Functional Programming

WS 2010/11

Christian Sternagel (VO)
Friedrich Neurauter (PS)   Ulrich Kastlunger (PS)

Computational Logic
Institute of Computer Science
University of Innsbruck

November 3, 2010

Today's Topics

- 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

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
- 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

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
- 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
- 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
- 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
- later it was shown that both models of computation are equivalent
- i.e., 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
- 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
- later it was shown that both models of computation are equivalent
- i.e., Turing-complete is the same as definable in the λ-Calculus
- λ-Calculus is underlying much of functional programming

Syntax - λ-Terms

- grammar

$$
\begin{array}{rcll}
t & \stackrel{\text{def}}{=} & x & \text{variable} \\
 & \mid & (\lambda x.\, t) & \text{(lambda) abstraction} \\
 & \mid & (t\ \ t) & \text{application}
\end{array}
$$

- all terms over set of variables $\mathcal{V}$ are denoted by $\mathcal{T}(\mathcal{V})$

Syntax - λ-Terms

- grammar

$$
\begin{array}{rcll}
t & \stackrel{\text{def}}{=} & x & \text{variable} \\
 & \mid & (\lambda x.\, t) & \text{(lambda) abstraction} \\
 & \mid & (t\ \ t) & \text{application}
\end{array}
$$

- all terms over set of variables $\mathcal{V}$ are denoted by $\mathcal{T}(\mathcal{V})$

Examples

$$
\begin{gathered}
(\lambda x.\, y) \\
(\lambda x.\, (\lambda y.\, x)) \\
(\lambda x.\, (\lambda y.\, (\lambda z.\, ((x\ \ z)\ \ (y\ \ z))))) \\
(\lambda x.\, ((\lambda y.\, (\lambda z.\, (z\ \ y)))\ \ x))
\end{gathered}
$$

Conventions

- to ease writing and reading there are some conventions
- abstraction associates to the right
- application associates to the left
- application binds stronger than abstraction (e.g., $\lambda x.\, x\ z$ is equal to $\lambda x.\, (x\ z)$ and not to $(\lambda x.\, x)\ z$)
- nested lambdas are combined

Conventions

- to ease writing and reading there are some conventions
- abstraction associates to the right
- application associates to the left
- application binds stronger than abstraction (e.g., $\lambda x.\, x\ z$ is equal to $\lambda x.\,(x\ z)$ and not to $(\lambda x.\, x)\ z$)
- nested lambdas 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 writing and reading there are some conventions
- abstraction associates to the right
- application associates to the left
- application binds stronger than abstraction (e.g., $\lambda x.\, x\ z$ is equal to $\lambda x.\, (x\ z)$ and not to $(\lambda x.\, x)\ z$)
- nested lambdas 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

λ-Calculus

- $\lambda x.\, \text{ADD}\ x\ 1$
- $(\lambda x.\, \text{ADD}\ x\ 1)\ 2$
- IF TRUE 1 0
- PAIR 2 4
- FST (PAIR 2 4)

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-Terms and Haskell

λ-Calculus

- $\lambda x.\,\mathrm{ADD}\ x\ 1$
- $(\lambda x.\,\mathrm{ADD}\ x\ 1)\ 2$
- IF TRUE 1 0
- PAIR 2 4
- FST (PAIR 2 4)

Haskell

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

Remark

- in the above
- '0', '1', '2', '4', 'ADD', 'FST', 'IF', 'PAIR', and 'TRUE' are just abbreviations for more complex λ-terms
- supposed to "encode" the behavior of `0`, `1`, `2`, `4`, `(+)`, ...

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 a "function" to an "argument"
- in the λ-Calculus, "functions" as well as "arguments" are just λ-terms
- the rule

$$(\lambda x.\, s)\ t \quad \to_\beta \quad s\{x/t\}$$

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

Examples

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

Examples

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

Examples

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

Examples

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

Examples

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

Examples

$$
\begin{aligned}
(\lambda x.\, x)\ (\lambda x.\, x) &\quad \to_\beta \quad \lambda x.\, x \\
(\lambda xy.\, y)\ (\lambda x.\, x) &\quad \to_\beta \quad \lambda y.\, y \\
(\lambda xyz.\, x\ z\ (y\ z))\ (\lambda x.\, x) &\quad \to_\beta \quad \lambda yz.\, (\lambda x.\, x)\ z\ (y\ z) \\
(\lambda x.\, x\ x)\ (\lambda x.\, x\ x) &\quad \to_\beta \quad (\lambda x.\, x\ x)\ (\lambda x.\, x\ x) \\
\lambda x.\, x &\qquad\quad \text{no } \beta\text{-step possible}
\end{aligned}
$$

(Free and Bound) Variables of a Term

- set of variables of a term

$$\mathcal{V}\mathrm{ar}(t) \stackrel{\text{def}}{=} \begin{cases} \{t\} & \text{if } t = x \\ \{x\} \cup \mathcal{V}\mathrm{ar}(u) & \text{if } t = \lambda x.\, u \\ \mathcal{V}\mathrm{ar}(u) \cup \mathcal{V}\mathrm{ar}(v) & \text{if } t = u\ v \end{cases}$$

(Free and Bound) Variables of a Term

- set of variables of a term

$$\mathcal{V}\mathrm{ar}(t) \stackrel{\text{def}}{=} \begin{cases} \{t\} & \text{if } t = x \\ \{x\} \cup \mathcal{V}\mathrm{ar}(u) & \text{if } t = \lambda x.\, u \\ \mathcal{V}\mathrm{ar}(u) \cup \mathcal{V}\mathrm{ar}(v) & \text{if } t = u\ v \end{cases}$$

- set of free variables of a term

$$\mathcal{F}\mathcal{V}\mathrm{ar}(t) \stackrel{\text{def}}{=} \begin{cases} \{t\} & \text{if } t = x \\ \mathcal{F}\mathcal{V}\mathrm{ar}(u) \setminus \{x\} & \text{if } t = \lambda x.\, u \\ \mathcal{F}\mathcal{V}\mathrm{ar}(u) \cup \mathcal{F}\mathcal{V}\mathrm{ar}(v) & \text{if } t = u\ v \end{cases}$$

(Free and Bound) Variables of a Term

- set of variables of a term

$$\mathcal{V}\mathrm{ar}(t) \stackrel{\text{def}}{=} \begin{cases} \{t\} & \text{if } t = x \\ \{x\} \cup \mathcal{V}\mathrm{ar}(u) & \text{if } t = \lambda x.\, u \\ \mathcal{V}\mathrm{ar}(u) \cup \mathcal{V}\mathrm{ar}(v) & \text{if } t = u\ v \end{cases}$$

- set of free variables of a term

$$\mathcal{FV}\mathrm{ar}(t) \stackrel{\text{def}}{=} \begin{cases} \{t\} & \text{if } t = x \\ \mathcal{FV}\mathrm{ar}(u) \setminus \{x\} & \text{if } t = \lambda x.\, u \\ \mathcal{FV}\mathrm{ar}(u) \cup \mathcal{FV}\mathrm{ar}(v) & \text{if } t = u\ v \end{cases}$$

- set of bound variables of a term

$$\mathcal{BV}\mathrm{ar}(t) \stackrel{\text{def}}{=} \begin{cases} \varnothing & \text{if } t = x \\ \{x\} \cup \mathcal{BV}\mathrm{ar}(u) & \text{if } t = \lambda x.\, u \\ \mathcal{BV}\mathrm{ar}(u) \cup \mathcal{BV}\mathrm{ar}(v) & \text{if } t = u\ v \end{cases}$$

Examples

term t	$\mathcal{V}\text{ar}(t)$	$\mathcal{FV}\text{ar}(t)$	$\mathcal{BV}\text{ar}(t)$
$\lambda x.\, x$			
$x\ y$			
$(\lambda x.\, x)\ x$			
$\lambda x.\, x\ y\ z$			

Examples

term t	$\mathcal{V}\mathrm{ar}(t)$	$\mathcal{FV}\mathrm{ar}(t)$	$\mathcal{BV}\mathrm{ar}(t)$
$\lambda x.\, x$	$\{x\}$		
$x\ y$			
$(\lambda x.\, x)\ x$			
$\lambda x.\, x\ y\ z$			

Examples

term t	$\mathcal{V}\mathrm{ar}(t)$	$\mathcal{F}\mathcal{V}\mathrm{ar}(t)$	$\mathcal{B}\mathcal{V}\mathrm{ar}(t)$
$\lambda x.\, x$	$\{x\}$	$\varnothing$	
$x\ y$			
$(\lambda x.\, x)\ x$			
$\lambda x.\, x\ y\ z$			

Examples

term t	$\mathcal{V}\mathrm{ar}(t)$	$\mathcal{FV}\mathrm{ar}(t)$	$\mathcal{BV}\mathrm{ar}(t)$
$\lambda x.\, x$	$\{x\}$	$\varnothing$	$\{x\}$
$x\ y$			
$(\lambda x.\, x)\ x$			
$\lambda x.\, x\ y\ z$			

Examples

term t	$\mathcal{V}\mathrm{ar}(t)$	$\mathcal{F}\mathcal{V}\mathrm{ar}(t)$	$\mathcal{B}\mathcal{V}\mathrm{ar}(t)$
$\lambda x.\, x$	$\{x\}$	$\varnothing$	$\{x\}$
$x\ y$	$\{x, y\}$		
$(\lambda x.\, x)\ x$			
$\lambda x.\, x\ y\ z$			

Examples

term t	$\mathcal{V}\mathrm{ar}(t)$	$\mathcal{F}\mathcal{V}\mathrm{ar}(t)$	$\mathcal{B}\mathcal{V}\mathrm{ar}(t)$
$\lambda x.\, x$	$\{x\}$	$\varnothing$	$\{x\}$
$x\ y$	$\{x, y\}$	$\{x, y\}$	
$(\lambda x.\, x)\ x$			
$\lambda x.\, x\ y\ z$			

Examples

term t	$\mathcal{V}\mathrm{ar}(t)$	$\mathcal{F}\mathcal{V}\mathrm{ar}(t)$	$\mathcal{B}\mathcal{V}\mathrm{ar}(t)$
$\lambda x.\, x$	$\{x\}$	$\varnothing$	$\{x\}$
$x\ y$	$\{x, y\}$	$\{x, y\}$	$\varnothing$
$(\lambda x.\, x)\ x$			
$\lambda x.\, x\ y\ z$			

Examples

term t	$\mathcal{V}\mathrm{ar}(t)$	$\mathcal{F}\mathcal{V}\mathrm{ar}(t)$	$\mathcal{B}\mathcal{V}\mathrm{ar}(t)$
$\lambda x.\, x$	$\{x\}$	$\varnothing$	$\{x\}$
$x\ y$	$\{x, y\}$	$\{x, y\}$	$\varnothing$
$(\lambda x.\, x)\ x$	$\{x\}$		
$\lambda x.\, x\ y\ z$			

Examples

term t	$\mathcal{V}\mathrm{ar}(t)$	$\mathcal{FV}\mathrm{ar}(t)$	$\mathcal{BV}\mathrm{ar}(t)$
$\lambda x.\, x$	$\{x\}$	$\varnothing$	$\{x\}$
$x\ y$	$\{x, y\}$	$\{x, y\}$	$\varnothing$
$(\lambda x.\, x)\ x$	$\{x\}$	$\{x\}$	
$\lambda x.\, x\ y\ z$			

Examples

term t	$\mathcal{V}\mathrm{ar}(t)$	$\mathcal{F}\mathcal{V}\mathrm{ar}(t)$	$\mathcal{B}\mathcal{V}\mathrm{ar}(t)$
$\lambda x.\, x$	$\{x\}$	$\varnothing$	$\{x\}$
$x\ y$	$\{x, y\}$	$\{x, y\}$	$\varnothing$
$(\lambda x.\, x)\ x$	$\{x\}$	$\{x\}$	$\{x\}$
$\lambda x.\, x\ y\ z$			

Examples

term t	$\mathcal{V}\mathrm{ar}(t)$	$\mathcal{F}\mathcal{V}\mathrm{ar}(t)$	$\mathcal{B}\mathcal{V}\mathrm{ar}(t)$
$\lambda x.\, x$	$\{x\}$	$\varnothing$	$\{x\}$
$x\ y$	$\{x, y\}$	$\{x, y\}$	$\varnothing$
$(\lambda x.\, x)\ x$	$\{x\}$	$\{x\}$	$\{x\}$
$\lambda x.\, x\ y\ z$	$\{x, y, z\}$		

Examples

term t	$\mathcal{V}\mathrm{ar}(t)$	$\mathcal{F}\mathcal{V}\mathrm{ar}(t)$	$\mathcal{B}\mathcal{V}\mathrm{ar}(t)$
$\lambda x.\, x$	$\{x\}$	$\varnothing$	$\{x\}$
$x\ \ y$	$\{x, y\}$	$\{x, y\}$	$\varnothing$
$(\lambda x.\, x)\ \ x$	$\{x\}$	$\{x\}$	$\{x\}$
$\lambda x.\, x\ \ y\ \ z$	$\{x, y, z\}$	$\{y, z\}$	

Examples

term t	$\mathcal{V}\mathrm{ar}(t)$	$\mathcal{F}\mathcal{V}\mathrm{ar}(t)$	$\mathcal{B}\mathcal{V}\mathrm{ar}(t)$
$\lambda x.\, x$	$\{x\}$	$\varnothing$	$\{x\}$
$x\ y$	$\{x, y\}$	$\{x, y\}$	$\varnothing$
$(\lambda x.\, x)\ x$	$\{x\}$	$\{x\}$	$\{x\}$
$\lambda x.\, x\ y\ z$	$\{x, y, z\}$	$\{y, z\}$	$\{x\}$

Substitutions

- a substitution (for terms) is a function from variables to terms

$$\sigma : \mathcal{V} \to \mathcal{T}(\mathcal{V})$$

- 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

- a substitution (for terms) is a function from variables to terms

$$\sigma : \mathcal{V} \to \mathcal{T}(\mathcal{V})$$

- 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 (first try)

- applying substitution $\sigma = \{x/s\}$ to term t is denoted by $t\sigma$

Applying Substitutions to Terms (first try)

- applying substitution $\sigma = \{x/s\}$ to term t is denoted by $t\sigma$
- and defined by

$$t\sigma \stackrel{\text{def}}{=} \begin{cases} s & \text{if } t = x \\ y & \text{if } t = y \neq x \\ (u\sigma)\ (v\sigma) & \text{if } t = u\ v \\ \lambda x.\, u & \text{if } t = \lambda x.\, u \\ \lambda y.\, (u\sigma) & \text{if } t = \lambda y.\, u \text{ with } x \neq y \end{cases}$$

Applying Substitutions to Terms (first try)

- applying substitution $\sigma = \{x/s\}$ to term t is denoted by $t\sigma$
- and defined by

$$t\sigma \stackrel{\text{def}}{=} \begin{cases} s & \text{if } t = x \\ y & \text{if } t = y \neq x \\ (u\sigma)\ (v\sigma) & \text{if } t = u\ v \\ \lambda x.\, u & \text{if } t = \lambda x.\, u \\ \lambda y.\, (u\sigma) & \text{if } t = \lambda y.\, u \text{ with } x \neq y \end{cases}$$

- i.e., bound variables are not substituted

Applying Substitutions to Terms (first try)

- applying substitution $\sigma = \{x/s\}$ to term t is denoted by $t\sigma$
- and defined by

$$t\sigma \stackrel{\text{def}}{=} \begin{cases} s & \text{if } t = x \\ y & \text{if } t = y \neq x \\ (u\sigma)\ (v\sigma) & \text{if } t = u\ v \\ \lambda x.\, u & \text{if } t = \lambda x.\, u \\ \lambda y.\, (u\sigma) & \text{if } t = \lambda y.\, u \text{ with } x \neq y \end{cases}$$

- i.e., bound variables are not substituted

Examples

- $\sigma = \{x/\lambda x.\, x\}$
- $x\sigma = \lambda x.\, x$
- $y\sigma = y$
- $(\lambda x.\, x)\sigma = \lambda x.\, x$

Contexts and Subterms

- contexts are special terms, having a single occurrence of the special symbol “hole” $\square$

Contexts and Subterms

- contexts are special terms, having a single occurrence of the special symbol "hole" $\square$
- defined by $C \stackrel{\text{def}}{=} \square \mid \lambda x.\, C \mid C\ t \mid t\ C$

Contexts and Subterms

- contexts are special terms, having a single occurrence of the special symbol "hole" $\Box$
- defined by $C \stackrel{\text{def}}{=} \Box \mid \lambda x.\, C \mid C\ t \mid t\ C$
- $C[t]$ denotes replacing $\Box$ by t in C (result is a term, since no "hole" anymore)

Contexts and Subterms

- contexts are special terms, having a single occurrence of the special symbol "hole" $\Box$
- defined by $C \stackrel{\text{def}}{=} \Box \mid \lambda x.\, C \mid C\ t \mid t\ C$
- $C[t]$ denotes replacing $\Box$ by t in C (result is a term, since no "hole" anymore)
- we say that term s is a subterm of the term t, whenever there is some context C, s.t., $t = C[s]$

Contexts and Subterms

- contexts are special terms, having a single occurrence of the special symbol “hole” $\Box$
- defined by $C \stackrel{\text{def}}{=} \Box \mid \lambda x.\, C \mid C\ t \mid t\ C$
- $C[t]$ denotes replacing $\Box$ by t in C (result is a term, since no “hole” anymore)
- we say that term s is a subterm of the term t, whenever there is some context C, s.t., $t = C[s]$
- if moreover $C \neq \Box$ (i.e., context is nonempty), then s is a proper subterm of t

Contexts and Subterms

- contexts are special terms, having a single occurrence of the special symbol "hole" $\square$
- defined by $C \stackrel{\text{def}}{=} \square \mid \lambda x.\, C \mid C\ t \mid t\ C$
- $C[t]$ denotes replacing $\square$ by t in C (result is a term, since no "hole" anymore)
- we say that term s is a subterm of the term t, whenever there is some context C, s.t., $t = C[s]$
- if moreover $C \neq \square$ (i.e., context is nonempty), then s is a proper subterm of t

Examples

- consider $C_1 = \square$, $C_2 = x\ \square$, and $C_3 = \lambda x.\, \square\ x$
- $C_1[\lambda x.\, x] = \lambda x.\, x$
- $C_2[y] = x\ y$
- $C_3[\lambda xy.\, x] = \lambda x.\, (\lambda xy.\, x)\ x$

The β-Rule (formal definition)

- term s (β-)reduces to term t in one step iff

$$\exists C\ x\ u\ v.\, s = C[(\lambda x.\, u)\ v] \wedge t = C[u\{x/v\}]$$

The β-Rule (formal definition)

- term s (β-)reduces to term t in one step iff

$$\exists C\ x\ u\ v.\, s = C[(\lambda x.\, u)\ v] \wedge t = C[u\{x/v\}]$$

- in words: *if s has a subterm of the form* $(\lambda x.\, u)\ v$ *(an abstraction/function applied to an argument), then replacing this subterm by* $u\{x/v\}$ *is a* β*-step*

The β-Rule (formal definition)

- term s (β-)reduces to term t in one step iff

$$\exists C\ x\ u\ v.\, s = C[(\lambda x.\, u)\ v] \wedge t = C[u\{x/v\}]$$

- in words: *if s has a subterm of the form* $(\lambda x.\, u)\ v$ *(an abstraction/function applied to an argument), then replacing this subterm by* $u\{x/v\}$ *is a* β*-step*
- we call $(\lambda x.\, u)\ v$ a redex (short for *red*ucible *ex*pression), and

The β-Rule (formal definition)

- term s (β-)reduces to term t in one step iff

$$\exists C\ x\ u\ v.\, s = C[(\lambda x.\, u)\ v] \wedge t = C[u\{x/v\}]$$

- in words: *if s has a subterm of the form* $(\lambda x.\, u)\ v$ *(an abstraction/function applied to an argument), then replacing this subterm by* $u\{x/v\}$ *is a* β*-step*
- we call $(\lambda x.\, u)\ v$ a redex (short for *red*ucible *ex*pression), and
- $u\{x/v\}$ the contractum

The β-Rule (formal definition)

- term s (β-)reduces to term t in one step iff

$$\exists C\ x\ u\ v.\, s = C[(\lambda x.\, u)\ v] \wedge t = C[u\{x/v\}]$$

- in words: *if* s *has a subterm of the form* $(\lambda x.\, u)\ v$ *(an abstraction/function applied to an argument), then replacing this subterm by* $u\{x/v\}$ *is a* β*-step*
- we call $(\lambda x.\, u)\ v$ a redex (short for *red*ucible *ex*pression), and
- $u\{x/v\}$ the contractum
- $s \rightarrow_\beta^* t$ denotes a sequence $s = t_1 \rightarrow_\beta t_2 \rightarrow_\beta \cdots \rightarrow_\beta t_n = t$ with $n \geq 0$ (s (β-)reduces to t)

The β-Rule (formal definition)

- term s (β-)reduces to term t in one step iff

$$\exists C\ x\ u\ v.\, s = C[(\lambda x.\, u)\ v] \wedge t = C[u\{x/v\}]$$

- in words: *if s has a subterm of the form* $(\lambda x.\, u)\ v$ *(an abstraction/function applied to an argument), then replacing this subterm by* $u\{x/v\}$ *is a* β*-step*
- we call $(\lambda x.\, u)\ v$ a redex (short for *red*ucible *ex*pression), and
- $u\{x/v\}$ the 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 (i.e., $n > 0$) is denoted by $s \to_\beta^+ t$

Exercise

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

$$K\ \Omega$$

$$K_*\ \Omega$$

$$I\ \Omega$$

Problem - Variable Capture

- consider $\lambda xy.\,x$

Problem - Variable Capture

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

Problem - Variable Capture

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

Problem - Variable Capture

- consider $\lambda xy.\, x$
- 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 $\lambda xy.\, x$
- 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 the free y was bound when substituting for x

Problem - Variable Capture

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

Solution

- modify definition of applying substitutions to terms
- rename bound variables to avoid capture of free variables

Applying Substitutions to Terms

- let $\sigma = \{x/s\}$
- new definition

$$t\sigma \stackrel{\text{def}}{=} \begin{cases} s & \text{if } t = x \\ y & \text{if } t = y \neq x \\ (u\sigma)\ (v\sigma) & \text{if } t = u\ v \\ \lambda x.\, u & \text{if } t = \lambda x.\, u \\ \lambda y.\, (u\sigma) & \text{if } t = \lambda y.\, u \text{ with } x \neq y \text{ and } y \notin \mathcal{FV}\mathrm{ar}(s) \\ \lambda z.\, (u\{y/z\}\sigma) & \text{if } t = \lambda y.\, u \text{ with } x \neq y \text{ and } y \in \mathcal{FV}\mathrm{ar}(s) \end{cases}$$

- where z is assumed to be fresh (i.e., it is unequal to x and y, and does neither occur in u nor in s)

What are the Results of Computations?

- we do only have λ-terms

What are the Results of Computations?

- we do only have λ-terms
- i.e., we have to express “functions” and “values” as λ-terms

What are the Results of Computations?

- we do only have λ-terms
- i.e., we have to express “functions” and “values” 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
- i.e., we have to express "functions" and "values" as λ-terms
- as long as β-steps are applicable, terms are not "stable"
- thus, we define values to be terms, for which no β-step is applicable (so called "normal forms"; abbreviation NF)

What are the Results of Computations?

- we do only have λ-terms
- i.e., we have to express "functions" and "values" as λ-terms
- as long as β-steps are applicable, terms are not "stable"
- thus, we define values to be terms, for which no β-step is applicable (so called "normal forms"; abbreviation NF)

Examples

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

Encoding Data Types

Booleans and Conditionals

- in Haskell: `True`, `False`, and **`if`** b **`then`** t **`else`** e
- in the λ-Calculus:

$$\mathrm{TRUE} \stackrel{\text{def}}{=} \lambda xy.\, x \qquad \text{“ignore second argument”}$$

$$\mathrm{FALSE} \stackrel{\text{def}}{=} \lambda xy.\, y \qquad \text{“ignore first argument”}$$

$$\mathrm{IF} \stackrel{\text{def}}{=} \lambda xyz.\, x\ y\ z$$

Booleans and Conditionals

- in Haskell: `True`, `False`, and **`if`** b **`then`** t **`else`** e
- in the λ-Calculus:

$$\begin{aligned} \mathrm{TRUE} &\stackrel{\text{def}}{=} \lambda xy.\, x && \text{“ignore second argument”} \\ \mathrm{FALSE} &\stackrel{\text{def}}{=} \lambda xy.\, y && \text{“ignore first argument”} \\ \mathrm{IF} &\stackrel{\text{def}}{=} \lambda xyz.\, x\ y\ z \end{aligned}$$

Examples

$$\begin{aligned} \mathrm{IF}\ \mathrm{TRUE}\ x\ y \\ \mathrm{IF}\ \mathrm{FALSE}\ x\ y \end{aligned}$$

Booleans and Conditionals

- in Haskell: `True`, `False`, and **`if`** b **`then`** t **`else`** e
- in the λ-Calculus:

$$\begin{aligned} \mathrm{TRUE} &\stackrel{\text{def}}{=} \lambda xy.\, x && \text{“ignore second argument”} \\ \mathrm{FALSE} &\stackrel{\text{def}}{=} \lambda xy.\, y && \text{“ignore first argument”} \\ \mathrm{IF} &\stackrel{\text{def}}{=} \lambda xyz.\, x\ y\ z \end{aligned}$$

Examples

$$\begin{aligned} \mathrm{IF}\ \mathrm{TRUE}\ x\ y \quad &\rightarrow_\beta^+ \quad \mathrm{TRUE}\ x\ y \\ \mathrm{IF}\ \mathrm{FALSE}\ x\ y & \end{aligned}$$

Booleans and Conditionals

- in Haskell: `True`, `False`, and **`if`** b **`then`** t **`else`** e
- in the λ-Calculus:

$$\mathrm{TRUE} \stackrel{\text{def}}{=} \lambda xy.\, x \qquad \text{“ignore second argument”}$$

$$\mathrm{FALSE} \stackrel{\text{def}}{=} \lambda xy.\, y \qquad \text{“ignore first argument”}$$

$$\mathrm{IF} \stackrel{\text{def}}{=} \lambda xyz.\, x\ y\ z$$

Examples

$$\mathrm{IF}\ \mathrm{TRUE}\ x\ y \quad \rightarrow_{\beta}^{+} \quad \mathrm{TRUE}\ x\ y \quad \rightarrow_{\beta}^{+} \quad x$$

$$\mathrm{IF}\ \mathrm{FALSE}\ x\ y$$

Booleans and Conditionals

- in Haskell: `True`, `False`, and **`if`** b **`then`** t **`else`** e
- in the λ-Calculus:

$$\mathsf{TRUE} \stackrel{\text{def}}{=} \lambda xy.\, x \qquad \text{“ignore second argument”}$$

$$\mathsf{FALSE} \stackrel{\text{def}}{=} \lambda xy.\, y \qquad \text{“ignore first argument”}$$

$$\mathsf{IF} \stackrel{\text{def}}{=} \lambda xyz.\, x\ y\ z$$

Examples

$$\mathsf{IF}\ \mathsf{TRUE}\ x\ y \ \rightarrow_\beta^+\ \mathsf{TRUE}\ x\ y \ \rightarrow_\beta^+\ x$$

$$\mathsf{IF}\ \mathsf{FALSE}\ x\ y \ \rightarrow_\beta^+\ \mathsf{FALSE}\ x\ y$$

Booleans and Conditionals

- in Haskell: `True`, `False`, and **`if`** b **`then`** t **`else`** e
- in the λ-Calculus:

$$\mathsf{TRUE} \stackrel{\text{def}}{=} \lambda xy.\, x \qquad \text{“ignore second argument”}$$

$$\mathsf{FALSE} \stackrel{\text{def}}{=} \lambda xy.\, y \qquad \text{“ignore first argument”}$$

$$\mathsf{IF} \stackrel{\text{def}}{=} \lambda xyz.\, x\ y\ z$$

Examples

$$\mathsf{IF}\ \mathsf{TRUE}\ x\ y \rightarrow_\beta^+ \mathsf{TRUE}\ x\ y \rightarrow_\beta^+ x$$

$$\mathsf{IF}\ \mathsf{FALSE}\ x\ y \rightarrow_\beta^+ \mathsf{FALSE}\ x\ y \rightarrow_\beta^+ y$$

Natural Numbers

- define n-fold application of “function”

$$s^0 \ t \stackrel{\text{def}}{=} t$$

$$s^{n+1} \ t \stackrel{\text{def}}{=} s \ (s^n \ t)$$

- Church numerals represent numbers
- the number n is represented by the term $\lambda fx.\, f^n \ x$ (i.e., a function that applies its first argument f, n-times to its second argument x)

Natural Numbers

- define n-fold application of "function"

$$s^0\ t \stackrel{\text{def}}{=} t$$

$$s^{n+1}\ t \stackrel{\text{def}}{=} s\ (s^n\ t)$$

- Church numerals represent numbers
- the number n is represented by the term $\lambda fx.\, f^n\ x$ (i.e., a function that applies its first argument f, n-times to its second argument x)

Haskell vs. λ-Calculus

`0`	$0 \stackrel{\text{def}}{=} \lambda fx.\, x$
`1`	$1 \stackrel{\text{def}}{=} \lambda fx.\, f\ x$
`n`	$\mathsf{N} \stackrel{\text{def}}{=} \lambda fx.\, f^n\ x$
`(+)`	$\mathsf{ADD} \stackrel{\text{def}}{=} \lambda mnfx.\, m\ f\ (n\ f\ x)$
`(*)`	$\mathsf{MUL} \stackrel{\text{def}}{=} \lambda mnf.\, m\ (n\ f)$
`(^)`	$\mathsf{EXP} \stackrel{\text{def}}{=} \lambda mn.\, n\ m$

Pairs - Haskell vs. λ-Calculus

`(,)`	$\text{PAIR} \stackrel{\text{def}}{=} \lambda xyf.\, f\ x\ y$
`fst`	$\text{FST} \stackrel{\text{def}}{=} \lambda p.\, p\ \text{TRUE}$
`snd`	$\text{SND} \stackrel{\text{def}}{=} \lambda p.\, p\ \text{FALSE}$

Pairs - Haskell vs. λ-Calculus

`(,)`	$\mathsf{PAIR} \stackrel{\text{def}}{=} \lambda xyf.\, f\ x\ y$
`fst`	$\mathsf{FST} \stackrel{\text{def}}{=} \lambda p.\, p\ \mathsf{TRUE}$
`snd`	$\mathsf{SND} \stackrel{\text{def}}{=} \lambda p.\, p\ \mathsf{FALSE}$

Lists - Haskell vs. λ-Calculus

`(:)`	$\mathsf{CONS} \stackrel{\text{def}}{=} \lambda xy.\, \mathsf{PAIR}\ \mathsf{FALSE}\ (\mathsf{PAIR}\ x\ y)$
`head`	$\mathsf{HEAD} \stackrel{\text{def}}{=} \lambda z.\, \mathsf{FST}\ (\mathsf{SND}\ z)$
`tail`	$\mathsf{TAIL} \stackrel{\text{def}}{=} \lambda z.\, \mathsf{SND}\ (\mathsf{SND}\ z)$
`[]`	$\mathsf{NIL} \stackrel{\text{def}}{=} \lambda x.\, x$
`null`	$\mathsf{NULL} \stackrel{\text{def}}{=} \mathsf{FST}$

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: first try

$$\text{LENGTH} \stackrel{\text{def}}{=} \lambda x.\, \text{IF (NULL } x\text{) 0 (ADD 1 (LENGTH (TAIL } x\text{)))}$$

Recursion

- Haskell function

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

- in λ-Calculus: first try

$$\mathsf{LENGTH} \stackrel{\text{def}}{=} \lambda x.\,\mathsf{IF}\ (\mathsf{NULL}\ x)\ 0\ (\mathsf{ADD}\ 1\ (\mathsf{LENGTH}\ (\mathsf{TAIL}\ x)))$$

- 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: first try

$$\text{LENGTH} \stackrel{\text{def}}{=} \lambda x.\, \text{IF}\ (\text{NULL}\ x)\ 0\ (\text{ADD}\ 1\ (\text{LENGTH}\ (\text{TAIL}\ x)))$$

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

$$\text{LENGTH} \stackrel{\text{def}}{=} \lambda f x.\, \text{IF}\ (\text{NULL}\ x)\ 0\ (\text{ADD}\ 1\ (f\ (\text{TAIL}\ x)))$$

Recursion

- Haskell function

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

- in λ-Calculus: first try

$$\mathsf{LENGTH} \stackrel{\text{def}}{=} \lambda x.\, \mathsf{IF}\ (\mathsf{NULL}\ x)\ 0\ (\mathsf{ADD}\ 1\ (\mathsf{LENGTH}\ (\mathsf{TAIL}\ x)))$$

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

$$\mathsf{LENGTH} \stackrel{\text{def}}{=} \lambda f x.\, \mathsf{IF}\ (\mathsf{NULL}\ x)\ 0\ (\mathsf{ADD}\ 1\ (f\ (\mathsf{TAIL}\ x)))$$

- 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: first try

$$\mathsf{LENGTH} \stackrel{\text{def}}{=} \lambda x.\, \mathsf{IF}\ (\mathsf{NULL}\ x)\ 0\ (\mathsf{ADD}\ 1\ (\mathsf{LENGTH}\ (\mathsf{TAIL}\ x)))$$

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

$$\mathsf{LENGTH} \stackrel{\text{def}}{=} \lambda f x.\, \mathsf{IF}\ (\mathsf{NULL}\ x)\ 0\ (\mathsf{ADD}\ 1\ (f\ (\mathsf{TAIL}\ x)))$$

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

$$\mathsf{LENGTH} \stackrel{\text{def}}{=} \mathsf{Y}\ (\lambda f x.\, \mathsf{IF}\ (\mathsf{NULL}\ x)\ 0\ (\mathsf{ADD}\ 1\ (f\ (\mathsf{TAIL}\ x))))$$

Recursion

- Haskell function

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

- in λ-Calculus: first try

$$\mathsf{LENGTH} \stackrel{\text{def}}{=} \lambda x.\, \mathsf{IF}\ (\mathsf{NULL}\ x)\ 0\ (\mathsf{ADD}\ 1\ (\mathsf{LENGTH}\ (\mathsf{TAIL}\ x)))$$

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

$$\mathsf{LENGTH} \stackrel{\text{def}}{=} \lambda f x.\, \mathsf{IF}\ (\mathsf{NULL}\ x)\ 0\ (\mathsf{ADD}\ 1\ (f\ (\mathsf{TAIL}\ x)))$$

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

$$\mathsf{LENGTH} \stackrel{\text{def}}{=} \mathsf{Y}\ (\lambda f x.\, \mathsf{IF}\ (\mathsf{NULL}\ x)\ 0\ (\mathsf{ADD}\ 1\ (f\ (\mathsf{TAIL}\ x))))$$

- missing: find appropriate Y

The Y-Combinator

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

$$\mathsf{Y}\ t \leftrightarrow^*_\beta t\ (\mathsf{Y}\ t)$$

  for every term t
- this is called the fixed point property
- the definition is somewhat complicated

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

The Y-Combinator

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

$$\mathsf{Y}\ t \leftrightarrow_\beta^* t\ (\mathsf{Y}\ t)$$

  for every term t

- this is called the fixed point property
- the definition is somewhat complicated

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

Example - Length

- recall that $\mathsf{LENGTH} = \mathsf{Y}\ g$ with
  $g = \lambda fx.\,\mathsf{IF}\ (\mathsf{NULL}\ x)\ 0\ (\mathsf{ADD}\ 1\ (f\ (\mathsf{TAIL}\ x)))$
- by the fixed point property we obtain
  $\mathsf{LENGTH} \leftrightarrow_\beta^* g\ \mathsf{LENGTH}$, which takes care of replacing the additional parameter f in g by the definition of LENGTH

Exercises (for November 12th)

1. Read the lecture notes about the lambda-calculus.
2. Use the conventions to simplify $\lambda x.\,(\lambda y.\,(\lambda z.\,((z\ (x\ y))\ x)))$. Drop the conventions in the term $\lambda abcd.\, a\ b\ c\ d$.
3. Consider the term $t = \lambda x.\, f\ (x\ x)$, find all possible contexts C and terms s, s.t., $t = C[s]$ (those are the subterms of t).
4. Consider $\mathsf{F} \overset{\text{def}}{=} \lambda fxy.\, f\ y\ x$. What does F do? What does F $(\lambda xy.\, x)$ do? Reduce F $(\lambda xy.\, x)$ to NF.
5. Using the type

```
data Term = Var String | Lab String Term
          | App Term Term
```

   implement functions `vars`, `freeVars`, `boundVars` (all of type `Term -> [String]`), computing the respective lists of variables.
6. Implement a function `applySubst :: String -> Term -> Term -> Term`, where `applySubst` $x\ s\ t$ computes $t\{x/s\}$.