

Functional Programming

WS 2010/11

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

Computational Logic
Institute of Computer Science
University of Innsbruck

October 27, 2010



Intermediate Wrap-Up

Today's Topics

- Intermediate Wrap-Up
- User-Defined Types / Trees
- Input and Output

Prelude Functions You Should Know

- infix operators and special syntax
`(<=)`, `(<)`, `(==)`, `(>=)`, `(>)`, `(||)`, `(-)`, `(,)`, `(:)`,
`(</=)`, `(.)`, `(*)`, `(&&)`, `(+)`, `[]`, `[<m>..`
- other Prelude functions
`abs`, `compare`, `concat`, `const`, `div`, `drop`, `error`,
`even`, `filter`, `foldr`, `fromInteger`, `fst`, `head`,
`init`, `last`, `length`, `lines`, `map`, `max`, `min`, `mod`,
`negate`, `not`, `null`, `otherwise`, `product`,
`putStrLn`, `read`, `replicate`, `reverse`, `show`,
`signum`, `snd`, `splitAt`, `sum`, `tail`, `take`, `unlines`,
`unwords`, `words`, `zip`, `zipWith`

Syntax You Should Recognize

- **anonymous functions** / functions without names
`(\x -> 2*x) -- an anonymous function for doubling`
- **infix operators** and **sections**
 - `(+) = (\x y -> x + y)` infix to prefix
 - `x `f` y = f x y` prefix to infix
 - `(a>) = (\x -> a > x)` argument smaller than a?
 - `(>b) = (\x -> x > b)` argument greater than b?
- **patterns** and **guards**
`headIfPositive xs = case xs of`
`x:_ | x > 0 -> x`
- **list comprehensions**
`filter p xs == [x | x <- xs, p x]`
`map f xs == [f x | x <- xs]`
`concat (map f xs) == [y | x <- xs, y <- f x]`
`map (\x -> map ((,)x) ys) xs ==`
`[(x,y) | x <- xs, y <- ys]`

5/25

Equational Reasoning

- a function definition in Haskell is a (set of conditional) equation(s)
- if conditions are met, we may “replace equals by equals”
- in this way we may **evaluate** function calls by applying equations stepwise, until we reach final result

Kinds of Conditions

- “**if** $\langle b \rangle$ **then** $\langle t \rangle$ **else** $\langle e \rangle$ ” is $\langle t \rangle$, when $\langle b \rangle$ is true; and $\langle e \rangle$, otherwise
- “**case** $\langle e \rangle$ **of** $\{ \langle p_1 \rangle -> \langle e_1 \rangle; \dots; \langle p_n \rangle -> \langle e_n \rangle \}$ ” is $\langle e_i \rangle$, if $\langle e \rangle$ first matches $\langle p_i \rangle$

Primitive Operations

- for primitive operations (like $(+)$, $(*)$, \dots), we assume predefined equations
- e.g., $1 + 2 = 3$, $0 * 10 = 0$, \dots

7/25

Types and Classes

- **type signatures**, annotate functions by types
`range :: Int -> Int -> [Int]`
`range m n | m > n = []`
`| otherwise = m : range (m+1) n`
- **type synonyms**, mnemonic names for types
`type Height = Int`
`type Width = Int`
- **type classes** and **class constraints** - for every function f , specific to class C , type inference adds a C -constraint to type

Example - Type Constraints

- without type signature, we get
`ghci> :t range`
`range :: (Ord a, Num a) => a -> a -> [a]`
- $m > n$, hence m and n of class **Ord** and m and n of same type
- $m+1$, hence m of class **Num**
- m and n of same type, hence n of class **Num**

6/25

Examples - Equational Reasoning

- definition
`zip (x:xs) (y:ys) = (x,y) : zip xs ys`
`zip _ _ = []`
- evaluate `zip [1,2,3] ['a','b']`
- definition
`factorial n | n <= 1 = 1`
`| otherwise = n * factorial (n-1)`
- evaluate `factorial 3`
- definition
`head xs = case xs of x:_ -> x`
- evaluate `head "ab"`
- definition
`prod xs = if null xs then 1`
`else head xs * prod (tail xs)`
- evaluate `prod [5,6]`

8/25

User-Defined Types / Trees

9/25

Automatically Deriving Type Class Instances

- for some type classes it is possible to automatically derive instances for algebraic data types
- e.g.,

```
data List a = Nil | Cons a (List a)
deriving (Eq, Show, Read)
```
- now, we are able to use `(==)`, `show`, and `read` for `Lists`

Examples

```
ghci> Nil == Cons 1 Nil
False
ghci> show (Cons 1 (Cons 2 Nil))
"Cons 1 (Cons 2 Nil)"
ghci> read it :: List Int
Cons 1 (Cons 2 Nil)
```

11/25

Data Declarations / Algebraic Data Types

- new types are introduced by

$$\begin{aligned} \text{data } \langle T \rangle \alpha_1 \cdots \alpha_n &= \langle C_1 \rangle \tau_{11} \cdots \tau_{1m_1} \\ &\quad | \quad \vdots \\ &\quad | \quad \langle C_k \rangle \tau_{k1} \cdots \tau_{km_k} \end{aligned}$$

- where $\langle T \rangle$ is the name of the new type (starting with a capital letter) taking n type parameters α_1 to α_n
- and $\langle C_i \rangle$ is the name of the i -th (data) **constructor**, taking m_i arguments of types τ_{i1} to τ_{im_i} (which may contain only type variables among α_1 to α_n)

Examples

- `data Bool = False | True`
- `data List a = Nil | Cons a (List a)`
- `data Pair a b = Pair a b`

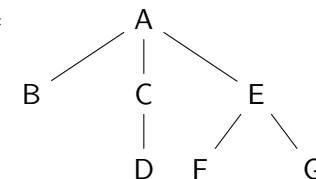
constructors and type names live in different name spaces_{10/25}

Definition - Tree

- (**rooted**) tree $T = (N, E)$
- with set of nodes N
- and set of edges/vertices $E \subseteq N \times N$
- unique root of T ($root(T) \in N$) without predecessor
- all other nodes have exactly one predecessor

Example

- $N = \{A, B, C, D, E, F, G\}$
- $E = \{(A, B), (A, C), (A, E), (C, D), (E, F), (E, G)\}$
- $root(T) = A$
- $T =$

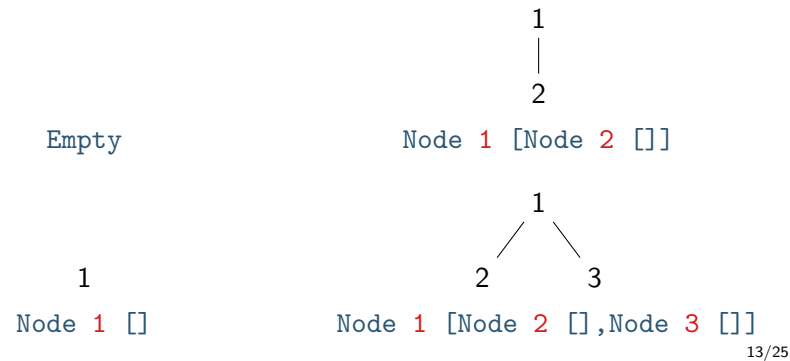


12/25

Trees in Haskell

- possible type for trees with arbitrary nodes
`data Tree a = Empty | Node a [Tree a]`
- a tree is either empty (0 nodes) or there is at least one node with content of type `a` and an arbitrary number of successor trees

Examples



Creating Trees from List

- the easy way
`fromList [] = Empty`
`fromList (x:xs) = Node x Empty (fromList xs)`
- the fair way
`make [] = Empty`
`make xs = Node z (make ys) (make zs)`
 where `m = length xs `div` 2`
 `(ys,z:zs) = splitAt m xs`
- ordered
`searchTree = foldr insert Empty`
 where `insert x Empty = Node x Empty Empty`
 `insert x (Node y l r)`
 | `x < y` = `Node y (insert x l) r`
 | otherwise = `Node y l (insert x r)`

Binary Trees

- restrict number of successors (maximal 2)
- type
`data BTree a = Empty | Node a (BTree a) (BTree a)`
 deriving (Eq, Show, Read)

Functions on Binary Trees

- computing the number of nodes
`size :: BTree a -> Integer`
`size Empty = 0`
`size (Node _ l r) = size l + size r + 1`
- height - length of longest path from root to some leaf plus one
`height :: BTree a -> Integer`
`height Empty = 0`
`height (Node _ l r) = max (height l) (height r) + 1`

Transforming Trees into Lists

```
flatten Empty = []  
flatten (Node x l r) = flatten l ++ [x] ++ flatten r
```

A Sorting Algorithm for Lists

```
sort = flatten . searchTree
```

Input and Output

17/25

IO and the Type System

- consider

```
ghci> :load welcomeIO.hs
ghci> :t putStrLn
putStrLn :: String -> IO ()
ghci> :t getLine
getLine  :: IO String
ghci> :t main
main    :: IO ()
```
- `IO a` is the type of IO actions delivering results of type `a` (in addition to their IO operations)

Examples

- `String -> IO ()` - after supplying a string, we obtain an IO action (in the case of `putStrLn`, “printing”)
- `IO ()` - just IO (in the case of `main`, run our program)
- `IO String` - do some IO and deliver a string (in the case of `getLine`, the user-input)

19/25

An Initial Example

- write the file `welcomeIO.hs`

```
main = do
  putStrLn "Greetings! What's your name?"
  name <- getLine
  putStrLn (
    "Welcome to Haskell's IO, " ++ name ++ "!"
```
- compile it with GHC via

```
$ ghc --make welcomeIO.hs
```
- and run it

```
$ ./welcomeIO
Greetings! What's your name?
```

Notes

- `putStrLn` prints a string + newline
- `getLine` reads a line from standard input
- new: `do` and `<-`

18/25

Further Notes

- IO actions (everything of type `IO a`) are just descriptions of what should be done; nothing is actually done at time of specification
- only `main` may start execution of IO actions
- inside IO actions, order is important; IO actions are executed in order of appearance (once execution starts); the result of a sequence of IO actions is the result of the **last** action
- inside IO actions, `x <- action` (where `action :: IO a`) may be used to bind the result value of `action` (which has type `a`) to the name `x` (but seriously, this is actually only done, once execution starts)
- `<x> <- <a>` is not available outside IO actions

Implications

- once we are inside an IO action, we cannot escape
- strict separation between purely functional code and IO
- when `IO a` does not appear inside type signature, we can be absolutely sure that no IO (“side-effect”) is performed

20/25

Using Pure Code Inside IO Actions

- consider the program `reply.hs`

```
reply :: String -> String
reply name =
  "Pleased to meet you, " ++ name ++ ".\n" ++
  "Your name contains " ++ n ++ " characters."
  where n = show (length name)

main :: IO ()
main = do
  putStrLn "Greetings again. What's your name?"
  name <- getLine
  let niceReply = reply name
  putStrLn niceReply
```
- i.e., we may use `let <x> = <f>` (there is no `in` here!) to bind the result of the pure function `<f>` to the name `<x>`

21/25

Examples - Imitating Some GNU Commands

- `cat.hs` - print file contents

```
main = do
  [file] <- getArgs
  s <- readFile file
  putStrLn s
```
- `wc.hs` - count newlines/words/bytes in input

```
count s = ns ++ " " ++ ws ++ " " ++ bs ++ "\n"
  where ns = show (length (lines s))
        ws = show (length (words s))
        bs = show (length s)

main = interact count
```
- `uniq.hs` - omit repeated lines of input

```
main = interact (unlines . nub . lines)
```
- `sort.hs` - sort input

```
main = interact (unlines . sort . lines)
```

23/25

Some Simple IO Functions

- `return :: a -> IO a` - turn anything into an IO action
- `getArgs :: IO [String]` get command line arguments
- `putChar :: Char -> IO ()` - print character
- `putStr :: String -> IO ()` - print string
- `putStrLn :: String -> IO ()` - print string + newline
- `getChar :: IO Char` - read single character from stdin
- `getLine :: IO String` - read line (excluding newline)
- `interact :: (String -> String) -> IO ()` - use function that gets input as string and produces output as string
- `type FilePath = String`
- `readFile :: FilePath -> IO String` - read file content
- `writeFile :: FilePath -> String -> IO ()`
- `appendFile :: FilePath -> String -> IO ()`

22/25

Notes

- `getArgs :: IO [String]` is in `System.Environment`
- `nub :: Eq a => [a] -> [a]` is in `Data.List`; eliminates duplicates
- `sort :: Ord a => [a] -> [a]` is in `Data.List`; sorts a list

Do Some IO Action for Each Argument

- ```
foreach :: [a] -> (a -> IO ()) -> IO ()
foreach [] io = return ()
foreach (a:as) io = do {io a; foreach as io}
```
- better `cat.hs`

```
main = do
 files <- getArgs
 foreach files readAndPrint
 where readAndPrint file = do
 s <- readFile file
 putStrLn s
```

24/25

## Exercises (for November 5th)

1. read chapter 7 of Real World Haskell
2. evaluate the two function calls `foldr (-) 0 [1,2,3]` and `foldl (-) 0 [1,2,3]` by equational reasoning (using the definitions from the standard `Prelude`)
3. implement the predicate `isSorted :: Ord a => BTree a -> Bool`, checking whether the given tree is a search tree
4. write a program `Grep.hs` that, given a string, echos every line of its standard input, containing this string
5. modify `Grep.hs` to also print line numbers of matching lines
6. implement a function `showBTree :: Show a => BTree a -> String` that prints a nice ASCII version of a binary tree