universität
innsbruck

Functional Programming

Week 5 – Expressions, Recursion on Numbers

René Thiemann   Philipp Anrain   Marc Bußjäger   Benedikt Dornauer   Manuel Eberl
Christina Kohl   Sandra Reitinger   Christian Sternagel

Department of Computer Science

Last Lecture

- type variables: a, b, ... represent any type
- parametric polymorphism
    - one implementation that can be used for various types
    - polymorphic datatypes, e.g., **data** List a = Empty | Cons a (List a)
    - polymorphic functions, e.g., append :: List a -> List a -> List a
    - type constraints, e.g., sumList :: Num a => List a -> a
- predefined types: [a], Maybe a, Either a b, (a1,...,aN)
- predefined type classes
    - arithmetic except division: Num a
    - arithmetic including division: Fractional a
    - equality between elements: Eq a
    - smaller than and greater than: Ord a
    - conversion to Strings: Show a

This Lecture

- type synonyms
- expressions revisited
- recursion involving numbers

Type Synonyms

Type Synonyms

- Haskell offers a mechanism to create synonyms of types via the keyword **type**
  type TConstr a1 ... aN = ty
    - TConstr is a fresh name for a type constructor
    - a1 ... aN is a list of type variables
    - ty is a type that may contain any of the type variables   *man apart constructor*
    - there is no new (value-)constructor
    - ty may not include TConstr itself, i.e., no recursion allowed
- example
  data PersonDT = Person (String, Integer) -- name & year of birth
  type PersonTS = (String, Integer)
    - the types PersonTS and (String, Integer) are identical,
      ((("Jane", 1980) :: PersonTS) :: (String, Integer)) :: PersonTS
    - the types PersonDT is different from both (String, Integer) and PersonTS;
      ("Bob", 2002) is of type PersonTS, but not of type PersonDT;
      Person ("Bob", 2002) is of type PersonDT, but not of type PersonTS

Type Synonyms – Applications, Strings

- example applications of type synonyms
  - avoid creation of new datatypes: **type** Person = (String,Integer)
  - increase readability of code

    `type Did nb - [(N)]`

    ```
    type Month = Int
    type Day   = Int
    type Year  = Int
    type Date = (Day, Month, Year)

    createDate :: Day -> Month -> Year -> Date
    createDate d m y = (d, m, y)

    -- createDate is logically equivalent to the following function,
    -- but the type synonyms help to make the code more readable

    createDate :: Int -> Int -> Int -> (Int, Int, Int)
    createDate x y z = (x, y, z)
    ```
- in Haskell: **type** String = [Char]
  - in particular "hello" is identical to ['h', 'e', 'l', 'l', 'o']
  - all functions on lists can be applied to Strings as well, e.g. (++) :: [a] -> [a] -> [a]

Expressions Revisited

Function Definitions Revisited

- current form of function definitions

```
f :: ty                    -- optional type definition
f pat11 ... pat1M = expr1   -- first defining equation
...
f pat1M ... patNM = exprN   -- last  defining equation
```

  where expressions consist of literals, variables, and function- or constructor applications

- observations
  - case analysis only possible via patterns in left-hand sides of equations
  - case analysis on right-hand sides often desirable
  - work-around via auxiliary functions possible
  - better solution: extension of expressions

if-then-else

- most primitive form of case analysis: if-then-else
- functionality: return one of two possible results, depending on a Boolean value
  ```
  ite :: Bool -> a -> a -> a
  ite True  x y = x
  ite False x y = y
  ```
  Just gehört zu Maybe
- example application: lookup a value in a key/value-list
  ```
  lookup :: Eq a => a -> [(a,b)] -> Maybe b
  lookup x ((k,v) : ys) = ite (x == k) (Just v) (lookup x ys)
  lookup _ _ = Nothing
  ```
- if-then-else is predefined: `if ... then ... else ...`
  ```
  lookup x ((k,v) : ys) = if x == k then Just v else lookup x ys
  ```
- there is no if-then (without the else) in Haskell:
  what should be the result if the Boolean is false? — in C ?:
- remark: also `lookup` is predefined in Haskell;
  Prelude content (functions, (type-)constructors, type classes, . . .) is typeset in (green)
  vordefiniert

Case Analysis via Pattern Matching

- observation: often case analysis is required on computed values
- implementation possible via auxiliary functions
- example: evaluation of expressions with meaningful error messages

  ```haskell
  data Expr a = Var String | ... -- Numbers, Addition, ...
  eval :: Num a => [(String,a)] -> Expr a -> a
  eval ass ...      = ...                    -- all the other cases
  eval ass (Var x) = aux (lookup x ass) x -- case analysis on lookup x ass
  aux (Just i) _ = i
  aux _ x = error ("assignment does not include variable " ++ x)
  ```

- disadvantages
  - local values need to be passed as arguments to auxiliary function (here: x)
  - pollution of name space by auxiliary functions
    (aux, aux1, aux2, auX, helper, fHelper, ...)
- note: if-then-else is not sufficient for above example

Case Expressions

- case expressions support arbitrary pattern matching directly in right-hand sides

  ```
  case expr of
    pat1 -> expr1
    ...
    patN -> exprN
  ```

  - match `expr` against `pat1` to `patN` top to bottom
  - if `patI` is first match, then case-expression is evaluated to `exprI`

- example from previous slide without auxiliary function

  ```
  eval ass (Var x) = case lookup x ass of
    Just i -> i
    _ -> error ("assignment does not include variable " ++ x)
  ```
  Variablen können nach dem erstmaligen entkommen ,wiederbenutzel werden

The Layout Rule

- problem: define groups (of patterns, of function definitions, ...)
- items that start in same column are grouped together
- by increasing indentation, single item may span multiple lines
- groups end when indentation decreases
- script content is group, start nested group by `where`, `let`, `do`, or `of`
- **ignore layout:** enclose groups in '{' and '}' and separate items by ';'

Examples

with layout:
```
and b1 b2 = case b1 of
  True -> case b2 of
    True -> True
    False -> False
  False -> False
```

without layout:
```
and b1 b2 = case b1 of
  { True -> case b2 of
  { True -> True; False -> False };
    False -> False }
```

White-Space in Haskell

- because of layout rule, white-space in Haskell matters
  (in contrast to many other programming languages)
- avoid tabulators in Haskell scripts
  (tab-width of editor vs. Haskell-compiler)

Example

```
and1 b1 b2 = case b1 of          and2 b1 b2 = case b1 of
  True -> case b2 of               True -> case b2 of
    True -> True                     True -> True
    False -> False                 False -> False
```

```
ghci> and1 True False
False
```

```
ghci> and2 True False
*** error: non-exhaustive patterns
```

The `let` Construct

- `let`-expressions are used for local definitions
- syntax

[handwritten: Pattern Variablen Name …]

[handwritten: funktionen und Variablen können hier verwendet werden]

```
    let
      pat                     = expr  -- definition by pattern matching
      fname pat1 ... patN =   expr    -- function definition
    in expr                          -- result
```

[handwritten: nur bis dahin]

- each `let`-expression may contain several definitions (order irrelevant)
- definitions result in new variable-bindings and functions
  - may be used in every expression expr above
  - are not visible outside `let`-expression

Number of Real Roots via `let` Construct

```haskell
-- Prelude type and function for comparing two numbers
data Ordering = EQ | LT | GT
compare :: Ord a => a -> a -> Ordering

-- task: determine number of real roots of ax^2 + bx + c
numRoots a b c = let
    disc = b^2 - 4 * a * c   -- local variable
    analyse EQ = 1           -- local function
    analyse LT = 0
    analyse GT = 2
  in analyse (compare disc 0)
```

The `where` Construct

- `where` is similar to `let`, used for <mark>local definitions</mark>

- syntax
  ```
  f pat1 .. patM = expr            -- defining equation (or case)
      where  pat                   = expr  -- pattern matching
             fname pat1 .. patN = expr  -- function definitions
  ```

- each `where` may consist of several definitions (order irrelevant)

- local definitions introduce new variables and functions
  - may be used in every expression `expr` above
  - are not visible outside defining equation / case-expression

- remark: in contrast to `let`, when using `where` the defining equation of `f` is given first
  ```
  numRoots a b c = analyse (compare disc 0) where
      disc = b^2 - 4 * a * c   -- local variable
      analyse EQ = 1           -- local function
      analyse LT = 0
      analyse GT = 2
  ```

Guarded Equations

- defining equations within a function definition can be guarded
- syntax:

von oben nach unten

```
fname pat1 ... patM "="
    | cond1 = expr1
    | cond2 = expr2
    | ...
    where ... -- optional where-block
where each condI is a Boolean expression
```

- whenever `condI` is first condition that evaluates to `True`, then result is `exprI`
- next defining equation of `fname` considered, if no condition is satisfied

```
numRoots a b c
    | disc > 0   = 2
    | disc == 0  = 1
    | otherwise  = 0          -- otherwise = True
    where disc = b^2 - 4 * a * c -- disc is shared among cases
```

mathematische Analogie

Example: Roots

- task: compute the sum of the roots of a quadratic polynomial
- solution with potential runtime errors

```haskell
roots :: Double -> Double -> Double -> (Double, Double)
roots a b c
  | a == 0 = error "not quadratic"
  | d <  0 = error "no real roots"
  | otherwise = ((- b - r) / e, (- b + r) / e)
  where d = b * b - 4 * a * c
        e = 2 * a
        r = sqrt d

sumRoots :: Double -> Double -> Double -> Double
sumRoots a b c = let
    (x, y) = roots a b c  -- pattern match in let
  in x + y
```

- note: non-variable patterns in `let` are usually only used if they cannot fail; otherwise, use `case` instead of `let`

Example: Roots (Continued)

- task: compute the sum of the roots of a quadratic polynomial
- solution with explicit failure via `Maybe`-type

```
roots :: Double -> Double -> Double -> Maybe (Double, Double)
roots a b c
  | a == 0 = Nothing
  | d <  0 = Nothing
  | otherwise = Just ((- b - r) / e, (- b + r) / e)
  where d = b * b - 4 * a * c
        e = 2 * a
        r = sqrt d

sumRoots :: Double -> Double -> Double -> Maybe Double
sumRoots a b c =
  case roots a b c of          -- case for explicit error handling
    Just (x, y) -> Just (x + y) -- nested pattern matching
    n -> Nothing                -- can't be replaced by n -> n! (types)
```
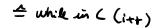
Recursion on Numbers

Recursion on Numbers

- recursive function
  ```
  f pat1 ... patN = ... (f expr1 ... exprN) ...
  ```
  where input arguments should somehow be larger than arguments in recursive call:
  ```
  (pat1, ..., patN) > (expr1, ..., exprN)   -- for some relation >
  ```
- decrease often happens in one specific argument (the i-th argument always gets smaller)
- so far the decrease in size was always w.r.t. tree size
  - length of list gets smaller
  - arithmetic expressions (`Expr`) are decomposed, i.e., number of constructors is decreased
- if argument is a number (tree size is always 1), then still recursion is possible;
  example: the value of number might decrease
- frequent cases
  - some number i is decremented until it becomes 0   (while $i \neq 0 \ldots i := i - 1$)
  - some number i is incremented until it reaches some bound n (while $i < n \ldots i := i + 1$)

$$\hat{=} \text{ while in } < \text{ (i++)}$$

Example: Factorial Function

- mathematical definition: $n! = n \cdot (n-1) \cdot \ldots \cdot 2 \cdot 1$, $0! = 1$
- implementation D: count downwards
  ```
  factorial :: Integer -> Integer
  factorial 0 = 1
  factorial n = n * factorial (n - 1)
  ```
  - in every recursive call the value of `n` is decreased
  - `factorial n` does not terminate if `n` is negative (hit Ctrl-C in ghci to stop computation)
- implementation U: count upwards, use accumulator (here: `r` stores accumulated (r)esult)
  ```
  factorial :: Integer -> Integer
  factorial n = fact 1 1 where
    fact r i
      | i <= n = fact (i * r) (i + 1)
      | otherwise = r
  ```
  - in every recursive call the value of `n - i` is decreased
  - implementation U is equivalent to imperative program (with local variables `r` and `i`)

Example: Combined Recursion

- recursion on trees and numbers can be combined   1st element = 0

- example: compute the n-th element of a list

```haskell
nth :: ~~~~~~~~~~~~~~~~ Int -> [a] -> a
nth 0 (x : _) = x                   -- indexing starts from 0
nth n (_ : xs) = nth (n - 1) xs  -- decrease of number and list-length
nth _ _ = error "no nth-element"
```

- example: take the first n-elements of a list

```haskell
take :: Int -> [a] -> [a]
take _ [] = []
take n (x:xs)
  | n <= 0 = []
  | otherwise = x : take (n - 1) xs -- decrease of number and list-length
```

- remarks
  - both `take` and n-th element (`!!`) are predefined
  - `drop` is predefined function that removes the first n-elements of a list
  - equality: `take n xs ++ drop n xs == xs`

Example: Creating Ranges of Values

- task: given lower bound l and upper bound u, compute list of numbers $[l, l+1, \ldots, u]$
- algorithm: increment l until $l > u$ and always add l to front of list
  ```
  range l u
      | l <= u = l : range (l + 1) u
      | otherwise = []
  ```
- remark: (a generalized version of) `range l u` is predefined and written `[l .. u]`
- example: concise definition of factorial function
  - `factorial n = product [1 .. n]`
    where `product :: Num a => [a] -> a` computes the product of a list of numbers

Summary

- type synonyms via `type`
- expressions with local definitions and case analysis
- recursion on numbers