5. The Untyped Lambda-Calculus (Types and Programming Languages)

Kwanghoon Choi

Software Languages and Systems Laboratory
Chonnam National University

Week 5

Overview

This chapter introduces the *untyped or pure lambda calculus* (Alonzo Church 1936, 1941).

Every complex programming language can be understood by formulating it as

- a tiny core calculus capturing the language's essential mechanisms, together with
- a collection of convenient *derived forms* whose behavior is understood by translating them into the core.

(Peter Landin 1964, 1965, 1966; John McCarthy 1959, 1981)

The lambda calculus has seen widespread use in the design and implementation of PLs and in the study of type systems.

The core calculus in Ch.5, its type system in Ch.9, and the derived forms in Ch.11, Ch.13, Ch.14.

Overview (Cont.)

The lambda calculus can be enriched in a variety of ways.

First, it is often convenient to add special concrete syntax for features like numbers, tuples, records, etc., whose behavior can already be simulated in the core language (Ch.11).

More interestingly, we can add more complex features such as mutable reference cells (Ch.13) or nonlocal exception handling (Ch.14), which can be modeled in the core language only by using rather heavy translations.

Such extensions lead eventually to languages such as ML, Haskell, or Scheme.

The extensions to the core language often involve extensions to the type system as well.

5.1 Basics

Procedural (or functional) abstraction: Instead of writing the same calculation over and over, we write a procedure or function that performs the calculation generically.

(5*4*3*2*1) + (7*6*5*4*3*2*1) - (3*2*1)
vs. factorial(5) + factorial(7) - factorial(3)
    where factorial(n) = if n=0 then 1 else n*factorial(n-1)

Notation:
"λn. ..." as a shorthand for "the function that, for each n, yields ..."

factorial = λn. if n=0 then 1 else n*factorial(n-1)

5.1 Basics

The lambda-calculus (or λ-calculus) embodies this kind of function definition and application in the purest possible form.

The syntax of the lambda calculus

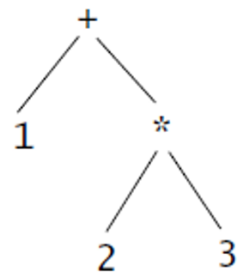
$$t \ ::= \ x \quad | \quad \lambda x.t \quad | \quad t \ t$$

where the three sorts of terms are called variable, abstraction, and application, respectively.

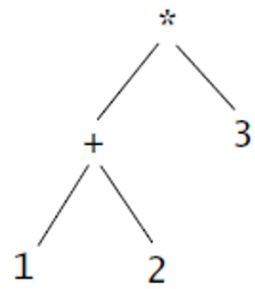
5.1 Basics: abstract syntax tree

Note: “s t u” is read as “(s t) u”, which is different from “s (t u)”.

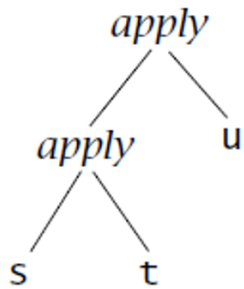
1+2*3



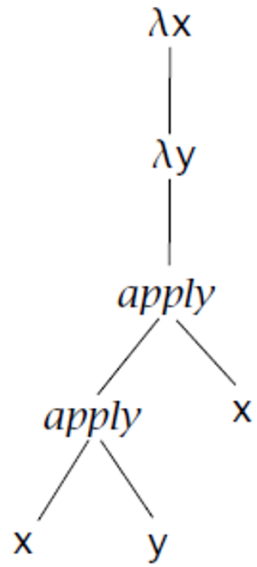
(1+2)*3



s t u



λx. λy. x y x



5.1 Basics: variables vs. meta variables

The metavariables t (as well as s and u) to stand for an arbitrary term while the lambda calculus variable x (as well as y and z) to stand for an arbitrary variable.

In a sentence like "the term λx. λy. x y has the form λz.s where z = x and s = λy. x y,"

- z and s are metavariables, whereas
- x and y are (lambda calculus) variables.

5.1 Basics: scope

The variable x is *bound* when it occurs in the body of t of an abstraction λx.t. Otherwise it is *free*.

- x is bound in λx. x
- x is bound in λz. λx. λy. x (y z)
- x is free in x y
- x is free in λy. x y
- the first occurrence of x is bound and the second is free in (λx. x) x

cf. λx is called a *binder*.

A term with no free variables is said to be *closed*. Closed terms are called *combinators*.

- For example, "λx. x" is an (identity) combinator.

5.1 Basics: operational semantics

Substitution $[x \mapsto u]\ t$: to replace all occurrences of x in t by u

A single step evaluation $(\rightarrow)$: $\boxed{(\lambda x.\ t)\ u \quad \rightarrow \quad [x \mapsto u]t}$

cf. It is also called a reduction (or β-reduction).

- $(\lambda x.\ x)\ y \quad \rightarrow \quad y$
- $(\lambda x.\ x\ (\lambda x.\ x))\ \ (u\ r) \quad \rightarrow \quad u\ r\ (\lambda x.\ x)$

Several different evaluation strategies for the lambda calculus

- full beta-reduction, normal order reduction,
- call-by-name, *call-by-value*

5.2 Programming in the Lambda-Calculus

A warm-up exercises to get familiar with the lambda calculus by developing a number of standard examples of programming

- Multiple arguments
- Church booleans
- Pairs
- Church numerals
- Enriching the calculus
- Recursion

5.2 Programming λ-calculus: Multiple Arguments

Multi-argument functions in a programming language,
"add = λ(x,y). x+y"

In the lambda calculus, add = λx. λy. x+y

$$
\begin{array}{ll}
 & \text{add } 1\ 2 \\
= & (\text{add } 1)\ 2 \\
= & ((\lambda x.\ \lambda y.\ x+y)\ 1)\ 2 \\
\rightarrow & (\lambda y.\ 1+y)\ 2 \\
\rightarrow & 1+2 \\
= & 3
\end{array}
$$

Higher-order functions: a function that takes or returns another function (e.g., add)

This transformation of multi-argument functions into higher-order functions is called *currying* in honor of Haskell B. Curry.

5.2 Programming λ-calculus: Church Booleans

Encoding boolean values and conditionals

The terms tru and fls can be viewed as *representing* the boolean values "true" and "false".

- tru $= \lambda$t. λf. t
- fls $= \lambda$t. λf. f

We can define test as:

- test $= \lambda$l. λm. λn. l m n with the property that
  (1) (test b v w) reduces to v when b is tru, and
  (2) it reduces to w when b is fls.

Q. Verify if test tru v w $\rightarrow \cdots \rightarrow$ v.

Q. Verify if test fls v w $\rightarrow \cdots \rightarrow$ w.

5.2 Programming λ-calculus: Church Booleans (Cont.)

We can also define boolean operators like logical conjunction as functions:

- and $= \lambda$b. λc. b c fls

Q. Verify if (and tru tru) becomes tru and the others (and tru fls, and fls tru, and fls fls) become fls).

Q. Define logical or and not functions.

5.2 Programming λ-calculus: Pairs

Making pairs, and selecting the first/the second member of a pair.

- pair v w
- (pair (pair a b) c)
- (pair (pair a b) (pair c d))
- fst (pair v w) becomes v.
- snd (pair v w) becomes w.

The functions pair, fst, and snd:

- pair = λf. λs. λb. b f s
- fst = λp. p tru
- snd = λp. p fls

Q. Show the last two properties on fst (pair v w) and snd (pair v w).

5.2 Programming λ-calculus: Church numerals

Each number n is represented by a function c_n that takes two arguments, s and z, and applies s, n times, to z.

- $c_0 = \lambda s.\ \lambda z.\ z$
- $c_1 = \lambda s.\ \lambda z.\ s\ z$
- $c_2 = \lambda s.\ \lambda z.\ s\ (s\ z)$
- $c_3 = \lambda s.\ \lambda z.\ s\ (s\ (s\ z))$
- $\ldots$
- $c_n = \lambda s.\ \lambda z.\ s^n\ z$

Let us define addone to add 1 to an arbitrary number as

- addone $= \lambda n.\ \lambda s.\ \lambda z.\ s\ (n\ s\ z)$
- For example, addone c_1
  $\rightarrow \lambda s.\ \lambda z.\ s\ (c_1\ s\ z)$
  $\rightarrow \lambda s.\ \lambda z.\ s\ ((\lambda z.\ s\ z)\ z)$
  $\rightarrow \lambda s.\ \lambda z.\ s\ (s\ z) = c_2$

5.2 Programming λ-calculus: Church numerals (Cont.)

The addition and multiplication of Church numerals by terms plus and times

- plus $= \lambda$m. λn. λs. λz. m s (n s z)
- times $= \lambda$m. λn. m (plus n) c_0

Q. Try to reduce the following terms:

- plus c_2 c_1
- times c_2 c_1

5.2 Programming λ-calculus: Church numerals (Cont.)

To test whether a Church numeral is zero, let us define a term iszro:

- iszro = λm. m (λx. fls) tru

Q. Compute the following terms using iszro:

- iszro c_1
- iszro (times c_0 c_2)

5.2 Programming λ-calculus: Enriching the Calculus

Although we have seen that booleans, numbers, and the operations on them can be encoded in the pure lambda-calculus, it is often convenient to include the primitive booleans and numbers.

The symbol λ to refer to the pure lambda-calculus

The symbol λNB for the enriched system with booleans and arithmetic expressions from Ch.3

5.2 Programming λ-calculus: Recursion

The *divergent* combinator cf. Combinators $\equiv$ closed terms

- omega = (λx. x x) (λx. x x)
- It only reduces to itself.

The omega combinator has a useful generalization called the *fixed-point combinator*[1]

- fix = λf. (λx. f (λy. x x y)) (λx. f (λy. x x y))

Use fix to define a recursive function,
factorial = λn. if n=0 then 1 else n * factorial (n-1):

[1] Also called as (call-by-value) Y-combinator. A simpler version is fix = λf. f (fix f), called as (call-by-name) Y-combinator.

5.2 Programming λ-calculus: Recursion (Cont.)

To define a recursive function,

factorial = λn. if n=0 then 1 else n * factorial (n-1):

First, make it *non-recursive* by replacing the recursive occurrence of factorial by an extra parameter fct as:

nonrec_factorial = λfct. λn. if n=0 then 1 else n * fct (n-1)

Then, apply fix to the non-recursive version to obtain the recursive behavior

factorial = fix nonrec_factorial

Q. Compute (factorial c_3).

5.3 Formalities: the syntax & operational semantics of λ-calculus

Syntax:

Term `t`	::=	`x`	variable
	|	λ`x.t`	abstraction
	|	`t t`	application
Value `v`	::=	λ`x.t`	

Evaluation:

$$\frac{\texttt{t1} \rightarrow \texttt{t1'}}{\texttt{t1 t2} \rightarrow \texttt{t1' t2}} \quad \text{(E-App1)}$$

$$\frac{\texttt{t2} \rightarrow \texttt{t2'}}{\texttt{v1 t2} \rightarrow \texttt{v1 t2'}} \quad \text{(E-App2)}$$

$$(\lambda\texttt{x. t})\ \texttt{v} \rightarrow [\texttt{x} \mapsto \texttt{v}]\ \texttt{t} \quad \text{(E-AppAbs)}$$

5.3 Formalities: Free variables, FV

FV(t) : the set of *free variables* of a term t

$$
\begin{aligned}
\texttt{FV(x)} &= \texttt{\{ x \}} \\
\texttt{FV(}\lambda\texttt{x. t1)} &= \texttt{FV(t1) \textbackslash\ \{ x \}} \\
\texttt{FV(t1 t2)} &= \texttt{FV(t1)} \cup \texttt{FV(t2)}
\end{aligned}
$$

Q. Show the process of FV(λz. λx. λy. x (y z)).

Q. Define the size function for the lambda terms.

Q. Show $|FV(t)| \leq size(t)$.

5.3 Formalities: Substitution

The substitution operation $[x \mapsto s]\ t$:

$$
\begin{array}{lcll}
[x \mapsto s]\ x & = & s & \\
[x \mapsto s]\ y & = & y & \text{if } y \neq x \\
[x \mapsto s]\ (\lambda y.\ t1) & = & \lambda y.\ [x \mapsto s]t1 & \text{if } y \neq x \text{ and } y \notin FV(s) \\
[x \mapsto s]\ (t1\ t2) & = & ([x \mapsto s]\ t1)\ ([x \mapsto s]\ t2) &
\end{array}
$$

Examples:

- $[x \mapsto (\lambda z.\ z\ w)]\ (\lambda y.\ x) = \lambda y.\ \lambda z.\ z\ w$
- $[x \mapsto y]\ (\lambda x.\ x) \neq \lambda x.\ y$ (It should be $\lambda x.\ x$)
- $[x \mapsto z]\ (\lambda z.\ x) \neq \lambda z.\ z$ (It should be $\lambda w.\ z$ after renaming z into w)

Alpha-conversion: renaming bound variables into new named ones
For example, $[x \mapsto y\ z](\lambda y.\ x\ y)$ is renamed as $[x \mapsto y\ z](\lambda w.\ x\ w)$