can encode unbounded search through arbitrary recursion.
  ▸ The λ-Calculus thus captures all recursive functions.

The λ-calculus can encode (in a standard way) algebraic data types such as lists and trees.
(Half-)Topic Summary

- Equational Logic
  - connection to $\rightarrow^*$
  - inconsistent assumptions
  - soundness & completeness
- Soundness via Church-Rosser
  - Diamond chasing and parallel reduction
- Standardisation
  - Normal order evaluation finds normal forms
- Computation
  - Church Numerals
  - Encoding Other Types
  - Encoding the Recursive Functions