Lambda Calculus

A4M36TPJ, 2015/2016

Computable Functions

- **Computable functions** are the basic objects of study in computability theory.
- **Computable functions** are the formalized analogue of the intuitive notion of algorithm.
- Every **computable function** has a **finite procedure** giving explicit, unambiguous instructions on how to compute it.
- This procedure has to be encoded in the finite alphabet used by the computational model, so there are only **countably many computable functions**. For example, functions may be encoded using a string of bits (the alphabet $\Sigma = \{0, 1\}$).

Uncomputable

- The real numbers are uncountable so most **real numbers are not computable**.
- Every computable number is **definable**, but not vice versa. There are many definable, noncomputable real numbers.
- Example: Binary representation of halting problem

Procedure for Computing a Computable Function

- There must be **exact instructions** (i.e. a program), **finite in length**, for the procedure.

- If the procedure is given a **k-tuple x** in the domain of f, then after **a finite number** of discrete steps the procedure must terminate and produce f(**x**).

- If the procedure is given a **k-tuple x** which is not in the domain of f, then the procedure might go on forever, never halting. Or it might get stuck at some point, but it **must not** pretend to produce a value for f at **x**.

Lambda Calculus

- Developed to study effectively **computable functions**.
- Introduced in 1930 by Alonzo Church.
- **Smallest universal programming language**
- Any **computable function** can be expressed and evaluated using **Lambda Calculus** => Equivalent to **Turing Machines**.
- Became a strong theoretical foundation for the family of **functional programming languages**.

Expressions

<expression> := <name> | <function> | <application>

<function> := λ <name>.<expression>

<application> := <expression><expression>

Evaluation

- Expression can be surrounded with **parenthesis** for clarity.
  - If E is an expression, (E) is the same expression.
- Function application associates from the left.
  - $E_1E_2E_3E_4\ldots E_n$ is evaluated as $(\ldots(((E_1E_2)E_3)E_4)\ldots E_n)$

Lambda Expression

- Lambda expression is an anonymous function definition.

$$\lambda x.x$$

Application

- Functions can be applied to other expression. Here is an example application:

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

- The identity function is applied to *y*.

- To apply the function we do the following substitution:

$$(\lambda x.x)y = [y/x]x = y$$

- [y/x] means that all occurrences of *x* are substituted by *y* in the expression to the right.

Lambda Expression Arguments

- The name of the arguments in function definitions do not carry any meaning in themselves.

- They are just “placeholders”. Therefore:

$$\lambda x.x \equiv \lambda y.y \equiv \lambda z.z \equiv \lambda t.t$$

- $A \equiv B$ means that A is a synonym of B

Free and Bound Variables

- In the function λx.x we say that x is “**bound**” since its occurrence in the body is preceded by λx.
- A name not preceded by λ is called “**free**”.

(λx.xy)

(λx.x)(λy.yx)

Free Variables

- **Variable is free** in an expression if **one of the following three cases holds**:

  - <name> is free in <name>.

  - <name> is free in λ<name$_1$>.<exp> if the identifier <name>≠<name$_1$> and <name> is free in <exp>.

  - <name> is free in E_1E_2 if <name> is free in E_1 or if it is free in E_2.

Bound Variables

- A variable <name> is **bound** if **one of two cases holds**:
  - <name> is bound in λ<name$_1$>.<exp> if the identifier <name> = <name$_1$> or if <name> is bound in <exp>.
  - <name> is bound in E_1E_2 if <name> is bound in E_1 or if it is bound in E_2.

Free and Bound Variables

- The same identifier can occur free and bound in the same expression:

$$(\lambda x.xz)(\lambda z.z)$$

Substitutions

- In λ-calculus we **do not give names to functions**.
- To simplify the notation we will use **capital letters, digits and other symbols as synonyms** for some function definitions.
- For example I is a synonym for $(\lambda x.x)$

$$II \equiv (\lambda x.x)(\lambda x.x)$$

$$II \equiv (\lambda x.x)(\lambda z.z) \equiv I$$

Substitutions

- Avoid mixing up free occurrences of an identifier with bound ones.

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

- Incorrect result is:

$$(\lambda y.yy)$$

- Why?

Substitutions

- If the function *λx.exp* is applied to *E*, we substitute **all free occurrences** of *x* in *exp* with *E*.
- We **rename the bound variable** of the same name in *exp* before substitution.
- Variable names are only “placeholders” in λ-calculus, they are not important.

Substitutions

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

In this expression we associate the argument *x* with *y*. In the body:

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

only **first *x* is free** and can be substituted. Before substituting we **rename the variable *y* to avoid mixing** its free and its bound occurrences.

$[y/x](\lambda t.(x(\lambda x.xt))) = (\lambda t.(y(\lambda x.xt)))$

Functions with Multiple Arguments

- How to represent functions with multiple arguments in λ-calculus?
  - We do not need them.
  - We can use currying to transform functions
  - Given a function *f* of type: $(X \times Y) \longrightarrow Z$, currying it makes a function curry*(f)*: $X \longrightarrow Y \longrightarrow Z$.

Arithmetic

- We expect from a programming language that it should be capable of doing arithmetical calculations.
- Numbers in λ-calculus can be represented as in **Peano axioms** starting from **zero**.
- suc(zero) to represent “1”, suc(suc(zero)) to represent “2” and so on.

Arithmetic

- Zero can be represented as $(\lambda s.(\lambda z.z))$ or abbreviated $(\lambda sz.z)$.
- Then we can define:

$$1 \equiv \lambda sz.s(z)$$

$$2 \equiv \lambda sz.s(s(z))$$

$$3 \equiv \lambda sz.s(s(s(z)))$$

…

Successor Function

- The function applied to “0” returns “1”, applied to “1” returns “2” and so on.

$$S \equiv \lambda wyx.y(wyx)$$

- This function applied to our representation of zero yields:

$$S0 \equiv (\lambda wyx.y(wyx))(\lambda sz.z)$$

$$\lambda yx.y((\lambda sz.z)yx) = \lambda yx.y((\lambda z.z)x) = \lambda yx.y(x) \equiv 1$$

$$S1 \equiv (\lambda wyx.y(wyx))(\lambda sz.s(z)) = \lambda yx.y((\lambda sz.s(z))yx) = \lambda yx.y(y(x)) \equiv 2$$

Addition

- Addition can be obtained immediately by noting that the body sz of our definition of the number 1.
- If we want to add say 2 and 3, we just apply the successor function two times to 3.

$$2S3 \equiv (\lambda sz.s(sz))(\lambda wyx.y(wyx))(\lambda uv.u(u(uv)))$$
$$(\lambda wyx.y((wy)x))((\lambda wyx.y((wy)x))(\lambda uv.u(u(uv)))) \equiv SS3$$

Multiplication

$(\lambda xyz.x(yz))$

$(\lambda xyz.x(yz))22$

Conditionals

- We define two functions True:

$$T \equiv \lambda xy.x$$

- and False:

$$F \equiv \lambda xy.y$$

Logical Operations

- The **AND** function of two arguments can be defined as

$$\wedge \equiv \lambda xy.xy(\lambda uv.v) \equiv \lambda xy.xyF$$

- The **OR** function of two arguments can be defined as

$$\vee \equiv \lambda xy.x(\lambda uv.u)y \equiv \lambda xy.xTy$$

- **Negation** of one argument can be defined as

$$\neg \equiv \lambda x.x(\lambda uv.v)(\lambda ab.a) \equiv \lambda x.xFT$$

- The **negation** function **applied to "true"** is

$$\neg T \equiv \lambda x.x(\lambda uv.v)(\lambda ab.a)(\lambda cd.c)$$

- which reduces to

$$TFT \equiv (\lambda cd.c)(\lambda uv.v)(\lambda ab.a) = (\lambda uv.v) \equiv F$$

Conditional Test

- It is very convenient in a programming language to have a function which is **true** if a number is zero and **false** otherwise.

$$Z \equiv \lambda x.xF\neg F$$

- To understand how this function works, note that

$$0\ f\ a \equiv (\lambda sz.z)fa = a$$

- that is, the function f applied zero times to the argument a yields a. On the other hand, F applied to any argument yields the identity function

$$F\ a \equiv (\lambda xy.y)a = \lambda y.y \equiv I$$

Conditional Test

- We can now test if the function Z works correctly. The function applied to zero yields

$$Z0 \equiv (\lambda x.xF\neg F)0 = 0F\neg F = \neg F = T$$

- because F applied 0 times to ¬ yields ¬. The function Z applied to any other number N yields

$$ZN \equiv (\lambda x.xF\neg F)N = NF\neg F$$

- The function F is then applied N times to ¬. But F applied to anything is the identity, so that the above expression reduces for any number N greater than zero to

$$IF = F$$

Recursion

- Recursive functions can be defined in the λ calculus using a function which calls a function y and then regenerates itself. This can be better understood by considering the following function Y:

$$Y \equiv (\lambda y.(\lambda x.y(xx))(\lambda x.y(xx)))$$

- This function applied to a function R yields:

$$YR = (\lambda x.R(xx))(\lambda x.R(xx))$$

- which further reduced yields:

$$R((\lambda x.R(xx))(\lambda x.R(xx))))$$

- but this means that YR = R(YR), that is, the function R is evaluated using the recursive call YR as the first argument.