



Funktionale Programmierung

Woche 9 – Generisches Fold, Sichtbarkeit, Module

René Thiemann

Philipp Dablander Joshua Ocker Michael Schaper Lilly Schönherr Adam Pescoller

Institut für Informatik

Letzte Vorlesung – Predule Funktionen

`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`

RT et al. (IFI @ UIBK)

Woche 9

2/24

Letzte Vorlesung – List Comprehensions

- List Comprehension

- Gestalt: `[(x,y,z) | x <- [1..n], let y = x ^ 2, y > 100, Just z <- f y]`
- besteht aus Guards, Generatoren und lokalen Definitionen
- Übersetzung mittels `concatMap`

- Beispiele

`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]`

Weitere Beispiel-Anwendungen: Sortieren und Duplikat-Entfernung

- Beispiel für List Comprehension: Quicksort

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

- Beispiel für `foldr` und List Comprehension: entfernen von Duplikaten

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

Fold für beliebige Datentypen

Fold für beliebige Datentypen

- betrachten Sie `foldr f e`
 - Idee: ersetze `[]` durch `e` und jedes `(:)` durch `f`
 - generalisiere diese Idee für ein `fold` auf beliebigen Datentypen
 - `fold` ersetzt jeden n -stelligen Konstruktor durch eine n -stellige Funktion
- Beispiele (`foldMaybe` heißt `maybe` in Prelude und `foldEither` heißt `either`)

```
foldMaybe :: b -> (a -> b) -> Maybe a -> b
foldMaybe e f (Just x) = f x
foldMaybe e f 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
```

Beispiel: Fold für arithmetische Ausdrücke

```
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) (\ n m -> n + m + 1)
```

Zusammenfassung zu Fold

- eine `fold`-Funktion kann für viele Datentypen definiert werden
 - `fold` ersetzt Konstruktoren durch Funktionen
- nachdem `fold` für einen Datentyp definiert wurde, können viele rekursive Algorithmen durch Aufrufe von `fold` ersetzt werden
- neben dem hier vorgestellten `fold` gibt es auch die Einschränkung auf binäre `fold`s, bei denen man alle Elemente (in einer Liste, einem Baum, etc.) binär verknüpft; Details: siehe Übungsblatt 8 und Typklasse `Foldable`

Beispiel: der Typ von `foldr` ist nicht auf Listen beschränkt

```
foldr :: Foldable t => (a -> b -> b) -> b -> t a -> b
foldr :: (a -> b -> b) -> b -> [a] -> b
foldr :: (a -> b -> b) -> b -> Maybe a -> b
foldr :: (a -> b -> b) -> b -> Either a a -> b
```

Sichtbarkeit

Scope

- betrachten Sie folgendes Programm (1 Compiler-Fehler)

```
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 ... = ...
```

- Sichtbarkeit
 - es benötigt Regeln, um Mehrdeutigkeit aufzulösen
 - die Sichtbarkeits-Regeln definieren, welche Namen von Variablen, Funktionen, Typen, ... an welcher Programm-Position sichtbar sind
 - Sichtbarkeit kann kontrolliert werden, um größere Programme besser zu strukturieren (Import und Export, interne und externe Funktionalität)

Sichtbarkeit von Namen (von Variablen, Funktionen, ...)

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

- wir nehmen an, dass `name_i` im wirklichen Programm nur `name` ist; das zusätzliche `_i` wird in den Folien benutzt, um unterschiedliche Vorkommen von `name` zu unterscheiden
- umbenanntes Haskell-Programm

```
radius_1 = 15
area_1 radius_2 = pi_1^2 * radius_3
```
- Sichtbarkeit und Bezug von Namen in rechten Seiten von definierenden Gleichungen
 - bezieht sich `radius_3` auf `radius_2` oder `radius_1`?
 - worauf bezieht sich `pi_1`?
- Daumenregel für die Suche nach dem Bezug für `name`: man sucht **von innen nach außen**
 - stellen Sie sich den abstrakten Syntax Baum eines Haskell-Ausdrucks vor
 - immer wenn man ein `let`, `where`, `case`, oder eine Funktions-Definition sieht, in der `name` **gebunden** wird, dann ist der Bezug dieser **lokale** Name
 - wenn nichts gefunden wird, dann suche nach einer **globalen** Funktion `name`, auch in Prelude
- Beispiel: `radius_3` bezieht sich auf `radius_2`, `pi_1` auf `Prelude.pi`

Lokale Namen in Case-Ausdrücken

- betrachte `case expr of { pat1 -> expr1; ...; patN -> exprN }`
 - jedes `patI` bindet die Variablen, die in `patI` vorkommen
 - diese Variablen können in `exprI` genutzt werden
 - diese neu gebundenen Variablen in `patI` binden stärker als alle zuvor gebundenen Variablen
- Beispiel Haskell Ausdruck

```
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` bezieht sich auf `x_2` (da `x_2` weiter innen ist als `x_1`)
 - `xs_6` bezieht sich auf `xs_5` (da `xs_5` weiter innen ist als `xs_3`)
 - `xs_4` bezieht sich auf `xs_3`
 - `xs_1`, `xs_2`, `ys_1`, `ys_2`, und `ys_3` sind in diesem Ausdruck nicht gebunden (Bezüge müssen weiter außen hergestellt werden)

Lokale Namen in Let-Ausdrücken

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

- alle Variablen in `pat1 ... patN` und alle Namen `f1 ... fM` sind gebunden
- diese können in `expr`, in `expr1` und in jedem `fexprJ` genutzt werden
- Variablen in `patsJ` binden am stärksten, aber nur 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` und `y_5` beziehen sich auf `y_1`
 - `x_3` bezieht sich auf `x_2` weil `x_2` stärker bindet als `x_1`
 - `x_4` und `x_5` beziehen sich auf `x_1`
 - `f_3` und `f_4` beziehen sich auf `f_2` da `f_2` stärker bindet als `f_1`
 - `g_1`, `g_3` und `g_4` beziehen sich auf `g_2`
 - `f_5` bezieht sich auf `f_1`
 - `id_1` ist nicht in diesem Ausdruck gebunden

Globale Funktions-Definitionen

- betrachte definierende Gleichung `fname pats = expr`
 - alle Variables in `pats` sind lokal gebunden und können in `expr` genutzt werden
 - `fname` ist **nicht lokal** gebunden, wird aber in eine **globale Namens-Tabelle** eingetragen
 - alle Namen in `expr` ohne lokalen Bezug werden in der globalen Namens-Tabelle gesucht
 - die Suche in der globalen Tabelle muss ein **eindeutiges** Ergebnis liefern
- `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` und `length_1/2` werden in globaler Tabelle gespeichert
 - globale Tabelle hat Mehrdeutigkeit: `length_1/2` vs. `Prelude.length`
 - `pi_1` ist nicht lokal gebunden und hat daher den Bezug zu `Prelude.pi`
 - `radius_3` bezieht sich lokal auf `radius_2` und nicht global auf `radius_1`
 - `xs_2` bezieht sich auf `xs_1`
 - `length_3` ist nicht lokal gebunden, und führt wegen der Mehrdeutigkeit zu einem Compiler-Fehler

Globale vs. Lokale Definition

```
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
```

- Definitionen 1 und 2 compilieren, weil sich kein `length` in einer rechten Seite befinden, bei dem Suche in globaler Tabelle erforderlich ist
- Definition 3 compiliert nicht
- Definitionen 1 und 2 führen allerdings zu Mehrdeutigkeiten in der globalen Tabelle
 → betrachte Haskell's Modul-System

Module

Module

- bislang
 - ein Haskell Programm besteht aus **einer** Datei, die mehrere Definitionen enthalten kann
 - alle globalen Definitionen sind für AnwenderInnen sichtbar
- ```
-- 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 für Module
 - strukturiere Programme in kleinere **wiederverwendbare** Teile, ohne dabei Kopien zu erstellen
 - unterscheide zwischen **internen und externen** Definitionen
 - klare Schnittstelle für AnwenderInnen des Moduls
 - stelle Invarianten sicher
 - verbessere Wartbarkeit der Programme

Beispiel: Rationale Zahlen

```
module Rat(Rat, createRat, numerator, denominator) where
data Rat = Rat Integer Integer
normalize = ... -- cancels fractions: 2 / 4 -> 1 / 2
createRat n d = normalize $ Rat n d
numerator (Rat n d) = n
...
instance Num Rat where ...
instance Show Rat where ...
```

- externe AnwenderInnen wissen, dass ein Typ `Rat` existiert
- sie sehen Funktionen `createRat`, `numerator` and `denominator`
- sie haben keinen Zugriff auf Konstruktor `Rat` und können daher nicht Ausdrücke wie `Rat 2 4` bilden, die eine Invariante verletzen, z.B. gekürzte Brüche
- sie können Berechnungen mit rationalen Zahlen durchführen, da Sie Zugriff auf `(+)` der Klasse `Num` haben, insbesondere auch für die `Num`-Instanz `Rat`
- ebenso können sie rationale Zahlen mittels `show` anzeigen

Module in Haskell

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

- jeder `ModuleName` beginnt mit einem Großbuchstaben
- jedes Modul wird normalerweise in Datei `ModuleName.hs` gespeichert
- wenn eine Haskell-Datei keine `module` Deklaration enthält, fügt ghci `module Main where` zu Beginn ein
- `exportList` ist eine Komma-getrennte Liste von Funktions-Namen und Typ-Namen, nur diese Funktionen und Typen werden für AnwenderInnen des Moduls sichtbar
- wenn `(exportList)` fehlt, dann wird alles exportiert
- für Typen gibt es unterschiedliche Grade des Exports
 - `module Name(Type)` exportiert den Typ `Type`, aber nicht die Konstruktoren von `Type`
 - `module Name(Type(..))` exportiert Typ `Type` und die Konstruktoren

Beispiel: Rationale Zahlen – Verbesserte Implementierung

da externe Nutzer kein Ausdrücke wie `Rat 2 4` bilden können, können wir davon ausgehen, dass alle Funktionen nur normalisierte rationale Zahlen entgegennehmen, unter der Annahme, dass die eigene Implementierung in diesem Modul nur normalisierte Brüche generiert

```
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)
```

Beispiel Anwendung

```
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 ...
```

- es kann mehrere `import` Deklarationen geben
- importierte Namen werden nicht automatisch exportiert
 - wenn man `PiApprox` importiert, wird `Rat` sichtbar, aber nicht `createRat`
 - um `Rat` und `PiApprox` zu verwenden, müssen beide Module importiert werden:

```
import PiApprox
import Rat
```

Umgang mit Mehrdeutigkeiten

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

```
-- Problem.hs
module Problem where
```

```
import Foo
```

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

- Problem: Worauf bezieht sich `pi` in Definition von `area`? (globaler Name)
- globale Tabelle ist mehrdeutig: `pi` ist definiert in `Prelude`, `Foo`, und in `Problem`
- Mehrdeutigkeit bleibt, auch wenn die Definitionen identisch sind
- eine Lösung wird durch **Qualifier** geboten: Mehrdeutigkeit manuell auflösen durch Verwendung von `ModuleName.name` anstelle von `name`
 - definiere `area r = Problem.pi * r^2` in `Problem.hs`
(oder `area r = Prelude.pi * r^2`)

Qualifizierte Importe

```
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)
```

Zusammenfassung

- Sichtbarkeits-Regeln klären den Bezug von Funktions- und Variablen-Namen
- größere Programme können in **Module** strukturiert werden
 - explizite **export-Listen** dienen der Unterscheidung von interner und externer Funktionalität
 - Vorteil: Änderungen der internen Teile eines Moduls `M` sind möglich, ohne dabei den Code anpassen zu müssen, der `M` importiert, solange die exportierten Funktionen die gleichen Namen und Typen haben
 - wenn kein Modul angegeben wird, wird `Main` als Modul-Name genutzt
 - weitere Informationen zu Modulen
<https://www.haskell.org/onlinereport/haskell2010/haskellch5.html>