IB015 Neimperativní programování

Časová složitost, Typové třídy, Moduly

Jiří Barnat
Libor Škarvada

Časová složitost

Časová složitost algoritmu

Podstata

- Časová složitost funkce popisuje **délku výpočtu** v nejhorším případě vzhledem k velikosti vstupních parametrů.

Délka výpočtu v nejhorším případě

- Maximální počet redukčních kroků přes všechny možné výpočty aplikace programu na vstupní parametry stejné velikosti.

Časová složitost algoritmu

Podstata

- Časová složitost funkce popisuje **délku výpočtu** v nejhorším případě vzhledem k velikosti vstupních parametrů.

Délka výpočtu v nejhorším případě

- Maximální počet redukčních kroků přes všechny možné výpočty aplikace programu na vstupní parametry stejné velikosti.

Na délce záleží!

Funkce pro reverzi seznamu

Reverze seznamu funkce `reverse'`

- ```
  reverse' :: [a] -> [a]
  reverse' [] = []
  reverse' (x:s) = reverse' s ++ [x]
  ```
- ```
  (++) :: [a] -> [a] -> [a]
  [] ++ t = t
  (x:s) ++ t = x : (s++t)
  ```

Reverze seznamu funkce `reverse`

- ```
  reverse :: [a] -> [a]
  reverse = rev []
            where rev s [] = s
                  rev s (x:t) = rev (x:s) t
  ```
  ```

Reverze seznamu funkcí `reverse'`

```
reverse' :: [a] -> [a]
reverse' [] = []
reverse' (x:s) = reverse' s ++ [x]

(++) :: [a] -> [a] -> [a]
[] ++ t = t
(x:s) ++ t = x : (s++t)
```

```
     reverse' [1,2,3]
⇝    reverse' [2,3] ++ [1]
⇝    (reverse' [3] ++ [2]) ++ [1]
⇝    ((reverse' [] ++ [3]) ++ [2]) ++ [1]
⇝    (([] ++ [3]) ++ [2]) ++ [1]

⇝    ([3] ++ [2]) ++ [1]

⇝    (3 : ([] ++ [2])) ++ [1]
⇝    (3 : [2]) ++ [1]

⇝    3 : ([2] ++ [1])
⇝    3 : (2 : ([] ++ [1]))
⇝    3 : (2 : [1])   ≡   [3,2,1]
```

Reverze seznamu funkcí `reverse'`

- Délka výpočtu funkce při aplikaci na seznam délky n.
- **n+1**

```
   reverse' [1,2,3]
⇝  reverse' [2,3] ++ [1]
⇝  (reverse' [3] ++ [2]) ++ [1]
⇝  ((reverse' [] ++ [3]) ++ [2]) ++ [1]
⇝  (([] ++ [3]) ++ [2]) ++ [1]

⇝  ([3] ++ [2]) ++ [1]

⇝  (3 : ([] ++ [2])) ++ [1]
⇝  (3 : [2]) ++ [1]

⇝  3 : ([2] ++ [1])
⇝  3 : (2 : ([] ++ [1]))
⇝  3 : (2 : [1])
```

Reverze seznamu funkcí `reverse'`

- Délka výpočtu funkce při aplikaci na seznam délky n.
- n+1 + **1**

```
     reverse' [1,2,3]
⇝    reverse' [2,3] ++ [1]
⇝    (reverse' [3] ++ [2]) ++ [1]
⇝    ((reverse' [] ++ [3]) ++ [2]) ++ [1]
⇝    (([] ++ [3]) ++ [2]) ++ [1]

⇝    ([3] ++ [2]) ++ [1]

⇝    (3 : ([] ++ [2])) ++ [1]
⇝    (3 : [2]) ++ [1]

⇝    3 : ([2] ++ [1])
⇝    3 : (2 : ([] ++ [1]))
⇝    3 : (2 : [1])
```

Reverze seznamu funkcí `reverse’`

- Délka výpočtu funkce při aplikaci na seznam délky *n*.
- n+1 + 1 + **2**

```
     reverse’ [1,2,3]
⇝    reverse’ [2,3] ++ [1]
⇝    (reverse’ [3] ++ [2]) ++ [1]
⇝    ((reverse’ [] ++ [3]) ++ [2]) ++ [1]
⇝    (([] ++ [3]) ++ [2]) ++ [1]

⇝    ([3] ++ [2]) ++ [1]

⇝    (3 : ([] ++ [2])) ++ [1]
⇝    (3 : [2]) ++ [1]

⇝    3 : ([2] ++ [1])
⇝    3 : (2 : ([] ++ [1]))
⇝    3 : (2 : [1])
```

Reverze seznamu funkcí `reverse'`

- Délka výpočtu funkce při aplikaci na seznam délky *n*.
- n+1 + 1 + 2 **+ 3 + ... + n**

```
   reverse' [1,2,3]
⇝ reverse' [2,3] ++ [1]
⇝ (reverse' [3] ++ [2]) ++ [1]
⇝ ((reverse' [] ++ [3]) ++ [2]) ++ [1]
⇝ (([] ++ [3]) ++ [2]) ++ [1]

⇝ ([3] ++ [2]) ++ [1]

⇝ (3 : ([] ++ [2])) ++ [1]
⇝ (3 : [2]) ++ [1]

⇝ 3 : ([2] ++ [1])
⇝ 3 : (2 : ([] ++ [1]))
⇝ 3 : (2 : [1])
```

Reverze seznamu funkcí `reverse'`

- Délka výpočtu funkce při aplikaci na seznam délky *n*.
- n+1 + 1 + 2 + 3 + ... + n

```
     reverse' [1,2,3]
⇝    reverse' [2,3] ++ [1]
⇝    (reverse' [3] ++ [2]) ++ [1]
⇝    ((reverse' [] ++ [3]) ++ [2]) ++ [1]
⇝    (([] ++ [3]) ++ [2]) ++ [1]

⇝    ([3] ++ [2]) ++ [1]

⇝    (3 : ([] ++ [2])) ++ [1]
⇝    (3 : [2]) ++ [1]

⇝    3 : ([2] ++ [1])
⇝    3 : (2 : ([] ++ [1]))
⇝    3 : (2 : [1])
```

Reverze seznamu funkcí `reverse`

```
reverse :: [a] -> [a]
reverse = rev []
          where rev s [] = s
                rev s (x:t) = rev (x:s) t
```

```
   reverse [1,2,3]
⇝  rev [] [1,2,3]
⇝  rev [1] [2,3]
⇝  rev [2,1] [3]
⇝  rev [3,2,1] []
⇝  [3,2,1]
```

Reverze seznamu funkcí `reverse`

- Délka výpočtu funkce při aplikaci na seznam délky n.
- **1**

	reverse [1,2,3]
$\leadsto$	**rev [] [1,2,3]**
$\leadsto$	`rev [1] [2,3]`
$\leadsto$	`rev [2,1] [3]`
$\leadsto$	`rev [3,2,1] []`
$\leadsto$	`[3,2,1]`

Reverze seznamu funkcí `reverse`

- Délka výpočtu funkce při aplikaci na seznam délky *n*.
- 1 **+ n**

```
   reverse [1,2,3]
⇝  rev [] [1,2,3]
⇝  rev [1] [2,3]
⇝  rev [2,1] [3]
⇝  rev [3,2,1] []
⇝  [3,2,1]
```

Reverze seznamu funkcí `reverse`

- Délka výpočtu funkce při aplikaci na seznam délky n.
- $1 + \text{n} \mathbf{+ 1}$

```
      reverse [1,2,3]
⇝    rev [] [1,2,3]
⇝    rev [1] [2,3]
⇝    rev [2,1] [3]
⇝    rev [3,2,1] []
⇝    [3,2,1]
```

Reverze seznamu funkcí `reverse`

- Délka výpočtu funkce při aplikaci na seznam délky *n*.
- $1 + n + 1$

```
    reverse [1,2,3]
⇝   rev [] [1,2,3]
⇝   rev [1] [2,3]
⇝   rev [2,1] [3]
⇝   rev [3,2,1] []
⇝   [3,2,1]
```

Asymptotický růst funkcí

Pozorování

- Při určování časové složitosti algoritmů je nepraktické a často i obtížné určovat tuto složitost přesně.
- Funkce vyjadřující délku výpočtu vzhledem k velikosti parametru klasifikujeme podle **asymptotického chování**.

Asymptotický růst funkcí

- Při zápisu funkční hodnoty v proměnné n **rozhoduje nejrychleji rostoucí člen**. U něj navíc zanedbáváme kladnou multiplikativní konstantu.
- Podle toho hovoříme o funkcích lineárních, kvadratických, exponenciálních apod.

Přehled asymptotických funkcí

t(n)		**růst funkce t**
$1, \quad 20, \quad 729, \quad 2^{64}$		konstantní
$2\log n + 5, \quad 3\log_2 n + \log_2(\log_2 n)$		logaritmický
$n, \quad 2n+1, \quad n+\sqrt{n}$	*lineární*	polynomiální
$n\log n, \quad 3n\log n + 6n + 9$	*n log n*	
$n^2, \quad 3n^2+4n-1, \quad n^2+10\log n$	*kvadratický*	
$n^3, \quad n^3+3n^2$	*kubický*	
2^n $\left(\frac{1+\sqrt{5}}{2}\right)^n$ 3^n		exponenciální

Asymptotická složitost `reverse` a `reverse‘`

reverse‘

- Počet redukčních kroků výrazu `reverse’` $[x_1, \ldots, x_n]$ na každém seznamu délky n je

$$n + 1 + 1 + 2 + 3 + \cdots + n = \frac{n^2 + 3n + 2}{2}$$

Složitost funkce `reverse’` je **kvadratická** vzhledem k délce obraceného seznamu.

reverse

- Počet redukčních kroků výrazu `reverse` $[x_1, \ldots, x_n]$ na každém seznamu délky n je

$$1 + n + 1 = n + 2$$

Složitost funkce `reverse` je **lineární** vzhledem k délce obraceného seznamu.

Časová složitost algoritmu a problému

Časová složitost algoritmu

- Posuzuje konkrétní algoritmus.
- Nevypovídá a jiných algoritmech pro řešení téhož problému.

Časová složitost problému

- Daný problém je možné řešit různými algoritmy.
- Složitost problému vypovídá a časové složitosti nejlepšího možného algoritmu pro řešení problému.
- Určovat složitost problému je výrazně obtížnější, než určování složitosti algoritmu.
- Bez znalosti složitosti problému nelze určit, zda daný algoritmus pro řešení problému je optimální.

Počítání mocniny

Definice funkcí

```
mocnina' :: Int -> Int -> Int
mocnina' m 0 = 1
mocnina' m n = m * mocnina' m (n-1)

mocnina :: Int -> Int -> Int
mocnina m 0 = 1
mocnina m n = if even n then r else m * r
          where r = mocnina (m * m) (n `div` 2)
```

Složitost výpočtu vzhledem k exponentu

- Složitost funkce `mocnina'` je **lineární**.
- Složitost funkce `mocnina` je **logaritmická**.

Výpočet Fibonacciho číselné řady

Definice funkcí

```
fib' :: Integer -> Integer
fib' 0 = 0
fib' 1 = 1
fib' n = fib' (n-2) + fib' (n-1)

fib :: Integer -> Integer
fib = f 0 1
    where f a _ 0 = a
          f a b k = f b (a+b) (k-1)
```

Složitost vzhledem k argumentu

- Složitost funkce `fib'` je **exponenciální**.
- Složitost funkce `fib` je **lineární**.

Pozor

- Časová složitost popisuje délku výpočtu **v nejhorším případě** pro danou velikost argumentu.

Příklad

- Vyšetřujeme časovou složitost funkce `ins` vzhledem k jejímu druhému parametru.
- Funkce `ins` zařazuje prvek do seřazeného seznamu.

```
ins :: Int -> [Int] -> [Int]
ins x [] = [x]
ins x (y:t) = if x <= y then x : y : t else y : ins x t
```

Různé délky výpočtu

- Počet kroků při volání `ins x [x`$_1$`, ..., x`$_n$`]` je různý.
- Nejkratší výpočet má délku 3:

  `ins 1 [2,4,6,8]` $\leadsto^3$ `[1,2,4,6,8]`
- Nejdelší výpočet má délku $3n + 1$:

  `ins 9 [2,4,6,8]` $\leadsto^{3*4+1}$ `[2,4,6,8,9]`

Časová složitost

- Časová složitost funkce `ins` je **lineární** vzhledem k velikosti jejího druhého argumentu (tj. vzhledem k délce seznamu).

Časová složitost a redukční strategie

Pozorování

- Časová složitost závisí nejen na algoritmu (způsobu definování funkce), ale také na redukční strategii.

Příklad

- Uvažme funkcí pro uspořádání prvků v seznamu pomocí postupného zařazování.

```
inssort :: Ord a => [a] -> [a]
inssort = foldr ins []
          where ins x [] = [x]
                ins x (y:t) = if x <= y then x : y : t
                                        else y : ins x t
```

- Princip řazení funkcí `inssort`

$$
\begin{array}{ll}
 & \texttt{inssort [}x_1\texttt{, }x_2\texttt{, ..., }x_{n-1}\texttt{, }x_n\texttt{]} \\
\rightsquigarrow & \texttt{foldr ins [] [}x_1\texttt{, }x_2\texttt{, ..., }x_{n-1}\texttt{, }x_n\texttt{]} \\
\rightsquigarrow^{n+1} & \texttt{ins }x_1\texttt{ (ins }x_2\texttt{ (...(ins }x_{n-1}\texttt{ (ins }x_n\texttt{ [])) ...))}
\end{array}
$$

Příklad závislosti časové složitosti na redukční strategii

Definice funkce

- ```haskell
  inssort :: Ord a => [a] -> [a]
  inssort = foldr ins []
            where ins x [] = [x]
                  ins x (y:t) = if x <= y then x : y : t
                                          else y : ins x t
  ```
- ```haskell
  minim = head . inssort
  ```

Striktní vyhodnocování (nejhorší případ – seznam je klesající)

$$
\begin{array}{ll}
 & \texttt{minim [}x_1\texttt{, ..., }x_n\texttt{]} \\
\leadsto & \texttt{(head.inssort) [}x_1\texttt{,...,}x_n\texttt{]} \\
\leadsto & \texttt{head (inssort [}x_1\texttt{,...,}x_n\texttt{])} \\
\leadsto & \texttt{head (foldr ins [] [}x_1\texttt{,...,}x_n\texttt{])} \\
\leadsto^{n+1} & \texttt{head (ins }x_1\texttt{ (...(ins }x_{n-2}\texttt{ (ins }x_{n-1}\texttt{ (ins }x_n\texttt{[]))))...))} \\
\leadsto^{3\cdot 0+1} & \texttt{head (ins }x_1\texttt{ (...(ins }x_{n-2}\texttt{ (ins }x_{n-1}\texttt{ [}x_n\texttt{]))...))} \\
\leadsto^{3\cdot 1+1} & \texttt{head (ins }x_1\texttt{ (...(ins }x_{n-2}\texttt{ [}x_n\texttt{,}x_{n-1}\texttt{])...))} \\
\leadsto^{3\cdot 2+1} & \texttt{head (ins }x_1\texttt{ (...[}x_n\texttt{,}x_{n-1}\texttt{,}x_{n-2}\texttt{]...))} \\
\vdots & \\
\leadsto^{3\cdot(n-2)+1} & \texttt{head (ins }x_1\texttt{ [}x_n\texttt{,...,}x_2\texttt{])} \\
\leadsto^{3\cdot(n-1)+1} & \texttt{head [}x_n\texttt{,...,}x_1\texttt{]} \\
\leadsto & x_n
\end{array}
$$

Příklad závislosti časové složitosti na redukční strategii

Definice funkce

- ```
  inssort :: Ord a => [a] -> [a]
  inssort = foldr ins []
            where ins x [] = [x]
                  ins x (y:t) = if x <= y then x : y : t
                                            else y : ins x t
  ```
- ```
  minim = head . inssort
  ```

Líné vyhodnocování (nejhorší případ – nejmenší prvek na konci)

$$
\begin{array}{ll}
 & \texttt{minim [}x_1\texttt{,...,}x_n\texttt{]} \\
\leadsto & \texttt{(head.inssort) [}x_1\texttt{,...,}x_n\texttt{]} \\
\leadsto & \texttt{head (inssort [}x_1\texttt{,...,}x_n\texttt{])} \\
\leadsto & \texttt{head (foldr ins [] [}x_1\texttt{,...,}x_n\texttt{])} \\
\leadsto^{n+1} & \texttt{head (ins } x_1 \texttt{ (...(ins } x_{n-2} \texttt{ (ins } x_{n-1} \texttt{ (ins } x_n \texttt{ [])))...))} \\
\leadsto & \texttt{head (ins } x_1 \texttt{ (...(ins } x_{n-2} \texttt{ (ins } x_{n-1} \texttt{ (} x_n \texttt{ : [])))...))} \\
\leadsto^{3} & \texttt{head (ins } x_1 \texttt{ (...(ins } x_{n-2} \texttt{ (} x_n \texttt{ :(ins } x_{n-1} \texttt{ [])))...))} \\
\leadsto^{3} & \texttt{head (ins } x_1 \texttt{ (...(} x_n \texttt{ : (ins } x_{n-2} \texttt{ (ins } x_{n-1} \texttt{ [])))...))} \\
\vdots & \\
\leadsto^{3} & \texttt{head (ins } x_1 \texttt{ (} x_n \texttt{ :(ins } x_2 \texttt{ (ins } x_3 \texttt{ (...(ins } x_{n-1} \texttt{ [])...)))))} \\
\leadsto^{3} & \texttt{head (} x_n \texttt{ :(ins } x_1 \texttt{ (ins } x_2 \texttt{ (... (ins } x_{n-1} \texttt{ [])...))))} \\
\leadsto & x_n
\end{array}
$$

Příklad závislosti časové složitosti na redukční strategii

Striktní vyhodnocování

- Počet redukčních kroků výrazu `minim [x1,...,xn]` je:

$$3 + n + 1 + \sum_{k=0}^{n-1}(3k + 1) + 1 = \frac{3n^2 + n + 10}{2}$$

- Při striktním vyhodnocování má funkce **kvadratickou** časovou složitost.

Líné vyhodnocování

- Počet redukčních kroků výrazu `minim [x1,...,xn]` je:

$$3 + n + 1 + 1 + 3.(n - 1) + 1 = 4n + 3$$

- Při líném vyhodnocování má funkce **lineární** časovou složitost.

Vliv redukční strategie na časovou složitost

Pozorování

- Není pravda, že časová složitost výpočtu se při líném a striktním vyhodnocování vždy liší.
- Pokud se časová složitost liší, může se lišit víc než o jeden řádek ve zmiňované tabulce asymptotických růstů funkcí.

Příklady

- Konstantní (líně) versus exponenciální (striktně):

```
f n = const n (fib' n)
```

- Lineární líně i striktně:

  `length [a`$_1$`,...,a`$_n$`]`

Typové třídy

Připomenutí klasifikace typů

Monomorfní typy

- `not :: Bool -> Bool`
  `(&&) :: Bool -> Bool -> Bool`

Polymorfní typy

- `length :: [a] -> Int`
  `flip :: (a -> b -> c) -> b -> a -> c`

Kvalifikované typy

- `(==), (/=) :: Eq a => a -> a -> Bool`
  `sum, product :: Num a => [a] -> a`
  `minimum, maximum :: Ord a => [a] -> a`
  `print :: Show a => a -> IO ()`

Typové třídy

Význam

- Identifikují společné vlastnosti různých typů.
- Umožňují definici funkcí polymorfních typů zúžených na typy požadovaných vlastností.

Programátorský význam

- Definice a použití typových tříd umožňují sdílet kód funkcí, které dělají totéž, avšak pracují s hodnotami různých typů.
- Sdílení kódu funkcí, které dělají totéž, by měl být **svatý grál** všech programátorů.

Příklad typové třídy a jejího využití

Typová třída Eq

```haskell
class Eq a where
    (==), (/=) :: a -> a -> Bool
    x /= y = not (x == y)
```

Přidružení typů k typové třídě (deklarace instance)

```haskell
instance Eq Bool where
    False == False = True
    True == True = True
    _ == _ = False
```

```haskell
instance Eq Int where
    (==) = primEqInt
```

```haskell
instance (Eq a, Eq b) => Eq (a,b) where
    (x,y) == (u,v) = x == u && y == v
```

Využití typové třídy jinou typovou třídou

Typová třída Ord využívající typovou třídu Eq

- ```
  class (Eq a) => Ord a where
      (<=), (>=), (<), (>) :: a -> a -> Bool
      max, min :: a -> a -> a
      x >= y = y <= x
      x < y = x <= y && x /= y
      x > y = y < x
      max x y = if x>= y then x else y
      min x y = if x<= y then x else y
  ```

Deklarace instance

- ```
  instance Ord Bool where
      False <= _ = True
      _ <= True = True
      _ <= _ = False
  ```
- ```
  instance (Ord a, Ord b) => Ord (a,b) where
      (x,y) <= (u,v) = x < u || (x == u && y <= v)
  ```
  ```

Přenos vlastností typu na složený typ

Pozorování

- Instanciací lze přenést vlastnosti typu na složené typy.

Příklad

- Rozšíření uspořadatelnosti hodnot typu na uspořadatelnost seznamů hodnot daného typu.
-
```
instance (Ord a) => Ord [a] where
    [] <= _ = True
    (_:_) <= [] = False
    (x:s) <= (y:t) = x < y || (x == y && s <= t)
```

Definice typové třídy

- `class` $\left[\ (C_1\ a, \ldots, C_k\ a)\ \texttt{=>}\ \right]\ C\ a$

$$\left[\begin{array}{ll} \texttt{where} & op_1 :: typ_1 \\ & \quad\vdots \\ & op_n :: typ_n \\ & \left[\begin{array}{l} default_1 \\ \quad\vdots \\ default_m \end{array}\right] \end{array}\right]$$

Deklarace instance

- `instance` $\left[\ (C_1\ a_1, \ldots, C_k\ a_k)\ \texttt{=>}\ \right]\ C\ typ$

$$\left[\begin{array}{ll} \texttt{where} & valdef_1 \\ & \quad\vdots \\ & valdef_n \end{array}\right]$$

Přetížení

Přetížení

- Má-li třída více než jednu instanci, jsou její funkce **přetíženy**.

Přetížení operací

- Jedna operace je pro několik různých typů operandů definována obecně různým způsobem.
- To, která definice operace se použije při výpočtu, závisí na typu operandů, se kterými operace pracuje.
- Srovnej s parametricky polymorfními operacemi, které jsou definovány jednotně pro všechny typy operandů.

Příklad přetížení operace

Typová třída Num

- ```
  class (Eq a, Show a) => Num a where
      (+), (-), (*) :: a -> a -> a
      negate, abs, signum :: a -> a
  ```

Přetížení operací při deklaraci instancí

- ```
  instance Num Int where
      (+) = primPlusInt
      ⋮
  ```
- ```
  instance Num Integer where
      (+) = primPlusInteger
      ⋮
  ```
- ```
  instance Num Float where
      (+) = primPlusFloat
      ⋮
  ```

Implicitní deklarace instance

- V Haskellu lze deklarovat datový typ jako instanci typové třídy (nebo více typových tříd) též implicitně, pomocí klausule `deriving` v definici datového typu.
- Při implicitní deklaraci instance se požadované funkce definují automaticky podle způsobu zápisu hodnot definovaného typu
- Funkce `(==)` se při implicitní deklaraci instance realizuje jako syntaktická rovnost.

Příklad

- ```
  data Nat = Zero | Succ Nat
     deriving (Eq, Show)
  ```
  ```

Moduly a modulární návrh programů

Modulární návrh

Motivace

- Oddělení nezávislých, znovupoužitelných, logicky ucelených částí kódu do separátních celků – **modulů**.

Zapouzdření

- Při definici modulu je nutné explicitně vyjmenovat funkce, které mají být viditelné a použitelné mimo rozsah modulu, tzv. **exportované** funkce.
- Ostatní funkce a datové typy definované v modulu nejsou z vnějšku modulu viditelné.
- Moduly by měly exportovat jen to, co je nutné.
- Modul může exportovat hodnoty, typy a typové konstruktory, typové a konstruktorové třídy, jména modulů.

Definice Modulu v Haskellu

Obecná definice

- [module Jméno [(export$_1$, ..., export$_n$)] where]

  [import M_1 [spec$_1$]
  ⋮
  import M_m [spec$_m$]]

  [globální_deklarace]

Automatické doplnění definice

- Není-li uvedena hlavička, doplní se

  ```
  module Main (main) where
  ```

- Nevyskytuje-li se mezi importovanými moduly $M_1, \ldots M_m$ modul `Prelude`, doplní se

  ```
  import Prelude
  ```

Main

Hlavní funkce

- Program musí mít definovanou hlavní funkci – funkci `main` .
- Právě jeden modul v programu musí být `Main` .

Modul `Main`

- Modul `Main` musí exportovat hodnotu

  `main :: IO` τ

  pro nějaký typ τ, (obvykle $\tau =$ `()`).

Příklad modulu – Datový typ Fifo

Datový typ Fifo

- Datový kontejner (struktura, která uchovává prvky) přistupovaný operacemi **vlož prvek** a **vyber prvek**.
- Prvky jsou z datové struktury odebírány v tom pořadí, ve kterém byly vkládány.
- First-In-First-Out = FIFO
- Operace by měly mít konstantní časovou složitost.

Realizace v Haskellu

- Definice modulu `Fifo`
- Použití modulu:

```
import Fifo
```

Příklad Modulu – Datový typ Fifo

```
module Fifo (FifoTyp, emptyq, headq, enqueue, dequeue) where

data FifoTyp a = Q [a] [a]

emptyq :: FifoTyp a
emptyq = Q [] []

enqueue :: a -> FifoTyp a -> FifoTyp a
enqueue x (Q h t) = Q h (x:t)

headq :: FifoTyp a -> a
headq (Q (x:_) _) = x
headq (Q [] []) = error "headq: prázdná fronta"
headq (Q [] t) = headq (Q (reverse t) [])

dequeue :: FifoTyp a -> FifoTyp a
dequeue (Q (_:h) t) = Q h t
dequeue (Q [] []) = error "dequeue: prázdná fronta"
dequeue (Q [] t) = dequeue (Q (reverse t) [])
```