Computational Logic

Functional Programming

Christian Sternagel Harald Zankl Evgeny Zuenko

Department of Computer Science
University of Innsbruck

WS 2017/2018

Lecture 5

Topics

abstract data types, algebraic data types, binary search trees, combinator parsing, efficiency, encoding data types as lambda-terms, evaluation strategies, formal verification, first steps, guarded recursion, Haskell introduction, higher-order functions, historical overview, implementing a type checker, induction, infinite data structures, input and output, lambda-calculus, lazy evaluation, list comprehensions, lists, modules, pattern matching, polymorphism, property-based testing, reasoning about functional programs, recursive functions, sets, strings, tail recursion, trees, tupling, type checking, type inference, types, types and type classes, unification, user-defined types

Topics

abstract data types, algebraic data types, binary search trees, combinator parsing, efficiency, encoding data types as lambda-terms, evaluation strategies, formal verification, first steps, guarded recursion, Haskell introduction, higher-order functions, historical overview, implementing a type checker, induction, infinite data structures, input and output, lambda-calculus, lazy evaluation, list comprehensions, lists, modules, pattern matching, polymorphism, property-based testing, reasoning about functional programs, recursive functions, sets, strings, tail recursion, trees, tupling, type checking, type inference, types, types and type classes, unification, user-defined types

Overview

- Introduction to the λ-Calculus
- Encoding Data Types

Introduction to the λ-Calculus

Origin

- search for general framework in which every algorithm can be defined

Origin

- search for general framework in which every algorithm can be defined
- “universal language” (concerning computation)

Origin

- search for general framework in which every algorithm can be defined
- "universal language" (concerning computation)
- in 1936, Alonzo Church introduced the λ-Calculus in his paper *An Unsolvable Problem of Elementary Number Theory*, AJM **58**(2), pages 345–363, doi:10.2307/2371045

Origin

- search for general framework in which every algorithm can be defined
- "universal language" (concerning computation)
- in 1936, Alonzo Church introduced the λ-Calculus in his paper *An Unsolvable Problem of Elementary Number Theory*, AJM **58**(2), pages 345–363, doi:10.2307/2371045
- in 1937, Alan Turing introduced Turing Machines in his paper *On Computable Numbers with an Application to the Entscheidungsproblem*, LMS, **42**(2), pages 230–265, doi:10.1112/plms/s2-42.1.230

Origin

- search for general framework in which every algorithm can be defined
- "universal language" (concerning computation)
- in 1936, Alonzo Church introduced the λ-Calculus in his paper *An Unsolvable Problem of Elementary Number Theory*, AJM **58**(2), pages 345–363, doi:10.2307/2371045
- in 1937, Alan Turing introduced Turing Machines in his paper *On Computable Numbers with an Application to the Entscheidungsproblem*, LMS, **42**(2), pages 230–265, doi:10.1112/plms/s2-42.1.230
- later it was shown that both models of computation are equivalent

Origin

- search for general framework in which every algorithm can be defined
- "universal language" (concerning computation)
- in 1936, Alonzo Church introduced the λ-Calculus in his paper *An Unsolvable Problem of Elementary Number Theory*, AJM **58**(2), pages 345–363, doi:10.2307/2371045
- in 1937, Alan Turing introduced Turing Machines in his paper *On Computable Numbers with an Application to the Entscheidungsproblem*, LMS, **42**(2), pages 230–265, doi:10.1112/plms/s2-42.1.230
- later it was shown that both models of computation are equivalent
- that is, Turing-complete is the same as definable in the λ-Calculus

Origin

- search for general framework in which every algorithm can be defined
- "universal language" (concerning computation)
- in 1936, Alonzo Church introduced the λ-Calculus in his paper *An Unsolvable Problem of Elementary Number Theory*, AJM **58**(2), pages 345–363, doi:10.2307/2371045
- in 1937, Alan Turing introduced Turing Machines in his paper *On Computable Numbers with an Application to the Entscheidungsproblem*, LMS, **42**(2), pages 230–265, doi:10.1112/plms/s2-42.1.230
- later it was shown that both models of computation are equivalent
- that is, Turing-complete is the same as definable in the λ-Calculus
- λ-Calculus is underlying much of functional programming

Syntax – λ-Terms

- grammar

$$\begin{array}{rcll} t & ::= & x & \text{variable/atom} \\ & | & (t\ t) & \text{application} \\ & | & (\lambda x.\, t) & \text{(lambda) abstraction} \end{array}$$

- set of lambda terms is denoted Λ

Syntax – λ-Terms

- grammar

$$\begin{array}{rcll} t & ::= & x & \text{variable/atom} \\ & | & (t\ t) & \text{application} \\ & | & (\lambda x.\, t) & \text{(lambda) abstraction} \end{array}$$

- set of lambda terms is denoted Λ

Examples

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

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

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

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

Conventions (to ease reading and writing)

- outermost parentheses are dropped
- application binds stronger than abstraction, e.g., $\lambda x.\, x\ z$ is equal to $\lambda x.\, (x\ z)$ and not to $(\lambda x.\, x)\ z$
- applications associate to the left
- nested abstractions associate to the right and variables are combined

Conventions (to ease reading and writing)

- outermost parentheses are dropped
- application binds stronger than abstraction, e.g., $\lambda x.\, x\ z$ is equal to $\lambda x.\, (x\ z)$ and not to $(\lambda x.\, x)\ z$
- applications associate to the left
- nested abstractions associate to the right and variables are combined

Examples (using Conventions)

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

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

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

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

Conventions (to ease reading and writing)

- outermost parentheses are dropped
- application binds stronger than abstraction, e.g., $\lambda x.\, x\ z$ is equal to $\lambda x.\, (x\ z)$ and not to $(\lambda x.\, x)\ z$
- applications associate to the left
- nested abstractions associate to the right and variables are combined

Examples (using Conventions)

$$\lambda x.\, y$$

$$\lambda xy.\, x$$

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

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

Conventions (to ease reading and writing)

- outermost parentheses are dropped
- application binds stronger than abstraction, e.g., $\lambda x.\, x\ z$ is equal to $\lambda x.\, (x\ z)$ and not to $(\lambda x.\, x)\ z$
- applications associate to the left
- nested abstractions associate to the right and variables are combined

Examples (using Conventions)

$$\lambda x.\, y$$

$$\lambda xy.\, x$$

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

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

Note

- nested lambdas are "functions with multiple arguments"
- e.g., $\lambda xyz.\, t$ is a function taking 3 arguments

λ-Terms and Haskell

Haskell

- `(\x -> x + 1)`
- `(\x -> x + 1) 2` $= 3$
- `if True then 1 else 0` $= 1$
- `(,) 2 4` $=$ `(2, 4)`
- `fst (2, 4)` $= 2$

λ-Calculus

- $\lambda x.\,\text{add}\ x\ 1$
- $(\lambda x.\,\text{add}\ x\ 1)\ 2$
- if True $1\ 0$
- Pair $2\ 4$
- fst $(\text{Pair}\ 2\ 4)$

λ-Terms and Haskell

Haskell	λ-Calculus
`(\x -> x + 1)`	$\lambda x.\,\text{add}\ x\ 1$
`(\x -> x + 1) 2 = 3`	$(\lambda x.\,\text{add}\ x\ 1)\ 2$
`if True then 1 else 0 = 1`	if True $1\ 0$
`(,) 2 4 = (2, 4)`	Pair $2\ 4$
`fst (2, 4) = 2`	fst $(\text{Pair}\ 2\ 4)$

Remark

- '0', '1', '2', '4', 'add', 'fst', 'if', 'Pair', and 'True' are abbreviations for more complex λ-terms
- supposed to "encode" the behavior of 0, 1, 2, 4, (+), ...

Computation

- manipulate terms to “compute” some “result”

Computation

- manipulate terms to “compute” some “result”
- what are the rules?

Computation

- manipulate terms to “compute” some “result”
- what are the rules?
- it turns out that a single rule is enough

Computation

- manipulate terms to "compute" some "result"
- what are the rules?
- it turns out that a single rule is enough

The β-Rule (informal definition)

- intuition: apply "function" to "argument"
- in λ-Calculus both "functions" and "arguments" are λ-terms
- the β-rule

$$(\lambda x.\, s)\; t \quad \to_\beta \quad s[x := t]$$

- in words: *when applying function $(\lambda x.\, s)$ to input t, replace every occurrence of x in body of function (that is, s) by t*

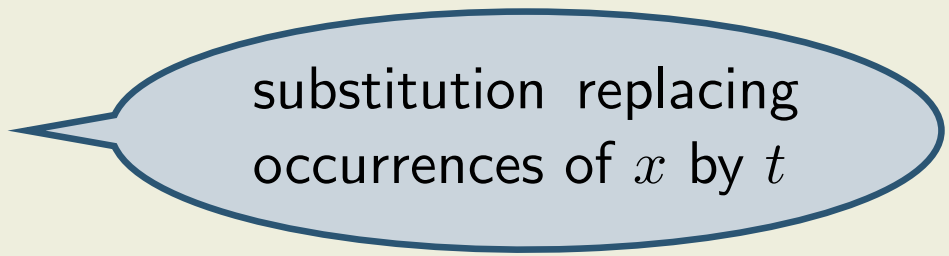
Computation

- manipulate terms to "compute" some "result"
- what are the rules?
- it turns out that a single rule is enough

The β-Rule (informal definition)

- intuition: apply "function" to "argument"
- in λ-Calculus both "functions" and "arguments" are λ-terms
- the β-rule

$$(\lambda x.\, s)\ t \quad \to_\beta \quad s[x := t]$$



- in words: *when applying function $(\lambda x.\, s)$ to input t, replace every occurrence of x in body of function (that is, s) by t*

The β-Rule

$$(\lambda x.\, s)\ t \quad \to_\beta \quad s[x := t]$$

Examples

$$(\lambda x.\, x)\ (\lambda x.\, x)$$

$$(\lambda xy.\, y)\ (\lambda x.\, x)$$

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

$$(\lambda x.\, x\ x)\ (\lambda x.\, x\ x)$$

$$\lambda x.\, x$$

The β-Rule

$$(\lambda x.\, s)\ t \quad \to_\beta \quad s[x := t]$$

Examples

$$\begin{aligned}
(\lambda x.\, x)\ (\lambda x.\, x) &\to_\beta \lambda x.\, x \\
(\lambda xy.\, y)\ (\lambda x.\, x) & \\
(\lambda xyz.\, x\ z\ (y\ z))\ (\lambda x.\, x) & \\
(\lambda x.\, x\ x)\ (\lambda x.\, x\ x) & \\
\lambda x.\, x &
\end{aligned}$$

The β-Rule

$$(\lambda x.\, s)\ t \quad \to_\beta \quad s[x := t]$$

Examples

$$\begin{aligned}
(\lambda x.\, x)\ (\lambda x.\, x) &\to_\beta \lambda x.\, x \\
(\lambda xy.\, y)\ (\lambda x.\, x) &\to_\beta \lambda y.\, y \\
(\lambda xyz.\, x\ z\ (y\ z))\ (\lambda x.\, x) & \\
(\lambda x.\, x\ x)\ (\lambda x.\, x\ x) & \\
\lambda x.\, x &
\end{aligned}$$

The β-Rule

$$(\lambda x.\, s)\ t \quad \to_\beta \quad s[x := t]$$

Examples

$$\begin{aligned}
(\lambda x.\, x)\ (\lambda x.\, x) &\to_\beta \lambda x.\, x \\
(\lambda xy.\, y)\ (\lambda x.\, x) &\to_\beta \lambda y.\, y \\
(\lambda xyz.\, x\ z\ (y\ z))\ (\lambda x.\, x) &\to_\beta \lambda yz.\, (\lambda x.\, x)\ z\ (y\ z) \\
(\lambda x.\, x\ x)\ (\lambda x.\, x\ x) & \\
\lambda x.\, x &
\end{aligned}$$

The β-Rule

$$(\lambda x.\, s)\ t \quad \to_\beta \quad s[x := t]$$

Examples

$$(\lambda x.\, x)\ (\lambda x.\, x) \ \to_\beta \ \lambda x.\, x$$

$$(\lambda xy.\, y)\ (\lambda x.\, x) \ \to_\beta \ \lambda y.\, y$$

$$(\lambda xyz.\, x\ z\ (y\ z))\ (\lambda x.\, x) \ \to_\beta \ \lambda yz.\, (\lambda x.\, x)\ z\ (y\ z) \ \to_\beta \ \lambda yz.\, z\ (y\ z)$$

$$(\lambda x.\, x\ x)\ (\lambda x.\, x\ x)$$

$$\lambda x.\, x$$

The β-Rule

$$(\lambda x.\, s)\ t \quad \to_\beta \quad s[x := t]$$

Examples

$$\begin{aligned}
(\lambda x.\, x)\ (\lambda x.\, x) &\to_\beta \lambda x.\, x \\
(\lambda xy.\, y)\ (\lambda x.\, x) &\to_\beta \lambda y.\, y \\
(\lambda xyz.\, x\ z\ (y\ z))\ (\lambda x.\, x) &\to_\beta \lambda yz.\, (\lambda x.\, x)\ z\ (y\ z) \to_\beta \lambda yz.\, z\ (y\ z) \\
(\lambda x.\, x\ x)\ (\lambda x.\, x\ x) &\to_\beta (\lambda x.\, x\ x)\ (\lambda x.\, x\ x) \\
&\quad \lambda x.\, x
\end{aligned}$$

The β-Rule

$$(\lambda x.\, s)\ t \quad \to_\beta \quad s[x := t]$$

Examples

$$\begin{aligned}
(\lambda x.\, x)\ (\lambda x.\, x) &\to_\beta \lambda x.\, x \\
(\lambda xy.\, y)\ (\lambda x.\, x) &\to_\beta \lambda y.\, y \\
(\lambda xyz.\, x\ z\ (y\ z))\ (\lambda x.\, x) &\to_\beta \lambda yz.\, (\lambda x.\, x)\ z\ (y\ z) \to_\beta \lambda yz.\, z\ (y\ z) \\
(\lambda x.\, x\ x)\ (\lambda x.\, x\ x) &\to_\beta (\lambda x.\, x\ x)\ (\lambda x.\, x\ x) \to_\beta \cdots \\
&\quad \lambda x.\, x
\end{aligned}$$

The β-Rule

$$(\lambda x.\, s)\ t \quad \to_\beta \quad s[x := t]$$

Examples

$$(\lambda x.\, x)\ (\lambda x.\, x) \to_\beta \lambda x.\, x$$

$$(\lambda xy.\, y)\ (\lambda x.\, x) \to_\beta \lambda y.\, y$$

$$(\lambda xyz.\, x\ z\ (y\ z))\ (\lambda x.\, x) \to_\beta \lambda yz.\, (\lambda x.\, x)\ z\ (y\ z) \to_\beta \lambda yz.\, z\ (y\ z)$$

$$(\lambda x.\, x\ x)\ (\lambda x.\, x\ x) \to_\beta (\lambda x.\, x\ x)\ (\lambda x.\, x\ x) \to_\beta \cdots$$

$$\lambda x.\, x \qquad \text{no } \beta\text{-step possible}$$

Problem – Variable Capture

- consider $\lambda xy.\, x$

Problem – Variable Capture

- consider $\lambda xy.\, x$
- intended behavior: *take 2 arguments, ignore second, return first*

Problem – Variable Capture

- consider $\lambda xy.\, x$
- intended behavior: *take 2 arguments, ignore second, return first*
- consider $t = (\lambda xy.\, x)\ y\ z$

Problem – Variable Capture

- consider $\lambda xy.\, x$
- intended behavior: *take 2 arguments, ignore second, return first*
- consider $t = (\lambda xy.\, x)\ y\ z$
- we want y as result, but get $t \to_\beta (\lambda y.\, y)\ z \to_\beta z$

Problem – Variable Capture

- consider $\lambda xy.\, x$
- intended behavior: *take 2 arguments, ignore second, return first*
- consider $t = (\lambda xy.\, x)\ y\ z$
- we want y as result, but get $t \to_\beta (\lambda y.\, y)\ z \to_\beta z$
- clearly not intended (the problem was that free y was bound when substituting for x)

Problem – Variable Capture

- consider $\lambda xy.\, x$
- intended behavior: *take 2 arguments, ignore second, return first*
- consider $t = (\lambda xy.\, x)\ y\ z$
- we want y as result, but get $t \to_\beta (\lambda y.\, y)\ z \to_\beta z$
- clearly not intended (the problem was that free y was bound when substituting for x)

Solution

- do not allow arbitrary substitution (freshness constraints)

Problem – Variable Capture

- consider $\lambda xy.\, x$
- intended behavior: *take 2 arguments, ignore second, return first*
- consider $t = (\lambda xy.\, x)\ y\ z$
- we want y as result, but get $t \to_\beta (\lambda y.\, y)\ z \to_\beta z$
- clearly not intended (the problem was that free y was bound when substituting for x)

Solution

- do not allow arbitrary substitution (freshness constraints)
- work modulo consistent renaming of bound variables (α-equivalence)

Renaming Variables with Permutations

- permutation is bijection between variables (can be represented by finite list of swappings $(x_1 \rightleftharpoons y_1, \ldots, x_n \rightleftharpoons y_n)$)

Renaming Variables with Permutations

- permutation is bijection between variables (can be represented by finite list of swappings $(x_1 \rightleftharpoons y_1, \ldots, x_n \rightleftharpoons y_n)$)
- apply permutation π to term t, written $\pi \bullet t$

$$\pi \bullet x = \pi(x)$$
$$\pi \bullet (t\ u) = (\pi \bullet t)\ (\pi \bullet u)$$
$$\pi \bullet (\lambda x.\, t) = \lambda \pi(x).\, (\pi \bullet t)$$

Renaming Variables with Permutations

- permutation is bijection between variables (can be represented by finite list of swappings $(x_1 \rightleftharpoons y_1, \ldots, x_n \rightleftharpoons y_n)$)
- apply permutation π to term t, written $\pi \bullet t$

$$\pi \bullet x = \pi(x)$$
$$\pi \bullet (t\ u) = (\pi \bullet t)\ (\pi \bullet u)$$
$$\pi \bullet (\lambda x.\, t) = \lambda \pi(x).\, (\pi \bullet t)$$

Example

- $(x \rightleftharpoons y) \bullet (\lambda xy.\, x\ (\lambda zx.\, z\ x))$

Renaming Variables with Permutations

- permutation is bijection between variables (can be represented by finite list of swappings $(x_1 \rightleftharpoons y_1, \ldots, x_n \rightleftharpoons y_n)$)
- apply permutation π to term t, written $\pi \bullet t$

$$\pi \bullet x = \pi(x)$$
$$\pi \bullet (t\ u) = (\pi \bullet t)\ (\pi \bullet u)$$
$$\pi \bullet (\lambda x.\, t) = \lambda \pi(x).\, (\pi \bullet t)$$

Example

- $(x \rightleftharpoons y) \bullet (\lambda xy.\, x\ (\lambda zx.\, z\ x)) = \lambda yx.\, y\ (\lambda zy.\, z\ y)$

Renaming Variables with Permutations

- permutation is bijection between variables (can be represented by finite list of swappings $(x_1 \rightleftharpoons y_1, \ldots, x_n \rightleftharpoons y_n)$)
- apply permutation π to term t, written $\pi \bullet t$

$$\pi \bullet x = \pi(x)$$
$$\pi \bullet (t\ u) = (\pi \bullet t)\ (\pi \bullet u)$$
$$\pi \bullet (\lambda x.\, t) = \lambda \pi(x).\, (\pi \bullet t)$$

Example

- $(x \rightleftharpoons y) \bullet (\lambda xy.\, x\ (\lambda zx.\, z\ x)) = \lambda yx.\, y\ (\lambda zy.\, z\ y)$
- $(x \rightleftharpoons z, y \rightleftharpoons x, z \rightleftharpoons y) \bullet (z\ y)$

Renaming Variables with Permutations

- permutation is bijection between variables (can be represented by finite list of swappings $(x_1 \rightleftharpoons y_1, \ldots, x_n \rightleftharpoons y_n)$)
- apply permutation π to term t, written $\pi \bullet t$

$$\pi \bullet x = \pi(x)$$
$$\pi \bullet (t\ u) = (\pi \bullet t)\ (\pi \bullet u)$$
$$\pi \bullet (\lambda x.\, t) = \lambda \pi(x).\, (\pi \bullet t)$$

Example

- $(x \rightleftharpoons y) \bullet (\lambda xy.\, x\ (\lambda zx.\, z\ x)) = \lambda yx.\, y\ (\lambda zy.\, z\ y)$
- $(x \rightleftharpoons z, y \rightleftharpoons x, z \rightleftharpoons y) \bullet (z\ y) = y\ x$

Free Variables and Freshness

- set of free variables of a term

$$\mathcal{F}(x) = \{x\}$$
$$\mathcal{F}(t\ u) = \mathcal{F}(t) \cup \mathcal{F}(u)$$
$$\mathcal{F}(\lambda x.\, t) = \mathcal{F}(t) \setminus \{x\}$$

Free Variables and Freshness

- set of free variables of a term

$$\mathcal{F}(x) = \{x\}$$
$$\mathcal{F}(t\ u) = \mathcal{F}(t) \cup \mathcal{F}(u)$$
$$\mathcal{F}(\lambda x.\, t) = \mathcal{F}(t) \setminus \{x\}$$

- freshness constraint $X \sharp o$ – set of variables X is fresh for given syntactic object o, e.g., if o is term then $X \sharp o$ iff $X \cap \mathcal{F}(o) = \varnothing$

Free Variables and Freshness

- set of free variables of a term

$$\mathcal{F}(x) = \{x\}$$
$$\mathcal{F}(t\ u) = \mathcal{F}(t) \cup \mathcal{F}(u)$$
$$\mathcal{F}(\lambda x.\, t) = \mathcal{F}(t) \setminus \{x\}$$

- freshness constraint $X \sharp o$ – set of variables X is fresh for given syntactic object o, e.g., if o is term then $X \sharp o$ iff $X \cap \mathcal{F}(o) = \varnothing$
- for single variable write $x \sharp o$ instead of $\{x\} \sharp o$

Free Variables and Freshness

- set of free variables of a term

$$\mathcal{F}(x) = \{x\}$$
$$\mathcal{F}(t\ u) = \mathcal{F}(t) \cup \mathcal{F}(u)$$
$$\mathcal{F}(\lambda x.\, t) = \mathcal{F}(t) \setminus \{x\}$$

- freshness constraint $X \sharp o$ – set of variables X is fresh for given syntactic object o, e.g., if o is term then $X \sharp o$ iff $X \cap \mathcal{F}(o) = \varnothing$
- for single variable write $x \sharp o$ instead of $\{x\} \sharp o$

Examples

term t	$\mathcal{F}(t)$	$x \sharp t$	$\{y, z\} \sharp t$
$\lambda x.\, x$			
$x\ y$			
$(\lambda x.\, x)\ x$			
$\lambda x.\, x\ y\ z$			

Free Variables and Freshness

- set of free variables of a term

$$\mathcal{F}(x) = \{x\}$$
$$\mathcal{F}(t\ u) = \mathcal{F}(t) \cup \mathcal{F}(u)$$
$$\mathcal{F}(\lambda x.\, t) = \mathcal{F}(t) \setminus \{x\}$$

- freshness constraint $X \sharp o$ – set of variables X is fresh for given syntactic object o, e.g., if o is term then $X \sharp o$ iff $X \cap \mathcal{F}(o) = \varnothing$
- for single variable write $x \sharp o$ instead of $\{x\} \sharp o$

Examples

term t	$\mathcal{F}(t)$	$x \sharp t$	$\{y, z\} \sharp t$
$\lambda x.\, x$	$\varnothing$		
$x\ y$			
$(\lambda x.\, x)\ x$			
$\lambda x.\, x\ y\ z$			

Free Variables and Freshness

- set of free variables of a term

$$\mathcal{F}(x) = \{x\}$$
$$\mathcal{F}(t\ u) = \mathcal{F}(t) \cup \mathcal{F}(u)$$
$$\mathcal{F}(\lambda x.\, t) = \mathcal{F}(t) \setminus \{x\}$$

- freshness constraint $X \sharp o$ – set of variables X is fresh for given syntactic object o, e.g., if o is term then $X \sharp o$ iff $X \cap \mathcal{F}(o) = \varnothing$
- for single variable write $x \sharp o$ instead of $\{x\} \sharp o$

Examples

term t	$\mathcal{F}(t)$	$x \sharp t$	$\{y, z\} \sharp t$
$\lambda x.\, x$	$\varnothing$	✔	
$x\ y$			
$(\lambda x.\, x)\ x$			
$\lambda x.\, x\ y\ z$			

Free Variables and Freshness

- set of free variables of a term

$$\mathcal{F}(x) = \{x\}$$
$$\mathcal{F}(t\ u) = \mathcal{F}(t) \cup \mathcal{F}(u)$$
$$\mathcal{F}(\lambda x.\, t) = \mathcal{F}(t) \setminus \{x\}$$

- freshness constraint $X \sharp o$ – set of variables X is fresh for given syntactic object o, e.g., if o is term then $X \sharp o$ iff $X \cap \mathcal{F}(o) = \varnothing$
- for single variable write $x \sharp o$ instead of $\{x\} \sharp o$

Examples

term t	$\mathcal{F}(t)$	$x \sharp t$	$\{y, z\} \sharp t$
$\lambda x.\, x$	$\varnothing$	✔	✔
$x\ y$			
$(\lambda x.\, x)\ x$			
$\lambda x.\, x\ y\ z$			

Free Variables and Freshness

- set of free variables of a term

$$\mathcal{F}(x) = \{x\}$$
$$\mathcal{F}(t\ u) = \mathcal{F}(t) \cup \mathcal{F}(u)$$
$$\mathcal{F}(\lambda x.\, t) = \mathcal{F}(t) \setminus \{x\}$$

- freshness constraint $X \sharp o$ – set of variables X is fresh for given syntactic object o, e.g., if o is term then $X \sharp o$ iff $X \cap \mathcal{F}(o) = \varnothing$
- for single variable write $x \sharp o$ instead of $\{x\} \sharp o$

Examples

term t	$\mathcal{F}(t)$	$x \sharp t$	$\{y, z\} \sharp t$
$\lambda x.\, x$	$\varnothing$	✔	✔
$x\ y$	$\{x, y\}$		
$(\lambda x.\, x)\ x$			
$\lambda x.\, x\ y\ z$			

Free Variables and Freshness

- set of free variables of a term

$$\mathcal{F}(x) = \{x\}$$
$$\mathcal{F}(t\ u) = \mathcal{F}(t) \cup \mathcal{F}(u)$$
$$\mathcal{F}(\lambda x.\, t) = \mathcal{F}(t) \setminus \{x\}$$

- freshness constraint $X \sharp o$ – set of variables X is fresh for given syntactic object o, e.g., if o is term then $X \sharp o$ iff $X \cap \mathcal{F}(o) = \varnothing$
- for single variable write $x \sharp o$ instead of $\{x\} \sharp o$

Examples

term t	$\mathcal{F}(t)$	$x \sharp t$	$\{y, z\} \sharp t$
$\lambda x.\, x$	$\varnothing$	✔	✔
$x\ y$	$\{x, y\}$	✘	
$(\lambda x.\, x)\ x$			
$\lambda x.\, x\ y\ z$			

Free Variables and Freshness

- set of free variables of a term

$$\mathcal{F}(x) = \{x\}$$
$$\mathcal{F}(t\ u) = \mathcal{F}(t) \cup \mathcal{F}(u)$$
$$\mathcal{F}(\lambda x.\, t) = \mathcal{F}(t) \setminus \{x\}$$

- freshness constraint $X \sharp o$ – set of variables X is fresh for given syntactic object o, e.g., if o is term then $X \sharp o$ iff $X \cap \mathcal{F}(o) = \varnothing$
- for single variable write $x \sharp o$ instead of $\{x\} \sharp o$

Examples

term t	$\mathcal{F}(t)$	$x \sharp t$	$\{y, z\} \sharp t$
$\lambda x.\, x$	$\varnothing$	✔	✔
$x\ y$	$\{x, y\}$	✘	✘
$(\lambda x.\, x)\ x$			
$\lambda x.\, x\ y\ z$			

Free Variables and Freshness

- set of free variables of a term

$$\mathcal{F}(x) = \{x\}$$
$$\mathcal{F}(t\ u) = \mathcal{F}(t) \cup \mathcal{F}(u)$$
$$\mathcal{F}(\lambda x.\, t) = \mathcal{F}(t) \setminus \{x\}$$

- freshness constraint $X \sharp o$ – set of variables X is fresh for given syntactic object o, e.g., if o is term then $X \sharp o$ iff $X \cap \mathcal{F}(o) = \varnothing$
- for single variable write $x \sharp o$ instead of $\{x\} \sharp o$

Examples

term t	$\mathcal{F}(t)$	$x \sharp t$	$\{y, z\} \sharp t$
$\lambda x.\, x$	$\varnothing$	✔	✔
$x\ y$	$\{x, y\}$	✘	✘
$(\lambda x.\, x)\ x$	$\{x\}$		
$\lambda x.\, x\ y\ z$			

Free Variables and Freshness

- set of free variables of a term

$$\mathcal{F}(x) = \{x\}$$
$$\mathcal{F}(t\ u) = \mathcal{F}(t) \cup \mathcal{F}(u)$$
$$\mathcal{F}(\lambda x.\, t) = \mathcal{F}(t) \setminus \{x\}$$

- freshness constraint $X \sharp o$ – set of variables X is fresh for given syntactic object o, e.g., if o is term then $X \sharp o$ iff $X \cap \mathcal{F}(o) = \varnothing$
- for single variable write $x \sharp o$ instead of $\{x\} \sharp o$

Examples

term t	$\mathcal{F}(t)$	$x \sharp t$	$\{y, z\} \sharp t$
$\lambda x.\, x$	$\varnothing$	✔	✔
$x\ y$	$\{x, y\}$	✘	✘
$(\lambda x.\, x)\ x$	$\{x\}$	✘	
$\lambda x.\, x\ y\ z$			

Free Variables and Freshness

- set of free variables of a term

$$\mathcal{F}(x) = \{x\}$$
$$\mathcal{F}(t\ u) = \mathcal{F}(t) \cup \mathcal{F}(u)$$
$$\mathcal{F}(\lambda x.\, t) = \mathcal{F}(t) \setminus \{x\}$$

- freshness constraint $X \sharp o$ – set of variables X is fresh for given syntactic object o, e.g., if o is term then $X \sharp o$ iff $X \cap \mathcal{F}(o) = \varnothing$
- for single variable write $x \sharp o$ instead of $\{x\} \sharp o$

Examples

term t	$\mathcal{F}(t)$	$x \sharp t$	$\{y, z\} \sharp t$
$\lambda x.\, x$	$\varnothing$	✔	✔
$x\ y$	$\{x, y\}$	✘	✘
$(\lambda x.\, x)\ x$	$\{x\}$	✘	✔
$\lambda x.\, x\ y\ z$			

Free Variables and Freshness

- set of free variables of a term

$$\mathcal{F}(x) = \{x\}$$
$$\mathcal{F}(t\ u) = \mathcal{F}(t) \cup \mathcal{F}(u)$$
$$\mathcal{F}(\lambda x.\, t) = \mathcal{F}(t) \setminus \{x\}$$

- freshness constraint $X \sharp o$ – set of variables X is fresh for given syntactic object o, e.g., if o is term then $X \sharp o$ iff $X \cap \mathcal{F}(o) = \varnothing$
- for single variable write $x \sharp o$ instead of $\{x\} \sharp o$

Examples

term t	$\mathcal{F}(t)$	$x \sharp t$	$\{y, z\} \sharp t$
$\lambda x.\, x$	$\varnothing$	✔	✔
$x\ y$	$\{x, y\}$	✘	✘
$(\lambda x.\, x)\ x$	$\{x\}$	✘	✔
$\lambda x.\, x\ y\ z$	$\{y, z\}$		

Free Variables and Freshness

- set of free variables of a term

$$\mathcal{F}(x) = \{x\}$$
$$\mathcal{F}(t\ u) = \mathcal{F}(t) \cup \mathcal{F}(u)$$
$$\mathcal{F}(\lambda x.\, t) = \mathcal{F}(t) \setminus \{x\}$$

- freshness constraint $X \sharp o$ – set of variables X is fresh for given syntactic object o, e.g., if o is term then $X \sharp o$ iff $X \cap \mathcal{F}(o) = \varnothing$
- for single variable write $x \sharp o$ instead of $\{x\} \sharp o$

Examples

term t	$\mathcal{F}(t)$	$x \sharp t$	$\{y, z\} \sharp t$
$\lambda x.\, x$	$\varnothing$	✔	✔
$x\ y$	$\{x, y\}$	✘	✘
$(\lambda x.\, x)\ x$	$\{x\}$	✘	✔
$\lambda x.\, x\ y\ z$	$\{y, z\}$	✔	

Free Variables and Freshness

- set of free variables of a term

$$\mathcal{F}(x) = \{x\}$$
$$\mathcal{F}(t\ u) = \mathcal{F}(t) \cup \mathcal{F}(u)$$
$$\mathcal{F}(\lambda x.\, t) = \mathcal{F}(t) \setminus \{x\}$$

- freshness constraint $X \sharp o$ – set of variables X is fresh for given syntactic object o, e.g., if o is term then $X \sharp o$ iff $X \cap \mathcal{F}(o) = \varnothing$
- for single variable write $x \sharp o$ instead of $\{x\} \sharp o$

Examples

term t	$\mathcal{F}(t)$	$x \sharp t$	$\{y, z\} \sharp t$
$\lambda x.\, x$	$\varnothing$	✔	✔
$x\ y$	$\{x, y\}$	✘	✘
$(\lambda x.\, x)\ x$	$\{x\}$	✘	✔
$\lambda x.\, x\ y\ z$	$\{y, z\}$	✔	✘

α-Equivalence

- t is α-equivalent to u, written $t \equiv_\alpha u$, if u can be obtained by consistently renaming bound variables

α-Equivalence

- t is α-equivalent to u, written $t \equiv_\alpha u$, if u can be obtained by consistently renaming bound variables
- $\equiv_\alpha$ is an equivalence relation (reflexive, symmetric, transitive)

α-Equivalence

- t is α-equivalent to u, written $t \equiv_\alpha u$, if u can be obtained by consistently renaming bound variables
- $\equiv_\alpha$ is an equivalence relation (reflexive, symmetric, transitive)
- from now on α-equivalent terms are considered equal

α-Equivalence

- t is α-equivalent to u, written $t \equiv_\alpha u$, if u can be obtained by consistently renaming bound variables
- $\equiv_\alpha$ is an equivalence relation (reflexive, symmetric, transitive)
- from now on α-equivalent terms are considered equal
- that is, we work on α-equivalence classes

α-Equivalence

- t is α-equivalent to u, written $t \equiv_\alpha u$, if u can be obtained by consistently renaming bound variables
- $\equiv_\alpha$ is an equivalence relation (reflexive, symmetric, transitive)
- from now on α-equivalent terms are considered equal
- that is, we work on α-equivalence classes

Examples

- $\lambda xy.\, x \equiv_\alpha \lambda yx.\, y$

α-Equivalence

- t is α-equivalent to u, written $t \equiv_\alpha u$, if u can be obtained by consistently renaming bound variables
- $\equiv_\alpha$ is an equivalence relation (reflexive, symmetric, transitive)
- from now on α-equivalent terms are considered equal
- that is, we work on α-equivalence classes

Examples

- $\lambda xy.\, x \equiv_\alpha \lambda yx.\, y$ since $\lambda xy.\, x \equiv_\alpha (x \rightleftharpoons z) \bullet (\lambda xy.\, x)$

α-Equivalence

- t is α-equivalent to u, written $t \equiv_\alpha u$, if u can be obtained by consistently renaming bound variables
- $\equiv_\alpha$ is an equivalence relation (reflexive, symmetric, transitive)
- from now on α-equivalent terms are considered equal
- that is, we work on α-equivalence classes

Examples

- $\lambda xy.\, x \equiv_\alpha \lambda yx.\, y$ since $\lambda xy.\, x \equiv_\alpha (x \rightleftharpoons z) \bullet (\lambda xy.\, x) = \lambda zy.\, z$

α-Equivalence

- t is α-equivalent to u, written $t \equiv_\alpha u$, if u can be obtained by consistently renaming bound variables
- $\equiv_\alpha$ is an equivalence relation (reflexive, symmetric, transitive)
- from now on α-equivalent terms are considered equal
- that is, we work on α-equivalence classes

Examples

- $\lambda xy.\, x \equiv_\alpha \lambda yx.\, y$ since $\lambda xy.\, x \equiv_\alpha (x \rightleftharpoons z) \bullet (\lambda xy.\, x) = \lambda zy.\, z \equiv_\alpha (y \rightleftharpoons x) \bullet (\lambda zy.\, z)$

α-Equivalence

- t is α-equivalent to u, written $t \equiv_\alpha u$, if u can be obtained by consistently renaming bound variables
- $\equiv_\alpha$ is an equivalence relation (reflexive, symmetric, transitive)
- from now on α-equivalent terms are considered equal
- that is, we work on α-equivalence classes

Examples

- $\lambda xy.\, x \equiv_\alpha \lambda yx.\, y$ since $\lambda xy.\, x \equiv_\alpha (x \rightleftharpoons z) \bullet (\lambda xy.\, x) = \lambda zy.\, z \equiv_\alpha (y \rightleftharpoons x) \bullet (\lambda zy.\, z) = \lambda zx.\, z$

α-Equivalence

- t is α-equivalent to u, written $t \equiv_\alpha u$, if u can be obtained by consistently renaming bound variables
- $\equiv_\alpha$ is an equivalence relation (reflexive, symmetric, transitive)
- from now on α-equivalent terms are considered equal
- that is, we work on α-equivalence classes

Examples

- $\lambda xy.\, x \equiv_\alpha \lambda yx.\, y$ since $\lambda xy.\, x \equiv_\alpha (x \rightleftharpoons z) \bullet (\lambda xy.\, x) = \lambda zy.\, z \equiv_\alpha (y \rightleftharpoons x) \bullet (\lambda zy.\, z) = \lambda zx.\, z \equiv_\alpha (z \rightleftharpoons y) \bullet (\lambda zx.\, z)$

α-Equivalence

- t is α-equivalent to u, written $t \equiv_\alpha u$, if u can be obtained by consistently renaming bound variables
- $\equiv_\alpha$ is an equivalence relation (reflexive, symmetric, transitive)
- from now on α-equivalent terms are considered equal
- that is, we work on α-equivalence classes

Examples

- $\lambda xy.\, x \equiv_\alpha \lambda yx.\, y$ since $\lambda xy.\, x \equiv_\alpha (x \rightleftharpoons z) \bullet (\lambda xy.\, x) = \lambda zy.\, z \equiv_\alpha (y \rightleftharpoons x) \bullet (\lambda zy.\, z) = \lambda zx.\, z \equiv_\alpha (z \rightleftharpoons y) \bullet (\lambda zx.\, z) = \lambda yx.\, y$

α-Equivalence

- t is α-equivalent to u, written $t \equiv_\alpha u$, if u can be obtained by consistently renaming bound variables
- $\equiv_\alpha$ is an equivalence relation (reflexive, symmetric, transitive)
- from now on α-equivalent terms are considered equal
- that is, we work on α-equivalence classes

Examples

- $\lambda xy.\, x \equiv_\alpha \lambda yx.\, y$ since $\lambda xy.\, x \equiv_\alpha (x \rightleftharpoons z) \bullet (\lambda xy.\, x) = \lambda zy.\, z \equiv_\alpha (y \rightleftharpoons x) \bullet (\lambda zy.\, z) = \lambda zx.\, z \equiv_\alpha (z \rightleftharpoons y) \bullet (\lambda zx.\, z) = \lambda yx.\, y$
- α-equivalence class of $\lambda x.\, y\ x$ (remember: all elements considered the same term)

$$\{\lambda x.\, y\ x, \quad \lambda z.\, y\ z, \quad \lambda a.\, y\ a, \quad \lambda x'.\, y\ x', \quad \lambda b_1.\, y\ b_1, \quad \ldots\}$$

Substitutions

- substitution (for terms) is function from variables to terms
- we only need substitutions replacing a single variable
- hence, we can always write $[x := t]$ for the substitution replacing x by t and leaving all other variables unchanged

Substitutions

- substitution (for terms) is function from variables to terms
- we only need substitutions replacing a single variable
- hence, we can always write $[x := t]$ for the substitution replacing x by t and leaving all other variables unchanged

Example

- consider $\sigma = [x := \lambda x.\, x]$
- then $\sigma(x) = \lambda x.\, x$ and
- $\sigma(y) = y$ for all $y \neq x$

Applying Substitutions to Terms

- applying substitution $\sigma = [x := s]$ to term t is denoted by $t\sigma$

Applying Substitutions to Terms

- applying substitution $\sigma = [x := s]$ to term t is denoted by $t\sigma$
- and defined by

$$
\begin{aligned}
x\sigma &= s && \\
y\sigma &= y && \text{if } x \neq y \\
(t\ u)\sigma &= (t\sigma)\ (u\sigma) && \\
(\lambda y.\, t)\sigma &= \lambda y.\, (t\sigma) && \text{if } y \sharp (x, s)
\end{aligned}
$$

Applying Substitutions to Terms

- applying substitution $\sigma = [x := s]$ to term t is denoted by $t\sigma$
- and defined by

$$
\begin{aligned}
x\sigma &= s && \\
y\sigma &= y && \text{if } x \neq y \\
(t\ u)\sigma &= (t\sigma)\ (u\sigma) && \\
(\lambda y.\, t)\sigma &= \lambda y.\, (t\sigma) && \text{if } y \sharp (x, s)
\end{aligned}
$$

- that is, bound variables are not substituted

Applying Substitutions to Terms

- applying substitution $\sigma = [x := s]$ to term t is denoted by $t\sigma$
- and defined by

$$
\begin{aligned}
x\sigma &= s && \\
y\sigma &= y && \text{if } x \neq y \\
(t\ u)\sigma &= (t\sigma)\ (u\sigma) && \\
(\lambda y.\, t)\sigma &= \lambda y.\, (t\sigma) && \text{if } y \sharp (x, s)
\end{aligned}
$$

- that is, bound variables are not substituted
- **note:** freshness constraint can always be satisfied by renaming

Applying Substitutions to Terms

- applying substitution $\sigma = [x := s]$ to term t is denoted by $t\sigma$
- and defined by

$$
\begin{aligned}
x\sigma &= s && \\
y\sigma &= y && \text{if } x \neq y \\
(t\ u)\sigma &= (t\sigma)\ (u\sigma) && \\
(\lambda y.\, t)\sigma &= \lambda y.\, (t\sigma) && \text{if } y \sharp (x, s)
\end{aligned}
$$

- that is, bound variables are not substituted
- **note:** freshness constraint can always be satisfied by renaming

Examples

- $\sigma = [x := \lambda x.\, x]$
- $x\sigma = \lambda x.\, x$
- $y\sigma = y$
- $(\lambda x.\, x)\sigma$

Applying Substitutions to Terms

- applying substitution $\sigma = [x := s]$ to term t is denoted by $t\sigma$
- and defined by

$$
\begin{aligned}
x\sigma &= s && \\
y\sigma &= y && \text{if } x \neq y \\
(t\ u)\sigma &= (t\sigma)\ (u\sigma) && \\
(\lambda y.\, t)\sigma &= \lambda y.\, (t\sigma) && \text{if } y \sharp (x, s)
\end{aligned}
$$

- that is, bound variables are not substituted
- **note:** freshness constraint can always be satisfied by renaming

Examples

- $\sigma = [x := \lambda x.\, x]$
- $x\sigma = \lambda x.\, x$
- $y\sigma = y$
- $(\lambda x.\, x)\sigma = (\lambda y.\, y)\sigma$

Applying Substitutions to Terms

- applying substitution $\sigma = [x := s]$ to term t is denoted by $t\sigma$
- and defined by

$$
\begin{aligned}
x\sigma &= s && \\
y\sigma &= y && \text{if } x \neq y \\
(t\ u)\sigma &= (t\sigma)\ (u\sigma) && \\
(\lambda y.\, t)\sigma &= \lambda y.\, (t\sigma) && \text{if } y \sharp (x, s)
\end{aligned}
$$

- that is, bound variables are not substituted
- **note:** freshness constraint can always be satisfied by renaming

Examples

- $\sigma = [x := \lambda x.\, x]$
- $x\sigma = \lambda x.\, x$
- $y\sigma = y$
- $(\lambda x.\, x)\sigma = (\lambda y.\, y)\sigma = \lambda y.\, (y\sigma)$

Applying Substitutions to Terms

- applying substitution $\sigma = [x := s]$ to term t is denoted by $t\sigma$
- and defined by

$$
\begin{aligned}
x\sigma &= s && \\
y\sigma &= y && \text{if } x \neq y \\
(t\ u)\sigma &= (t\sigma)\ (u\sigma) && \\
(\lambda y.\, t)\sigma &= \lambda y.\, (t\sigma) && \text{if } y \sharp (x, s)
\end{aligned}
$$

- that is, bound variables are not substituted
- **note:** freshness constraint can always be satisfied by renaming

Examples

- $\sigma = [x := \lambda x.\, x]$
- $x\sigma = \lambda x.\, x$
- $y\sigma = y$
- $(\lambda x.\, x)\sigma = (\lambda y.\, y)\sigma = \lambda y.\, (y\sigma) = \lambda y.\, y$

The β-Rule (formal definition)

- inductive definition of one step β-reduction

$$\frac{}{(\lambda x.\, t)\ u \to_\beta t[x := u]}\ \text{(root)} \qquad \frac{s \to_\beta t}{s\ u \to_\beta t\ u}\ \text{(app-l)}$$

$$\frac{s \to_\beta t}{\lambda x.\, s \to_\beta \lambda x.\, t}\ \text{(abs)} \qquad \frac{s \to_\beta t}{u\ s \to_\beta u\ t}\ \text{(app-r)}$$

The β-Rule (formal definition)

- inductive definition of one step β-reduction

$$\frac{}{(\lambda x.\, t)\ u \to_\beta t[x := u]}\ \text{(root)} \qquad \frac{s \to_\beta t}{s\ u \to_\beta t\ u}\ \text{(app-l)}$$

$$\frac{s \to_\beta t}{\lambda x.\, s \to_\beta \lambda x.\, t}\ \text{(abs)} \qquad \frac{s \to_\beta t}{u\ s \to_\beta u\ t}\ \text{(app-r)}$$

- in words: *if s has subterm of form $(\lambda x.\, t)\ u$, then replacing it by $t[x := u]$ is a β-step*

The β-Rule (formal definition)

- inductive definition of one step β-reduction

$$\frac{}{(\lambda x.\, t)\ u \to_\beta t[x := u]}\ \text{(root)} \qquad \frac{s \to_\beta t}{s\ u \to_\beta t\ u}\ \text{(app-l)}$$

$$\frac{s \to_\beta t}{\lambda x.\, s \to_\beta \lambda x.\, t}\ \text{(abs)} \qquad \frac{s \to_\beta t}{u\ s \to_\beta u\ t}\ \text{(app-r)}$$

- in words: *if s has subterm of form $(\lambda x.\, t)\ u$, then replacing it by $t[x := u]$ is a β-step*
- we call $(\lambda x.\, t)\ u$ a redex (short for *red*ucible *ex*pression), and

The β-Rule (formal definition)

- inductive definition of one step β-reduction

$$\frac{}{(\lambda x.\, t)\ u \to_\beta t[x := u]}\ \text{(root)} \qquad \frac{s \to_\beta t}{s\ u \to_\beta t\ u}\ \text{(app-l)}$$

$$\frac{s \to_\beta t}{\lambda x.\, s \to_\beta \lambda x.\, t}\ \text{(abs)} \qquad \frac{s \to_\beta t}{u\ s \to_\beta u\ t}\ \text{(app-r)}$$

- in words: *if s has subterm of form $(\lambda x.\, t)\ u$, then replacing it by $t[x := u]$ is a β-step*
- we call $(\lambda x.\, t)\ u$ a redex (short for *red*ucible *ex*pression), and
- $t[x := u]$ its contractum

The β-Rule (formal definition)

- inductive definition of one step β-reduction

$$\frac{}{(\lambda x.\, t)\ u \to_\beta t[x := u]}\ \text{(root)} \qquad \frac{s \to_\beta t}{s\ u \to_\beta t\ u}\ \text{(app-l)}$$

$$\frac{s \to_\beta t}{\lambda x.\, s \to_\beta \lambda x.\, t}\ \text{(abs)} \qquad \frac{s \to_\beta t}{u\ s \to_\beta u\ t}\ \text{(app-r)}$$

- in words: *if s has subterm of form $(\lambda x.\, t)\ u$, then replacing it by $t[x := u]$ is a β-step*
- we call $(\lambda x.\, t)\ u$ a redex (short for *red*ucible *ex*pression), and
- $t[x := u]$ its contractum
- $s \to_\beta^* t$ denotes a sequence $s = t_1 \to_\beta t_2 \to_\beta \cdots \to_\beta t_n = t$ with $n \geq 0$ (s (β-)reduces to t)

The β-Rule (formal definition)

- inductive definition of one step β-reduction

$$\frac{}{(\lambda x.\, t)\ u \to_\beta t[x := u]}\ \text{(root)} \qquad \frac{s \to_\beta t}{s\ u \to_\beta t\ u}\ \text{(app-l)}$$

$$\frac{s \to_\beta t}{\lambda x.\, s \to_\beta \lambda x.\, t}\ \text{(abs)} \qquad \frac{s \to_\beta t}{u\ s \to_\beta u\ t}\ \text{(app-r)}$$

- in words: *if s has subterm of form $(\lambda x.\, t)\ u$, then replacing it by $t[x := u]$ is a β-step*
- we call $(\lambda x.\, t)\ u$ a redex (short for *red*ucible *ex*pression), and
- $t[x := u]$ its contractum
- $s \to_\beta^* t$ denotes a sequence $s = t_1 \to_\beta t_2 \to_\beta \cdots \to_\beta t_n = t$ with $n \geq 0$ (s (β-)reduces to t)
- a nonempty sequence (that is, $n > 0$) is denoted by $s \to_\beta^+ t$

Exercise

- consider $\Omega = (\lambda x.\, x\ x)\ (\lambda x.\, x\ x)$,
- $K = \lambda xy.\, x$,
- $K_* = \lambda xy.\, y$, and
- $I = \lambda x.\, x$
- reduce the following λ-terms

$$K\ \Omega$$
$$K_*\ \Omega$$
$$I\ \Omega$$

What are the Results of Computations?

- we do only have λ-terms

What are the Results of Computations?

- we do only have λ-terms
- thus, we have to express "functions" and "results" as λ-terms

What are the Results of Computations?

- we do only have λ-terms
- thus, we have to express "functions" and "results" as λ-terms
- as long as β-steps are applicable, terms are not "stable"

What are the Results of Computations?

- we do only have λ-terms
- thus, we have to express "functions" and "results" as λ-terms
- as long as β-steps are applicable, terms are not "stable"
- one possibility: consider terms for which no β-step is applicable (so called "normal forms"; abbreviation NF) result

What are the Results of Computations?

- we do only have λ-terms
- thus, we have to express "functions" and "results" as λ-terms
- as long as β-steps are applicable, terms are not "stable"
- one possibility: consider terms for which no β-step is applicable (so called "normal forms"; abbreviation NF) result

Examples

- $\lambda x.\, x$ is in NF
- $(\lambda x.\, x)\ y$ is not in NF, since $(\lambda x.\, x)\ y \to_\beta y$ (with NF y)

Encoding Data Types

Booleans and Conditionals

Haskell	λ-Calculus	
• `True`	• True $= \lambda xy.\, x$	"ignore second argument"
• `False`	• False $= \lambda xy.\, y$	"ignore first argument"
• **`if`** b **`then`** t **`else`** e	• if $= \lambda xyz.\, x\ y\ z$	

Booleans and Conditionals

Haskell	λ-Calculus	
• `True`	• True $= \lambda xy.\, x$	"ignore second argument"
• `False`	• False $= \lambda xy.\, y$	"ignore first argument"
• **`if`** b **`then`** t **`else`** e	• if $= \lambda xyz.\, x\ y\ z$	

Examples

if True $x\ y$

if False $x\ y$

Booleans and Conditionals

Haskell	λ-Calculus	
• `True`	• True $= \lambda xy.\, x$	"ignore second argument"
• `False`	• False $= \lambda xy.\, y$	"ignore first argument"
• **`if`** b **`then`** t **`else`** e	• if $= \lambda xyz.\, x\ y\ z$	

Examples

$$\begin{array}{l} \text{if True } x\ y \\ \text{if False } x\ y \end{array} \quad \to_\beta^+ \quad \text{True } x\ y$$

Booleans and Conditionals

Haskell	λ-Calculus	
• `True`	• True $= \lambda xy.\, x$	"ignore second argument"
• `False`	• False $= \lambda xy.\, y$	"ignore first argument"
• **`if`** b **`then`** t **`else`** e	• if $= \lambda xyz.\, x\ y\ z$	

Examples

$$\text{if True } x\ y \quad \to_\beta^+ \quad \text{True } x\ y \quad \to_\beta^+ \quad x$$
$$\text{if False } x\ y$$

Booleans and Conditionals

Haskell	λ-Calculus	
• `True`	• True $= \lambda xy.\, x$	"ignore second argument"
• `False`	• False $= \lambda xy.\, y$	"ignore first argument"
• `if` b `then` t `else` e	• if $= \lambda xyz.\, x\ y\ z$	

Examples

$$\begin{array}{lll} \text{if True } x\ y & \to_\beta^+ & \text{True } x\ y \to_\beta^+ x \\ \text{if False } x\ y & \to_\beta^+ & \text{False } x\ y \end{array}$$

Booleans and Conditionals

Haskell	λ-Calculus	
• `True`	• True $= \lambda xy.\, x$	"ignore second argument"
• `False`	• False $= \lambda xy.\, y$	"ignore first argument"
• `if` b `then` t `else` e	• if $= \lambda xyz.\, x\ y\ z$	

Examples

$$\begin{array}{llllll} \text{if True } x\ y & \to_\beta^+ & \text{True } x\ y & \to_\beta^+ & x \\ \text{if False } x\ y & \to_\beta^+ & \text{False } x\ y & \to_\beta^+ & y \end{array}$$

Natural Numbers – Church Numerals

- define n-fold application of "function"

$$s^0\ t = t$$
$$s^{n+1}\ t = s\ (s^n\ t)$$

- number n is represented by term $\lambda f x.\ f^n\ x$

Natural Numbers – Church Numerals

- define n-fold application of "function"

$$s^0\ t = t$$
$$s^{n+1}\ t = s\ (s^n\ t)$$

function that applies first argument n-times to second argument

- number n is represented by term $\lambda f x.\ f^n\ x$

Natural Numbers – Church Numerals

- define n-fold application of "function"

$$s^0\ t = t$$
$$s^{n+1}\ t = s\ (s^n\ t)$$

- number n is represented by term $\lambda f x.\ f^n\ x$

Haskell vs. λ-Calculus

Haskell	λ-Calculus
`0`	$0 = \lambda f x.\ x$
`1`	$1 = \lambda f x.\ f\ x$
`n`	$n = \lambda f x.\ f^n\ x$
`(+)`	$\text{add} = \lambda m n f x.\ m\ f\ (n\ f\ x)$
`(*)`	$\text{mul} = \lambda m n f.\ m\ (n\ f)$
`(^)`	$\text{exp} = \lambda m n.\ n\ m$

Pairs

Haskell	λ-Calculus
• `(,)`	• Pair $= \lambda xyf.\, f\ x\ y$
• `fst`	• fst $= \lambda p.\, p$ True
• `snd`	• snd $= \lambda p.\, p$ False

Pairs

Haskell	λ-Calculus
`(,)`	Pair $= \lambda xyf.\, f\ x\ y$
`fst`	fst $= \lambda p.\, p$ True
`snd`	snd $= \lambda p.\, p$ False

Lists

Haskell	λ-Calculus
`(:)`	Cons $= \lambda xy.$ Pair False (Pair $x\ y$)
`head`	head $= \lambda z.$ fst (snd z)
`tail`	tail $= \lambda z.$ snd (snd z)
`[]`	Nil $= \lambda x.\, x$
`null`	null $=$ fst

Recursion

- Haskell function

```
length x = if null x then 0
           else 1 + length (tail x)
```

Recursion

- Haskell function

```
length x = if null x then 0
           else 1 + length (tail x)
```

- in λ-Calculus

$$\text{length} \stackrel{?}{=} \lambda x.\, \text{if } (\text{null } x)\ 0\ (\text{add } 1\ (\text{length } (\text{tail } x)))$$

Recursion

- Haskell function

```
length x = if null x then 0
           else 1 + length (tail x)
```

- in λ-Calculus

$$\text{length} \stackrel{?}{=} \lambda x.\, \text{if } (\text{null } x)\ 0\ (\text{add } 1\ (\text{length } (\text{tail } x)))$$

- problem: length is not allowed to occur on right-hand side

Recursion

- Haskell function

```
length x = if null x then 0
           else 1 + length (tail x)
```

- in λ-Calculus

$$\text{length} \stackrel{?}{=} \lambda x.\, \text{if } (\text{null } x)\ 0\ (\text{add } 1\ (\text{length } (\text{tail } x)))$$

- problem: length is not allowed to occur on right-hand side
- try to cope by adding additional argument

$$\text{length}' = \lambda f x.\, \text{if } (\text{null } x)\ 0\ (\text{add } 1\ (f\ (\text{tail } x)))$$

Recursion

- Haskell function

```
length x = if null x then 0
           else 1 + length (tail x)
```

- in λ-Calculus

$$\text{length} \stackrel{?}{=} \lambda x.\, \text{if } (\text{null } x)\ 0\ (\text{add } 1\ (\text{length } (\text{tail } x)))$$

- problem: length is not allowed to occur on right-hand side
- try to cope by adding additional argument

$$\text{length}' = \lambda f x.\, \text{if } (\text{null } x)\ 0\ (\text{add } 1\ (f\ (\text{tail } x)))$$

- idea: at some point f should be replaced by length again

Recursion

- Haskell function

```
length x = if null x then 0
           else 1 + length (tail x)
```

- in λ-Calculus

$$\text{length} \stackrel{?}{=} \lambda x.\, \text{if } (\text{null } x)\ 0\ (\text{add } 1\ (\text{length } (\text{tail } x)))$$

- problem: length is not allowed to occur on right-hand side
- try to cope by adding additional argument

$$\text{length}' = \lambda f x.\, \text{if } (\text{null } x)\ 0\ (\text{add } 1\ (f\ (\text{tail } x)))$$

- idea: at some point f should be replaced by length again
- partial solution:

$$\text{length} = \mathsf{Y}\ \text{length}'$$

Recursion

- Haskell function

```
length x = if null x then 0
           else 1 + length (tail x)
```

- in λ-Calculus

$$\text{length} \stackrel{?}{=} \lambda x.\, \text{if } (\text{null } x)\ 0\ (\text{add } 1\ (\text{length } (\text{tail } x)))$$

- problem: length is not allowed to occur on right-hand side
- try to cope by adding additional argument

$$\text{length}' = \lambda f x.\, \text{if } (\text{null } x)\ 0\ (\text{add } 1\ (f\ (\text{tail } x)))$$

- idea: at some point f should be replaced by length again
- partial solution:

$$\text{length} = \mathsf{Y}\ \text{length}'$$

- missing: find appropriate Y

The Y-Combinator

- **note:** combinator is λ-term without free variables

The Y-Combinator

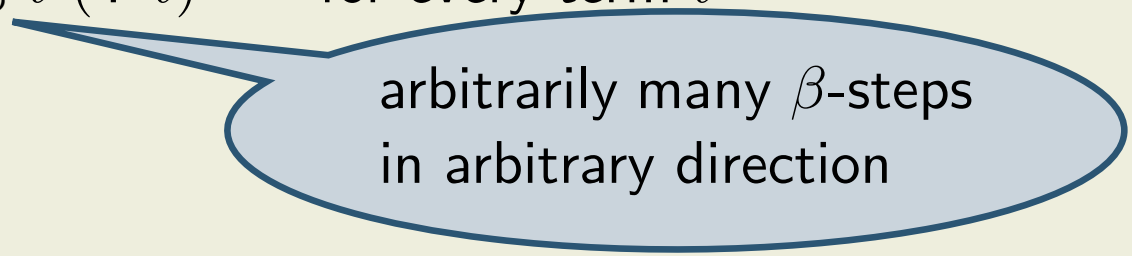
- **note:** combinator is λ-term without free variables
- Haskell Curry found combinator Y, satisfying

$$\mathsf{Y}\ t \equiv_\beta t\ (\mathsf{Y}\ t) \qquad \text{for every term } t$$

The Y-Combinator

- **note:** combinator is λ-term without free variables
- Haskell Curry found combinator Y, satisfying

$$\mathsf{Y}\ t \equiv_\beta t\ (\mathsf{Y}\ t) \qquad \text{for every term } t$$



The Y-Combinator

- **note:** combinator is λ-term without free variables
- Haskell Curry found combinator Y, satisfying

$$\mathsf{Y}\ t \equiv_\beta t\ (\mathsf{Y}\ t) \qquad \text{for every term } t$$

- fixed point property

The Y-Combinator

- **note:** combinator is λ-term without free variables
- Haskell Curry found combinator Y, satisfying

$$\mathsf{Y}\ t \equiv_\beta t\ (\mathsf{Y}\ t) \qquad \text{for every term } t$$

- fixed point property
- definition of Y is (somewhat complicated)

$$\mathsf{Y} = \lambda f.\,(\lambda x.\,f\ (x\ x))\ (\lambda x.\,f\ (x\ x))$$

The Y-Combinator

- **note:** combinator is λ-term without free variables
- Haskell Curry found combinator Y, satisfying

$$\mathsf{Y}\ t \equiv_\beta t\ (\mathsf{Y}\ t) \qquad \text{for every term } t$$

$\mathsf{Y}\ t$

- fixed point property
- definition of Y is (somewhat complicated)

$$\mathsf{Y} = \lambda f.\,(\lambda x.\,f\ (x\ x))\ (\lambda x.\,f\ (x\ x))$$

The Y-Combinator

- **note:** combinator is λ-term without free variables
- Haskell Curry found combinator Y, satisfying

$$\mathsf{Y}\ t \equiv_\beta t\ (\mathsf{Y}\ t) \qquad \text{for every term } t$$

$$\mathsf{Y}\ t \to_\beta (\lambda x.\, t\ (x\ x))\ (\lambda x.\, t\ (x\ x))$$

- fixed point property
- definition of Y is (somewhat complicated)

$$\mathsf{Y} = \lambda f.\, (\lambda x.\, f\ (x\ x))\ (\lambda x.\, f\ (x\ x))$$

The Y-Combinator

- **note:** combinator is λ-term without free variables
- Haskell Curry found combinator Y, satisfying

$$\mathsf{Y}\ t \equiv_\beta t\ (\mathsf{Y}\ t) \qquad \text{for every term } t$$

$$\mathsf{Y}\ t \to_\beta (\lambda x.\, t\ (x\ x))\ (\lambda x.\, t\ (x\ x)) \to_\beta t\ ((\lambda x.\, t\ (x\ x))\ (\lambda x.\, t\ (x\ x)))$$

- fixed point property
- definition of Y is (somewhat complicated)

$$\mathsf{Y} = \lambda f.\, (\lambda x.\, f\ (x\ x))\ (\lambda x.\, f\ (x\ x))$$

The Y-Combinator

- **note:** combinator is λ-term without free variables
- Haskell Curry found combinator Y, satisfying

$$\mathsf{Y}\ t \equiv_\beta t\ (\mathsf{Y}\ t) \qquad \text{for every term } t$$

$$\mathsf{Y}\ t \to_\beta (\lambda x.\, t\ (x\ x))\ (\lambda x.\, t\ (x\ x)) \to_\beta t\ ((\lambda x.\, t\ (x\ x))\ (\lambda x.\, t\ (x\ x)))$$

$$_\beta \leftarrow t\ (\mathsf{Y}\ t)$$

- fixed point property
- definition of Y is (somewhat complicated)

$$\mathsf{Y} = \lambda f.\, (\lambda x.\, f\ (x\ x))\ (\lambda x.\, f\ (x\ x))$$

The Y-Combinator

- **note:** combinator is λ-term without free variables
- Haskell Curry found combinator Y, satisfying

$$\mathsf{Y}\ t \equiv_\beta t\ (\mathsf{Y}\ t) \qquad \text{for every term } t$$

$$\mathsf{Y}\ t \to_\beta (\lambda x.\, t\ (x\ x))\ (\lambda x.\, t\ (x\ x)) \to_\beta t\ ((\lambda x.\, t\ (x\ x))\ (\lambda x.\, t\ (x\ x)))$$

$$_\beta \leftarrow t\ (\mathsf{Y}\ t)$$

- fixed point property
- definition of Y is (somewhat complicated)

$$\mathsf{Y} = \lambda f.\, (\lambda x.\, f\ (x\ x))\ (\lambda x.\, f\ (x\ x))$$

Example – Length

- recall that length $=$ Y g with
  $g = \lambda f x.\, \mathsf{if}\ (\mathsf{null}\ x)\ 0\ (\mathsf{add}\ 1\ (f\ (\mathsf{tail}\ x)))$
- by fixed point property we obtain length $\equiv_\beta g$ length, taking care of replacing additional parameter f in g by definition of length

Exercises (for November 17th)

1. Read the lecture notes on the lambda calculus until Section 5.
2. Use the conventions to simplify $\lambda x.\,(\lambda y.\,(\lambda z.\,((z\ (x\ y))\ x)))$ and drop the conventions in the term $\lambda abcd.\,a\ b\ c\ d$.
3. Consider F $= \lambda f x y.\,f\ y\ x$. What does F do? What does F $(\lambda x y.\,x)$ do? Reduce F $(\lambda x y.\,x)$ to NF.
4. Using `(\\)` and `union` from `Data.List` as well as the types
   ```
   type Id = String
   data Term = Var Id | App Term Term | Abs Id Term
   ```
   implement functions `vars` and `fvs` (both of type `Term -> [Id]`) computing the set (in the form of a list without duplicates) of all variables and free variables occurring in a term, respectively.
5. Implement a function `freshName` that, given a variable `x` and a list of variables `ys`, computes a pair `(x', f)` where `x'` is fresh for `ys` and `f :: Term -> Term` replaces all occurrences of `x` in a term by `x'`.
6. With the help of `vars`, `fvs`, and `freshName`, implement a function `subst :: String -> Term -> Term -> Term`, where `subst` $x\ s\ t$ computes $t[x := s]$, satisfying the freshness constraint.