Haskell Style Guide

Björn Peemöller

15. April 2013

Dieses Dokument enthält eine Anleitung für den bevorzugten Haskell-Programmierstil im Rahmen der Vorlesung "Fortgeschrittene Programmierung". Der Stil ist angelehnt an den Haskell Style Guide von Johan Tibell, wurde jedoch an die Bedürfnisse der Vorlesung angepasst.

Formatierung

Zeilenlänge

Die maximale Zeilenlänge beträgt *80 Zeichen*.

Einrückung

Tabulatoren sind böse, denn je nach Einrückungstiefe für einen Tabulator kann ein Programm eine andere Semantik bekommen. Daher sollen Leerzeichen zur Einrückung verwendet werden, wobei Codeblöcke allgemein um *2 Leerzeichen* einzurücken sind. Die meisten Texteditoren können auch so eingestellt werden, dass ein Tabulator automatisch durch 2 Leerzeichen ersetzt wird.

Lokale Definitionen

Das `where`-Schlüsselwort wird um *1 Zeichen* eingerückt, die lokalen Definitionen ein weiteres Leerzeichen, insgesamt also *2 Zeichen*. Existiert nur eine lokale Definition, so kann diese auch direkt hinter dem `where` folgen. Mehrere lokale Wertdefinitionen sollten am Gleichheitszeichen ausgerichtet werden, lokale Funktionen sind wie Top-Level-Funktionen auszurichten:

```
f y = x + y
 where y = 1
```

```haskell
g x = x + val1 + secondVal
 where
  val1      = 1
  secondVal = if isZero x then 1 else 2

  -- local function
  isZero 0 = True
  isZero _ = False
```

Leerzeilen

Zwischen zwei Top-Level-Definitionen sollte jeweils eine Leerzeile stehen, zwischen Typsignatur und Implementierung gehört keine Leerzeile. Kommentare zu Top-Level-Definitionen werden ebenfalls nicht abgegrenzt.

```haskell
-- f increments the argument by one
f :: Int -> Int
f x = x + 1

-- fInv decrements the argument by one
fInv x = x - 1
```

Leerzeichen

Vor und hinter zweistellige Operatoren wie `(++)` ist jeweils ein Leerzeichen zu setzen.

```haskell
-- bad
"Not"++"good"

-- good
"Very" ++ "good"
```

Bei arithmetischen Ausdrücken wie `n+1` kann man (muss man aber nicht) davon abweichen.

```haskell
n * (n+1) / 2
```

Analog zum Deutschen folgt nach einem Komma ein Leerzeichen, davor wird keins gesetzt:

```haskell
aList  = [1, 2, 3]
aTuple = (True, "True", 1)
```

Nach einem Lambda kann ein Leerzeichen gesetzt werden, insbesondere wenn direkt nach dem Lambda ein Muster folgt. Folgt nach dem Lambda eine Variable, kann das Leerzeichen aber auch entfallen:

```
map (\ (_:_) -> True)
map (\ x -> x + 1)
map (\x -> x + 1)
```

Typsignaturen

In Typsignaturen sind jeweils Leerzeichen um den Funktionspfeil `->` zu setzen.

```
map :: (a -> b) -> [a] -> [b]
```

Die Bestandteile der Typsignatur stehen üblicherweise alle in einer Zeile, sofern dies nicht die maximale Zeilenlänge überschreitet. Bei sehr langen Signaturen, oder wenn einzelne Typen mit einem Kommentar versehen werden sollen, sind die Funktionspfeile untereinander auszurichten:

```
uncurry10 :: (a -> b -> c -> d -> e -> f -> g -> h -> i -> j -> k)
          -> (a, b, c, d, e, f, g, h, i, j)
          -> k

area :: Int -- ^ width
     -> Int -- ^ height
     -> Int -- ^ area
```

`data`-Deklarationen

Die Konstruktoren in einer `data`-Deklarationen sollten untereinander ausgerichtet werden.

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

Bei wenigen, kurzen Konstruktoren können diese auch in einer Zeile geschrieben werden:

```
data Bit = Zero | One
```

Records (`data`-Deklarationen mit benannten Selektoren) sind wie folgt auszurichten:

```
data Person = Person
  { firstName :: String
  , lastName  :: String
  , age       :: Int
  } deriving (Eq, Show)
```

Listen- und Tupeldeklarationen

Die Elemente längerer Listen sollten wie folgt ausgerichtet werden:

```
exceptions =
  [ InvalidStatusCode
  , MissingContentHeader
  , InternalServerError
  ]
```

Optional kann man auch den ersten Zeilenumbruch weglassen:

```
directions = [ North
             , East
             , South
             , West
             ]
```

Kurze Listen müssen natürlich nicht vertikal ausgerichtet werden:

```
short = [1, 2, 3]
```

Die gleichen Regeln können auch auf Tupel angewandt werden:

```
t       = (1, True)
ignored = ( InvalidStatusCode
          , MissingContentHeader
          )
```

Modulkopf

Eine kurze Exportliste, die in eine Zeile passt, kann wie folgt geschrieben werden:

```
module Set (Set, empty) where
```

Längere Exportlisten sind wie folgt auszurichten:

```
module Data.Set
  ( Set
  , empty
  , singleton
  , member
  ) where
```

Optional können auch mehrere Namen in einer Zeile aufgeführt werden.

Importdeklarationen

Die Importliste wird zunächst anhand der folgenden 3 Kategorien geordnet:

1. Standardmodule, z.B. `Data.List`, `System.IO`
2. Eventuell verwendete Fremdbibliotheken (hier weniger relevant)
3. Andere eigene Module

Die Importliste jeder Kategorie sollte alphabetisch sortiert sein. Mit Ausnahme der Prelude sollten die benutzten Funktionen explizit oder qualifiziert importiert werden. Von dieser Regel kann in Ausnahmefällen abgewichen werden, wenn z.B. sehr viele Namen eines Moduls importiert werden:

```
import           Data.List        (isInfixOf)
import qualified Data.Set  as Set

import SecondParty.Module1 (fun)
import ThirdParty.Module1  (($$$))

import MyUtilsModule -- import everything
```

Pattern Matching

Besitzt eine Funktion mehrere Regeln, so sind dieselben Parameter jeweils untereinander anzuordnen:

```
and True True = True
and _    _    = False
```

Auch die Gleichheitszeichen sollen untereinander stehen.

Wächter (Guards)

Wächter können entweder direkt hinter dem Pattern Matching folgen oder werden in der folgenden Zeile eingerückt:

```
f x y z | x == y    = z
        | otherwise = z + 1

g x y z
  | x == y && not z = 1
  | otherwise       = 0
```

Bei langen Bedingungen kann auch das Gleichheitszeichen in die nächste Zeile rutschen:

```
f x y z
  | thisIsAVeryLongConditionWhichNeedsAllTheSpaceAvailableInTheLine x y z
  = 42
  | otherwise
  = 0
```

`If-then-else`-Ausdrücke

Grundsätzlich sollten Pattern Matching und Guards dem Schreiben von `if-then-else`-Ausdrücken vorgezogen werden. Kurze `if-then-else`-Ausdrücke können, sofern es die Zeilenlänge erlaubt, in einer Zeile notiert werden:

```
f (if x then 0 else 1) 42
```

Ansonsten sind die beiden Fälle einzurücken, wobei `then` und `else` untereinander stehen sollen:

```
foo = if ...
        then ...
        else ...
```

`case`-Ausdrücke

Die Alternativen von `case`-Ausdrücken können entweder ausgerichtet werden als

```
foobar = case something of
  Just j  -> foo
  Nothing -> bar
```

oder als

```
foobar = case something of
           Just j  -> foo
           Nothing -> bar
```

Die Pfeile `->` sollten ebenfalls untereinander stehen um die Lesbarkeit zu erhöhen.

`do`-Blöcke

Die Anweisungen in `do`-Blöcken können entweder direkt hinter dem `do` beginnen oder werden in der nächsten Zeile beginnend eingerückt:

```
echo = do name <- getLine
          putStrLn name
```

```haskell
greet = do
  putStr "How is your name? "
  name <- getLine
  putStrLn ("Hello " ++ name ++ "!")
```

Kommentare

Sprache und Satzbau

Kommentare sollten in Englisch geschrieben und korrekte Sätze sein. Sie sollen mit einem Großbuchstaben beginnen und auch Zeichensetzung ist nicht verboten.

Top-Level Definitionen

Alle Top-Level-Funktionen sollen kommentiert (insbesondere solche, die exportiert werden) und mit einer Typsignatur versehen werden. Für die Kommentare sollte die Haddock-Syntax verwendet werden.

```haskell
-- | Send a message on a socket.  The socket must be in a connected
-- state.  Returns the number of bytes sent.  Applications are
-- responsible for ensuring that all data has been sent.
send :: Socket      -- ^ Connected socket
     -> ByteString  -- ^ Data to send
     -> IO Int      -- ^ Bytes sent
```

Funktionskommentare sollten soviel Informationen enthalten, dass man sie benutzen kann, ohne die Implementierung anzuschauen.

Datentypen sind entsprechend ähnlich zu formatieren, bei Records mit expliziten Selektoren (Labels) sind auch diese zu kommentieren:

```haskell
-- |A natural person
data Person = Person
  { firstName :: String  -- ^ First name
  , lastName  :: String  -- ^ Last name
  , age       :: Int     -- ^ Age
  } deriving (Eq, Show)
```

Längere Kommentare für Felder werden über dem Feld angegeben:

```haskell
data Record = Record
  { -- | This is a very very very long comment that is split over
    -- multiple lines.
    field1 :: String
```

```
    -- | This is a second very very very long comment that is split
    -- over multiple lines.
  , field2 :: Int
  }
```

Kommentare am Zeilenende

Kommentare am Zeilenende sollten durch 2 Leerzeichen abgesetzt sein:

```
foo :: Int -> Int
foo n = salt * 32 + 9
  where salt = 453645243  -- Magic hash salt.
```

Namensgebung

Bei Namen, die aus mehreren logischen Teilen bestehen (z.B. mehrere Worte), werden diese Teile durch Binnenmajuskel (Camel-Case) aneinandergefügt:

```
thisIsTheAnswer = 42

data BankAccount = ...
```

Parameternamen werden in Haskell üblicherweise sehr kurz gehalten, wobei man sich an die folgende Daumenregel halten sollte:

> Je größer der Sichtbarkeitsbereich einer Variablen, desto länger der Name.

`x` kann also gerne in einer einzeiligen Funktion verwendet werden, als Top-Level-Funktion ist der Bezeichner hingegen ungeeignet.

Darüber hinaus gibt es einige Konvention für kurze Parameternamen:

- `f` bezeichnet oftmals eine Funktion vom Typ `a -> b`
- `n`, `m`, usw. stehen oft für natürliche Zahlen vom Typ `Int` bzw. `Integer`
- `x`, `y`, `z` stehen oft für Werte polymorpher Typen
- Ein angehängtes `s` deutet oft eine Liste an. Einige Beispiele sind: `xs :: [a]`, `fs :: [a -> b]`, `xss :: [[a]]`
- `p` steht oft für ein Prädikat vom Typ `a -> Bool`
- Angehängte Häkchen wie `x'` stehen oft für veränderte oder aktualisierte Werte, z.B. bei einem Akkumulator:

  ```
  let acc' = updateAcc acc in ...
  ```

- `k` und `v` werden gerne benutzt um Schlüssel bzw. Wert in einer Zuordnung zu bezeichen (für *key* bzw. *value*).

Compiler-Warnungen

Jeglicher Code soll mit der Compiler-Option `-Wall` ohne Warnungen kompiliert werden können.