Automatic Deduction — LVA 703522

Introduction to Isabelle

Clemens Ballarin

Universität Innsbruck

λ

Contents

λ-Calculus

λ-Calculus

Alonzo Church

- Lived 1903–1995
- Supervised people like Alan Turing, Stephen Kleene
- Famous for Church-Turing thesis, λ-calculus, first undecidability results
- invented λ-calculus in 1930's

λ-calculus

- Originally meant as foundation of mathematics
- Important applications in theoretical computer science
- Foundation of computability and functional programming

Untyped λ-Calculus

- Turing-complete model of computation
- A simple way of writing down functions

Basic intuition:

Instead of	$f(x) = x + 5$
write	$f = \lambda x.\ x + 5$

$\lambda x.\ x + 5$

- a term
- a nameless function
- that adds 5 to its parameter

Function Application

For applying arguments to functions:

Instead of	$f(x)$
write	$f\ x$

Example $(\lambda x.\ x + 5)\ a$

Evaluating (computation) in $(\lambda x.\ t)\ a$ replace x by a in t

Example $(\lambda x.\ x + 5)\ (a + b)$ evaluates to $(a + b) + 5$

That's the idea.
Now formal.

Syntax

Terms $\quad t \; ::= \; v \mid c \mid (t\ t) \mid (\lambda x.\ t)$

$v, x \in V; \quad c \in C; \quad V, C$ sets of names

- v, x variables
- c constants
- $(t\ t)$ application
- $(\lambda x.\ t)$ abstraction

Conventions

- Leave out parentheses where possible
- List variables instead of multiple λ

Example

Instead of $(\lambda y.\ (\lambda x.\ (x\ y)))$ write $\lambda y\ x.\ x\ y$

Rules

- List variables: $\lambda x.\ (\lambda y.\ t) \;=\; \lambda x\ y.\ t$
- Application binds to the left: $x\ y\ z \;=\; (x\ y)\ z \neq x\ (y\ z)$
- Abstraction binds to the right:
  $\lambda x.\ x\ y \;=\; \lambda x.\ (x\ y) \neq (\lambda x.\ x)\ y$
- Leave out outermost parentheses

Getting Used to the Syntax

Example

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

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

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

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

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

Computation

Intuition

- Replace parameter by argument.
- This is called β-reduction.

Example

$$(\lambda x\ y.\ f\ (y\ x))\ \ 5\ \ (\lambda x.\ x) \longrightarrow_\beta$$
$$(\lambda y.\ f\ (y\ 5))\ \ (\lambda x.\ x) \longrightarrow_\beta$$
$$f\ ((\lambda x.\ x)\ 5) \longrightarrow_\beta$$
$$f\ 5$$

Defining Computation

β-reduction

$$
\begin{array}{lllllll}
 & & & & (\lambda x.\ s)\ t & \longrightarrow_\beta & s[x \leftarrow t] \\
s & \longrightarrow_\beta & s' & \Longrightarrow & (s\ t) & \longrightarrow_\beta & (s'\ t) \\
t & \longrightarrow_\beta & t' & \Longrightarrow & (s\ t) & \longrightarrow_\beta & (s\ t') \\
s & \longrightarrow_\beta & s' & \Longrightarrow & (\lambda x.\ s) & \longrightarrow_\beta & (\lambda x.\ s')
\end{array}
$$

Still to do: define $s[x \leftarrow t]$

Defining Substitution

Easy concept. Small problem: variable capture.

Example $(\lambda x.\ x\ z)[z \leftarrow x]$

We do not want $(\lambda x.\ x\ x)$ as result.

What do we want?

In $(\lambda y.\ y\ z)\ [z \leftarrow x] = (\lambda y.\ y\ x)$ there would be no problem.

So, solution is: rename bound variables.

Free Variables

Bound variables

In $(\lambda x.\ t)$, x is a bound variable.

Free variables

$FV(t)$ denotes free variables of t

$$FV(x) = \{x\} \qquad FV(s\ t) = FV(s) \cup FV(t)$$
$$FV(c) = \{\} \qquad FV(\lambda x.\ t) = FV(t) \setminus \{x\}$$

Example

$$FV(\lambda x.\ (\lambda y.\ (\lambda x.\ x)\ y)\ y\ x) = \{y\}$$

Term t is called closed if $FV(t) = \{\}$

Substitution

$$
\begin{aligned}
x\ [x \leftarrow t] &= t \\
y\ [x \leftarrow t] &= y && \text{if } x \neq y \\
c\ [x \leftarrow t] &= c
\end{aligned}
$$

$$(s_1\ s_2)\ [x \leftarrow t] = (s_1[x \leftarrow t]\ s_2[x \leftarrow t])$$

$$(\lambda x.\ s)\ [x \leftarrow t] = (\lambda x.\ s)$$

$$(\lambda y.\ s)\ [x \leftarrow t] = (\lambda y.\ s[x \leftarrow t])$$

$$\text{if } x \neq y \text{ and } y \notin FV(t)$$

$$(\lambda y.\ s)\ [x \leftarrow t] = (\lambda z.\ s[y \leftarrow z][x \leftarrow t])$$

$$\text{if } x \neq y \text{ and } z \notin FV(t) \cup FV(s)$$

Substitution Example

$$
\begin{aligned}
& (x \;\; (\lambda x.\ x) \;\; (\lambda y.\ z\ x))[x \leftarrow y] \\
= \quad & (x[x \leftarrow y]) \;\; ((\lambda x.\ x)[x \leftarrow y]) \;\; ((\lambda y.\ z\ x)[x \leftarrow y]) \\
= \quad & y\ (\lambda x.\ x)\ (\lambda y'.\ z\ y)
\end{aligned}
$$

α-Conversion

Bound names are irrelevant

$\lambda x.\ x$ and $\lambda y.\ y$ denote the same function.

α-conversion

$s =_\alpha t$ means $s = t$ up to renaming of bound variables.

Formally

$$
\begin{array}{lllllll}
 & & & & (\lambda x.\ t) & \longrightarrow_\alpha & (\lambda y.\ t[x \leftarrow y]) \\
 & & & & & & \quad \text{if } y \notin FV(t) \\
s & \longrightarrow_\alpha & s' & \Longrightarrow & (s\ t) & \longrightarrow_\alpha & (s'\ t) \\
t & \longrightarrow_\alpha & t' & \Longrightarrow & (s\ t) & \longrightarrow_\alpha & (s\ t') \\
s & \longrightarrow_\alpha & s' & \Longrightarrow & (\lambda x.\ s) & \longrightarrow_\alpha & (\lambda x.\ s')
\end{array}
$$

$$s =_\alpha t \quad \text{iff} \quad s \longrightarrow_\alpha^* t$$

($\longrightarrow_\alpha^*$ = transitive, reflexive closure of $\longrightarrow_\alpha$ = multiple steps)

α-Conversion

Equality in Isabelle is equality modulo α-conversion.

if $s =_\alpha t$ then s and t are syntactically equal.

Examples

$$
\begin{array}{ll}
 & x\ (\lambda x\ y.\ x\ y) \\
=_\alpha & x\ (\lambda y\ x.\ y\ x) \\
=_\alpha & x\ (\lambda z\ y.\ z\ y) \\
\neq_\alpha & z\ (\lambda z\ y.\ z\ y) \\
\neq_\alpha & x\ (\lambda x\ x.\ x\ x)
\end{array}
$$

Back to β

We have defined β-reduction: $\longrightarrow_\beta$

Notations and Concepts

- β-conversion: $s =_\beta t$ iff $\exists n.\ s \longrightarrow_\beta^* n \wedge t \longrightarrow_\beta^* n$
- t is reducible if there is an s such that $t \longrightarrow_\beta s$
- $(\lambda x.\ s)\ t$ is called a redex (reducible expression)
- t is reducible iff it contains a redex
- If it is not reducible, t is in normal form
- t has a normal form if there is an irreducible s such that $t \longrightarrow_\beta^* s$

Does Every λ-Term Have a Normal Form?

No!

Example

$$
\begin{array}{l}
(\lambda x.\ x\ x)\ (\lambda x.\ x\ x) \ \longrightarrow_\beta \\
(\lambda x.\ x\ x)\ (\lambda x.\ x\ x) \ \longrightarrow_\beta \\
(\lambda x.\ x\ x)\ (\lambda x.\ x\ x) \ \longrightarrow_\beta \ \ldots
\end{array}
$$

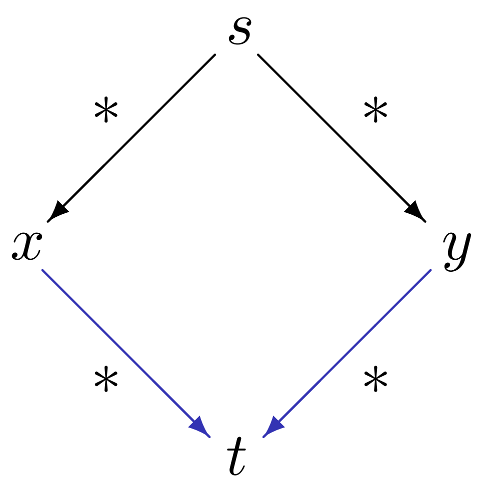
(but: $(\lambda x\ y.\ y)\ ((\lambda x.\ x\ x)\ (\lambda x.\ x\ x)) \ \longrightarrow_\beta \ \lambda y.\ y$)

λ-calculus is not terminating.

β-Reduction is Confluent

Confluence

$s \longrightarrow_\beta^* x \wedge s \longrightarrow_\beta^* y \Longrightarrow \exists t.\ x \longrightarrow_\beta^* t \wedge y \longrightarrow_\beta^* t$



Order of reduction does not matter for result.

Normal forms in λ-calculus are unique.

β-Reduction is Confluent

Example

$(\lambda x\ y.\ y)\ ((\lambda x.\ x\ x)\ a) \longrightarrow_\beta (\lambda x\ y.\ y)\ (a\ a) \longrightarrow_\beta \lambda y.\ y$

$(\lambda x\ y.\ y)\ ((\lambda x.\ x\ x)\ a) \longrightarrow_\beta \lambda y.\ y$

η-Reduction

Another case of trivially equal functions: $t = (\lambda x.\ t\ x)$

Definition

$$
\begin{array}{lllllll}
 & & & & (\lambda x.\ t\ x) & \longrightarrow_\eta & t \qquad \text{if } x \notin FV(t) \\
s & \longrightarrow_\eta & s' & \Longrightarrow & (s\ t) & \longrightarrow_\eta & (s'\ t) \\
t & \longrightarrow_\eta & t' & \Longrightarrow & (s\ t) & \longrightarrow_\eta & (s\ t') \\
s & \longrightarrow_\eta & s' & \Longrightarrow & (\lambda x.\ s) & \longrightarrow_\eta & (\lambda x.\ s')
\end{array}
$$

$$s =_\eta t \quad \text{iff} \quad \exists n.\ s \longrightarrow_\eta^* n \wedge t \longrightarrow_\eta^* n$$

Example

$(\lambda x.\ f\ x)\ (\lambda y.\ g\ y) \longrightarrow_\eta (\lambda x.\ f\ x)\ g \longrightarrow_\eta f\ g$

- η-reduction is confluent and terminating.
- $\longrightarrow_{\beta\eta}$ is confluent.
  $\longrightarrow_{\beta\eta}$ means $\longrightarrow_\beta$ and $\longrightarrow_\eta$ steps are both allowed.
- Equality in Isabelle is also modulo η-conversion.

In Fact . . .

Equality in Isabelle is modulo α-, β-, and η-conversion.

We will see later why that is possible.

So, What Can You Do With λ-Calculus?

λ-calculus is very expressive, you can encode:

- Logic, set theory
- Turing machines, functional programs, etc.

Examples

$$\begin{aligned}\texttt{true} &\equiv \lambda x\ y.\ x \\ \texttt{false} &\equiv \lambda x\ y.\ y \\ \texttt{if} &\equiv \lambda z\ x\ y.\ z\ x\ y\end{aligned}$$

$$\texttt{if true}\ x\ y \longrightarrow_\beta^* x$$

$$\texttt{if false}\ x\ y \longrightarrow_\beta^* y$$

Now, `not`, `and`, `or`, etc is easy:

$$\begin{aligned}\texttt{not} &\equiv \lambda x.\ \texttt{if}\ x\ \texttt{false true} \\ \texttt{and} &\equiv \lambda x\ y.\ \texttt{if}\ x\ y\ \texttt{false} \\ \texttt{or} &\equiv \lambda x\ y.\ \texttt{if}\ x\ \texttt{true}\ y\end{aligned}$$

More Examples

Church Numerals

$$
\begin{aligned}
0 &\equiv \lambda f\ x.\ x \\
1 &\equiv \lambda f\ x.\ f\ x \\
2 &\equiv \lambda f\ x.\ f\ (f\ x) \\
3 &\equiv \lambda f\ x.\ f\ (f\ (f\ x)) \\
\ldots
\end{aligned}
$$

Numeral n takes arguments f and x, applies f n-times to x.

$$
\begin{aligned}
\texttt{iszero} &\equiv \lambda n.\ n\ (\lambda x.\ \texttt{false})\ \texttt{true} \\
\texttt{succ} &\equiv \lambda n\ f\ x.\ f\ (n\ f\ x) \\
\texttt{add} &\equiv \lambda m\ n.\ \lambda f\ x.\ m\ f\ (n\ f\ x)
\end{aligned}
$$

Fixed Points

$$(\lambda x\ f.\ f\ (x\ x\ f))\ \ (\lambda x\ f.\ f\ (x\ x\ f))\ \ t \longrightarrow_\beta$$
$$(\lambda f.\ f\ ((\lambda x\ f.\ f\ (x\ x\ f))\ \ (\lambda x\ f.\ f\ (x\ x\ f))\ f))\ \ t \longrightarrow_\beta$$
$$t\ \ ((\lambda x\ f.\ f\ (x\ x\ f))\ (\lambda x\ f.\ f\ (x\ x\ f))\ t)$$

$$\mu = (\lambda x f.\ f\ (x\ x\ f))\ (\lambda x f.\ f\ (x\ x\ f))$$
$$\mu\ t \longrightarrow_\beta t\ (\mu\ t) \longrightarrow_\beta t\ (t\ (\mu\ t)) \longrightarrow_\beta t\ (t\ (t\ (\mu\ t))) \longrightarrow_\beta \ldots$$

μ is Turing's fixed point operator

Nice, but ...

As a mathematical foundation, λ does not work.
It is inconsistent.

- **Frege** (Predicate Logic, $\sim$ 1879):
  allows arbitrary quantification over predicates
- **Russel** (1901): Paradox $R \equiv \{X | X \notin X\}$
- **Whitehead & Russel** (Principia Mathematica, 1910-1913):
  fix the problem
- **Church** (1930):
  λ-calculus as logic, `true`, `false`, $\wedge$, ... as λ-terms

Problem

with	$\{x \mid P\ x\} \equiv \lambda x.\ P\ x$ and $x \in M \equiv M\ x$
you can write	$R \equiv \lambda x.\ \texttt{not}\ (x\ x)$
and get	$(R\ R) =_\beta \texttt{not}\ (R\ R)$

We Have Learned so Far...

- λ-calculus syntax
- Free variables, substitution
- β-reduction
- α- and η-conversion
- β-reduction is confluent
- λ-calculus is very expressive (Turing complete)
- λ-calculus is inconsistent

Demo

λ-calculus is Inconsistent

We have seen:
Can find term R such that $R\ R\ =_\beta\ \texttt{not}(R\ R)$

There are more terms that do not make sense:

1 2, `true false`, etc.

Solution: rule out ill-formed terms by using types.
(Church 1940)

Introducing Types

Idea: assign a type to each "sensible" λ-term.

Examples

- for "term t has type α" write $t :: \alpha$
- if x has type α then $\lambda x.\ x$ is a function from α to α
  Write: $(\lambda x.\ x) :: \alpha \Rightarrow \alpha$
- for $s\ t$ to be sensible:
  s must be function
  t must be right type for parameter

  If $s :: \alpha \Rightarrow \beta$ and $t :: \alpha$ then $(s\ t) :: \beta$

That's the idea.
Now formal.

Syntax for $\lambda^{\rightarrow}$

Terms $\quad t \quad ::= \quad v \mid c \mid (t\ t) \mid (\lambda x.\ t)$

$v, x \in V, \quad c \in C, \quad V, C$ sets of names

Types $\quad \tau \quad ::= \quad \mathtt{b} \mid \nu \mid \tau \Rightarrow \tau$

$\mathtt{b} \in \{\mathtt{bool}, \mathtt{int}, \ldots\}$ base types

$\nu \in \{\alpha, \beta, \ldots\}$ type variables

$$\alpha \Rightarrow \beta \Rightarrow \gamma \quad = \quad \alpha \Rightarrow (\beta \Rightarrow \gamma)$$

Contexts

Γ: function from variable and constant names to types.

Term t has type τ in context Γ: $\quad \Gamma \vdash t :: \tau$

Examples

$$\Gamma \vdash (\lambda x.\ x) :: \alpha \Rightarrow \alpha$$

$$[y \mapsto \texttt{int}] \vdash y :: \texttt{int}$$

$$[z \mapsto \texttt{bool}] \vdash (\lambda y.\ y)\ z :: \texttt{bool}$$

$$[] \vdash \lambda f\ x.\ f\ x :: (\alpha \Rightarrow \beta) \Rightarrow \alpha \Rightarrow \beta$$

A term t is **well-typed** or **type-correct**
if there are Γ and τ such that $\Gamma \vdash t :: \tau$

Type Checking Rules

Variables: $$\Gamma \vdash x :: \Gamma(x)$$

Application: $$\frac{\Gamma \vdash t_1 :: \tau_2 \Rightarrow \tau_1 \qquad \Gamma \vdash t_2 :: \tau_2}{\Gamma \vdash (t_1\ t_2) :: \tau_1}$$

Abstraction: $$\frac{\Gamma[x \mapsto \tau_1] \vdash t :: \tau_2}{\Gamma \vdash (\lambda x.\ t) :: \tau_1 \Rightarrow \tau_2}$$

Example Type Derivation

$$\dfrac{\dfrac{[x \mapsto \alpha, y \mapsto \beta] \vdash x :: \alpha}{[x \mapsto \alpha] \vdash \lambda y.\ x :: \beta \Rightarrow \alpha}}{[] \vdash \lambda x\ y.\ x :: \alpha \Rightarrow \beta \Rightarrow \alpha}$$

More Complex Example

$$\dfrac{\dfrac{\dfrac{\dfrac{\Gamma \vdash f :: \alpha \Rightarrow (\alpha \Rightarrow \beta) \qquad \Gamma \vdash x :: \alpha}{\Gamma \vdash f\ x :: \alpha \Rightarrow \beta} \qquad \Gamma \vdash x :: \alpha}{\Gamma \vdash f\ x\ x :: \beta}}{[f \mapsto \alpha \Rightarrow \alpha \Rightarrow \beta] \vdash \lambda x.\ f\ x\ x :: \alpha \Rightarrow \beta}}{[] \vdash \lambda f\ x.\ f\ x\ x :: (\alpha \Rightarrow \alpha \Rightarrow \beta) \Rightarrow \alpha \Rightarrow \beta}$$

$$\Gamma = [f \mapsto \alpha \Rightarrow \alpha \Rightarrow \beta, x \mapsto \alpha]$$

More General Types

A term can have more than one type.

Example

$[] \vdash \lambda x.\ x :: \mathtt{bool} \Rightarrow \mathtt{bool}$
$[] \vdash \lambda x.\ x :: \alpha \Rightarrow \alpha$

Some types are more general than others:

$\tau \lesssim \sigma$ if there is a substitution S such that $\tau = S(\sigma)$

Examples

$$\mathtt{int} \Rightarrow \mathtt{bool} \quad \lesssim \quad \alpha \Rightarrow \beta \quad \lesssim \quad \beta \Rightarrow \alpha \quad \not\lesssim \quad \alpha \Rightarrow \alpha$$

Most General Types

Fact: each type-correct term has a most general type.

Formally:

$$\Gamma \vdash t :: \tau \quad \Longrightarrow \quad \exists \sigma.\, \Gamma \vdash t :: \sigma \wedge (\forall \sigma'.\, \Gamma \vdash t :: \sigma' \Longrightarrow \sigma' \lesssim \sigma)$$

It can be found by executing the typing rules backwards.

- **type checking:** checking if $\Gamma \vdash t :: \tau$ for given Γ and τ
- **type inference:** computing Γ and τ such that $\Gamma \vdash t :: \tau$

Type checking and type inference on $\lambda^{\rightarrow}$ are decidable.

What about β-Reduction?

Definition of β reduction stays the same.

Fact: Well-typed terms stay well typed-during β-reduction

Formally: $\Gamma \vdash s :: \tau \ \land \ s \longrightarrow_{\beta} t \Longrightarrow \Gamma \vdash t :: \tau$

This property is called subject reduction.

What about Termination?

β-reduction in $\lambda^{\rightarrow}$ always terminates.

(Alan Turing, 1942)

- $=_{\beta}$ **is decidable**
  To decide if $s =_{\beta} t$, reduce s and t to normal form (always exists, because $\longrightarrow_{\beta}$ terminates), and compare result.
- $=_{\alpha\beta\eta}$ **is decidable**
  This is why Isabelle can automatically reduce each term to $\beta\eta$-normal form.

What Does This Mean for Expressiveness?

Not all computable functions can be expressed in $\lambda^{\rightarrow}$!

How can typed functional languages then be Turing complete?

Fact:

Each computable function can be encoded as closed, type correct $\lambda^{\rightarrow}$ term using $Y :: (\tau \Rightarrow \tau) \Rightarrow \tau$ as only constant.

- Y is Church's fixed point operator
- $Y \; t =_\beta t \; (Y \; t)$
- used for recursion

Summary

- Simply typed lambda calculus $\lambda^{\to}$ is consistent.
- β-reduction in $\lambda^{\to}$ satisfies subject reduction.
- β-reduction in $\lambda^{\to}$ always terminates.
- α-equivalent terms are equal in Isabelle.
- Isabelle automatically reduces to $\beta\eta$-normal form.