CSci 8980, Fall 2012
Specifying and Reasoning About Computational Systems
The (Untyped and Typed) Lambda Calculus

Gopalan Nadathur

Department of Computer Science and Engineering
University of Minnesota

Lectures in Fall 2012

Resources

Some books about the lambda calculus:

- Introduction to Combinators and λ-Calculus, J. Roger Hindley and Jonathan P. Seldin, Cambridge University Press
- The Lambda Calculus: Its Syntax and Semantics, Henk Brendregt, North Holland, Amsterdam
- Proofs and Types, J.-Y. Girard, Y. Lafont and P. Taylor, Cambridge University Press
- Introduction to Lambda Calculus, Henk Barendregt and Erik Barendsen, online notes

The Purpose of the Lambda Calculus

A formal system for studying the concept of functionality

Two different views can be taken of functions:

- as objects given by sets of ordered pairs
- as rules for carrying out a computation

The lambda calculus takes the latter as the primitive notion

Possible applications for the lambda calculus:

- device for defining and studying computability
- foundation for/formalization of a logic of functions
- vehicle for actually carrying out computations
- framework for studying programming language questions

The last two applications underlie our interest in the lambda calculus

Varieties of Lambda Calculi

Different versions of the lambda calculus have been studied and used in programming language research

These versions can be categorized based on

- typing
  - untyped, as underlying Lisp or Scheme
  - simply typed, as used in languages like C and Pascal
  - polymorphically typed as in ML and Haskell
  - second order polymorphically typed
  - dependently typed
- constraints on the use of function arguments
  - unconstrainted as in typical programming languages
  - requiring argument to be non-vacuous (λ**I**)
  - requiring the argument to be used exactly once (linear)
- evaluation order: strict versus non-strict lambda calculus

We will look primarily at the unconstrained λ-calculus without types and with simple types

The Starting Point

The lambda calculus arose from making a distinction between

- expressions with free variables, and
- functions

In common mathematical parlance, there is a confusion between these notions

For example

- "$(x^2 + x)^2$ is greater than 1000"
- "$(x^2 + x)^2$ is a computable function"

This ambiguity causes problems in formal settings

Church proposed notation to distinguish the two uses:

- use $(x^2 + x)^2$ to denote a (context dependent) value
- write $\lambda x\,(x^2 + x)^2$ to denote a function of x

Formalizing the Lambda Notation

Treating the "lambda notation" seriously requires attention to a few surrounding issues:

- Argument name must be treated as being unimportant

  E.g., $\lambda x\,(x^2 + x)^2$ and $\lambda y\,(y^2 + y)^2$ must be considered equivalent
- Function application must be interpreted properly

  In a logical context, $(\lambda x\,(x^2 + x)^2)\ (5)$ and $(5^2 + 5)^2$ must be recognized to be equal

  In a computational setting, the former should evaluate to the latter
- Substitution during evaluation must pay attention to binding

  For example $(\lambda x\,(\lambda y\,(y^2 + x)))\ (y)$ should evaluate to $\lambda z\,(z^2 + y)$ and not to $\lambda y\,(y^2 + y)$

Defining these operations correctly leads to a surprisingly rich notion of functions

Functions of Multiple Arguments

Is it enough to treat only single argument functions?

Yes, for two different reasons:

- Multiple arguments can be treated via encoding and decoding functions

  For example, the function defined by

  $h(x, y) = x - y$

  can be given by $\lambda z\,(fst(z) - snd(z))$

  Moreover, projection and pairing functions can be defined in the λ-calculus
- Functionality based on multiple arguments can be treated completely by the iterated application of single arguments

  For example, h above becomes simply $\lambda x\,\lambda y\,(x - y)$

  This device is referred to as "currying"

Syntax of the Lambda Calculus

We assume two sets of symbols at the outset

$\mathcal{V}$ a countably infinite set of variable symbols

$\mathcal{C}$ a countably infinite set of constant symbols

Definition

A lambda term (or term, for short) is defined inductively as follows:

- any symbol in $\mathcal{V}$ or $\mathcal{C}$ is a term

  atomic terms
- if t_1 and t_2 are terms then so is $(t_1\ t_2)$

  application of t_1 to t_2
- if $x \in \mathcal{V}$ and t is a term, then so is $(\lambda x\, t)$

  abstraction that binds x and has t as its scope

The formation rules lead naturally to a subterm relation that we will use in further discussions

Some Comments and Conventions

- Parentheses in terms may be dropped using the following conventions
  - abstraction is right associative

    I.e. $\lambda x \lambda y\, M$ is to be read as $(\lambda x\, (\lambda y\, M))$
  - application is left associative

    I.e. $M_1\ M_2\ M_3$ is to be read as $((M_1\ M_2)\ M_3)$
  - application binds stronger than abstraction

    I.e. $\lambda x\, M_1\ M_2$ is to be read as $(\lambda x\, (M_1\ M_2))$
- Often we assume $\mathcal{C}$ is $\emptyset$ to get what is called the *pure* lambda calculus
- We sometimes assume symbols in $\mathcal{C}$ with special conventions

  E.g. in $\lambda x\, (x^2 + x)^2$

  However, these symbols are typically *uninterpreted*

Some Comments and Conventions (Continued)

- The language is higher-order

  Arguments can be functions: $(\lambda x\, x)\ (\lambda x\, x)$

  Results can be functions: $(\lambda x\, \lambda y\, (x + y))\ 2$
- The language is currently typeless

  $(\lambda x\, (x^2 + x)^2)\ 2$ and $(\lambda x\, (x^2 + x)^2)\ (\lambda x\, x)$ are both fine

  Later we may add types to rule out some expressions
- Self application is permitted

  $(2\ 2)$ is also a term

  can also be meaningful: $(\lambda x\, (x\ x))\ (\lambda y\, y)$

  However, we need the “function-as-a-rule” idea to make sense of this

Recap: Syntax of the Lambda Calculus

The language is parameterized by the sets $\mathcal{V}$ and $\mathcal{C}$ of variable and constant symbols

Definition

The collection of lambda terms is the smallest set such that

- any symbol in $\mathcal{V}$ or $\mathcal{C}$ is a term

  atomic terms
- if t_1 and t_2 are terms then so is $(t_1\ t_2)$

  application of t_1 to t_2
- if $x \in \mathcal{V}$ and t is a term, then so is $(\lambda x\, t)$

  abstraction that binds x and has t as its scope

Several conventions in place for reducing the number of parentheses

Some Observations about the Language

- The language is higher-order

  Arguments can be functions: $(\lambda x\, x)\ (\lambda x\, x)$

  Results can be functions: $(\lambda x\, \lambda y\, (x + y))\ 2$
- The language is currently typeless

  $(\lambda x\, (x^2 + x)^2)\ 2$ and $(\lambda x\, (x^2 + x)^2)\ (\lambda x\, x)$ are both fine

  Later we may add types to rule out some expressions
- Self application is permitted

  $(2\ 2)$ is also a term

  can also be meaningful: $(\lambda x\, (x\ x))\ (\lambda y\, y)$

  However, we need the “function-as-a-rule” idea to make sense of this

Free and Bound Variables

Based on the construction rules for terms, we can talk about *occurrences* of variables

Definition

- An occurrence of a variable x in a term is said to be *bound* if it appears within a subpart of the form $(\lambda x\, t)$
- The occurrence is *free* if it is not bound
- A *variable* is free in a term if it has a free occurrence and it is bound if it has a bound occurrence

We use the following notation

$\mathcal{BV}(t)$	the set of bound variables of t
$\mathcal{FV}(t)$	the set of free variables of t

Note that $\mathcal{BV}(t)$ and $\mathcal{FV}(t)$ need not be disjoint

Substitution Into Lambda Terms

Definition

$N[x := M]$, *the logically correct substitution of M for x into N*, is defined by induction on N as follows:

- If $N \in \mathcal{V}$ then
  - if N is x then $N[x := M]$ is M
  - otherwise $N[x := M]$ is N
- If $N \in \mathcal{C}$ then $N[x := M]$ is N
- If $N = (N_1\ N_2)$ then $N[x := M] = (N_1[x := M]\ N_2[x := M])$
- If $N = \lambda y\, N_1$ then
  - if $y = x$ then $N[x := M] = N$
  - if $y \neq x$ and either $y \notin \mathcal{FV}(M)$ or $x \notin \mathcal{FV}(N)$ then $N[x := M] = \lambda y\, (N_1[x := M])$
  - otherwise $N[x := M] = \lambda z\, (N_1[y := z][x := M])$ where z is the first variable such that $z \notin \mathcal{FV}(N_1)$ and $z \notin \mathcal{FV}(M)$

Alpha Conversion

A relation between terms that formalizes the inconsequentiality of name choices for bound variables

Definition

Let $\lambda x\, P$ be a subterm of N and let $y \in \mathcal{V}$ be such that $y \notin \mathcal{FV}(P)$

Then M results from N by an α-step if M is obtained by replacing $\lambda x\, P$ in N by $\lambda y\, (P[x := y])$

Definition

N α-converts to M if M can be obtained from N by a finite sequence of α-steps

Notation $N \equiv_\alpha M$

Proposition: $\equiv_\alpha$ is an equivalence relation

Proof left as exercise; only nontrivial part is showing symmetry

Beta Conversion

A relation between terms that formalizes function evaluation

Definition

- A term of the form $((\lambda x\, M)\ N)$ is called a β-redex
- A term P β-contracts to Q if
  - P contains a β-redex $(\lambda x\, M)\ N$
  - Q is obtained by replacing this redex with the term $M[x := N]$

  Notation: $P \rhd_{1\beta} Q$
- P β-reduces to Q if Q results from P by a finite sequence of β-contractions and α-steps

  Notation: $P \rhd_\beta Q$
- P β-converts to Q if Q results from P by a finite sequence of β-contractions, inverse β-contractions and α-steps

  Notation: $P \equiv_\beta Q$

Examples

- $(\lambda x\,(x^2 + x)^2)\ 2 \quad \rhd_{1\beta} \quad (2^2 + 2)^2$
- $(\lambda x\, \lambda y\,(y\ x))\ y \quad \rhd_{1\beta} \quad \lambda w\,(w\ y)$
- $(\lambda x\,(x\ x\ y))\ (\lambda x\,(x\ x\ y))$

$$\begin{array}{ll} \rhd_{1\beta} & (\lambda x\,(x\ x\ y))\ (\lambda x\,(x\ x\ y))\ y \\ \rhd_{1\beta} & (\lambda x\,(x\ x\ y))\ (\lambda x\,(x\ x\ y))\ y\ y \\ \rhd_{1\beta} & \ldots \end{array}$$

Beta Conversion as an Equality Notion

That this is a good notion is a consequence of the following proposition

Proposition

$\equiv_\beta$ is an equivalence relation

Proof Sketch

By adding a few extra α-steps, we can eliminate renaming in substitution

Then the invertibility of β-contractions depends only on the invertibility of α-steps

This we have already seen to be true

Beta Normal Forms

We can think of β-reduction as function evaluation

Then we can think of terms to which this operation can no more be applied as the *values* produced

Definition

- A term containing no β-redexes is called a β-normal form
- If $P \rhd_\beta Q$ and Q is a β-normal form, then Q is called a β-normal form of P

Examples

$(\lambda w\ w\ y)$ is a β-normal form

Further, it is a β-normal form of $(\lambda x\,\lambda y\,(y\ x))\ y$

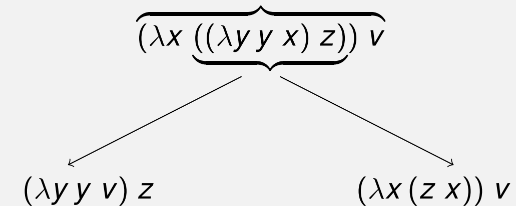
Similarly, $(2^2 + 2)^2$ is a β-normal form of $\lambda x\,(x^2 + x)^2\ 2$

$(\lambda x\ x\ x)\ (\lambda x\ x\ x)$ has no β-normal form; the λ-calculus can support non-terminating computations

Uniqueness of Beta Normal Forms

There can be many reduction paths from a given term

For example, consider



In this case we can "close" the diagram but is this possible in general?

A source of complexity: contraction can duplicate redexes

A possible solution: generalize to sequences of contractions

Unfortunately, this gets a bit complicated because contraction can create redexes
E.g. consider $(\lambda x\,(\lambda y\,(y\ x))\ (\lambda x\ x))$

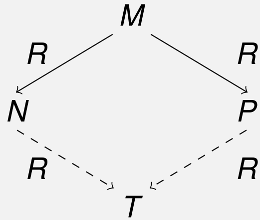
Confluence and the Church Rosser Property

Let $\mathcal{R}$ be a relation on terms and let $\mathcal{R}^*$ be its reflexive and transitive closure

R satisfies the *diamond property* if

$$\forall M, N, P(\ \langle M, N\rangle \in \mathcal{R} \text{ and } \langle M, P\rangle \in \mathcal{R} \\ \Rightarrow \exists T(\langle N, T\rangle \in \mathcal{R} \text{ and } \langle P, T\rangle \in \mathcal{R})\ )$$

Pictorially,



A relation $\mathcal{R}$ is said to be *confluent* or *Church-Rosser* just in the case that $\mathcal{R}^*$ satisfies the diamond property

The Church-Rosser Theorem

Theorem: $\rhd_{1\beta} \cup \alpha\text{-step}$ is confluent

The proof is non-trivial and yields many deep insights into the lambda calculus

Corollary *(Uniqueness of Normal Forms)*

If P has M and N as β-normal forms then $M \equiv_\alpha N$

Proof

By the Church-Rosser property,

$$\exists T \text{ such that } M \rhd_\beta T \text{ and } N \rhd_\beta T$$

Since there are no β-redexes in M and N, in fact

$$M \equiv_\alpha T \quad \text{and} \quad N \equiv_\alpha T$$

But then, since $\equiv_\alpha$ is an equivalence relation, $M \equiv_\alpha N$

Finding Normal Forms

- There are untyped lambda terms that lead to non-terminating β-reductions

  For example, consider $(\lambda x\, x\ x)\ (\lambda x\, x\ x)$

- How we choose β-redexes in normalization can make a difference in finding normal forms

  For example consider

  $$\overbrace{(\lambda x\, \lambda y\, y)\ (\underbrace{(\lambda x\, x\ x)\ (\lambda x\, x\ x)})}$$

  This term has a β-normal form and also an infinite reduction sequence

- There is a strategy—the leftmost outermost or normal reduction strategy—that will always produce a normal form when it exists

A Calculus for Formalizing Normal Reduction

Evaluation strategies can be characterized by an inference system

$$\frac{}{t \leadsto t}\ \textit{atom}, t \in \mathcal{C} \cup \mathcal{V} \qquad \frac{t \leadsto t'}{\lambda x\, t \leadsto \lambda x\, t'}\ \textit{abs}$$

$$\frac{t_1 \leadsto_w t_1' \quad t[x := t_2] \leadsto t'}{(t_1\ t_2) \leadsto t'}\ \textit{red}, t_1' \equiv_\alpha \lambda x\, t$$

$$\frac{t_1 \leadsto_w t_1' \quad t_2 \leadsto t_2'}{(t_1\ t_2) \leadsto (t_1'\ t_2')}\ \textit{app}, t_1' \not\equiv_\alpha \lambda x\, t$$

$$\frac{}{t \leadsto_w t}\ \textit{w-atom}, t \in \mathcal{C} \cup \mathcal{V} \qquad \frac{}{\lambda x\, t \leadsto_w \lambda x\, t}\ \textit{w-abs}$$

$$\frac{t_1 \leadsto_w t_1' \quad t[x := t_2] \leadsto_w t'}{(t_1\ t_2) \leadsto_w t'}\ \textit{w-red}, t_1' \equiv_\alpha \lambda x\, t$$

$$\frac{t_1 \leadsto_w t_1' \quad t_2 \leadsto t_2'}{(t_1\ t_2) \leadsto_w (t_1'\ t_2')}\ \textit{w-app}, t_1' \not\equiv_\alpha \lambda x\, t$$

Representing Mathematical Operations

In expressions such as $\lambda x\,(x^2 + x)^2$ arithmetic operations are represented by uninterpreted constant symbols

In real programming languages such as Scheme, ML or Haskell, these become interpreted or builtin operations

Our present question: Can these be defined in the pure λ-calculus?

This conveys some information about the strength of the lambda calculus

What Would Representation Mean?

We would first have to think of encodings of natural numbers

Let us write $\overline{m}$ to denote the encoding of m

Then we would say that, for example, we can represent or define addition on natural numbers

if we can write a λ-term f such that

$$\forall m, n \in \mathcal{N}\ (f\ \overline{m}\ \overline{n}) \rhd_\beta \overline{m+n}$$

Church Numerals and Lambda Definability

- Church proposed encoding the natural number n as the n-fold product forming function:

$$\overline{0} = \lambda f\,\lambda x\,x$$
$$\overline{1} = \lambda f\,\lambda x\,f\ x$$
$$\ldots$$
$$\overline{n} = \lambda f\,\lambda x\,\underbrace{f\,(f\ \ldots\ (f}_{n}\ x)\ldots) = \lambda f\,\lambda x\,f^n(x)$$

- Lambda definability then is the correlate of representation

  A term M lambda defines an n-place partial function φ on natural numbers if
  - $(M\ \overline{m_1}\ \cdots\ \overline{m_n}) \rhd_\beta \overline{\varphi(m_1, \ldots, m_n)}$ whenever $\varphi(m_1, \ldots, m_n)$ is defined, and
  - $(M\ \overline{m_1}\ \cdots\ \overline{m_n})$ does not have a β-normal form if $\varphi(m_1, \ldots, m_n)$ is not defined

  Note that nothing is said of M's behaviour on terms that do not encode natural numbers

Encoding Common Arithmetic Functions

Successor

$succ = \lambda n\,\lambda f\,\lambda x\,f\ (n\ f\ x)$

Addition

$plus = \lambda m\,\lambda n\,\lambda f\,\lambda x\,m\ f\ (n\ f\ x)$

Multiplication

$mult = \lambda m\,\lambda n\,\lambda f\,m\ (n\ f)$

Exponentiation

$exp = \lambda m\,\lambda n\,(m\ n)$

For example,

$$((exp\ \overline{3}\ \overline{2})\ f\ x) = (\lambda x\,(\overline{2}^3)\ x)\ f\ x$$
$$= (\overline{2}^3)\ f\ x = f^8(x)$$

Encoding Pairs

We want three lambda terms *pair*, *fst* and *snd* such that

$$fst\,(pair\ x\ y) = x$$
$$snd\,(pair\ x\ y) = y$$

Here is a possible choice:

$$pair = \lambda x\,\lambda y\,\lambda f\,(f\ x\ y)$$
$$fst = \lambda z\,z\,(\lambda x\,\lambda y\,x)$$
$$snd = \lambda z\,z\,(\lambda x\,\lambda y\,y)$$

Encoding the Predecessor Function

The function to be represented

$$f(k) = \begin{cases} 0 & \text{if } k = 0 \\ k - 1 & \text{otherwise (i.e. if } k > 0) \end{cases}$$

Here is a way to construct a representation:

- Start with the pair $\langle \overline{0}, \overline{0} \rangle$
  $zz = (\lambda x\,\lambda y\,\lambda f\,(f\ x\ y))\ (\lambda f\,\lambda x\,x)\ (\lambda f\,\lambda x\,x)$
- Apply to this k times the function
  $$f(\langle \overline{m}, \overline{n} \rangle) = \langle \overline{m+1}, \overline{m} \rangle$$
  Given by the λ-term
  $F = \lambda m\,pair\,(succ\,(\mathrm{fst}\ m))\ (\mathrm{fst}\ m)$
- Pick out the second element at the end

Combining all these we get

$$pred = \lambda k\,(snd\,(k\ F\ zz))$$

Encoding Some Other Standard Functions

Boolean Values and Operations

$true = \lambda x\,\lambda y\,x$

$false = \lambda x\,\lambda y\,y$

$cond = \lambda b\,b$

$\mathrm{and} = \lambda u\,\lambda v\ cond\ u\ v\ false$

$\mathrm{or} = \lambda u\,\lambda v\ cond\ u\ true\ v$

Exercises

zerop, *monus*, equality over natural numbers

Fact: The lambda definable functions are exactly the ones that can be described by Turing machines

Fixed Point Theorem

Theorem

There is a closed term Y such that for any f

$$Y\ f \rhd_\beta f\ (Y\ f)$$

In other words, Y yields a fixed point of f

Proof

Let $Z = \lambda f\,\lambda x\,(x\ (f\ f\ x))$

Then define Y to be $(Z\ Z)$

Now consider

$$\begin{aligned} Y\ f = (Z\ Z\ f) &\rhd_\beta \lambda x\,(x\ (Z\ Z\ x))\ f \\ &\rhd_\beta f\ (Z\ Z\ f) \\ &= f\ (Y\ f) \end{aligned}$$

Fixed Point Theorem (Continued)

Corollary

In the λ-calculus, we can solve any equation of the form

$$x\ y_1\ \cdots\ y_n = t$$

in the sense that there is a term X such that

$$X\ y_1\ \cdots\ y_n = t[x := X]$$

Proof

Let $X = (Y\ \lambda x\, \lambda y_1\ \ldots \lambda y_n\, t)$

Then we have

$$X \rhd_\beta (\lambda x\, \lambda y_1\ \ldots \lambda y_n\, t)\ X$$
$$\rhd_\beta \lambda y_1\ \ldots \lambda y_n\, t[x := X]$$

But then

$$X\ y_1\ \ldots\ y_n \rhd_\beta t[x := X]$$

Corollary justifies recursion in programming languages

Recursive Definitions in Programming Languages

The corollary justifies definitions like the following

*fact n = if (n = 0) then 1 else (n * (fact (n - 1)))*

The problem here is that *fact* is used in its own definition

But this is solved by the corollary:

$$fact = Y\ (\lambda f\, \lambda n\, (cond\ (= n\ 0)\ 1\ (*\ n\ (f\ (-\ n\ 1)))))$$

It is also illustrative to see how this would be evaluated:

$$\begin{aligned} fact\ 2 &= (\overbrace{(Y\ (\lambda f\, \lambda n\, (cond\ (= n\ 0)\ 1\ (*\ n\ (f\ (-\ n\ 1)))))}^{fact}\ 2) \\ &\rhd_\beta (\lambda n\, cond\ (= n\ 0)\ 1\ (*\ n\ ((Y \ldots)\ (-\ n\ 1))))\ 2 \\ &\rhd_\beta cond\ (= 2\ 0)\ 1\ (*\ 2\ ((Y \ldots)\ \ (-\ 2\ 1))) \\ &\rhd_\beta (*\ 2\ (\ (Y \ldots)\ (-\ 2\ 1)) \rhd_\beta (*\ 2\ (*\ 1\ (*\ ((Y \ldots)\ (-\ 1\ 1))))) \\ &\rhd_\beta (*\ 2\ (*\ 1\ 1)) \end{aligned}$$

Notice that normal reduction is needed to make this work!

Church-Rosser Theorem for Beta Conversion

β-conversion is intended to provide a notion of equality between terms

But how do we determine if two terms are equal?

A thought: How about reducing both to a normal form and comparing?

This would be possible only if we know that the normal forms must be the same for “equal” terms

Theorem *(Church Rosser Theorem for β-conversion)*

If $P \equiv_\beta Q$ then there is a T such that $P \rhd_\beta T$ and $Q \rhd_\beta T$

Consequences of the theorem:

- The equality notion is consistent (e.g. $\lambda x\, \lambda y\, x \not\equiv_\beta \lambda x\, \lambda y\, y$)
- When normal forms are guaranteed to exist, we get a decision procedure for β-equality

A Deductive Calculus for β-Equality

We shall call the formal system $\lambda\beta$

Its formulas are $M = N$ where M and N are λ-terms

Its rules are the following

$$\frac{}{\lambda x\, M = \lambda y\, M[x := y]}\ \alpha, y \notin \mathcal{FV}(M) \qquad \frac{}{(\lambda x\, M)\ N = M[x := N]}\ \beta$$

$$\frac{M = M'}{(N\ M) = (N\ M')}\ \mu \qquad \frac{N = N'}{(N\ M) = (N'\ M)}\ \nu$$

$$\frac{M = M'}{\lambda x\, M = \lambda x\, M'}\ \xi \qquad \frac{}{M = M}\ \rho$$

$$\frac{M = N \quad N = P}{M = P}\ \tau \qquad \frac{M = N}{N = M}\ \sigma$$

If $M = N$ is derivable using these rules we write $\vdash_{\lambda\beta} M = N$

Proposition: $M \equiv_\beta N$ if and only if $\vdash_{\lambda\beta} M = N$

Extensionality for Functions

There is a natural *extensionality* property that might be expected of a logic of functions:

$$(\forall x (f\ x) = (g\ x)) \Rightarrow f = g$$

This property is *not* true for $\equiv_\beta$: *e.g.* $y \not\equiv_\beta \lambda x\,(y\ x)$

We can get a richer notion of equality by adding the following rule to β-conversion:

> Replacing a subterm $\lambda x\,(M\ x)$ by M or vice versa provided $x \notin \mathcal{FV}(M)$

The two directions of this rule are known as η-contraction and η-expansion, respectively

The resulting equality notion is called $\beta\eta$-conversion and is written $\equiv_{\beta\eta}$

Capturing Extensionality in a Deductive Calculus

An alternative approach to formalizing extensionality is to add the following rule to the $\lambda\beta$ calculus:

$$\frac{}{\lambda x\,(M\ x) = M}\ \eta, x \notin \mathcal{FV}(M)$$

The resulting calculus is called $\lambda\beta\eta$ and we write $\vdash_{\lambda\beta\eta} M = N$ if $M = N$ is derivable in it

Proposition:

- $M \equiv_{\beta\eta} N$ if and only if $\vdash_{\beta\eta} M = N$
- The extensionality property holds of the $\lambda\beta\eta$ calculus

Proof Sketch

The first part is proved by induction in both directions

For the second, let $\vdash_{\lambda\beta\eta} (M\ x) = (N\ x)$ for any x

Pick x such that $x \notin \mathcal{FV}(M) \cup \mathcal{FV}(N)$ and use the ξ, η, symmetry and transitivity rules to conclude $\vdash_{\lambda\beta\eta} M = N$

Types and the Lambda Calculus

Untyped terms may, in some senses, be a bit too general:

- From an evaluation perspective, terms are eventually typed and not paying attention to this can lead to run-time errors
- From a logical perspective, lack of types leads to an inconsistent logic of functions

However, adding types can also lead to problems

- Types can disallow terms that have meaningful computational content, leading to loss of expressivity
- Types may also require you to encode the same computation in more complicated ways

These problems can be alleviated by adding a few well-chosen combinators and using more flexible typing schemes

Moreover, from a representational perspective the reduction in computational power can actually be an advantage

The Simply Typed Lambda Calculus

Here the types and terms are kept distinct and the former mainly classifies the latter

Moreover, types correspond to fixed, primitive sets of objects and functions over such sets

Formally, we first assume a nonempty set $\mathcal{S}$ of *sorts* and a set $\mathcal{TC}$ of type constructors, each of specified arity

Then the collection of all types $\mathcal{T}$ is the smallest set such that

- every sort is included in it — *atomic types*
- if $c \in \mathcal{TC}$ has arity n and $t_1, \ldots, t_n \in \mathcal{T}$ then $(c\ t_1\ \ldots\ t_n) \in \mathcal{T}$ — *constructed atomic types*
- if $t_1 \in \mathcal{T}$ and $t_2 \in \mathcal{T}$ then $t_1 \rightarrow t_2 \in \mathcal{T}$ — *function types*

Convention: $\rightarrow$ is right associative and constructor application binds tighter than $\rightarrow$

Simply Typed Lambda Terms

Let

Γ be an *environment* assigning types to variables

Σ be a *context* assigning types to constants

We write $x : \tau \in \Gamma$ and $c : \tau \in \Sigma$ to show these assignments

Then, a λ-term t is typeable if a judgment of the form $\Gamma \vdash_\Sigma t : \tau$ can be derived using the following rules

$$\frac{c : \tau \in \Sigma}{\Gamma \vdash_\Sigma c : \tau}\ \mathit{const} \qquad \frac{x : \tau \in \Gamma}{\Gamma \vdash_\Sigma x : \tau}\ \mathit{var}$$

$$\frac{\Gamma \vdash_\Sigma t_1 : \sigma \to \tau \quad \Gamma \vdash_\Sigma t_2 : \sigma}{\Gamma \vdash_\Sigma (t_1\ t_2) : \tau}\ \mathit{app}$$

$$\frac{x : \sigma \in \Gamma \quad \Gamma \vdash_\Sigma t : \tau}{\Gamma \vdash_\Sigma \lambda x\, t : \sigma \to \tau}\ \mathit{abs}$$

Also, t has type τ if $\Gamma \vdash_\Sigma t : \tau$ is derivable

Types and Lambda Terms

- Typeability adds a constraint to well-formedness
- As a result of types, many "useful" terms are disallowed, *e.g.*, *exp*, self application, the *Y* combinator
- However, we can add such terms through constants and rules for defining them

  For example,

$$\frac{B \leadsto \mathit{true} \quad L \leadsto V}{\mathit{cond}\ B\ L\ R \leadsto V}\ \mathit{condL} \qquad \frac{B \leadsto \mathit{false} \quad R \leadsto V}{\mathit{cond}\ B\ L\ R \leadsto V}\ \mathit{condR}$$

$$\frac{(F\ (\mathit{fix}\ F)) \leadsto V}{\mathit{fix}\ F \leadsto V}\ \mathit{fix}$$

  In fact, this is what is done in actual typed programming languages
- The types are "simple" like in Pascal and C, i.e. no polymorphism is present

Deciding Equality in the Simply Typed Language

Our focus is on the simply typed calculus *without* any interpreted constants where

- Every reduction sequence terminates, hence every term has a "normal form"
- Normal forms are unique up to α-conversion

This yields a simple algorithm for comparing terms

A normal form has the structure

$$\lambda x_1 \ldots \lambda x_n (@\ t_1\ \ldots\ t_n)$$

where @ is a constant, in $\{x_1, \ldots x_n\}$ or a free variable

The term is *flexible* in the last case and rigid otherwise

Terminology: binder, argument and head of the term

Compare also with the structure of a first-order term

Also note: $\beta\eta$-normal form, $\beta\eta$-long normal form

Parameterizing Lambda Terms with Types

- The types of certain λ-terms can be parameterized so that they are well-typed no matter what type is picked for the parameter

  For example, the term $\lambda x\, x$ has the type $\alpha \to \alpha$ for any choice of type for α

  Similarly, list constructors *nil* and *::* and functions like *append*, *map* etc can all be parameterized by types
- At a logical level, we can realize such parameterization as follows:
  - Introduce a new type that represents parameterization
  - Let terms have such parameterized types
  - Let terms (of suitable types) be applied to types to generate a particular "typed version" of the term
- This is exactly what System F or the second-order lambda calculus does

The Types of System F

We assume a collection of type variables $\mathcal{TV}$ in addition to the sorts $\mathcal{S}$ and type constructors $\mathcal{TC}$

Then the types are the smallest collection satisfying the following

- any element of $\mathcal{TV} \cup \mathcal{S}$ is a type
- if $c \in \mathcal{TC}$ has arity n and $t_1, \ldots, t_n$ are types then $(c\ t_1\ \ldots\ t_n)$ is a type
- if σ and τ are types then so is $(\sigma \rightarrow \tau)$
- for any $X \in \mathcal{TV}$ and any type τ, $(\Pi X\, \tau)$ is a type

The Terms of System F

We again define terms and their associated types via judgments of the form $\Gamma \vdash_\Sigma t : \tau$

The rules for deriving such judgments

$$\frac{c : \tau \in \Sigma}{\Gamma \vdash_\Sigma c : \tau}\ \mathit{const} \qquad \frac{x : \tau \in \Gamma}{\Gamma \vdash_\Sigma x : \tau}\ \mathit{var}$$

$$\frac{x : \sigma \in \Gamma \quad \Gamma \vdash_\Sigma t : \tau}{\Gamma \vdash_\Sigma \lambda x\, t : \sigma \rightarrow \tau}\ \mathit{abs} \qquad \frac{\Gamma \vdash_\Sigma t_1 : \sigma \rightarrow \tau \quad \Gamma \vdash_\Sigma t_2 : \sigma}{\Gamma \vdash_\Sigma (t_1\ t_2) : \tau}\ \mathit{app}$$

$$\frac{\Gamma \vdash_\Sigma t : \tau}{\Gamma \vdash_\Sigma \Lambda X\, t : \Pi X\, \tau}\ \mathit{typabs}$$

provided X is not free in the type of any $x \in \mathcal{FV}(t)$

$$\frac{\Gamma \vdash_\Sigma t : \Pi X\, \rho \quad \tau \equiv \rho[X := \sigma]}{\Gamma \vdash_\Sigma (t\ \sigma) : \tau}\ \mathit{typapp}$$

with suitable definitions of substitution and alpha equality for types

Odds and Ends

- Typically, constants and variables will start out with quantified types in new setting

  For example, we would expect the following of *::*

  $:: :\ \ \Pi X\, X \rightarrow \mathit{list}\ X \rightarrow \mathit{list}\ X$

  We would then have to supply these types as arguments to construct concrete terms, *e.g.*

  $(::\ \mathit{int}\ 1\ (\mathit{nil}\ \mathit{int}))$

  Exercise: try to show that this term is well-formed
- ML uses a restricted version of this kind of typing
  - type quantification only permitted at the outermost level
  - type application is implicit
- System F (and hence also STLT) have the property that every reduction sequence (defined in the obvious way) must terminate