Introduction to the Simply Typed Lambda Calculus

Anton Pirogov

October 6, 2014

Table of contents I

Why type systems?

Many popular languages do not require tracking types of variables, which seems rather comfortable. Why should you care about type systems?

- Type checking prevents common and trivial bugs

  E.g. JavaScript or PHP: `1000 == "1e3"` is **true!**

  ⇒ weak + dynamic typing = have fun debugging!

- Working with types – "First think, then code"

  ⇒ cleaner, better organized results

- Types are never-outdated documentation!

- Type inference ⇒ not necesserily verbose

Why type systems?

Many popular languages do not require tracking types of variables, which seems rather comfortable. Why should you care about type systems?

- Type checking prevents common and trivial bugs

  E.g. JavaScript or PHP: `1000 == "1e3"` is **true!**

  $\Rightarrow$ weak + dynamic typing = have fun debugging!

- Working with types – "First think, then code"

  $\Rightarrow$ cleaner, better organized results

- Types are never-outdated documentation!

- Type inference $\Rightarrow$ not necesserily verbose

Why type systems?

Many popular languages do not require tracking types of variables, which seems rather comfortable. Why should you care about type systems?

- Type checking prevents common and trivial bugs

  E.g. JavaScript or PHP: `1000 == "1e3"` is **true!**

  ⇒ weak + dynamic typing = have fun debugging!

- Working with types – "First think, then code"

  ⇒ cleaner, better organized results

- Types are never-outdated documentation!
- Type inference ⇒ not necesserily verbose

Why type systems?

Many popular languages do not require tracking types of variables, which seems rather comfortable. Why should you care about type systems?

- Type checking prevents common and trivial bugs

  E.g. JavaScript or PHP: `1000 == "1e3"` is **true!**

  ⇒ weak + dynamic typing = have fun debugging!

- Working with types – "First think, then code"

  ⇒ cleaner, better organized results

- Types are never-outdated documentation!

- Type inference ⇒ not necesserily verbose

Why type systems?

Many popular languages do not require tracking types of variables, which seems rather comfortable. Why should you care about type systems?

- Type checking prevents common and trivial bugs
  E.g. JavaScript or PHP: `1000 == "1e3"` is **true!**
  $\Rightarrow$ weak + dynamic typing = have fun debugging!
- Working with types – "First think, then code"
  $\Rightarrow$ cleaner, better organized results
- Types are never-outdated documentation!
- Type inference $\Rightarrow$ not necesserily verbose

Untyped Lambda Calculus

Some facts:

- fundamental formal system for computation
- introduced by Alonzo Church in 1936
- shown to be equivalent to Turing Machines in 1937
- used especially in type theory + PL research
- mother of all functional programming languages
  (especially ML and LISP family)

Syntax

Definition (Syntax of λ-calculus)

$\texttt{t} ::= \texttt{x}$	*variable*
$\lambda\texttt{x}.\ \texttt{t}$	*abstraction*
$\texttt{t}\ \texttt{t}$	*application*

- Abstraction = Function
- We will sometimes use parentheses
- When not, assume $\lambda\texttt{x}.\ \ldots = (\lambda\texttt{x}.\ \ldots)$
- $\lambda\texttt{x}.\ \texttt{x}\ \texttt{y}$: x is *bound*, y is *free*

Syntax

Definition (Syntax of λ-calculus)

$\mathtt{t} ::= \mathtt{x}$		*variable*
	$\lambda\mathtt{x}.\,\mathtt{t}$	*abstraction*
	$\mathtt{t}\ \mathtt{t}$	*application*

- Abstraction = Function
- We will sometimes use parentheses
- When not, assume $\lambda\mathtt{x}.\ \ldots = (\lambda\mathtt{x}.\ \ldots)$
- $\lambda\mathtt{x}.\ \mathtt{x}\ \mathtt{y}$: x is *bound*, y is *free*

Evaluation

- one abstraction has exactly **one** parameter
- abstraction + application = *redex* (reducible expression)
- non-reducible terms = *values*
- in pure λ-calculus: abstraction is only kind of data

  $\Rightarrow$ computations always return other abstractions (only possible value!)
- *beta reduction*: one step of redex evaluation

  $(\lambda x.\, t_1)\, t_2 \to [x \mapsto t_2]t_1$

  $\Rightarrow$ function evaluation = term substitution
- We use *call-by-value* evaluation:

  evaluate terms left to right (depth-first),

  if remaining term is redex, recursively continue evaluating
- Other evaluation strategies also possible

Evaluation

- one abstraction has exactly **one** parameter
- abstraction + application = *redex* (reducible expression)
- non-reducible terms = *values*
- in pure λ-calculus: abstraction is only kind of data

  $\Rightarrow$ computations always return other abstractions (only possible value!)
- *beta reduction*: one step of redex evaluation

  $(\lambda x.\, t_1)\, t_2 \rightarrow [x \mapsto t_2]t_1$

  $\Rightarrow$ function evaluation = term substitution
- We use *call-by-value* evaluation:

  evaluate terms left to right (depth-first),

  if remaining term is redex, recursively continue evaluating
- Other evaluation strategies also possible

Evaluation

- one abstraction has exactly **one** parameter
- abstraction + application = *redex* (reducible expression)
- non-reducible terms = *values*
- in pure λ-calculus: abstraction is only kind of data

  $\Rightarrow$ computations always return other abstractions (only possible value!)
- *beta reduction*: one step of redex evaluation

  $(\lambda x.\, t_1)\, t_2 \rightarrow [x \mapsto t_2]t_1$

  $\Rightarrow$ function evaluation = term substitution
- We use *call-by-value* evaluation:

  evaluate terms left to right (depth-first),

  if remaining term is redex, recursively continue evaluating
- Other evaluation strategies also possible

Evaluation

- one abstraction has exactly **one** parameter
- abstraction + application = *redex* (reducible expression)
- non-reducible terms = *values*
- in pure λ-calculus: abstraction is only kind of data

  $\Rightarrow$ computations always return other abstractions (only possible value!)
- *beta reduction*: one step of redex evaluation

  $(\lambda x.\, t_1)\, t_2 \rightarrow [x \mapsto t_2]t_1$

  $\Rightarrow$ function evaluation = term substitution
- We use *call-by-value* evaluation:

  evaluate terms left to right (depth-first),

  if remaining term is redex, recursively continue evaluating
- Other evaluation strategies also possible

Naming of variables

What is wrong in the following evaluation?

$$[\mathrm{x} \mapsto \mathrm{z}](\lambda \mathrm{z}.\mathrm{x}) = (\lambda \mathrm{z}.[\mathrm{x} \mapsto \mathrm{z}]\mathrm{x}) = (\lambda \mathrm{z}.\mathrm{z})$$

Naming of variables

What is wrong in the following evaluation?

$$[\mathrm{x} \mapsto \mathrm{z}](\lambda \mathrm{z}.\mathrm{x}) = (\lambda \mathrm{z}.[\mathrm{x} \mapsto \mathrm{z}]\mathrm{x}) = (\lambda \mathrm{z}.\mathrm{z})$$

We changed the meaning from a constant function to an identity function!

Two possible solutions:

1. Allow substitution only if bound variable in abstraction not free in right-hand term of the substitution
2. Rename bound variable to unused name before applying such a substitution: $[\mathrm{x} \mapsto \mathrm{z}](\lambda \mathrm{z}.\mathrm{x}) = [\mathrm{x} \mapsto \mathrm{z}](\lambda \mathrm{y}.\mathrm{x}) = (\lambda \mathrm{y}.[\mathrm{x} \mapsto \mathrm{z}]\mathrm{x}) = (\lambda \mathrm{y}.\mathrm{z})$

$\Rightarrow$ called *alpha-conversion*, terms identical modulo names are α*-equivalent*

Naming of variables

What is wrong in the following evaluation?

$$[\mathrm{x} \mapsto \mathrm{z}](\lambda \mathrm{z}.\mathrm{x}) = (\lambda \mathrm{z}.[\mathrm{x} \mapsto \mathrm{z}]\mathrm{x}) = (\lambda \mathrm{z}.\mathrm{z})$$

We changed the meaning from a constant function to an identity function!

Two possible solutions:

1. Allow substitution only if bound variable in abstraction not free in right-hand term of the substitution
2. Rename bound variable to unused name before applying such a substitution: $[\mathrm{x} \mapsto \mathrm{z}](\lambda \mathrm{z}.\mathrm{x}) = [\mathrm{x} \mapsto \mathrm{z}](\lambda \mathrm{y}.\mathrm{x}) = (\lambda \mathrm{y}.[\mathrm{x} \mapsto \mathrm{z}]\mathrm{x}) = (\lambda \mathrm{y}.\mathrm{z})$

$\Rightarrow$ called *alpha-conversion*, terms identical modulo names are α*-equivalent*

Naming of variables

What is wrong in the following evaluation?

$$[\mathrm{x} \mapsto \mathrm{z}](\lambda \mathrm{z}.\mathrm{x}) = (\lambda \mathrm{z}.[\mathrm{x} \mapsto \mathrm{z}]\mathrm{x}) = (\lambda \mathrm{z}.\mathrm{z})$$

We changed the meaning from a constant function to an identity function!

Two possible solutions:

1. Allow substitution only if bound variable in abstraction not free in right-hand term of the substitution
2. Rename bound variable to unused name before applying such a substitution: $[\mathrm{x} \mapsto \mathrm{z}](\lambda \mathrm{z}.\mathrm{x}) = [\mathrm{x} \mapsto \mathrm{z}](\lambda \mathrm{y}.\mathrm{x}) = (\lambda \mathrm{y}.[\mathrm{x} \mapsto \mathrm{z}]\mathrm{x}) = (\lambda \mathrm{y}.\mathrm{z})$

$\Rightarrow$ called *alpha-conversion*, terms identical modulo names are α*-equivalent*

Currying and partial application

$$(\lambda x.\ \lambda y.\ x\ y)\ a\ b \to (\lambda y.\ a\ y)\ b \to a\ b$$

- Nested abstractions ‘simulate’ functions with multiple arguments
- Technique called *currying*, named after Haskell Curry
  (but thought to go back to Moses Schönfinkel)
- Inverse action – applying only some arguments to a curried function before e.g. passing it somewhere else is called *partial application*
- Here: Successive substitutions $[x \mapsto a]$ and $[y \mapsto b]$
  = passing first and second argument one after the other

Currying and partial application

$$(\lambda x.\ \lambda y.\ x\ y)\ a\ b \rightarrow (\lambda y.\ a\ y)\ b \rightarrow a\ b$$

- Nested abstractions 'simulate' functions with multiple arguments
- Technique called *currying*, named after Haskell Curry (but thought to go back to Moses Schönfinkel)
- Inverse action – applying only some arguments to a curried function before e.g. passing it somewhere else is called *partial application*
- Here: Successive substitutions $[x \mapsto a]$ and $[y \mapsto b]$ = passing first and second argument one after the other

Currying and partial application

$$(\lambda x.\ \lambda y.\ x\ y)\ a\ b \rightarrow (\lambda y.\ a\ y)\ b \rightarrow a\ b$$

- Nested abstractions ‘simulate’ functions with multiple arguments
- Technique called *currying*, named after Haskell Curry
  (but thought to go back to Moses Schönfinkel)
- Inverse action – applying only some arguments to a curried function
  before e.g. passing it somewhere else is called *partial application*
- Here: Successive substitutions $[x \mapsto a]$ and $[y \mapsto b]$
  = passing first and second argument one after the other

Currying and partial application

$$(\lambda \mathtt{x}.\ \lambda \mathtt{y}.\ \mathtt{x}\ \mathtt{y})\ \mathtt{a}\ \mathtt{b} \to (\lambda \mathtt{y}.\ \mathtt{a}\ \mathtt{y})\ \mathtt{b} \to \mathtt{a}\ \mathtt{b}$$

- Nested abstractions ‘simulate’ functions with multiple arguments
- Technique called *currying*, named after Haskell Curry
  (but thought to go back to Moses Schönfinkel)
- Inverse action – applying only some arguments to a curried function
  before e.g. passing it somewhere else is called *partial application*
- Here: Successive substitutions $[\mathtt{x} \mapsto \mathtt{a}]$ and $[\mathtt{y} \mapsto \mathtt{b}]$
  = passing first and second argument one after the other

Currying and partial application

$$(\lambda \mathtt{x}.\ \lambda \mathtt{y}.\ \mathtt{x}\ \mathtt{y})\ \mathtt{a}\ \mathtt{b} \to (\lambda \mathtt{y}.\ \mathtt{a}\ \mathtt{y})\ \mathtt{b} \to \mathtt{a}\ \mathtt{b}$$

- Nested abstractions ‘simulate’ functions with multiple arguments
- Technique called *currying*, named after Haskell Curry (but thought to go back to Moses Schönfinkel)
- Inverse action – applying only some arguments to a curried function before e.g. passing it somewhere else is called *partial application*
- Here: Successive substitutions $[\mathtt{x} \mapsto \mathtt{a}]$ and $[\mathtt{y} \mapsto \mathtt{b}]$ = passing first and second argument one after the other

Church booleans

Q: How can we calculate something meaningful, having only abstractions?

A: Find special abstractions we will treat as booleans ⇒ *Church booleans*

$$\mathtt{tru} = \lambda\mathtt{t}.\,\lambda\mathtt{f}.\,\mathtt{t}$$

$$\mathtt{fls} = \lambda\mathtt{t}.\,\lambda\mathtt{f}.\,\mathtt{f}$$

$$\mathtt{not} = \lambda\mathtt{b}.\,\mathtt{b}\ \mathtt{fls}\ \mathtt{tru}$$

$$\mathtt{and} = \lambda\mathtt{b}.\,\lambda\mathtt{c}.\,\mathtt{b}\ \mathtt{c}\ \mathtt{fls}$$

$$\mathtt{or} = \lambda\mathtt{b}.\,\lambda\mathtt{c}.\,\mathtt{b}\ \mathtt{tru}\ \mathtt{c}$$

$$\mathtt{test} = \lambda\mathtt{l}.\,\lambda\mathtt{m}.\,\lambda\mathtt{n}.\,\mathtt{l}\ \mathtt{m}\ \mathtt{n}$$

Problem: `test` always evaluates both arguments (= if-branches)

Solution: Wrap the arguments in dummy abstractions, unpack afterwards

Church booleans

Q: How can we calculate something meaningful, having only abstractions?

A: Find special abstractions we will treat as booleans ⇒ *Church booleans*

$\texttt{tru} = \lambda\texttt{t}.\,\lambda\texttt{f}.\,\texttt{t}$

$\texttt{fls} = \lambda\texttt{t}.\,\lambda\texttt{f}.\,\texttt{f}$

$\texttt{not} = \lambda\texttt{b}.\,\texttt{b fls tru}$

$\texttt{and} = \lambda\texttt{b}.\,\lambda\texttt{c}.\,\texttt{b c fls}$

$\texttt{or} = \lambda\texttt{b}.\,\lambda\texttt{c}.\,\texttt{b tru c}$

$\texttt{test} = \lambda\texttt{l}.\,\lambda\texttt{m}.\,\lambda\texttt{n}.\,\texttt{l m n}$

Problem: `test` always evaluates both arguments (= if-branches)

Solution: Wrap the arguments in dummy abstractions, unpack afterwards

Church booleans

Q: How can we calculate something meaningful, having only abstractions?

A: Find special abstractions we will treat as booleans ⇒ *Church booleans*

$$\texttt{tru} = \lambda \texttt{t}.\ \lambda \texttt{f}.\ \texttt{t}$$

$$\texttt{fls} = \lambda \texttt{t}.\ \lambda \texttt{f}.\ \texttt{f}$$

$$\texttt{not} = \lambda \texttt{b}.\ \texttt{b}\ \texttt{fls}\ \texttt{tru}$$

$$\texttt{and} = \lambda \texttt{b}.\ \lambda \texttt{c}.\ \texttt{b}\ \texttt{c}\ \texttt{fls}$$

$$\texttt{or} = \lambda \texttt{b}.\ \lambda \texttt{c}.\ \texttt{b}\ \texttt{tru}\ \texttt{c}$$

$$\texttt{test} = \lambda \texttt{l}.\ \lambda \texttt{m}.\ \lambda \texttt{n}.\ \texttt{l}\ \texttt{m}\ \texttt{n}$$

Problem: `test` always evaluates both arguments (= if-branches)

Solution: Wrap the arguments in dummy abstractions, unpack afterwards

Church booleans

Q: How can we calculate something meaningful, having only abstractions?

A: Find special abstractions we will treat as booleans ⇒ *Church booleans*

$$\texttt{tru} = \lambda \texttt{t}.\ \lambda \texttt{f}.\ \texttt{t}$$

$$\texttt{fls} = \lambda \texttt{t}.\ \lambda \texttt{f}.\ \texttt{f}$$

$$\texttt{not} = \lambda \texttt{b}.\ \texttt{b fls tru}$$

$$\texttt{and} = \lambda \texttt{b}.\ \lambda \texttt{c}.\ \texttt{b c fls}$$

$$\texttt{or} = \lambda \texttt{b}.\ \lambda \texttt{c}.\ \texttt{b tru c}$$

$$\texttt{test} = \lambda \texttt{l}.\ \lambda \texttt{m}.\ \lambda \texttt{n}.\ \texttt{l m n}$$

Problem: `test` always evaluates both arguments (= if-branches)

Solution: Wrap the arguments in dummy abstractions, unpack afterwards

Church booleans

Q: How can we calculate something meaningful, having only abstractions?

A: Find special abstractions we will treat as booleans ⇒ *Church booleans*

$$\texttt{not} = \lambda \texttt{b}.\ \texttt{b}\ \texttt{fls}\ \texttt{tru}$$

$$\texttt{tru} = \lambda \texttt{t}.\ \lambda \texttt{f}.\ \texttt{t} \qquad \texttt{and} = \lambda \texttt{b}.\ \lambda \texttt{c}.\ \texttt{b}\ \texttt{c}\ \texttt{fls}$$

$$\texttt{fls} = \lambda \texttt{t}.\ \lambda \texttt{f}.\ \texttt{f} \qquad \texttt{or} = \lambda \texttt{b}.\ \lambda \texttt{c}.\ \texttt{b}\ \texttt{tru}\ \texttt{c}$$

$$\texttt{test} = \lambda \texttt{l}.\ \lambda \texttt{m}.\ \lambda \texttt{n}.\ \texttt{l}\ \texttt{m}\ \texttt{n}$$

Problem: `test` always evaluates both arguments (= if-branches)

Solution: Wrap the arguments in dummy abstractions, unpack afterwards

Church booleans

Q: How can we calculate something meaningful, having only abstractions?

A: Find special abstractions we will treat as booleans ⇒ *Church booleans*

$$\mathtt{tru} = \lambda \mathtt{t}.\, \lambda \mathtt{f}.\, \mathtt{t}$$

$$\mathtt{fls} = \lambda \mathtt{t}.\, \lambda \mathtt{f}.\, \mathtt{f}$$

$$\mathtt{not} = \lambda \mathtt{b}.\, \mathtt{b\ fls\ tru}$$

$$\mathtt{and} = \lambda \mathtt{b}.\, \lambda \mathtt{c}.\, \mathtt{b\ c\ fls}$$

$$\mathtt{or} = \lambda \mathtt{b}.\, \lambda \mathtt{c}.\, \mathtt{b\ tru\ c}$$

$$\mathtt{test} = \lambda \mathtt{l}.\, \lambda \mathtt{m}.\, \lambda \mathtt{n}.\, \mathtt{l\ m\ n}$$

Problem: `test` always evaluates both arguments (= if-branches)

Solution: Wrap the arguments in dummy abstractions, unpack afterwards

Church booleans

Q: How can we calculate something meaningful, having only abstractions?

A: Find special abstractions we will treat as booleans ⇒ *Church booleans*

$$\mathtt{tru} = \lambda \mathtt{t}.\, \lambda \mathtt{f}.\, \mathtt{t}$$

$$\mathtt{fls} = \lambda \mathtt{t}.\, \lambda \mathtt{f}.\, \mathtt{f}$$

$$\mathtt{not} = \lambda \mathtt{b}.\, \mathtt{b}\ \mathtt{fls}\ \mathtt{tru}$$

$$\mathtt{and} = \lambda \mathtt{b}.\, \lambda \mathtt{c}.\, \mathtt{b}\ \mathtt{c}\ \mathtt{fls}$$

$$\mathtt{or} = \lambda \mathtt{b}.\, \lambda \mathtt{c}.\, \mathtt{b}\ \mathtt{tru}\ \mathtt{c}$$

$$\mathtt{test} = \lambda \mathtt{l}.\, \lambda \mathtt{m}.\, \lambda \mathtt{n}.\, \mathtt{l}\ \mathtt{m}\ \mathtt{n}$$

Problem: `test` always evaluates both arguments (= if-branches)

Solution: Wrap the arguments in dummy abstractions, unpack afterwards

Church numerals

There is also an encoding for natural numbers, called *Church numerals*:

$$\mathtt{c}_0 = \lambda \mathtt{s}.\, \lambda \mathtt{z}.\, \mathtt{z} \qquad \mathtt{scc} = \lambda \mathtt{n}.\, \lambda \mathtt{s}.\, \lambda \mathtt{z}.\, \mathtt{s}\ (\mathtt{n}\ \mathtt{s}\ \mathtt{z})$$

$$\mathtt{c}_1 = \lambda \mathtt{s}.\, \lambda \mathtt{z}.\, \mathtt{s}\ \mathtt{z} \qquad \mathtt{plus} = \lambda \mathtt{m}.\, \lambda \mathtt{n}.\, \lambda \mathtt{s}.\, \lambda \mathtt{z}.\, \mathtt{m}\ \mathtt{s}\ (\mathtt{n}\ \mathtt{s}\ \mathtt{z})$$

$$\mathtt{c}_2 = \lambda \mathtt{s}.\, \lambda \mathtt{z}.\, \mathtt{s}\ (\mathtt{s}\ \mathtt{z}) \qquad \mathtt{times} = \lambda \mathtt{m}.\, \lambda \mathtt{n}.\, \mathtt{m}\ (\mathtt{plus}\ \mathtt{n})\ \mathtt{c}_0$$

$$\ldots \qquad \mathtt{iszro} = \lambda \mathtt{m}.\, \mathtt{m}\ (\lambda \mathtt{x}.\, \mathtt{fls})\ \mathtt{tru}$$

- Subtraction also possible, but more tricky
- With subtraction we also get equality:

$$\mathtt{eq} = \lambda \mathtt{n}.\, \lambda \mathtt{m}.\, \mathtt{and}\ (\mathtt{iszro}\ (\mathtt{minus}\ \mathtt{n}\ \mathtt{m}))\ (\mathtt{iszro}\ (\mathtt{minus}\ \mathtt{m}\ \mathtt{n}))$$

Church numerals

There is also an encoding for natural numbers, called *Church numerals*:

$$c_0 = \lambda s.\ \lambda z.\ z \qquad scc = \lambda n.\ \lambda s.\ \lambda z.\ s\ (n\ s\ z)$$

$$c_1 = \lambda s.\ \lambda z.\ s\ z \qquad plus = \lambda m.\ \lambda n.\ \lambda s.\ \lambda z.\ m\ s\ (n\ s\ z)$$

$$c_2 = \lambda s.\ \lambda z.\ s\ (s\ z) \qquad times = \lambda m.\ \lambda n.\ m\ (plus\ n)\ c_0$$

$$\ldots \qquad iszro = \lambda m.\ m\ (\lambda x.\ fls)\ tru$$

- Subtraction also possible, but more tricky
- With subtraction we also get equality:

$$eq = \lambda n.\ \lambda m.\ and\ (iszro\ (minus\ n\ m))$$
$$(iszro\ (minus\ m\ n))$$

Church numerals

There is also an encoding for natural numbers, called *Church numerals*:

$$\mathtt{c}_0 = \lambda \mathtt{s}.\, \lambda \mathtt{z}.\, \mathtt{z}$$

$$\mathtt{c}_1 = \lambda \mathtt{s}.\, \lambda \mathtt{z}.\, \mathtt{s}\ \mathtt{z}$$

$$\mathtt{c}_2 = \lambda \mathtt{s}.\, \lambda \mathtt{z}.\, \mathtt{s}\ (\mathtt{s}\ \mathtt{z})$$

$\dots$

$$\mathtt{scc} = \lambda \mathtt{n}.\, \lambda \mathtt{s}.\, \lambda \mathtt{z}.\, \mathtt{s}\ (\mathtt{n}\ \mathtt{s}\ \mathtt{z})$$

$$\mathtt{plus} = \lambda \mathtt{m}.\, \lambda \mathtt{n}.\, \lambda \mathtt{s}.\, \lambda \mathtt{z}.\, \mathtt{m}\ \mathtt{s}\ (\mathtt{n}\ \mathtt{s}\ \mathtt{z})$$

$$\mathtt{times} = \lambda \mathtt{m}.\, \lambda \mathtt{n}.\, \mathtt{m}\ (\mathtt{plus}\ \mathtt{n})\ \mathtt{c}_0$$

$$\mathtt{iszro} = \lambda \mathtt{m}.\, \mathtt{m}\ (\lambda \mathtt{x}.\, \mathtt{fls})\ \mathtt{tru}$$

- Subtraction also possible, but more tricky
- With subtraction we also get equality:

$$\mathtt{eq} = \lambda \mathtt{n}.\, \lambda \mathtt{m}.\, \mathtt{and}\ (\mathtt{iszro}\ (\mathtt{minus}\ \mathtt{n}\ \mathtt{m}))\ (\mathtt{iszro}\ (\mathtt{minus}\ \mathtt{m}\ \mathtt{n}))$$

Church numerals

There is also an encoding for natural numbers, called *Church numerals*:

$\mathtt{c}_0 = \lambda \mathtt{s}.\ \lambda \mathtt{z}.\ \mathtt{z}$

$\mathtt{c}_1 = \lambda \mathtt{s}.\ \lambda \mathtt{z}.\ \mathtt{s}\ \mathtt{z}$

$\mathtt{c}_2 = \lambda \mathtt{s}.\ \lambda \mathtt{z}.\ \mathtt{s}\ (\mathtt{s}\ \mathtt{z})$

$\dots$

$\mathtt{scc} = \lambda \mathtt{n}.\ \lambda \mathtt{s}.\ \lambda \mathtt{z}.\ \mathtt{s}\ (\mathtt{n}\ \mathtt{s}\ \mathtt{z})$

$\mathtt{plus} = \lambda \mathtt{m}.\ \lambda \mathtt{n}.\ \lambda \mathtt{s}.\ \lambda \mathtt{z}.\ \mathtt{m}\ \mathtt{s}\ (\mathtt{n}\ \mathtt{s}\ \mathtt{z})$

$\mathtt{times} = \lambda \mathtt{m}.\ \lambda \mathtt{n}.\ \mathtt{m}\ (\mathtt{plus}\ \mathtt{n})\ \mathtt{c}_0$

$\mathtt{iszro} = \lambda \mathtt{m}.\ \mathtt{m}\ (\lambda \mathtt{x}.\ \mathtt{fls})\ \mathtt{tru}$

- Subtraction also possible, but more tricky
- With subtraction we also get equality:

$$\mathtt{eq} = \lambda \mathtt{n}.\ \lambda \mathtt{m}.\ \mathtt{and}\ (\mathtt{iszro}\ (\mathtt{minus}\ \mathtt{n}\ \mathtt{m}))\ (\mathtt{iszro}\ (\mathtt{minus}\ \mathtt{m}\ \mathtt{n}))$$

Church numerals

There is also an encoding for natural numbers, called *Church numerals*:

$$\mathtt{c}_0 = \lambda \mathtt{s}.\ \lambda \mathtt{z}.\ \mathtt{z}$$

$$\mathtt{c}_1 = \lambda \mathtt{s}.\ \lambda \mathtt{z}.\ \mathtt{s}\ \mathtt{z}$$

$$\mathtt{c}_2 = \lambda \mathtt{s}.\ \lambda \mathtt{z}.\ \mathtt{s}\ (\mathtt{s}\ \mathtt{z})$$

...

$$\mathtt{scc} = \lambda \mathtt{n}.\ \lambda \mathtt{s}.\ \lambda \mathtt{z}.\ \mathtt{s}\ (\mathtt{n}\ \mathtt{s}\ \mathtt{z})$$

$$\mathtt{plus} = \lambda \mathtt{m}.\ \lambda \mathtt{n}.\ \lambda \mathtt{s}.\ \lambda \mathtt{z}.\ \mathtt{m}\ \mathtt{s}\ (\mathtt{n}\ \mathtt{s}\ \mathtt{z})$$

$$\mathtt{times} = \lambda \mathtt{m}.\ \lambda \mathtt{n}.\ \mathtt{m}\ (\mathtt{plus}\ \mathtt{n})\ \mathtt{c}_0$$

$$\mathtt{iszro} = \lambda \mathtt{m}.\ \mathtt{m}\ (\lambda \mathtt{x}.\ \mathtt{fls})\ \mathtt{tru}$$

- Subtraction also possible, but more tricky
- With subtraction we also get equality:

$$\mathtt{eq} = \lambda \mathtt{n}.\ \lambda \mathtt{m}.\ \mathtt{and}\ (\mathtt{iszro}\ (\mathtt{minus}\ \mathtt{n}\ \mathtt{m}))\ (\mathtt{iszro}\ (\mathtt{minus}\ \mathtt{m}\ \mathtt{n}))$$

Church numerals

There is also an encoding for natural numbers, called *Church numerals*:

$$\mathtt{c}_0 = \lambda \mathtt{s}.\ \lambda \mathtt{z}.\ \mathtt{z}$$

$$\mathtt{c}_1 = \lambda \mathtt{s}.\ \lambda \mathtt{z}.\ \mathtt{s}\ \mathtt{z}$$

$$\mathtt{c}_2 = \lambda \mathtt{s}.\ \lambda \mathtt{z}.\ \mathtt{s}\ (\mathtt{s}\ \mathtt{z})$$

$\ldots$

$$\mathtt{scc} = \lambda \mathtt{n}.\ \lambda \mathtt{s}.\ \lambda \mathtt{z}.\ \mathtt{s}\ (\mathtt{n}\ \mathtt{s}\ \mathtt{z})$$

$$\mathtt{plus} = \lambda \mathtt{m}.\ \lambda \mathtt{n}.\ \lambda \mathtt{s}.\ \lambda \mathtt{z}.\ \mathtt{m}\ \mathtt{s}\ (\mathtt{n}\ \mathtt{s}\ \mathtt{z})$$

$$\mathtt{times} = \lambda \mathtt{m}.\ \lambda \mathtt{n}.\ \mathtt{m}\ (\mathtt{plus}\ \mathtt{n})\ \mathtt{c}_0$$

$$\mathtt{iszro} = \lambda \mathtt{m}.\ \mathtt{m}\ (\lambda \mathtt{x}.\ \mathtt{fls})\ \mathtt{tru}$$

- Subtraction also possible, but more tricky
- With subtraction we also get equality:

$$\mathtt{eq} = \lambda \mathtt{n}.\ \lambda \mathtt{m}.\ \mathtt{and}\ (\mathtt{iszro}\ (\mathtt{minus}\ \mathtt{n}\ \mathtt{m}))\ (\mathtt{iszro}\ (\mathtt{minus}\ \mathtt{m}\ \mathtt{n}))$$

Church numerals

There is also an encoding for natural numbers, called *Church numerals*:

$$\mathtt{c}_0 = \lambda \mathtt{s}.\, \lambda \mathtt{z}.\, \mathtt{z}$$

$$\mathtt{c}_1 = \lambda \mathtt{s}.\, \lambda \mathtt{z}.\, \mathtt{s}\ \mathtt{z}$$

$$\mathtt{c}_2 = \lambda \mathtt{s}.\, \lambda \mathtt{z}.\, \mathtt{s}\ (\mathtt{s}\ \mathtt{z})$$

...

$$\mathtt{scc} = \lambda \mathtt{n}.\, \lambda \mathtt{s}.\, \lambda \mathtt{z}.\, \mathtt{s}\ (\mathtt{n}\ \mathtt{s}\ \mathtt{z})$$

$$\mathtt{plus} = \lambda \mathtt{m}.\, \lambda \mathtt{n}.\, \lambda \mathtt{s}.\, \lambda \mathtt{z}.\, \mathtt{m}\ \mathtt{s}\ (\mathtt{n}\ \mathtt{s}\ \mathtt{z})$$

$$\mathtt{times} = \lambda \mathtt{m}.\, \lambda \mathtt{n}.\, \mathtt{m}\ (\mathtt{plus}\ \mathtt{n})\ \mathtt{c}_0$$

$$\mathtt{iszro} = \lambda \mathtt{m}.\, \mathtt{m}\ (\lambda \mathtt{x}.\, \mathtt{fls})\ \mathtt{tru}$$

- Subtraction also possible, but more tricky
- With subtraction we also get equality:

$$\mathtt{eq} = \lambda \mathtt{n}.\, \lambda \mathtt{m}.\, \mathtt{and}\ (\mathtt{iszro}\ (\mathtt{minus}\ \mathtt{n}\ \mathtt{m}))\ (\mathtt{iszro}\ (\mathtt{minus}\ \mathtt{m}\ \mathtt{n}))$$

Adding real booleans and numbers

No real programming language uses Church encoded data ⇒ inefficient!

Easy to extend syntax to support primitive data types as atomic values:

- Booleans: add `true`, `false`, `if t then t else t`
- Numbers: add `0`, `succ`, `pred`, `iszero`

Evaluation:

- `if`-condition evaluated ⇒ replace `if`-expression by correct branch
- `succ` + `pred` form redex ⇒ when they meet, remove
- `iszero 0` evaluates to `true`, otherwise `false`

Easy to convert between Church-encoded and primitive values, e.g.:

$$\texttt{realbool} = \lambda b.\ b\ \texttt{true}\ \texttt{false}$$

$$\texttt{churchbool} = \lambda b.\ \texttt{if}\ b\ \texttt{then}\ \texttt{tru}\ \texttt{else}\ \texttt{fls}$$

Adding real booleans and numbers

No real programming language uses Church encoded data ⇒ inefficient!

Easy to extend syntax to support primitive data types as atomic values:

- Booleans: add `true`, `false`, `if t then t else t`
- Numbers: add `0`, `succ`, `pred`, `iszero`

Evaluation:

- `if`-condition evaluated ⇒ replace `if`-expression by correct branch
- `succ` + `pred` form redex ⇒ when they meet, remove
- `iszero 0` evaluates to `true`, otherwise `false`

Easy to convert between Church-encoded and primitive values, e.g.:

$$\mathtt{realbool} = \lambda b.\, b\ \mathtt{true}\ \mathtt{false}$$

$$\mathtt{churchbool} = \lambda b.\, \mathtt{if}\ b\ \mathtt{then}\ \mathtt{tru}\ \mathtt{else}\ \mathtt{fls}$$

Adding real booleans and numbers

No real programming language uses Church encoded data ⇒ inefficient!

Easy to extend syntax to support primitive data types as atomic values:

- Booleans: add `true`, `false`, `if t then t else t`
- Numbers: add `0`, `succ`, `pred`, `iszero`

Evaluation:

- `if`-condition evaluated ⇒ replace `if`-expression by correct branch
- `succ` + `pred` form redex ⇒ when they meet, remove
- `iszero 0` evaluates to `true`, otherwise `false`

Easy to convert between Church-encoded and primitive values, e.g.:

$$\mathtt{realbool} = \lambda b.\, b\ \mathtt{true}\ \mathtt{false}$$

$$\mathtt{churchbool} = \lambda b.\, \mathtt{if}\ b\ \mathtt{then}\ \mathtt{tru}\ \mathtt{else}\ \mathtt{fls}$$

Adding real booleans and numbers

No real programming language uses Church encoded data ⇒ inefficient!

Easy to extend syntax to support primitive data types as atomic values:

- Booleans: add `true`, `false`, `if t then t else t`
- Numbers: add `0`, `succ`, `pred`, `iszero`

Evaluation:

- `if`-condition evaluated ⇒ replace `if`-expression by correct branch
- `succ` + `pred` form redex ⇒ when they meet, remove
- `iszero 0` evaluates to `true`, otherwise `false`

Easy to convert between Church-encoded and primitive values, e.g.:

$$\texttt{realbool} = \lambda b.\, b\ \texttt{true}\ \texttt{false}$$

$$\texttt{churchbool} = \lambda b.\, \texttt{if}\ b\ \texttt{then}\ \texttt{tru}\ \texttt{else}\ \texttt{fls}$$

- there are many possible extensions to the pure calculus:
- more primitive types, lists, tuples, recursion ($\rightarrow$ looping), ...
- either as part of formal definition or as *syntactic sugar*

  $\Rightarrow$ convenient notation for constructions that are possible,

  but are verbose/ugly/hard to use with base definition
- sugar helps keeping the core language clean and simple
- we do not add more stuff, finally move on to types ...

Motivation

Q: What about input like `if 0 then true else 0` or `succ false`?

A: Depending on the concrete expression and defined semantics:

- evaluation gets stuck at undefined state ($\rightarrow$ runtime-error)
- **worse**: evaluation continues, producing garbage, possibly undetected!

We need a way to easily and automatically check input **before** actual evaluation and only accept *well-typed* input that is playing by the rules!

Solution:

- Assign each function a type of the form $T_1 \rightarrow T_2$
- read: function taking value of type T_1, returning value of type T_2
- $\rightarrow$ = *type constructor*, T_n = *type variable*
- $\rightarrow$ is right-associative: $A \rightarrow B \rightarrow C = A \rightarrow (B \rightarrow C)$

Motivation

Q: What about input like `if 0 then true else 0` or `succ false`?

A: Depending on the concrete expression and defined semantics:

- evaluation gets stuck at undefined state ($\rightarrow$ runtime-error)
- **worse**: evaluation continues, producing garbage, possibly undetected!

We need a way to easily and automatically check input **before** actual evaluation and only accept *well-typed* input that is playing by the rules!

Solution:

- Assign each function a type of the form $T_1 \rightarrow T_2$
- read: function taking value of type T_1, returning value of type T_2
- $\rightarrow$ = *type constructor*, T_n = *type variable*
- $\rightarrow$ is right-associative: $A \rightarrow B \rightarrow C = A \rightarrow (B \rightarrow C)$

Motivation

Q: What about input like `if 0 then true else 0` or `succ false`?

A: Depending on the concrete expression and defined semantics:

- evaluation gets stuck at undefined state ($\rightarrow$ runtime-error)
- **worse**: evaluation continues, producing garbage, possibly undetected!

We need a way to easily and automatically check input **before** actual evaluation and only accept *well-typed* input that is playing by the rules!

Solution:

- Assign each function a type of the form $T_1 \rightarrow T_2$
- read: function taking value of type T_1, returning value of type T_2
- $\rightarrow$ = *type constructor*, T_n = *type variable*
- $\rightarrow$ is right-associative: $A \rightarrow B \rightarrow C = A \rightarrow (B \rightarrow C)$

Motivation

Q: What about input like `if 0 then true else 0` or `succ false`?

A: Depending on the concrete expression and defined semantics:

- evaluation gets stuck at undefined state ($\rightarrow$ runtime-error)
- **worse**: evaluation continues, producing garbage, possibly undetected!

We need a way to easily and automatically check input **before** actual evaluation and only accept *well-typed* input that is playing by the rules!

Solution:

- Assign each function a type of the form $T_1 \rightarrow T_2$
- read: function taking value of type T_1, returning value of type T_2
- $\rightarrow$ = *type constructor*, T_n = *type variable*
- $\rightarrow$ is right-associative: $A \rightarrow B \rightarrow C = A \rightarrow (B \rightarrow C)$

Motivation

Q: What about input like `if 0 then true else 0` or `succ false`?

A: Depending on the concrete expression and defined semantics:

- evaluation gets stuck at undefined state ($\rightarrow$ runtime-error)
- **worse**: evaluation continues, producing garbage, possibly undetected!

We need a way to easily and automatically check input **before** actual evaluation and only accept *well-typed* input that is playing by the rules!

Solution:

- Assign each function a type of the form $T_1 \rightarrow T_2$
- read: function taking value of type T_1, returning value of type T_2
- $\rightarrow$ = *type constructor*, T_n = *type variable*
- $\rightarrow$ is right-associative: $A \rightarrow B \rightarrow C = A \rightarrow (B \rightarrow C)$

Motivation

Q: What about input like `if 0 then true else 0` or `succ false`?

A: Depending on the concrete expression and defined semantics:

- evaluation gets stuck at undefined state ($\rightarrow$ runtime-error)
- **worse**: evaluation continues, producing garbage, possibly undetected!

We need a way to easily and automatically check input **before** actual evaluation and only accept *well-typed* input that is playing by the rules!

Solution:

- Assign each function a type of the form $T_1 \rightarrow T_2$
- read: function taking value of type T_1, returning value of type T_2
- $\rightarrow$ = *type constructor*, T_n = *type variable*
- $\rightarrow$ is right-associative: $A \rightarrow B \rightarrow C = A \rightarrow (B \rightarrow C)$

Motivation

Q: What about input like `if 0 then true else 0` or `succ false`?

A: Depending on the concrete expression and defined semantics:

- evaluation gets stuck at undefined state ($\rightarrow$ runtime-error)
- **worse**: evaluation continues, producing garbage, possibly undetected!

We need a way to easily and automatically check input **before** actual evaluation and only accept *well-typed* input that is playing by the rules!

Solution:

- Assign each function a type of the form $T_1 \rightarrow T_2$
- read: function taking value of type T_1, returning value of type T_2
- $\rightarrow$ = *type constructor*, T_n = *type variable*
- $\rightarrow$ is right-associative: $A \rightarrow B \rightarrow C = A \rightarrow (B \rightarrow C)$

Syntax

Definition (Syntax of simply typed λ-calculus ($\lambda^{\rightarrow}$))

$\mathtt{t} ::=$	$\mathtt{x}$	*variable*
	$\lambda\mathtt{x}\colon \mathbf{T}.\ \mathtt{t}$	*abstraction*
	$\mathtt{t}$	*application*

- Invented by Church in 1940
- Only superficial difference: every abstraction gets type annotation
- *simply typed* = only way to construct types is $\rightarrow$

We need some rules for correct type annotation. Some new notation first:

- Let Γ be a set of assumptions about types of terms, e.g. free variables, called *typing context*
- $\Gamma \vdash \mathtt{t} : \mathtt{T}$ means 'under given assumptions the term t has the type T'
- Γ can be $\emptyset \Rightarrow$ can be omitted in that case: $\vdash \mathtt{t} : \mathtt{T}$ or $\mathtt{t} : \mathtt{T}$
- $\frac{A}{B}$ is a *deduction rule*, means implication like $A \Rightarrow B$

We need some rules for correct type annotation. Some new notation first:

- Let Γ be a set of assumptions about types of terms,
  e.g. free variables, called *typing context*
- $\Gamma \vdash \texttt{t} : \texttt{T}$ means 'under given assumptions the term t has the type T'
- Γ can be $\emptyset \Rightarrow$ can be omitted in that case: $\vdash \texttt{t} : \texttt{T}$ or $\texttt{t} : \texttt{T}$
- $\dfrac{A}{B}$ is a *deduction rule*, means implication like $A \Rightarrow B$

We need some rules for correct type annotation. Some new notation first:

- Let Γ be a set of assumptions about types of terms,

  e.g. free variables, called *typing context*
- $\Gamma \vdash \texttt{t} : \texttt{T}$ means 'under given assumptions the term t has the type T'
- Γ can be $\emptyset \Rightarrow$ can be omitted in that case: $\vdash \texttt{t} : \texttt{T}$ or $\texttt{t} : \texttt{T}$
- $\frac{A}{B}$ is a *deduction rule*, means implication like $A \Rightarrow B$

We need some rules for correct type annotation. Some new notation first:

- Let Γ be a set of assumptions about types of terms,

  e.g. free variables, called *typing context*
- $\Gamma \vdash \mathtt{t} : \mathtt{T}$ means 'under given assumptions the term t has the type T'
- Γ can be $\emptyset \Rightarrow$ can be omitted in that case: $\vdash \mathtt{t} : \mathtt{T}$ or $\mathtt{t} : \mathtt{T}$
- $\dfrac{A}{B}$ is a *deduction rule*, means implication like $A \Rightarrow B$

Typing rules

Definition (Typing of variables (T-Var))

$$\frac{x : T \in \Gamma}{\Gamma \vdash x : T}$$

Definition (Typing of abstractions (T-Abs))

$$\frac{\Gamma, x : T_1 \vdash t_2 : T_2}{\Gamma \vdash \lambda x : T_1 .\, t_2 : T_1 \to T_2}$$

Definition (Typing of applications (T-App))

$$\frac{\Gamma \vdash t_1 : T_{11} \to T_{12} \quad \Gamma \vdash t_2 : T_{11}}{\Gamma \vdash t_1\ t_2 : T_{12}}$$

Typing rules

Definition (Typing of variables (T-Var))

$$\frac{x : T \in \Gamma}{\Gamma \vdash x : T}$$

Definition (Typing of abstractions (T-Abs))

$$\frac{\Gamma, x : T_1 \vdash t_2 : T_2}{\Gamma \vdash \lambda x : T_1 .\, t_2 : T_1 \to T_2}$$

Definition (Typing of applications (T-App))

$$\frac{\Gamma \vdash t_1 : T_{11} \to T_{12} \quad \Gamma \vdash t_2 : T_{11}}{\Gamma \vdash t_1\ t_2 : T_{12}}$$

Typing rules

Definition (Typing of variables (T-Var))

$$\frac{\mathtt{x} : \mathtt{T} \in \Gamma}{\Gamma \vdash \mathtt{x} : \mathtt{T}}$$

Definition (Typing of abstractions (T-Abs))

$$\frac{\Gamma, \mathtt{x} : \mathtt{T}_1 \vdash \mathtt{t}_2 : \mathtt{T}_2}{\Gamma \vdash \lambda \mathtt{x} : \mathtt{T}_1 .\, \mathtt{t}_2 : \mathtt{T}_1 \to \mathtt{T}_2}$$

Definition (Typing of applications (T-App))

$$\frac{\Gamma \vdash \mathtt{t}_1 : \mathtt{T}_{11} \to \mathtt{T}_{12} \qquad \Gamma \vdash \mathtt{t}_2 : \mathtt{T}_{11}}{\Gamma \vdash \mathtt{t}_1\ \mathtt{t}_2 : \mathtt{T}_{12}}$$

Typing rules for booleans and numbers

Definition (T-True, T-False, T-If)

$$\texttt{true} : \texttt{Bool} \qquad \texttt{false} : \texttt{Bool} \qquad \frac{\Gamma \vdash \texttt{t}_1 : \texttt{Bool} \quad \Gamma \vdash \texttt{t}_2 : \texttt{T} \quad \Gamma \vdash \texttt{t}_3 : \texttt{T}}{\Gamma \vdash \texttt{if}\ \texttt{t}_1\ \texttt{then}\ \texttt{t}_2\ \texttt{else}\ \texttt{t}_3 : \texttt{T}}$$

Note that $\texttt{t}_2$ and $\texttt{t}_3$ in the if-expression must have the *same* type T!

Definition (T-Zero, T-Succ, T-Pred, T-IsZero)

$$0 : \texttt{Nat} \qquad \frac{\Gamma \vdash \texttt{t}_1 : \texttt{Nat}}{\Gamma \vdash \texttt{succ}\ \texttt{t}_1 : \texttt{Nat}} \qquad \frac{\Gamma \vdash \texttt{t}_1 : \texttt{Nat}}{\Gamma \vdash \texttt{pred}\ \texttt{t}_1 : \texttt{Nat}} \qquad \frac{\Gamma \vdash \texttt{t}_1 : \texttt{Nat}}{\Gamma \vdash \texttt{iszero}\ \texttt{t}_1 : \texttt{Bool}}$$

Typing rules for booleans and numbers

Definition (T-True, T-False, T-If)

$$\texttt{true} : \texttt{Bool} \qquad \texttt{false} : \texttt{Bool} \qquad \frac{\Gamma \vdash \texttt{t}_1 : \texttt{Bool} \quad \Gamma \vdash \texttt{t}_2 : \texttt{T} \quad \Gamma \vdash \texttt{t}_3 : \texttt{T}}{\Gamma \vdash \texttt{if}\ \texttt{t}_1\ \texttt{then}\ \texttt{t}_2\ \texttt{else}\ \texttt{t}_3 : \texttt{T}}$$

Note that $\texttt{t}_2$ and $\texttt{t}_3$ in the if-expression must have the *same* type T!

Definition (T-Zero, T-Succ, T-Pred, T-IsZero)

$$0 : \texttt{Nat} \qquad \frac{\Gamma \vdash \texttt{t}_1 : \texttt{Nat}}{\Gamma \vdash \texttt{succ}\ \texttt{t}_1 : \texttt{Nat}} \qquad \frac{\Gamma \vdash \texttt{t}_1 : \texttt{Nat}}{\Gamma \vdash \texttt{pred}\ \texttt{t}_1 : \texttt{Nat}} \qquad \frac{\Gamma \vdash \texttt{t}_1 : \texttt{Nat}}{\Gamma \vdash \texttt{iszero}\ \texttt{t}_1 : \texttt{Bool}}$$

Deduction example

Let's prove

$$\mathtt{f} : \mathtt{Bool} \to \mathtt{Bool} \vdash \lambda \mathtt{x} : \mathtt{Bool}.\, \mathtt{f}\,(\mathtt{if}\,\mathtt{x}\,\mathtt{then}\,\mathtt{false}\,\mathtt{else}\,\mathtt{x}) : \mathtt{Bool} \to \mathtt{Bool}$$

Proof.

$$\dfrac{\dfrac{\mathtt{f} : \mathtt{Bool} \to \mathtt{Bool} \in \mathtt{f} : \mathtt{Bool} \to \mathtt{Bool}}{\mathtt{f} : \mathtt{Bool} \to \mathtt{Bool} \vdash \mathtt{f} : \mathtt{Bool} \to \mathtt{Bool}}\,T-Var \quad \dfrac{\dfrac{\mathtt{x} : \mathtt{Bool} \in \mathtt{x} : \mathtt{Bool}}{\mathtt{x} : \mathtt{Bool} \vdash \mathtt{x} : \mathtt{Bool}}\,T-Var \quad \dfrac{}{\mathtt{false} : \mathtt{Bool}}\,T-False}{\mathtt{x} : \mathtt{Bool} \vdash \mathtt{if}\,\mathtt{x}\,\mathtt{then}\,\mathtt{false}\,\mathtt{else}\,\mathtt{x} : \mathtt{Bool}}\,T-If}{\dfrac{\mathtt{f} : \mathtt{Bool} \to \mathtt{Bool}, \mathtt{x} : \mathtt{Bool} \vdash \mathtt{f}\,(\mathtt{if}\,\mathtt{x}\,\mathtt{then}\,\mathtt{false}\,\mathtt{else}\,\mathtt{x}) : \mathtt{Bool}}{\mathtt{f} : \mathtt{Bool} \to \mathtt{Bool} \vdash \lambda \mathtt{x} : \mathtt{Bool}.\, \mathtt{f}\,(\mathtt{if}\,\mathtt{x}\,\mathtt{then}\,\mathtt{false}\,\mathtt{else}\,\mathtt{x}) : \mathtt{Bool} \to \mathtt{Bool}}\,T-Abs}\,T-App$$

□

Deduction example

Let's prove

$$\mathtt{f} : \mathtt{Bool} \to \mathtt{Bool} \vdash \lambda \mathtt{x} : \mathtt{Bool}.\ \mathtt{f}\ (\mathtt{if\ x\ then\ false\ else\ x}) : \mathtt{Bool} \to \mathtt{Bool}$$

Proof.

$$\dfrac{\dfrac{\dfrac{\mathtt{f} : \mathtt{Bool} \to \mathtt{Bool} \in \mathtt{f} : \mathtt{Bool} \to \mathtt{Bool}}{\mathtt{f} : \mathtt{Bool} \to \mathtt{Bool} \vdash \mathtt{f} : \mathtt{Bool} \to \mathtt{Bool}}\ T-Var \qquad \dfrac{\dfrac{\mathtt{x} : \mathtt{Bool} \in \mathtt{x} : \mathtt{Bool}}{\mathtt{x} : \mathtt{Bool} \vdash \mathtt{x} : \mathtt{Bool}}\ T-Var \qquad \dfrac{}{\mathtt{false} : \mathtt{Bool}}\ T-False}{\mathtt{x} : \mathtt{Bool} \vdash \mathtt{if\ x\ then\ false\ else\ x} : \mathtt{Bool}}\ T-If}{\mathtt{f} : \mathtt{Bool} \to \mathtt{Bool}, \mathtt{x} : \mathtt{Bool} \vdash \mathtt{f}\ (\mathtt{if\ x\ then\ false\ else\ x}) : \mathtt{Bool}}\ T-App}{\mathtt{f} : \mathtt{Bool} \to \mathtt{Bool} \vdash \lambda \mathtt{x} : \mathtt{Bool}.\ \mathtt{f}\ (\mathtt{if\ x\ then\ false\ else\ x}) : \mathtt{Bool} \to \mathtt{Bool}}\ T-Abs$$

□

Deduction example

Let's prove

$$\mathtt{f} : \mathtt{Bool} \rightarrow \mathtt{Bool} \vdash \lambda \mathtt{x} : \mathtt{Bool}.\, \mathtt{f}\,(\mathtt{if}\ \mathtt{x}\ \mathtt{then}\ \mathtt{false}\ \mathtt{else}\ \mathtt{x}) : \mathtt{Bool} \rightarrow \mathtt{Bool}$$

Proof.

$$\dfrac{\dfrac{\dfrac{\mathtt{f} : \mathtt{Bool} \rightarrow \mathtt{Bool} \in \mathtt{f} : \mathtt{Bool} \rightarrow \mathtt{Bool}}{\mathtt{f} : \mathtt{Bool} \rightarrow \mathtt{Bool} \vdash \mathtt{f} : \mathtt{Bool} \rightarrow \mathtt{Bool}}\,T-Var \quad \dfrac{\dfrac{\mathtt{x} : \mathtt{Bool} \in \mathtt{x} : \mathtt{Bool}}{\mathtt{x} : \mathtt{Bool} \vdash \mathtt{x} : \mathtt{Bool}}\,T-Var \quad \dfrac{}{\mathtt{false} : \mathtt{Bool}}\,T-False}{\mathtt{x} : \mathtt{Bool} \vdash \mathtt{if}\ \mathtt{x}\ \mathtt{then}\ \mathtt{false}\ \mathtt{else}\ \mathtt{x} : \mathtt{Bool}}\,T-If}{\mathtt{f} : \mathtt{Bool} \rightarrow \mathtt{Bool}, \mathtt{x} : \mathtt{Bool} \vdash \mathtt{f}\,(\mathtt{if}\ \mathtt{x}\ \mathtt{then}\ \mathtt{false}\ \mathtt{else}\ \mathtt{x}) : \mathtt{Bool}}\,T-App}{\mathtt{f} : \mathtt{Bool} \rightarrow \mathtt{Bool} \vdash \lambda \mathtt{x} : \mathtt{Bool}.\, \mathtt{f}\,(\mathtt{if}\ \mathtt{x}\ \mathtt{then}\ \mathtt{false}\ \mathtt{else}\ \mathtt{x}) : \mathtt{Bool} \rightarrow \mathtt{Bool}}\,T-Abs$$

□

Deduction example

Let's prove

$$\mathtt{f} : \mathtt{Bool} \to \mathtt{Bool} \vdash \lambda \mathtt{x} : \mathtt{Bool}.\ \mathtt{f}\ (\mathtt{if}\ \mathtt{x}\ \mathtt{then}\ \mathtt{false}\ \mathtt{else}\ \mathtt{x}) : \mathtt{Bool} \to \mathtt{Bool}$$

Proof.

$$\dfrac{\dfrac{\dfrac{\mathtt{f} : \mathtt{Bool} \to \mathtt{Bool} \in \mathtt{f} : \mathtt{Bool} \to \mathtt{Bool}}{\mathtt{f} : \mathtt{Bool} \to \mathtt{Bool} \vdash \mathtt{f} : \mathtt{Bool} \to \mathtt{Bool}}\ T-Var \qquad \dfrac{\dfrac{\mathtt{x} : \mathtt{Bool} \in \mathtt{x} : \mathtt{Bool}}{\mathtt{x} : \mathtt{Bool} \vdash \mathtt{x} : \mathtt{Bool}}\ T-Var \quad \dfrac{}{\mathtt{false} : \mathtt{Bool}}\ T-False}{\mathtt{x} : \mathtt{Bool} \vdash \mathtt{if}\ \mathtt{x}\ \mathtt{then}\ \mathtt{false}\ \mathtt{else}\ \mathtt{x} : \mathtt{Bool}}\ T-If}{\mathtt{f} : \mathtt{Bool} \to \mathtt{Bool}, \mathtt{x} : \mathtt{Bool} \vdash \mathtt{f}\ (\mathtt{if}\ \mathtt{x}\ \mathtt{then}\ \mathtt{false}\ \mathtt{else}\ \mathtt{x}) : \mathtt{Bool}}\ T-App}{\mathtt{f} : \mathtt{Bool} \to \mathtt{Bool} \vdash \lambda \mathtt{x} : \mathtt{Bool}.\ \mathtt{f}\ (\mathtt{if}\ \mathtt{x}\ \mathtt{then}\ \mathtt{false}\ \mathtt{else}\ \mathtt{x}) : \mathtt{Bool} \to \mathtt{Bool}}\ T-Abs$$

□

Deduction example

Let's prove

$$\mathtt{f} : \mathtt{Bool} \to \mathtt{Bool} \vdash \lambda \mathtt{x} : \mathtt{Bool}.\ \mathtt{f}\ (\mathtt{if\ x\ then\ false\ else\ x}) : \mathtt{Bool} \to \mathtt{Bool}$$

Proof.

$$\dfrac{\dfrac{\dfrac{\mathtt{f} : \mathtt{Bool} \to \mathtt{Bool} \in \mathtt{f} : \mathtt{Bool} \to \mathtt{Bool}}{\mathtt{f} : \mathtt{Bool} \to \mathtt{Bool} \vdash \mathtt{f} : \mathtt{Bool} \to \mathtt{Bool}}\ T-Var \qquad \dfrac{\dfrac{\mathtt{x} : \mathtt{Bool} \in \mathtt{x} : \mathtt{Bool}}{\mathtt{x} : \mathtt{Bool} \vdash \mathtt{x} : \mathtt{Bool}}\ T-Var \quad \dfrac{}{\mathtt{false} : \mathtt{Bool}}\ T-False}{\mathtt{x} : \mathtt{Bool} \vdash \mathtt{if\ x\ then\ false\ else\ x} : \mathtt{Bool}}\ T-If}{\mathtt{f} : \mathtt{Bool} \to \mathtt{Bool}, \mathtt{x} : \mathtt{Bool} \vdash \mathtt{f}\ (\mathtt{if\ x\ then\ false\ else\ x}) : \mathtt{Bool}}\ T-App$$

$$\dfrac{\mathtt{f} : \mathtt{Bool} \to \mathtt{Bool}, \mathtt{x} : \mathtt{Bool} \vdash \mathtt{f}\ (\mathtt{if\ x\ then\ false\ else\ x}) : \mathtt{Bool}}{\mathtt{f} : \mathtt{Bool} \to \mathtt{Bool} \vdash \lambda \mathtt{x} : \mathtt{Bool}.\ \mathtt{f}\ (\mathtt{if\ x\ then\ false\ else\ x}) : \mathtt{Bool} \to \mathtt{Bool}}\ T-Abs$$

□

Deduction example

Let's prove

$$\mathtt{f}:\mathtt{Bool}\to\mathtt{Bool}\vdash\lambda\mathtt{x}:\mathtt{Bool}.\,\mathtt{f}\,(\mathtt{if\ x\ then\ false\ else\ x}):\mathtt{Bool}\to\mathtt{Bool}$$

Proof.

$$\dfrac{\dfrac{\dfrac{\mathtt{f}:\mathtt{Bool}\to\mathtt{Bool}\in\mathtt{f}:\mathtt{Bool}\to\mathtt{Bool}}{\mathtt{f}:\mathtt{Bool}\to\mathtt{Bool}\vdash\mathtt{f}:\mathtt{Bool}\to\mathtt{Bool}}\ T-Var\quad\dfrac{\dfrac{\mathtt{x}:\mathtt{Bool}\in\mathtt{x}:\mathtt{Bool}}{\mathtt{x}:\mathtt{Bool}\vdash\mathtt{x}:\mathtt{Bool}}\ T-Var\quad\dfrac{}{\mathtt{false}:\mathtt{Bool}}\ T-False}{\mathtt{x}:\mathtt{Bool}\vdash\mathtt{if\ x\ then\ false\ else\ x}:\mathtt{Bool}}\ T-If}{\mathtt{f}:\mathtt{Bool}\to\mathtt{Bool},\mathtt{x}:\mathtt{Bool}\vdash\mathtt{f}\,(\mathtt{if\ x\ then\ false\ else\ x}):\mathtt{Bool}}\ T-App}{\mathtt{f}:\mathtt{Bool}\to\mathtt{Bool}\vdash\lambda\mathtt{x}:\mathtt{Bool}.\,\mathtt{f}\,(\mathtt{if\ x\ then\ false\ else\ x}):\mathtt{Bool}\to\mathtt{Bool}}\ T-Abs$$

□

Deduction example

Let's prove

$$\mathtt{f} : \mathtt{Bool} \to \mathtt{Bool} \vdash \lambda \mathtt{x} : \mathtt{Bool}.\, \mathtt{f}\,(\mathtt{if\ x\ then\ false\ else\ x}) : \mathtt{Bool} \to \mathtt{Bool}$$

Proof.

$$\dfrac{\dfrac{\dfrac{\mathtt{f} : \mathtt{Bool} \to \mathtt{Bool} \in \mathtt{f} : \mathtt{Bool} \to \mathtt{Bool}}{\mathtt{f} : \mathtt{Bool} \to \mathtt{Bool} \vdash \mathtt{f} : \mathtt{Bool} \to \mathtt{Bool}}\ T-Var \qquad \dfrac{\dfrac{\mathtt{x} : \mathtt{Bool} \in \mathtt{x} : \mathtt{Bool}}{\mathtt{x} : \mathtt{Bool} \vdash \mathtt{x} : \mathtt{Bool}}\ T-Var \quad \dfrac{}{\mathtt{false} : \mathtt{Bool}}\ T-False}{\mathtt{x} : \mathtt{Bool} \vdash \mathtt{if\ x\ then\ false\ else\ x} : \mathtt{Bool}}\ T-If}{\mathtt{f} : \mathtt{Bool} \to \mathtt{Bool}, \mathtt{x} : \mathtt{Bool} \vdash \mathtt{f}\,(\mathtt{if\ x\ then\ false\ else\ x}) : \mathtt{Bool}}\ T-App}{\mathtt{f} : \mathtt{Bool} \to \mathtt{Bool} \vdash \lambda \mathtt{x} : \mathtt{Bool}.\, \mathtt{f}\,(\mathtt{if\ x\ then\ false\ else\ x}) : \mathtt{Bool} \to \mathtt{Bool}}\ T-Abs$$

□

Properties of typing

Two important theorems can be shown for $\lambda^{\rightarrow}$ by structural induction:

Theorem (Progress)

Suppose $\mathtt{t}$ *is a closed, well-typed term (that is,* $\vdash \mathtt{t} : \mathtt{T}$ *for some* $\mathtt{T}$*). Then either* $\mathtt{t}$ *is a value or else there is some* $\mathtt{t}'$ *with* $\mathtt{t} \rightarrow \mathtt{t}'$*.*

Theorem (Preservation)

If $\Gamma \vdash \mathtt{t} : \mathtt{T}$ *and* $\mathtt{t} \rightarrow \mathtt{t}'$*, then* $\Gamma \vdash \mathtt{t}' : \mathtt{T}$*.*

Properties of typing

Two important theorems can be shown for $\lambda^{\rightarrow}$ by structural induction:

Theorem (Progress)

Suppose $\texttt{t}$ is a closed, well-typed term (that is, $\vdash \texttt{t} : \texttt{T}$ for some $\texttt{T}$). Then either $\texttt{t}$ is a value or else there is some $\texttt{t}'$ with $\texttt{t} \rightarrow \texttt{t}'$.

Theorem (Preservation)

If $\Gamma \vdash \texttt{t} : \texttt{T}$ and $\texttt{t} \rightarrow \texttt{t}'$, then $\Gamma \vdash \texttt{t}' : \texttt{T}$.

Properties of typing

Two important theorems can be shown for $\lambda^{\rightarrow}$ by structural induction:

Theorem (Progress)

Suppose $\mathtt{t}$ *is a closed, well-typed term (that is,* $\vdash \mathtt{t} : \mathtt{T}$ *for some* $\mathtt{T}$*). Then either* $\mathtt{t}$ *is a value or else there is some* $\mathtt{t'}$ *with* $\mathtt{t} \rightarrow \mathtt{t}'$*.*

Theorem (Preservation)

If $\Gamma \vdash \mathtt{t} : \mathtt{T}$ *and* $\mathtt{t} \rightarrow \mathtt{t}'$*, then* $\Gamma \vdash \mathtt{t}' : \mathtt{T}$*.*

Properties of typing

What does it mean?

- **Progress:**

  Every well-typed term can be reduced to a value
- **Preservation:**

  Every well-typed term evaluates to a well-typed term with the same type
- progress+preservation=***type safety***

  $\Rightarrow$ well-typed terms never get stuck during evaluation!

Properties of typing

- Another property that can be shown:

  type erasure does not influence evaluation $\Rightarrow$ Types can be (and are often!) removed during compilation, if everything is ok

- Reverse action – *type reconstruction*:

  finding a possible type of a term with incomplete type annotations

- if the reconstruction possible, the term is *typable*, if not:

  either invalid term or insufficient information

Properties of typing

- Another property that can be shown:

  type erasure does not influence evaluation $\Rightarrow$ Types can be (and are often!) removed during compilation, if everything is ok
- Reverse action – *type reconstruction*:

  finding a possible type of a term with incomplete type annotations
- if the reconstruction possible, the term is *typable*, if not:

  either invalid term or insufficient information

$$\texttt{doubleNat} = \lambda \texttt{f} : \texttt{Nat} \to \texttt{Nat}.\ \lambda \texttt{x} : \texttt{Nat}.\ \texttt{f}\ (\texttt{f}\ \texttt{x})$$

$$\texttt{doubleBool} = \lambda \texttt{f} : \texttt{Bool} \to \texttt{Bool}.\ \lambda \texttt{x} : \texttt{Bool}.\ \texttt{f}\ (\texttt{f}\ \texttt{x})$$

$$\texttt{doubleAll} = \ ?$$

$$\texttt{doubleNat} = \lambda \texttt{f} : \texttt{Nat} \to \texttt{Nat}.\ \lambda \texttt{x} : \texttt{Nat}.\ \texttt{f}\ (\texttt{f}\ \texttt{x})$$

$$\texttt{doubleBool} = \lambda \texttt{f} : \texttt{Bool} \to \texttt{Bool}.\ \lambda \texttt{x} : \texttt{Bool}.\ \texttt{f}\ (\texttt{f}\ \texttt{x})$$

$$\texttt{doubleAll} = \ ?$$

$$\mathtt{doubleNat} = \lambda \mathtt{f} : \mathtt{Nat} \to \mathtt{Nat}.\, \lambda \mathtt{x} : \mathtt{Nat}.\, \mathtt{f}\ (\mathtt{f}\ \mathtt{x})$$

$$\mathtt{doubleBool} = \lambda \mathtt{f} : \mathtt{Bool} \to \mathtt{Bool}.\, \lambda \mathtt{x} : \mathtt{Bool}.\, \mathtt{f}\ (\mathtt{f}\ \mathtt{x})$$

$$\mathtt{doubleAll} = \ ?$$

Problem:

For each type we need to duplicate identical code with other type annotations!

$$\texttt{doubleNat} = \lambda \texttt{f} : \texttt{Nat} \to \texttt{Nat}.\ \lambda \texttt{x} : \texttt{Nat}.\ \texttt{f}\ (\texttt{f}\ \texttt{x})$$

$$\texttt{doubleBool} = \lambda \texttt{f} : \texttt{Bool} \to \texttt{Bool}.\ \lambda \texttt{x} : \texttt{Bool}.\ \texttt{f}\ (\texttt{f}\ \texttt{x})$$

$$\texttt{doubleAll} = \ ?$$

Problem:

For each type we need to duplicate identical code with other type annotations!

We want something like Java Generics / C++ Templates

$\Rightarrow$ we need to extend $\lambda^{\to}$ with *parametric polymorphism*

$$\texttt{doubleNat} = \lambda \texttt{f} : \texttt{Nat} \to \texttt{Nat}.\, \lambda \texttt{x} : \texttt{Nat}.\, \texttt{f}\,(\texttt{f}\ \texttt{x})$$

$$\texttt{doubleBool} = \lambda \texttt{f} : \texttt{Bool} \to \texttt{Bool}.\, \lambda \texttt{x} : \texttt{Bool}.\, \texttt{f}\,(\texttt{f}\ \texttt{x})$$

$$\texttt{doubleAll} = \ ?$$

Problem:

For each type we need to duplicate identical code with other type annotations!

We want something like Java Generics / C++ Templates

$\Rightarrow$ we need to extend $\lambda^{\to}$ with *parametric polymorphism*

$\rightsquigarrow$ System F (Girard, 1972)

System F

Definition (Syntax of System F)

$\mathtt{t} ::=$	$\mathtt{x}$	*variable*
	$\lambda\mathtt{x} : \mathtt{T}.\, \mathtt{t}$	*abstraction*
	$\mathtt{t}$	*application*
	$\lambda\mathtt{X}.\, \mathtt{t}$	**type abstraction**
	$\mathtt{t}[\mathtt{T}]$	**type application**

uppercase letters = type variables, lowercase letters = terms

System F

- type abstraction/application works similar to normal, but we substitute type variables: $(\lambda X.\ t_{12})\ [T_2] \to [X \mapsto T_2]t_{12}$
- before annotated types had to be concrete, now they are abstracted
  $\Rightarrow$ we need new types and typing rules to express this
- type abstractions get a *universal type* of the form $\forall X.T$
- now we have two different type constructors: $\to$ and $\forall$
  $\Rightarrow$ *second-order lambda calculus*

System F

- type abstraction/application works similar to normal, but we substitute type variables: $(\lambda \mathtt{X}.\ \mathtt{t}_{12})\ [\mathtt{T}_2] \rightarrow [\mathtt{X} \mapsto \mathtt{T}_2]\mathtt{t}_{12}$
- before annotated types had to be concrete, now they are abstracted
  $\Rightarrow$ we need new types and typing rules to express this
- type abstractions get a *universal type* of the form $\forall\mathtt{X}.\mathtt{T}$
- now we have two different type constructors: $\rightarrow$ and $\forall$
  $\Rightarrow$ *second-order lambda calculus*

System F

- type abstraction/application works similar to normal, but we substitute type variables: $(\lambda X.\ t_{12})\ [T_2] \rightarrow [X \mapsto T_2]t_{12}$
- before annotated types had to be concrete, now they are abstracted
  $\Rightarrow$ we need new types and typing rules to express this
- type abstractions get a *universal type* of the form $\forall X.T$
- now we have two different type constructors: $\rightarrow$ and $\forall$
  $\Rightarrow$ *second-order lambda calculus*

Rules for universal types

Definition (Typing of type abstractions (T-TAbs))

$$\frac{\Gamma, \mathtt{X} \vdash \mathtt{t}_2 : \mathtt{T}_2}{\Gamma \vdash \lambda \mathtt{X}.\ \mathtt{t}_2 : \forall \mathtt{X}.\mathtt{T}_2}$$

Definition (Typing of type applications (T-TApp))

$$\frac{\Gamma \vdash \mathtt{t}_1 : \forall \mathtt{X}.\mathtt{T}_{12}}{\Gamma \vdash \mathtt{t}_1\ [\mathtt{T}_2] : [\mathtt{X} \mapsto \mathtt{T}_2]\mathtt{T}_{12}}$$

Rules for universal types

Definition (Typing of type abstractions (T-TAbs))

$$\frac{\Gamma, \mathtt{X} \vdash \mathtt{t}_2 : \mathtt{T}_2}{\Gamma \vdash \lambda \mathtt{X}.\ \mathtt{t}_2 : \forall \mathtt{X}.\mathtt{T}_2}$$

Definition (Typing of type applications (T-TApp))

$$\frac{\Gamma \vdash \mathtt{t}_1 : \forall \mathtt{X}.\mathtt{T}_{12}}{\Gamma \vdash \mathtt{t}_1\ [\mathtt{T}_2] : [\mathtt{X} \mapsto \mathtt{T}_2]\mathtt{T}_{12}}$$

Examples

$\mathtt{id} = \lambda \mathtt{X}.\ \lambda \mathtt{x} : \mathtt{X}.\ \mathtt{x}$ type $\forall X.X \to X$

$\mathtt{idNat} = \mathtt{id}\ [\mathtt{Nat}] = \lambda \mathtt{x} : \mathtt{Nat}.\ \mathtt{x}$ type $Nat \to Nat$

$$\mathtt{double} = \lambda \mathtt{X}.\ \lambda \mathtt{f} : \mathtt{X} \to \mathtt{X}.\ \lambda \mathtt{x} : \mathtt{X}.\ \mathtt{f}\ (\mathtt{f}\ \mathtt{x})$$

type $\forall X.(X \to X) \to X \to X$

$$\mathtt{dblBool} = \mathtt{double}\ [\mathtt{Bool}]$$

$$= \lambda \mathtt{f} : \mathtt{Bool} \to \mathtt{Bool}.\ \lambda \mathtt{x} : \mathtt{Bool}.\ \mathtt{f}\ (\mathtt{f}\ \mathtt{x})$$

type $(Bool \to Bool) \to Bool \to Bool$

Examples

$$\texttt{id} = \lambda \texttt{X}.\ \lambda \texttt{x} : \texttt{X}.\ \texttt{x} \qquad \text{type } \forall X.X \to X$$

$$\texttt{idNat} = \texttt{id}\ [\texttt{Nat}] = \lambda \texttt{x} : \texttt{Nat}.\ \texttt{x} \qquad \text{type Nat} \to \text{Nat}$$

$$\texttt{double} = \lambda \texttt{X}.\ \lambda \texttt{f} : \texttt{X} \to \texttt{X}.\ \lambda \texttt{x} : \texttt{X}.\ \texttt{f}\ (\texttt{f}\ \texttt{x})$$

$$\text{type } \forall X.(X \to X) \to X \to X$$

$$\texttt{dblBool} = \texttt{double}\ [\texttt{Bool}]$$

$$= \lambda \texttt{f} : \texttt{Bool} \to \texttt{Bool}.\ \lambda \texttt{x} : \texttt{Bool}.\ \texttt{f}\ (\texttt{f}\ \texttt{x})$$

$$\text{type } (\text{Bool} \to \text{Bool}) \to \text{Bool} \to \text{Bool}$$

Examples

$$\mathtt{id} = \lambda \mathtt{X}.\ \lambda \mathtt{x} : \mathtt{X}.\ \mathtt{x} \qquad \text{type } \forall X.X \to X$$

$$\mathtt{idNat} = \mathtt{id}\ [\mathtt{Nat}] = \lambda \mathtt{x} : \mathtt{Nat}.\ \mathtt{x} \qquad \text{type Nat} \to \text{Nat}$$

$$\mathtt{double} = \lambda \mathtt{X}.\ \lambda \mathtt{f} : \mathtt{X} \to \mathtt{X}.\ \lambda \mathtt{x} : \mathtt{X}.\ \mathtt{f}\ (\mathtt{f}\ \mathtt{x})$$

$$\text{type } \forall X.(X \to X) \to X \to X$$

$$\begin{aligned}\mathtt{dblBool} &= \mathtt{double}\ [\mathtt{Bool}] \\ &= \lambda \mathtt{f} : \mathtt{Bool} \to \mathtt{Bool}.\ \lambda \mathtt{x} : \mathtt{Bool}.\ \mathtt{f}\ (\mathtt{f}\ \mathtt{x})\end{aligned}$$

$$\text{type } (\text{Bool} \to \text{Bool}) \to \text{Bool} \to \text{Bool}$$

Examples

$$\texttt{id} = \lambda \texttt{X}.\ \lambda \texttt{x} : \texttt{X}.\ \texttt{x} \qquad \text{type } \forall X.X \to X$$

$$\texttt{idNat} = \texttt{id}\ [\texttt{Nat}] = \lambda \texttt{x} : \texttt{Nat}.\ \texttt{x} \qquad \text{type Nat} \to \text{Nat}$$

$$\texttt{double} = \lambda \texttt{X}.\ \lambda \texttt{f} : \texttt{X} \to \texttt{X}.\ \lambda \texttt{x} : \texttt{X}.\ \texttt{f}\ (\texttt{f}\ \texttt{x})$$

$$\text{type } \forall X.(X \to X) \to X \to X$$

$$\texttt{dblBool} = \texttt{double}\ [\texttt{Bool}]$$

$$= \lambda \texttt{f} : \texttt{Bool} \to \texttt{Bool}.\ \lambda \texttt{x} : \texttt{Bool}.\ \texttt{f}\ (\texttt{f}\ \texttt{x})$$

$$\text{type } (\text{Bool} \to \text{Bool}) \to \text{Bool} \to \text{Bool}$$

Examples

$$\texttt{quad} = \lambda \texttt{X}.\ \texttt{double}\ [\texttt{X} \to \texttt{X}]\ (\texttt{double}\ [\texttt{X}]) \quad \textbf{type}\ \forall X.(X \to X) \to X \to X$$

Examples

$$\texttt{quad} = \lambda \texttt{X}.\ \texttt{double}\ [\texttt{X} \to \texttt{X}]\ (\texttt{double}\ [\texttt{X}]) \quad \textbf{type}\ \forall X.(X \to X) \to X \to X$$

Wait, what?!

Examples

$$\texttt{quad} = \lambda \texttt{X}.\ \texttt{double}\ [\texttt{X} \to \texttt{X}]\ (\texttt{double}\ [\texttt{X}]) \quad \textbf{type}\ \forall X.(X \to X) \to X \to X$$

Wait, what?! Yes, it is correct. Let's evaluate it:

$$\begin{aligned}
\texttt{quad} &= \lambda \texttt{X}.\ \texttt{double}\ [\texttt{X} \to \texttt{X}]\ (\texttt{double}\ [\texttt{X}]) \\
&= \lambda \texttt{X}.\ (\lambda \texttt{f} : (\texttt{X} \to \texttt{X}) \to \texttt{X} \to \texttt{X}.\ \lambda \texttt{a} : \texttt{X} \to \texttt{X}.\ \texttt{f}\ (\texttt{f}\ \texttt{a}))\ (\texttt{double}\ [\texttt{X}]) \\
&= \lambda \texttt{X}.\ \lambda \texttt{a} : \texttt{X} \to \texttt{X}.\ \texttt{double}\ [\texttt{X}]\ (\texttt{double}\ [\texttt{X}]\ \texttt{a}) \\
&= \lambda \texttt{X}.\ \lambda \texttt{a} : \texttt{X} \to \texttt{X}.\ (\lambda \texttt{g} : \texttt{X} \to \texttt{X}.\ \lambda \texttt{b} : \texttt{X}.\ \texttt{g}\ (\texttt{g}\ \texttt{b}))\ (\texttt{double}\ [\texttt{X}]\ \texttt{a}) \\
&= \lambda \texttt{X}.\ \lambda \texttt{a} : \texttt{X} \to \texttt{X}.\ \lambda \texttt{b} : \texttt{X}.\ \texttt{double}\ [\texttt{X}]\ \texttt{a}\ (\texttt{double}\ [\texttt{X}]\ \texttt{a}\ \texttt{b}) \\
&= \lambda \texttt{X}.\ \lambda \texttt{a} : \texttt{X} \to \texttt{X}.\ \lambda \texttt{b} : \texttt{X}.\ \texttt{a}\ (\texttt{a}\ (\texttt{a}\ (\texttt{a}\ \texttt{b})))
\end{aligned}$$

Examples

$$\texttt{quad} = \lambda \texttt{X}.\ \texttt{double}\ [\texttt{X} \to \texttt{X}]\ (\texttt{double}\ [\texttt{X}]) \quad \text{type } \forall X.(X \to X) \to X \to X$$

Wait, what?! Yes, it is correct. Let's evaluate it:

$$\begin{aligned}
\texttt{quad} &= \lambda \texttt{X}.\ \texttt{double}\ [\texttt{X} \to \texttt{X}]\ (\texttt{double}\ [\texttt{X}]) \\
&= \lambda \texttt{X}.\ (\lambda \texttt{f} : (\texttt{X} \to \texttt{X}) \to \texttt{X} \to \texttt{X}.\ \lambda \texttt{a} : \texttt{X} \to \texttt{X}.\ \texttt{f}\ (\texttt{f}\ \texttt{a}))\ (\texttt{double}\ [\texttt{X}]) \\
&= \lambda \texttt{X}.\ \lambda \texttt{a} : \texttt{X} \to \texttt{X}.\ \texttt{double}\ [\texttt{X}]\ (\texttt{double}\ [\texttt{X}]\ \texttt{a}) \\
&= \lambda \texttt{X}.\ \lambda \texttt{a} : \texttt{X} \to \texttt{X}.\ (\lambda \texttt{g} : \texttt{X} \to \texttt{X}.\ \lambda \texttt{b} : \texttt{X}.\ \texttt{g}\ (\texttt{g}\ \texttt{b}))\ (\texttt{double}\ [\texttt{X}]\ \texttt{a}) \\
&= \lambda \texttt{X}.\ \lambda \texttt{a} : \texttt{X} \to \texttt{X}.\ \lambda \texttt{b} : \texttt{X}.\ \texttt{double}\ [\texttt{X}]\ \texttt{a}\ (\texttt{double}\ [\texttt{X}]\ \texttt{a}\ \texttt{b}) \\
&= \lambda \texttt{X}.\ \lambda \texttt{a} : \texttt{X} \to \texttt{X}.\ \lambda \texttt{b} : \texttt{X}.\ \texttt{a}\ (\texttt{a}\ (\texttt{a}\ (\texttt{a}\ \texttt{b})))
\end{aligned}$$

Examples

$$\texttt{quad} = \lambda X.\ \texttt{double}\ [X \to X]\ (\texttt{double}\ [X]) \quad \textbf{type}\ \forall X.(X \to X) \to X \to X$$

Wait, what?! Yes, it is correct. Let's evaluate it:

$$\begin{aligned}
\texttt{quad} &= \lambda X.\ \texttt{double}\ [X \to X]\ (\texttt{double}\ [X]) \\
&= \lambda X.\ (\lambda f : (X \to X) \to X \to X.\ \lambda a : X \to X.\ f\ (f\ a))\ (\texttt{double}\ [X]) \\
&= \lambda X.\ \lambda a : X \to X.\ \texttt{double}\ [X]\ (\texttt{double}\ [X]\ a) \\
&= \lambda X.\ \lambda a : X \to X.\ (\lambda g : X \to X.\ \lambda b : X.\ g\ (g\ b))\ (\texttt{double}\ [X]\ a) \\
&= \lambda X.\ \lambda a : X \to X.\ \lambda b : X.\ \texttt{double}\ [X]\ a\ (\texttt{double}\ [X]\ a\ b) \\
&= \lambda X.\ \lambda a : X \to X.\ \lambda b : X.\ a\ (a\ (a\ (a\ b)))
\end{aligned}$$

Examples

$$\texttt{quad} = \lambda X.\ \texttt{double}\ [X \to X]\ (\texttt{double}\ [X]) \quad \textbf{type}\ \forall X.(X \to X) \to X \to X$$

Wait, what?! Yes, it is correct. Let's evaluate it:

$$\begin{aligned}
\texttt{quad} &= \lambda X.\ \texttt{double}\ [X \to X]\ (\texttt{double}\ [X]) \\
&= \lambda X.\ (\lambda f : (X \to X) \to X \to X.\ \lambda a : X \to X.\ f\ (f\ a))\ (\texttt{double}\ [X]) \\
&= \lambda X.\ \lambda a : X \to X.\ \texttt{double}\ [X]\ (\texttt{double}\ [X]\ a) \\
&= \lambda X.\ \lambda a : X \to X.\ (\lambda g : X \to X.\ \lambda b : X.\ g\ (g\ b))\ (\texttt{double}\ [X]\ a) \\
&= \lambda X.\ \lambda a : X \to X.\ \lambda b : X.\ \texttt{double}\ [X]\ a\ (\texttt{double}\ [X]\ a\ b) \\
&= \lambda X.\ \lambda a : X \to X.\ \lambda b : X.\ a\ (a\ (a\ (a\ b)))
\end{aligned}$$

Examples

$$\texttt{quad} = \lambda \texttt{X}.\ \texttt{double}\ [\texttt{X} \to \texttt{X}]\ (\texttt{double}\ [\texttt{X}]) \quad \textbf{type}\ \forall X.(X \to X) \to X \to X$$

Wait, what?! Yes, it is correct. Let's evaluate it:

$$\begin{aligned}
\texttt{quad} &= \lambda \texttt{X}.\ \texttt{double}\ [\texttt{X} \to \texttt{X}]\ (\texttt{double}\ [\texttt{X}]) \\
&= \lambda \texttt{X}.\ (\lambda \texttt{f} : (\texttt{X} \to \texttt{X}) \to \texttt{X} \to \texttt{X}.\ \lambda \texttt{a} : \texttt{X} \to \texttt{X}.\ \texttt{f}\ (\texttt{f}\ \texttt{a}))\ (\texttt{double}\ [\texttt{X}]) \\
&= \lambda \texttt{X}.\ \lambda \texttt{a} : \texttt{X} \to \texttt{X}.\ \texttt{double}\ [\texttt{X}]\ (\texttt{double}\ [\texttt{X}]\ \texttt{a}) \\
&= \lambda \texttt{X}.\ \lambda \texttt{a} : \texttt{X} \to \texttt{X}.\ (\lambda \texttt{g} : \texttt{X} \to \texttt{X}.\ \lambda \texttt{b} : \texttt{X}.\ \texttt{g}\ (\texttt{g}\ \texttt{b}))\ (\texttt{double}\ [\texttt{X}]\ \texttt{a}) \\
&= \lambda \texttt{X}.\ \lambda \texttt{a} : \texttt{X} \to \texttt{X}.\ \lambda \texttt{b} : \texttt{X}.\ \texttt{double}\ [\texttt{X}]\ \texttt{a}\ (\texttt{double}\ [\texttt{X}]\ \texttt{a}\ \texttt{b}) \\
&= \lambda \texttt{X}.\ \lambda \texttt{a} : \texttt{X} \to \texttt{X}.\ \lambda \texttt{b} : \texttt{X}.\ \texttt{a}\ (\texttt{a}\ (\texttt{a}\ (\texttt{a}\ \texttt{b})))
\end{aligned}$$

Examples

$$\texttt{quad} = \lambda\texttt{X}.\ \texttt{double}\ [\texttt{X} \to \texttt{X}]\ (\texttt{double}\ [\texttt{X}]) \quad \textbf{type}\ \forall X.(X \to X) \to X \to X$$

Wait, what?! Yes, it is correct. Let's evaluate it:

$$\begin{aligned}
\texttt{quad} &= \lambda\texttt{X}.\ \texttt{double}\ [\texttt{X} \to \texttt{X}]\ (\texttt{double}\ [\texttt{X}]) \\
&= \lambda\texttt{X}.\ (\lambda\texttt{f} : (\texttt{X} \to \texttt{X}) \to \texttt{X} \to \texttt{X}.\ \lambda\texttt{a} : \texttt{X} \to \texttt{X}.\ \texttt{f}\ (\texttt{f}\ \texttt{a}))\ (\texttt{double}\ [\texttt{X}]) \\
&= \lambda\texttt{X}.\ \lambda\texttt{a} : \texttt{X} \to \texttt{X}.\ \texttt{double}\ [\texttt{X}]\ (\texttt{double}\ [\texttt{X}]\ \texttt{a}) \\
&= \lambda\texttt{X}.\ \lambda\texttt{a} : \texttt{X} \to \texttt{X}.\ (\lambda\texttt{g} : \texttt{X} \to \texttt{X}.\ \lambda\texttt{b} : \texttt{X}.\ \texttt{g}\ (\texttt{g}\ \texttt{b}))\ (\texttt{double}\ [\texttt{X}]\ \texttt{a}) \\
&= \lambda\texttt{X}.\ \lambda\texttt{a} : \texttt{X} \to \texttt{X}.\ \lambda\texttt{b} : \texttt{X}.\ \texttt{double}\ [\texttt{X}]\ \texttt{a}\ (\texttt{double}\ [\texttt{X}]\ \texttt{a}\ \texttt{b}) \\
&= \lambda\texttt{X}.\ \lambda\texttt{a} : \texttt{X} \to \texttt{X}.\ \lambda\texttt{b} : \texttt{X}.\ \texttt{a}\ (\texttt{a}\ (\texttt{a}\ (\texttt{a}\ \texttt{b})))
\end{aligned}$$

System F

- As you see, parametric polymorphism is very expressive
- Haskell programs desugar to an ext. System F form during compilation
- preservation and progress theorems still hold in System F $\Rightarrow$ **type-safe**
- **type reconstruction undecidable** $\Rightarrow$ not all annotations can be omitted
- Languages based on System F have artificial restrictions on valid terms to keep partial reconstruction possible

System F

- As you see, parametric polymorphism is very expressive
- Haskell programs desugar to an ext. System F form during compilation
- preservation and progress theorems still hold in System F ⇒ **type-safe**
- **type reconstruction undecidable** ⇒ not all annotations can be omitted
- Languages based on System F have artificial restrictions on valid terms to keep partial reconstruction possible

Type theory and OOP

- Important branch: λ-calculi with *subtyping* (Reynolds, Cardelli (1980's))
- Theoretical foundation of inheritance in OOP
- Extension: *Subtyping relation* with new set of deduction rules
- Says which types can be treated as more general types

  $\Rightarrow$ Functions can ignore specialisation and work on more inputs
- Efforts to prove type safety of Java (first by Drossopoulou, Eisenbach and Khurshid (1999))

  $\Rightarrow$ using calculi with subtyping which resemble Java subsets

Type theory and OOP

- Important branch: λ-calculi with *subtyping* (Reynolds, Cardelli (1980's))
- Theoretical foundation of inheritance in OOP
- Extension: *Subtyping relation* with new set of deduction rules
- Says which types can be treated as more general types

  $\Rightarrow$ Functions can ignore specialisation and work on more inputs
- Efforts to prove type safety of Java (first by Drossopoulou, Eisenbach and Khurshid (1999))

  $\Rightarrow$ using calculi with subtyping which resemble Java subsets

Type theory and OOP

- Important branch: λ-calculi with *subtyping* (Reynolds, Cardelli (1980's))
- Theoretical foundation of inheritance in OOP
- Extension: *Subtyping relation* with new set of deduction rules
- Says which types can be treated as more general types

  $\Rightarrow$ Functions can ignore specialisation and work on more inputs
- Efforts to prove type safety of Java (first by Drossopoulou, Eisenbach and Khurshid (1999))

  $\Rightarrow$ using calculi with subtyping which resemble Java subsets

Type theory and logic

- *Curry-Howard-Correspondence:* (Curry (1958), Howard (1980))
  isomorphism: types ≈ propositions, terms ≈ proofs!
  ⇒ Connection between constructive logic and computer science
- E.g. used for tools like *Coq* – interactive theorem prover
- Helps the user formulating assertions and finding proofs
- proof-checking = type-checking the program!
- Such tools often based on calculi with *dependent types*
  ⇒ types like $\texttt{Array}\ \texttt{n} \to \texttt{Array}\ (\texttt{n}+\texttt{1})$ possible

Type theory and logic

- *Curry-Howard-Correspondence:* (Curry (1958), Howard (1980))

  isomorphism: types ≈ propositions, terms ≈ proofs!

  ⇒ Connection between constructive logic and computer science
- E.g. used for tools like *Coq* – interactive theorem prover
- Helps the user formulating assertions and finding proofs
- proof-checking = type-checking the program!
- Such tools often based on calculi with *dependent types*

  ⇒ types like $\texttt{Array}\ \texttt{n} \rightarrow \texttt{Array}\ (\texttt{n} + \texttt{1})$ possible

Type theory and logic

- *Curry-Howard-Correspondence:* (Curry (1958), Howard (1980))
  isomorphism: types $\approx$ propositions, terms $\approx$ proofs!
  $\Rightarrow$ Connection between constructive logic and computer science
- E.g. used for tools like *Coq* – interactive theorem prover
- Helps the user formulating assertions and finding proofs
- proof-checking = type-checking the program!
- Such tools often based on calculi with *dependent types*
  $\Rightarrow$ types like $\texttt{Array}\ \texttt{n} \to \texttt{Array}\ (\texttt{n} + \texttt{1})$ possible

Conclusion

Q: What do we get using type systems in the context of software verification?

A:

- our program will compile and execute **(catch syntax errors)**
- our functions will take and return the intended data types
  (catch violations of our mental model/the designed API)
- we can **control** which functions can do which **effects**,
  prevent specific values to be taken out of **context**
  (e.g. the Monad typeclass in Haskell)
- With dependent types: prove **almost arbitrary properties**,
  e.g. check for array-out-of-bound errors on compile time or
  even prove that a sorting algorithm sorts correctly

Conclusion

Q: What do we get using type systems in the context of software verification?

A:

- our program will compile and execute **(catch syntax errors)**
- our functions will take and return the intended data types
  (catch violations of our mental model/the designed API)
- we can **control** which functions can do which **effects**,
  prevent specific values to be taken out of **context**
  (e.g. the Monad typeclass in Haskell)
- With dependent types: prove **almost arbitrary properties**,
  e.g. check for array-out-of-bound errors on compile time or
  even prove that a sorting algorithm sorts correctly

Conclusion

Q: What do we get using type systems in the context of software verification?

A:

- our program will compile and execute **(catch syntax errors)**
- our functions will take and return the intended data types
  (catch violations of our mental model/the designed API)
- we can **control** which functions can do which **effects**,
  prevent specific values to be taken out of **context**
  (e.g. the Monad typeclass in Haskell)
- With dependent types: prove **almost arbitrary properties**,
  e.g. check for array-out-of-bound errors on compile time or
  even prove that a sorting algorithm sorts correctly

Conclusion

Q: What do we get using type systems in the context of software verification?

A:

- our program will compile and execute **(catch syntax errors)**
- our functions will take and return the intended data types
  (catch violations of our mental model/the designed API)
- we can **control** which functions can do which **effects**,
  prevent specific values to be taken out of **context**
  (e.g. the Monad typeclass in Haskell)
- With dependent types: prove **almost arbitrary properties**,
  e.g. check for array-out-of-bound errors on compile time or
  even prove that a sorting algorithm sorts correctly

Conclusion

Q: What do we get using type systems in the context of software verification?

A:

- our program will compile and execute **(catch syntax errors)**
- our functions will take and return the intended data types
  (catch violations of our mental model/the designed API)
- we can **control** which functions can do which **effects**,
  prevent specific values to be taken out of **context**
  (e.g. the Monad typeclass in Haskell)
- With dependent types: prove **almost arbitrary properties**,
  e.g. check for array-out-of-bound errors on compile time or
  even prove that a sorting algorithm sorts correctly

Conclusion

- more powerful type systems require more work by the developer
- a clear model of the types in an application is necessery
- types are also a form of formal specification –
  as usual, it is balance between more safety and more additional work
- λ-calculi are an important model to prove properties and develop new algorithms and abstractions
- results can be and are transferred to real-world programming languages
  $\Rightarrow$ developers profit from better, safer and more expressive tools :)

Conclusion

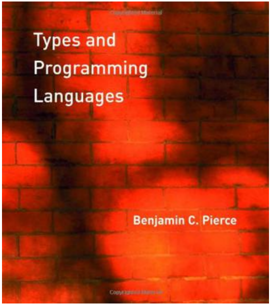
- more powerful type systems require more work by the developer
- a clear model of the types in an application is necessery
- types are also a form of formal specification –

  as usual, it is balance between more safety and more additional work
- λ-calculi are an important model to prove properties and develop new algorithms and abstractions
- results can be and are transferred to real-world programming languages

  $\Rightarrow$ developers profit from better, safer and more expressive tools :)

Conclusion

- more powerful type systems require more work by the developer
- a clear model of the types in an application is necessery
- types are also a form of formal specification –
  as usual, it is balance between more safety and more additional work
- λ-calculi are an important model to prove properties and develop new algorithms and abstractions
- results can be and are transferred to real-world programming languages
  $\Rightarrow$ developers profit from better, safer and more expressive tools :)



Benjamin C. Pierce. *Types and Programming Languages*, MIT Press, 2002

A. Church, *An unsolvable problem of elementary number theory*, American Journal of Mathematics, Volume 58, No. 2. (April 1936), pp. 345-363.

A. Church, *A Formulation of the Simple Theory of Types*, Journal of Symbolic Logic, Volume 5 (1940).

A. M. Turing, *Computability and λ-Definability*, The Journal of Symbolic Logic, Vol. 2, No. 4. (Dec., 1937), pp. 153-163.

H. Curry, R. Feys *Combinatory Logic Vol. I, Amsterdam, North-Holland* (1958)

W. A. Howard *The formulae-as-types notion of construction* in Essays on Combinatory Logic, Lambda Calculus and Formalism, Boston, MA (Sep. 1980), pp. 479-490

- John C. Reynolds *Definitional Interpreters for Higher-Order Programming Languages*, Higher-Order and Symbolic Computation 11 (1998)
- J. Girard *Interprétation fonctionnelle et élimination des coupures de l'arithmétique d'ordre supérieur*, Université Paris (1974)
- John C. Reynolds *Using category theory to design implicit conversions and generic operators*, Lecture Notes in Computer Science vol. 94, Springer-Verlag (1980)
- L. Cardelli *A semantics of multiple inheritance*, Lecture Notes in Computer Science vol. 173, pp. 51-67, Springer-Verlag (1984)
- S. Drossopoulou, S. Eisenbach, S. Khurshid *Is the Java Type System Sound?*, Theory and Practice of Object Systems (1999)