Funktionale Programmierung
Funktionen und Typen II

1. Funktionen Reloaded
   - Funktionstyp
   - Mehrstellige Funktionen
   - Currying und höhere Funktionen

2. Partielle Funktionen und optionale Werte
   - Der Maybe Datentyp
   - Der Either Datentyp

Eine ("reine") Funktion stellt jeder Eingabe x genau eine Ausgabe $f(x)$ gegenüber. Eine reine Funktion hat keinerlei Nebeneffekte.

Eine reine Funktion:

```
f :: Integer -> Integer
f x = 3 * x
```

In Haskell sind Funktionen Werte vone einem Typ der Gestalt `a -> b`. In der Mengenanalogie ergibt sich folgende Entsprechung:

Ein Ausdruck von der Form

$$\mathit{Typ}_1 \to \mathit{Typ}_2$$

entspricht der Menge

$$\{f \mid f : \mathit{Typ}_1 \to \mathit{Typ}_2\}.$$

Der Ausdruck "`f:a -> b`" ist somit als "`f` ist eine Funktion von `a` nach `b`" zu lesen.

Der Pfeil -> wird rechtsassoziativ gelesen.

Der Typ

```
a -> b -> c -> d
```

ist beispielsweise als

```
a -> (b -> (c -> d))
```

zu interpretieren.

Aufgabe

Gegeben sind folgende (teilweise) implizit geklammerte Typen:

- `String -> Int -> String`
- `a -> (b -> c) -> d`
- `(a -> b) -> c -> d`

Schreiben Sie die Klammerung explizit auf.

Aus einer Funktionsdefinition (und dem Kontext) lässt sich der Typ der Fubnktion oft ableiten. Wenn der Typ einer Funktion automatisch abgeleitet wird, spricht man von Typinferenz ("type inference").

```
Prelude> c f g x = g (f x)
Prelude> :t c
c :: (t1 -> t2) -> (t2 -> t3) -> t1 -> t3
Prelude>
```

Man kann den Typ einer Funktion auch “von Hand” bestimmen.

Beispiel

```
1  a b c d e = c (b d) (b e)
```

- `d` und `e` haben irgendwelche Typen: `d:D` und `e:E`
- Weil wir `b` sowohl auf `d` sowie auch auf `e` anwenden, ist `E = D` und der Typ von `b` von der Form `b:D->B`.
- Weil wir `c` auf zwei Argumente vom Typ `B` anwenden, ist der Typ von `c` von der Form `c:B -> B -> C` (insbesondere ist der Term `a b c d e` vom Typ `C`).
- Für den Term `a` erhalten wir daher durch Einsetzen den Typ `a:  (D -> B) -> (B -> B -> C) -> D -> D -> C`

Formal werden Typensysteme oft via einem formalen System und dazugehörigen Typisierungsregeln spezifiziert. Ein sehr einfaches solches System ist der “Simply typed lambda calculus” (`en.wikipedia.org/wiki/Simply_typed_lambda_calculus`)

Aufgabe

Versuchen Sie (möglichst allgemeine) Typen für folgenden Ausdrücke von Hand zu bestimmen. Gehen Sie möglichst methodisch vor.

```haskell
a1 b = b (17 :: Int)
a2 b = b 15
a3 b c = c b
x y z = y (z y)
```

Definition

Funktionen deren Eingabe aus mehreren Argumenten besteht, nennt man mehrstellige Funktionen.

Beispiele mehrstelliger Funktion:

$$add(x, y) = x + y$$

$$max(x, y, z) = \begin{cases} x & \text{falls } x \geq y, z \\ y & \text{falls } y \geq x, z \\ z & \text{sonst} \end{cases}$$

$$avg(x_1, \ldots, x_n) = \frac{1}{n} \sum_{i=1}^{n} x_i$$

Zwei Sichtweisen auf mehrstellige Funktionen:

[1] Wegen der Rechtsassoziativität werden die Klammern typischerweise nicht geschrieben.

Zwei Sichtweisen auf mehrstellige Funktionen:

- In der ersten Variante versteht man eine *n*-stellige Funktion als Funktion, die *n*-Tupel als Eingabewerte akzeptiert. Eine mehrstellige Funktion nach dieser Auffassung hat dann einen Typ von der Form:

```
(a1,...,an) -> b
```

[1] Wegen der Rechtsassoziativität werden die Klammern typischerweise nicht geschrieben.

Zwei Sichtweisen auf mehrstellige Funktionen:

- In der ersten Variante versteht man eine n-stellige Funktion als Funktion, die n-Tupel als Eingabewerte akzeptiert. Eine mehrstellige Funktion nach dieser Auffassung hat dann einen Typ von der Form:

```
(a1,...,an) -> b
```

- In der zweiten Variante versteht man eine n-Stellige Funktion als Funktion, die $n-1$-stellige Funktionen zurückgibt. Eine mehrstellige Funktion nach dieser Auffassung hat dann einen Typ von der Form[1]:

```
a1 -> (a2 -> ... an)..)
```

[1] Wegen der Rechtsassoziativität werden die Klammern typischerweise nicht geschrieben.

Beide Sichtweisen auf mehrstellige Funktionen werden in Haskell direkt unterstützt:

```haskell
max1 :: (Int, Int) -> Int
max1 (x, y) = if x > y then x else y

max2 :: Int -> Int -> Int
max2 x y = if x > y then x else y
```

Unter Currying und Uncurrying versteht man das Übersetzen zwischen den genannten Ansätzen. Informell lassen sich diese Übersetzungen wie

folgt als Funktionen beschreiben:

```
curry f a1 .. an = f (a1,..,an)
uncurry f (a1,.., an) = f a1 .. an
```

Aufgabe

Implementieren Sie Currying

```
curry::  ((a,b) -> c) -> a -> b -> c
```

und Uncurrying

```
uncurry::  (a -> b -> c) -> (a,b) -> c
```

für zweistellige Funktionen.

Partielle Anwendung bedeutet eine “curried” Funktion nicht “erschöpfend” mit Argumenten zu versehen:

```
plus4 :: Num a => a -> a
plus4 = (+) 4
```

Partielle Anwendung bedeutet eine “curried” Funktion nicht “erschöpfend” mit Argumenten zu versehen:

```
plus4 :: Num a => a -> a
plus4 = (+) 4
```

Bemerkung

- *Partielle Anwendung (partial application) ist die ganz normale Anwendung (in der zweiten Sichtweise) von mehrstellige Funktionen.*

Partielle Anwendung bedeutet eine “curried” Funktion nicht “erschöpfend” mit Argumenten zu versehen:

```
plus4 :: Num a => a -> a
plus4 = (+) 4
```

Bemerkung

- *Partielle Anwendung (partial application) ist die ganz normale Anwendung (in der zweiten Sichtweise) von mehrstellige Funktionen.*
- *Partielle Anwendung hat nichts mit partiellen Funktionen zu tun.*

Partielle Anwendung bedeutet eine “curried” Funktion nicht “erschöpfend” mit Argumenten zu versehen:

```haskell
plus4 :: Num a => a -> a
plus4 = (+) 4
```

Bemerkung

- *Partielle Anwendung (partial application) ist die ganz normale Anwendung (in der zweiten Sichtweise) von mehrstellige Funktionen.*
- *Partielle Anwendung hat nichts mit partiellen Funktionen zu tun.*
- *Partielle Anwendung ist ein Mittel der funktionalen Programmierung um “generischen” Code zu erzeugen.*

Definition

Eine Funktion höherer Ordnung[2], ist eine Funktion, die Funktionen als Argumente erhält oder Funktionen als Ergebnis liefert.

[2] Engl. higher-order function. Manchmal auch Kombinator oder Funktional.

Die Möglichkeit Funktionen höherer Ordnung zu deklarieren und daraus via Komposition und partieller Anwendung weitere Funktionen zusammenzubauen, konstituiert ein Mittel zur Abstraktion. Wir betrachten im Folgenden einige Beispiele.

Beispiel

Eine höhere Funktion zum doppelten Anwenden einer gegebenen Funktion:

```
twice :: (a -> a) -> a -> a
twice f x = f $ f x
```

Beispiel

Das klassische Beispiel einer höheren Funktion ist die Komposition:

```
(.) :: (b -> c) -> (a -> b) -> a -> c
(.) f g x = f $ g x
```

Sie kann beispielsweise dazu verwendet werden, eine (höhere) Funktion zum doppelten Anwenden einer gegebenen Funktion umzusetzen:

```
twice :: (a -> a) -> a -> a
twice f = f . f
```

Aufgabe

Implementieren Sie eine Funktion für *n*-maliges Anwenden einer gegebenen Funktion.

```
many :: Int -> (a -> a) -> a -> a
```

Was fällt Ihnen auf?

```haskell
sum' [] = 0
sum' (x:xs) = x + sum' xs

prod' [] = 1
prod' (x:xs) = x * prod' xs

sumSq [] = 0
sumSq (x:xs) = x * x + sumSq xs

prodEvens [] = 1
prodEvens (x:xs)
    | x `mod` 2 ==0 = x * prodEvens xs
    | otherwise = prodEvens xs
```

```haskell
sumF xs = foldl (+) 0 xs

prodF xs = foldl (*) 1 xs

sumSqF xs = foldl (\acc y -> acc + y * y) 0 xs

prodEvensF xs = foldl f 1 xs
where
    f a x
        | x `mod` 2 == 0 = a * x
        | otherwise = a
```

Neben den verschiedenen “fold” funktionen gibt es weitere (oft Instanzen von folds) höhere Funktionen speziell für Listen (foldl, foldr, map, filter,...).

Für eigene Typen (z.B. `T`) lassen sich sehr allgemein "folds" implentieren indem man wie folgt vorgeht:

1. Analysieren der Signaturen von Konstruktoren des Typs.
2. Alle Vorkommen von `T` ersetzen durch einen Typparameter (z.B. `b`).
3. Eine Funktion erstellen, die für jeden Konstruktor des Types ein Argument akzeptiert, das den wie Oben beschriebenen modifizierten Typ hat.

Wir erhalten dadurch eine "generische" Funktion um aus Werten vom Typ `T` Werte vom Typ `b` zu bauen.

Beispiel

Der Typ:

```
data BTree a
    = Node a (BTree a) (BTree a)
    | Empty
```

Beispiel

Die Signaturen der Konstruktoren:

```
Node :: a -> BTree a -> BTree a -> BTree a
Empty :: BTree a
```

Die modifizierten Typen:

```
Node :: a -> b -> b -> b
Empty :: b
```

Beispiel

Die Funktion

```haskell
bTree
    :: (a -> b -> b -> b)
    -> b
    -> BTree a
    -> b
bTree _ empty Empty = empty
bTree node empty (Node a t1 t2) = node a (recurse t1) (recurse t2)
    where
        recurse = bTree node empty
```

Beispiel

Anwendung: Bäume in Latex anzeigen:

```
btTex :: Show a => BTree a -> String
btTex = bTree (\a b c ->
    "[" ++ (show a) ++ b ++ c ++ "]") ""
```

Aufgabe

Implementieren Sie eine Funktion zum berechnen der Tiefe eines Baumes:

```
btDepth :: BTree a -> Integer
btDepth = bTree ? ?
```

Partielle Funktionen und optionale Werte

Definition

Eine partielle Funktion $f : X \hookrightarrow Y$ ist eine Funktion $f : X' \to Y$, wobei $X' \subseteq X$.

Partielle Funktionen und optionale Werte

Definition

Eine partielle Funktion $f : X \hookrightarrow Y$ ist eine Funktion $f : X' \to Y$, wobei $X' \subseteq X$.

- Eine partielle Funktion gibt (eventuell) für gewisse Eingaben keinen Funktionswert zurück.

Partielle Funktionen und optionale Werte

Definition

Eine partielle Funktion $f : X \hookrightarrow Y$ ist eine Funktion $f : X' \to Y$, wobei $X' \subseteq X$.

- Eine partielle Funktion gibt (eventuell) für gewisse Eingaben keinen Funktionswert zurück.
- Partielle Funktionen sind in der Mathematik und insbesondere in der Informatik häufig. Beispiel $f(x, y) = \frac{x}{y}$.

Viele funktionale Programmiersprachen stellen einen Typ zur Modellierung von optionalen/partiellen Rückgabewerten bereit[3]. Der Maybe Typ von Haskell ist ein einfacher Summentyp:

```
data Maybe a = Just a | Nothing
```

[3] F#,Scala: Option, Haskell: Maybe, ..

Mit den Maybe Typ können partielle Funktionen explizit modelliert werden.

```
f :: Fractional a => a -> a -> Maybe a
f x y = case y of
    0 -> Nothing
    _ -> Just (x / y)
```

Der Maybe Typ im Vergleich zu `Null`-Referenzen:

- Durch Verwendung eines Maybe Typs wird auf Typ-Ebene explizit gemacht, dass eine gegebene Funktion partiell ist. Im Gegensatz dazu, ist der Unterschied zwischen einer `Null`-Referenz und einem validen Objekt für das Typensystem "unsichtbar".
- Die Komponierbarkeit bleibt dank verschiedener Hilfsfunktionen erhalten (vgl. Code).

Um mehr Information zu übergeben, wieso eine Funktion an einer bestimmten Stelle keinen (geeigneten) Rückgabewert liefert (z.B. Error handling) gibt es den `Either a b` Typ. Alternativ kann natürlich immer auch ein massgeschneiderter Datentyp (und die dazu passenden Instanzen) implementiert werden.

```
data Either a b
    = Left a   -- Fehler mit Information a
    | Right b  -- Das gute Resultat

data Result a b c =
    = Success a
    | XError b
    | YError c
```