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
  - unconstrained as in typical programming languages
  - requiring argument to be non-vacuous ($\lambda$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 $\lambda$-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 \(x\)

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
  \[ \lambda z \ (\text{fst}(z) - \text{snd}(z)) \]
  Moreover, projection and pairing functions can be defined in the \(\lambda\)-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”

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

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 \(C\) is \(\emptyset\) to get what is called the pure lambda calculus

- We sometimes assume symbols in \(C\) with special conventions
  - E.g. in \(\lambda x (x^2 + x)^2\)
  - However, these symbols are typically uninterpreted

---

### Recap: Syntax of the Lambda Calculus

The language is parameterized by the sets \(V\) and \(C\) of variable and constant symbols

**Definition**

The collection of lambda terms is the smallest set such that

- any symbol in \(V\) or \(C\) 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 \(x \in V\) 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 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

---

### 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 \( N[x := M] = M \)
  - otherwise \( N[x := M] = N \)
- If \( N \in \mathcal{C} \) then \( N[x := M] = N \)
- 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 \( \alpha \)-step if \( M \) is obtained by replacing \( \lambda x \, P \) in \( N \) by \( \lambda y \, (P[x := y]) \).

**Definition**

\( N \alpha\)-converts to \( M \) if \( M \) can be obtained from \( N \) by a finite sequence of \( \alpha \)-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 \( \beta \)-redex.
- A term \( P \) \( \beta \)-contracts to \( Q \) if
  - \( P \) contains a \( \beta \)-redex \( (\lambda x \, M) \, N \)
  - \( Q \) is obtained by replacing this redex with the term \( M[x := N] \)

  Notation: \( P \triangleright_{1\beta} Q \)

- \( P \) \( \beta \)-reduces to \( Q \) if \( Q \) results from \( P \) by a finite sequence of \( \beta \)-contractions and \( \alpha \)-steps.

  Notation: \( P \triangleright_{\beta} Q \)

- \( P \) \( \beta \)-converts to \( Q \) if \( Q \) results from \( P \) by a finite sequence of \( \beta \)-contractions, inverse \( \beta \)-contractions and \( \alpha \)-steps.

  Notation: \( P \equiv_{\beta} Q \)
Examples

- $((\lambda x (x^2 + x)^2) 2 \triangleright_{1} 2) 2$
- $((\lambda x \lambda y (y x)) y \triangleright_{1} \lambda w (w y))$
- $((\lambda x (x x y)) (\lambda x (x x y)))$
  - $\triangleright_{1} (\lambda x (x x y)) (\lambda x (x x y)) y$
  - $\triangleright_{1} (\lambda x (x x y)) (\lambda x (x x y)) y y$
  - $\triangleright_{1} \ldots$

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 $\alpha$-steps, we can eliminate renaming in substitution.

Then the invertibility of $\beta$-contractions depends only on the invertibility of $\alpha$-steps.

This we have already seen to be true.

Beta Normal Forms

We can think of $\beta$-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 $\beta$-redexes is called a $\beta$-normal form.
- If $P \triangleright_{\beta} Q$ and $Q$ is a $\beta$-normal form, then $Q$ is called a $\beta$-normal form of $P$.

**Examples**

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

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

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

Uniqueness of Beta Normal Forms

There can be many reduction paths from a given term.

For example, consider

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

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

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

Pictorially,

```
        M
        |   |
        R   R
        |   |
N         |
        |
        R
        |
        |
        P
```

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

Finding Normal Forms

- There are untyped lambda terms that lead to non-terminating $\beta$-reductions.
  
  For example, consider $\lambda x x x (\lambda x x x)$

- How we choose $\beta$-redexes in normalization can make a difference in finding normal forms.
  
  For example consider

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

  This term has a $\beta$-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.

The Church-Rosser Theorem

**Theorem:** $\triangleright_{1,\beta} \cup \alpha$-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 $\beta$-normal forms then $M \equiv_\alpha N$.

**Proof**

By the Church-Rosser property,

$$\exists T \text{ such that } M \triangleright_{1,\beta} T \text{ and } N \triangleright_{1,\beta} T$$

Since there are no $\beta$-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, $M \equiv_\alpha N$.

A Calculus for Formalizing Normal Reduction

Evaluation strategies can be characterized by an inference system:

- **atom**, $t \in \mathcal{C} \cup \mathcal{V}$
  
  $t \overset{\text{abs}}{\rightarrow} t'$

- **variable**, $t \in \mathcal{C} \cup \mathcal{V}$
  
  $t \overset{\text{w-abs}}{\rightarrow} \lambda x t$

- **application**, $t = t_1 t_2$
  
  $t \overset{\text{w-app}}{\rightarrow} (t_1' t_2')$

- **redex**, $t = t[x := t_2]$
  
  $t \overset{\text{red, } t_1' \equiv_\alpha \lambda x t}{\rightarrow} t'$

- **variable**, $t \in \mathcal{C} \cup \mathcal{V}$
  
  $t \overset{\text{w-red}}{\rightarrow} \lambda x t$

This term has a $\beta$-normal form and also an infinite reduction sequence.
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 built-in operations.

Our present question: Can these be defined in the pure $\lambda$-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 $\lambda$-term $f$ such that

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

Church Numerals and Lambda Definability

- Church proposed encoding the natural number $n$ as the $n$-fold product forming function:
  - $\overline{0} = \lambda f \lambda x x$
  - $\overline{1} = \lambda f \lambda x f x$
  - $\ldots$
  - $\overline{n} = \lambda f \lambda x f (\underbrace{f \ldots (f x)}_{n}) = \lambda f \lambda x f^n(x)$

- Lambda definability then is the correlate of representation:
  A term $M$ lambda defines an $n$-place partial function $\varphi$ on natural numbers if
    1. $(M \overline{m_1} \ldots \overline{m_n}) \triangleright_\beta \varphi(m_1, \ldots, m_n)$ whenever $\varphi(m_1, \ldots, m_n)$ is defined, and
    2. $(M \overline{m_1} \ldots \overline{m_n})$ does not have a $\beta$-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

$\text{succ} = \lambda n \lambda f \lambda x f (n f x)$

Addition

$\text{plus} = \lambda m \lambda n \lambda f \lambda x m f (n f x)$

Multiplication

$\text{mult} = \lambda m \lambda n \lambda f f m (n f)$

Exponentiation

$\text{exp} = \lambda m \lambda n (m n)$

For example,

$$((\text{exp} \overline{3} \overline{2}) f x) = (\lambda x (\overline{2}^3) x) f x$$
$$= (\overline{2}^3) f x = f^3(x)$$
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 $\langle 0, 0 \rangle$
  $$
  zz = (\lambda x \lambda y \lambda f (f x y)) (\lambda f \lambda x x) (\lambda f \lambda x x)
  $$
- Apply to this $k$ times the function
  $$
  f(\langle m, m \rangle) = \langle m+1, m \rangle
  $$
- Given by the $\lambda$-term
  $$
  F = \lambda m \lambda pair (\lambda x (\lambda x x)) (\lambda f \lambda y y)
  $$
- 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

$$
true = \lambda x \lambda y x \\
false = \lambda x \lambda y y \\
cond = \lambda b b
$$

and $= \lambda u \lambda v cond u v false$

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 $f$

$$
Y f \triangleright^\beta f (Y f)
$$

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

$$
Y f = (Z Z f) \triangleright^\beta \lambda x (x (Z Z x)) f \\
\triangleright^\beta f (Z Z f) \\
= f (Y f)
$$
Corollary

In the \( \lambda \)-calculus, we can solve any equation of the form
\[ x \ y_1 \cdot \ldots \cdot y_n = t \]
in the sense that there is a term \( X \) such that
\[ X \ y_1 \cdot \ldots \cdot y_n = t[ x := X ] \]

Proof

Let \( X = ( Y \ \lambda x \ y_1 \cdot \ldots \cdot y_n \ t ) \)
Then we have
\[ X \triangleright_{\beta} ( \lambda x \ y_1 \cdot \ldots \cdot y_n \ t ) \ X \]
\[ \triangleright_{\beta} \ y_1 \cdot \ldots \cdot y_n \ t [ x := X ] \]
But then
\[ X \ y_1 \cdot \ldots \cdot y_n \triangleright_{\beta} t [ x := X ] \]

Corollary justifies recursion in programming languages

The corollary justifies definitions like the following
\[ \text{fact} \ n = \text{if} \ (n = 0) \text{ then } 1 \text{ else } (n \ast \text{fact} \ (n - 1)) \]

The problem here is that \( \text{fact} \) is used in its own definition
But this is solved by the corollary:
\[ \text{fact} = Y \ ( \lambda f \ n \ (\text{cond} \ (= n 0) \ 1 \ (\ast n \ (f \ (- n 1)))) ) \]

It is also illustrative to see how this would be evaluated:
\[ \text{fact} \ 2 = \left( \left( \left( Y \ ( \lambda f \ n \ (\text{cond} \ (= n 0) \ 1 \ (\ast n \ (f \ (- n 1)))) ) \right) \ 2 \right) \right) \]
\[ \triangleright_{\beta} \ (\lambda n \ (\text{cond} \ (= n 0) \ 1 \ (\ast n \ ((Y \ldots) \ (- n 1)))) ) \ 2 \]
\[ \triangleright_{\beta} \ (\ast 2 \ ((Y \ldots) \ (- 2 1))) \triangleright_{\beta} (\ast 2 \ (\ast 1 \ ((Y \ldots) \ (- 1 1)))) \]
\[ \triangleright_{\beta} (\ast 2 \ (\ast 1 1)) \]

Notice that normal reduction is needed to make this work!

**Church-Rosser Theorem for Beta Conversion**

\( \beta \)-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 \( \beta \)-conversion)**

If \( P \equiv_{\beta} Q \) then there is a \( T \) such that \( P \triangleright_{\beta} T \) and \( Q \triangleright_{\beta} T \)

Consequences of the theorem:
- The equality notion is consistent (e.g. \( \lambda x \ y x \not\equiv_{\beta} \lambda x \ y y \))
- When normal forms are guaranteed to exist, we get a decision procedure for \( \beta \)-equality

**A Deductive Calculus for \( \beta \)-Equality**

We shall call the formal system \( \lambda \beta \)

Its formulas are \( M = N \) where \( M \) and \( N \) are \( \lambda \)-terms

Its rules are the following

\[ \frac{\lambda x \ M = \lambda y \ M}{\alpha, y \not\in \mathcal{FV}(M)} \]
\[ \frac{(\lambda x \ M) \ N = M[x := N]}{\beta} \]
\[ \frac{M = M'}{(N \ M) = (N' \ M') \mu} \]
\[ \frac{N = N'}{(N \ M) = (N' \ M') \nu} \]
\[ \frac{\lambda x \ M = \lambda x \ M'}{\xi} \]
\[ \frac{M = N \quad N = P}{M = P} \tau \]
\[ \frac{M = N \quad N = M}{M = M} \rho \]
\[ \frac{M = N \quad N = M}{M = M} \sigma \]

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 \(\beta\)-conversion:

Replacing a subterm \(\lambda x (M x)\) by \(M\) or vice versa provided \(x \not\in \mathcal{FV}(M)\)

The two directions of this rule are known as \(\eta\)-contraction and \(\eta\)-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:

\[
\frac{\lambda x (M x) = M}{\eta, x \not\in \mathcal{FV}(M)}
\]

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 \(x \not\in \mathcal{FV}(M) \cup \mathcal{FV}(N)\) and use the \(\xi\), \(\eta\), 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 \(S\) of *sorts* and a set \(\mathcal{T}\) 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
- if \(c \in \mathcal{T}\) has arity \(n\) and \(t_1, \ldots, t_n \in \mathcal{T}\) then \((c t_1 \ldots t_n) \in \mathcal{T}\)
- if \(t_1 \in \mathcal{T}\) and \(t_2 \in \mathcal{T}\) then \(t_1 \rightarrow t_2 \in \mathcal{T}\)

**Convention:** \(\rightarrow\) is right associative and constructor application binds tighter than \(\rightarrow\)
Simply Typed Lambda Terms

Let

\[ \Gamma \] be an environment assigning types to variables
\[ \Sigma \] be a context assigning types to constants

We write \( x : \tau \in \Gamma \) and \( c : \tau \in \Sigma \) to show these assignments.

Then, a \( \lambda \)-term \( t \) is typeable if a judgment of the form \( \Gamma \vdash \Sigma t : \tau \) can be derived using the following rules:

1. **Const** rule:
   \[ \frac{c : \tau \in \Sigma}{\Gamma \vdash \Sigma c : \tau} \]

2. **Var** rule:
   \[ \frac{x : \tau \in \Gamma}{\Gamma \vdash \Sigma x : \tau} \]

3. **App** rule:
   \[ \frac{\Gamma \vdash \Sigma t_1 : \sigma \to \tau \quad \Gamma \vdash \Sigma t_2 : \sigma}{\Gamma \vdash \Sigma (t_1 t_2) : \tau} \]

4. **Abs** rule:
   \[ \frac{x : \sigma \in \Gamma}{\Gamma \vdash \Sigma \lambda x t : \sigma \to \tau} \]

Also, \( t \) has type \( \tau \) 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.*, \( \text{exp} \), self application, the \( \text{Y} \) combinator.
- However, we can add such terms through constants and rules for defining them.

For example,

\[
\begin{align*}
B \sim \text{true} & \quad L \sim \text{V} \quad \text{cond}L \\
B \sim \text{false} & \quad R \sim \text{V} \quad \text{condR} \\
(F \text{(fix } F)) \sim \text{V}
\end{align*}
\]

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 \( \alpha \)-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 \( \lambda \)-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 \( \alpha \).

Similarly, list constructors \( \text{nil} \) and \( :: \) and functions like \( \text{append} \), \( \text{map} \) 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 \( TV \) in addition to the sorts \( S \) and type constructors \( TC \).

Then the types are the smallest collection satisfying the following:

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

The Terms of System F

We again define terms and their associated types via judgments of the form \( \Gamma \vdash t : \tau \).

The rules for deriving such judgments:

\[
\begin{align*}
\frac{c : \tau \in \Sigma}{\Gamma \vdash \Sigma c : \tau} & \quad \text{const} \\
\frac{x : \tau \in \Gamma}{\Gamma \vdash \Sigma x : \tau} & \quad \text{var} \\
\frac{x : \sigma \in \Gamma \quad \Gamma \vdash t : \tau}{\Gamma \vdash \Sigma \lambda x \: t : \sigma \rightarrow \tau} & \quad \text{abs} \\
\frac{\Gamma \vdash t_1 : \sigma \rightarrow \tau \quad \Gamma \vdash t_2 : \sigma}{\Gamma \vdash \Sigma (t_1 \: t_2) : \tau} & \quad \text{app} \\
\frac{\Gamma \vdash \Sigma t : \tau \quad \Sigma t_1 : \Pi X \rho \quad \tau \equiv \rho[X := \sigma]}{\Gamma \vdash \Sigma (t \: \sigma) : \tau} & \quad \text{typapp}
\end{align*}
\]

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

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 \text{list} X \rightarrow \text{list} X

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

  (:: \: \text{int} 1 \: (\text{nil 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