Introduction to λ-calculus

Nils Gesbert

MOSIG

Main Idea

One fundamental notion in mathematics is that of a function.

Main Idea

One fundamental notion in mathematics is that of a function.

A function is something which yields a result when given an argument.

Main Idea

One fundamental notion in mathematics is that of a function.

A function is something which yields a result when given an argument.

How is a function usually defined?

Given a function's definition and a value for its argument, how is the result of the function for this argument usually calculated?

Main Idea

One fundamental notion in mathematics is that of a function.

A function is something which yields a result when given an argument.

How is a function usually defined?

By a formula containing a variable to represent the function's argument.

$$f(x) = x^2 + 2x + 1$$

Given a function's definition and a value for its argument, how is the result of the function for this argument usually calculated?

Main Idea

One fundamental notion in mathematics is that of a function.

A function is something which yields a result when given an argument.

How is a function usually defined?

By a formula containing a variable to represent the function's argument.

$$f(x) = x^2 + 2x + 1$$

Given a function's definition and a value for its argument, how is the result of the function for this argument usually calculated?

By replacing (substituting) the variable with the argument's value.

$$f(3) = 3^2 + 2 \cdot 3 + 1 = 16$$

Main Idea

One fundamental notion in mathematics is that of a function.

A function is something which yields a result when given an argument.

How is a function usually defined?

By a formula containing a variable to represent the function's argument.

$$f(x) = x^2 + 2x + 1$$

Given a function's definition and a value for its argument, how is the result of the function for this argument usually calculated?

By replacing (substituting) the variable with the argument's value.

$$f(3) = 3^2 + 2 \cdot 3 + 1 = 16$$

λ-calculus is a formal language which represents that, and just that.

Context and History

Broad context: research on formal foundations for mathematics

formal logic systems, axiomatic theories (set theory, Peano arithmetic...)

Context and History

Broad context: research on formal foundations for mathematics

formal logic systems, axiomatic theories (set theory, Peano arithmetic...)

λ-calculus is originally a part of a formal logic system, proposed by Alonzo Church in 1931 and later abandoned.

Context and History

Broad context: research on formal foundations for mathematics
formal logic systems, axiomatic theories (set theory, Peano arithmetic...)

λ-calculus is originally a part of a formal logic system, proposed by Alonzo Church in 1931 and later abandoned.

Important problem in the 1930s: Some functions can be defined theoretically, but we do not know how to calculate their value for a given argument.

How to characterise functions which are 'effectively calculable'?
(for example, on natural numbers)

Context and History

Broad context: research on formal foundations for mathematics

formal logic systems, axiomatic theories (set theory, Peano arithmetic...)

λ-calculus is originally a part of a formal logic system, proposed by Alonzo Church in 1931 and later abandoned.

Important problem in the 1930s: Some functions can be defined theoretically, but we do not know how to calculate their value for a given argument.

How to characterise functions which are 'effectively calculable'? (for example, on natural numbers)

1934: Gödel and Herbrand define recursive functions: functions defined by sets of equations satisfying certain properties.

1936: Church, Kleene and Rosser prove that recursive functions are equivalent to λ-definable functions.

1936: Turing defines computability in terms of 'automatic machines' and proves that the computable functions are exactly the λ-definable functions.

Outline

Outline

Features of the language

Variables: $x, y, z, \ldots$

Two syntactic constructions:

- Function definition: $\lambda x \cdot \mathit{expr}$

  the function f defined by $f(x) = \mathit{expr}$

- Function application: $\mathit{expr}_1\ \mathit{expr}_2$

  the function expr_1 applied to the argument expr_2

Features of the language

Variables: $x, y, z, \ldots$

Two syntactic constructions:

- Function definition: $\lambda x \cdot expr$
  the function f defined by $f(x) = expr$
- Function application: $expr_1\ expr_2$
  the function $expr_1$ applied to the argument $expr_2$
- Parentheses to avoid ambiguity

Features of the language

Variables: $x, y, z, \ldots$

Two syntactic constructions:

- Function definition: $\lambda x \cdot \mathit{expr}$

  the function f defined by $f(x) = \mathit{expr}$

- Function application: $\mathit{expr}_1\ \mathit{expr}_2$

  the function expr_1 applied to the argument expr_2

- Parentheses to avoid ambiguity

One mechanism:

- Substitution: $[\mathit{expr}]\{x \mapsto \mathit{expr}'\}$ is the expression obtained by replacing all occurrences of x in expr with expr'.

Examples

- The identity function: **id** $= \lambda x \cdot x$

Examples

- The identity function: $\mathbf{id} = \lambda x \cdot x$
- The function which applies the identity function to its argument: $\lambda y \cdot \mathbf{id}\, y = \lambda y \cdot ((\lambda x \cdot x)\, y)$

Examples

- The identity function: $\mathbf{id} = \lambda x \cdot x$
- The function which applies the identity function to its argument: $\lambda y \cdot \mathbf{id}\, y = \lambda y \cdot ((\lambda x \cdot x)\, y)$
- The same function, applied to the identity function: $(\lambda y \cdot \mathbf{id}\, y)\, \mathbf{id}$

Examples

- The identity function: $\mathbf{id} = \lambda x \cdot x$
- The function which applies the identity function to its argument: $\lambda y \cdot \mathbf{id}\, y = \lambda y \cdot ((\lambda x \cdot x)\, y)$
- The same function, applied to the identity function: $(\lambda y \cdot \mathbf{id}\, y)\, \mathbf{id}$

(These three expressions actually denote the same function)

Examples

- The identity function: $\mathbf{id} = \lambda x \cdot x$
- The function which applies the identity function to its argument: $\lambda y \cdot \mathbf{id}\, y = \lambda y \cdot ((\lambda x \cdot x)\, y)$
- The same function, applied to the identity function: $(\lambda y \cdot \mathbf{id}\, y)\, \mathbf{id}$

(These three expressions actually denote the same function)

- The constant function which always returns the identity function: $\lambda y \cdot \mathbf{id}$

Examples

- The identity function: $\mathbf{id} = \lambda x \cdot x$
- The function which applies the identity function to its argument: $\lambda y \cdot \mathbf{id}\, y = \lambda y \cdot ((\lambda x \cdot x)\, y)$
- The same function, applied to the identity function: $(\lambda y \cdot \mathbf{id}\, y)\, \mathbf{id}$

(These three expressions actually denote the same function)

- The constant function which always returns the identity function: $\lambda y \cdot \mathbf{id}$

We can define functions of two arguments by encoding them with two λs: giving the first argument yields a function which waits for the second one.

Examples

- The identity function: $\mathbf{id} = \lambda x \cdot x$
- The function which applies the identity function to its argument: $\lambda y \cdot \mathbf{id}\, y = \lambda y \cdot ((\lambda x \cdot x)\, y)$
- The same function, applied to the identity function: $(\lambda y \cdot \mathbf{id}\, y)\, \mathbf{id}$

(These three expressions actually denote the same function)

- The constant function which always returns the identity function: $\lambda y \cdot \mathbf{id}$

We can define functions of two arguments by encoding them with two λs: giving the first argument yields a function which waits for the second one.

- The function which applies its first argument to its second argument: $\lambda x \cdot \lambda y \cdot (x\, y)$

Examples

- The identity function: $\mathbf{id} = \lambda x \cdot x$
- The function which applies the identity function to its argument: $\lambda y \cdot \mathbf{id}\, y = \lambda y \cdot ((\lambda x \cdot x)\, y)$
- The same function, applied to the identity function: $(\lambda y \cdot \mathbf{id}\, y)\, \mathbf{id}$

(These three expressions actually denote the same function)

- The constant function which always returns the identity function: $\lambda y \cdot \mathbf{id}$

We can define functions of two arguments by encoding them with two λs: giving the first argument yields a function which waits for the second one.

- The function which applies its first argument to its second argument: $\lambda x \cdot \lambda y \cdot (x\, y)$

Abbreviation: $\lambda x, y \cdot x\, y$

More examples

So, everything is a function. What operations do we know on functions?

More examples

So, everything is a function. What operations do we know on functions?

Composition: $(f \circ g)(x) = f(g(x))$

More examples

So, everything is a function. What operations do we know on functions?

Composition: $(f \circ g)(x) = f(g(x))$
In λ-calculus, the $\circ$ operator can be defined by:

$$\textbf{comp} = \lambda f, g \cdot \lambda x \cdot (f\,(g\,x))$$

$f \circ g$ is then written $\textbf{comp}\, f\, g$ as a λ-expression (prefix notation)

More examples

So, everything is a function. What operations do we know on functions?

Composition: $(f \circ g)(x) = f(g(x))$
In λ-calculus, the $\circ$ operator can be defined by:

$$\textbf{comp} = \lambda f, g \cdot \lambda x \cdot (f\,(g\,x))$$

$f \circ g$ is then written **comp** $f\,g$ as a λ-expression (prefix notation)

We can compose a function with itself ('square' it):

More examples

So, everything is a function. What operations do we know on functions?

Composition: $(f \circ g)(x) = f(g(x))$
In λ-calculus, the $\circ$ operator can be defined by:

$$\mathbf{comp} = \lambda f, g \cdot \lambda x \cdot (f\,(g\,x))$$

$f \circ g$ is then written $\mathbf{comp}\, f\, g$ as a λ-expression (prefix notation)

We can compose a function with itself (‘square’ it):

$$h^2 = h \circ h = \mathbf{comp}\, h\, h = (\lambda f, g \cdot \lambda x \cdot (f\,(g\,x)))\, h\, h$$

More examples

So, everything is a function. What operations do we know on functions?

Composition: $(f \circ g)(x) = f(g(x))$
In λ-calculus, the $\circ$ operator can be defined by:

$$\mathbf{comp} = \lambda f, g \cdot \lambda x \cdot (f\,(g\,x))$$

$f \circ g$ is then written $\mathbf{comp}\, f\, g$ as a λ-expression (prefix notation)

We can compose a function with itself ('square' it):

$$h^2 = h \circ h = \mathbf{comp}\, h\, h = (\lambda f, g \cdot \lambda x \cdot (f\,(g\,x)))\, h\, h$$

Applying is done by substituting h for both f and g: it yields

$$\lambda x \cdot (h\,(h\,x))$$

More examples

So, everything is a function. What operations do we know on functions?

Composition: $(f \circ g)(x) = f(g(x))$
In λ-calculus, the $\circ$ operator can be defined by:

$$\textbf{comp} = \lambda f, g \cdot \lambda x \cdot (f\,(g\,x))$$

$f \circ g$ is then written $\textbf{comp}\,f\,g$ as a λ-expression (prefix notation)

We can compose a function with itself ('square' it):

$$h^2 = h \circ h = \textbf{comp}\,h\,h = (\lambda f, g \cdot \lambda x \cdot (f\,(g\,x)))\,h\,h$$

Applying is done by substituting h for both f and g: it yields

$$\lambda x \cdot (h\,(h\,x))$$

Hence the squaring function $\mathbf{2} = \lambda h \cdot \lambda x \cdot (h\,(h\,x))$

A possible encoding of numbers

Everything is a function, but a lot of things can be represented by particular functions...

A possible encoding of numbers

Everything is a function, but a lot of things can be represented by particular functions...

How could we represent numbers?

A possible encoding of numbers

Everything is a function, but a lot of things can be represented by particular functions...

How could we represent numbers?

The number n could be the operation which transforms a function f into f^n, f composed with itself n times:

A possible encoding of numbers

Everything is a function, but a lot of things can be represented by particular functions...

How could we represent numbers?

The number n could be the operation which transforms a function f into f^n, f composed with itself n times:

0. $\lambda f \cdot \lambda x \cdot x$
1. $\lambda f \cdot \lambda x \cdot (f\, x)$
2. $\lambda f \cdot \lambda x \cdot (f\,(f\, x))$
3. $\lambda f \cdot \lambda x \cdot (f\,(f\,(f\, x)))$
4. $\lambda f \cdot \lambda x \cdot (f\,(f\,(f\,(f\, x))))$

Exercise

With this encoding of numbers, how would we write:

- The successor function **succ**, defined by **succ**$(n) = n + 1$?
- Addition (as a function of two parameters)?
- Multiplication?

If n and m are functions representing numbers, what does the expression $m\,n$ represent?

Exercise

With this encoding of numbers, how would we write:

- The successor function **succ**, defined by **succ**$(n) = n + 1$?

  $f^{n+1}(x) = f(f^n(x))$

- Addition (as a function of two parameters)?

- Multiplication?

  $f^{n\times m} = (f^n)^m$

If n and m are functions representing numbers, what does the expression $m\, n$ represent?

Testing whether a number is 0

Consider the function defined mathematically as follows:

$$\textbf{if-zero-then-else}(n, x, y) = \begin{cases} x & \text{if } n = 0 \\ y & \text{if } n > 0 \end{cases}$$

Can we define it in λ-calculus?

Testing whether a number is 0

Consider the function defined mathematically as follows:

$$\textbf{if-zero-then-else}(n, x, y) = \begin{cases} x & \text{if } n = 0 \\ y & \text{if } n > 0 \end{cases}$$

Can we define it in λ-calculus?

Idea: if g is a constant function, then $g \circ g = g$.

Testing whether a number is 0

Consider the function defined mathematically as follows:

$$\textbf{if-zero-then-else}(n, x, y) = \begin{cases} x & \text{if } n = 0 \\ y & \text{if } n > 0 \end{cases}$$

Can we define it in λ-calculus?

Idea: if g is a constant function, then $g \circ g = g$.
Thus, $g^n = g$ for any $n > 0$.

Testing whether a number is 0

Consider the function defined mathematically as follows:

$$\textbf{if-zero-then-else}(n, x, y) = \begin{cases} x & \text{if } n = 0 \\ y & \text{if } n > 0 \end{cases}$$

Can we define it in λ-calculus?

Idea: if g is a constant function, then $g \circ g = g$.
Thus, $g^n = g$ for any $n > 0$.
But $g^0 = \textbf{id}$.

Outline

Formal definitions

Let $\mathcal{V}$ be a countably infinite set of variables.
Expressions of the λ-calculus (also called λ-terms) are defined inductively as follows:

- Any variable $v \in \mathcal{V}$ is an expression;
- If v is a variable and *expr* is an expression, then $(\lambda v \cdot \mathit{expr})$ is an expression (called an abstraction);
- If expr_1 and expr_2 are expressions, then $(\mathit{expr}_1\ \mathit{expr}_2)$ is an expression (called an application).

(This corresponds to the terms generated by a very simple grammar.)

Formal definitions

Let $\mathcal{V}$ be a countably infinite set of variables.
Expressions of the λ-calculus (also called λ-terms) are defined inductively as follows:

- Any variable $v \in \mathcal{V}$ is an expression;
- If v is a variable and *expr* is an expression, then $(\lambda v \cdot expr)$ is an expression (called an abstraction);
- If $expr_1$ and $expr_2$ are expressions, then $(expr_1\ expr_2)$ is an expression (called an application).

(This corresponds to the terms generated by a very simple grammar.)

The reason to assume $\mathcal{V}$ is infinite is so that, for any expression *expr*, there always exists a variable $x \in \mathcal{V}$ which does not appear in *expr*.

Abbreviations

Omission of parentheses:

- Application has precedence to the left: $expr_1\, expr_2\, expr_3$ means $((expr_1\, expr_2)\, expr_3)$.
- Application has precedence over abstraction: $\lambda x \cdot expr_1\, expr_2$ means $(\lambda x \cdot (expr_1\, expr_2))$.

Multiple abstraction:

- $\lambda x, y, z \cdot expr$ means $(\lambda x \cdot (\lambda y \cdot (\lambda z \cdot expr)))$, etc.

Free and bound variables

The sets of free and bound variables of an expression *expr* are defined as follows:

Free and bound variables

The sets of free and bound variables of an expression *expr* are defined as follows:

Variable: If $expr = v$ then $FV(expr) = \{v\}$ and $BV(expr) = \emptyset$;

Free and bound variables

The sets of free and bound variables of an expression *expr* are defined as follows:

Variable: If $expr = v$ then $FV(expr) = \{v\}$ and $BV(expr) = \emptyset$;

Abstraction: If $expr = \lambda v \cdot expr'$ then $FV(expr) = FV(expr') \setminus \{v\}$ and $BV(expr) = BV(expr') \cup \{v\}$;

Free and bound variables

The sets of free and bound variables of an expression *expr* are defined as follows:

Variable: If $expr = v$ then $FV(expr) = \{v\}$ and $BV(expr) = \emptyset$;

Abstraction: If $expr = \lambda v \cdot expr'$ then $FV(expr) = FV(expr') \setminus \{v\}$ and $BV(expr) = BV(expr') \cup \{v\}$;

Application: If $expr = expr_1\ expr_2$ then $FV(expr) = FV(expr_1) \cup FV(expr_2)$ and $BV(expr) = BV(expr_1) \cup BV(expr_2)$.

Free and bound variables

The sets of free and bound variables of an expression *expr* are defined as follows:

Variable: If $expr = v$ then $FV(expr) = \{v\}$ and $BV(expr) = \emptyset$;

Abstraction: If $expr = \lambda v \cdot expr'$ then $FV(expr) = FV(expr') \setminus \{v\}$ and $BV(expr) = BV(expr') \cup \{v\}$;

Application: If $expr = expr_1\ expr_2$ then $FV(expr) = FV(expr_1) \cup FV(expr_2)$ and $BV(expr) = BV(expr_1) \cup BV(expr_2)$.

A closed expression is an expression with no free variables.

Free and bound variables

The sets of free and bound variables of an expression *expr* are defined as follows:

Variable: If $expr = v$ then $FV(expr) = \{v\}$ and $BV(expr) = \emptyset$;

Abstraction: If $expr = \lambda v \cdot expr'$ then $FV(expr) = FV(expr') \setminus \{v\}$ and $BV(expr) = BV(expr') \cup \{v\}$;

Application: If $expr = expr_1\ expr_2$ then
$FV(expr) = FV(expr_1) \cup FV(expr_2)$ and
$BV(expr) = BV(expr_1) \cup BV(expr_2)$.

A closed expression is an expression with no free variables.

As standard in mathematics, the names of bound variables are not significant: $\lambda x \cdot x$ and $\lambda y \cdot y$ are the same function.
In λ-calculus, two expressions which differ only by the name of their bound variables are called α-equivalent.

Free and bound variables

The sets of free and bound variables of an expression *expr* are defined as follows:

Variable: If $expr = v$ then $FV(expr) = \{v\}$ and $BV(expr) = \emptyset$;

Abstraction: If $expr = \lambda v \cdot expr'$ then $FV(expr) = FV(expr') \setminus \{v\}$ and $BV(expr) = BV(expr') \cup \{v\}$;

Application: If $expr = expr_1\ expr_2$ then $FV(expr) = FV(expr_1) \cup FV(expr_2)$ and $BV(expr) = BV(expr_1) \cup BV(expr_2)$.

A closed expression is an expression with no free variables.

As standard in mathematics, the names of bound variables are not significant: $\lambda x \cdot x$ and $\lambda y \cdot y$ are the same function.
In λ-calculus, two expressions which differ only by the name of their bound variables are called α-equivalent.

This strong equivalence will be denoted $\equiv$.

Reduction

The relation of reduction, $\longrightarrow$, describes how an expression evolves when an abstraction is applied to an argument.

Reduction

The relation of reduction, $\longrightarrow$, describes how an expression evolves when an abstraction is applied to an argument.

It is defined as follows:

$$(\lambda x \cdot \mathit{expr})\, \mathit{expr}' \longrightarrow [\mathit{expr}]\{x \mapsto \mathit{expr}'\};$$

Reduction

The relation of reduction, $\longrightarrow$, describes how an expression evolves when an abstraction is applied to an argument.

It is defined as follows:

- If bound variables of *expr* contain neither x nor any free variable of $expr'$, then $(\lambda x \cdot expr)\, expr' \longrightarrow [expr]\{x \mapsto expr'\}$;

Reduction

The relation of reduction, $\longrightarrow$, describes how an expression evolves when an abstraction is applied to an argument.

It is defined as follows:

- If bound variables of *expr* contain neither x nor any free variable of $expr'$, then $(\lambda x \cdot expr)\, expr' \longrightarrow [expr]\{x \mapsto expr'\}$;
- If $expr_1 \equiv expr'_1$ and $expr'_1 \longrightarrow expr_2$, then $expr_1 \longrightarrow expr_2$;

Reduction

The relation of reduction, $\longrightarrow$, describes how an expression evolves when an abstraction is applied to an argument.

It is defined as follows:

- If bound variables of *expr* contain neither x nor any free variable of $expr'$, then $(\lambda x \cdot expr)\, expr' \longrightarrow [expr]\{x \mapsto expr'\}$;
- If $expr_1 \equiv expr_1'$ and $expr_1' \longrightarrow expr_2$, then $expr_1 \longrightarrow expr_2$;
- If $expr_1$ contains a subexpression $expr_1'$ and $expr_1' \longrightarrow expr_2'$, let $expr_2$ be the result of replacing $expr_1'$ with $expr_2'$ in $expr_1$. Then: $expr_1 \longrightarrow expr_2$.

Example

Let $\mathbf{f} = (\lambda x \cdot (\lambda x \cdot x)\, x)\, \lambda x \cdot x$.

Example

Let $\mathbf{f} = (\lambda x \cdot (\lambda x \cdot x)\, x)\, \lambda x \cdot x$.

The first rule does not apply (variable conflict).
But we have $\mathbf{f} \equiv (\lambda x \cdot (\lambda y \cdot y)\, x)\, \lambda z \cdot z$.

Example

Let $\mathbf{f} = (\lambda x \cdot (\lambda x \cdot x)\, x)\, \lambda x \cdot x$.

The first rule does not apply (variable conflict).
But we have $\mathbf{f} \equiv (\lambda x \cdot (\lambda y \cdot y)\, x)\, \lambda z \cdot z$.

Thus, $\mathbf{f} \longrightarrow [(\lambda y \cdot y)\, x]\{x \mapsto \lambda z \cdot z\} = (\lambda y \cdot y)\, \lambda z \cdot z$ (second rule).

Example

Let $\mathbf{f} = (\lambda x \cdot (\lambda x \cdot x)\, x)\, \lambda x \cdot x$.

The first rule does not apply (variable conflict).
But we have $\mathbf{f} \equiv (\lambda x \cdot (\lambda y \cdot y)\, x)\, \lambda z \cdot z$.

Thus, $\mathbf{f} \longrightarrow [(\lambda y \cdot y)\, x]\{x \mapsto \lambda z \cdot z\} = (\lambda y \cdot y)\, \lambda z \cdot z$ (second rule).

But we also have $(\lambda y \cdot y)\, x \longrightarrow x$.
Hence, by third rule: $\mathbf{f} \longrightarrow (\lambda x \cdot x)\, \lambda z \cdot z$.

Example

Let $\mathbf{f} = (\lambda x \cdot (\lambda x \cdot x)\, x)\, \lambda x \cdot x$.

The first rule does not apply (variable conflict).
But we have $\mathbf{f} \equiv (\lambda x \cdot (\lambda y \cdot y)\, x)\, \lambda z \cdot z$.

Thus, $\mathbf{f} \longrightarrow [(\lambda y \cdot y)\, x]\{x \mapsto \lambda z \cdot z\} = (\lambda y \cdot y)\, \lambda z \cdot z$ (second rule).

But we also have $(\lambda y \cdot y)\, x \longrightarrow x$.
Hence, by third rule: $\mathbf{f} \longrightarrow (\lambda x \cdot x)\, \lambda z \cdot z$.

In general, $\longrightarrow$ is nondeterministic (we can have $expr \longrightarrow expr'$ and $expr \longrightarrow expr''$ with $expr' \not\equiv expr''$).

Multiple reduction, β-equivalence

Reduction steps can be chained.
We write $expr_1 \Longrightarrow expr_2$ if we can go from $expr_1$ to $expr_2$ by chaining one or more reduction steps.

Multiple reduction, β-equivalence

Reduction steps can be chained.
We write $expr_1 \Longrightarrow expr_2$ if we can go from $expr_1$ to $expr_2$ by chaining one or more reduction steps.

Let $expr_1$ and $expr_2$ be two expressions.
Suppose there exists $expr_3$ such that both $expr_1 \Longrightarrow expr_3$ and $expr_2 \Longrightarrow expr_3$.

Multiple reduction, β-equivalence

Reduction steps can be chained.
We write $expr_1 \Longrightarrow expr_2$ if we can go from $expr_1$ to $expr_2$ by chaining one or more reduction steps.

Let $expr_1$ and $expr_2$ be two expressions.
Suppose there exists $expr_3$ such that both $expr_1 \Longrightarrow expr_3$ and $expr_2 \Longrightarrow expr_3$.
We can then consider that $expr_1$ and $expr_2$ denote the same mathematical function.

Multiple reduction, β-equivalence

Reduction steps can be chained.
We write $expr_1 \Longrightarrow expr_2$ if we can go from $expr_1$ to $expr_2$ by chaining one or more reduction steps.

Let $expr_1$ and $expr_2$ be two expressions.
Suppose there exists $expr_3$ such that both $expr_1 \Longrightarrow expr_3$ and $expr_2 \Longrightarrow expr_3$.
We can then consider that $expr_1$ and $expr_2$ denote the same mathematical function.
This is called β-equivalence and written $\sim$.

Multiple reduction, β-equivalence

Reduction steps can be chained.
We write $expr_1 \Longrightarrow expr_2$ if we can go from $expr_1$ to $expr_2$ by chaining one or more reduction steps.

Let $expr_1$ and $expr_2$ be two expressions.
Suppose there exists $expr_3$ such that both $expr_1 \Longrightarrow expr_3$ and $expr_2 \Longrightarrow expr_3$.
We can then consider that $expr_1$ and $expr_2$ denote the same mathematical function.
This is called β-equivalence and written $\sim$.

Exercise:

- Check that, with our encoding of numbers and operations, $\mathbf{plus}\,\mathbf{2}\,\mathbf{2} \not\equiv \mathbf{mult}\,\mathbf{2}\,\mathbf{2} \not\equiv \mathbf{2}\,\mathbf{2}$, but $\mathbf{plus}\,\mathbf{2}\,\mathbf{2} \sim \mathbf{mult}\,\mathbf{2}\,\mathbf{2} \sim \mathbf{2}\,\mathbf{2}$.
- Are our different definitions of **plus** β-equivalent?

Outline

Informal overview and first examples

Formal definitions

Fundamental properties of λ-calculus
- Normal forms, unicity of normal forms
- Turing-completeness

More advanced examples

Reduction strategies

Typed λ-calculus

Normal forms

Some expressions cannot be reduced (if they contain no subexpression of the form $(\lambda x \cdot \mathit{expr})\, \mathit{expr}'$).

Normal forms

Some expressions cannot be reduced (if they contain no subexpression of the form $(\lambda x \cdot \mathit{expr})\, \mathit{expr}')$.
Examples: x; $x\, y$; $\lambda x \cdot x$.

Normal forms

Some expressions cannot be reduced (if they contain no subexpression of the form $(\lambda x \cdot expr)\, expr')$.
Examples: x; $x\, y$; $\lambda x \cdot x$.

Such expressions are called normal forms. We write $expr \not\to$ to indicate that $expr$ is a normal form.

Normal forms

Some expressions cannot be reduced (if they contain no subexpression of the form $(\lambda x \cdot \mathit{expr})\, \mathit{expr}'$).
Examples: x; $x\,y$; $\lambda x \cdot x$.

Such expressions are called normal forms. We write $\mathit{expr} \not\to$ to indicate that *expr* is a normal form.

Theorem (Church and Rosser): Let *expr* be an expression. Suppose $\mathit{expr} \Longrightarrow \mathit{expr}_1 \not\to$ and $\mathit{expr} \Longrightarrow \mathit{expr}_2 \not\to$.

Normal forms

Some expressions cannot be reduced (if they contain no subexpression of the form $(\lambda x \cdot \mathit{expr})\, \mathit{expr}'$).
Examples: x; $x\,y$; $\lambda x \cdot x$.

Such expressions are called normal forms. We write $\mathit{expr} \not\rightarrow$ to indicate that *expr* is a normal form.

Theorem (Church and Rosser): Let *expr* be an expression. Suppose $\mathit{expr} \Longrightarrow \mathit{expr}_1 \not\rightarrow$ and $\mathit{expr} \Longrightarrow \mathit{expr}_2 \not\rightarrow$.
Then $\mathit{expr}_1 \equiv \mathit{expr}_2$.

Normal forms

Some expressions cannot be reduced (if they contain no subexpression of the form $(\lambda x \cdot \mathit{expr})\, \mathit{expr}')$.
Examples: x; $x\, y$; $\lambda x \cdot x$.

Such expressions are called normal forms. We write $\mathit{expr} \not\rightarrow$ to indicate that *expr* is a normal form.

Theorem (Church and Rosser): Let *expr* be an expression. Suppose $\mathit{expr} \Longrightarrow \mathit{expr}_1 \not\rightarrow$ and $\mathit{expr} \Longrightarrow \mathit{expr}_2 \not\rightarrow$.
Then $\mathit{expr}_1 \equiv \mathit{expr}_2$.
In other words, even though $\longrightarrow$ is nondeterministic, applying it repeatedly until we reach a normal form always gives the same result (up to renaming).

Normal forms

Some expressions cannot be reduced (if they contain no subexpression of the form $(\lambda x \cdot expr)\, expr')$.
Examples: x; $x\, y$; $\lambda x \cdot x$.

Such expressions are called normal forms. We write $expr \not\rightarrow$ to indicate that $expr$ is a normal form.

Theorem (Church and Rosser): Let $expr$ be an expression. Suppose $expr \Longrightarrow expr_1 \not\rightarrow$ and $expr \Longrightarrow expr_2 \not\rightarrow$.
Then $expr_1 \equiv expr_2$.
In other words, even though $\longrightarrow$ is nondeterministic, applying it repeatedly until we reach a normal form always gives the same result (up to renaming).

This result is called the normal form of $expr$.

Normal forms

Some expressions cannot be reduced (if they contain no subexpression of the form $(\lambda x \cdot \mathit{expr})\, \mathit{expr}')$.
Examples: x; $x\, y$; $\lambda x \cdot x$.

Such expressions are called normal forms. We write $\mathit{expr} \not\rightarrow$ to indicate that expr is a normal form.

Theorem (Church and Rosser): Let expr be an expression. Suppose $\mathit{expr} \Longrightarrow \mathit{expr}_1 \not\rightarrow$ and $\mathit{expr} \Longrightarrow \mathit{expr}_2 \not\rightarrow$.
Then $\mathit{expr}_1 \equiv \mathit{expr}_2$.
In other words, even though $\longrightarrow$ is nondeterministic, applying it repeatedly until we reach a normal form always gives the same result (up to renaming).

This result is called the normal form of expr.

Exercise: calculate the normal form of **mult 2 2**.

Exercise

- Prove that all closed normal forms are abstractions, i. e. of the form $\lambda x \cdot \mathit{expr}$.

- Can you write an expression which has no normal form?

- Two functions are equal in the mathematical sense if they always both give the same result for a given argument (extensional equality).
  Verify that, even though $\mathbf{id} \not\sim \mathbf{1}$, these two functions are equal in that sense.

  This is called η-equivalence; we will not use it.

λ-definability

Let $f : \mathbb{N}^k \to \mathbb{N}$ be a mathematical function of k arguments from natural numbers to natural numbers. Let $\mathrm{Dom}(f)$ be the set of k-uples for which f is defined.
If n is a number, we write $\langle n \rangle$ the encoding of this number in λ-calculus.

λ-definability

Let $f : \mathbb{N}^k \to \mathbb{N}$ be a mathematical function of k arguments from natural numbers to natural numbers. Let $\mathrm{Dom}(f)$ be the set of k-uples for which f is defined.
If n is a number, we write $\langle n \rangle$ the encoding of this number in λ-calculus.

f is said to be λ-definable if there exists an expression $\mathbf{f}$ of the λ-calculus such that:

- For all $(n_1, \ldots, n_k) \in \mathrm{Dom}(f)$, we have:

$$\mathbf{f}\,\langle n_1 \rangle \ \ldots \ \langle n_k \rangle \Longrightarrow \langle f(n_1, \ldots, n_k) \rangle$$

- For all $(n_1, \ldots, n_k) \in \mathbb{N}^k \setminus \mathrm{Dom}(f)$, $\mathbf{f}\,\langle n_1 \rangle \ \ldots \ \langle n_k \rangle$ has no normal form.

λ-definability

Let $f\colon \mathbb{N}^k \to \mathbb{N}$ be a mathematical function of k arguments from natural numbers to natural numbers. Let $\mathrm{Dom}(f)$ be the set of k-uples for which f is defined.
If n is a number, we write $\langle n \rangle$ the encoding of this number in λ-calculus.

f is said to be λ-definable if there exists an expression **f** of the λ-calculus such that:

- For all $(n_1, \ldots, n_k) \in \mathrm{Dom}(f)$, we have:

$$\mathbf{f}\,\langle n_1 \rangle \ \ldots \ \langle n_k \rangle \Longrightarrow \langle f(n_1, \ldots, n_k) \rangle$$

- For all $(n_1, \ldots, n_k) \in \mathbb{N}^k \setminus \mathrm{Dom}(f)$, $\mathbf{f}\,\langle n_1 \rangle \ \ldots \ \langle n_k \rangle$ has no normal form.

Theorem (Turing): A function is λ-definable if and only if it is computable by a [Turing] machine.

Outline

Pairs

We can encode functions of several arguments. What about functions which return several results?

Pairs

We can encode functions of several arguments. What about functions which return several results?

Can we encode a pair of values into one single value?

Pairs

We can encode functions of several arguments. What about functions which return several results?

Can we encode a pair of values into one single value?

Wanted properties of a pair:

- We can build it from the two values;
- We can extract from it either the first or the second value.

Pairs

We can encode functions of several arguments. What about functions which return several results?

Can we encode a pair of values into one single value?

Wanted properties of a pair:

- We can build it from the two values;
- We can extract from it either the first or the second value.

pair $x\ y$ could be a function which returns either x or y depending on its argument (recall the test of slide 11).

Pairs

We can encode functions of several arguments. What about functions which return several results?

Can we encode a pair of values into one single value?

Wanted properties of a pair:

- We can build it from the two values;
- We can extract from it either the first or the second value.

pair $x\ y$ could be a function which returns either x or y depending on its argument (recall the test of slide 11).

Simplest way to do it: **pair** $= \lambda x, y \cdot \lambda z \cdot z\ x\ y$

Pairs

We can encode functions of several arguments. What about functions which return several results?

Can we encode a pair of values into one single value?

Wanted properties of a pair:

- We can build it from the two values;
- We can extract from it either the first or the second value.

pair $x\,y$ could be a function which returns either x or y depending on its argument (recall the test of slide 11).

Simplest way to do it: **pair** $= \lambda x, y \cdot \lambda z \cdot z\,x\,y$

The functions giving back either value from a pair are then:
fst $= \lambda p \cdot p\,\lambda x, y \cdot x$
snd $= \lambda p \cdot p\,\lambda x, y \cdot y$

The predecessor function

We now want to define a function returning $\langle n-1\rangle$ when given $\langle n\rangle$.

The predecessor function

We now want to define a function returning $\langle n-1 \rangle$ when given $\langle n \rangle$. In other words, given the operation 'apply a function n times', we want to build the operation 'apply a function $n-1$ times'.

The predecessor function

We now want to define a function returning $\langle n-1 \rangle$ when given $\langle n \rangle$. In other words, given the operation 'apply a function n times', we want to build the operation 'apply a function $n-1$ times'.

Idea: we apply n times a function g which applies f but also remembers the result of the previous iteration.

The predecessor function

We now want to define a function returning $\langle n-1 \rangle$ when given $\langle n \rangle$. In other words, given the operation 'apply a function n times', we want to build the operation 'apply a function $n-1$ times'.

Idea: we apply n times a function g which applies f but also remembers the result of the previous iteration.
Such a function could take and return a **pair**.

The predecessor function

We now want to define a function returning $\langle n-1\rangle$ when given $\langle n\rangle$. In other words, given the operation 'apply a function n times', we want to build the operation 'apply a function $n-1$ times'.

Idea: we apply n times a function g which applies f but also remembers the result of the previous iteration.
Such a function could take and return a **pair**.

For example:
if $g(cur, prev) = (f(cur), cur)$, then $g^n(x, y) = (f^n(x), f^{n-1}(x))$ (if $n > 0$).

Thus, $\textbf{snd} \circ g^n(x, y) = f^{n-1}(x)$.

The predecessor function

We now want to define a function returning $\langle n-1 \rangle$ when given $\langle n \rangle$. In other words, given the operation 'apply a function n times', we want to build the operation 'apply a function $n-1$ times'.

Idea: we apply n times a function g which applies f but also remembers the result of the previous iteration.
Such a function could take and return a **pair**.

For example:
if $g(cur, prev) = (f(cur), cur)$, then $g^n(x, y) = (f^n(x), f^{n-1}(x))$ (if $n > 0$).

Thus, $\mathbf{snd} \circ g^n(x, y) = f^{n-1}(x)$.

In λ-calculus: $\mathbf{g} = \lambda p \cdot \mathbf{pair}\,(f\,(\mathbf{fst}\,p))\,(\mathbf{fst}\,p)$

The predecessor function

We now want to define a function returning $\langle n-1\rangle$ when given $\langle n\rangle$.
In other words, given the operation 'apply a function n times', we want to build the operation 'apply a function $n-1$ times'.

Idea: we apply n times a function g which applies f but also remembers the result of the previous iteration.
Such a function could take and return a **pair**.

For example:
if $g(cur, prev) = (f(cur), cur)$, then $g^n(x, y) = (f^n(x), f^{n-1}(x))$ (if $n > 0$).

Thus, $\mathbf{snd} \circ g^n(x, y) = f^{n-1}(x)$.

In λ-calculus: $\mathbf{g} = \lambda p \cdot \mathbf{pair}\,(f\,(\mathbf{fst}\,p))\,(\mathbf{fst}\,p)$

And so we can write:

$$\mathbf{pred} = \lambda n, f, x \cdot \mathbf{snd}\,(n\,\mathbf{g}\,(\mathbf{pair}\,x\,x))$$

(In that example, $\mathbf{pred}\,\mathbf{0} \Longrightarrow \mathbf{0}$.)

Data structures

The encoding of pairs can be generalised to tuples of any size.

Structures with a fixed number of fields are just a variant of tuples.

Data structures

The encoding of pairs can be generalised to tuples of any size.

Structures with a fixed number of fields are just a variant of tuples.

But how to encode data of variable length?

For example, an optional value.

Data structures

The encoding of pairs can be generalised to tuples of any size.

Structures with a fixed number of fields are just a variant of tuples.

But how to encode data of variable length?

For example, an optional value.

Wanted properties of an option:

- There is a constant **none** representing an option with no value;

Data structures

The encoding of pairs can be generalised to tuples of any size.

Structures with a fixed number of fields are just a variant of tuples.

But how to encode data of variable length?

For example, an optional value.

Wanted properties of an option:

- There is a constant **none** representing an option with no value;
- There is a constructor **some** such that **some** x represents an option containing x;

Data structures

The encoding of pairs can be generalised to tuples of any size.

Structures with a fixed number of fields are just a variant of tuples.

But how to encode data of variable length?

For example, an optional value.

Wanted properties of an option:

- There is a constant **none** representing an option with no value;
- There is a constructor **some** such that **some** x represents an option containing x;
- When using an option, we can decide what to do depending whether the option contains a value or not.

Data structures

The encoding of pairs can be generalised to tuples of any size.

Structures with a fixed number of fields are just a variant of tuples.

But how to encode data of variable length?

For example, an optional value.

Wanted properties of an option:

- There is a constant **none** representing an option with no value;
- There is a constructor **some** such that **some** x represents an option containing x;
- When using an option, we can decide what to do depending whether the option contains a value or not.

  For example, writing **switch** *option case-none case-some* which returns *case-none* if the option is empty and applies *case-some* to the option's content otherwise.

Data structures

The encoding of pairs can be generalised to tuples of any size.

Structures with a fixed number of fields are just a variant of tuples.

But how to encode data of variable length?

For example, an optional value.

Wanted properties of an option:

- There is a constant **none** representing an option with no value;
- There is a constructor **some** such that **some** x represents an option containing x;
- When using an option, we can decide what to do depending whether the option contains a value or not.

  For example, writing **switch** *option case-none case-some* which returns *case-none* if the option is empty and applies *case-some* to the option's content otherwise.

One possibility: **none** and **some** x are functions of two parameters; these two parameters represent *case-none* and *case-some*. In that case, **switch** has nothing to do (but we can use it for clarity).

$\textbf{none} = \lambda n, s \cdot n$; $\textbf{some} = \lambda x, n, s \cdot s\,x$; $\textbf{switch} = \lambda o, n, s \cdot o\,n\,s$

Exercises

- Write a function **if-smaller-then-else** of 4 arguments m, n, x, y, which, assuming m and n encode numbers, returns x if $m \leqslant n$ and y otherwise.

- Propose a way to encode binary trees whose internal nodes are labelled with numbers.

Recursion

Structures like trees or lists are recursive (a tree has subtrees).

Recursion

Structures like trees or lists are recursive (a tree has subtrees).
Working with such structures calls for recursive functions.

Recursion

Structures like trees or lists are recursive (a tree has subtrees).
Working with such structures calls for recursive functions.

For example, inserting a new value in a binary search tree:

Recursion

Structures like trees or lists are recursive (a tree has subtrees).
Working with such structures calls for recursive functions.

For example, inserting a new value in a binary search tree:

- empty tree $\rightarrow$ return a new tree containing just the new value;

Recursion

Structures like trees or lists are recursive (a tree has subtrees). Working with such structures calls for recursive functions.

For example, inserting a new value in a binary search tree:

- empty tree $\rightarrow$ return a new tree containing just the new value;
- nonempty tree $\rightarrow$ compare the new value with the root label, and insert it into the left or right subtree (recursive call) depending on the result.

Recursion

Structures like trees or lists are recursive (a tree has subtrees).
Working with such structures calls for recursive functions.

For example, inserting a new value in a binary search tree:

- empty tree $\rightarrow$ return a new tree containing just the new value;
- nonempty tree $\rightarrow$ compare the new value with the root label, and insert it into the left or right subtree (recursive call) depending on the result.

Like this:

$$
\begin{aligned}
\textbf{insert} = \lambda t, v \cdot \textbf{switch}\, t \quad & (\textbf{tree}\, v\, \textbf{none}\, \textbf{none}) \\
& \lambda w, l, r \cdot \textbf{if-smaller-then-else}\, v\, w \\
& \qquad (\textbf{tree}\, w\, (\textbf{insert}\, l\, v)\, r) \\
& \qquad (\textbf{tree}\, w\, l\, (\textbf{insert}\, r\, v))
\end{aligned}
$$

Recursion

Structures like trees or lists are recursive (a tree has subtrees).
Working with such structures calls for recursive functions.

For example, inserting a new value in a binary search tree:

- empty tree $\rightarrow$ return a new tree containing just the new value;
- nonempty tree $\rightarrow$ compare the new value with the root label, and insert it into the left or right subtree (recursive call) depending on the result.

Like this:

$$
\begin{aligned}
\mathbf{insert} = \lambda t, v \cdot \mathbf{switch}\, t \quad & (\mathbf{tree}\, v\, \mathbf{none}\, \mathbf{none}) \\
& \lambda w, l, r \cdot \mathbf{if\text{-}smaller\text{-}then\text{-}else}\, v\, w \\
& \qquad (\mathbf{tree}\, w\, (\mathbf{insert}\, l\, v)\, r) \\
& \qquad (\mathbf{tree}\, w\, l\, (\mathbf{insert}\, r\, v))
\end{aligned}
$$

But in principle, a function in λ-calculus cannot call itself.

Haskell Curry's Y combinator

A workaround (among others) is Curry's Y combinator:

$$\mathbf{Y} = \lambda f \cdot (\lambda x \cdot f\,(x\,x))\,\lambda x \cdot f\,(x\,x)$$

Haskell Curry's Y combinator

A workaround (among others) is Curry's Y combinator:

$$\mathbf{Y} = \lambda f \cdot (\lambda x \cdot f\,(x\,x))\,\lambda x \cdot f\,(x\,x)$$

This expression has no normal form!

Haskell Curry's Y combinator

A workaround (among others) is Curry's Y combinator:

$$\mathbf{Y} = \lambda f \cdot (\lambda x \cdot f\,(x\,x))\,\lambda x \cdot f\,(x\,x)$$

This expression has no normal form!
We have $\mathbf{Y} \Longrightarrow \lambda f \cdot f\,(f\,(\ldots((\lambda x \cdot f\,(x\,x))\,\lambda x \cdot f\,(x\,x))\ldots))$ for any number of fs.

Haskell Curry's Y combinator

A workaround (among others) is Curry's Y combinator:

$$\mathbf{Y} = \lambda f \cdot (\lambda x \cdot f\,(x\,x))\,\lambda x \cdot f\,(x\,x)$$

This expression has no normal form!
We have $\mathbf{Y} \Longrightarrow \lambda f \cdot f\,(f\,(\ldots((\lambda x \cdot f\,(x\,x))\,\lambda x \cdot f\,(x\,x))\ldots))$ for any number of fs.

In fact, we have $\mathbf{Y}\,g \sim g(\mathbf{Y}g)$.

Haskell Curry's Y combinator

A workaround (among others) is Curry's Y combinator:

$$\mathbf{Y} = \lambda f \cdot (\lambda x \cdot f\,(x\,x))\,\lambda x \cdot f\,(x\,x)$$

This expression has no normal form!
We have $\mathbf{Y} \Longrightarrow \lambda f \cdot f\,(f\,(\ldots((\lambda x \cdot f\,(x\,x))\,\lambda x \cdot f\,(x\,x))\ldots))$ for any number of fs.

In fact, we have $\mathbf{Y}\,g \sim g(\mathbf{Y}g)$.

So, let:
$\mathbf{ins} = \lambda i, t, v \cdot \mathbf{switch}\; t\,(\mathbf{tree}\; v\;\mathbf{none}\;\mathbf{none})$
$\quad \lambda w, l, r \cdot \mathbf{if\text{-}smaller\text{-}then\text{-}else}\; v\; w\,(\mathbf{tree}\; w\,(i\; l\; v)\; r)\,(\mathbf{tree}\; w\; l\,(i\; r\; v))$
and let $\mathbf{insert} = \mathbf{Y}\,\mathbf{ins}$.

Haskell Curry's Y combinator

A workaround (among others) is Curry's Y combinator:

$$\mathbf{Y} = \lambda f \cdot (\lambda x \cdot f\,(x\,x))\,\lambda x \cdot f\,(x\,x)$$

This expression has no normal form!
We have $\mathbf{Y} \Longrightarrow \lambda f \cdot f\,(f\,(\ldots((\lambda x \cdot f\,(x\,x))\,\lambda x \cdot f\,(x\,x))\ldots))$ for any number of fs.

In fact, we have $\mathbf{Y}\,g \sim g(\mathbf{Y}g)$.

So, let:

$\mathbf{ins} = \lambda i, t, v \cdot \mathbf{switch}\; t\,(\mathbf{tree}\; v\; \mathbf{none}\; \mathbf{none})$
$\quad \lambda w, l, r \cdot \textbf{if-smaller-then-else}\; v\; w\,(\mathbf{tree}\; w\,(i\; l\; v)\; r)\,(\mathbf{tree}\; w\; l\,(i\; r\; v))$

and let $\mathbf{insert} = \mathbf{Y}\,\mathbf{ins}$.

Then $\mathbf{insert}\; t\; v \sim \mathbf{ins}\,\mathbf{insert}\; t\; v$. This is what we want!

Outline

Informal overview and first examples

Formal definitions

Fundamental properties of λ-calculus

More advanced examples

Reduction strategies
- Full reduction strategies
- Incomplete reductions: the programming language point of view

Typed λ-calculus

Infinite reduction sequences

By definition, any expression without a normal form must admit an infinite reduction sequence $expr_1 \longrightarrow expr_2 \longrightarrow \ldots \longrightarrow \ldots$

Infinite reduction sequences

By definition, any expression without a normal form must admit an infinite reduction sequence $expr_1 \longrightarrow expr_2 \longrightarrow \ldots \longrightarrow \ldots$

It is also possible for an expression with a normal form to admit such an infinite sequence, if:

Infinite reduction sequences

By definition, any expression without a normal form must admit an infinite reduction sequence $expr_1 \longrightarrow expr_2 \longrightarrow \ldots \longrightarrow \ldots$

It is also possible for an expression with a normal form to admit such an infinite sequence, if:

- some subexpression has no normal form, but
- this subexpression can disappear in a reduction.

Infinite reduction sequences

By definition, any expression without a normal form must admit an infinite reduction sequence $expr_1 \longrightarrow expr_2 \longrightarrow \ldots \longrightarrow \ldots$

It is also possible for an expression with a normal form to admit such an infinite sequence, if:

- some subexpression has no normal form, but
- this subexpression can disappear in a reduction.

Typical example:

Infinite reduction sequences

By definition, any expression without a normal form must admit an infinite reduction sequence $expr_1 \longrightarrow expr_2 \longrightarrow \ldots \longrightarrow \ldots$

It is also possible for an expression with a normal form to admit such an infinite sequence, if:

- some subexpression has no normal form, but
- this subexpression can disappear in a reduction.

Typical example: $(\lambda x \cdot \mathbf{id})\,((\lambda x \cdot x\,x)\,(\lambda x \cdot x\,x))$

Normal order of reduction

The normal order of reduction means that, when several reductions of *expr* are possible, we always choose the one which involves the leftmost possible occurrence of λ.

Example: consider $\lambda x \cdot (\lambda y \cdot (\lambda z \cdot z)\, y)\, ((\lambda t \cdot t)\, x)$:

Normal order of reduction

The normal order of reduction means that, when several reductions of *expr* are possible, we always choose the one which involves the leftmost possible occurrence of λ.

Example: consider $\lambda x \cdot (\lambda y \cdot (\lambda z \cdot z)\, y)\, ((\lambda t \cdot t)\, x)$:

- There are three possibilities of reduction: one involving λy, one involving λz, and one involving λt.

Normal order of reduction

The normal order of reduction means that, when several reductions of *expr* are possible, we always choose the one which involves the leftmost possible occurrence of λ.

Example: consider $\lambda x \cdot (\lambda y \cdot (\lambda z \cdot z)\, y)\, ((\lambda t \cdot t)\, x)$:

- There are three possibilities of reduction: one involving λy, one involving λz, and one involving λt.
- The leftmost one involves λy, this is the one we apply first.

Normal order of reduction

The normal order of reduction means that, when several reductions of *expr* are possible, we always choose the one which involves the leftmost possible occurrence of λ.

Example: consider $\lambda x \cdot (\lambda y \cdot (\lambda z \cdot z)\, y)\, ((\lambda t \cdot t)\, x)$:

- There are three possibilities of reduction: one involving λy, one involving λz, and one involving λt.
- The leftmost one involves λy, this is the one we apply first.

Theorem: Applying the normal order of reduction always reaches the normal form, if a normal form exists.

Normal order of reduction

The normal order of reduction means that, when several reductions of *expr* are possible, we always choose the one which involves the leftmost possible occurrence of λ.

Example: consider $\lambda x \cdot (\lambda y \cdot (\lambda z \cdot z)\, y)\, ((\lambda t \cdot t)\, x)$:

- There are three possibilities of reduction: one involving λy, one involving λz, and one involving λt.
- The leftmost one involves λy, this is the one we apply first.

Theorem: Applying the normal order of reduction always reaches the normal form, if a normal form exists.

Intuitively: We always apply a function before looking inside either its body or its arguments.

Normal order of reduction

The normal order of reduction means that, when several reductions of *expr* are possible, we always choose the one which involves the leftmost possible occurrence of λ.

Example: consider $\lambda x \cdot (\lambda y \cdot (\lambda z \cdot z)\, y)\, ((\lambda t \cdot t)\, x)$:

- There are three possibilities of reduction: one involving λy, one involving λz, and one involving λt.
- The leftmost one involves λy, this is the one we apply first.

Theorem: Applying the normal order of reduction always reaches the normal form, if a normal form exists.

Intuitively: We always apply a function before looking inside either its body or its arguments.
This way, if a subexpression with no normal form can get discarded, we make sure this happens before we try to reduce it.

Normal order of reduction

The normal order of reduction means that, when several reductions of *expr* are possible, we always choose the one which involves the leftmost possible occurrence of λ.

Example: consider $\lambda x \cdot (\lambda y \cdot (\lambda z \cdot z)\, y)\, ((\lambda t \cdot t)\, x)$:

- There are three possibilities of reduction: one involving λy, one involving λz, and one involving λt.
- The leftmost one involves λy, this is the one we apply first.

Theorem: Applying the normal order of reduction always reaches the normal form, if a normal form exists.

Intuitively: We always apply a function before looking inside either its body or its arguments.
This way, if a subexpression with no normal form can get discarded, we make sure this happens before we try to reduce it.

Exercise: reduce $\mathbf{Y}\,\mathbf{0}$ using the normal order of reduction.

Applicative order of reduction

On the opposite, the applicative strategy restricts the reduction rule $(\lambda x \cdot \mathit{expr})\, \mathit{expr}' \longrightarrow [\mathit{expr}]\{x \mapsto \mathit{expr}'\}$ by only allowing it if expr' is in normal form.

Applicative order of reduction

On the opposite, the applicative strategy restricts the reduction rule $(\lambda x \cdot \mathit{expr})\, \mathit{expr'} \longrightarrow [\mathit{expr}]\{x \mapsto \mathit{expr'}\}$ by only allowing it if $\mathit{expr'}$ is in normal form.

In other words, we must always reduce the argument of a function fully before applying that function.

Applicative order of reduction

On the opposite, the applicative strategy restricts the reduction rule $(\lambda x \cdot \mathit{expr})\, \mathit{expr}' \longrightarrow [\mathit{expr}]\{x \mapsto \mathit{expr}'\}$ by only allowing it if expr' is in normal form.

In other words, we must always reduce the argument of a function fully before applying that function.

This strategy does not completely enforce the reduction order, but can be implemented for example by always starting with the rightmost λ possible.

Applicative order of reduction

On the opposite, the applicative strategy restricts the reduction rule $(\lambda x \cdot \mathit{expr})\, \mathit{expr}' \longrightarrow [\mathit{expr}]\{x \mapsto \mathit{expr}'\}$ by only allowing it if expr' is in normal form.

In other words, we must always reduce the argument of a function fully before applying that function.

This strategy does not completely enforce the reduction order, but can be implemented for example by always starting with the rightmost λ possible.

Theorem: Reducing an expression *expr* with the applicative strategy reaches a normal form only if every subexpression of *expr* has a normal form.

Applicative order of reduction

On the opposite, the applicative strategy restricts the reduction rule $(\lambda x \cdot \mathit{expr})\, \mathit{expr}' \longrightarrow [\mathit{expr}]\{x \mapsto \mathit{expr}'\}$ by only allowing it if expr' is in normal form.

In other words, we must always reduce the argument of a function fully before applying that function.

This strategy does not completely enforce the reduction order, but can be implemented for example by always starting with the rightmost λ possible.

Theorem: Reducing an expression *expr* with the applicative strategy reaches a normal form only if every subexpression of *expr* has a normal form.

This is useful if we consider the meaning of an expression to be its normal form, and we do not want a meaningful expression to have meaningless parts — this corresponds to Church's original idea.

The programming language point of view

If we want to use functions like **Y**, saying that the 'value' of an expression is its normal form is too restrictive (even **insert** has no normal form, but it is an interesting function).

The programming language point of view

If we want to use functions like **Y**, saying that the ‘value’ of an expression is its normal form is too restrictive (even **insert** has no normal form, but it is an interesting function).

If we see λ-calculus as a programming language, we can consider the body of an abstraction as ‘instructions’ for computing the result.

The programming language point of view

If we want to use functions like **Y**, saying that the 'value' of an expression is its normal form is too restrictive (even **insert** has no normal form, but it is an interesting function).

If we see λ-calculus as a programming language, we can consider the body of an abstraction as 'instructions' for computing the result.
In that 'operational' view, instructions inside the body are followed only once an argument has been given.

The programming language point of view

If we want to use functions like **Y**, saying that the 'value' of an expression is its normal form is too restrictive (even **insert** has no normal form, but it is an interesting function).

If we see λ-calculus as a programming language, we can consider the body of an abstraction as 'instructions' for computing the result.
In that 'operational' view, instructions inside the body are followed only once an argument has been given.

In programming-language strategy, reductions never occur inside abstractions.
Abstractions are called values, and reduction stops as soon as it reaches a value. This is called evaluating the expression.

The programming language point of view

If we want to use functions like **Y**, saying that the 'value' of an expression is its normal form is too restrictive (even **insert** has no normal form, but it is an interesting function).

If we see λ-calculus as a programming language, we can consider the body of an abstraction as 'instructions' for computing the result.
In that 'operational' view, instructions inside the body are followed only once an argument has been given.

In programming-language strategy, reductions never occur inside abstractions.
Abstractions are called values, and reduction stops as soon as it reaches a value. This is called evaluating the expression.

Drawback: Apparently different values may in fact be β-equivalent.

Strict or lazy evaluation

The call-by-name evaluation strategy corresponds to normal order. It always terminates if terminating is possible.

The strict or call-by-value evaluation strategy corresponds to applicative order. It always evaluates arguments before applying a function, which may prevent terminating.

Strict or lazy evaluation

The call-by-name evaluation strategy corresponds to normal order.
It always terminates if terminating is possible.

Call-by-name avoids evaluating an argument which will not be used.
But it may lead to evaluating the same argument more than once.
A variant is call-by-need or lazy evaluation: occurrences of the argument variable are replaced with a pointer to the unevaluated argument, and once it is evaluated they all point to the result.

The strict or call-by-value evaluation strategy corresponds to applicative order. It always evaluates arguments before applying a function, which may prevent terminating.

Features of strict vs. lazy evaluation

In pure λ-calculus, evaluation order does not matter.
However, most programming languages allow control instructions (typically I/O: write to a file etc.) in the bodies of functions. In that case, evaluation order can matter.

Features of strict vs. lazy evaluation

In pure λ-calculus, evaluation order does not matter.
However, most programming languages allow control instructions (typically I/O: write to a file etc.) in the bodies of functions. In that case, evaluation order can matter.

'Impure' λ-based languages use strict order (easier to predict).

Features of strict vs. lazy evaluation

In pure λ-calculus, evaluation order does not matter.

However, most programming languages allow control instructions (typically I/O: write to a file etc.) in the bodies of functions. In that case, evaluation order can matter.

'Impure' λ-based languages use strict order (easier to predict).

In strict evaluation, unlike in complete applicative-order reduction, it is still possible to postpone a reduction for later by wrapping it in a function.

Features of strict vs. lazy evaluation

In pure λ-calculus, evaluation order does not matter.
However, most programming languages allow control instructions (typically I/O: write to a file etc.) in the bodies of functions. In that case, evaluation order can matter.

'Impure' λ-based languages use strict order (easier to predict).

In strict evaluation, unlike in complete applicative-order reduction, it is still possible to postpone a reduction for later by wrapping it in a function.

For example: $\mathbf{if\text{-}zero\text{-}then\text{-}else}\,\mathbf{0}\,\mathbf{0}\,((\lambda x \cdot x\,x)\,(\lambda x \cdot x\,x))$
does not terminate.

But $(\mathbf{if\text{-}zero\text{-}then\text{-}else}\,\mathbf{0}\,(\lambda y \cdot \mathbf{0})\,\lambda y \cdot\,(\lambda x \cdot x\,x)\,(\lambda x \cdot x\,x))\,\mathbf{id}$
does terminate.

Features of strict vs. lazy evaluation

In pure λ-calculus, evaluation order does not matter.
However, most programming languages allow control instructions (typically I/O: write to a file etc.) in the bodies of functions. In that case, evaluation order can matter.

'Impure' λ-based languages use strict order (easier to predict).

In strict evaluation, unlike in complete applicative-order reduction, it is still possible to postpone a reduction for later by wrapping it in a function.

For example: $\textbf{if-zero-then-else}\,\mathbf{0}\,\mathbf{0}\,((\lambda x \cdot x\,x)\,(\lambda x \cdot x\,x))$ does not terminate.

But $(\textbf{if-zero-then-else}\,\mathbf{0}\,(\lambda y \cdot \mathbf{0})\,\lambda y \cdot\, (\lambda x \cdot x\,x)\,(\lambda x \cdot x\,x))\,\textbf{id}$ does terminate.

Similarly, $\mathbf{Y} = \lambda f \cdot (\lambda x \cdot f\,(x\,x))\,\lambda x \cdot f\,(x\,x)$ can be replaced with $\mathbf{Z} = \lambda f \cdot (\lambda x \cdot f\,(\lambda y \cdot x\,x\,y))\,\lambda x \cdot f\,(\lambda y \cdot x\,x\,y)$.

Features of strict vs. lazy evaluation

In pure λ-calculus, evaluation order does not matter.
However, most programming languages allow control instructions (typically I/O: write to a file etc.) in the bodies of functions. In that case, evaluation order can matter.

'Impure' λ-based languages use strict order (easier to predict).

In strict evaluation, unlike in complete applicative-order reduction, it is still possible to postpone a reduction for later by wrapping it in a function.

For example: $\textbf{if-zero-then-else}\,\mathbf{0}\,\mathbf{0}\,((\lambda x \cdot x\,x)\,(\lambda x \cdot x\,x))$ does not terminate.

But $(\textbf{if-zero-then-else}\,\mathbf{0}\,(\lambda y \cdot \mathbf{0})\,\lambda y \cdot\, (\lambda x \cdot x\,x)\,(\lambda x \cdot x\,x))\,\textbf{id}$ does terminate.

Similarly, $\mathbf{Y} = \lambda f \cdot (\lambda x \cdot f\,(x\,x))\,\lambda x \cdot f\,(x\,x)$ can be replaced with $\mathbf{Z} = \lambda f \cdot (\lambda x \cdot f\,(\lambda y \cdot x\,x\,y))\,\lambda x \cdot f\,(\lambda y \cdot x\,x\,y)$.

In practice, strict languages make exceptions for some special functions (if, and, or, switch. . .) which are always evaluated lazily. And they allow defining recursive functions directly.

λ-calculus in programming languages: a few dates

- λ-calculus is defined in the 1930s.

λ-calculus in programming languages: a few dates

- λ-calculus is defined in the 1930s.
- Digital programmable computers appear in the 1940s. They are first programmed with mechanical switches, then in machine language, then in assembly language.

λ-calculus in programming languages: a few dates

- λ-calculus is defined in the 1930s.
- Digital programmable computers appear in the 1940s. They are first programmed with mechanical switches, then in machine language, then in assembly language.
- Higher-level programming is developed in the 1950s. First widely-used languages: Fortran (1957); Lisp (1958); Algol (1958); Cobol (1959).

λ-calculus in programming languages: a few dates

- λ-calculus is defined in the 1930s.
- Digital programmable computers appear in the 1940s. They are first programmed with mechanical switches, then in machine language, then in assembly language.
- Higher-level programming is developed in the 1950s. First widely-used languages: Fortran (1957); Lisp (1958); Algol (1958); Cobol (1959).
- Lisp introduces the `lambda` keyword. But it does not implement substitution properly.

λ-calculus in programming languages: a few dates

- λ-calculus is defined in the 1930s.
- Digital programmable computers appear in the 1940s. They are first programmed with mechanical switches, then in machine language, then in assembly language.
- Higher-level programming is developed in the 1950s. First widely-used languages: Fortran (1957); Lisp (1958); Algol (1958); Cobol (1959).
- Lisp introduces the `lambda` keyword. But it does not implement substitution properly.
- In the 1970s, functional programming starts developing. Scheme is a Lisp variant following λ-calculus semantics. ML (ancestor of SML, OCaml, F#) implements a typed λ-calculus. Both are impure and strict.

λ-calculus in programming languages: a few dates

- λ-calculus is defined in the 1930s.
- Digital programmable computers appear in the 1940s. They are first programmed with mechanical switches, then in machine language, then in assembly language.
- Higher-level programming is developed in the 1950s. First widely-used languages: Fortran (1957); Lisp (1958); Algol (1958); Cobol (1959).
- Lisp introduces the `lambda` keyword. But it does not implement substitution properly.
- In the 1970s, functional programming starts developing. Scheme is a Lisp variant following λ-calculus semantics. ML (ancestor of SML, OCaml, F#) implements a typed λ-calculus. Both are impure and strict.
- Miranda (1985) implements lazy evaluation.

λ-calculus in programming languages: a few dates

- λ-calculus is defined in the 1930s.
- Digital programmable computers appear in the 1940s. They are first programmed with mechanical switches, then in machine language, then in assembly language.
- Higher-level programming is developed in the 1950s. First widely-used languages: Fortran (1957); Lisp (1958); Algol (1958); Cobol (1959).
- Lisp introduces the `lambda` keyword. But it does not implement substitution properly.
- In the 1970s, functional programming starts developing. Scheme is a Lisp variant following λ-calculus semantics. ML (ancestor of SML, OCaml, F#) implements a typed λ-calculus. Both are impure and strict.
- Miranda (1985) implements lazy evaluation.
- 1990: first version of Haskell (named after Haskell Curry), now the reference lazy functional language, which features a sophisticated type system.

Outline

Base values and type errors

In theoretical λ-calculus, basic values like numbers are represented by particular functions.
In practice, on a computer, better-suited internal representations are used for data, and the syntax is extended to allow literal constants in addition to variables.

Base values and type errors

In theoretical λ-calculus, basic values like numbers are represented by particular functions.
In practice, on a computer, better-suited internal representations are used for data, and the syntax is extended to allow literal constants in addition to variables.

But then, applying a number to an argument is not possible.

Base values and type errors

In theoretical λ-calculus, basic values like numbers are represented by particular functions.
In practice, on a computer, better-suited internal representations are used for data, and the syntax is extended to allow literal constants in addition to variables.

But then, applying a number to an argument is not possible.

Lisp variants (like Scheme), or languages like Python, use dynamic typing to deal with that. If at some point a number should be applied to an argument, the program stops with an error.

Base values and type errors

In theoretical λ-calculus, basic values like numbers are represented by particular functions.
In practice, on a computer, better-suited internal representations are used for data, and the syntax is extended to allow literal constants in addition to variables.

But then, applying a number to an argument is not possible.

Lisp variants (like Scheme), or languages like Python, use dynamic typing to deal with that. If at some point a number should be applied to an argument, the program stops with an error.

But λ-calculus, as a formal language, is well suited to analyzing expressions statically, before reduction, to make sure such type errors cannot happen.

The simply-typed λ-calculus

Originally defined by Church to filter out ‘paradoxical’ expressions from λ-calculus, it predates computers.

The simply-typed λ-calculus

Originally defined by Church to filter out 'paradoxical' expressions from λ-calculus, it predates computers.

Principle: we start from a certain set $\mathcal{B}$ of base types and extend the syntax of expressions to allow literal constants, distinct from variables. Each literal has one of the base types.

The simply-typed λ-calculus

Originally defined by Church to filter out ‘paradoxical’ expressions from λ-calculus, it predates computers.

Principle: we start from a certain set $\mathcal{B}$ of base types and extend the syntax of expressions to allow literal constants, distinct from variables. Each literal has one of the base types.

Then types τ are defined inductively:

- Base types are types, representing the various literals;
- If τ_1 and τ_2 are types, $\tau_1 \to \tau_2$ is a type, representing the functions which take a parameter of type τ_1 and return a result of type τ_2.

The simply-typed λ-calculus

Originally defined by Church to filter out ‘paradoxical’ expressions from λ-calculus, it predates computers.

Principle: we start from a certain set $\mathcal{B}$ of base types and extend the syntax of expressions to allow literal constants, distinct from variables. Each literal has one of the base types.

Then types τ are defined inductively:

- Base types are types, representing the various literals;
- If τ_1 and τ_2 are types, $\tau_1 \to \tau_2$ is a type, representing the functions which take a parameter of type τ_1 and return a result of type τ_2.

Example: **plus** has type $\mathit{int} \to (\mathit{int} \to \mathit{int})$

The simply-typed λ-calculus

Originally defined by Church to filter out 'paradoxical' expressions from λ-calculus, it predates computers.

Principle: we start from a certain set $\mathcal{B}$ of base types and extend the syntax of expressions to allow literal constants, distinct from variables. Each literal has one of the base types.

Then types τ are defined inductively:

- Base types are types, representing the various literals;
- If τ_1 and τ_2 are types, $\tau_1 \to \tau_2$ is a type, representing the functions which take a parameter of type τ_1 and return a result of type τ_2.

Example: **plus** has type $int \to (int \to int)$
Commonly, $\to$ is taken to have precedence on the right, so we can write this type $int \to int \to int$.

The simply-typed λ-calculus (continued)

The syntax of abstractions becomes $\lambda x : \tau \cdot expr$ where τ is a type.

The simply-typed λ-calculus (continued)

The syntax of abstractions becomes $\lambda x : \tau \cdot \mathit{expr}$ where τ is a type.

Types of all expressions can then be computed using typing rules.

The simply-typed λ-calculus (continued)

The syntax of abstractions becomes $\lambda x : \tau \cdot \mathit{expr}$ where τ is a type.

Types of all expressions can then be computed using typing rules.

Typing rules look like: $\dfrac{\text{premise ('if' part)}}{\text{conclusion ('then' part)}}$.

The simply-typed λ-calculus (continued)

The syntax of abstractions becomes $\lambda x : \tau \cdot \mathit{expr}$ where τ is a type.

Types of all expressions can then be computed using typing rules.

Typing rules look like: $\dfrac{\text{premise ('if' part)}}{\text{conclusion ('then' part)}}$.

They use typing environments Γ associating *variables* to *types*, and typing judgements $\Gamma \vdash \mathit{expr} : \tau$ indicating that: 'if the variables have the types specified in Γ, then *expr* has the type τ'.

The simply-typed λ-calculus (continued)

The syntax of abstractions becomes $\lambda x : \tau \cdot expr$ where τ is a type.

Types of all expressions can then be computed using typing rules.

Typing rules look like: $\dfrac{\text{premise ('if' part)}}{\text{conclusion ('then' part)}}$.

They use typing environments Γ associating *variables* to *types*, and typing judgements $\Gamma \vdash expr : \tau$ indicating that: 'if the variables have the types specified in Γ, then *expr* has the type τ'.

And the rules of simply-typed λ-calculus are:

variable

$$\frac{\Gamma(x) = \tau}{\Gamma \vdash x : \tau}$$

literal

$$\frac{expr \text{ is a literal of type } \tau}{\Gamma \vdash expr : \tau}$$

The simply-typed λ-calculus (continued)

The syntax of abstractions becomes $\lambda x : \tau \cdot \mathit{expr}$ where τ is a type.

Types of all expressions can then be computed using typing rules.

Typing rules look like: $\dfrac{\text{premise ('if' part)}}{\text{conclusion ('then' part)}}$.

They use typing environments Γ associating *variables* to *types*, and typing judgements $\Gamma \vdash \mathit{expr} : \tau$ indicating that: 'if the variables have the types specified in Γ, then *expr* has the type τ'.

And the rules of simply-typed λ-calculus are:

variable

$$\frac{\Gamma(x) = \tau}{\Gamma \vdash x : \tau}$$

literal

$$\frac{\mathit{expr} \text{ is a literal of type } \tau}{\Gamma \vdash \mathit{expr} : \tau}$$

application

$$\frac{\Gamma \vdash \mathit{expr}_1 : \tau \to \tau' \qquad \Gamma \vdash \mathit{expr}_2 : \tau}{\Gamma \vdash \mathit{expr}_1\ \mathit{expr}_2 : \tau'}$$

The simply-typed λ-calculus (continued)

The syntax of abstractions becomes $\lambda x : \tau \cdot \mathit{expr}$ where τ is a type.

Types of all expressions can then be computed using typing rules.

Typing rules look like: $\dfrac{\text{premise ('if' part)}}{\text{conclusion ('then' part)}}$.

They use typing environments Γ associating *variables* to *types*, and typing judgements $\Gamma \vdash \mathit{expr} : \tau$ indicating that: 'if the variables have the types specified in Γ, then *expr* has the type τ'.

And the rules of simply-typed λ-calculus are:

variable
$$\frac{\Gamma(x) = \tau}{\Gamma \vdash x : \tau}$$

literal
$$\frac{\mathit{expr} \text{ is a literal of type } \tau}{\Gamma \vdash \mathit{expr} : \tau}$$

application
$$\frac{\Gamma \vdash \mathit{expr}_1 : \tau \to \tau' \qquad \Gamma \vdash \mathit{expr}_2 : \tau}{\Gamma \vdash \mathit{expr}_1\ \mathit{expr}_2 : \tau'}$$

abstraction
$$\frac{\Gamma, x : \tau \vdash \mathit{expr} : \tau'}{\Gamma \vdash \lambda x : \tau \cdot \mathit{expr} : \tau \to \tau'}$$

The simply-typed λ-calculus (continued)

The syntax of abstractions becomes $\lambda x : \tau \cdot \mathit{expr}$ where τ is a type.

Types of all expressions can then be computed using typing rules.

Typing rules look like: $\dfrac{\text{premise ('if' part)}}{\text{conclusion ('then' part)}}$.

They use typing environments Γ associating *variables* to *types*, and typing judgements $\Gamma \vdash \mathit{expr} : \tau$ indicating that: 'if the variables have the types specified in Γ, then *expr* has the type τ'.

And the rules of simply-typed λ-calculus are:

variable
$$\frac{\Gamma(x) = \tau}{\Gamma \vdash x : \tau}$$

literal
$$\frac{\mathit{expr} \text{ is a literal of type } \tau}{\Gamma \vdash \mathit{expr} : \tau}$$

application
$$\frac{\Gamma \vdash \mathit{expr}_1 : \tau \to \tau' \qquad \Gamma \vdash \mathit{expr}_2 : \tau}{\Gamma \vdash \mathit{expr}_1\ \mathit{expr}_2 : \tau'}$$

abstraction
$$\frac{\Gamma, x : \tau \vdash \mathit{expr} : \tau'}{\Gamma \vdash \lambda x : \tau \cdot \mathit{expr} : \tau \to \tau'}$$

Closed expressions are typed in empty environments.

In that case, we write $\vdash \mathit{expr} : \tau$.

Features of the simply-typed λ-calculus

Some expressions cannot be typed, in particular applications when the argument's type does not match the expected type: this corresponds to static type errors.

Features of the simply-typed λ-calculus

Some expressions cannot be typed, in particular applications when the argument's type does not match the expected type: this corresponds to static type errors.

Type preservation: if $\Gamma \vdash expr : \tau$ and $expr \longrightarrow expr'$, then $\Gamma \vdash expr' : \tau$.

Features of the simply-typed λ-calculus

Some expressions cannot be typed, in particular applications when the argument's type does not match the expected type: this corresponds to static type errors.

Type preservation: if $\Gamma \vdash \mathit{expr} : \tau$ and $\mathit{expr} \longrightarrow \mathit{expr}'$, then $\Gamma \vdash \mathit{expr}' : \tau$.
This is expected of any static type system.

Features of the simply-typed λ-calculus

Some expressions cannot be typed, in particular applications when the argument's type does not match the expected type: this corresponds to static type errors.

Type preservation: if $\Gamma \vdash expr : \tau$ and $expr \longrightarrow expr'$, then $\Gamma \vdash expr' : \tau$.
This is expected of any static type system.

Strong normalisation: if $\Gamma \vdash expr : \tau$, then there exists a normal form $expr'$ such that $expr \Longrightarrow expr'$ (and $\Gamma \vdash expr' : \tau$).

Features of the simply-typed λ-calculus

Some expressions cannot be typed, in particular applications when the argument's type does not match the expected type: this corresponds to static type errors.

Type preservation: if $\Gamma \vdash \mathit{expr} : \tau$ and $\mathit{expr} \longrightarrow \mathit{expr}'$, then $\Gamma \vdash \mathit{expr}' : \tau$.
This is expected of any static type system.

Strong normalisation: if $\Gamma \vdash \mathit{expr} : \tau$, then there exists a normal form expr' such that $\mathit{expr} \Longrightarrow \mathit{expr}'$ (and $\Gamma \vdash \mathit{expr}' : \tau$).
This is unusual and specific to this type system.

Features of the simply-typed λ-calculus

Some expressions cannot be typed, in particular applications when the argument's type does not match the expected type: this corresponds to static type errors.

Type preservation: if $\Gamma \vdash \mathit{expr} : \tau$ and $\mathit{expr} \longrightarrow \mathit{expr}'$, then $\Gamma \vdash \mathit{expr}' : \tau$.
This is expected of any static type system.

Strong normalisation: if $\Gamma \vdash \mathit{expr} : \tau$, then there exists a normal form expr' such that $\mathit{expr} \Longrightarrow \mathit{expr}'$ (and $\Gamma \vdash \mathit{expr}' : \tau$).
This is unusual and specific to this type system.

Since the type of an expression can be easily computed by an algorithm, and the evaluation of any typable expression always terminates, it means the simply-typed λ-calculus is not Turing-complete anymore.

Features of the simply-typed λ-calculus

Some expressions cannot be typed, in particular applications when the argument's type does not match the expected type: this corresponds to static type errors.

Type preservation: if $\Gamma \vdash \mathit{expr} : \tau$ and $\mathit{expr} \longrightarrow \mathit{expr}'$, then $\Gamma \vdash \mathit{expr}' : \tau$.
This is expected of any static type system.

Strong normalisation: if $\Gamma \vdash \mathit{expr} : \tau$, then there exists a normal form expr' such that $\mathit{expr} \Longrightarrow \mathit{expr}'$ (and $\Gamma \vdash \mathit{expr}' : \tau$).
This is unusual and specific to this type system.

Since the type of an expression can be easily computed by an algorithm, and the evaluation of any typable expression always terminates, it means the simply-typed λ-calculus is not Turing-complete anymore.
In particular, recursive functions are not typable in this system.

Possible extensions

Algebraic datatypes: product types (to represent tuples) and choice types (to represent options, or more generally a container whose content may have several different types).

The combination of the two allows defining lists, trees, etc.

Possible extensions

Algebraic datatypes: product types (to represent tuples) and choice types (to represent options, or more generally a container whose content may have several different types).

The combination of the two allows defining lists, trees, etc.

A specific rule for recursion: instead of using **Y** (which is not typable), we use a special syntax **letrec** $f : \tau = expr$ where f is allowed to occur inside *expr*.

Possible extensions

Algebraic datatypes: product types (to represent tuples) and choice types (to represent options, or more generally a container whose content may have several different types).

The combination of the two allows defining lists, trees, etc.

A specific rule for recursion: instead of using **Y** (which is not typable), we use a special syntax **letrec** $f : \tau = expr$ where f is allowed to occur inside *expr*.

Then the typing rule says: $\dfrac{\Gamma, f : \tau \vdash expr : \tau}{\Gamma \vdash \textbf{letrec } f : \tau = expr : \tau}$

Possible extensions

Algebraic datatypes: product types (to represent tuples) and choice types (to represent options, or more generally a container whose content may have several different types).

The combination of the two allows defining lists, trees, etc.

A specific rule for recursion: instead of using **Y** (which is not typable), we use a special syntax **letrec** $f : \tau = expr$ where f is allowed to occur inside *expr*.

Then the typing rule says: $\dfrac{\Gamma, f : \tau \vdash expr : \tau}{\Gamma \vdash \textbf{letrec}\ f : \tau = expr : \tau}$

With this rule, we lose strong normalisation.

Possible extensions

Algebraic datatypes: product types (to represent tuples) and choice types (to represent options, or more generally a container whose content may have several different types).

The combination of the two allows defining lists, trees, etc.

A specific rule for recursion: instead of using **Y** (which is not typable), we use a special syntax **letrec** $f : \tau = expr$ where f is allowed to occur inside *expr*.

Then the typing rule says: $\dfrac{\Gamma, f : \tau \vdash expr : \tau}{\Gamma \vdash \textbf{letrec } f : \tau = expr : \tau}$

With this rule, we lose strong normalisation.

Polymorphism: Some functions can work with arguments of any type: for example, the identity function. In the simply-typed λ-calculus, there is a different identity function for each type.

Possible extensions

Algebraic datatypes: product types (to represent tuples) and choice types (to represent options, or more generally a container whose content may have several different types).

The combination of the two allows defining lists, trees, etc.

A specific rule for recursion: instead of using **Y** (which is not typable), we use a special syntax **letrec** $f : \tau = expr$ where f is allowed to occur inside *expr*.

Then the typing rule says: $\dfrac{\Gamma, f : \tau \vdash expr : \tau}{\Gamma \vdash \textbf{letrec } f : \tau = expr : \tau}$

With this rule, we lose strong normalisation.

Polymorphism: Some functions can work with arguments of any type: for example, the identity function. In the simply-typed λ-calculus, there is a different identity function for each type. The type language can be extended with polymorphic types like $\forall\alpha.\alpha \to \alpha$.

Possible extensions

Algebraic datatypes: product types (to represent tuples) and choice types (to represent options, or more generally a container whose content may have several different types).

The combination of the two allows defining lists, trees, etc.

A specific rule for recursion: instead of using **Y** (which is not typable), we use a special syntax **letrec** $f : \tau = expr$ where f is allowed to occur inside *expr*.

Then the typing rule says: $\dfrac{\Gamma, f : \tau \vdash expr : \tau}{\Gamma \vdash \textbf{letrec}\ f : \tau = expr : \tau}$

With this rule, we lose strong normalisation.

Polymorphism: Some functions can work with arguments of any type: for example, the identity function. In the simply-typed λ-calculus, there is a different identity function for each type.

The type language can be extended with polymorphic types like $\forall \alpha . \alpha \to \alpha$.

For example, **comp** in such a system has type $\forall \alpha, \beta, \gamma . (\alpha \to \beta) \to (\gamma \to \beta) \to \gamma \to \alpha$.

Type inference

Type inference can be used to remove the need for type annotations in $\lambda x : \tau \cdot expr$.

Type inference

Type inference can be used to remove the need for type annotations in $\lambda x : \tau \cdot expr$.

The Hindley-Milner type system, first used in the ML language, is the most common type inference system for the λ-calculus.

Type inference

Type inference can be used to remove the need for type annotations in $\lambda x : \tau \cdot expr$.

The Hindley-Milner type system, first used in the ML language, is the most common type inference system for the λ-calculus.

Idea: To infer the type of $\lambda x \cdot expr$, a type variable is created to represent the type of x.

Type inference

Type inference can be used to remove the need for type annotations in $\lambda x : \tau \cdot expr$.

The Hindley-Milner type system, first used in the ML language, is the most common type inference system for the λ-calculus.

Idea: To infer the type of $\lambda x \cdot expr$, a type variable is created to represent the type of x. Then when analyzing $expr$, equations involving this type variable are generated.

Type inference

Type inference can be used to remove the need for type annotations in $\lambda x : \tau \cdot \mathit{expr}$.

The Hindley-Milner type system, first used in the ML language, is the most common type inference system for the λ-calculus.

Idea: To infer the type of $\lambda x \cdot \mathit{expr}$, a type variable is created to represent the type of x. Then when analyzing *expr*, equations involving this type variable are generated.

The equations are then solved using the unification algorithm. If there is a solution, it gives the type of the expression.

Type inference

Type inference can be used to remove the need for type annotations in $\lambda x : \tau \cdot expr$.

The Hindley-Milner type system, first used in the ML language, is the most common type inference system for the λ-calculus.

Idea: To infer the type of $\lambda x \cdot expr$, a type variable is created to represent the type of x. Then when analyzing $expr$, equations involving this type variable are generated.

The equations are then solved using the unification algorithm. If there is a solution, it gives the type of the expression.

Example: $\mathbf{comp} = \lambda f, g, x \cdot f\,(g\,x)$

Type inference

Type inference can be used to remove the need for type annotations in $\lambda x : \tau \cdot \mathit{expr}$.

The Hindley-Milner type system, first used in the ML language, is the most common type inference system for the λ-calculus.

Idea: To infer the type of $\lambda x \cdot \mathit{expr}$, a type variable is created to represent the type of x. Then when analyzing *expr*, equations involving this type variable are generated.

The equations are then solved using the unification algorithm. If there is a solution, it gives the type of the expression.

Example: $\mathbf{comp} = \lambda f, g, x \cdot f\,(g\,x)$

- We start with $f : \alpha, g : \beta, x : \gamma$.

Type inference

Type inference can be used to remove the need for type annotations in $\lambda x : \tau \cdot expr$.

The Hindley-Milner type system, first used in the ML language, is the most common type inference system for the λ-calculus.

Idea: To infer the type of $\lambda x \cdot expr$, a type variable is created to represent the type of x. Then when analyzing $expr$, equations involving this type variable are generated.

The equations are then solved using the unification algorithm. If there is a solution, it gives the type of the expression.

Example: $\mathbf{comp} = \lambda f, g, x \cdot f\,(g\,x)$

- We start with $f : \alpha, g : \beta, x : \gamma$.
- g is applied to x, therefore $\beta = \gamma \to \delta$.

Type inference

Type inference can be used to remove the need for type annotations in $\lambda x : \tau \cdot expr$.

The Hindley-Milner type system, first used in the ML language, is the most common type inference system for the λ-calculus.

Idea: To infer the type of $\lambda x \cdot expr$, a type variable is created to represent the type of x. Then when analyzing $expr$, equations involving this type variable are generated.

The equations are then solved using the unification algorithm. If there is a solution, it gives the type of the expression.

Example: $\mathbf{comp} = \lambda f, g, x \cdot f\,(g\,x)$

- We start with $f : \alpha, g : \beta, x : \gamma$.
- g is applied to x, therefore $\beta = \gamma \to \delta$.
- $(g\,x)$ has type δ and f is applied to it, therefore $\alpha = \delta \to \varepsilon$.

Type inference

Type inference can be used to remove the need for type annotations in $\lambda x : \tau \cdot expr$.

The Hindley-Milner type system, first used in the ML language, is the most common type inference system for the λ-calculus.

Idea: To infer the type of $\lambda x \cdot expr$, a type variable is created to represent the type of x. Then when analyzing *expr*, equations involving this type variable are generated.

The equations are then solved using the unification algorithm. If there is a solution, it gives the type of the expression.

Example: **comp** $= \lambda f, g, x \cdot f\,(g\,x)$

- We start with $f : \alpha, g : \beta, x : \gamma$.
- g is applied to x, therefore $\beta = \gamma \to \delta$.
- $(g\,x)$ has type δ and f is applied to it, therefore $\alpha = \delta \to \varepsilon$.
- Then $f\,(g\,x)$ has type ε, and **comp** $: \alpha \to \beta \to \gamma \to \varepsilon$.

Type inference

Type inference can be used to remove the need for type annotations in $\lambda x : \tau \cdot \mathit{expr}$.

The Hindley-Milner type system, first used in the ML language, is the most common type inference system for the λ-calculus.

Idea: To infer the type of $\lambda x \cdot \mathit{expr}$, a type variable is created to represent the type of x. Then when analyzing *expr*, equations involving this type variable are generated.

The equations are then solved using the unification algorithm. If there is a solution, it gives the type of the expression.

Example: **comp** $= \lambda f, g, x \cdot f\,(g\,x)$

- We start with $f : \alpha, g : \beta, x : \gamma$.
- g is applied to x, therefore $\beta = \gamma \to \delta$.
- $(g\,x)$ has type δ and f is applied to it, therefore $\alpha = \delta \to \varepsilon$.
- Then $f\,(g\,x)$ has type ε, and **comp** $: \alpha \to \beta \to \gamma \to \varepsilon$.
- Applying unification gives the final type $(\delta \to \varepsilon) \to (\gamma \to \delta) \to \gamma \to \varepsilon$.

Type inference

Type inference can be used to remove the need for type annotations in $\lambda x : \tau \cdot expr$.

The Hindley-Milner type system, first used in the ML language, is the most common type inference system for the λ-calculus.

Idea: To infer the type of $\lambda x \cdot expr$, a type variable is created to represent the type of x. Then when analyzing *expr*, equations involving this type variable are generated.

The equations are then solved using the unification algorithm. If there is a solution, it gives the type of the expression.

Example: **comp** $= \lambda f, g, x \cdot f\,(g\,x)$

- We start with $f : \alpha, g : \beta, x : \gamma$.
- g is applied to x, therefore $\beta = \gamma \to \delta$.
- $(g\,x)$ has type δ and f is applied to it, therefore $\alpha = \delta \to \varepsilon$.
- Then $f\,(g\,x)$ has type ε, and **comp** $: \alpha \to \beta \to \gamma \to \varepsilon$.
- Applying unification gives the final type $(\delta \to \varepsilon) \to (\gamma \to \delta) \to \gamma \to \varepsilon$.
- In the end, the remaining variables are generalized with $\forall$.