Typen und Typklassen

Funktionale Programmierung

Prof. Dr. Oliver Braun

Letzte Änderung: 09.10.2018 06:54

Typensignaturen

- jeder Ausdruck in Haskell hat einen Typ
- manche Ausdrücke haben einen konkreten Typ, z.B.

```
't' :: Char
"fun" :: [Char]
True :: Bool
```

- manche Ausdrücke haben einen polymorphen Typ, z.B.

```
13 :: Num a ⇒ a
```

- einen polymorphen Typ können wir einschränken, z.B.

```
x = 13 :: Integer
```

- Funktionen haben auch einen Typ, z.B.

```
not :: Bool → Bool
```

Der Funktionstyp

- auch Funktionen sind Daten

```
data (→) a b
```

  d.h. $(\rightarrow)$ ist ein Typkonstruktor

- Funktionsapplikation ist linksassoziativ, aber der Typkonstruktor $(\rightarrow)$ ist rechtsassoziativ

```
infixr 0 `(→)`
```

Typconstraints

- Typvariablen können durch Typconstraints eingeschränkt werden, z.B.

```
(+) :: Num a ⇒ a → a → a
(/) :: Fractional a ⇒ a → a → a
```

- mehrere Constraints werden geklammert

```
(Num a, Num b) ⇒ a → b → b
(Ord a, Num a) ⇒ a → a → Ordering
```

Typconstraints und konkrete Typen

- durch Angabe eines konkreten Typs können wir den Typ weiter einschränken

```
fifteen = 15 -- :: Num a ⇒ a
fifteenInt = fifteen :: Int
fifteenDouble = fifteen :: Double
```

- in Haskell gibt es keine implizite Umwandlung, aber im Beispiel ist `fifteen` ja polymorph

```
fifteenDouble + fifteen -- ok
fifteenInt + fifteen    -- ok
```

- aber

```
fifteenInt + fifteenDouble -- Typfehler
```

Currying

- wie im λ-Kalkül nimmt jede Haskell-Funktion maximal ein Argument
- trotzdem definieren wir Funktionen so als ob sie mehr als einen Parameter hätten, z.B.

```
add x y = x + y
```

  - der Typ von `add` ist

```
add :: Num a ⇒ a → a → a
-- bzw. da rechtsassoziativ
add :: Num a ⇒ a → (a → a)
```

- die Definition von `add` ist *syntactic sugar* und heisst *currysiert*
- Currying ist das Verschachteln mehrerer Funktionen mit einem Parameter und erzeugt die Illusion, dass die definierte Funktion mehrere Parameter hätte

Partielle Anwendung

- nachdem

```
add :: Num a ⇒ a → a → a
add x y = x + y
```

  eine currysierte Funktion ist, können wir Sie natürlich auch nur auf ein Argument anwenden, z.B.

```
addFive :: Num a ⇒ a → a
addFive = add 5
```

Funktionen currying und uncurrying

- eine nicht currysierte Funktion bekommt alle Argumente in ein Tupel verpackt, z.B.

```
addUncurry :: Num a ⇒ (a, a) → a
addUncurry (x,y) = x + y
```

- in der Prelude gibt es zwei Funktionen um currysierte in nicht currysierte Funktionen umzuwandlen und umgekehrt:

```
curry :: ((a, b) → c) → a → b → c
uncurry :: (a → b → c) → (a, b) → c
```

- so können wir beispielsweise definieren

```
addSeven = curry addUncurry 7
```

Sectioning

- Sectioning ist partielle Anwendung von Operatoren
- muss in runden Klammern geschrieben werden
- ein Operand wird einfach weg gelassen
- Beispiele:

```haskell
(^2)
(2<)
(++"!!!")
isBetween2And12 = (`elem` [2..12])
```

- das basiert übrigens auf der η-Konversion aus dem λ-Kalkül

$$\lambda x.(elem[2..12])x \rightsquigarrow (elem[1..12])$$

Polymorphismus

- Haskell hat zwei Arten von Polymorphismus
  - **parametric polymorphism**, auch *ad-hoc polymorphism*, z.B.

    ```
    id :: a → a
    ```

  - **constrained polymorphism**, über Typklassen realisiert, z.B.

    ```
    elem :: Eq a ⇒ a → [a] → Bool
    ```

- in Haskell können auch Literale polymorph sein, z.B.

```
42 :: Num a ⇒ a
```

Brauche **Double**, biete **Int**

- Keine implizite, aber explizite Umwandlung mit polymorphen Funktionen, z.B.

```
fromIntegral :: (Num b, Integral a) ⇒ a → b
```

- Beispiel

```
13 / fromIntegral (length [1..8])
```

```
13 :: Num a ⇒ a
length [1..8] :: Int
(/) :: Fractional a ⇒ a → a → a
```

Typinferenz

- Haskell kann den Typ selbst berechnen/inferieren
- Haskell nutzt dazu eine Erweiterung des Damas-Hindley-Milner-Typsystems
- Haskell inferiert den generellsten (polymorphen) Typ für einen Ausdruck
- der Compiler beginnt mit Ausdrücken deren Typ er kennt und inferiert damit die Typen der anderen Ausdrücke

Typinferenz — Beispiel

```
add x y = x + y
```

- der Compiler weiß bereits

```
(+) :: Num a ⇒ a → a → a
```

- damit muss gelten

```
x :: Num a ⇒ a
y :: Num a ⇒ a
x + y :: Num a ⇒ a
```

- damit kann der Compiler inferieren

```
add :: Num a ⇒ a → a → a
```

- durch Angabe eines Typs können wir **add** einschränken, z.B.

```
add :: Integer → Integer → Integer
```

Typklassen

- Typklassen fassen gleichartige Typen zusammen, z.B.
  - `Num` — Zahltypen oder
  - `Eq` — Typen deren Werte verglichen werden können
- Konzept ähnlich *Interfaces* in anderen Programmiersprachen
- nützlich um typsicher Polymorphismus umzusetzen (*constrained polymorphism*)

Beispiel: Datentyp **Bool**

```
Prelude> :info Bool
data Bool = False | True
instance Bounded Bool
instance Enum Bool
instance Eq Bool
instance Ord Bool
instance Read Bool
instance Show Bool
```

Beispiel: Typklasse **Eq**

```
Prelude> :info Eq
class Eq a where
  (==) :: a → a → Bool
  (/=) :: a → a → Bool
instance Eq a ⇒ Eq [a]
instance Eq Ordering
instance Eq Int
 ...
```

- außerhalb der Typklasse:

```
(==) :: Eq a ⇒ a → a → Bool
(/=) :: Eq a ⇒ a → a → Bool
```

Beispiel: Datentyp Tupel

```haskell
data (,) a b = (,) a b
instance (Eq a, Eq b) ⇒ Eq (a, b)
instance (Ord a, Ord b) ⇒ Ord (a, b)
instance (Read a, Read b) ⇒ Read (a, b)
instance (Show a, Show b) ⇒ Show (a, b)
```

Typklassen herleiten lassen

- für viele Typklassen sind die Definitionen trivial
    - z.B. `Eq`, `Ord`, `Show`, `Read`, …
- daher enthält der GHC einen Mechanismus mit dem man die kanonische Definition herleiten lassen kann
    - engl. *typeclass deriving*

Instanzen von Typklassen

- um für einen Typ eine Instanz einer Typklasse zu definieren,
  - müssen die in der Typklasse enthaltenen Funktionen definiert werden
- für einige gibt es bereits *default*-Implementierungen, so dass nur Teile tatsächlich implementiert werden müssen,
  - z.B.

```haskell
class Eq a where
  (==) :: a → a → Bool
  (/=) :: a → a → Bool
  {-# MINIMAL (==) | (/=) #-}
```

  es reicht also `(==)` **oder** `(/=)` zu implementieren

Beispiel: `Trivial`

- eigener Datentyp

```
data Trivial = Trivial
```

- Vergleichen:

```
Trivial == Trivial
No instance for (Eq Trivial) arising
  from a use of '=='
In the expression: Trivial == Trivial
In an equation for 'it': it = Trivial == Trivial
```

- Typklassen-Instanz für `Trivial`

```
instance Eq Trivial where
  Trivial == Trivial = True
```

Beispiel: Wochentag

```haskell
data DayOfWeek =
  Mon | Tue | Weds | Thu | Fri | Sat | Sun

instance Eq DayOfWeek where
  Mon  == Mon  = True
  Tue  == Tue  = True
  Weds == Weds = True
  Thu  == Thu  = True
  Fri  == Fri  = True
  Sat  == Sat  = True
  Sun  == Sun  = True
  _    == _    = False
```

Beispiel: Wochentag mit Tag im Monat

```haskell
type DayOfMonth = Int
data Date = Date DayOfWeek DayOfMonth

instance Eq Date where
  (Date weekday dayOfMonth) ==
    (Date weekday' dayOfMonth')
      =     weekday == weekday'
        && dayOfMonth == dayOfMonth'
```

Instanzen herleiten

```haskell
data DayOfWeek =
  Mon | Tue | Weds | Thu | Fri | Sat | Sun
  deriving (Eq, Ord, Show)
```

Partielle Funktionen

- Funktionen, die nicht für alle Werte definiert sind
- Beispiel

```
f :: Int → Bool
f 2 = True
```

```
Prelude> f 3
** Exception: <interactive>:8:19-28:
    Non-exhaustive patterns in function f
```

Compiler, please help me

- lassen Sie sich warnen `-Wall`:

```
Prelude> :set -Wall
Prelude> f 2 = True


<interactive>:13:1: warning: [-Wincomplete-patterns]
    Pattern match(es) are non-exhaustive
    In an equation for ‘f’:
        Patterns not matched:
            p where p is not one of {2}
```

- und die totale Funktion:

```
f :: Int → Bool
f 2 = True
f _ = False
```

Instanzen für Typklassen mit Parametern (1/2)

- Typ

```haskell
data Identity a = Identity a
```

- erster Versuch

```haskell
instance Eq (Identity a) where
  (==) (Identity v) (Identity v') = v == v'
```

- aber …

Instanzen für Typklassen mit Parametern (2/2)

```
No instance for (Eq a) arising from a use of ‘==’
Possible fix: add (Eq a) to the
context of the instance declaration
In the expression: v == v'
In an equation for ‘==’:
  (==) (Identity v) (Identity v') = v == v'
In the instance declaration for ‘Eq (Identity a)’
```

- Lösung:

```
instance Eq a ⇒ Eq (Identity a) where
  (==) (Identity v) (Identity v') = v == v'
```

Typklasse **Enum**

```haskell
class Enum a where
  succ :: a → a
  pred :: a → a
  toEnum :: Int → a
  fromEnum :: a → Int
  enumFrom :: a → [a]
  enumFromThen :: a → a → [a]
  enumFromTo :: a → a → [a]
  enumFromThenTo :: a → a → a → [a]
  {-# MINIMAL toEnum, fromEnum #-}
```

- Syntactic sugar, z.B.
  - statt `enumFromThenTo 1 10 100`
  - auch `[1,10..100]`

Weitere Typklassen

- `Num`
- `Integral`
- `Fractional`
- `Ord`
- `Show`
- `Read`
- uvm.