Programmierung und Modellierung mit Haskell

Fragestunde

Martin Hofmann Steffen Jost

LFE Theoretische Informatik, Institut für Informatik,
Ludwig-Maximilians Universität, München

8. Mai 2014

Genauigkeit / Faulheit

Beispiel:

Bei der Aufgabe mit dem Riesenfass Weisswürschte und Brezen, wird irgendwie mit “passt schon” herum argumentiert.

Besser:

Einfach alle Fälle aufzählen und **danach** einzeln behandeln.
So wird nichts übersehen und kein Fall vergessen.

Generell bei Aufgaben:

Bevor man ratlos dasitzt und grübelt, erst einmal Aufschreiben, was man weiß, und was man am Ende benötigt!

Pattern Matching

Bei Pattern-Matching sind Konstruktoren mit Argumenten immer zu Klammern;
die Reihenfolge der Argumente ist zu beachten!

```
data Maybe a    = Nothing | Just a
data Either a b = Left a  | Right b

foo :: Maybe Int -> Either Bool String -> String
foo  Just x   Left y   = "NOT OK"      -- Syntaxfehler
foo (Just x) (Left y)  = "OK"
foo Nothing  (Right _) = "OK"
```

Konstruktoren ohne Argumente müssen im Pattern-Matching nicht geklammert werden.

Pattern Matching mit Listen

Schreibweise `[1,2,3]` ist nur Kurzform für `1:2:3:[]`

Pattern Matching mit Listen

Schreibweise `[1,2,3]` ist nur Kurzform für `1:2:3:[]`

`(:)` ist ein Infix-Konstruktor; Konstruktoren mit Argumenten sind im Mustervergleich immer zu klammern!

Pattern Matching mit Listen

Schreibweise `[1,2,3]` ist nur Kurzform für `1:2:3:[]`

`(:)` ist ein Infix-Konstruktor; Konstruktoren mit Argumenten sind im Mustervergleich immer zu klammern!

```
bar :: [a] -> String
bar []             = "Leere Liste"
bar [x]            = "Genau 1 Element"
bar [x,y]          = "Genau 2 Elemente"
bar [x,y,z]        = "Genau 3 Elemente"
bar (h:t)          = "Mindestens 1 Element"
bar (x:y:zs)       = "Mindestens 2 Elemente"
bar (x1:x2:x3:xs)  = "Mindestens 3 Elemente"
```

Pattern Matching mit Listen

Schreibweise `[1,2,3]` ist nur Kurzform für `1:2:3:[]`

`(:)` ist ein Infix-Konstruktor; Konstruktoren mit Argumenten sind im Mustervergleich immer zu klammern!

```
bar :: [a] -> String
bar []             = "Leere Liste"
bar [x]            = "Genau 1 Element"
bar [x,y]          = "Genau 2 Elemente"
bar [x,y,z]        = "Genau 3 Elemente"
bar (h:t)          = "Mindestens 1 Element"
bar (x:y:zs)       = "Mindestens 2 Elemente"
bar (x1:x2:x3:xs) = "Mindestens 3 Elemente"
```

Typen der Variablen im Beispiel: (“one x”, “some xs”)

```
h,x,y,z,x1,x2,x3 ::  a    -- ein Element der Liste
t,xs,zs          :: [a]   -- eine (Rest-) Liste
```

Pattern vor Wächter

Pattern Matching bevorzugen vor Wächtern:

```
data Maybe a = Nothing | Just a

fooOK :: Maybe a -> Int
fooOK Nothing  = 0
fooOK (Just x) = 1


fooBAD x
 | x == Nothing = 0
 | otherwise    = 1
```

Pattern vor Wächter

Pattern Matching bevorzugen vor Wächtern:

```
data Maybe a = Nothing | Just a

fooOK :: Maybe a -> Int
fooOK Nothing  = 0
fooOK (Just x) = 1

fooBAD :: Eq a => Maybe a -> Int
fooBAD x
 | x == Nothing = 0
 | otherwise    = 1
```

Anwendung von `(==)` erzwingt Einschränkung auf Typklasse `EQ`:
für Gleichheit muss der komplette Wert betrachtet werden!
Bei Pattern-Matching wird nur der Konstruktor verglichen,
aber nicht notwendigerweise der Wert!

Funktionsdefinition

Was bedeutet "Funktion mit 3 Argumenten"?

1. Möglichkeit

```
foo :: a -> b -> c -> (a,b,c)
```

explizit geklammert auch

```
foo :: a -> (b -> (c -> (a,b,c)))
```

2. Möglichkeit

```
foo :: (a,b,c) -> (a,b,c)
```

Funktionsdefinition

Was bedeutet "Funktion mit 3 Argumenten"?

1. Möglichkeit

```
foo :: a -> b -> c -> (a,b,c)
```

explizit geklammert auch

```
foo :: a -> (b -> (c -> (a,b,c)))
```

2. Möglichkeit

```
foo :: (a,b,c) -> (a,b,c)
```

eigentlich ein Argument, ein Paket mit 3 Teilen

Wenn kein Typ angegeben wurde, dann ist die Unterscheidung momentan egal.

Beim Rückgabetyp haben wir hier keine Wahl!

Rekursion

Definition einer rekursiven Funktion:

- Zuerst offensichtliche Fallunterscheidungen mit Pattern-Matching behandeln
- Bei Behandlung der einzelnen Fälle: daran denken, dass man eine Funktion auf "kleinere" Argumente anwenden kann, welche genau dieses Problem löst – nämlich die Funktion, welche man gerade definiert!
- Sicherstellen, dass Funktion vollständig definiert ist:
  - Sind alle Fallunterscheidungen vollständig?
  - Werden partielle Funktionen aufgerufen (`head`,`div x 0`,...)?
- Sicherstellen, dass alle rekursiven Aufrufe terminieren!

  Abstiegsfunktion: ein Maß für die "Schwierigkeit" der gegebenen Argumente. Ist dieses Maß 0, so können/müssen die Argumente ohne Rekursion verarbeitet werden.

Kompilierfehler

Folgendes Proramm kompiliert nicht:

```
main = print result

result = []
```

Kompilierfehler

Folgendes Proramm kompiliert nicht:

```
main = print result

result = []
```

Lösung

```
main = print result
result :: [Int]
result = []
```

Um eine Liste in einen String umzuwandeln, muss Haskell wissen, welche Version von `show` angewendet wird, um die Elemente in Strings umzuwandeln.
Bei der leeren Liste darf GHC hier nicht entscheiden, welche Funktion für die (nicht vorhandenen) Elemente verwendet werden soll. Die Typsignatur erzwingt diese Entscheidung bewusst.