Tag 5

Wahrheitswerte und Tupel

Heute führen wir noch zwei weitere Typen bzw. Arten von Typen ein, bevor wir dann in den nächsten Tagen die interaktive Umgebung verlassen und richtige kleine Programme schreiben werden, weil man dort besser mit dem noch fehlenden wichtigen Konzept der Rekursion arbeiten kann.
Wahrheitswerte gibt es in vielen Sprachen, oft werden sie aber lediglich durch sepzielle Zahlenwerte wie `0` und `1` simuliert. In Haskell gibt es einen eigenen Datentyp `Bool` für Wahrheitswerte. Dieser Typ hat zwei Elemente, nämlich `True` und `False`.

```
Prelude> True
True
it :: Bool

Prelude> False
False
it :: Bool
```

Der Datentyp `Bool` wäre relativ nutzlos ohne die Möglichkeit, in Abhängigkeit von einem Wahrheitswert unterschiedliche Aktionen auszuführen. In Haskell gibt es daher eine `if-then-else`-Konstruktion, mit der man genau dies tun kann. So ein Konstrukt ist aus vielen Programmiersprachen bekannt, aber meist wird es etwas anders verwendet als in Haskell. Wir fangen mit einem Beispiel an:

```
Prelude> if True then 1 else 0
1
it :: Integer
```

Ein `if-then-else`-Ausdruck hat drei Argumente, eine *Bedingung*, einen `then`-*Teil* und einen `else`-*Teil*. Wenn die Bedingung wahr ist, dann ist der Wert des Gesamtausdrucks der `then`-Teil. Ist die Bedingung falsch, so ist der Wert des Konstrukts der `else`-Teil. Darum wird obiges Beispiel zu `1` ausgewertet.
Interessanterweise hat also auch ein `if-then-else`-Konstrukt einen Typ! Dies ist ein Unterschied zu den meisten Programmiersprachen. Dort dient eine solche Abfrage zur Gruppierung von Anweisungen. In Haskell ist es einfach eine Möglichkeit, aus drei Ausdrücken (Bedingung, `then`-Teil und `else`-Teil) einen neuen Ausdruck zu machen. Als solches ist `if-then-else`, abgesehen von der etwas speziellen Syntax, nichts weiteres als eine normale Funktion. Wir können, wenn wir wollen, die spezielle Syntax verstecken und eine Funktion für solche Abfragen definieren. Im GHCi können wir zum Beispiel sagen:

```
Prelude> let cond = (\c t e -> if c then t else e)
cond :: forall t. Bool -> t -> t -> t
```

In Hugs können wir an der interaktiven Kommandozeile keine eigenen Funktionsnamen definieren, aber den Lambda-Ausdruck können wir eingeben und zum Beispiel seinen Typ erfragen:

```
Prelude> :t (\c t e -> if c then t else e)
\c t e -> if c then t else e :: Bool -> a -> a -> a
```

Der Typ bestätigt die obige informelle Beschreibung der Wirkungsweise von `if-then-else`. Wir haben eine Funktion mit drei Argumenten. Das erste, die Bedingung, muß vom Typ Bool sein. Der Typ des `then`- und des `else`-Teils ist nicht näher festgelegt, aber beide Teile müssen denselben Typ haben. Da die Typprüfung eines Programmes nämlich zur Übersetzungszeit durchgeführt wird, die Bedingung eines `if-then-else`-Ausdrucks aber erst zur Ausführungszeit ausgewertet werden kann, ist die Typkorrektheit eines Programmes nur dann zu garantieren, wenn der Typ des Resultats unabhängig von der Bedingung ist.
Eine Konsequenz aus diesen Überlegungen ist auch, daß es – im Gegensatz wiederum zu den meisten anderen Sprachen – keine Form des `if-then-else` ohne einen `else`-Teil gibt, denn was sollte dann der Wert (oder Typ) des Ausdrucks sein, wenn die Bedingung nicht erfüllt ist?
Natürlich gibt es auch eine ganze Menge von vordefinierten Operatoren und Funktionen zur Verwendung mit Wahrheitswerten, zum Beispiel:

```
(||) :: Bool -> Bool -> Bool                (logisches „oder“)
(&&) :: Bool -> Bool -> Bool                (logisches „und“)
not  :: Bool -> Bool                        (logisches „nicht“)
or   :: [Bool] -> Bool                      („oder“ für Listen)
and  :: [Bool] -> Bool                      („und“ für Listen)
null :: forall a. [a] -> Bool               (leere Liste?)
(==) :: forall a. (Eq a) => a -> a -> Bool  (Gleichheit)
(/=) :: forall a. (Eq a) => a -> a -> Bool  (Un-Gleichheit)
```

Bei den letzten beiden Funktionen in dieser Liste begegnen wir wieder einmal einem Typ mit Prädikat. Der Gleichheits- und Ungleichheitsoperator kann auf zwei Werte eines übereinstimmenden Typs angewendet werden, wenn für diesen das Prädikat `Eq` erfüllt ist. Alle Typen, die wir bislang diskutiert haben – mit der Ausnahme von Funktionen –, erlauben das Überprüfen von Werten auf Gleichheit, erfüllen also das Prädikat `Eq`. Damit ist `Eq` ein „häufigeres“ Prädikat als etwa `Num` für Zahlenwerte oder `Float` für Fließkommazahlen, die wir schon früher gesehen haben.
Listen von auf Gleichheit überprüfbaren Typen sind automatisch auch wieder vergleichbar, so daß zum Beispiel

```
Prelude> [2,3] == [2,3,4]
False
```

möglich ist.

Aufgabe 1

Schreibe eine Funktion, die zwei Integer-Parameter hat. Der erste soll gerade und der zweite ungerade sein. Benutze dafür die vordefinierte Funktion `even`, um zu bestimmen, ob eine Zahl gerade ist.

Aufgabe 2

Re-Implementiere die Funktion `even` unter Verwendung der generelleren Funktion `mod`, die den Rest bei einer Integer-Division bestimmt. (Achtung: Dies ist ohne die Verwendung von `if` möglich!)

Im Zentrum des zweiten Teils der heutigen Lektion stehen die Tupel-Typen. Mit diesen kann man mehrere Werte (eventuell unterschiedlichen Typs) bündeln und gruppieren. Damit sind sie ähnlich zu Listen. Listen können eine beliebige Anzahl von Elementen desselben Typs enthalten. Ein Tupel enthält eine feste Anzahl von Elementen (das heißt, die Anzahl der Elemente ist aus dem Typ ablesbar), aber die Elemente können verschieden Typen haben.
Tupel werden in runde Klammern gesetzt, und ihre Elemente werden mit Kommas voneinander getrennt:

```
Prelude> (1,1)
(1,1)
it :: (Integer, Integer)

Prelude> :t (\x -> x,'a',False)
forall t. (t -> t, Char, Bool)
```

Hier können wir noch einmal sehen: Wir können verschiedene Anzahlen von Elementen beliebiger Typen zu Tupeln zusammenfassen. Aus dem jeweiligen Typ geht sowohl die „Länge“ des Tupel hervor als auch die Typen der einzelnen Elemente.
Für Paare, also Tupel mit zwei Elementen, gibt es vordefinierte Projektionsfunktionen, um an die einzelnen Elemente wieder heranzukommen:

`fst :: forall a b. (a,b) -> a` (selektiert die erste Komponente)
`snd :: forall a b. (a,b) -> b` (selektiert die zweite Komponente)

Aufgabe 3

Schreibe eine Funktion des Typs `forall b a. (a,b) -> (b,a)`, die die Komponenten eines Paares (egal mit welchen Elementtypen) vertauscht.

Tupel können auch geschachtelt werden. Mit anderen Worten: Tupel können ihrerseits wiederum Tupel enthalten, etwa:

```
Prelude> (1,(1,1))
(1,(1,1))
it :: (Integer, (Integer, Integer))
```

Ein solches verschachteltes Paar ist *nicht* dasselbe wie das entsprechende Tripel `(1,1,1)`!
Tupel bieten eine Möglichkeit, um Funktionen mit mehreren Argumenten zu realisieren. Statt wirklich mehrerer Argumente übergibt man einfach ein Tupel, welches alle Argumente enthält. Wie wir in den bisherigen Lektionen allerdings schon gesehen haben, werden Funktionen mit

mehreren Argumenten in Haskell meistens eleganter mittels Currying umgesetzt. Soll eine Funktion aber mehrere Werte als Resultat zurückgeben, dann ist Currying keine Option, wohingegen Tupel eine gute Möglichkeit darstellen.

Aufgabe 4

Wie wir gelernt haben, gibt es zwei Möglichkeiten, in Haskell eine Funktion mit zwei Argumenten zu implementieren: Mittels Currying oder durch Verwendung eines Paares. Haskell selbst bietet zwei Funktionen, um zwischen diesen beiden Sichtweisen zu konvertieren:

```
curry   :: forall a b c. ((a,b) -> c) -> a -> b -> c
uncurry :: forall a b c. (a -> b -> c) -> (a,b) -> c
```

Probiere an `uncurry (+)` aus, daß es tatsächlich funktioniert, und versuche, die beiden Funktionen `curry` und `uncurry` mittels Lambda-Abstraktion zu reimplementieren.

Es gibt übrigens keine 1-Tupel. Für jeden Ausdruck `x` ist die Schreibweise `(x)` äquivalent zu `x` (Klammern dienen ja auch einfach der Gruppierung von Ausdrücken, wie wir zu Beginn bei den Grundrechenoperationen schon gesehen haben). Es gibt aber einen speziellen 0-Tupel-Typ, genannt „Unit“. Der Typ sowie sein einziger Wert werden mit `()` bezeichnet:

```
Prelude> ()
()
it :: ()
```

Die Präsenz dieses Typs scheint derzeit vermutlich nicht sonderlich hilfreich, aber wir werden später noch Verwendungsmöglichkeiten kennenlernen.

Lernziele Tag 5

- `Bool` ist ein Typ mit zwei Werten, `True` und `False`.
- Das `if`-Konstrukt dient zum Programmieren von Fallunterscheidungen mit Hilfe von Wahrheitswerten.
- Tupel können benutzt werden, um beliebig viele Ausdrücke zu gruppieren. Der Typ eines Tupels reflektiert sowohl seine Länge als auch die Typen seiner Komponenten.