universität innsbruck

WS 2026/2027



Funktionale Programming

Woche 4 – Polymorphismus

René Thiemann
Maria Fritzler Adam Pescoller Michael Schaper Lilly Schönherr

Institut für Informatik

Letzte Vorlesung

- Funktions Definitionen mit Pattern Matching
  - mehrere definierende Gleichungen für eine Funktion
  - Gleichungen werden von oben nach unten zur Auswertung probiert
- Pattern
  - `x`, `_`, `CName pat1 ... patN`, `x@pat`
  - Variablen dürfen nicht mehrfach verwendet werden
  - Pattern beschreiben die Struktur einer möglichen Eingabe
- Rekursion
  - definierte Gleichung von Funktion `f` darf `f` in rechter Seite nutzen

    ```
    f pat1 ... patN = ... (f expr1 ... exprN) ...
    ```
  - Argumente in rekursivem Aufruf sollten kleiner als in linker Seite sein

Beispiel auf Listen

- Aufgabe 1: Verkettung zweier Listen, d.h., Verkettung von $[1,5]$ und $[3]$ liefert $[1,5,3]$
- Vorbereitung: Repräsentation von Listen in Haskell

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

- Lösung 1: Pattern Matching und Rekursion auf dem ersten Argument

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

Erklärung der zweiten Gleichung

- zuerst verkette die Listen `xs` und `ys` (`append xs ys`), dann füge `x` vorne hinzu

- Aufgabe 2: Berechne letztes Element einer Liste
- Lösung: betrachte 3 Fälle (Liste mit ≥ 2 Elementen, mit einem Element, leere Liste)

```
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"
```

Beispiel mit Datentypen `Expr` und `List`

- Datentyp für einfache mathematische Ausdrücke

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

- Aufgabe: erstelle Liste aller Zahlen, die im Ausdruck vorkommen
- Lösung

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

- Anmerkungen
  - rechte Seite der 1. Gleichung muss `Cons x Empty` sein, und nicht nur `x`:
    Resultat muss eine Liste von Zahlen sein, keine Zahl
  - `numbers` (und auch `append`) ist definiert mittels struktureller Rekursion:
    rufe die Funktion rekursiv auf für jedes rekursive Argument eines Datentyps
    (`e1` und `e2` für `Plus e1 e2`, und `e` für `Negate e`, aber nicht `x` für `Number x`)

Dekomposition and Hilfs-Funktionen

- während der Definition neuer Funktionen fehlt oft eine Funktionalität
- Beispiel-Aufgabe: definiere eine Funktion, um alle Duplikate in einer Liste zu entfernen
- Lösung:

```
remdups Empty = Empty
remdups (Cons x xs) = Cons x (remove x (remdups xs))
- Unteraufgabe:  "remove x xs" entfernt jedes x aus Liste 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
```

- Anmerkungen
  - Lösung nutzt strukturelle Rekursion: `remdups (Cons x xs)` ruft `remdups xs` auf
  - alternative Lösung mit nicht-struktureller Rekursion: ersetze 2. Gleichung durch

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

Parametrischer Polymorphismus

Limitation der Datentyp Definitionen

- Aufgabe: definiere Datentyp für Listen von Zahlen und Listenlängen-Funktion

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

- Aufgabe: definiere Datentyp für Listen von Zeichenketten und Funktion, um Listenlänge zu berechnen

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

- Beobachtungen
  - Datentypen und Funktionen sind nahezu identisch: einziger Unterschied:
    Typ der Elemente (`Integer`/`String`) and Namen (Typen / Konstruktoren / Funktionen)
  - eigene Listen-Kopie für jeden neuen Element-Typ ist nicht erstrebenswert
    - Definition der immer gleichen Funktionalität ist langweilig
    - Änderung der Implementierung ist Fehler-anfällig und arbeitsintensiv – Änderung jeder Kopie
  - Ziel: generischer Datentyp für Listen, sowie Funktionen, die für jede Liste arbeiten

Zwei Arten von Polymorphismus

- parametrischer Polymorphismus
  - Idee: erstelle eine Definition, die vielseitig benutzt werden kann
  - Beispiel
    - eine Datentyp Definition für beliebige Listen (Parameter: Typ der Elemente)
    - eine Datentyp Definition für beliebige Paare (Parameter: Typ 1. und 2. Komponente)
    - ...
    - eine Funktions-Definition, die auf parametrischen Listen, Paaren, ... arbeitet
      Beispiel: Länge, Verkettung zweier Listen, 1. Komponente eines Paars, ...
- ad-hoc Polymorphismus
  - Idee: biete ähnliche Funktionalität mit gleichem Namen für unterschiedliche Typen an
  - Beispiel
    - `(==)` ist der Gleichheits-Operator; unterschiedliche Implementierung für `String`, `Int`, ...
    - `(+)` ist Additions-Operator; unterschiedliche Implementierung für `Int`, `Float`, ...
    - `minBound` gibt kleinsten Wert für beschränkte Typen; unterschiedliche Implementierung für `Int`, `Char`, ...
  - Vorteil: einheitlicher Zugriff (== anstelle von `==Int`, `==String`, `==Double`)

Typvariablen

- Definition polymorpher Typen und Funktionen benötigt Typvariablen
- Typvariablen
  - beginnen mit Kleinbuchstaben; oft werden Einzelbuchstaben verwendet, z.B., `a`, `b`, ...
  - eine Typvariable repräsentiert jeden Typ
  - Typvariablen können durch (konkretere) Typen substituiert werden
- Typ `ty1` ist allgemeiner als `ty2`, falls `ty2` aus Typ `ty1` mittels Substitution entsteht
- wichtig: es ist erlaubt, allgemeine Typen durch spezielle zu ersetzen;
  wenn `expr :: ty1` und `ty1` allgemeiner als `ty2` ist, dann gilt auch `expr :: ty2`
- Typen `ty1` und `ty2` sind äquivalent wenn `ty1` allgemeiner als `ty2` ist und umgekehrt
- Beispiel
  - `a` ist allgemeiner als jeder andere Typ
  - `a -> b -> a` ist allgemeiner als `Int -> Char -> Int` (`a`/`Int`, `b`/`Char`), `a -> Bool -> a` (`a`/`a`, `b`/`Bool`), `c -> c -> c` (`a`/`c`, `b`/`c`)
  - `a -> b -> a` ist äquivalent zu `b -> a -> b`
  - `a -> b -> a` ist nicht allgemeiner als `a -> b -> c`
  - `someFun True x y = x` ist eine Funktion mit Typ `Bool -> b -> c -> b` (`True`: `a`, `x`: `b`, `y`: `c`, `x`: `d`, `Bool`: `a`/`Bool`, `b`: `d`/`b`)

Typen – Formal Definiert

- bislang: Definition von (einfachen) Haskell Ausdrücken und Pattern
- jetzt: Definition von Typen
- Vorbereitung: Typkonstruktoren (`TConstr`)
  - ähnlich zu (Wert-)Konstruktoren (`Cons`, `True`, ...)
    - beginnen mit Großbuchstaben
    - haben eine Stelligkeit
  - Unterschied zu Konstruktoren: Typkonstruktoren erstellen Typen
- Haskell Typen werden auf folgende Arten gebildet
  - `a` — Typvariable
  - `TConstr ty1 ... tyN` — Typkonstruktor mit Stelligkeit `N` angewandt auf `N` Typen
  - `(ty)` — Klammern sind möglich
- Beispiele (Typkonstrukt. der Stelligkeit 0: `Char`, `Bool`, `Integer`, ...; Stelligkeit 2: `->`)
  - `->` ohne die beiden Argumente ist kein Typ
  - `a -> Int` – Typ von Funktionen mit beliebiger Eingabe und Ausgabe vom Typ `Int`
  - `Bool -> (a -> Int)` – F. mit Eingabe `Bool` und Resultat ist Funktion mit Typ `a -> Int`
  - `Bool -> a -> Int` – gleich wie zuvor (!), `->` assoziiert nach rechts
  - `(Bool -> a) -> Int` – Eingabe ist Funktion mit Typ `Bool -> a`, Ausgabe ist `Int`

Typklassen-Einschränkungen und vordefinierte Typklassen

- manchmal muss eine Typvariable `a` eingeschränkt werden, zu einer Typklasse zu gehören
  - Typ `a`, auf dem `(+)`, `(-)`, `(*)` definiert ist: Typklasse `Num a`
  - Typ `a`, auf dem zudem `(/)` definiert ist: Typklasse `Fractional a`
  - Typ `a`, auf dem `(==)`, `(/=)` definiert ist: Typklasse `Eq a`
  - Typ `a`, auf dem zudem `(<)`, `(<=)`, ... definiert ist: Typklasse `Ord a`
  - Typ `a`, auf dem `show :: a -> String` definiert ist: Typklasse `Show a`
- diese Typklassen-Einschränkungen (type class assertions) werden mit `=>` notiert
- Beispiele

```
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
```

- Typ-Substitutionen müssen Typklassen-Einschränkungen beachten
  - `g False True` ist nicht erlaubt, da `Bool` keine Instanz von `Num` ist
  - `i (5 :: Int)` ist erlaubt, da `Int` Instanz von `Num` und `Show` ist

Datentypen mit parametrischem Polymorphismus

- vorige Definition

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

- neue Definition

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

  - neue Definition is allgemeiner: `K` kann auch 0 sein
  - `a1 ... aK` müssen unterschiedliche Variablen sein
  - `TConstr` ist ein neuer Typkonstruktor mit Stelligkeit `K`
  - `a1 ... aK` können in Typen `typeI_J` genutzt werden, aber keine anderen Typvariablen
  - `CName1 :: type1_1 -> ... -> type1_N1 -> TConstr a1 ... aK`, etc.

Beispiele mit parametrischem Polymorphismus

Parametrische Listen

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

- `List` ist ein 1-stelliger Typkonstruktor
- Beispiel Typen
  - `List a` – Liste mit Elementen vom Typ `a`
  - `List Integer` – Liste von Zahlen
  - `List Bool` – Liste von Booleans
  - `List (List Integer)` – Liste mit Elementen, die selber Listen von Zahlen sind
- Typ der Konstruktoren
  - `Empty :: List a`
  - `Cons :: a -> List a -> List a`
- Beispiel Werte
  - `Empty :: List a`, `Empty :: List Integer`, `Empty :: List (List Bool)`, ...
  - `Cons 7 (Cons 5 Empty) :: List Integer`, `Cons True Empty :: List Bool`, ...
  - `Cons (Cons 7 (Cons 5 Empty)) (Cons Empty Empty) :: List (List Int)`

    (`7`: `Int`, `5`: `Int`, `Empty`: `List Int`, `Cons 7 (Cons 5 Empty)`: `List Int`, `Empty`: `List Int`, `Empty`: `List (List Int)`, `Cons Empty Empty`: `List (List Int)`)
  - `Cons True (Cons 7 Empty)` nicht erlaubt, kein einheitlicher Element-Typ

Funktionen auf parametrischen Listen

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

- Beispiel Programme

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

first :: List a -> a
first (Cons x _) = x
first _ = error "first on empty list"
```

Parametrische Listen (fortgesetzt)

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

- Funktions-Definitionen können Typklassen-Einschränkungen erzwingen
  - Beispiel: ersetze alle Vorkommen von `x` durch `y` in einer Liste

    ```
    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
    ```

  - Einschränkung `Eq a =>` ist notwendig, weil Elemente mit == verglichen werden
- Funktions-Definitionen können einen konkreten Typ erzwingen
  - Beispiel: ersetze alle Vorkommen von `'A'` durch `'B'` in einer Liste

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

  - wichtig: da `replace` Einschränkung `Eq a` hat, und `a` mit `Char` in `replaceAB` substituiert wird, wird geprüft, dass `Char` wirklich Instanz von Klasse `Eq` ist

Listen in Haskell

- der Typ `List` ist vordefiniert in Haskell
- Unterschiede sind in der Notation
  - statt `List a` schreibt man `[a]`
  - statt `Empty` scheibt man `[]`
  - statt `Cons x xs` schreibt man `x : xs` (und `:` wird "Cons" genannt)
  - zusammengefasst: Haskells Typ für Listen ist:

    ```
    data [a] = [] | a : [a]
    ```

- der Listen Konstruktor `(:)` assoziiert nach rechts:
  `1 : 2 : 3 : [] = 1 : (2 : (3 : []))`
- spezielle Listen Syntax für endliche Listen: `[1, 2, 3] = 1 : 2 : 3 : []`
- Beispiel: `append` auf Haskells Typ für Listen

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

Tupel

- Tupel werden z.B. genutzt, um mehrere Werte als ein Tupel als Resultat zu liefern; Beispiel: Division-mit-Rest Funktion liefert zwei Zahlen, den Quotienten und den Rest
- Tupel können leicht definiert werden

```
data Unit = Unit                    -- Tupel mit 0 Eintraegen
data Pair a b = Pair a b            -- Tupel mit 2 Eintraegen
data Triple a b c = Triple a b c    -- Tupel mit 3 Eintraegen
```

- Beispiel: finde Wert von Schlüssel `'y'` in Liste mit Schlüssel/Wert-Paaren

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

Anmerkung: normalerweise würde man den Schlüssel als Argument übergeben und nicht als `'y'` fixieren

Tupel in Haskell

- Tupel sind in Haskell vordefiniert (nicht nötig, `Pair`, `Triple`, ... selber zu definieren)
- für jedes $n \neq 1$ bietet Haskell:
  - einen Typkonstruktor `( , ..., )` (mit n Einträgen)
  - einen (Wert-)Konstruktor `( , ..., )` (mit n Einträgen)
- Beispiele
  - `Pair a b` und `Triple a b c` sind äquivalent zu `(a, b)` und `(a, b, c)`
  - `(5, True, "foo") :: (Int, Bool, String)`
  - `() :: ()`
  - `(5)` ist genau die Zahl 5, es gibt kein 1-Tupel
  - `(1, 2, 3)` ist nicht das gleiche wie `((1, 2), 3)` oder `(1, (2, 3))`
- Beispiel Programm der vorigen Folie mit vordefinierten Tupeln

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

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

- `Maybe` ist vordefinierter Haskell Typ für optionale Werte
- Beispiel Anwendung: sichere Division ohne Laufzeitfehler

```
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` ist vordefinierter Haskell Typ um alternative Werte zu spezifizieren
- Beispiel Anwendung: optionale Werte mit Fehler-Meldung

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

Zusammenfassung

- Nutzung von Typvariablen und parametrischem Polymorphismus
  - Datentypen mit Typvariablen
  - polymorphe Funktionen, inklusive Typklassen-Einschränkungen
    (Beispiel: `f :: (Eq a, Show b) => a -> Bool -> a -> b -> String`, ...)
- vordefinierte Datentypen
  - Listen `[a]`
  - Tupel (Produkt-Typ) `(..,..,..)`
  - optionale Werte `Maybe a`
  - Alternativen (Summen-Typ) `Either a b`
- vordefinierte Typklassen
  - Arithmetik exklusive Division: `Num a`
  - Arithmetik inklusive Division: `Fractional a`
  - Gleichheit und Ungleichheit: `Eq a`
  - Kleiner und Größer: `Ord a`
  - Konvertierung in `String`s: `Show a`