Course 2D1453, 2006-07

Advanced Formal Methods

Lecture 2: Lambda calculus

Mads Dam
KTH/CSC

Some material from B. Pierce: TAPL + some from G. Klein, NICTA

λ–calculus

- Alonzo Church, 1903-1995
  - Church-Turing thesis
  - First undecidability results
  - Invented λ-calculus in '30's
- λ-Calculus
  - Intended as foundation for mathematics
  - Discovered to be inconsistent, so interest faded (see later)
  - Foundational importance in programming languages
  - Lisp, McCarthy 1959
  - Programming languages and denotational semantics
    - Landin, Scott, Strachey 60's and 70's
  - Now an indispensable tool in logic, proof theory, semantics, type systems

Untyped λ-calculus - Basic Idea

- Turing complete model of computation
- Notation for abstract functions

λx. x + 5:

Name of function that takes one argument and adds 5 to it

I.e. a function f: x ↦ x + 5

But:

- Basic λ-calculus talks only about functions
- Not numbers or other data types
- They are not needed!

Function application

- λ-abstraction sufficient for introducing functions
- To *use* functions need to be able to
  - Refer to them:
    Use variables x, y, z
    For instance in λx. x – the identity function
  - Apply them to an argument:
    Application f x, same as f(x)
    (λx. x + 5) 4
- To compute with functions need to be able to evaluate them
  - (λx. x + 5) 4 evaluates to 4 + 5
- But language is untyped – (λx. x + 5) (λy. y) is also ok

Terms, Variables, Syntax

Assume a set of variables x, y, z

Term syntax in BNF:

t ::= x | λx. t | t t

Conventions:

- List bound variables
  λx y . t = λx. (λy. t)
- Application binds to left:
  x y z = (x y) z
- Abstraction binds to right:
  λx. x y = λx. (x y)

Example: λx y z. x z (y z)

β–reduction

The fundamental principle of computation in λ-calculus is replacement of formal by actual parameters

Example: (λx y. f (y x)) 5 (λx.x)

Substitution:

- t[s/x] is t with s substituted for variable x
- Must avoid variable capture

β–reduction: $\dfrac{-}{(\lambda x.\ t)\ s \to_\beta t[s/x]}$ $\quad \dfrac{s \to_\beta s'}{s\ t \to_\beta s'\ t}$

$\dfrac{t \to_\beta t'}{s\ t \to_\beta s\ t'}$ $\quad \dfrac{t \to_\beta t'}{\lambda x.\ t \to_\beta \lambda x.\ t'}$

Side Track: Evaluation Order

Redex: Term of the shape (λx. t) t'

As defined, β-reduction is highly nondeterministic

Not determined which redex to reduce next

Example:

(λx. x) ((λy x. (λy. y) x) (λz. z z))

$\to_\beta$ λy x. ((λy. y) x (λz. z z)

$\to_\beta$ (λx. x) (λx. (λy. y) x)

$\to_\beta$ (λx. x) ((λy x. x) (λz. z z))

Evaluation Order, II

Full β-reduction: → is $\to_\beta$

(λx. x) ((λy x. (λy. y) x) (λz. z z))
→ (λx. x) ((λy x. x) (λz. z z))
→ (λy x. x) (λz. z z)
→ (λx. x)

Normal order reduction:

Reduce leftmost, outermost redex first

(λx. x) ((λy x. (λy. y) x) (λz. z z))
→ (λy x. (λy. y) x) (λz. z z)
→ λx. (λy. y) x
→ λx. x

Evaluation Order, III

Call-by-name reduction:

Leftmost, outermost, but no reduction under λ

(λx. x) ((λy x. (λy. y) x) (λz. z z))
→ (λy x. (λy. y) x) (λz. z z)
→ λx. (λy. y) x

Call-by-need: Variant that uses *sharing* to avoid reevaluation (Haskell)

Call-by-value

Outermost, arguments must be value (= λ-abstraction)

(λx. x) ((λy x. (λy. y) x) (λz. z z))
→ (λx. x) ((λx. (λy. y) x))
→ (λx. (λy. y) x)

Programming in λ-Calculus

Can encode lots of stuff:

- Turing machines, functional programs
- Logic and set theory

Booleans: Define

tt == λx y. x
ff == λx y. y
if == λx y z. x y z
and == λx y. x y ff

Example: if tt t s, and tt t

Exercise 1: Define boolean "or" and "not" functions

Pairing

Define:

pair == λf s b. b f s
fst == λp. p tt
snd == λp. P ff

Example: Try fst(pair t s)

Church Numerals

Church numerals:

$\underline{0}$ == λ s z. z
$\underline{1}$ == λ s z. s z
$\underline{2}$ == λ s z. s (s z)
$\underline{3}$ == λ s z. s (s (s z))
...

That is, $\underline{n}$ is the function that applies its first argument n times to its second argument!

iszero == λn. n (λ x. ff) tt
succ == λn s z. s (n s z)
plus == λm n s z. m s (n s z)
times == λm n.m (plus n) $\underline{0}$

Church Numerals - Exercises

Exercise: Define exponentiation, i.e. a term for raising one Church numeral to the power of another.

Predecessor is a little tricky

zz == pair $\underline{0}$ $\underline{0}$

ss == λp. pair (snd p) (succ (snd p))

prd == λn. fst (n ss zz)

Exercise 2: Use prd to define a subtraction function

Exercise 3: Define a function equal that test two numbers for equality and returns a Church boolean.

Church Numerals, More Exercises

Exercise 4: Define "Church lists". A list [x,y,z] can be thought of as a function taking two arguments c (for cons) and n (for nil), and returns c x (c y (c z n)), similar to the Church numerals. Write a function nil representing the empty list, and a function cons that takes arguments h and (a function representing) a list tl, and returns the representation of the list with head h and tail tl. Write a function isnil that takes a list and return a Church boolean, and a function head. Finally, use an encoding similar to that of prd to write a tail function.

Normal Forms and Divergence

Normal form for →:

Term t for which no s exists such that t → s

There are terms in λ-calculus without normal forms:

omega == (λ x. x x) (λ x. x x)

→ omega

omega is said to be *divergent*, non-terminating

Fixed Points

Define:

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

We see that:

fix f → (λx. f (λy. x x y)) (λx. f (λy. x x y))

→ f (λy. (λx. f (λy. x x y)) (λx. f (λy. x x y)) y)

"=" f (λx. f (λy. x x y)) (λx. f (λy. x x y))

== f(fix f)

"=" is actually η-conversion, see later

fix can be used to define recursive functions

Define first g = λf."body of function to be defined" f

Then fix g is the result

Recursion

Define

factbody == λf. λn. if (equal n $\underline{0}$) $\underline{1}$ (times n (f (prd n)))

factorial == fix factbody

Exercise 5: Compute factorial $\underline{n}$ for some n

Exercise 6: Write a function that sums all members of a list of Church numerals

Free and Bound Variables

Now turn to some formalities about λ-terms

FV(t): The set of free variables of term t

FV(x) = {x}

FV(t s) = FV(t) ∪ FV(s)

FV(λx. t) = FV(t) – {x}

Example.

Bound variable: In λx.t, x is a bound variable

Closed term t: FV(t) = ∅

Substitution, I

Tricky business

Attempt #1:
- $x[s/x] = s$
- $y[s/x] = y$, if $x \neq y$
- $(\lambda y.\ t)[s/x] = \lambda y.(t[s/x])$
- $(t_1\ t_2)[s/x] = (t_1[s/x])\ (t_2[s/x])$

But then:
- $(\lambda x.\ x)[y/x] = \lambda x.\ y$

The bound variable x is turned into free variable y!

Substitution, II

Attempt #2:
- $x[s/x] = s$
- $y[s/x] = y$, if $x \neq y$
- $(\lambda y.\ t)[s/x] = \lambda y.\ t$, if $x = y$
- $(\lambda\ y.\ t)[s/x] = \lambda\ y.(t[s/x])$, if $x \neq y$
- $(t_1\ t_2)[s/x] = (t_1[s/x])\ (t_2[s/x])$

Better, but now:
- $(\lambda x.\ y)[x/y] = \lambda x.\ x$

Capture of bound variable!

Substitution, III

Attempt #3:
- $x[s/x] = s$
- $y[s/x] = y$, if $x \neq y$
- $(\lambda y.\ t)[s/x] = \lambda y.\ t$, if $x = y$
- $(\lambda y.\ t)[s/x] = \lambda y.(t[s/x])$, if $x \neq y$ and $y \notin FV(s)$
- $(t_1\ t_2)[s/x] = (t_1[s/x])\ (t_2[s/x])$

Even better, but now $(\lambda x.\ y)[x/y]$ is undefined

Alpha-conversion

Solution: Work with terms up to renaming of bound variables

Alpha-conversion: Terms that are identical up to choice of bound variables are interchangeable in all contexts

$t_1 =_\alpha t_2$: t_1 and t_2 are identical up to alpha-conversion

Convention: If $t_1 =_\alpha t_2$ then $t_1 = t_2$
Example: $\lambda x\ y.\ x\ y\ z$

Really working with terms modulo $=_\alpha$

All operations must respect $=_\alpha$

Substitution, IV

Final attempt:
- $x[s/x] = s$
- $y[s/x] = y$, if $x \neq y$
- $(\lambda y.\ t)[s/x] = \lambda y.(t[s/x])$, if $x \neq y$ and $y \notin FV(s)$
- $(t_1\ t_2)[s/x] = (t_1[s/x])\ (t_2[s/x])$

Clause for case $x = y$ not needed due to $=_\alpha$

Now:
$(\lambda x.\ t)[s/x]$
$\quad = (\lambda y.\ t[y/x])[s/x]$, where $y \notin FV(t) \cup \{x\} \cup FV(s)$
$\quad = \lambda y.\ t[y/x][s/x]$

Confluence

Confluence, or Church-Rosser (CR) property:
- if $s \to^* s_1$ and $s \to^* s_2$ then there is t such that $s_1 \to^* t$ and $s_2 \to^* t$

$$\begin{tikzcd} & s \arrow[dl, "*"'] \arrow[dr, "*"] & \\ s_1 \arrow[dr, dashed, "*"'] & & s_2 \arrow[dl, dashed, "*"] \\ & t & \end{tikzcd}$$

Full β reduction in λ-calculus is confluent
Order of reduction does not matter for result
Normal forms in λ-calculus are unique

Example: Use example slide 7

Conversion and Reduction

Primary concept is reduction $\rightarrow$

β–*conversion* $s =_\beta t$:

- s and t have common reduct under $\rightarrow_\beta^*$
- Exists s' such that $s \rightarrow_\beta^* s'$ and $t \rightarrow_\beta^* s'$

t *reducible* if s exists such that $t \rightarrow s$

- If and only if t contains a redex $(\lambda x.t_1)\ t_2$
- Exercise 7: Show this formally.

Then s is *reduct* of t under $\rightarrow$

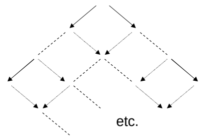
Normal Forms

If t not reducible then t is in *normal form*

t *has normal form* (under $\rightarrow$):
there is some s in normal form such that $t \rightarrow^* s$

Proposition: If $\rightarrow$ is CR then normal forms, if they exist, are unique.

Proof: Diagram chasing



η–Conversion

Principle of extensionality:

Terms are determined by their functional behaviour

So: If two terms are equal for all arguments, then they should be regarded as the same

So if $x \notin FV(t)$ then $t = \lambda x.\ t\ x$

Congruence closure rules

η–reduction:

$$\frac{x \notin FV(t)}{\lambda x.\ t\ x \rightarrow_\eta t} \qquad \frac{s \rightarrow_\eta s'}{s\ t \rightarrow_\eta s'\ t}$$

$$\frac{t \rightarrow_\eta t'}{s\ t \rightarrow_\eta s\ t'} \qquad \frac{t \rightarrow_\eta t'}{\lambda\ x.\ t \rightarrow_\eta \lambda\ x.\ t'}$$

η–Conversion, II

Example: $(\lambda\ x.\ f\ x)\ (\lambda\ y.\ g\ y) \rightarrow_\eta^* f\ g$

$\rightarrow_{\beta\eta}$: Both β and η reductions allowed

Both $\rightarrow_\eta$ and $\rightarrow_{\beta\eta}$ are CR

Equality in Isabelle is modulo α, β, η

Some History

λ–calculus was originally intended as foundation for mathematics

Frege (predicate logic, ~1879):
Allows arbitrary quantification over predicates

Russell (1901): Paradox $D = \{X \mid X \notin X\}$

Russell and Whitehead (Principia Mathematica, 1910-13):
Types and type orders, fix the problem

Church (1930): λ-calculus as logic

"Inconsistency" of λ-calculus

Logical sentences coded as λ-terms

- $x \in P == P\ x$
- $\{x \mid P\ x\} == \lambda x.\ P\ x$

Define

- $D == \lambda\ x.\ not(x\ x)$

Then (Russell's paradox)

- $D\ D =_\beta not(D\ D)$

Exercise 8

Prove the following lemma concerning the relation $\rightarrow_{\beta}$:

Lemma: If $t \rightarrow_{\beta} s$ then $t = t_1[t_2/x]$ for some x, t_1, t_2 such that x occurs exactly once in t_1, and such that

- t_2 has the form $(\lambda y.t_{2,1})\ t_{2,2}$ (for some y, $t_{2,1}$, $t_{2,2}$)
- $s = t_1[t_{2,1}[t_{2,2}/y]/x]$

Use this lemma to conclude that there are t, t', s, s' such that $t \rightarrow_{\beta} t'$, $s \rightarrow\beta\ s'$, but $t\ s \rightarrow_{\beta} t'\ s'$ does *not* hold