Fortgeschrittene Funktionale Programmierung

Teil 5: Records in Haskell

Steffen Jost

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

30. Mai 2017

Teil 5: Records in Haskell

Record-Syntax

Konstruktoren mit vielen Typ-gleichen Argumenten werden schnell unübersichtlich:

```
data Person' = Person'  String Int Int Int Int
p0 = Person' "Tyrion" 135 26 7 0

data Person  = Person { name::String, height,age,mates,offspring::Int }
p2 = Person  {height=166, age=35, offspring=3, mates=3, name="Cersei"}
p3 = Person  "Jaimie" 187 35 1 3      -- alte Syntax auch noch erlaubt
```

Record-Syntax erlaubt es, die *Argumente eines Konstruktors* zu benennen; diese werden dann auch als **Felder** bezeichnet.

- Reihenfolge der Felder innerhalb geschweifter Klammern ist immer beliebig
- Datentypen können nachträglich zu Records gemacht werden: Werden keine geschweiften Klammern verwendet, gilt die Reihenfolge wie in der Definition, wie üblich.

Pattern Matching mit Records

Pattern Matching darf Record Syntax verwenden;
die Reihenfolge der Felder ist beliebig:

```
data Person = Person { name::String,
                       height,age,mates,offspring::Int}

showPerson :: Person -> String
showPerson Person { age=a, name=n } = n ++ ' ':show a

hasChildren :: Person -> Bool
hasChildren Person { offspring=n } | n > 0 = True
hasChildren _ = False

> showPerson p2
"Cersei 35"
```

Das Matching darf auch partiell sein, d.h. es müssen nicht alle Felder gemachted werden.

Pattern Matching mit Records

Pattern Matching darf Record Syntax verwenden;
die Reihenfolge der Felder ist beliebig:

```
data Person = Person { name::String,
                       height,age,mates,offspring::Int}

showPerson :: Person -> String
showPerson Person { age=a, name=n } = n ++ ' ':show a

hasChildren :: Person -> Bool
hasChildren Person { offspring=n } = n > 0
```

```
> showPerson p2
"Cersei 35"
```

Das Matching darf auch partiell sein, d.h. es müssen nicht alle Felder gemachted werden.

Record Projektionen

Projektionen (engl. selector functions) werden für jeden Feldnamen automatisch definiert:

```
data Person = Person { name::String,
                       height,age,mates,offspring::Int}
> :t name
name :: Person -> String

> name p3
"Jaimie"

> :t age
age :: Person -> Int

> age p3
35
```

Projektionen und newtype

Die automatisch definierten Projektionen nutzt man gerne bei newtype Deklarationen aus, da in diesem Fall eine Projektion fast immer nützlich ist, um den Wrapper-Konstruktor zu entfernen:

```
newtype ZipList a = ZipList { getZipList :: [a] }

> :t getZipList
getZipList :: ZipList a -> [a]


> getZipList $ ZipList [(*2),(+2)] <*> ZipList [5,7]
[10,9]

> ZipList [(*2),(+2)] <*> ZipList [5,7]
ZipList {getZipList = [10,9]}
```

Record Pseudoupdate

```
data Person = Person { name::String,
                       height,age,mates,offspring::Int}
p1 = Person "Tyrion" 135 26 7 0
```

Funktionale "Field-updates" sind ebenfalls möglich. Dabei werden natürlich Kopien erstellt — denn bestehende Werte werden in der funktionalen Welt ja nie verändert!

```
p4 = p1 { name = "Imp" }
p5 = p1 { mates = 2 + mates p1 }

> p5
Person {name = "Tyrion", height = 135, age = 26,
        mates = 9, offspring = 0}
> p1
Person {name = "Tyrion", height = 135, age = 26,
        mates = 7, offspring = 0}
```

Record Refactoring

Records kann man bei Bedarf nachträglich leicht erweitern:

```
data Person = Person { name::String,
                       height,age,mates,offspring::Int}

p6 = Person { name="Daenerys", height=157, offspring=0 }
```

ist gültiger Code (wenn z.B. `age` und `mates` nachträglich in die Definition aufgenommen wurden). Es wird interpretiert als:

```
p6 = Person "Daenerys" 157 undefined undefined 0
```

GHC kann in diesem Fall Warnungen ausgeben, dass nicht alle Felder eines Records initialisiert wurden.

Diese Warnungen sollte man nach einer solchen nachträglich Erweiterung des Record-Typs aber auch abarbeiten!

Standardwerte für Records definieren

Um die Wartbarkeit zu erhöhen ist es oft ratsam, einen globalen Record mit Default-Werten anzulegen:

```
data Bar = Bar { bar :: Int, baz :: Int, quux :: Int }

barDefault = Bar { bar = 1, baz = 2, quux = 3 }

myBar = barDefault { bar = 69, baz = 42 } -- quux == 3
```

Ansatt einen neuen Record anzulegen, wird lediglich ein Field-Update des Standardwerts durchgeführt

Fügt man später ein weiteres Feld ein, muss man lediglich den Default-Wert anpassen.
Nachteil: Der Kompiler gibt dann keine Warnungen mehr heraus, wenn nicht alle Felder *korrekt* initialisiert werden.

Records und Summentypen

Record-Syntax kann auch mit Alternativen verwendet werden:

```
data UniP = Student { name::String, matrikel::Int }
          | Dozent  { name::String, titel::String }

u1 = Dozent  { name="Steffen", titel="Dr" }
u2 = Student { name="Max",  matrikel=666  }

isDozent :: UniP -> Bool
isDozent Dozent {} = True
isDozent    _      = False

isSteffen :: UniP -> Bool
isSteffen p = "Steffen" == name p
```

Partielle Projektionen

Record-Syntax kann auch mit Alternativen verwendet werden:

```
data UniP = Student { name::String, matrikel::Int }
          | Dozent  { name::String, titel::String }
```

Achtung Automatisch definierte Projektionen werden zu partiellen Funktionen, falls nicht jeder Konstruktor alle Felder enthält

```
u1 = Dozent  { name="Steffen", titel="Dr" }
u2 = Student { name="Max",  matrikel=666  }

> titel u1
"Dr"
> titel u2
"*** Exception: No match in record selector titel"
```

Records und Summentypen

Schlecht

```
data Obst = Apfel { preis,gewicht :: Double }
          | Birne { preis,gewicht :: Double }
```

Schlechter Stil, wenn jeder Konstruktor alle Felder hat, da dies zu unnötig vielen Fallunterscheidungen mit dupliziertem Code führt.

Besser

```
data Frucht = Apfel | Birne
data Obst   = Obst { preis,gewicht :: Double
                   , frucht :: Frucht }
```

Dank verschachtelter Patterns (ggf. Guards) verliert man nichts:

```
istTeurerApfel :: Obst -> Bool
istTeurerApfel Obst {frucht=Apfel,preis=p} = p > 7
istTeurerApfel _ = False
```

DisambiguateRecordFields

Feldnamen sollten innerhalb eines Moduls eindeutig sein; ansonsten überdeckt die letzte Definition eine vorangegangene Definition. Spracherweiterung `DisambiguateRecordFields` erlaubt Verwendung identischer Feldnamen in Situationen, welche aufgrund der Typisierung eindeutig sind.

```
module M where
  data S = MkS { x :: Int, y :: Bool }

module Foo where
  import M

  data T = MkT { x :: Int }

  ok1 (MkS { x = n }) = n+1    -- Unambiguous
  ok2 n = MkT { x = n+1 }      -- Unambiguous

  bad1 k = k { x = 3 }         -- Ambiguous
  bad2 k = x k                 -- Ambiguous
```

Dies ist besonders dann nützlich, wenn verschiedene Module verwendet werden, welche die gleichen Feldnamen definieren.

DuplicateRecordFields

Spracherweiterung `DuplicateRecordFields` erlaubt identische Feldnamen für verschiedene Records innerhalb eines Moduls. Hierfür findet nur eine sehr eingeschränkte Typinferenz statt:

```
data Foo = Foo { x :: Int, y :: Bool }
data Bar = Bar { z :: Int, y :: Double }

f :: Bar -> Double
f x = y (x :: Bar)   -- okay

g :: Bar -> Double
g x = y x            -- not accepted
```

Hinweis Subtyping zwischen Record-Typen gibt es in Haskell grundsätzlich nicht. Auch wenn ein Typ strikt mehr Felder besitzt als ein anderer, so bleiben es voneinander verschiedene Typen.

NamedFieldPuns

Es ist üblich, lokale Variablen in Pattern-Matches identisch zum Feld des Matchings zu benennen:

```
data Bar = Bar {a,b,c,d,e :: Int}
foo :: Bar -> Int
foo (Bar {a=a, b=b, e=e}) = a+b+e
```

GHC Spracherweiterung `NamedFieldPuns` vermeidet die dabei auftretenden Wiederholungen:

```
{-# LANGUAGE NamedFieldPuns #-}
foo (Bar {a, b, e}) = a+b+e
```

Ist eine Variable im Scope, deren Namen identisch zu einem Feldnamen ist, darf man diese auch zur Konstruktion verwenden:

```
let c = 1 in Bar {c}
-- anstatt
let c = 1 in Bar {c = c}
```

RecordWildCards

Wem das immer noch zu viel Tipparbeit ist, schreibt nur noch `..`
dank der Spracherweiterung `RecordWildCards`:

```
{-# LANGUAGE RecordWildCards #-}
data Bar = Bar {a,b,c,d,e :: Int}

foo :: Bar -> Int
foo Bar { a = 1, .. }) = b + c + d + e

bar :: Bar
bar = let { a=1; b=2; c=3; d=4; e=5 } in Bar {..}
```

Anstatt dem längerem äquivalenten Code:

```
foo (Bar { a=1, b=b, c=c, d=d, e=e }) = b + c + d + e

bar = let { a=1; b=2; c=3; d=4; e=5 }
      in Bar { a=a, b=b, c=c, d=d, e=e }
```

RecordWildCards – Besonderheiten

Fallstrick: Fehlt eine Variable mit passenden Feldnamen im Sichtbarkeitsbereich, dann bleibt dieses Feld einfach undefiniert!

```
data Bar = Bar {a,b,c,d,e :: Int}

foo1 a c e = Bar {..}
foo2 a c e = Bar {a=a, c=c, e=e}
foo3 a c e = Bar {a=a, b=undefined, c=c, d=undefined,e=e}
```

Alle drei Definition sind für GHC äquivalent; aber für einen Menschen vielleicht nicht! $\Rightarrow$ Lesbarkeit kann dadurch leiden!

```
> c $ foo1 0 1 2
c
> b $ foo1 0 1 2
*** Exception: RecordWildCards.hs:12:13-20: Missing field
```

Record-Update mit WildCards ist dagegen gar nicht erlaubt.

Zusammenfassung Record-Syntax

Record-Syntax...

- ...erlaubt die Benennung von Konstruktor-Argumenten
- ...erlaubt, momentan unbenötigte Konstruktor-Argumente auszublenden, anstatt immer alle aufzuzählen zu müssen
- ...definiert automatisch (partielle) Projektionen
- ...bietet funktionale Updates durch Kopieren an
- ...kann die Wartbarkeit von Code erhöhen

Records sind in Haskell keine besondere Datenstruktur; es ist lediglich eine notationelle Vereinfachung!

Dementsprechend kennt Haskell auch kein Record-Subtyping