universität innsbruck

WS 2023/2024

Functional Programming

Week 9 – Generic Fold, Scope, Modules

René Thiemann James Fox Lukas Hofbauer Christian Sternagel Tobias Niederbrunner

Department of Computer Science

Last Lecture – Library Functions

```
foldr :: (a -> b -> b) -> b -> [a] -> b -- also: foldr1, foldl

take, drop :: Int -> [a] -> [a]
splitAt :: Int -> [a] -> ([a], [a])

takeWhile, dropWhile :: (a -> Bool) -> [a] -> [a]
span :: (a -> Bool) -> [a] -> ([a], [a])

zipWith :: (a -> b -> c) -> [a] -> [b] -> [c]
zip :: [a] -> [b] -> [(a, b)]
unzip :: [(a, b)] -> ([a], [b])

words, lines :: String -> [String]
unwords, unlines :: [String] -> String

concatMap :: (a -> [b]) -> [a] -> [b]

($) :: (a -> b) -> a -> b
```

Last Lecture – List Comprehension

- list comprehension
  - shape: `[ (x,y,z) | x <- [1..n], let y = x ^ 2, y > 100, Just z <- f y]`
  - consists of guards, generators, local declarations
  - translated via `concatMap`
- examples

```
prime n = n >= 2 && null [ x | x <- [2 .. n - 1], n `mod` x == 0]

ptriples n = [ (x,y,z) |
  x <- [1..n], y <- [x..n], z <- [y..n], x^2 + y^2 == z^2]
```

Further Example Applications: Sorting and Removing Duplicates

- example for list comprehension: quicksort

```
qsort [] = []
qsort (x : xs) =
   qsort [y | y <- xs, y < x] ++ [x] ++ qsort [y | y <- xs, y >= x]
```

- example for fold and list comprehension: removing duplicates of a list

```
remdups = foldr (\ x xs -> [x | not $ x `elem` xs] ++ xs) []
```

Fold on Arbitrary Datatypes

Fold on Arbitrary Datatypes

- recall `foldr f e`
  - main idea: replace `[]` by `e` and every `(:)` by `f`
  - generalize the idea of a fold to arbitrary datatypes

fold replaces every n-ary constructor with a user-provided n-ary function

- examples

```
foldMaybe :: (a -> b) -> b -> Maybe a -> b
foldMaybe f e (Just x) = f x
foldMaybe f e Nothing = e

foldEither :: (a -> c) -> (b -> c) -> Either a b -> c
foldEither f g (Left x)  = f x
foldEither f g (Right y) = g y
```

Example: Fold on Arithmetic Expressions

```
data Expr v a = Number a | Var v | Plus (Expr v a) (Expr v a)

foldExpr :: (a -> b) -> (v -> b) -> (b -> b -> b) -> Expr v a -> b
foldExpr fn _ _ (Number x) = fn x
foldExpr _ fv _ (Var v) = fv v
foldExpr fn fv fp (Plus e1 e2) = fp (foldExpr fn fv fp e1) (foldExpr fn fv fp e2)

eval :: Num a => (v -> a) -> Expr v a -> a
eval v = foldExpr id v (+)

variables :: Expr v a -> [v]
variables = foldExpr (const []) (\ v -> [v]) (++)   -- const x = \ _ -> x

substitute :: (v -> Expr w a) -> Expr v a -> Expr w a
substitute s = foldExpr Number s Plus

renameVars :: (v -> w) -> Expr v a -> Expr w a
renameVars r = substitute (Var . r)

countAdditions :: Expr v a -> Int
countAdditions = foldExpr (const 0) (const 0) ((+) . (+1))
```

Summary on Fold

- a fold-function can be defined for most datatypes

  fold replaces constructors by functions

- after having programmed fold for an individual datatype, one can define many recursive algorithms just by suitable invocations of fold

Scope

Scope

- consider program (1 compile error)

```
radius = 15
area radius = pi^2 * radius

squares x = [ x^2 | x <- [0 .. x]]

length [] = 0
length (_:xs) = 1 + length xs

data Rat = Rat Integer Integer
createRat n d = normalize $ Rat n d where normalize ... = ...
```

- scope
  - need rules to resolve ambiguities
  - scope defines which names of variables, functions, types, ... are visible at a given program position
  - control scope to structure larger programs (imports / exports)

Scope of Names

```
radius = 15
area radius = pi^2 * radius
```

- in the following we assume that `name_i` in the real code is always just `name` and the `_i` is used for addressing the different occurrences of `name`
- renamed Haskell program
  ```
  radius_1 = 15
  area_1 radius_2 = pi_1^2 * radius_3
  ```
- scope of names in right-hand sides of equations
  - is `radius_3` referring to `radius_2` or `radius_1`?
  - what is `pi_1` referring to?
- rule of thumb for searching `name`: search inside-out
  - think of abstract syntax tree of expression
  - whenever you pass a **`let`**, **`where`**, **`case`**, or function definition where `name` is bound, then refer to that local name
  - if nothing is found, then search global function `name`, also in Prelude
- `radius_3` refers to `radius_2`, `pi_1` to `Prelude.pi`

Local Names in Case-Expressions

- general case: `case expr of { pat1 -> expr1; ...; patN -> exprN }`
  - each `patI` binds the variables that occur in `patI`
  - these variables can be used in `exprI`
  - the newly bound variables of `patI` bind stronger than any previously bound variables
- example Haskell expression

```
case xs_1 of                -- renamed Haskell expression
  [] -> xs_2
  (x_1 : xs_3) -> case xs_4 ++ ys_1 of
    [] -> ys_2
    (x_2 : xs_5) -> x_3 : xs_6 ++ ys_3
```

  - `x_3` refers to `x_2` (since `x_2` is further inside than `x_1`)
  - `xs_6` refers to `xs_5` (since `xs_5` is further inside than `xs_3`)
  - `xs_4` refers to `xs_3`
  - `xs_1`, `xs_2`, `ys_1`, `ys_2`, and `ys_3` are not bound in this expression (the proper references need to be determined further outside)

Local Names in Let-Expressions

```
let {
  pat1 = expr1; ...; patN = exprN;
  f1 pats1 = fexpr1; ...; fM patsM = fexprM
} in expr
```

  - all variables in `pat1` ... `patN` and all names `f1` ... `fM` are bound
  - these can be used in `expr`, in each `exprI` and in each `fexprJ`
  - variables of `patsJ` bind strongest, but only in `fexprJ`

- ```
  let (x_1, y_1) = (y_2 + 1, 5)      -- renamed Haskell expression
      f_1 x_2 = x_3 + g_1 y_3 id_1
      g_2 y_4 f_2 = f_3 $ g_3 x_4 f_4
  in (f_5, g_4, x_5, y_5)
  ```
  - `y_2`, `y_3` and `y_5` refer to `y_1`
  - `x_3` refers to `x_2` since `x_2` binds stronger than `x_1`
  - `x_4` and `x_5` refer to `x_1`
  - `f_3` and `f_4` refer to `f_2` since `f_2` binds stronger than `f_1`
  - `g_1`, `g_3` and `g_4` refer to `g_2`
  - `f_5` refers to `f_1`
  - `id_1` is not bound in this expression
  ```

Global Function Definitions

- general case:
  `fname pats = expr`
  - all variables in `pats` are bound locally and can be used in `expr`
  - `fname` is not locally bound, but added to global lookup table
  - all variables/names in `expr` without local reference will be looked up in global lookup table
  - lookup in global table does not permit ambiguities
- ```
  radius_1 = 15                    -- renamed Haskell program
  area_2 radius_2 = pi_1^2 * radius_3
  length_1 [] = 0
  length_2 (_:xs_1) = 1 + length_3 xs_2
  ```
  - `radius_1`, `area_2` and `length_1/2` are stored in global lookup table
  - global lookup table has ambiguity: `length_1/2` vs. `Prelude.length`
  - `pi_1` is not locally bound and therefore refers to `Prelude.pi`
  - `radius_3` refers to local `radius_2` and not to global `radius_1`
  - `xs_2` refers to `xs_1`
  - `length_3` is not locally bound and because of mentioned ambiguity, this leads to a compile error
  ```

Global vs. Local Definitions

```haskell
length :: [a] -> Int

-- choose definition 1,
length = foldr (const (1 +)) 0

-- definition 2,
length =
  let { length [] = 0; length (x : xs) = 1 + length xs }
  in length

-- or definition 3
length [] = 0
length (_ : xs) = 1 + length xs
```

- definitions 1 and 2 compile since there is no `length` in the rhs that needs a global lookup
- in contrast, definition 3 does not compile
- still definitions 1 and 2 result in ambiguities in global lookup table
  → study Haskell's module system

Modules

Modules

- so far
  - Haskell program is a single file, consisting of several definitions
  - all global definitions are visible to user

```haskell
-- functions on rational numbers
data Rat = Rat Integer Integer         -- internal definition of datatype
normalize (Rat n d) = ...              -- internal function
createRat n d = normalize $ Rat n d    -- function for external usage
...
-- application: approximate pi to a certain precision
piApprox :: Integer -> Rat
piApprox p = ...
```

- motivation for modules
  - structure programs into smaller reusable parts without copying
  - distinguish between internal and external definitions
    - clear interface for users of modules
    - maintain invariants
    - improve maintainability

Modules in Haskell

```
-- first line of file ModuleName.hs
module ModuleName(exportList) where
-- standard Haskell type and function definitions
```

- each `ModuleName` has to start with uppercase letter
- each module is usually stored in separate file `ModuleName.hs`
- if Haskell file contains no `module` declaration, ghci inserts module name `Main`
- `exportList` is comma-separated list of function-names and type-names, these functions and types will be accessible for users of the module
- if `(exportList)` is omitted, then everything is exported
- for types there are different export possibilities
  - `module Name(Type)` exports `Type`, but no constructors of `Type`
  - `module Name(Type(..))` exports `Type` and its constructors

Example: Rational Numbers

```haskell
module Rat(Rat, createRat, numerator, denominator) where
data Rat = Rat Integer Integer
normalize = ...
createRat n d = normalize $ Rat n d
numerator (Rat n d) = n
...
instance Num Rat where ...
instance Show Rat where ...
```

- external users know that a type `Rat` exists
- they only see functions `createRat`, `numerator` and `denominator`
- they don't have access to constructor `Rat` and therefore cannot form expressions like `Rat 2 4` which break invariant of cancelled fractions
- they can perform calculations with rational numbers since they have access to `(+)` of class `Num`, etc., in particular for the instance `Rat`
- for the same reason, they can display rational numbers via `show`

Example: Rational Numbers – Improved Implementation

since external users cannot form expressions likes `Rat 2 4`, we may assume that only normalized rational numbers appear as input, provided that our implementation in this module obeys the invariant

```
module Rat(Rat, createRat, numerator, denominator) where
data Rat = Rat Integer Integer
  deriving Eq -- sound because of invariant

instance Show Rat where -- no normalization required
  show (Rat n d) = if d == 1 then show n else show n ++ "/" ++ show d

normalize = ...
createRat n d = normalize $ Rat n d

instance Num Rat where
  -- for negation no further normalization required
  negate (Rat n d) = Rat (- n) d

  -- multiplication requires normalization to obey invariant
  Rat n1 d1 * Rat n2 d2 = createRat (n1 * n2) (d1 * d2)
```

Example: Application

```haskell
module PiApprox(piApprox, Rat) where
-- Prelude is implicitly imported

-- import everything that is exported by module Rat
import Rat

-- or only import certain parts
import Rat(Rat, createRat)

-- import declarations must be before other definitions
piApprox :: Integer -> Rat
piApprox n = let initApprox = createRat 314 100 in ...
```

- there can be multiple **import** declarations
- what is imported is not automatically exported
  - when importing `PiApprox`, type `Rat` is visible, but `createRat` is not
  - if application requires both `Rat` and `PiApprox`, import both modules:

    ```haskell
    import PiApprox
    import Rat
    ```

Resolving Ambiguities

```haskell
-- Foo.hs
module Foo where pi = 3.1415

-- Problem.hs
module Problem where

import Foo

pi = 3.1415
area r = pi * r^2
```

- problem: what is `pi` in definition of `area`? (global name)
- lookup map is ambiguous: `pi` defined in `Prelude`, `Foo`, and `Problem`
- ambiguity persists, even if definition is identical
- solution via qualifier: disambiguate by using `ModuleName.name` instead of `name`
  - write `area r = Problem.pi * r^2` in `Problem.hs`
    (or `area r = Prelude.pi * r^2`)

Qualified Imports

```
module Foo where pi = 3.1415
module SomeLongModuleName where fun x = x + x

module ExampleQualifiedImports where

-- all imports of Foo have to use qualifier
import qualified Foo
-- result: no ambiguity on unqualified "pi"

import qualified SomeLongModuleName as S
-- "as"-syntax changes name of qualifier

area r = pi * r^2
myfun x = S.fun (x * x)
```

Summary

Summary

- scoping rules determine visibility of function names and variable names
- larger programs can be structured in modules
  - explicit export-lists to distinguish internal and external parts
  - advantage: changes of internal parts of module `M` are possible without having to change code that imports `M`, as long as exported functions of `M` have same names and types
  - if no module name is given: `Main` is used as module name
  - further information on modules
    https://www.haskell.org/onlinereport/modules.html