universität innsbruck

WS 2022/2023

Functional Programming

Week 4 – Polymorphism

René Thiemann Jonathan Bodemann James Fox Joshua Ocker Daniel Rainer
Daniel Ranalter Christian Sternagel

Department of Computer Science

Last Lecture

- function definitions by pattern matching
  - allow several equations for each function
  - equations are tried from top to bottom
- patterns
  - `x`, `_`, `CName pat1 ... patN`, `x@pat`
  - variable names must be distinct
  - patterns describe shape of inputs
- recursion
  - in a defining equation of function `f` one can use `f` already in the rhs

    ```
    f pat1 ... patN = ... (f expr1 ... exprN) ...
    ```
  - the arguments in each recursive call should be smaller than in the lhs

List Examples

- task 1: append two lists, e.g., appending $[1,5]$ and $[3]$ yields $[1,5,3]$
- prerequisite: concrete representation of abstract lists in Haskell

```haskell
data List = Empty | Cons Integer List
-- abstract list   [1,5]   is represented as   Cons 1 (Cons 5 Empty)
```

- solution to task 1: pattern matching and recursion on first argument

```haskell
append Empty ys         = ys
append (Cons x xs) ys = Cons x (append xs ys)
```

  interpretation of the second equation
  - first append the remaining list `xs` and `ys` (`append xs ys`), afterwards insert `x` in front of the result
- task 2: determine last element of list
- solution: consider three cases (list with at least two elements, singleton list, empty list)

```haskell
lastElem (Cons _ xs@(Cons _ _)) = lastElem xs
lastElem (Cons x _)             = x    -- here the order of eq. matters
lastElem Empty                  = error "empty list has no last element"
```

Example – Datatypes Expr and List

- consider datatype for expressions

```
data Expr = Number Integer | Plus Expr Expr | Negate Expr
```

- task: create list of all numbers that occur in expression
- solution

```
numbers :: Expr -> List
numbers (Number x)    = Cons x Empty
numbers (Plus e1 e2) = append (numbers e1) (numbers e2)
numbers (Negate e)    = numbers e
```

- remarks
  - the rhs of the first equation must be `Cons x Empty` and not just `x`:
    the result must be a list of numbers
  - `numbers` (and also `append`) is defined via structural recursion:

    invoke the function recursively for each recursive argument of a datatype

    (`e1` and `e2` for `Plus e1 e2`, and `e` for `Negate e`, but not `x` of `Number x`)

Decomposition and Auxiliary Functions

- during the definition of new functions, often some functionality is missing
- task: define a function to remove all duplicates from a list
- solution:

```
remdups Empty = Empty
remdups (Cons x xs) = Cons x (remove x (remdups xs))
-- subtask:  define "remove x xs" to delete each x from list xs
remove x Empty = Empty
remove x (Cons y ys) = rHelper (x == y) y (remove x ys)
rHelper True  _ xs = xs
rHelper False y xs = Cons y xs
```

- remarks
  - solution above uses structural recursion: `remdups (Cons x xs)` invokes `remdups xs`
  - alternative solution with non-structural recursion: replace 2nd equation by

    ```
    remdups (Cons x xs) = Cons x (remdups (remove x xs))
    ```

Parametric Polymorphism

Limitations of Datatype Definitions

- task: define a datatype for lists of numbers and a function to compute their length

```
data IntList = EmptyIL | ConsIL Integer IntList
lenIL EmptyIL       = 0
lenIL (ConsIL _ xs) = 1 + lenIL xs
```

- task: define a datatype for lists of strings and a function to compute their length

```
data StringList = EmptySL | ConsSL String StringList
lenSL EmptySL       = 0
lenSL (ConsSL _ xs) = 1 + lenSL xs
```

- observations
  - the datatype and function definitions are nearly identical:
    only difference is type of elements (`Integer`/`String`) and type/function/constructor names
  - creating a copy for each new element type is not desirable for many reasons
    - writing the same functionality over and over again initially is tedious and error-prone
    - changing the implementation later on is even more tedious and error-prone – integrate changes for every element type
  - aim: define one generic list datatype and functions on these generic lists – polymorphism

Two Kinds of Polymorphism

- parametric polymorphism
  - key idea: provide one definition that can be used in various ways
  - examples
    - a datatype definition for arbitrary lists (parametrized by type of elements)
    - a datatype definition for arbitrary pairs (parametrized by two types)
    - ...
    - a function definition that works on parametric lists, pairs, ...;
      examples: length, append two lists, first component of pair, ...
- ad-hoc polymorphism
  - key idea: provide similar functionality under same name for different types
  - examples
    - `(==)` is equality operator; different implementations for strings, integers, floats, ...
    - `(+)` is addition operator; different implementations for integers, floats, ...
    - `minBound` gives smallest value for bounded types; different implementations for `Int`, `Char`, ...
  - advantage: uniform access (instead of `==Int`, `==String`, `==Double`)

Type Variables

- definition of polymorphic types and functions requires type variables
- type variables
  - start with a lowercase letter; usually a single letter is used, e.g., `a`, `b`, ...
  - a type variable represents any type
  - type variables can be substituted by (more concrete) types
- type `ty1` is more general than `ty2` if `ty2` can be obtained from `ty1` by a type substitution
- important: it is allowed to replace generic types with more concrete ones; whenever `expr :: ty1` and `ty1` is more general than `ty2` then `expr :: ty2`
- types `ty1` and `ty2` are equivalent if `ty1` is more general than `ty2` and vice versa
- examples
  - `a` is more general than any other type
  - `a -> b -> a` is more general than $\underbrace{\texttt{Int -> Char -> Int}}_{\texttt{a/Int, b/Char}}$, $\underbrace{\texttt{a -> Bool -> a}}_{\texttt{a/a, b/Bool}}$, $\underbrace{\texttt{c -> c -> c}}_{\texttt{a/c, b/c}}$
  - `a -> b -> a` is equivalent to `b -> a -> b`
  - `a -> b -> a` is not more general than `a -> b -> c`
  - `someFun` $\underbrace{\texttt{True}}_{\texttt{a}}$ $\underbrace{\texttt{x}}_{\texttt{b}}$ $\underbrace{\texttt{y}}_{\texttt{c}}$ `=` $\underbrace{\texttt{x}}_{\texttt{d}}$ is a function with type $\underbrace{\texttt{Bool}}_{\texttt{a/Bool}}$`-> b -> c ->` $\underbrace{\texttt{b}}_{\texttt{d/b}}$

Types Revisited

- already known: definition of (basic) Haskell expressions and patterns
- now: definition of types
- prerequisite: type constructors (`TConstr`)
  - similarity to (value-)constructors (`Cons`, `True`, ...)
    - start with uppercase letter
    - have a fixed arity
  - difference to constructors: type constructors are used to construct types
- a Haskell type has one of the following three shapes
  - `a` — a type variable
  - `TConstr ty1 ... tyN` — a type constructor of arity `N` applied to `N` types
  - `(ty)` — parentheses are allowed
- examples (type constructors of arity 0: `Char`, `Bool`, `Integer`, ...; arity 2: `->`)
  - `->` without the two arguments is not a type
  - `a -> Int` – type of functions that take an arbitrary input and deliver an `Int`
  - `Bool -> (a -> Int)` – type of f. that take a `Bool` and deliver a f. of type `a -> Int`
  - `Bool -> a -> Int` – same as above (!), `->` associates to the right
  - `(Bool -> a) -> Int` – take a function of type `Bool -> a` as input, deliver an `Int`

Class Assertions and Predefined Type Classes

- often a type variable `a` needs to be constrained to belong to a certain type class
  - a type `a` for which `(+)`, `(-)`, `(*)` is defined: type class `Num a`
  - a type `a` for which `(/)` is defined: type class `Fractional a`
  - a type `a` for which `(==)`, `(/=)` is defined: type class `Eq a`
  - a type `a` for which `(<)`, `(<=)`, ... is defined: type class `Ord a`
  - a type `a` for which `show :: a -> String` is defined: type class `Show a`
- these constraints are called class assertions in Haskell, notation via `=>`
- examples

```
f x y = x                          -- f :: a -> b -> a
g x y = x + y - 3                  -- g :: Num a => a -> a -> a
h x y = "cmp is " ++ show (x < y)  -- h :: Ord a => a -> a -> String
i x = "result: " ++ show (x + 3)   -- i :: (Num a, Show a) => a -> String
```

- type substitutions need to respect class assertions
  - `g False True` is not allowed since `Bool` is not an instance of `Num`
  - `i (5 :: Int)` is allowed since `Int` is an instance of both `Num` and `Show`

Datatypes with Parametric Polymorphism

- previous definition

```
data TName =
     CName1 type1_1 ... type1_N1
   | ...
   | CNameM typeM_1 ... typeM_NM
```

- new definition

```
data TConstr a1 ... aK =
     CName1 type1_1 ... type1_N1
   | ...
   | CNameM typeM_1 ... typeM_NM
```

  - new definition is more general (`K` can be zero)
  - `a1 ... aK` have to be distinct type variables
  - `TConstr` is a new type constructor with arity `K`
  - `a1 ... aK` can be used in any of the types `typeI_J`, but no other type variables
  - `CName1 :: type1_1 -> ... -> type1_N1 -> TConstr a1 ... aK`, etc.

Examples using Parametric Polymorphism

Parametric Lists

```
data List a = Empty | Cons a (List a)
```

- `List` is unary type constructor
- example types
  - `List a` – list of arbitrary elements
  - `List Integer` – list of integers
  - `List Bool` – list of Booleans
  - `List (List Integer)` – list whose elements are lists of integers
- type of constructors
  - `Empty :: List a`
  - `Cons :: a -> List a -> List a`
- example values
  - `Empty :: List a`, `Empty :: List Integer`, `Empty :: List (List Bool)`, ...
  - `Cons 7 (Cons 5 Empty) :: List Integer`, `Cons True Empty :: List Bool`, ...
  - $\texttt{Cons}\ \underbrace{(\texttt{Cons}\ \underbrace{7}_{\texttt{Int}}\ (\texttt{Cons}\ \underbrace{5}_{\texttt{Int}}\ \underbrace{\texttt{Empty}}_{\texttt{List Int}}))}_{\texttt{List Int}}\ \underbrace{(\texttt{Cons}\ \underbrace{\texttt{Empty}}_{\texttt{List Int}}\ \underbrace{\texttt{Empty}}_{\texttt{List (List Int)}})}_{\texttt{List (List Int)}}\ \texttt{:: List (List Int)}$
  - `Cons True (Cons 7 Empty)` not allowed, cannot mix element types

Functions on Parametric Lists

```
data List a = Empty | Cons a (List a)
```

- example programs

```
len :: List a -> Int    -- parametric function definition
len Empty = 0
len (Cons _ xs) = 1 + len xs

first :: List a -> a
first (Cons x _) = x
```

Parametric Lists Continued

```
data List a = Empty | Cons a (List a)
```

- function definitions can enforce certain class assertions
  - example: replace all occurrences of `x` by `y` in a list

    ```
    replace :: Eq a => a -> a -> List a -> List a
    replace _ _ Empty = Empty
    replace x y (Cons z zs) = rHelper (x == z) y z (replace x y zs)
    rHelper True  y _ xs = Cons y xs
    rHelper False _ z xs = Cons z xs
    ```

  - class assertion `Eq a =>` is required since list elements are compared via `==`
- function definitions can enforce a concrete element type
  - example: replace all occurrences of `'A'` by `'B'` in a list

    ```
    replaceAB :: List Char -> List Char
    replaceAB xs = replace 'A' 'B' xs
    ```

  - important: since `replace` asserts class `Eq a`, and this `a` is instantiated by `Char` in `replaceAB`, it is checked that `Char` indeed is in type class `Eq`

Lists in Haskell

- the list type from previous three slides is actually predefined in Haskell
- only difference: names
  - instead of `List a` one writes `[a]`
  - instead of `Empty` one writes `[]`
  - instead of `Cons x xs` one writes `x : xs` (and `:` is called "Cons")
  - in total

    ```
    data [a] = [] | a : [a]
    ```
- list constructor `(:)` associates to the right:
  `1 : 2 : 3 : [] = 1 : (2 : (3 : []))`
- special list syntax for finite lists: `[1, 2, 3] = 1 : 2 : 3 : []`
- example: append on Haskell lists

  ```
  append :: [a] -> [a] -> [a]
  append [] ys       = ys
  append (x : xs) ys = x : append xs ys
  ```

Tuples

- tuples are a frequently used datatype to provide several outputs at once; example: a division-with-remainder function should return two numbers, the quotient and the remainder
- it is easy to define various tuples in Haskell

```haskell
data Unit = Unit                  -- tuple with 0 entries
data Pair a b = Pair a b          -- tuple with 2 entries
data Triple a b c = Triple a b c  -- tuple with 3 entries
```

- example: find value of key `'y'` in list of key/value-pairs

```haskell
findY :: [Pair Char a] -> a
findY []               = error "cannot find y"
findY (Pair 'y' v : _) = v
findY (_ : xs)         = findY xs
```

remark: one would usually define a function to search for arbitrary keys

Tuples in Haskell

- tuples are predefined in Haskell (so there is no need to define `Pair`, `Triple`, `...`)
- for every $n \neq 1$ Haskell provides:
  - a type constructor `( , ..., )` (with n entries)
  - a (value) constructor `( , ..., )` (with n entries)
- examples
  - `Pair a b` and `Triple a b c` are equivalent to `(a, b)` and `(a, b, c)`
  - `(5, True, "foo") :: (Int, Bool, String)`
  - `() :: ()`
  - `(5)` is just the number `5`, so no 1-tuple
  - `(1, 2, 3)` is neither the same as `((1, 2), 3)` nor as `(1, (2, 3))`
- example program from previous slide using predefined tuples

```haskell
findY :: [(Char, a)] -> a
findY []             = error "cannot find y"
findY (('y', v) : _) = v
findY (_ : xs)       = findY xs
```

```haskell
data Maybe a = Nothing | Just a
```

- `Maybe` is predefined Haskell type to specify optional results
- example application: safe division without runtime errors

```haskell
divSafe :: Double -> Double -> Maybe Double
divSafe x 0 = Nothing
divSafe x y = Just (x / y)

data Expr = Plus Expr Expr | Div Expr Expr | Number Double

eval :: Expr -> Maybe Double
eval (Number x) = Just x
eval (Plus x y) = plusMaybe (eval x) (eval y)
eval (Div x y)  = divMaybe (eval x) (eval y)

plusMaybe (Just x) (Just y) = Just (x + y)
plusMaybe _ _               = Nothing

divMaybe (Just x) (Just y) = divSafe x y
divMaybe _ _               = Nothing
```

```haskell
data Either a b = Left a | Right b
```

- `Either` is predefined Haskell type for specifying alternative results
- example application: model optional values with error messages

```haskell
divSafe :: Double -> Double -> Either String Double
divSafe x 0 = Left ("don't divide " ++ show x ++ " by 0")
divSafe x y = Right (x / y)

data Expr = Plus Expr Expr | Div Expr Expr | Number Double

eval :: Expr -> Either String Double
eval (Number x) = Right x
eval (Plus x y) = plusEither (eval x) (eval y)
eval (Div x y)  = divEither (eval x) (eval y)

divEither (Right x) (Right y) = divSafe x y
divEither e@(Left _) _        = e     -- new case analysis required
divEither _ e                 = e

plusEither ... = ...
```

Summary

- usage of type variables and parametric polymorphism
  - datatypes with type variables
  - polymorphic functions, potentially include class assertions
    (example: `f :: (Eq a, Show b) => a -> Bool -> a -> b -> String`, ...)
- predefined datatypes
  - lists `[a]`
  - tuples `(..,..,..)`
  - option type `Maybe a`
  - sum type `Either a b`
- 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`