Fortgeschrittene Funktionale Programmierung

6. und 7. Vorlesung

Janis Voigtländer

Universität Bonn

Wintersemester 2015/16

Zu einfache(n) Datentypen

Ein einfacher Datentyp:

$$\textbf{data}\ \text{Tree}\ a = \text{Leaf}\ a \mid \text{Branch}\ (\text{Tree}\ a)\ (\text{Tree}\ a)$$

Und zwei Funktionen darauf:

$$\begin{aligned}
&\texttt{mirror} :: \text{Tree}\ a \to \text{Tree}\ a \\
&\texttt{mirror}\ (\text{Branch}\ t_1\ t_2) = \text{Branch}\ (\texttt{mirror}\ t_2)\ (\texttt{mirror}\ t_1) \\
&\texttt{mirror}\ t \qquad\qquad\qquad\quad = t
\end{aligned}$$

$$\begin{aligned}
&\texttt{subst} :: \text{Tree}\ a \to \text{Tree}\ b \to \text{Tree}\ b \\
&\texttt{subst}\ t\ t' = \texttt{go}\ t \\
&\quad \textbf{where}\ \texttt{go}\ (\text{Branch}\ t_1\ t_2) = \text{Branch}\ (\texttt{go}\ t_1)\ (\texttt{go}\ t_2) \\
&\qquad\quad\ \ \texttt{go}\ _ \qquad\qquad\qquad\quad = t'
\end{aligned}$$

Angenommen, wir würden gern die Invariante ausdrücken, dass es sich um vollständige Binärbäume handelt (und dass obige Funktionen diese Eigenschaft erhalten).

Nested Datatypes

Rekursive Positionen auf der rechten Seite dürfen andere Typargumente haben als das auf der linken Seite:

data Tree a = Leaf a | Branch (Tree (a, a))

Beispielwerte:

$\texttt{tree}_0, \texttt{tree}_1, \texttt{tree}_2$ :: Tree Integer
$\texttt{tree}_0$ = Leaf 1
$\texttt{tree}_1$ = Branch (Leaf $(1, 2)$)
$\texttt{tree}_2$ = Branch (Branch (Leaf $((1, 2), (3, 4))$))

Nested Datatypes

Rekursive Positionen auf der rechten Seite dürfen andere Typargumente haben als das auf der linken Seite:

data Tree a = Leaf a | Branch (Tree (a, a))

Etwas eingängiger modelliert:

newtype Leaf a = Leaf a
data Branch a = Branch a a
data Tree a = Zero a | Succ (Tree (Branch a))

$\texttt{tree}_0, \texttt{tree}_1, \texttt{tree}_2$ :: Tree (Leaf Integer)
$\texttt{tree}_0$ = Zero (Leaf 1)
$\texttt{tree}_1$ = Succ (Zero (Branch (Leaf 1) (Leaf 2)))
$\texttt{tree}_2$ = Succ (Succ (Zero (Branch (Branch (Leaf 1) (Leaf 2))
(Branch (Leaf 3) (Leaf 4)))))

Und wie programmieren wir jetzt auf diesen Typen?

Nested Datatypes

Benötigt polymorphe Rekursion:

```
mirror :: Tree (Leaf a) → Tree (Leaf a)
mirror t = go t id
  where go :: Tree b → (b → b) → Tree b
        go (Succ t) f = Succ (go t (λ(Branch t1 t2)
                                    → Branch (f t2) (f t1)))
        go (Zero x) f = Zero (f x)
```

Test:

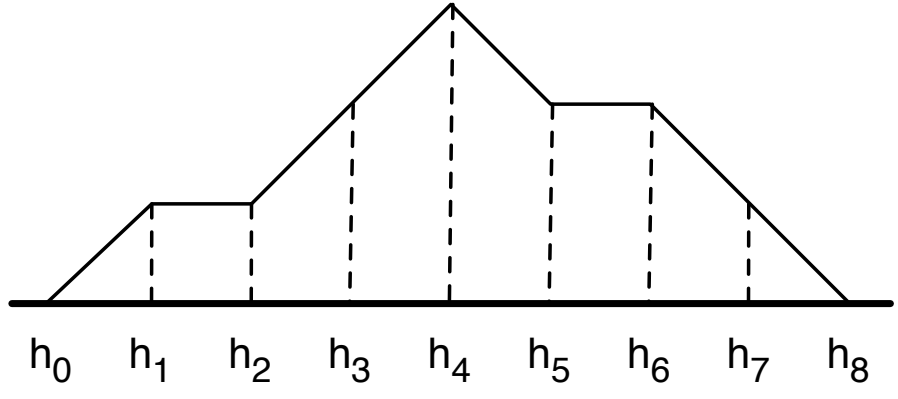
```
> mirror tree2
Succ (Succ (Zero (Branch (Branch (Leaf 4) (Leaf 3))
                         (Branch (Leaf 2) (Leaf 1)))))
```

Und was ist mit `subst`???

Aufgabe 3: Alle Alpen

Eine Schülergruppe möchte ein eigenes 2D-Computerspiel realisieren. Lilli ist für die Generierung der Hintergrundbilder zuständig. Die Szenerie soll ein Gebirge sein, und Lilli schlägt eine einfache Methode zur Beschreibung von Gebirgszügen vor, nämlich als Folge von Höhenwerten. Aber: Ist diese Darstellung variantenreich genug?

Lilli präzisiert ihre Idee: Ein Gebirgszug der Länge N sei eine Folge $(h_0, h_1, \ldots, h_N)$ von $N+1$ nicht-negativen ganzen Zahlen mit $h_0 = h_N = 0$ und $|h_i - h_{i-1}| \leq 1$ für $i = 1, \ldots, N$. Zum Beispiel ist $(0,1,1,2,3,2,2,1,0)$ ein Gebirgszug der Länge 8.



Nun möchte sie ein Programm schreiben, das ihr erlaubt, ihre Idee zu prüfen. Versetze dich in ihre Lage und bearbeite wie sie folgende

Aufgabe

1. Schlage eine Darstellung von Gebirgszügen in der Programmiersprache vor, die du in dieser Aufgabe benutzen möchtest.

Challenge

```
type Alpen = ...    -- „linear repräsentiert“
mountains :: Alpen → [Int]   -- injektiv,
                             -- liefert nur „legale“ Listen,
                             -- liefert alle legalen Listen
generate :: Int → [Alpen]
test = sort (map mountains (generate 4)) == [[0,0,0,0,0],
                                              [0,0,0,1,0],
                                              [0,0,1,0,0],
                                              [0,0,1,1,0],
                                              [0,1,0,0,0],
                                              [0,1,0,1,0],
                                              [0,1,1,0,0],
                                              [0,1,1,1,0],
                                              [0,1,2,1,0]]
```

Ein möglicher Ausgangspunkt

```
data Alpen = End | Up Alpen | Equal Alpen | Down Alpen

mountains :: Alpen → [Int]
mountains End          = [0]
mountains (Up ms)      = 0 : map (+1) (mountains ms)
mountains (Equal ms)   = 0 : mountains ms
mountains (Down ms)    = 0 : map (λx → x − 1) (mountains ms)
```

- injektiv?
- liefert nur legale Listen?
- liefert alle legalen Listen?

```
> mountains (Up (Down (Down (Up End))))
[0, 1, 0, −1, 0]
> mountains (Up End)
[0, 1]
> mountains (Down End)
[0, −1]
```

Ein möglicher Ausweg?

```
type Alpen = Mountains0

data Mountains0 = End | Up Mountains1 | Equal Mountains0
data Mountains1 = Up Mountains2 | Equal Mountains1
                | Down Mountains0
data Mountains2 = Up Mountains3 | Equal Mountains2
                | Down Mountains1
...

mountains :: Alpen → [Int]
mountains End        = [0]
mountains (Up ms)    = 0 : map (+1) (mountains ms)
mountains (Equal ms) = 0 : mountains ms
mountains (Down ms)  = 0 : map (λx → x − 1) (mountains ms)
```

Hmm, wir glauben nicht wirklich, dass das funktionieren wird, oder?

Aber vielleicht ja so?

type Alpen $=$ Mountains 0

data Mountains $k =$ Up (Mountains $(k+1)$) | Equal (Mountains k)
| Down (Mountains $(k-1)$)

Aber wie geht man dann mit End um? Und mit Down (Mountains -1)?

Etwa so?

data Mountains $k =$ Up (Mountains $(k+1)$) | Equal (Mountains k)
| Down (Mountains $(k-1)$) -- if $k > 0$
| End -- if $k = 0$

Oder so?

data Mountains $k =$ Up (Mountains $(k+1)$) | Equal (Mountains k)
| **if** $k > 0$ **then** Down (Mountains $(k-1)$) **else** ()
| **if** $k == 0$ **then** End **else** ()

Aber vielleicht ja so?

$$\textbf{type}\ \text{Alpen} = \text{Mountains}\ 0$$

$$\begin{aligned}\textbf{data}\ \text{Mountains}\ k = {} & \text{Up}\ (\text{Mountains}\ (k+1)) \mid \text{Equal}\ (\text{Mountains}\ k) \\ & \mid \text{Down}\ (\text{Mountains}\ (k-1))\end{aligned}$$

Aber wie geht man dann mit End um? Und mit Down (Mountains -1)?

Oder so?

$$\begin{aligned}\textbf{data}\ \text{Mountains}\ k = {} & \text{Up}\ (\text{Mountains}\ (k+1)) \mid \text{Equal}\ (\text{Mountains}\ k) \\ & \mid \textbf{if}\ k > 0\ \textbf{then}\ \text{Down}\ (\text{Mountains}\ (k-1))\ \textbf{else}\ () \\ & \mid \textbf{if}\ k == 0\ \textbf{then}\ \text{End}\ \textbf{else}\ ()\end{aligned}$$

Und dann vielleicht:

$$\begin{aligned}&\texttt{generate} :: \text{Int} \to [\text{Mountains}\ k] \\ &\texttt{generate}\ 0 = \textbf{if}\ k == 0\ \textbf{then}\ [\text{End}]\ \textbf{else}\ [\,] \\ &\texttt{generate}\ n = \texttt{map}\ \text{Up}\ (\texttt{generate}\ (n-1)) \\ &\qquad\qquad +\!\!+\ \texttt{map}\ \text{Equal}\ (\texttt{generate}\ (n-1)) \\ &\qquad\qquad +\!\!+\ \textbf{if}\ k > 0\ \textbf{then}\ \texttt{map}\ \text{Down}\ (\texttt{generate}\ (n-1))\ \textbf{else}\ [\,]\end{aligned}$$

That way lies madness (or Agda).

Or, does it (have to)?

Ein „kleines“, zunächst unscheinbares Sprachfeature:

```
{-# LANGUAGE GADTs, KindSignatures #-}
```

Dann möglich zu schreiben:

```
data Mountains :: * where
  End   :: Mountains
  Up    :: Mountains → Mountains
  Equal :: Mountains → Mountains
  Down  :: Mountains → Mountains
```

statt: `data Mountains = End | Up Mountains | Equal Mountains | Down Mountains`

Oder auch, etwa:

```
data Tree :: * → * where
  {Leaf :: a → Tree a; Branch :: Tree a → Tree a → Tree a}
```

statt: `data Tree a = Leaf a | Branch (Tree a) (Tree a)`

What's the big deal?

Dann auch möglich:

type Alpen = Mountains Zero

data Mountains :: $* \to *$ **where**
  End :: Mountains Zero
  Up :: Mountains (Succ k) $\to$ Mountains k
  Equal :: Mountains k $\to$ Mountains k
  Down :: Mountains k' $\to$ Mountains (Succ k')

deriving instance Show (Mountains k)

wobei:

data Zero
data Succ k

„Entspricht“:

data Mountains k = Up (Mountains $(k+1)$) | Equal (Mountains k)
  | Down (Mountains $(k-1)$) -- if $k > 0$
  | End -- if $k = 0$

What's the big deal?

Dann auch möglich:

type Alpen = Mountains Zero

data Mountains :: $* \to *$ **where**
  End :: Mountains Zero
  Up :: Mountains (Succ k) $\to$ Mountains k
  Equal :: Mountains k $\to$ Mountains k
  Down :: Mountains k' $\to$ Mountains (Succ k')

deriving instance Show (Mountains k)

wobei:

data Zero
data Succ k

Jetzt sind die problematischen (Up (Down (Down (Up End)))) und (Up End) nicht mehr wohlgetypt, und (Down End) hat nicht mehr den Typ Alpen!

Konvertierung?

```
mountains :: Mountains k → [Int]
mountains End          = [0]
mountains (Up ms)      = 0 : map (+1) (mountains ms)
mountains (Equal ms)   = 0 : mountains ms
mountains (Down ms)    = 0 : map (λx → x − 1) (mountains ms)
```

Was „tatsächlich“ vor sich geht:

```
mountains :: Mountains k → [Int]
mountains (End :: Mountains Zero)              = [0]
mountains (Up (ms :: Mountains (Succ k)))      = 0 : map (+1) ...
mountains (Equal (ms :: Mountains k))          = 0 : mountains ms
mountains (Down (ms :: Mountains k'))          = 0 : ...
```

- injektiv?
- liefert nur legale Listen (bei Aufruf auf Mountains Zero)?
- liefert alle legalen Listen (bei Aufruf auf Mountains Zero)?

Aber wie denn nun passende Werte generieren?

Im Prinzip:

```
generate :: Int → [Mountains k]
generate 0 | k ≡ Zero      = [End]
           | k ≡ (Succ k') = []
generate n | k ≡ Zero      = map Up (generate (n − 1))
                           ++ map Equal (generate (n − 1))
           | k ≡ (Succ k') = map Up (generate (n − 1))
                           ++ map Equal (generate (n − 1))
                           ++ map Down (generate (n − 1))
```

Mit Typklassenabstraktion!

```
class Generate k where
  generate :: Int → [Mountains k]

instance Generate Zero where
  generate 0 = [End]
  generate n = map Up (generate (n − 1))
             ++ map Equal (generate (n − 1))
```

Aber wie denn nun passende Werte generieren?

Im Prinzip:

```
generate :: Int → [Mountains k]
generate 0 | k ≡ (Succ k') = []
generate n | k ≡ (Succ k') = map Up (generate (n − 1))
                           ++ map Equal (generate (n − 1))
                           ++ map Down (generate (n − 1))
```

Mit Typklassenabstraktion!

```
class Generate k where
  generate :: Int → [Mountains k]

instance Generate Zero where
  generate 0 = [End]
  generate n = map Up (generate (n − 1))
             ++ map Equal (generate (n − 1))
```

Aber wie denn nun passende Werte generieren?

Mit Typklassenabstraktion!

```
class Generate k where
  generate :: Int → [Mountains k]

instance Generate Zero where
  generate 0 = [End]
  generate n = map Up (generate (n − 1))
               ++ map Equal (generate (n − 1))

instance Generate k' ⇒ Generate (Succ k') where
  generate 0 = []
  generate n = map Up (generate (n − 1))
               ++ map Equal (generate (n − 1))
               ++ map Down (generate (n − 1))

> sort (map mountains ((generate 4) :: [Alpen])) == ...
True
```

Challenge solved!

Populäre GADT-Anwendung: Length-Aware Lists

Analog zum Alpen-Beispiel:

data Vector :: $* \to * \to *$ **where**
  Nil :: Vector Zero a
  Cons :: $a \to$ Vector k $a \to$ Vector (Succ k) a

Beispiele:

`empty` = Nil
`list` = Cons 3 (Cons 2 Nil)

Automatisch inferierte Typen:

`empty` :: Vector Zero a
`list` :: Vector (Succ (Succ Zero)) Integer

Populäre GADT-Anwendung: Length-Aware Lists

Funktionen darauf:

```
hd :: Vector k a → a
hd (Cons x xs) = x
hd Nil         = error "empty vector"
```

Aber so natürlich kein Gewinn an Ausdruckskraft:

```
> hd empty
*** Exception: empty vector
```

Stattdessen:

```
hd :: Vector (Succ k) a → a
hd (Cons x xs) = x
```

Nun, bei `hd empty` Compile-Time-Error! Aber `hd list` okay.

Analog:

```
tl :: Vector (Succ k) a → Vector k a
tl (Cons x xs) = xs
```

Populäre GADT-Anwendung: Length-Aware Lists

Was ist mit interessanteren Funktionen, zum Beispiel folgender?

app Nil $ys = ys$
app (Cons x xs) ys = Cons x (app xs ys)

Welchen Typ könnte die denn haben? Sicher irgendeine „Einschränkung“ von Vector k $a \to$ Vector l $a \to$ Vector m a.

Ein mittlerweile etwas antiquierter Ansatz:
Typklassen über Phantomtypen.

class Add k l $m \mid k$ $l \to m$ **where**

instance Add Zero l l **where**
instance Add k l $m \Rightarrow$ Add (Succ k) l (Succ m) **where**

app :: Add k l $m \Rightarrow$ Vector k $a \to$ Vector l $a \to$ Vector m a

Woran erinnert das hoffentlich zumindest einige von Ihnen?

Populäre GADT-Anwendung: Length-Aware Lists

Moderner:

```
type family     Add (k :: *) (l :: *) :: *
type instance Add Zero      l = l
type instance Add (Succ k) l = Succ (Add k l)

app :: Vector k a → Vector l a → Vector (Add k l) a
app Nil          ys = ys
app (Cons x xs) ys = Cons x (app xs ys)
```

Noch moderner (aber nicht das Ende der Geschichte):

```
data Nat = Zero | Succ Nat

data Vector :: Nat → * → * where
  Nil  :: Vector Zero a
  Cons :: a → Vector k a → Vector (Succ k) a

type family Add (k :: Nat) (l :: Nat) :: Nat
...
```

Zurück zum Beispiel vollständiger Binärbäume

Nun:

```
data Tree :: Nat → ∗ → ∗ where
  Leaf   :: a → Tree Zero a
  Branch :: Tree k a → Tree k a → Tree (Succ k) a

tree₂ :: Tree (Succ (Succ Zero)) Integer
tree₂ = Branch (Branch (Leaf 1) (Leaf 2)) (Branch (Leaf 3) (Leaf 4))

mirror :: Tree k a → Tree k a
mirror (Branch t₁ t₂) = Branch (mirror t₂) (mirror t₁)
mirror t              = t

subst :: ∀k l a b.Tree k a → Tree l b → Tree (Add k l) b
subst t t′ = go t
  where go :: Tree k′ a → Tree (Add k′ l) b
        go (Branch t₁ t₂) = Branch (go t₁) (go t₂)
        go (Leaf _)       = t′
```

Und wozu könnte man GADTs noch gebrauchen?

Denken wir an einen Mini-Interpreter.

Zur Erinnerung:

data Expr = Lit Int | Add Expr Expr | Sub Expr Expr | Mul Expr Expr

```
expr   :: Parser Expr
term   :: Parser Expr
factor :: Parser Expr
nat    :: Parser Int

eval   :: Expr → Int
```

Angenommen, wir wollen unsere Sprache neben Arithmetik um Boolesche Bedingungen erweitern:

data Expr = Lit Int | Add Expr Expr | Sub Expr Expr | Mul Expr Expr
| Equal Expr Expr | Not Expr | And Expr Expr
| If Expr Expr Expr

Wie garantieren wir Typsicherheit der „eingebetteten“ Sprache?

Und wozu könnte man GADTs noch gebrauchen?

Wie garantieren wir Typsicherheit der „eingebetteten“ Sprache?

Idee:

```
data Expr :: * → * where
  Lit   :: Int → Expr Int
  Add   :: Expr Int → Expr Int → Expr Int
  Sub   :: Expr Int → Expr Int → Expr Int
  Mul   :: Expr Int → Expr Int → Expr Int
  Equal :: Eq t ⇒ Expr t → Expr t → Expr Bool
  Not   :: Expr Bool → Expr Bool
  And   :: Expr Bool → Expr Bool → Expr Bool
  If    :: Expr Bool → Expr t → Expr t → Expr t
```

Dann:

```
eval :: Expr t → t
```

Zur Übung

Definieren Sie für Alpen bzw. Mountains die Funktion `arbitrary` (in Instanzen der QuickCheck-Typklasse Arbitrary), so dass etwa folgender Code funktioniert (und etwas Sinnvolles tut):

```
main = sample $ do alps ← arbitrary :: Gen Alpen
                   return (mountains alps)
```