universität innsbruck

WS 2026/2027



Funktionale Programming

Woche 5 – Ausdrücke, Rekursion mit Zahlen

René Thiemann
Maria Fritzler Adam Pescoller Michael Schaper Lilly Schönherr

Institut für Informatik

Letzte Vorlesung

- Typvariablen: `a`, `b`, ... repräsentieren jeden Typen
- parametrischer Polymorphismus
  - eine Implementierung, die für verschiedene Typen genutzt werden kann
  - polymorphe Datentypen, z.B., `data List a = Empty | Cons a (List a)`
  - polymorphe Funktionen, z.B., `append :: List a -> List a -> List a`
  - Typklassen-Einschränkungen, z.B., `sumList :: Num a => List a -> a`
- vordefinierte Typen: `[a]`, `Maybe a`, `Either a b`, `(a1,...,aN)`
- vordefinierte Typklassen
  - Arithmetik ohne Division: `Num a`
  - Arithmetik mit Division: `Fractional a`
  - Gleichheit zwischen Elementen: `Eq a`
  - Kleiner- und Größer-Vergleiche: `Ord a`
  - Konvertierung zu `Strings`: `Show a`

Diese Vorlesung

- Typsynonyme
- neue Arten von Haskell-Ausdrücken
- Rekursion mit Zahlen

Typsynonyme

Typsynonyme

- Haskell bietet es an, Typsynonyme mittels Schlüsselwort `type` zu definieren
  `type TConstr a1 ... aN = ty`
  - `TConstr` ist ein neuer Name für einen Typkonstruktor
  - `a1 ... aN` ist eine Liste von Typvariablen
  - `ty` ist ein Typ, der Typvariablen `a1 ... aN` enthalten darf, aber keine anderen
  - `ty` darf nicht `TConstr` selbst enthalten, d.h., Rekursion ist nicht erlaubt
  - es gibt keine neuen (Wert-)Konstruktoren

Typsynonyme – Anwendungen, Strings

- Beispiel Anwendungen von Typsynonymen
  - nutze bekannte statt neue Datentypen: `type PersonTS = (String,Date)`
  - verbessere die Lesbarkeit von Programmen

```
type Month = Int
type Day   = Int
type Year  = Integer
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 synonyms help to make the code more readable

createDate :: Int -> Int -> Integer -> (Int, Int, Integer)
createDate x y z = (x, y, z)
```

- in Haskell: `type String = [Char]`
  - insbesondere ist `"hello"` das gleiche wie `['h', 'e', 'l', 'l', 'o']`
  - Listen-Funktionen sind auch `String`-Funktionen, z.B. `(++) :: [a] -> [a] -> [a]`

Vergleich von Typsynonymen und Datentypen

- Typsynonyme können immer auch alternativ als neuer Datentyp definiert werden
- Beispiel-Kodierung von Personen als Name und Geburtstag

```
type PersonTS = (String, Date)          -- pair of name and date
data PersonDT = Person (String, Date)   -- just add constructor Person
```

- Anmerkung: `PersonTS` und `PersonDT` sind unterschiedliche Typen
  - Typen `PersonTS` und `(String, Date)` sind identisch
  - Typ `PersonDT` unterscheidet sich von `(String, Date)` und `PersonTS`
  - `("Bob", (1,3,2002))` hat Typ `PersonTS`, aber nicht Typ `PersonDT`
  - `Person ("Bob", (1,3,2002))` hat Typ `PersonDT`, aber nicht Typ `PersonTS`
- Vorteile von Typsynonymen
  - zusätzliche Konstruktoren wie `Person` sind nicht erforderlich
  - Funktionen auf existierenden Typen können direkt genutzt werden, z.B. `fst`, um den Namen zu erhalten, anstatt separater Implementierung für `PersonDT`: `name (Person p) = fst p`
- Vorteile von Datentypen
  - separate Typklassen Instanzen sind möglich, z.B. eine eigene `show`-Funktion (Woche 6)
  - es gibt die Möglichkeit, die interne Repräsentation zu verbergen (Woche 9)

Neue Arten von Haskell-Ausdrücken

Wiederholung: Funktions-Definitionen

- momentane Form von Funktions-Definitionen

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

  wobei die Ausdrücke `exprI` aus Literalen, Variablen und Funktions- und Konstruktor-Anwendungen bestehen

- Beobachtung
  - Fallunterscheidung ist nur mittels Pattern auf linken Seiten der definierenden Gleichungen möglich
  - Fallunterscheidung in rechten Seiten wäre wünschenswert
  - Behelfslösung mit Hilfsfunktionen ist möglich
  - bessere Lösung: Erweiterung der erlaubten Haskell-Ausdrücke

if-then-else

- primitivste Form der Fallunterscheidung: if-then-else
- Funktionalität: liefere einen von zwei Werten, abhängig von einem Boolschen Wert

```
ite :: Bool -> a -> a -> a
ite True  x _ = x
ite False _ y = y
```

- Beispiel: Berechnung des Betrags einer Zahl

```
absoluteValue :: (Ord a, Num a) => a -> a
absoluteValue x = ite (x >= 0) x (- x)
```

- if-then-else ist vordefiniert: `if ... then ... else ...`

```
absoluteValue x = if x >= 0 then x else - x
```

- in Haskell gibt es kein if-then (ohne else):
  was ist Resultat, wenn der Boolesche Wert `False` ist?
- Anmerkung: `absoluteValue` ist als Funktion `abs` in Haskell vordefiniert, d.h. Teil der Prelude Bibliothek;
  Inhalt der Prelude (Funktionen, Konstruktoren, Typklassen, ...) wird in grün dargestellt

Fallunterscheidung mit Pattern Matching

- Beobachtung: oft ist eine Fallunterscheidung bzgl. berechneter Werte erforderlich
- Implementierung ist möglich mittels Hilfsfunktionen
- Beispiel aus Woche 3: Begrüßung einer Person

```
ageYear (Person (_, (16, 11, y))) = Just (2026 - y)
ageYear _  = Nothing

greetingOld p@(Person (name, _)) = gHelper name (ageYear p)
gHelper n Nothing = "Hello " ++ n
gHelper n (Just a) = "Hi " ++ n ++ ", you turned " ++ show a
```

- Nachteile
  - lokale Werte müssen als Argumente an Hilfsfunktionen übergeben werden (hier: `name`)
  - Verschmutzung des Namenraums durch Hilfsfunktionen (`aux`, `aux1`, `aux2`, `auX`, `helper`, `gHelper`, ...)
- Anmerkungen: if-then-else ist nicht ausreichend für obiges Beispiel

Case-Ausdrücke

- Case-Ausdrücke unterstützen Pattern-Matching in rechten Seiten

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

  - matchte `expr` beginnend mit `pat1` bis zu `patN` von oben nach unten
  - wenn `patI` das erste matchende Pattern ist, dann wertet der case-Ausdruck zu `exprI` aus

- Beispiel der vorigen Folie ohne Hilfsfunktion

```
greetingNew p@(Person (name, _)) = case ageYear p of
  Nothing -> "Hello " ++ name
  Just a -> "Hi " ++ name ++ ", you turned " ++ show a
```

Die Layout-Regel

- Problem: Definition von Gruppen (von Pattern, von Funktions-Definitionen, ...)
- Inhalt eines Haskell-Skripts ist eine Gruppe; verschachtelte Gruppen gibt es bei `where`, `let`, `do` und `of`
- Elemente, die in der gleichen Spalte beginnen, gehören der gleichen Gruppe an
- durch Einrückung kann ein Einzel-Element mehrere Zeilen umfassen
- eine Gruppe wird durch Verringerung der Einrückung beendet
- **ohne Layout:** umschließe Gruppen mit '{' und '}' und trenne Elemente mit ';'

Beispiele

mit Layout:

```
and b1 b2 = case b1 of
  True -> case b2 of
    True -> True
    False -> False
  False -> False
```

mit geschweiften Klammern (Layout egal):

```
and b1 b2 = case b1 of
  { True -> case b2 of
  { True -> True; False -> False };
    False -> False }
```

Leerraum in Haskell

- wegen der Layout-Regel ist Leerraum in Haskell wichtig
  (im Gegensatz zu vielen anderen Sprachen)
- benutzen Sie keine Tabulatoren in Haskell
  (Tabulator-Weite von Editor und Compiler kann sich unterscheiden)

Example

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

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

```
ghci> and1 True False
False

ghci> and2 True False
*** Exception: Non-exhaustive patterns in case
```

`let`-Ausdrücke

- `let`-Ausdrücke bieten die Möglichkeit lokaler Definitionen
- Syntax

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

- jeder `let`-Ausdruck kann mehrere Definitionen beinhalten
  (die Reihenfolge ist irrelevant)
- Definitionen resultieren in neuen Variablen-Bindungen und neuen Funktionen
  - diese können in jedem Ausdruck `expr` genutzt werden
  - es gibt keine Sichtbarkeit außerhalb des `let`-Ausdrucks

Beispiel: Anzahl der reellen Wurzeln mittels `let`

```
-- 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)
```

Lokale Definitionen mit `where`

- `where` ist ähnlich zu `let`, es wird für lokale Definitionen genutzt
- Syntax

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

- jedes `where` darf beliebig viele Definitionen enthalten (Reihenfolge egal)
- lokale Definitionen führen neue Variablen und Funktionen ein
  - diese dürfen in jedem Ausdruck `expr` genutzt werden
  - keine Sichtbarkeit außerhalb der definierenden Gleichung (oder des case-Ausdrucks)
- Anmerkung: im Unterschied zu `let` werden bei `where` die lokalen Definitionen nachträglich notiert

```
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
```

Bedingte Gleichungen (Guarded Equations)

- definierende Gleichungen einer Funktionsdefinition können Bedingungen haben
- Syntax:

```
fname pat1 ... patM
  | cond1 = expr1
  | cond2 = expr2
  | ...
  where ... -- optional where-block
```

  wobei jedes `condI` ein Boolscher Ausdruck ist
- wenn `condI` die erste Bedingung ist, die zu `True` auswertet, dann ist das Resultat `exprI`
- wenn keine Bedingung zutrifft, wird die nächste definierende Gleichung von `fname` genutzt

```
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
```

Beispiel: Nullstellen

- Aufgabe: berechne die Summe der Nullstellen eines quadratischen Polynoms
- Lösung mit möglichen Laufzeitfehlern

```
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
```

- Anmerkung: die Pattern im `let` sollten so sein, dass diese alle Fälle abdecken; ansonsten sollte man `case` anstelle von `let` verwenden

Beispiel: Nullstellen (Fortgesetzt)

- Aufgabe: berechne die Summe der Nullstellen eines quadratischen Polynoms
- Lösung mit explizitem Fehlschlag mittels `Maybe`-Typ

```
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                 -- cannot be replaced by n -> n! (types)
```

Rekursion mit Zahlen

Rekursion mit Zahlen

- rekursive Funktionen:

  ```
  f pat1 ... patN = ... (f expr1 ... exprN) ...
  ```

  wobei Eingabe Argumente größer als Argumente im rekursiven Aufruf sein sollten:

  ```
  (pat1, ..., patN) > (expr1, ..., exprN)   -- for some relation >
  ```

- Verkleinerung ist häufig in einem Argument gegeben (das i-te Argument wird kleiner)
- bislang war die Verkleinerung immer bzgl. der Baum-Größe
  - Liste wird kürzer
  - Anzahl der Konstruktoren in einem arithmetischen Ausdruck (`Expr`) wird kleiner
- Rekursion ist auch möglich, wenn das Argument eine Zahl ist (Baum-Größe = 1);
  Beispiel: der Wert der Zahl könnte kleiner werden
- häufige Fälle
  - Zahl i wird verkleinert, bis diese 0 erreicht (while $i \neq 0 \ldots i := i - 1$)
  - Zahl i wird erhöht, bis diese eine obere Schranke n erreicht (while $i < n \ldots i := i + 1$)

Beispiel: Fakultäts-Funktion

- mathematische Definition: $n! = n \cdot (n-1) \cdot \ldots \cdot 2 \cdot 1$, $0! = 1$
- Implementierung, die runter zählt

```
factorial :: Integer -> Integer
factorial 0 = 1
factorial n = n * factorial (n - 1)
```

  - in jedem rekursiven Aufruf wird der Wert von `n` verkleinert
  - `factorial n` terminiert nicht, falls `n` negativ ist (Ctrl-C im ghci unterbricht Berechnung)

- Implementierung, die hoch zählt; nutzt Akkumulator
  (hier: `r` speichert akkumuliertes (R)esultat)

```
factorial :: Integer -> Integer
factorial n = fact 1 1 where
  fact r i
    | i <= n = fact (i * r) (i + 1)
    | otherwise = r
```

  - in jedem rekursiven Aufruf wird `n - i` verringert
  - Implementierung ist äquivalent zu imperativem Programm (mit lokalen Variablen `r` und `i`)

Beispiel: Rekursion

- Rekursion auf Bäumen und auf Zahlen kann kombiniert werden
- Beispiel: berechne das n-te Element einer Liste

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

- Beispiel: nehme die ersten n Elemente einer Liste

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

- Anmerkungen
  - sowohl `take` als auch n-tes Element (`!!`) sind vordefiniert
  - `drop` is vordefinierte Funktion, welche die ersten n Elemente einer Liste entfernt
  - Zusammenhang: `take n xs ++ drop n xs == xs`

Beispiel: Erstelle Liste in einem Bereich

- Aufgabe: gegeben untere Grenze l und obere Grenze u, berechne Liste $[l, l+1, \ldots, u]$
- Algorithmus: inkrementiere l bis $l > u$ und füge jeweils l zu Beginn der Liste hinzu

```
range l u
  | l <= u = l : range (l + 1) u
  | otherwise = []
```

- Anmerkung: (eine verallgemeinerte Version von) `range l u` ist vordefiniert (`[l .. u]`)
- Beispiel: kompakte Definition der Fakultäts-Funktion
  - `factorial n = product [1 .. n]`
    wobei `product :: Num a => [a] -> a` das Produkt einer Liste von Zahlen berechnet

Zusammenfassung

- Typsynonyme mittels `type`
- neue Arten von Ausdrücken: lokale Definitionen und case-Ausdrücke
- Rekursion mit Zahlen