Functional Programming

Christian Sternagel Harald Zankl Evgeny Zuenko

Department of Computer Science
University of Innsbruck

WS 2017/2018

Lecture 6

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

- Evaluation Strategies
- Abstract Data Types
- Sets and Binary Search Trees

Evaluation Strategies

Recall λ-Terms

$$t ::= x \mid (t\ t) \mid (\lambda x.\, t)$$

Recall λ-Terms

$$t ::= x \mid (t\ t) \mid (\lambda x.\, t)$$

Examples

conventions	verbose	in words
$x\ y$	$(x\ y)$	"x applied to y"
$\lambda x.\, x$	$(\lambda x.\, x)$	"lambda x to x" (identity function)
$\lambda xy.\, x$	$(\lambda x.\, (\lambda y.\, x))$	"lambda x y to x"
$\lambda x.\, x\ x$	$(\lambda x.\, (x\ x))$	"lambda x to x applied to x"
$(\lambda x.\, x)\ x$	$((\lambda x.\, x)\ x)$	"lambda x to x, applied to x"

Recall β-Reduction

- term s (β-)reduces to term t in one step
- written: $s \to_\beta t$
- iff there is subterm $(\lambda x.\, u)\ v$ of s, s.t.,
- replacing $(\lambda x.\, u)\ v$ in s by $u[x := v]$ results in t

Recall β-Reduction

- term s (β-)reduces to term t in one step
- written: $s \rightarrow_\beta t$
- iff there is subterm $(\lambda x.\, u)\ v$ of s, s.t.,
- replacing $(\lambda x.\, u)\ v$ in s by $u[x := v]$ results in t

Examples

$$K = \lambda xy.\, x$$
$$I = \lambda x.\, x$$
$$\Omega = (\lambda x.\, x\ x)\ (\lambda x.\, x\ x)$$

Order of Evaluation

- consider `d x = x + x`
- the term `d (d 2)` may be evaluated as follows

```
d (d 2)
```

Order of Evaluation

- consider `d x = x + x`
- the term `d (d 2)` may be evaluated as follows

$$
\begin{array}{c}
\texttt{d (d 2)} \\
\swarrow \\
\texttt{d (2 + 2)}
\end{array}
$$

Order of Evaluation

- consider `d x = x + x`
- the term `d (d 2)` may be evaluated as follows

$$\begin{tikzcd}[column sep=small] & \texttt{d (d 2)} \arrow[dl] \arrow[dr] & \\ \texttt{d (2 + 2)} & & \texttt{d 2 + d 2} \end{tikzcd}$$

Order of Evaluation

- consider `d x = x + x`
- the term `d (d 2)` may be evaluated as follows

$$
\begin{tikzcd}
 & & \texttt{d (d 2)} \arrow[dl] \arrow[dr] & \\
 & \texttt{d (2 + 2)} \arrow[dl] & & \texttt{d 2 + d 2} \\
\texttt{d 4} & & &
\end{tikzcd}
$$

Order of Evaluation

- consider `d x = x + x`
- the term `d (d 2)` may be evaluated as follows

$$
\begin{tikzcd}
 & & \texttt{d (d 2)} \arrow[dl] \arrow[dr] & \\
 & \texttt{d (2 + 2)} \arrow[dl] \arrow[dr, bend left] & & \texttt{d 2 + d 2} \\
\texttt{d 4} & & \texttt{(2 + 2) + (2 + 2)} &
\end{tikzcd}
$$

Order of Evaluation

- consider `d x = x + x`
- the term `d (d 2)` may be evaluated as follows

$$
\begin{tikzcd}
 & & \texttt{d (d 2)} \arrow[dl] \arrow[dr] & \\
 & \texttt{d (2 + 2)} \arrow[ddl] \arrow[ddr, bend left] & & \texttt{d 2 + d 2} \arrow[dll] \\
 & \texttt{(2 + 2) + d 2} & & \\
\texttt{d 4} & & \texttt{(2 + 2) + (2 + 2)} &
\end{tikzcd}
$$

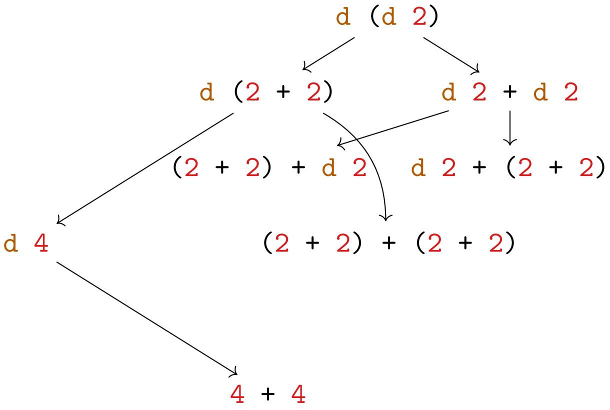
Order of Evaluation

- consider `d x = x + x`
- the term `d (d 2)` may be evaluated as follows

$$
\begin{tikzcd}
 & & & \mathtt{d\ (d\ 2)} \arrow[dl] \arrow[dr] & \\
 & & \mathtt{d\ (2\ +\ 2)} \arrow[ddll] \arrow[ddr, bend left] & & \mathtt{d\ 2\ +\ d\ 2} \arrow[dl] \arrow[d] \\
 & & & \mathtt{(2\ +\ 2)\ +\ d\ 2} & \mathtt{d\ 2\ +\ (2\ +\ 2)} \\
\mathtt{d\ 4} & & & \mathtt{(2\ +\ 2)\ +\ (2\ +\ 2)} &
\end{tikzcd}
$$

Order of Evaluation

- consider `d x = x + x`
- the term `d (d 2)` may be evaluated as follows



Order of Evaluation

- consider `d x = x + x`
- the term `d (d 2)` may be evaluated as follows

$$
\begin{tikzcd}
 & & \texttt{d (d 2)} \arrow[dl] \arrow[dr] & \\
 & \texttt{d (2 + 2)} \arrow[ddl] \arrow[ddr, bend left] & & \texttt{d 2 + d 2} \arrow[dll] \arrow[d] \\
 & \texttt{(2 + 2) + d 2} & & \texttt{d 2 + (2 + 2)} \\
\texttt{d 4} \arrow[ddr] & & \texttt{(2 + 2) + (2 + 2)} \arrow[dl] & \\
 & \texttt{4 + (2 + 2)} & & \\
 & \texttt{4 + 4} & &
\end{tikzcd}
$$

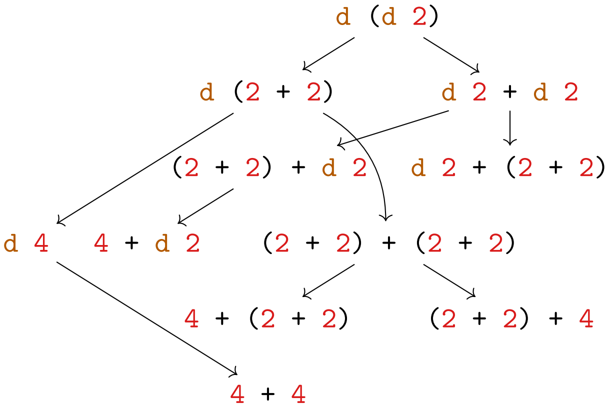
Order of Evaluation

- consider `d x = x + x`
- the term `d (d 2)` may be evaluated as follows

$$
\begin{tikzcd}
 & & \texttt{d (d 2)} \arrow[dl] \arrow[dr] & \\
 & \texttt{d (2 + 2)} \arrow[ddl] \arrow[ddr, bend left] & & \texttt{d 2 + d 2} \arrow[dll] \arrow[d] \\
 & \texttt{(2 + 2) + d 2} & & \texttt{d 2 + (2 + 2)} \\
\texttt{d 4} \arrow[ddr] & & \texttt{(2 + 2) + (2 + 2)} \arrow[dl] \arrow[dr] & \\
 & \texttt{4 + (2 + 2)} & & \texttt{(2 + 2) + 4} \\
 & \texttt{4 + 4} & &
\end{tikzcd}
$$

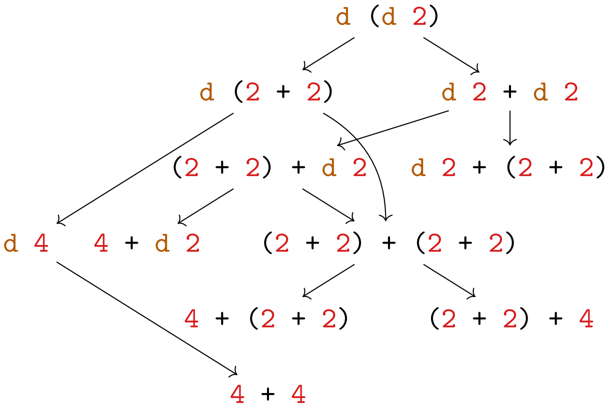
Order of Evaluation

- consider `d x = x + x`
- the term `d (d 2)` may be evaluated as follows



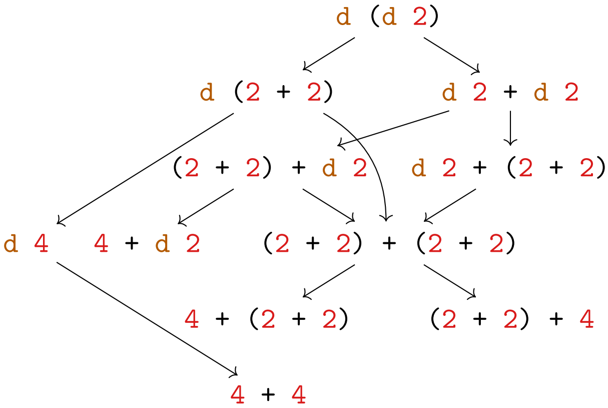
Order of Evaluation

- consider `d x = x + x`
- the term `d (d 2)` may be evaluated as follows



Order of Evaluation

- consider `d x = x + x`
- the term `d (d 2)` may be evaluated as follows



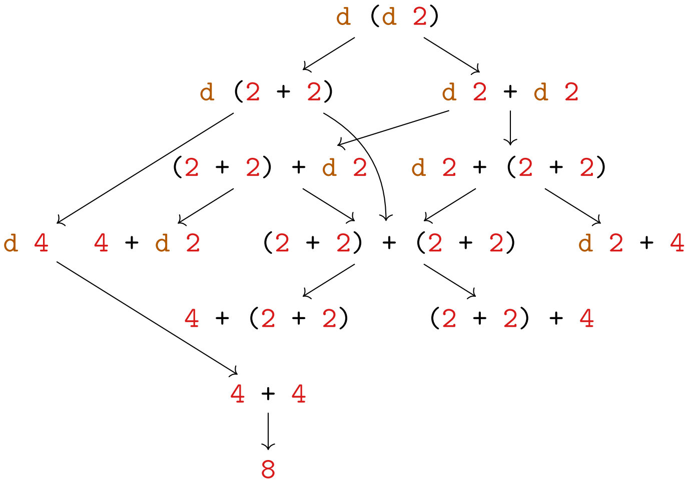
Order of Evaluation

- consider `d x = x + x`
- the term `d (d 2)` may be evaluated as follows

$$
\begin{tikzcd}
 & & & \texttt{d (d 2)} \arrow[dl] \arrow[dr] & & \\
 & & \texttt{d (2 + 2)} \arrow[ddll] \arrow[ddr, bend left] & & \texttt{d 2 + d 2} \arrow[dll] \arrow[d] & \\
 & & \texttt{(2 + 2) + d 2} \arrow[dl] \arrow[dr] & & \texttt{d 2 + (2 + 2)} \arrow[dl] \arrow[dr] & \\
\texttt{d 4} \arrow[ddrr] & \texttt{4 + d 2} & & \texttt{(2 + 2) + (2 + 2)} \arrow[dl] \arrow[dr] & & \texttt{d 2 + 4} \\
 & & \texttt{4 + (2 + 2)} & & \texttt{(2 + 2) + 4} & \\
 & & \texttt{4 + 4} & & &
\end{tikzcd}
$$

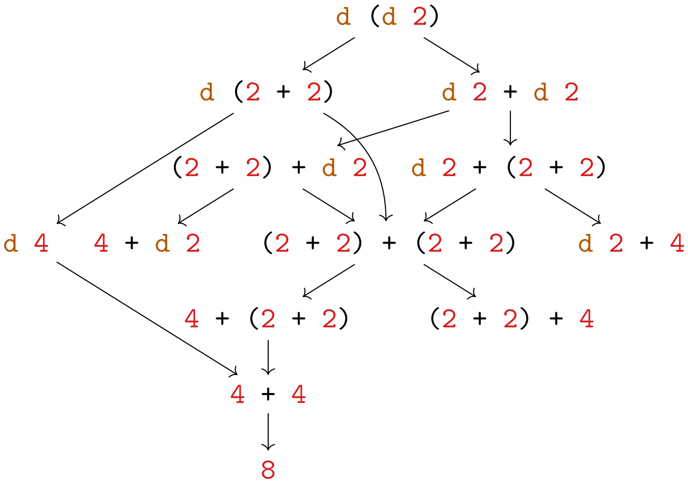
Order of Evaluation

- consider `d x = x + x`
- the term `d (d 2)` may be evaluated as follows



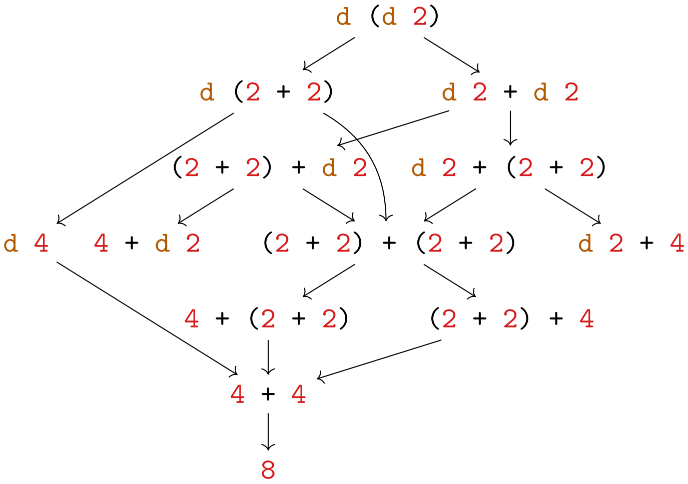
Order of Evaluation

- consider `d x = x + x`
- the term `d (d 2)` may be evaluated as follows



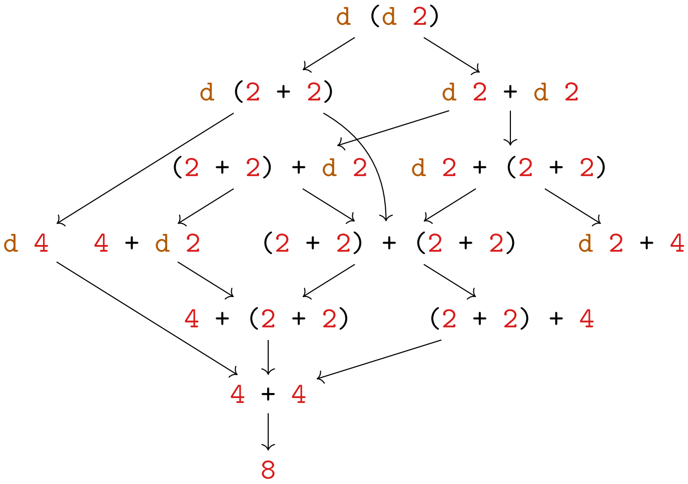
Order of Evaluation

- consider `d x = x + x`
- the term `d (d 2)` may be evaluated as follows



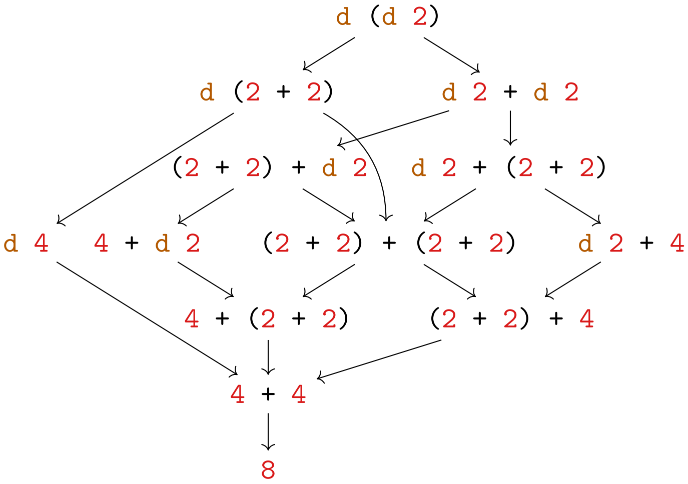
Order of Evaluation

- consider `d x = x + x`
- the term `d (d 2)` may be evaluated as follows



Order of Evaluation

- consider `d x = x + x`
- the term `d (d 2)` may be evaluated as follows



(Reduction) Strategies

- fix evaluation order
- call by value (idea: compute arguments before function calls)
- call by name (idea: compute arguments on demand only)

(Reduction) Strategies

what is called evaluation strategy in programming, is typically called reduction strategy in λ-calculus

- fix evaluation order
- call by value (idea: compute arguments before function calls)
- call by name (idea: compute arguments on demand only)

(Reduction) Strategies

- fix evaluation order
- call by value (idea: compute arguments before function calls)
- call by name (idea: compute arguments on demand only)

Example

- call by value

```
d (d 2)
```

- call by name

```
d (d 2)
```

(Reduction) Strategies

- fix evaluation order
- call by value (idea: compute arguments before function calls)
- call by name (idea: compute arguments on demand only)

Example

- call by value

  `d (d 2)` $=$ `d (2 + 2)`

- call by name

  `d (d 2)`

(Reduction) Strategies

- fix evaluation order
- call by value (idea: compute arguments before function calls)
- call by name (idea: compute arguments on demand only)

Example

- call by value

$$\begin{aligned} \texttt{d (d 2)} &= \texttt{d (2 + 2)} \\ &= \texttt{d 4} \end{aligned}$$

- call by name

$$\texttt{d (d 2)}$$

(Reduction) Strategies

- fix evaluation order
- call by value (idea: compute arguments before function calls)
- call by name (idea: compute arguments on demand only)

Example

- call by value

```
d (d 2) = d (2 + 2)
        = d 4
        = 4 + 4
```

- call by name

```
d (d 2)
```

(Reduction) Strategies

- fix evaluation order
- call by value (idea: compute arguments before function calls)
- call by name (idea: compute arguments on demand only)

Example

- call by value

```
d (d 2) = d (2 + 2)
        = d 4
        = 4 + 4
        = 8
```

- call by name

```
d (d 2)
```

(Reduction) Strategies

- fix evaluation order
- call by value (idea: compute arguments before function calls)
- call by name (idea: compute arguments on demand only)

Example

- call by value

$$
\begin{aligned}
\texttt{d (d 2)} &= \texttt{d (2 + 2)} \\
&= \texttt{d 4} \\
&= \texttt{4 + 4} \\
&= \texttt{8}
\end{aligned}
$$

- call by name

$$\texttt{d (d 2)} = \texttt{d 2 + d 2}$$

(Reduction) Strategies

- fix evaluation order
- call by value (idea: compute arguments before function calls)
- call by name (idea: compute arguments on demand only)

Example

- call by value

$$\begin{aligned} \texttt{d (d 2)} &= \texttt{d (2 + 2)} \\ &= \texttt{d 4} \\ &= \texttt{4 + 4} \\ &= \texttt{8} \end{aligned}$$

- call by name

$$\begin{aligned} \texttt{d (d 2)} &= \texttt{d 2 + d 2} \\ &= \texttt{(2 + 2) + d 2} \end{aligned}$$

(Reduction) Strategies

- fix evaluation order
- call by value (idea: compute arguments before function calls)
- call by name (idea: compute arguments on demand only)

Example

- call by value

$$\begin{aligned} \texttt{d (d 2)} &= \texttt{d (2 + 2)} \\ &= \texttt{d 4} \\ &= \texttt{4 + 4} \\ &= \texttt{8} \end{aligned}$$

- call by name

$$\begin{aligned} \texttt{d (d 2)} &= \texttt{d 2 + d 2} \\ &= \texttt{(2 + 2) + d 2} \\ &= \texttt{4 + d 2} \end{aligned}$$

(Reduction) Strategies

- fix evaluation order
- call by value (idea: compute arguments before function calls)
- call by name (idea: compute arguments on demand only)

Example

- call by value

$$\begin{aligned} \texttt{d (d 2)} &= \texttt{d (2 + 2)} \\ &= \texttt{d 4} \\ &= \texttt{4 + 4} \\ &= \texttt{8} \end{aligned}$$

- call by name

$$\begin{aligned} \texttt{d (d 2)} &= \texttt{d 2 + d 2} \\ &= \texttt{(2 + 2) + d 2} \\ &= \texttt{4 + d 2} \\ &= \texttt{4 + (2 + 2)} \end{aligned}$$

(Reduction) Strategies

- fix evaluation order
- call by value (idea: compute arguments before function calls)
- call by name (idea: compute arguments on demand only)

Example

- call by value

$$\begin{aligned} \texttt{d (d 2)} &= \texttt{d (2 + 2)} \\ &= \texttt{d 4} \\ &= \texttt{4 + 4} \\ &= \texttt{8} \end{aligned}$$

- call by name

$$\begin{aligned} \texttt{d (d 2)} &= \texttt{d 2 + d 2} \\ &= \texttt{(2 + 2) + d 2} \\ &= \texttt{4 + d 2} \\ &= \texttt{4 + (2 + 2)} \\ &= \texttt{4 + 4} \end{aligned}$$

(Reduction) Strategies

- fix evaluation order
- call by value (idea: compute arguments before function calls)
- call by name (idea: compute arguments on demand only)

Example

- call by value

$$
\begin{aligned}
\texttt{d (d 2)} &= \texttt{d (2 + 2)} \\
&= \texttt{d 4} \\
&= \texttt{4 + 4} \\
&= \texttt{8}
\end{aligned}
$$

- call by name

$$
\begin{aligned}
\texttt{d (d 2)} &= \texttt{d 2 + d 2} \\
&= \texttt{(2 + 2) + d 2} \\
&= \texttt{4 + d 2} \\
&= \texttt{4 + (2 + 2)} \\
&= \texttt{4 + 4} \\
&= \texttt{8}
\end{aligned}
$$

Applicative Order Reduction

- reduce rightmost innermost redex
- redex is innermost if it does not contain redexes itself

Applicative Order Reduction

- reduce rightmost innermost redex
- redex is innermost if it does not contain redexes itself

Example

- consider $t = (\lambda x.\, (\lambda y.\, y)\ x)\ z$
- $(\lambda y.\, y)\ x$ is innermost redex
- t is redex, but not innermost

Normal Order Reduction

- reduce leftmost outermost redex
- redex is outermost if it is not contained in another redex

Normal Order Reduction

- reduce leftmost outermost redex
- redex is outermost if it is not contained in another redex

Example

- consider $t = (\lambda x.\, (\lambda y.\, y)\ x)\ z$
- t is outermost redex
- $(\lambda y.\, y)\ x$ is redex, but not outermost

Exercises

- consider the λ-terms
- $S = \lambda xyz.\, x\ z\ (y\ z)$
- $K = \lambda xy.\, x$
- $I = \lambda x.\, x$
- reduce $S\ K\ I$ to NF using applicative order reduction
- reduce $S\ K\ I$ to NF using normal order reduction

Further Classification of λ-Terms

- a term that is not an application is called value

Further Classification of λ-Terms

- a term that is not an application is called value
- a term is called weak head normal form (WHNF) if it satisfies:

$$\begin{aligned}
\mathsf{whnf}(x) &= \mathsf{true} \\
\mathsf{whnf}(\lambda x.\, t) &= \mathsf{true} \\
\mathsf{whnf}((\lambda x.\, t)\ u) &= \mathsf{false} \\
\mathsf{whnf}(t\ u) &= \mathsf{whnf}(t)
\end{aligned}$$

Further Classification of λ-Terms

- a term that is not an application is called value
- a term is called weak head normal form (WHNF) if it satisfies:

$$\begin{aligned}
\mathsf{whnf}(x) &= \mathsf{true} \\
\mathsf{whnf}(\lambda x.\, t) &= \mathsf{true} \\
\mathsf{whnf}((\lambda x.\, t)\ u) &= \mathsf{false} \\
\mathsf{whnf}(t\ u) &= \mathsf{whnf}(t)
\end{aligned}$$

Examples

term t	value	WHNF
$(\lambda x.\, x)\ x$		
$x\ y$		
x		
$\lambda x.\, (\lambda y.\, y)\ x$		

Further Classification of λ-Terms

- a term that is not an application is called value
- a term is called weak head normal form (WHNF) if it satisfies:

$$\begin{aligned} \mathsf{whnf}(x) &= \mathsf{true} \\ \mathsf{whnf}(\lambda x.\, t) &= \mathsf{true} \\ \mathsf{whnf}((\lambda x.\, t)\ u) &= \mathsf{false} \\ \mathsf{whnf}(t\ u) &= \mathsf{whnf}(t) \end{aligned}$$

Examples

term t	value	WHNF
$(\lambda x.\, x)\ x$	✘	
$x\ y$		
x		
$\lambda x.\, (\lambda y.\, y)\ x$		

Further Classification of λ-Terms

- a term that is not an application is called value
- a term is called weak head normal form (WHNF) if it satisfies:

$$\begin{aligned}
\mathsf{whnf}(x) &= \mathsf{true} \\
\mathsf{whnf}(\lambda x.\, t) &= \mathsf{true} \\
\mathsf{whnf}((\lambda x.\, t)\ u) &= \mathsf{false} \\
\mathsf{whnf}(t\ u) &= \mathsf{whnf}(t)
\end{aligned}$$

Examples

term t	value	WHNF
$(\lambda x.\, x)\ x$	✘	✘
$x\ y$		
x		
$\lambda x.\, (\lambda y.\, y)\ x$		

Further Classification of λ-Terms

- a term that is not an application is called value
- a term is called weak head normal form (WHNF) if it satisfies:

$$\begin{aligned}\mathsf{whnf}(x) &= \mathsf{true}\\ \mathsf{whnf}(\lambda x.\, t) &= \mathsf{true}\\ \mathsf{whnf}((\lambda x.\, t)\ u) &= \mathsf{false}\\ \mathsf{whnf}(t\ u) &= \mathsf{whnf}(t)\end{aligned}$$

Examples

term t	value	WHNF
$(\lambda x.\, x)\ x$	✘	✘
$x\ y$	✘	
x		
$\lambda x.\, (\lambda y.\, y)\ x$		

Further Classification of λ-Terms

- a term that is not an application is called value
- a term is called weak head normal form (WHNF) if it satisfies:

$$\begin{aligned}
\mathsf{whnf}(x) &= \mathsf{true} \\
\mathsf{whnf}(\lambda x.\, t) &= \mathsf{true} \\
\mathsf{whnf}((\lambda x.\, t)\ u) &= \mathsf{false} \\
\mathsf{whnf}(t\ u) &= \mathsf{whnf}(t)
\end{aligned}$$

Examples

term t	value	WHNF
$(\lambda x.\, x)\ x$	✘	✘
$x\ y$	✘	✔
x		
$\lambda x.\, (\lambda y.\, y)\ x$		

Further Classification of λ-Terms

- a term that is not an application is called value
- a term is called weak head normal form (WHNF) if it satisfies:

$$\begin{aligned} \mathsf{whnf}(x) &= \mathsf{true} \\ \mathsf{whnf}(\lambda x.\, t) &= \mathsf{true} \\ \mathsf{whnf}((\lambda x.\, t)\ u) &= \mathsf{false} \\ \mathsf{whnf}(t\ u) &= \mathsf{whnf}(t) \end{aligned}$$

Examples

term t	value	WHNF
$(\lambda x.\, x)\ x$	✘	✘
$x\ y$	✘	✔
x	✔	
$\lambda x.\, (\lambda y.\, y)\ x$		

Further Classification of λ-Terms

- a term that is not an application is called value
- a term is called weak head normal form (WHNF) if it satisfies:

$$\begin{aligned} \mathsf{whnf}(x) &= \mathsf{true} \\ \mathsf{whnf}(\lambda x.\, t) &= \mathsf{true} \\ \mathsf{whnf}((\lambda x.\, t)\ u) &= \mathsf{false} \\ \mathsf{whnf}(t\ u) &= \mathsf{whnf}(t) \end{aligned}$$

Examples

term t	value	WHNF
$(\lambda x.\, x)\ x$	✘	✘
$x\ y$	✘	✔
x	✔	✔
$\lambda x.\, (\lambda y.\, y)\ x$		

Further Classification of λ-Terms

- a term that is not an application is called value
- a term is called weak head normal form (WHNF) if it satisfies:

$$\begin{aligned} \mathsf{whnf}(x) &= \mathsf{true} \\ \mathsf{whnf}(\lambda x.\, t) &= \mathsf{true} \\ \mathsf{whnf}((\lambda x.\, t)\ u) &= \mathsf{false} \\ \mathsf{whnf}(t\ u) &= \mathsf{whnf}(t) \end{aligned}$$

Examples

term t	value	WHNF
$(\lambda x.\, x)\ x$	✘	✘
$x\ y$	✘	✔
x	✔	✔
$\lambda x.\, (\lambda y.\, y)\ x$	✔	

Further Classification of λ-Terms

- a term that is not an application is called value
- a term is called weak head normal form (WHNF) if it satisfies:

$$\begin{aligned} \mathsf{whnf}(x) &= \mathsf{true} \\ \mathsf{whnf}(\lambda x.\, t) &= \mathsf{true} \\ \mathsf{whnf}((\lambda x.\, t)\ u) &= \mathsf{false} \\ \mathsf{whnf}(t\ u) &= \mathsf{whnf}(t) \end{aligned}$$

Examples

term t	value	WHNF
$(\lambda x.\, x)\ x$	✘	✘
$x\ y$	✘	✔
x	✔	✔
$\lambda x.\, (\lambda y.\, y)\ x$	✔	✔

Call by Value

- stop at values
- otherwise choose othermost redex whose right-hand side is value
- corresponds to strict (or eager) evaluation
- adopted by most programming languages

Call by Value

- stop at values
- otherwise choose othermost redex whose right-hand side is value
- corresponds to strict (or eager) evaluation
- adopted by most programming languages

Call by Name

- stop at WHNFs
- otherwise same as normal order (that is, leftmost outermost redex)
- corresponds to lazy evaluation (without memoization)
- adopted for example by Haskell

Abstract Data Types

Idea

- hide implementation details
- just provide interface
- allows us to change implementation (e.g., make more efficient) without breaking client code

Idea

- hide implementation details
- just provide interface
- allows us to change implementation (e.g., make more efficient) without breaking client code

Haskell

- consider module
```haskell
module M (T, ...) where
type T = C1 | ... | CN
```
- only name `T` is exported, but none of `C1` to `CN`
- thus we are not able to directly construct values of type `T`
- if we want to export `C1` to `CN`, we can use `T(..)` in export list

Characteristics of Sets

- order of elements not important
- no duplicates

Characteristics of Sets

- order of elements not important
- no duplicates

Examples

$$\{1, 2, 3, 5\} = \{5, 1, 3, 2\}$$
$$\{1, 1, 2, 2\} = \{1, 2\}$$

Characteristics of Sets

- order of elements not important
- no duplicates

Examples

$$\{1,2,3,5\} = \{5,1,3,2\}$$
$$\{1,1,2,2\} = \{1,2\}$$

Operations on Sets

description	notation	Haskell
empty set	$\varnothing$	`empty :: Set a`
insertion	$\{x\} \cup S$	`insert :: a -> Set a -> Set a`
membership	$e \in S$	`mem :: a -> Set a -> Bool`
union	$S \cup T$	`union :: Set a -> Set a -> Set a`
difference	$S \setminus T$	`diff :: Set a -> Set a -> Set a`

Example – Sets as Lists

```haskell
module Set (Set, empty, insert, mem, union, diff) where
import qualified Data.List as List
data Set a = Set [a]

empty :: Set a
empty = Set []

insert :: Eq a => a -> Set a -> Set a
insert x (Set xs) = Set $ List.nub $ x : xs

mem :: Eq a => a -> Set a -> Bool
x `mem` Set xs = x `elem` xs

union, diff :: Eq a => Set a -> Set a -> Set a
union (Set xs) (Set ys) = Set $ List.nub $ xs ++ ys
diff (Set xs) (Set ys) = Set $ xs List.\\ ys
```

Note – Imports

- `import M` imports all functions and types exported by module `M`

Note – Imports

- `import M` imports all functions and types exported by module `M`
- we may restrict to `f1`, ..., `fN`, writing `import M (f1, ..., fN)`

Note – Imports

- `import M` imports all functions and types exported by module `M`
- we may restrict to `f1`, ..., `fN`, writing `import M (f1, ..., fN)`
- by `import M hiding (f1, ..., fN)` we import everything except the functions `f1` to `fN`

Note – Imports

- `import M` imports all functions and types exported by module `M`
- we may restrict to `f1`, ..., `fN`, writing `import M (f1, ..., fN)`
- by `import M hiding (f1, ..., fN)` we import everything except the functions `f1` to `fN`
- `import qualified M` allows us to access all functions exported by `M` using prefix `M.`

Note – Imports

- `import M` imports all functions and types exported by module `M`
- we may restrict to `f1`, ..., `fN`, writing `import M (f1, ..., fN)`
- by `import M hiding (f1, ..., fN)` we import everything except the functions `f1` to `fN`
- `import qualified M` allows us to access all functions exported by `M` using prefix `M.`
- `import qualified M as N`, same as `import qualified M` but additionally rename `M` to `N`

New Types

- `data` with single constructor `Set` was used to hide implementation
- in this common special case use `newtype Set a = Set a` instead
- only difference: `newtype` has better performance than `data`

New Types

- `data` with single constructor `Set` was used to hide implementation
- in this common special case use `newtype Set a = Set a` instead
- only difference: `newtype` has better performance than `data`

Record Syntax

- for data type / new type `T`, instead of `C t1 ... tN`, we may use
- `C {n1 :: t1, ..., nN :: tN}` as constructor
- provides selector functions `n1 :: T -> t1`, ..., `nN :: T -> tN`

New Types

- `data` with single constructor `Set` was used to hide implementation
- in this common special case use `newtype Set a = Set a` instead
- only difference: `newtype` has better performance than `data`

Record Syntax

- for data type / new type `T`, instead of `C t1` ... `tN`, we may use
- `C {n1 :: t1, ..., nN :: tN}` as constructor
- provides selector functions `n1 :: T -> t1`, ..., `nN :: T -> tN`

New Types

- `data` with single constructor `Set` was used to hide implementation
- in this common special case use `newtype Set a = Set a` instead
- only difference: `newtype` has better performance than `data`

Record Syntax

- for data type / new type `T`, instead of `C t1 ... tN`, we may use
- `C {n1 :: t1, ..., nN :: tN}` as constructor
- provides selector functions `n1 :: T -> t1`, ..., `nN :: T -> tN`

Example

- `data Equation a = E { lhs :: a, rhs :: a }`

```
ghci> let e1 = E "10" "5+5"
ghci> let e2 = E { rhs = "5+5", lhs = "10" }
ghci> lhs e1
"10"
ghci> rhs e2
"5+5"
```

Sets and Binary Search Trees

The Type

- use type `BTree` without prefix: `import BTree (BTree(..))`
- import remaining functions from `BTree` with prefix
  `import qualified BTree`
- internal representation of set is binary tree (with selector `rep`)
  `newtype Set a = Set { rep :: BTree a }`

The Type

- use type `BTree` without prefix: `import BTree (BTree(..))`
- import remaining functions from `BTree` with prefix
  `import qualified BTree`
- internal representation of set is binary tree (with selector `rep`)
  `newtype Set a = Set { rep :: BTree a }`

Note

- `newtype Set a = Set { rep :: BTree a }` is almost the same as writing `type Set a = BTree a`
- additionally type system prevents us from "accidentally" (that is, without constructor `Set`) using `BTree`s as `Set`s
- no runtime penalty (in contrast to
  `data Set a = Set { rep :: BTree })`
- reason: `newtype` restricted to single constructor (usually of same name as newly introduced type)
- `data` may have arbitrarily many constructors (e.g., `Empty` and `Node`)

Empty Set

```haskell
empty :: Set a
empty = Set Empty
```

Empty Set

```haskell
empty :: Set a
empty = Set Empty
```

Membership

```haskell
mem :: Ord a => a -> Set a -> Bool
x `mem` s = x `memTree` rep s
  where
    memTree x Empty = False
    memTree x (Node y l r) =
      case compare x y of
        EQ -> True
        LT -> x `memTree` l
        GT -> x `memTree` r
```

Insertion

```haskell
insert :: Ord a => a -> Set a -> Set a
insert x s = Set $ insertTree x $ rep s

insertTree :: Ord a => a -> BTree a -> BTree a
insertTree x Empty          = Node x Empty Empty
insertTree x (Node y l r) =
  case compare x y of
    EQ -> Node y l r
    LT -> Node y (insertTree x l) r
    GT -> Node y l (insertTree x r)
```

Union

```
union :: Ord a => Set a -> Set a -> Set a
union s t = Set $ rep s `unionTree` rep t

unionTree :: Ord a => BTree a -> BTree a -> BTree a
unionTree Empty s          = s
unionTree (Node x l r) s =
  insertTree x $ l `unionTree` r `unionTree` s
```

Removing the Maximal Element

```
splitMaxTree :: BTree a -> Maybe (a, BTree a)
splitMaxTree Empty              = Nothing
splitMaxTree (Node x l Empty) = Just (x, l)
splitMaxTree (Node x l r)     =
  let Just (m, r') = splitMaxTree r
  in Just (m, Node x l r')
```

Removing the Maximal Element

```haskell
splitMaxTree :: BTree a -> Maybe (a, BTree a)
splitMaxTree Empty                = Nothing
splitMaxTree (Node x l Empty) = Just (x, l)
splitMaxTree (Node x l r)      =
  let Just (m, r') = splitMaxTree r
  in Just (m, Node x l r')
```

The Maybe Type

- Prelude: `data Maybe a = Just a | Nothing`
- used for type-based error handling
- if an error occurs, we return `Nothing`
- otherwise `Just` the result

Removing the Maximal Element

```haskell
splitMaxTree :: BTree a -> Maybe (a, BTree a)
splitMaxTree Empty                = Nothing
splitMaxTree (Node x l Empty) = Just (x, l)
splitMaxTree (Node x l r)     =
  let Just (m, r') = splitMaxTree r
  in Just (m, Node x l r')
```

The Maybe Type

- Prelude: `data Maybe a = Just a | Nothing`
- used for type-based error handling
- if an error occurs, we return `Nothing`
- otherwise `Just` the result

Example – Safe Head

```haskell
safeHead (x:_) = Just x
safeHead _     = Nothing
```

Remove Given Element

```haskell
removeTree :: Ord a => a -> BTree a -> BTree a
removeTree x Empty          = Empty
removeTree x (Node y l r) = case compare x y of
  LT -> Node y (removeTree x l) r
  GT -> Node y l (removeTree x r)
  EQ -> case splitMaxTree l of
    Nothing      -> r
    Just (m, l') -> Node m l' r
```

Remove Given Element

```
removeTree :: Ord a => a -> BTree a -> BTree a
removeTree x Empty          = Empty
removeTree x (Node y l r) = case compare x y of
  LT -> Node y (removeTree x l) r
  GT -> Node y l (removeTree x r)
  EQ -> case splitMaxTree l of
    Nothing      -> r
    Just (m, l') -> Node m l' r
```

For Binary Search Tree (BST)

- `x` smaller `y`: `x` can only occur in `l`
- `x` greater `y`: `x` can only occur in `r`
- `x` equals `y`: remove current node and
- combine `l` and `r` into new BST
- therefore, take maximum of `l` as new root
- guarantees that all other elements in `l` are smaller and
- that all elements in `r` are greater

Difference

```haskell
diff :: Ord a => Set a -> Set a -> Set a
diff s t = Set $ rep s `diffTree` rep t

diffTree :: Ord a => BTree a -> BTree a -> BTree a
diffTree t Empty        = t
diffTree t (Node x l r) =
  removeTree x t `diffTree` l `diffTree` r
```

Exercises (for November 24th)

1. Read Chapter 4 of Real World Haskell and Section 5 of the lecture notes on the lambda calculus.
2. Reduce 'add 2 3' to NF using applicative and normal order reduction.
3. Let `type Strat = Term -> [Term]` be the type of reduction strategies. Implement the strategy `root :: Strat` which applies a single β-step at the root (if possible).
4. Implement a strategy combinator `nested :: Strat -> Strat` that, given a strategy s, results in a new strategy which tries to apply s at all non-root positions.
5. Building on the previous functions, implement single-step call by name reduction `cbn :: Strat`.
6. Implement the function `equals :: Ord a => Set a -> Set a -> Bool`, checking whether two sets are equal.

Examples

- `root` x = `[]` – no beta-step possible
- `root` $((\lambda x.\, x)\ u)$ = `[`u`]` – root reduction
- `root` $(x\ (\lambda x.\, t)\ u)$ = `[]` – no redex at root position
- single beta-steps strictly below root position

$$\texttt{nested root}\ (((\lambda x.\, x)\ y)\ ((\lambda z.\, z)\ w)) = \\ \texttt{[}y\ ((\lambda z.\, z)\ w)\texttt{,}(\lambda x.\, x)\ y\ w\texttt{]}$$

- single-step call by name reduction

$$\texttt{cbn}\ (((\lambda x.\, x)\ ((\lambda y.\, y)\ z))\ ((\lambda w.\, w)\ v)) = \\ \texttt{[}((\lambda y.\, y)\ z)\ ((\lambda w.\, w)\ v)\texttt{]}$$