Haskell mit Stil

Johannes Waldmann, Leipzig

Überblick

zu jeder Programmiersprache gehören

- Syntax (wie sehen Programme aus?)
- Semantik (was bedeuten sie?)
- Pragmatik: welches Programm (von mehreren gleichbedeutenden) ist am nützlichsten?

Pragmatik für Haskell (dieser Vortrag)

- Syntax, • Funktionen,
- Daten, • Typklassen

Vielen Dank an Lutz Donnerhacke (IKS GmbH, Jena)
für Vorab-Diskussion per Email.

Dutch Layout

(Export-)Listen, Tupel, Records: — nicht so:

```
foo = [ bar,
        baz,
        bams ]
```

sondern so:

```
foo = [ bar
      , baz
      , bams
      ]
```

kann Zeilen leichter hinzufügen
und auskommentieren

Separieren oder Terminieren?

Kommentar von Henning Thielemann:

nicht so:

```
foo = [ bar
      , baz
      , bams
      ]
```

sondern so:

```
foo = bar :
      baz :
      bams :
      []
```

siehe
http://www.haskell.org/haskellwiki/Terminator_

Einrückung

nicht so:

```
foo = case bar of
         baz -> ...
```

sondern so:

```
foo =
    case bar of
        baz -> ...
```

Prinzip: Einrückung soll *nicht* von der Länge von Bezeichnern abhängen, weil diese sich evtl. ändert.

Klammern und Dollars

nicht so:

```
f ( g ( h bar ))
```

sondern so:

```
f $ g $ h bar
```

weniger Zeichen, leichter erweiterbar

oder so:

```
f . g . h $ bar
```

leichter transformierbar

```
comb = f . g . h
```


Datenflußrichtung

nicht so: `f $ g $ h bar`
sondern so:

```
(#) :: a -> (a -> b) -> b  ;  x # f = f x

bar # h # g # f
```

eigentlich sogar so (vgl. vorige Folie):

```
bar # h . g . f   -- DOES NOT WORK
```

aber der Punkt ist leider bereits für die Funktionskomposition in der falschen Reihenfolge vergeben (fragen Sie einen Algebraiker nach $f \circ g$)

Let statt Where

nicht so:

```
foo = f bar bar
   where  bar = ...
```

sondern so:

```
foo =
   let bar = ...
   in  f bar bar
```

Prinzip: Deklaration vor Benutzung
gegensätzliches Prinzip: unwichtige Details unterdrücken und später klären

List comprehensions/do

nicht so:

```
[ z * z
| x <- [ 1 .. 8 ]
, let z = 3 * x + 1
]
```

sondern so:

```
do x <- [ 1 .. 8 ]
   let z = 3 * x + 1
   return $ z * z
```

Prinzip: Deklaration vor Benutzung
gegensätzlich: Details unterdrücken

Namen oder keine Namen

nicht so:

```
zipWith (flip (-)) foo bar
```

sondern so:

```
do (x,y) <- zip foo bar
   return $ y - x
```

Gegenargument: variablenfreie Notation erleichtert Programmtransformationen

Kompromiß: `zipWith (\ x y -> y - x) foo bar`

Unterprogramme statt Kommentare

nicht so:

```
-- noch mehr kanten raten
edges_rest <- sequence $ replicate (edges conf) $ do
   from <- eins cs ; to <- eins rest ; return $ kante from to
```

sondern so:

```
edges_rest <- noch_mehr_kanten (edges conf) cs rest
...
noch_mehr_kanten n starts ends =
   sequence $ replicate n $ do
      from <- eins starts ; to <- eins ends ; return $ kante
```

logische Einheiten durch *primäre* Sprachmittel ausdrücken (Layout, Kommentare sind sekundär)

Datentypen

nicht so:

```
foo :: ([a],a,(a,a)) -> ...
foo x = ... fst $ snd3 ...
```

sondern so

```
data State a = State { input :: [a] , ... }
foo :: State a -> ...
```

Prinzip: vordefinierte Datentypen (Zahlen, Strings, Listen, Tupel) nur als (unsichbare) Bausteine für anwendungsspezifische Datentypen

Benannt anstatt positional

nicht so:

```
data Student = Student String String
```

sondern so:

```
data Student = Student
   { vorname :: String, nachname :: String }
```

Hinzufügen von Komponenten ist leichter

code smell: lange Parameterliste (die kann sich nämlich keiner merken) → Parameter-Objekt (Record) einführen

Gegenargument: partielle Anwendung schwerer

Qualifizierte Namen

nicht so:

```
module Foo where data Foo = ...
sizeFoo :: Foo -> Int
```

sondern so:

```
size :: Foo -> Int
import qualified Foo;   ... Foo.size ...
```

Prinzip: Typ- und Modul-Information sollte nicht Bestandteil von Namen sein (sondern durch primäre Mittel ausgedrückt werden)

Module

Prinzip: jedes Modul definiert genau einen Typ
...aber wie heißt der?

```
module Foo where data Foo = Foo { oof :: Int }

module Bar where import qualified Foo
x :: Foo.Foo ; x = Foo.Foo { Foo.oof = 9 }
```

Vorschlag:

```
module Foo where data Type = Make { oof :: Int

x :: Foo.Type ; x = Foo.Make { Foo.oof = 9 }
```

Problem: deriving Show (braucht dann FQN)

module/data — Diskussion

Haskell–Vorschlag

```
module Foo where data Type = Make { oof :: Int

x :: Foo.Type ; x = Foo.Make { Foo.oof = 9 }
```

in Java: Modul = Klasse (= Typ = Namensraum), deswegen ist auch die Benennung einfacher.

```
class Foo { int oof; }
Foo x = ... ;   ... x.foo ...
```

Entwicklung in Haskell: positionelle Records → benannte Records, Module → hierarchische Module; paßt heute nicht gut zusammen.

Interfaces (1)

nicht so:

```
sizeFoo :: Foo -> Int ; ... sizeFoo x ...
size    :: Foo -> Int ; ... Foo.size x ...
```

sondern so:

```
class Size s where size :: s -> Int

module Foo where instance Size Foo where ...

x :: Foo.Type ; ... size x ...
```

Prinzip: Eigenschaften konkreter Datentypen durch Schnittstellen formalisieren

Interfaces (2)

```
class C implements F { ... }
int foo (F x) { ... } -- Argument
F bar () { ... return new C (); } -- Resultat
   class F a ...; instance F C where ...;
   foo :: forall a . F a => a -> Int -- OK
   bar :: exists a . F a => Int -> a -- NO
```

Workaround,

```
data WF = forall a . F a => WF a
bar :: Int -> WF ; bar 5 = WF ( C 3 )
```

mit Trick (von Georg Martius):

```
instance F WF where f (WF x) = f x
```


Interfaces (3)

(scheinbarer) Widerspruch:

- anwendungsspezifische Datentypen
- wiederverwendbare (Standard-)Bibliotheken
  . . . die kennen ja die Datentypen noch gar nicht!

Lösung: wiederverwendbarer Code *muß* standardisierte Schnittstellen benutzen.
einziger Kommunikations-Datentyp sei *Funktion* (explizit: Higher Order Functions, implizit: Typklassen)
aber keine Ints, Strings, Listen, Data.Map