Übungsblatt 1

Typtheorie

Schreiben Sie alle **möglichen** Implementationen der folgenden Funktionen. Wozu könnte `fun2` nützlich sein?

```
fun1 :: a -> a
fun1 = _fun1

fun2 :: a -> b -> a
fun2 = _fun2

fun3 :: (Eq a) => a -> a -> Bool
fun3 = _fun3
```

Wir haben in der Vorlesung parametrisierte Typen kennengelernt. Der simpelste hiervon ist `Identity`, der nur einen anderen Typen einpackt.

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

Diese Definition stellt uns automatisch den Konstruktor `Identity :: a -> Identity a` zur Verfügung, der ein `a` einpackt. Schreiben Sie die Funktion

```
unIdentity :: Identity a -> a
unIdentity = _unIdentity
```

welche diesen Vorgang wieder rückgängig macht.

Angenommen, Sie hätten nun ein Wert vom Typen `Identity a` und eine Funktion mit dem Typen `a -> b`. Wie wenden Sie diese auf das `a` "innerhalb" des `Identity` an um ein `Identity b` herzustellen? Schreiben Sie also eine Funktion

```
mapIdentity :: (a -> b) -> Identity a -> Identity b
mapIdentity = _mapIdentity
```

Hinweis: Es gibt *zwei* prinzipielle Vorgehen dieses zu implementieren. Kommen Sie auf beide?

Funktionen sind auch nur Typen

Datentypen können auch Funktionen enthalten. Sehen Sie sich einmal den Datentypen

```
data Pred a = Pred (a -> Bool)
```

an. Hier wird ein Prädikat definiert, welches (gegeben einen Datentyp `a`) eine Funktion gespeichert hat, die `a` in einen `Bool` umwandeln kann (etwa um irgendwas zu filtern/selektieren/löschen/..., wenn man dies an eine weitere Funktion übergibt).

Auch hier können Sie eine Funktion schreiben, die das `Pred a` wieder "auspackt". Definieren Sie

```
unPred :: Pred a -> (a -> Bool)
unPred = _unPred
```

Da Haskell-Funktionen aber "gecurried" sind (mehr dazu in der Vorlesung), können Sie die Klammern hinten in der Signatur auch weglassen und erhalten `unPred :: Pred a -> a -> Bool`, was man zugleich als "wende `Pred a` an, wenn du ein `a` bekommst" lesen kann. In der Tat sind beide Funktionen identisch (wieso?).

Bonus

Was für eine Funktion bräuchten Sie um ein `Pred a` in ein `Pred b` umzuwandeln? Können Sie diese implementieren?

```
mapPred :: _fun -> Pred a -> Pred b
mapPred = _mapPred
```

Neue Typen erfinden

In Haskell ist ein zentraler Vorgehenspunkt das Definieren und Verwenden von eigenen Datentypen. Zur Erinnerung; es gibt zwei Möglichkeiten, die man miteinander kombinieren kann: `data Prod a b c = Prod a b c` (Produkttyp) benötigt sowohl `a`, `b` als auch `c` um einen Wert zu erzeugen, `data Sum a b = Sum1 a | Sum2 b` (Summentyp) braucht entweder ein `a` um durch den Konstruktor `Sum1` ein `Sum a b` zu erzeugen oder ein `b` um durch den Konstruktor `Sum2` ein `Sum a b` zu erzeugen.

Definieren Sie einen Datentypen `Vielleicht a`, der zwei Konstruktoren besitzt: Einen Konstruktor, mit dem durch ein `a` ein `Vielleicht a` konstruiert wird

und ein zweiter Konstruktor, der keinen Wert nimmt, sondern die "Abwesenheit eines `a`" symbolisieren soll.

Können Sie hier eine Funktion schreiben, die das `a` extrahiert? Wenn ja, implementieren Sie diese; wenn nein, geben Sie eine kurze Begründung.

Wie würden Sie mittels einer Funktion `a -> b` ein `Vielleicht a` in ein `Vielleicht b` wandeln? Implementieren Sie

```
mapVielleicht :: (a -> b) -> Vielleicht a -> Vielleicht b
mapVielleicht = _mapVielleicht
```

Bonus

Man kann Typen natürlich auch Schachteln. Worin liegt eigentlich der Unterschied zwischen einem `Pred (Vielleicht a)` und einem `Vielleicht (Pred a)`? Oder sind diese identisch?