CS351: The λ-calculus

James Power

16 October 2006

The lambda calculus

- Alonzo Church, 1936
- An alternative view of the 'meaning of computation'
- Is the core foundation for:
  - Theoretical computer science
  - Functional programming languages
  - Constructive logics
- Think of it this way: if you didn't have *any* programming language and had to build one, where would you start?

This lecture: overview

1. Syntax
2. Semantics (reduction and conversion)
3. Church Booleans
4. Church Numbers
5. A fixpoint operator

1. Syntax of the λ calculus

variable Any variable "v" is an expression

application Given any two expressions e_1 and e_2, then "$(e_1\ e_2)$" is a valid expression, and denotes the application of e_1 to e_2.

abstraction Given any variable v, and any expression e, then "$(\lambda v \cdot e)$" is an expression representing a function with v as the formal parameter, and e as the body of the function

Note that the application of f to x is written "functional style" as $(f\ x)$ and *not* the more common $f(x)$

2. Semantics of the λ calculus

To apply an expression of the form $\lambda x \cdot e_1$ to some other expression, say e_2, then we replace all occurrences of x in e_1 with e_2.

This process is known as β-reduction, and is symbolised by the "$\leadsto$" relation.

Formally, we write:

$$(\lambda x \cdot e_1)\ e_2 \quad \leadsto \quad e_1[x := e_2]$$

Here, the notation "$e_1[x := e_2]$" is used to denote the result of replacing all occurrences of x in e_1 with e_2

Other rules

While β-reduction is the main rule, other auxiliary concepts include some fairly obvious *conversions*

- α-conversion:

$$(\lambda x \cdot e_1) \quad \leadsto \quad (\lambda y \cdot e_1[x := y])$$

provided y does not appear free in e_1

- η-conversion:

$$(\lambda x \cdot e_1) \quad \leadsto \quad e_1$$

if x does not occur free in e_1

Reduction strategies

Suppose we were given the following expression to evaluate:

$$(\lambda x \cdot y)\ ((\lambda z \cdot z)\ u)$$

We have two choices of reductions here:

- Strict (or *eager*): First reduce the argument, and then apply the function
- Lazy: (or *non-strict*): First apply the function, and then reduce the function body

Redex and normal form

If we have an expression containing some sub-expression of the form $(\lambda x \cdot e_1)\ e_2$ then clearly this is a candidate for reduction. Such an expression is called a *reducible expression* or simply a redex. The difference between strict and lazy evaluation then is one of choice between different possible redexes.

An evaluation can be said to have completed when there are no more reductions possible; that is, when we have reduced to an expression which contains no more redexes.

Such expressions are important, and have a special name:

- An expression is said to be in normal form if it contains no redexes

Normal forms and Termination

- Not all λ expressions *have* a normal form; try reducing:

$$(\lambda x \cdot x\ x)\ (\lambda x \cdot x\ x)$$

- The evaluation strategy *can* matter; try reducing the following using strict and lazy evaluation:

$$\lambda y \cdot z\ ((\lambda x \cdot x\ x)\ (\lambda x \cdot x\ x))$$

- The *Halting Problem* tells us that there is no general procedure for deciding if a λ expression has a normal form.

The Church-Rosser Theorem

The **Church-Rosser Theorem** states that for any lambda expressions e, f and g,

- if $e \leadsto^* f$ and $e \leadsto^* g$
- then there exists some h such that $f \leadsto^* h$ and $g \leadsto^* h$

This is also known as the *diamond property* or, in a more general context, *confluence*.

Corollary: If an expression in the λ-calculus has a normal form, then it has at most *one* normal form.

Encoding "Data Types"

- So far the lambda calculus doesn't look vary powerful (or much like a real programming language
- However, it does have the power to express any computable function
- As an example of its power, we will show how the Booleans and natural numbers exist within the calculus. As a spin-off, this will also give us an if-then-else construct, and *primitive recursion*.
- Finally, we will derive a general scheme of recursion using *fixpoints*, which captures the full power of computational recursion (also called μ-recursion).

3. Church Booleans

Wanted: two expressions that are different, but have the same "pattern".

$$\begin{array}{lcl} \text{TRUE} & \doteq & \lambda x \cdot \lambda y \cdot x \\ \text{FALSE} & \doteq & \lambda x \cdot \lambda y \cdot y \end{array}$$

- Both expressions are closed
- These are in fact the smallest closed expressions, exhibiting some common structure, that are also definitely different.

Aside: the smallest closed expression in the λ-calculus is the identity function $(\lambda x \cdot x)$ which is basically a kind of "no-op".

Boolean functions

The Boolean values are actually their own canonical if-then-else operation. For convenience, we can make this explicit:

$$\text{COND} \doteq \lambda b \cdot \lambda x \cdot \lambda y \cdot b\ x\ y$$

We can then define the usual Boolean operations:

$$\begin{array}{lcl} \text{AND} & \doteq & \lambda a \cdot \lambda b \cdot \text{COND}\ a\ b\ \text{FALSE} \\ \text{OR} & \doteq & \lambda a \cdot \lambda b \cdot \text{COND}\ a\ \text{TRUE}\ b \\ \text{NOT} & \doteq & \lambda a \cdot \text{COND}\ a\ \text{FALSE TRUE} \end{array}$$

4. Church Numbers

A little more difficult than Booleans, since we need an infinite set of expressions that have the same basic pattern.

$$\begin{array}{lcl} C_0 & \doteq & \lambda f \cdot \lambda x \cdot x \\ C_1 & \doteq & \lambda f \cdot \lambda x \cdot f\ x \\ C_2 & \doteq & \lambda f \cdot \lambda x \cdot f\ (f\ x) \\ C_3 & \doteq & \lambda f \cdot \lambda x \cdot f\ (f\ (f\ x) \\ \vdots & & \\ C_n & \doteq & \lambda f \cdot \lambda x \cdot f^n\ x \end{array}$$

Basically, for any Church number C_k, the expression $(C_k\ g\ y)$ means "apply the function g exactly k times to y".

Numeric functions

As with Booleans, the Church numerals are their own (canonical) operator:

$$\text{ITER} \doteq \lambda n \cdot \lambda f \cdot \lambda x \cdot n\ f\ x$$

This is essentially a schema for primitive recursion, and allows us to define:

$$\begin{array}{rcl} \text{IS-EVEN} & \doteq & \lambda n \cdot \text{ITER}\ n\ \text{NOT TRUE} \\ \text{IS-ZERO} & \doteq & \lambda n \cdot \text{ITER}\ n\ (\lambda x \cdot \text{FALSE})\ \text{TRUE} \\ & & \\ \text{SUCC} & \doteq & \lambda n \cdot (\lambda f \cdot \lambda x \cdot f\ (n\ f\ x)) \\ \text{ADD} & \doteq & \lambda m \cdot \lambda n \cdot \text{ITER}\ n\ \text{SUCC}\ m \\ \text{MULT} & \doteq & \lambda m \cdot \lambda n \cdot \text{ITER}\ n\ (\text{ADD}\ m)\ C_0 \\ \text{POWER-OF} & \doteq & \lambda m \cdot \lambda n \cdot \text{ITER}\ n\ (\text{MULT}\ m)\ C_1 \end{array}$$

5. Fixpoints and recursion

The definition of a the *fixpoint* of a function is a standard concept from maths:

- For any function f and argument x, we say that x is a **fixpoint** of f if:

$$(f\ x) = x$$

A function may have no fixpoints, one unique fixpoint or many fixpoints.

Suppose we had a fixpoint operator, fix, that somehow worked out the fixpoint of a function. Then:

$$f\ (fix\ f) \quad = \quad (fix\ f)$$

Church's fixpoint operator

In the untyped lambda calculus this kind of equality is best represented by *reduction*, so we will seek to define an operator FIX with the property that:

$$(\text{FIX } f) \quad \rightsquigarrow^* \quad f\ (\text{FIX } f)$$

There are a number of fixpoint operators that can be defined; we will use the following (called "Church's fixpoint operator"):

$$\mathsf{Y} \quad \doteq \quad \lambda t\ \cdot\ (\lambda z\ \cdot\ t\ (z\ z))\ (\lambda z\ \cdot\ t\ (z\ z))$$

To see that this is indeed a fixpoint operator, assume we have some function f, and try reducing $(\mathsf{Y}\ f)$

Using fixpoints: example

The fixpoint operator can be used to define *any* recursive function. For example, without it, we might try to define* factorial as:

$$\text{FACT} = \lambda n\ \cdot\ \text{COND}\ (\text{IS-ZERO}\ n) \quad C_1 \quad (\text{MULT}\ n\ (\text{FACT}\ (\text{PRED}\ n)))$$

This is incorrect, since the definition is itself recursive. We use the fixpoint operator to remove that recursion:

$$\begin{array}{lcl} \text{FACT}' & \doteq & (\lambda \text{F}\ \cdot\ \lambda n\ \cdot\ \text{COND}\ (\text{IS-ZERO}\ n) \quad C_1 \quad (\text{MULT}\ n\ (\text{F}\ (\text{PRED}\ n)))) \\ \text{FACT} & \doteq & \mathsf{Y}\ \text{FACT}' \end{array}$$

**Assumes a suitable definition of the predecessor function* PRED

Other fixpoint operators

This is not the only fixpoint operator - there are many more.

One other famous one is Turing's fixpoint operator:

$$\mathsf{Y}_T \quad \doteq \quad (\lambda t\ \cdot\ \lambda z\ \cdot\ z(t\ t\ z)) \quad (\lambda t\ \cdot\ \lambda z\ \cdot\ x(t\ t\ z))$$

Exercise: Prove that this *is* a fixpoint operator, i.e. that for any f

$$(\mathsf{Y}_T f) \quad \rightsquigarrow^* \quad f\ (\mathsf{Y}_T\ f)$$

Where next?

- An alternative approach, based on the **combinators** S, K and I is due to Moses Schönfinkel and Haskell Curry (both worked at Göttingen under Hilbert)
- **Functional programming** in: LISP, ML, Haskell, ...
- The **Curry-Howard isomorphism** notes the similarities between the λ-calculus and constructive logic
- Higher-order logics and λ-calculi form the basis for **type theory**. Systems include *System F*, Martin-Löf type theory, the Calculus of Constructions, ...

References

- *Introduction to Lambda Calculus*, Henk Barendregt, Erik Barendsen, Technical report (Nijmegen), 1991.
  `http://citeseer.ist.psu.edu/barendregt94introduction.html`

- *Type Theory and Functional Programming*. Simon Thompson. Addison-Wesley, 1991.
  `http://www.cs.kent.ac.uk/people/staff/sjt/TTFP/`

- *Proofs and Types*, J-Y Girard, Y. Lafont and P. Taylor, Cambridge, 1989.
  `http://www.cs.man.ac.uk/~pt/stable/Proofs+Types.html`

- Wikipedia: `http://en.wikipedia.org/wiki/Lambda\_calculus`