Funktionale Programmierung

Funktionen und Typen I

1. Referenzielle Transparenz
   - Werte und Variablen
   - Nebeneffekte
   - Referenzielle Transparenz
2. Typen
   - Grundtypen in Haskell
   - Zusammmengesetzte Typen - Listentypen
   - Zusammengesetzte Typen - Funktionstypen
   - Zusammengesetzte Typen - Tupel
   - Zusammengesetzte Typen - Records
   - Zusammengesetzte Typen - Summen
   - Typklassen

Referenzielle Transparenz

Werte und Variablen

In der funktionalen Programmierung haben "Variablen", und damit Wertzuweisungen eine **fundamental** andere Bedeutung als in der imperativen Programmierung.

Die Zuweisung `x = 3` im Vergleich:

- Funktional: Der "Name" `x` benennt (in seinem Kontext), unabhängig von der Zeit, den **Wert** 3.
- Imperativ: Der "Name" `x` benennt einen **Ort** (Speicherbereich). Sein Wert ändert sich mit der Zeit, je nachdem was in besagtem Speicherbereich steht.

Konsequenzen:

- Die Wert-Variable-Relation ist im funktionalen Paradigma zeitunabhängig.
- Variablen im funktionalen Paradigma entsprechen eher "Konstanten" als Variablen $\rightarrow$ Stichwort "immutability".

Neben dem Verzicht Variablen zu verändern, wird in der funktionalen Programmierung darauf geachtet auch andere Nebeneffekte möglichst zu vermeiden oder mindestens zu isolieren.

"Interne" Nebeneffekte ändern den Zustand des Programms und kommen in rein funktionalen Sprachen nicht vor.

```
1 int square(int x){
2     if (cntr < 3) {
3         cntr++;
4         return x * x;
5         };
6     else failwith("enough is enough");
7 }
```

"Externe" Nebeneffekte verändern den Zustand des Kontextes (Aussenwelt) in den das Programm eingebettet ist und werden vom Rest des Programmes isoliert.

```
1 f n = "Blink LED n times"
```

oder

```
1 greet name = printLn "Hello" ++ name
```

Solche Nebeneffekte werden in funktionalen Sprachen möglichst gut isoliert und explizit gekennzeichnet.

Referenzielle Transparenz

Eigenschaften von Nebeneffektfreiem (reinem) funktionalen Code:

- Eine "Variable" kann in einem Ausdruck immer[1] durch ihren Wert ersetz werden. Der Wert (Bedeutung) des betreffenden Ausdrucks (Programmes) verändert sich dadurch nicht.
- Der Wert eines Ausdrucks hängt nur von den Werten seiner Teilausdrücke ab.
- Die Evaluation eines Ausdruckes ist unabhängig von Reihenfolge, in der seine Teilausdrücke evaluiert werden.

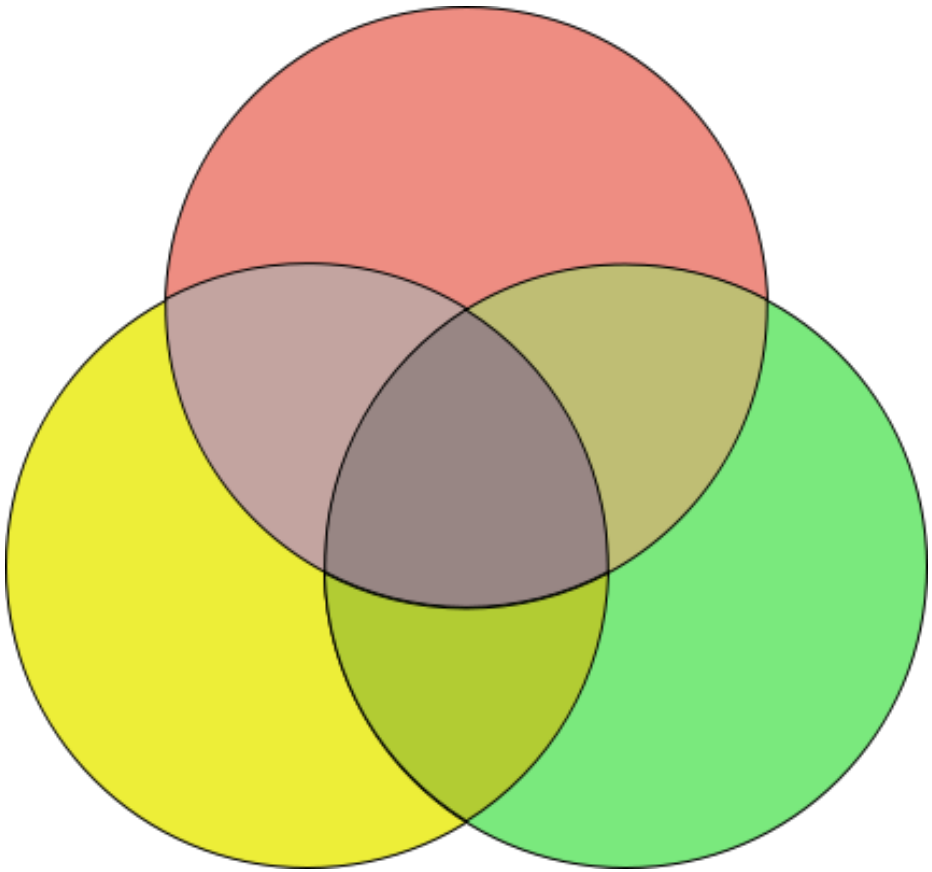
Diese Eigenschaften werden unter dem Begriff der referenziellen Transparenz zusammengefasst. Imperativer Code ist (im allgemeinen) nicht referenziell transparent.

[1] Unter berücksichtigung ihres Kontextes!

Vorteile, die sich aus referenzieller Transparenz ergeben:

- Mehr Flexibilität beim Auswerten von Ausdrücken (z.B. “lazy evaluation”).
- Einfachere Programmverifikation (weniger und einfachere/explizitere Abhängigkeiten).
- Erleichtert die Beweisführung, z.B. dass gewisse Optimierungen die Bedeutung eines transformierten Programmes nicht verändern.
- Erleichtert es Programme zu verstehen und zu entwerfen (“equational reasoning”).

Typen



Darüber was genau ein Typ sei gibt es viele[2] Meinungen:

> *D. L. Parnas, J. E. Shore and David Weiss identified five definitions of a "type" that were used - sometimes implicitly - in the literature:. . .*
> *Wikipedia*

[2] Offenbar mindestens fünf

Einige Meinungen sind etwas ausführlicher:

> *A type system is a tractable syntactic method of proving the absence of certain program behaviors by classifying phrases according to the kinds of values they compute.*
> *Benjamin Pierce, Types and Programming Languages*

Wir wollen für den Moment mit einer einfachen Analogie arbeiten:

- Typen $\simeq$ Mengen
- Anstelle von $x \in t$ schreibt man $x : t$ ($x :: t$ in Haskell
- Sie dienen dazu eine "minimale Konsistenz" von Programmen zur Kompilierzeit sicherzustellen.

Typensysteme werden meistens nach folgendem Schema spezifiziert:

- Angabe einer Menge von “primitiven”- oder “Grundtypen” (z.B. `Int`, `Char`, `Float`, etc.)
- Syntaktische Elemente, die es erlauben aus bestehenden Typen neue Typen zu bauen.
- Algorithmen, die überprüfen ob ein Programm zu einem gegebenen Typ passt (type-checker) oder sogar einen passenden Typ herleiten (type-inference).

Eine Auswahl an Grundtypen (String und Bool sind streng genommen keine Grundtypen sondern Zusammengesetzt) in Haskell:

- Der `Bool` Typ nimmt zwei mögliche Werte an, `True` oder `False`.
- Der `Char` Typ beinhaltet Zeichen wie `'a','B','/n'` etc.
- Der String Typ beinhaltet sequenzen von `Char` z.B. `"abc"`. Strings in Haskell sind Listen von Chars (und nicht sehr effizient).
- der `Int` Typ steht für ganze Zahlen im Bereich von $-2^{31} \leq x < 2^{31}$.
- `Integer` beinhaltet beliebige ganze Zahlen.

Zu jedem gegebenen Typ `a` gibt es einen Typ `[a]` der alle Listen beinhaltet, die Einträge vom Typ `a` beinhalten.

Beispiel

- `["abc", "xyz"]` ist vom Typ `[String]`
- `[1,2,3]` ist vom Typ `[Integer]`
- `[['a']]` ist vom Typ `[[Char]]`

Für alle Typen `a` und `b` beinhaltet der Typ `a->b` alle Funktionen, die eine Eingabe vom Typ `a` akzeptieren und eine Ausgabe vom Typ `b` zurückgeben.

Beispiel

Der Typ `String -> Bool` beinhaltet Funktionen, die einen String akzeptieren und jeweils `True` oder `False` zurückgeben.

Bemerkung

- *Funktionstypen lassen sich beliebig "verschachteln", wir werden später noch genauer auf diesen Umstand eingehen.*
- *Eine Funktion vom Typ `a->Bool` nennt man ein "Prädikat" auf `a` (entspricht einer Eigenschaft).*

Für alle Typen `a1`...`an` beinhaltet der Typ `(a,..,an)` alle Paare bei denen der i-te Eintrag vom Typ `ai` ist.

Beispiel

Der Typ `(Int, String, Char)` beinhaltet das Tripel $(1, "1", '1')$.

Bemerkung

- *Tupel lassen sich beliebig "verschachteln".*

Records sind Tupel, in denen die Einträge Labels (Namen) tragen.

Beispiel eines Records:

```
1 data Customer = Customer
2     { customerId :: Integer
3     , name :: String
4     }
```

Records definieren automatisch Funktionen um auf ihre Datenfelder zuzugreifen:

```
1 customerId :: Customer -> Integer
2 name :: Customer -> String
```

In der Mengenanalogie entspricht der Summentyp der Vereinigung von disjunkten Mengen[3]. In Haskell können Summen direkt deklariert werden:

```haskell
data Shape
    = Rectangle Float Float
    | Square Float
    | Circle Float
```

[3]Ein wesentlicher Unterschied besteht darin, dass den Elementen des Summentyps die Zugehörigkeit zum entsprechenden "Summanden" explizit mitgegeben wird.

Summentypen lassen sich direkt mit Patterns dekonstruieren:

```
1 area :: Shape -> Float
2 area (Rectangle a b) = a * b
3 area (Square a) = a * a
4 area (Circle r) = 3.14 * r * r
```

Summentypen können auch rekursiv sein:

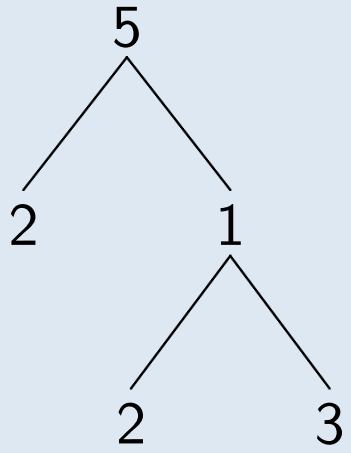
```
1 data Tree a
2     = Node (Tree a) a (Tree a)
3     | Leaf a
```

Rekursive Summen lassen sich mit Rekursion und Patterns dekonstruieren:

```haskell
-- Tiefe eines Baumes
depth :: Tree a -> Integer
depth (Node left _payload right) =
    1 + max (depth left) (depth right)
depth (Leaf _payload) = 1
```

Aufgabe

Geben Sie den Term an, der dem folgenden "`Tree Integer`" entspricht:



Aufgabe

Zeichnen Sie den zum folgenden Term passenden Baum:

```
1 Node (Leaf 1) 4 (Node (Leaf 4) 0 (Leaf 9))
```

Aufgabe

Die Karten eines Spieles (z.B.) Black-Jack bezeichnen entweder

- einen König (King),
- eine Dame (Queen),
- einen Buben (Jack),
- ein Ass (Ace)
- oder einen numerischen Wert.

Implementieren Sie einen Summentyp der diese Tatsache reflektieret.

Aufgabe

Die Werte der Karten (z.B. in Black-Jack) seien durch folgende Zuordnung gegeben:

- Figuren (King, Queen, Jack) haben den Wert 10
- Karten mit numerischem Wert haben ebendiesen Wert
- Das Ass hat entweder den Wert 1 oder den Wert 11

Implementieren Sie eine Funktion `value`, die den Wert einer Karte berechnet sowie einen Typ der Kartenwerte repräsentiert.

Listen[4] sind als Summentyp definiert:

```
1 data List a
2     = Cons a (List a)
3     | Nil
```

Wie wir bereits wissen, stellt Haskell, so wie auch viele funktionale Sprachen, für Listen einiges an “syntactic sugar”, sowie eine umfangreiche Bibliothek zur Verfügung.

[4] Einfach verkettete Listen

Aufgabe

Studieren Sie das Listen Modul (`https://hackage.haskell.org/package/base-4.14.1.0/docs/Data-List.html`) und testen Sie verschiedene Funktionen.

In einer Typklasse werden verschiedene Typen, die bestimmte Eigenschaften teilen zusammengefasst (ähnlich einem Interface).

Beispiel

Die Typklasse `Eq` enthält alle Typen deren Elemente vergleichbar sind. Dies wird deutlich, wenn wir die Signatur der Funktion `(==)` betrachten

```
(==) ::  Eq a => a -> a -> Bool
```

Indem wir diesen die in einer Typklasse festgelegten Funktionen auf einem Typ definieren, können wir diesen der entsprechenden Klasse hinzufügen (den Typ zu einer Instanz machen). Einige Instanzen lassen sich auch automatisch ableiten, dies wird syntaktisch mit dem `deriving` Keyword gemacht.

Beispiel

```
1 data Person = Person
2     { name :: String
3     , age :: Integer
4     , idNumber :: Integer
5     }
6
7 instance Eq Person where
8     (==) (Person _ _ id1) (Person _ _ id2) =
          id1 == id2
```

Viele Typklassen verlangen, dass ihre Instanzen (respektive deren Implementierung) gewisse Eigenschaften erfüllen, die nicht vom Compiler überprüft werden können (Unter Anderem soll == beispielsweise eine Äquivalenzrelation sein).
Wir werden später konkrete Typklassen (und ihre Regeln) im Detail anschauen.