Lambda Calculus and Combinators

Lecture 5

Mathematical foundations

- Functional languages have underlying mathematics that is particularly simple and beautiful
- Two motivations (as a programmer) to learn this:
  1. Deeper insight into the language itself
  2. More complete understanding when you read points other people make that are expressed in mathematical notation

What is Lambda Calculus?

- It was invented by Alonzo Church in the 1930s to formalise mathematical computation
- It is
  1. A notation to define functional application in an abstract mathematical way
  2. **Equivalence** and **reduction** rules to determine when two Lambda expressions have the same effect (or have, as functions, the same value)
- Use of + operator here is a shorthand not allowed in **pure** lambda calculus where everything is a function.
  - See later for how to represent + using functions

Lambda expressions

$$\lambda x.x + 1$$

$$\lambda y.y + 1$$

Pure lambda expression

$$\lambda f.\lambda x.\lambda p.f(x\,f\,p)$$

F# equivalents

```
fun x -> x + 1
```

```
fun y -> y + 1
```

```
fun f -> fun x -> fun p -> f (x f p)
```

Formal definition

The Lambda Calculus Λ is the set of terms $t \in \Lambda$ defined as:

1. $v \in \Lambda$ for v a variable $v_1, v_2, v_3, ...$
2. $\lambda v.T \in \Lambda$ for v a variable $v_1, v_2, v_3, ...$ and $T \in \Lambda$ is a *lambda abstraction* with bound variable v. All v which are *free variables* in T are *bound* by λv.
3. $(T1\ T2) \in \Lambda$ for $T1, T2 \in \Lambda$ represents $T1$ applied to $T2$

For convenience:

- a sequence $T1\ T2\ T3$ without brackets is left associated: $(T1\ T2)\ T3$
- The body of a lambda abstraction is extended right as far as possible: $\lambda x.x\ y = \lambda x.(x\ y)$ not $(\lambda x.x)\ y$.
- $\lambda v_1.\lambda v_2.$ can be abbreviated $\lambda v_1 v_2.$

Example

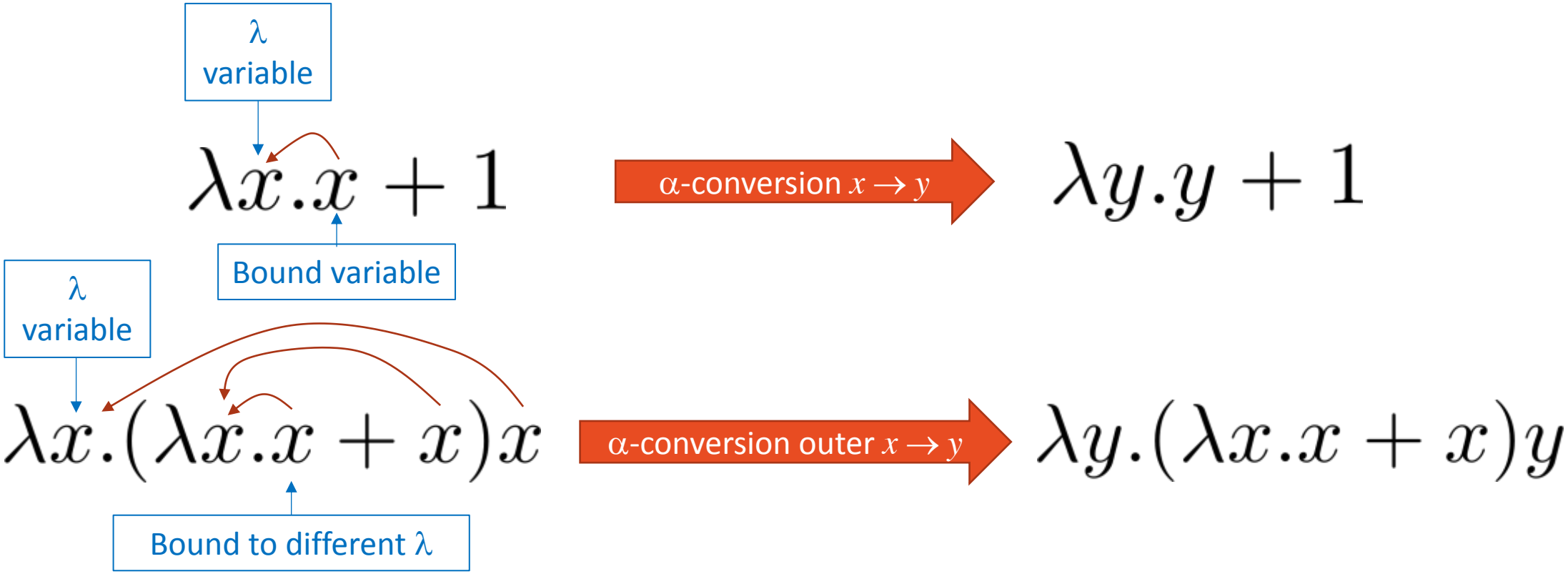
$$\lambda x.\lambda y.(\lambda x.z\ x\ y)\ \lambda y.x$$

Terminology

- v is a *free variable* of $T \in \Lambda$ if there is no $\lambda v.$ in T that binds v.

Alpha-conversion (renaming)

- Changing a λ variable *and all the variables it binds* to a new variable results in an equivalent lamba expression.



Beta-reduction (evaluation)

- A function $\lambda x.E$ applied to a parameter F is equivalent to E with all outermost-bound instances of x in E substituted by F.

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

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

- Inner (bound) instances of x are not substituted in β-reduction of the outer λx

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

Eta-conversion (function equality)

- Captures the intuition that two functions are equal if they always give the same results when a parameter is applied
- If f does not contain free (unbound) instances of x, β-reduction proves that for any y:

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

- Eta-conversion therefore states that if f does not contain free (unbound) instances of x

$$\lambda x.f\ x \rightarrow f$$

Numbers in pure Lambda Calculus

- Church numerals are high order functions that when given a function f return with its nth power f^n

$$n \equiv \lambda f.\lambda x.f^n\ x$$

$$0 \equiv \lambda f.\lambda x.x$$

$$1 \equiv \lambda f.\lambda x.f\ x$$

$$2 \equiv \lambda f.\lambda x.f\ (f\ x)$$

- The SUCC function - adding 1 - is easy to define:

$$\begin{aligned} \text{SUCC}\ n &= n+1 \Rightarrow \\ \text{SUCC}\ n\ f\ x &= f\ (n\ f\ x) \Rightarrow \\ \text{SUCC} &= \lambda n.\lambda f.\lambda x.f\ (n\ f\ x) \end{aligned}$$

$a + b$

- Using Church numerals + can be represented as a function PLUS in pure lambda calculus:

$$a + b \equiv \text{PLUS } a \ b$$

$$\text{PLUS } a \ b \ f \ x = f^{a+b} \ x = f^a (f^b \ x) = a \ f \ (b \ f \ x)$$

$$\text{PLUS} \equiv \lambda a.\lambda b.\lambda f.\lambda x.a \ f \ (b \ f \ x)$$

Recursion in Lambda Calculus

- Split any recursive function into a non-recursive "step" and a higher order **fixed point** function that implements the recursion.
- Define a **fixed point function** Y such that:

  `length` = Y `lengthStep`

- More generally:

$$Y\ f = f\ (Y\ f) = f\ (f\ (Y f))...$$
$$Y = \lambda f.f\ (Y\ f)$$

Y can be used to make **any** recursive function by choosing a suitable step function f.

```
let lengthStep f x =
    match x with
    | [] -> 0
    | h :: t -> 1 + f t

let rec length x =
    lengthStep length x

// let rec length = lengthStep length ?

let rec Y f x = f (Y f) x

let length1 x = Y lengthStep x
```

Types and lambda calculus

- F# and most other functional languages have a type system
- Lambda calculus does not. This makes it more expressive.
- Typed lambda calculi can be defined by adding a *type system* to the base calculus. Every term in the typed calculus has an associated *type*. You will often find new type systems defined this way.
- For example Traytel 2011 – an extension of Hindley Milner typing that allows convenient structural subtyping.

Combinators

- We have seen that functional languages are equivalent to the lambda calculus
- Haskell Curry pioneered the use of a transformed calculus equivalent to lambda calculus, but simpler, called the combinatory calculus
- This leads to a very neat way of implementing functional languages using just three constant functions I, K, S (and Y from the previous slide)
- The bracket abstraction algorithm (next slide) transforms any lambda calculus expression into an equivalent combinatory expression

Identity: $\lambda x.x \equiv I \Rightarrow I\ x = x$

Constant: $\lambda x.\lambda y.x \equiv K \Rightarrow K\ x\ y = x$

Split: $\lambda f.\lambda g.\lambda x.f\ x\ (g\ x) \equiv S \Rightarrow S\ f\ g\ x = f\ x\ (g\ x)$

Bracket abstraction (uses S, K and I)

- Define bracket abstraction $[x]\ E$ as the process of removing a single free variable x from an expression of free variables E :

$$([x]\ E)\ x = E$$

- Bracket abstraction is the equivalent of lambda abstraction since:

$$([x]\ E)\ x = (\lambda x.E)\ x$$

$$I\ x = x \qquad [x]\ x = I$$

$$K\ y\ x = y \qquad [x]\ E_c = K\ E_c \quad (E_c \text{ does not contain } x)$$

$$S\ f\ g\ x = (f\ x)(g\ x) \qquad [x]\ (E_1\ E_2) = S\ ([x]\ E_1)\ ([x]\ E_2)$$

A Real Example of bracket abstraction

- To write a useful function we need:
  - Arithmetic: `minus a b: int -> int -> int`
  - Comparison operator: `greaterThan a b: int -> int -> bool`
- Integers
  - Use *impure* integer literals and arithmetic (Church numerals are less efficient)
- Booleans
  - Use combinators for TRUE and FALSE!

$$\begin{aligned}&\text{TRUE}\ f\ g = f\\&\text{FALSE}\ f\ g = g\\&\text{ifThenElse}\ b\ f\ g = \text{if}\ b\ \text{then}\ f\ \text{else}\ g \equiv b\ f\ g\end{aligned}$$

- `abs` function

$$\text{abs}\ x = \text{ifThenElse}\ (\text{greaterThan}\ 0\ x)\ (\text{minus}\ 0\ x)\ x \equiv (\text{greaterThan}\ 0\ x)\ (\text{minus}\ 0\ x)\ x$$

- What is $[x]\ (\text{abs}\ x)$?

Example continued...

$$\text{abs} =$$

$$[x]\ (\text{greaterThan}\ 0\ x)\ (\text{minus}\ 0\ x)\ x =$$

$$S\ ([x]\ (\text{greaterThan}\ 0\ x)\ (\text{minus}\ 0\ x))\ ([x]\ \ x) =$$

$$S\ (([x]\ \text{greaterThan}\ 0\ x)\ ([x]\ \text{minus}\ 0\ x))\ ([x]\ \ x) =$$

$$S\ (S\ (\text{greaterThan}\ 0)\ (\text{minus}\ 0))\ I$$

Combinator reduction

- Combinators with enough parameters can be rewritten using the definition
  - e.g. S a b c → (a c) (b c)
- Compute: `abs -3`
  - Use **normal reduction order** reduce the leftmost outermost combinatory expression
  - Normal reduction ensures no unnecessary work is done
    - Conditionals are evaluated before **then** and **else** parts

$$\text{abs } -3 =$$

$$S\ (S\ (\text{greaterThan } 0)\ (\text{minus } 0))\ I\ -3 \rightarrow$$

$$S\ (\text{greaterThan } 0)\ (\text{minus } 0)\ -3\ (I\ -3) \rightarrow$$

$$\text{greaterThan } 0\ -3\ (\text{minus } 0\ -3)\ (I\ -3) \rightarrow$$

$$\text{TRUE}\ (\text{minus } 0\ -3)\ (I\ -3) \rightarrow$$

$$\text{minus } 0\ -3 \rightarrow$$

$$3$$

Reflections

- Anonymous functions (fun x -> ...) are called **lambda**'s in many programming languages in honour of Lambda calculus
- Lambda calculus is a very simple universal way to express pure functional programs and hence **any computable function**
  - Positive integers can be represented as pure high order functions using Church numerals. In practical systems, for efficiency, integers are represented by machine literals and binary operations making an impure lambda calculus
  - Booleans can be represented as pure high order functions that directly select then or else part of expression and naturally implement *if then else*.
- Lambda expressions can be equivalently represented using combinators
- Combinator reduction can be used to evaluate expressions
  - Normal reduction order minimises work done
  - See demo code for complete functional language implementation using combinators!