

gibt es in Haskell die Möglichkeit eigene Datentypen zu definieren, die gerade den Ansprüchen der jeweiligen Anwendung genügen.

Oft treten Gemeinsamkeiten zwischen verschiedenen Datentypen auf. Zum Beispiel können alle Zahlen sortiert werden, unabhängig davon ob sie vom Datentyp `Int` oder `Double` sind. Dieselbe Motivation, die uns dazu gebracht hat polymorphe Typen einzuführen, leitet uns jetzt dazu, Typklassen zu definieren. Diese gruppieren Typen mit gleichen Eigenschaften, damit Funktionen für die ganze Gruppe nur einmal entworfen werden müssen.

7.1 Typsynonyme mit `type`

In vielen Fällen ist es notwendig, verschiedene Dinge mit demselben Typ abzubilden. In einem Adressbuch bietet es sich zum Beispiel an, sowohl den Namen als auch die Adresse als `String` zu speichern. Das ganze Adressbuch kann dann als Liste von Tupeln mit Name und Adresse implementiert werden. Anschließend kann beispielsweise eine Funktion geschrieben werden, die eine Adresse zu einem Namen sucht:

```
eintrag :: String -> [(String, String)] -> String
eintrag name ((n,a):r)
| name == n = a
| otherwise = eintrag name r
eintrag []    = error "nicht enthalten"
```

Die Typen der Signatur sind allerdings ziemlich nichtssagend. Es ist nicht zu erkennen, ob das erste Argument des Tupels `(String, String)` ein Name oder eine Adresse sein soll. Um die Arbeitsweise dieser Funktion zu verstehen, muss der Programmcode an dieser Stelle genauer studiert werden.

Dieses Problem kann leicht umgangen werden, indem bereits vorhandene Typen wie der `String` einen neuen Namen erhalten. Wir sprechen in diesem Fall von einem Typsynonym. Das Schlüsselwort hierfür ist `type`. Übrigens ist `String` auch nur ein Typsynonym für den Listentyp `[Char]`.

Schauen wir uns die neue Version des kleinen Adressbuches an:

```
type Name      = String
type Adresse   = String
type Adressbuch = [(Name, Adresse)]

eintrag :: Name -> Adressbuch -> Adresse
...
```

Jetzt sind die Eingaben für die `eintrag`-Funktion verständlicher zu lesen.

Typsynonyme müssen nicht monomorph, also von einem festen Datentyp sein. Zum Beispiel sollen nicht unbedingt Namen und Adressen in Listen gespeichert werden, sondern beliebige (Schlüssel, Wert)-Paare. Dann werden einfach zwei Typvariablen hinter dem Typsynonym aufgelistet:

```
type NachschlagListe k v = [(k,v)]
type Adressbuch          = NachschlagListe String String
```

Auf diese Weise können Typen in Programmen viel aussagekräftiger formuliert werden, was die Verständlichkeit des Programms wesentlich erhöht.

Mit **type** vereinbarte Synonyme sind allerdings nicht sicher. Der Compiler macht keinen Unterschied zwischen Name und Adresse, er sieht nur **String**. So könnte fälschlicherweise eine Adresse in die Suchfunktion eingeben werden, ohne dass Haskell sich darüber beschwert. In den meisten Fällen bereitet das keine weiteren Probleme.

Manchmal möchte man die Typsicherheit aber nicht missen. Soll die Typsicherheit explizit gesichert werden, wird statt des Schlüsselwortes **type** das Schlüsselwort **newtype** verwendet. Die damit erzeugten Synonyme müssen dann aber explizit konvertiert werden. Dazu erhalten die Typen einen zusätzlichen Datenkonstruktor.

Schauen wir uns das an unserem Adressbuchbeispiel an:

```
newtype Name      = N String
newtype Adresse   = A String
type Adressbuch = [(Name,Adresse)]

eintrag (N name) ((N n, a):r)
| name == n = a
| otherwise = eintrag (N name) r
eintrag _ [] = error "nicht enthalten"
```

Wie in diesem Beispiel zu sehen ist, müssen die Datenkonstruktoren **N** und **A** explizit über die Eingabeparameter entfernt werden, um an die Werte heranzukommen. Die mit **newtype** erzeugten Typen können wir leider nicht mehr einfach auf der Konsole ausgeben lassen, das führt zu einer Fehlermeldung. Eine Lösung für dieses Problem wird in Abschn. 7.4 beschrieben.

Auch bei der Verwendung von **newtype** ist die Benutzung von Typparametern natürlich erlaubt.

7.2 Einfache algebraische Typen mit data und newtype

Ganz neue Datentypen werden durch das Schlüsselwort **data** erzeugt. Der einfachste neue Datentyp enthält nur einen einzigen Wert. Diesen können wir beispielsweise so definieren:

```
data Einfach = Wert
```

Jetzt ist Einfach ein neuer Datentyp, den wir für die Definition von Funktionen verwenden können.

```
f :: Einfach -> String
f Wert = "Das ist ein Wert!"
```

Leider können wir uns auch diesen neu definierten Wert nicht einfach ausgeben lassen:



```
Hugs> Wert
ERROR - Cannot find "show" function for:
*** Expression : Wert
*** Of type     : Einfach
```

Dieses Problem kann zunächst dadurch umgangen werden, indem deriving Show hinter die Definition des Datentyps geschrieben wird:

```
data Einfach = Wert deriving Show
```

Jetzt können wir den Wert des Datentypen ausgeben:

```
Hugs> Wert
Wert
```

Das soll uns an dieser Stelle erst einmal genügen. In einem der nächsten Kapitel werden wir diese Thematik genau behandeln und zeigen, was es damit auf sich hat.

Selbstverständlich ist ein Datentyp mit nur einem Wert nicht sonderlich spannend. Soll ein Datentyp mehrere Werte haben, werden diese durch einen senkrechten Strich getrennt:

```
data Wochenende = Sonntag | Samstag

f :: Wochende -> Int
f Sonntag = 23
f Samstag = 42
```

Einen Datentyp, der wie in diesem Beispiel nur null-stellige Datenkonstruktoren besitzt, bezeichnen wir als Aufzählungstypen.

Mit dieser Technik können wir uns zum Beispiel unseren eigenen Datentypen für boolesche Werte definieren und ihn für ein paar Logikfunktionen verwenden:

```
data Boolean = T | F deriving Show

not :: Boolean -> Boolean
not T    = F
not F    = T
```

```
and :: Boolean -> Boolean -> Boolean
and T x = x
and F _ = F
```

Wie an diesem Beispiel zu sehen ist, besteht, vom Namen abgesehen, kein Unterschied zwischen dem Datentyp für das Wochenende und dem Datentyp für die booleschen Werte. Es ist wichtig zu erkennen, dass die Namen für den Computer vollkommen bedeutungslos sind. Wir könnten ebensogut Logik mit den Tagen des Wochenendes machen. Die Bedeutung der Typen liegt ganz in unseren Händen. Darum ist es wichtig, sprechende Namen zu wählen oder besser noch erläuternde Kommentare zu hinterlassen.

In den Typdefinitionen können auch bereits vorhandene Typen wiederverwendet werden. Ein einfaches Beispiel ist die Box, die einen `Int` speichern kann. Sie kann ganz analog auch mit `newtype` vereinbart werden:

```
data Box = B Int

tueRein :: Int -> Box
tueRein = B

nimmtRaus :: Box -> Int
nimmtRaus (B n) = n
```

An der Funktion `tueRein` sehen wir, dass die Datenkonstruktoren wie Funktionen verwendet werden können. Davon können wir uns auch in der Konsole überzeugen:

```
Hugs> :t B
B :: Int -> Box
```

Es gibt ein paar kleine semantische Unterschiede zwischen `data` und `newtype`. (s. [16] Abschn. 4.2.3). Anders als mit `newtype` können mehr als nur einstellige Datenkonstruktoren definiert werden. Eine beliebige Anzahl von Typen kann hinter einem Datenkonstruktor stehen.

Schauen wir uns ein Beispiel an, in dem ein `Int`, ein `Float` und ein `String` gehalten werden kann:

```
data MultiBox = B Int Float String
```

Neben den monomorphen Typen, die wir bereits verwendet haben, gibt es wie bei `type` und `newtype` die Möglichkeit, Typvariablen bei der Definition einzusetzen.

So können wir beispielsweise eine Box für einen beliebigen Datentypen definieren:

```
data Box a = B a
```

Dabei müssen alle Typvariablen, wie bereits bekannt, vor dem Gleichheitszeichen einmal aufgelistet werden.

So wie die Datenkonstruktoren Funktionen sind, sind jetzt auch die Typnamen Funktionen, allerdings arbeiten sie nicht auf den Werten eines Datentyps, sondern auf dem Datentyp selbst. Box ist in diesem Beispiel eine Abbildung von einem beliebigen Typen *a* zu einem Typen in einer Box. Solche Funktionen werden analog zu den Datenkonstruktoren als Typkonstruktoren bezeichnet.

Die Typen der Typkonstruktoren haben mit *kind* (engl. Art, Gattung) einen besonderen Namen. Mit dem Befehl `:kind` lassen sich die Typen anzeigen. Die polymorphe Box hat einen Typparameter:

```
Hugs> :kind Box
Box :: * -> *
```

Die MultiBox hat hingegen keine Typparameter, somit hat sie den Typ:

```
Hugs> :kind MultiBox
MultiBox :: *
```

Mit diesen Mitteln können wir ein paar Datentypen nachimplementieren, die es in Haskell bereits gibt. Das fördert unser Verständnis und dient gleichzeitig als Fingerübung.

7.2.1 Datentyp Tupel

Tupel haben wir bereits in Abschn. 5.2 kennengelernt. Es gibt nur einen Datenkonstruktor, nennen wir ihn *P*, hinter dem zwei Elemente von verschiedenen Typen stehen können.

Wir brauchen also zwei Typvariablen für den Datentyp Tupel:

```
data Tupel a b = P a b
```

Damit können wir auch ganz einfach die Funktionen *fst* und *snd* definieren, die das erste bzw. zweite Tupelement zurückliefern:

```
fst (P a _) = a
snd (P _ a) = a
```

Tupel sind in Haskell tatsächlich so definiert. Die Schreibweise mit den Klammern ist nur syntaktischer Zucker, damit die Lesbarkeit erhöht wird.

7.2.2 Datentyp Either

Wir wissen, dass eine Funktion genau ein Ergebnis zurückliefert und dieses kann eigentlich nur von einem fest definierten Datentyp sein. Oft ist es notwendig, zwei verschiedene Datentypen als Ergebnis zu haben. Damit wir an dieser Stelle flexibel

sind aber trotzdem mathematisch korrekt bleiben, definieren wir einen Datentypen Either, der entweder einen Wert von einem Typen oder einen Wert eines anderen Typen zurückgeben kann.

Angenommen, eine Berechnung kann auf mehrere Arten fehlschlagen, dann soll entweder das richtige Ergebnis oder eine Fehlermeldung eines anderen Typen zurückgegeben werden.

Für diese Anwendungen gibt es Either, das wie folgt definiert ist:

```
data Either a b = Left a | Right b
```

Als Beispiel für eine Anwendung schauen wir uns die folgende Funktion an:

```
sqrtDiv n m
| n < 0      = Left "negative Wurzel"
| m == 0     = Left "Division durch 0"
| otherwise   = Right (sqrt n / m)
```

Diese Funktion berechnet $\frac{\sqrt{n}}{m}$. Dabei kann es zu zwei unterschiedlichen Fehlersituationen kommen, der Division durch Null und dem Ziehen einer negativen Wurzel. Beide Fehler fangen wir über Guards ab und liefern einen Fehlerwert vom Typ String. In allen anderen Fällen wird das korrekte Ergebnis als Float zurückgegeben.

7.2.3 Datentyp Maybe

In vielen Funktionen können Fehler auftreten, die den Benutzer nicht weiter interessieren. Er möchte lediglich wissen, ob die Berechnung erfolgreich war. Es ist also unnötig Either zu verwenden, da keine zusätzliche Daten zurückgegeben werden müssen.

Zum Beispiel wurde beim Durchsuchen eines Telefonbuchs der gewünschte Eintrag nicht gefunden. In diesem Fall soll ein Sonderwert zurückgegeben werden, der einen Fehlschlag signalisiert. Für diesen Zweck gibt es den Datentypen Maybe:

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

Mit der Funktion teilen haben wir ein kleines Anwendungsbeispiel:

```
teilen :: Int -> Int -> Maybe Int
teilen n 0 = Nothing           -- man kann nicht durch 0 teilen
teilen n m = Just (div n m)
```

Ist die Berechnung erfolgreich, wird der Wert mit Just zurückgeliefert und muss entsprechend mit Pattern matching wieder entfernt werden. Falls die Berechnung fehlschlägt, wird Nothing zurückgeliefert.

Abschließend noch ein kurzes Anwendungsbeispiel::

```
teilenMoeglich :: Int -> Int -> String
teilenMoeglich a b = case (teilen a b) of
    Just _ -> "Der Divisor war nicht 0"
    Nothing -> "Die Division ist nicht möglich"
```

Hier verwenden wir case (s. dazu Abschn. 3.1.5), um zu testen, ob Nothing zurückgegeben wurde und liefern einen entsprechenden String als Ergebnis zurück.

7.2.4 Datentypen mit mehreren Feldern

Sollte der Fall eintreten, dass Datenkonstruktoren mit mehreren Werten benötigt werden, kann es unschön sein, dieses mit Pattern matching zu verarbeiten, da viele Felder des Datentyps vielleicht gar nicht für jede Funktion interessant sind. Ein ähnliches Problem haben wir schon bei der Verwendung von Tupeln gesehen. Um ein Pattern match auf ein n-Tupel zu machen, müssen alle n Argumente aufgeschrieben werden.

Der Zugriff soll eigentlich auf ein bestimmtes Feld aus dem Datentyp erfolgen und dieses zurückgeben oder mit Funktionen ein einzelnes Feld verändern können. Hilfreich wären auch Funktionen, die bei der Eingabe nur eines der Datenfelder trotzdem ein neues Exemplar des Datentypen erzeugen können.

Schauen wir uns ein Beispiel dazu an:

```
data Eintrag = E
String -- Vorname
String -- Nachname
String -- Straße
String -- Stadt
String -- Land

getVorname :: Eintrag -> String
getVorname (E n _ _ _ _) = n

setVorname :: String -> Eintrag -> Eintrag
setVorname n (E _ a b c d) = E n a b c d

createVorname :: String -> Eintrag
createVorname n = E n undefined undefined undefined undefined

-- analog für die weiteren Felder
-- ...
```

An diesem schlechten Beispiel ist gut zu sehen, dass die stupide Schreibarbeit schnell außer Kontrolle geraten kann. Eigentlich kann diese Schreibarbeit genauso gut vom Compiler übernommen werden.

Dafür gibt es die sogenannte Recordsyntax:

```
data Eintrag = E {
    vorname :: String,
    nachname :: String,
    strasse :: String,
    stadt :: String,
    land :: String }
```

Da in diesem Fall alle Typen gleich sind, können wir sie weiter kürzen zu:

```
data Eintrag = E {
    vorname,
    nachname,
    strasse,
    stadt,
    land :: String }
```

Nicht nur die Felder sind mit dieser Syntax jetzt selbstdokumentierend, es werden auch alle Hilfsfunktionen vom Compiler automatisch generiert. Jetzt können wir die benannten Felder in typischer Pattern matching-Manier verwenden.

Um die verschiedenen syntaktischen Möglichkeiten zu beleuchten, definieren wir die Funktion vollerName auf drei verschiedene aber äquivalente Weisen. Außerdem zeigen wir, wie die Zugriffsfunktionen verwendet werden können, um einen existierenden Wert zu verändern bzw. einen neuen Wert zu erzeugen. Da wir bei der Funktion eintragMitNamen nur zwei der Felder explizit setzen, bleiben die anderen undefined:

```
-- drei äquivalente Funktionen
vollerName (E n nn _ _ _) = n ++ nn
vollerName' x             = vorname x ++ nachname x
vollerName''(E {vorname=n, nachname=nn}) = n ++ nn

setzeNamen vn nm x      = x {vorname = vn, nachname = nm}
eintragMitNamen vn nm = E {vorname = vn, nachname = nm}
-- die weiteren Felder sind automatisch undefined
```

Die Namen dürfen im selben Datentypen wiederverwendet werden, solange sich der Typ nicht ändert. Das gilt aber nicht bei verschiedenen Datentypen. Wir könnten beispielsweise einen alternativen Datenkonstruktor für kurze Einträge einführen, der keine Adresse enthält:

```
data Eintrag = E {
    vorname,
    nachname,
    strasse,
    stadt,
    land :: String} |
    K {
        vorname,
```

```
nachname :: String }
```

Für diesen Datentypen können wir jetzt Funktionen definieren, die sowohl für Exemplare von langen als auch für kurze Einträge funktionieren, indem wir nur vorname und nachname einsetzen.

Das wäre mit reinem Pattern matching nicht so komfortabel zu realisieren:

```
vollerName x = vorname x ++ nachname x
```

7.3 Rekursive Algebraische Typen

Wir haben bereits gesehen, dass Typkonstruktoren auf gewisse Weise als Funktionen über Typen interpretiert werden können. Rekursion lässt sich nicht nur als Entwurfstechnik für Funktionen verwenden, sondern auch für die Definition von Datentypen.

Die Listen haben wir bereits als rekursiven Datentypen kennengelernt. Eine Liste ist entweder leer oder sie besteht aus einem Kopfelement und einer Restliste. Als algebraischen Datentypen können wir das beispielsweise wie folgt definieren:

```
data List a = Nil | Cons a (List a)
```

Mit dieser Definition einer Liste können wir jetzt genauso arbeiten, wie mit den bekannten Listen. Im Folgenden werden beispielhaft ein paar uns bekannte Listenfunktionen für unseren Typen neu implementiert:

```
-- prüft, ob die Liste leer ist
null :: List a -> Bool
null Nil = True
null _ = False

-- liefert das erste Element der Liste
head :: List a -> a
head (Cons a _) = a
head _ = error "head auf leerer Liste"

-- liefert den Rest der Liste ohne erstes Element
tail :: List a -> List a
tail (Cons _ r) = r
tail Nil = error "tail auf leerer Liste"

-- konkatiniert zwei Listen
concat :: List a -> List a -> List a
concat Nil x = x
concat (Cons a r) x = Cons a (concat r x)
```

```
-- führt eine Funktion f auf alle Listenelemente aus
map :: (a -> b) -> List a -> List b
map _ Nil           = Nil
map f (Cons a r)   = Cons (f a) (map f r)
```

Natürlich lassen sich auch kompliziertere Rekursionsmuster angeben. Beispielsweise könnten wir eine wechselseitige Rekursion zwischen zwei Datentypen definieren:

```
data A = Cons Int B
data B = Cons String A
```

Das genügt erst einmal, um in die Thematik einzusteigen. In einem späteren Kapitel werden wir uns mit komplizierteren Datentypen genauer beschäftigen.

7.4 Automatische Instanzen von Typklassen

Wir haben schon das Problem gesehen, dass selbstdefinierte Datentypen nicht ohne weiteres auf der Konsole angezeigt werden können. Beispielsweise können wir einen Datentypen Saison für die vier Jahreszeiten definieren:

```
data Saison = Fruehling | Sommer | Herbst | Winter
```

Anschließend können wir den neuen Datentypen verwenden und beispielsweise s1 und s2 definieren:

```
s1, s2 :: Saison
s1 = Fruehling
s2 = Sommer
```

Versuchen wir an dieser Stelle allerdings s1 auf der Konsole anzuzeigen, erhalten wir die folgende Fehlermeldung:

```
Hugs> s1
ERROR - Cannot find "show" function for:
*** Expression : s1
*** Of type    : Saison
```

Der Grund für den Fehler ist, dass Haskell noch nicht weiß, wie der neue Datentyp auf der Konsole ausgeben werden soll. Wie wir bereits gesagt haben, besitzen neue Typen keinerlei Semantik für den Interpreter. Ebensowenig weiß er, wie zwei Jahreszeiten auf Gleichheit getestet werden sollen:

```
Hugs> s1 == s2
ERROR - Cannot infer instance
*** Instance   : Eq Saison
*** Expression : s1 == s2
```

In Haskell wird diese Problematik über sogenannte Typklassen gelöst. Eine Typklasse ist eine Menge von Typen, die bestimmte Eigenschaften erfüllen. Beispielsweise gibt es eine Typklasse für alle Typen, die sich in Strings konvertieren lassen und damit auf der Konsole ausgegeben werden können. Diese Klasse heißt Show und einmal haben wir sie bereits verwendet (s. Abschn. 7.2).

Es existiert mit Eq (von *equality*) eine Typklasse für alle Typen die sich auf Gleichheit testen lassen. Damit Haskell weiß, dass ein Typ die von einer Typklasse geforderten Eigenschaften hat, müssen wir den Typen zu einer Instanz dieser Typklasse machen und die benötigten Funktionen implementieren.

Um beispielsweise einen neuen Datentypen zu einer Instanz der Klasse Show zu machen, müssen wir eine Funktion schreiben, die die Werte dieses Typen in Strings umwandelt.

Es gibt einige Typklassen, bei denen Haskell automatisch diese Funktionen generieren kann. Für Show haben wir das bereits gesehen. Um automatisch eine Instanz erzeugen zu lassen, wird das Schlüsselwort deriving verwendet.

Im Folgenden werden sechs Typklassen vorgestellt, für die wir auf diese Weise automatische Instanzen erzeugen können. Wir wollen dazu unseren neuen Datentyp Saison zu diesen sechs Klassen hinzufügen. Dazu erweitern wir die Definition um das Schlüsselwort deriving:

```
data Saison = Fruehling | Sommer | Herbst | Winter
deriving(Eq, Ord, Enum, Show, Read, Bounded)
```

Dabei ist es eigentlich redundant Eq und Ord zu nennen, da Ord die Klasse Eq impliziert. Jetzt erhalten wir für s1 und s2 alle Funktionen aus diesen Typklassen.

7.4.1 Typklasse Show

Die Typen in dieser Klasse können mit der Funktion show zu Strings konvertiert und auf der Konsole ausgegeben werden:

```
Hugs> (s1,s2)
(Fruehling, Sommer)
```

7.4.2 Typklasse Read

Die Typen dieser Typklasse besitzen mit der read-Funktion eine Umkehrfunktion zu show. Aus einem String kann wieder ein Wert des Typen erzeugt werden:

```
Hugs> (read "Sommer") == s2
True
```

7.4.3 Typklasse Eq

Alle Typen in Eq unterstützen die Funktionen (==) und (/=). Jetzt können wir s1 und s2 vergleichen:

```
Hugs> s1 == s2
False
```

```
Hugs> s1 /= s2
True
```

Manche Typen lassen sich auch inherent nicht auf Gleichheit testen, darum kann keine automatische Instanz erstellt werden. Typen, die Funktionen enthalten, sind ein gutes Beispiel dafür.

7.4.4 Typklasse Ord

Alle Typen in Ord haben eine (<=) Relation auf ihren Typen und unterstützen alle weiteren Vergleichsoperatoren:

```
Hugs> s1 <= s2
True
```

Wenn eine Instanz für Ord existiert, ist dadurch automatisch auch eine Instanz für Eq gegeben. Sollte eine Instanz dieser Klasse automatisch generiert werden, so sind Datenkonstruktoren die in der Definition weiter links stehen kleiner. Bei rekursiven Datentypen wird eine lexikographische Ordnung erstellt. Werte mit einem niedrigeren Rekursionsgrad stehen dabei in der Ordnung weiter vorn. Beispielsweise steht die leere Liste in der Ordnung der Listen ganz vorn.

7.4.5 Typklasse Enum

Wenn ein Typ eine Instanz der Klasse Enum ist, lassen sich automatisch Listen mit Elementen dieses Typs generieren:

```
Hugs> [Fruehling .. Herbst]
[Fruehling, Sommer, Herbst]
```

Automatische Instanzen können nur für Aufzählungstypen erstellt werden.

7.4.6 Typklasse Bounded

Diese Klasse Bounded verlangt die Definition zweier Konstanten minBound und maxBound, die den kleinsten bzw. größten Wert diesen Typen liefern:

```
Hugs> (minBound) :: Saison
Fruehling
```

Bei automatisch erzeugten Instanzen, entspricht der ganz links stehende Datenkonstruktur dem minBound und entsprechend der ganz rechts stehende dem maxBound. Automatische Instanzen können nur für Aufzählungstypen erstellt werden.

7.5 Eingeschränkte Polymorphie

Mit unserem Wissen zu den Typklassen können wir jetzt ein paar neue Funktions-signaturen schreiben. Bisher haben wir entweder nur monomorphe Funktionen, die ausschließlich mit einem Typen arbeiten und polymorphe Funktionen kennengelernt, die auf allen Typen arbeiten.

In vielen Fällen ist es allerdings wünschenswert, Funktionen nur für Typen zu definieren, die einer bestimmten Klasse angehören.

So können wir beispielsweise nicht das Minimum jeder Liste bestimmen, aber auch nicht für jeden Typen eine neue Funktion zum Finden des Minimums schreiben.

Stattdessen können wir die Funktion auf solche Typen einschränken, die sich miteinander vergleichen lassen. Solche also, die der Typklasse Ord angehören:

```
minimum :: Ord a => [a] -> a
minimum = foldl1 min
```

Das Ord a => teilt dem Compiler mit, dass diese Funktion auf die Typen aus Ord eingeschränkt wird. Wenn mehr als eine Klasse angegeben werden soll, müssen diese in Klammern gesetzt und durch Kommata getrennt sein, z.B. (Ord a, Show a, Num b) =>

7.6 Manuelles Instanziieren

Alternativ zur automatischen Instanziierung von Typen zu Klassen lassen sich diese auch manuell vornehmen. Das ist in den Fällen erforderlich, wenn entweder eine automatische Instanz für den Typen nicht erzeugt werden kann oder die erzeugte Instanz nicht das Gewünschte leistet.

Wollen wir beispielsweise einen anderen String mit der show-Funktion generieren, als den Namen des Datenkonstruktors selbst, dann muss eine Instanz für den Datentypen manuell erzeugt werden. Dazu wird das Schlüsselwort `instance` verwendet:

```
instance <Klasse> <Datentyp> where
  ...
```

In `<Klasse>` steht die Typklasse, der dem Typ `<Datentyp>` angehören soll. Zuletzt müssen noch die von der Typklasse geforderten Funktionen angeben werden.

Ein kleines Beispiel wird den Sachverhalt besser beleuchten. Wir wollen einen Datentypen für boolesche Werte zur Typklasse Show hinzufügen. Um einen Typ der Klasse Show hinzuzufügen, müssen wir die Methode `show` implementieren, die einen Wert des Datentyps in einen String umwandelt.

Der Programmcode könnte beispielsweise wie folgt aussehen:

```
data Boolean = T | F

instance Show Boolean where
  show T = "Wahr"
  show F = "Falsch"
```

Bevor wir mit der Instanziierung beginnen können, müssen wir wissen, welche Funktionen für die jeweiligen Typklassen gefordert sind. Ein Blick in die Dokumentation hilft (Hugs [56], GHC [57]).

Oft ist es so, dass nicht alle Funktionen implementiert werden müssen, da es schon Defaultimplementierungen gibt. Deswegen ist es ratsam erst mal nach der „minimal complete definition“ zu suchen. So muss beispielsweise für `Eq` entweder `(==)` oder `(/=)` implementiert werden.

Um unseren Datentypen `Boolean` aus dem vorhergehenden Abschnitt manuell als Instanz der Klasse `Eq` zu setzen, haben wir jetzt drei Möglichkeiten zur Verfügung.

In Variante 1 überschreiben wir nur `(==)`:

```
instance Eq Boolean where
  T == T = True
  F == F = True
  _ == _ = False
```

In der zweiten Variante könnten wir den Operator `(/=)` überschreiben:

```
instance Eq Boolean where
  T /= F = True
  F /= T = True
  _ /= _ = True
```

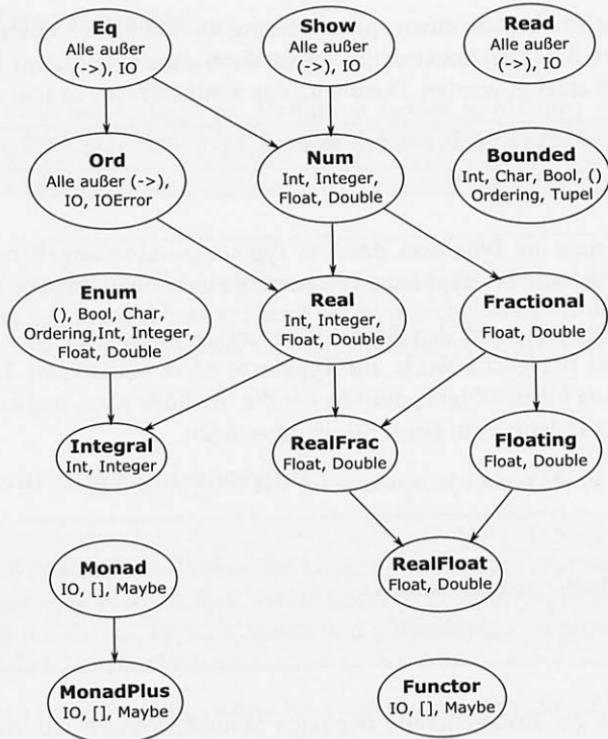


Abb. 7.1 Definierte Klassen der Prelude und ihre Zugehörigkeiten (s. [58])

Alternativ können wir auch beide Operatoren überschreiben:

```

instance Eq Boolean where
  T == T = True
  F == F = True
  _ == _ = False
  T /= F = True
  F /= T = True
  _ /= _ = False
  
```

In einigen Fällen gibt es auch Vorbedingungen für die Instanziierung. So muss eine Klasse bereits Instanz der Klasse Eq sein, bevor wir sie der Klasse Ord hinzufügen können. Für Instanzen der Klasse Ord müssen wir entweder (\leq) oder compare angeben. Eine von beiden genügt, obwohl noch weitere Funktionen von Ord bereitgestellt werden. Für alle anderen gibt es bereits Defaultimplementierungen, die uns das Leben sehr erleichtern.

Da wir den Datentyp Boolean bereits zur Instanz der Klasse Eq hinzugefügt haben, können wir Boolean jetzt auch als Instanz der Klasse Ord vereinbaren:

```
instance Ord Boolean where
  F <= T = True
  x <= y = x == y
```

Es gibt eine ganze Reihe von Typklassen mit Abhängigkeiten, die in der Prelude definiert werden (s. dazu Abb. 7.1).

Die vorgestellten Beispiele sollten ausreichend sein, um mit Hilfe dieser Abbildung und der Dokumentation von Haskell alle eigenen Instanziierungen vornehmen zu können.

7.7 Projekt: Symbolische Differentiation

Im folgenden Projekt wollen wir die in diesem Kapitel bereits gewonnenen Fähigkeiten zur Instanziierung von Datentypen zu Typklassen anwenden und weiter vertiefen. Dazu ist etwas Schulmathematik notwendig, aber ein wenig Wiederholung schadet sicher nicht.

Wenn wir eine Funktion definiert haben, die polymorph auf den Typen einer Typklasse C ist, sagt das bereits eine Menge über den Aufbau der Funktion aus. Zunächst scheint das nicht ganz intuitiv zu sein. Dadurch, dass nur die in der Typklasse vereinbarten Funktionen verwendet werden können, lassen sich aber einige interessante Dinge anstellen. Beispielsweise können wir für viele Funktionen automatisch die entsprechenden Umkehrfunktionen finden.

Wir wollen in diesem Abschnitt eine Funktion schreiben, die automatisch die Ableitung von Funktionen des Typs $\text{Num} \rightarrow \text{Num}$ bestimmt. Dabei wollen wir keine numerische Differentiation durchführen, sondern symbolisch ableiten, wie es aus der Schule bekannt sein sollte. Um es uns etwas einfacher zu halten, werden wir nur solche Ableitungen generieren, die überall definiert sind. Fallunterscheidungen, wie etwa beim Absolutwert, nehmen wir nicht vor.

Durch `Num` werden die Operationen `+`, `*`, `-` sowie `negate`, `abs` und `signum` bereitgestellt. Die letzten beiden werden wir aber aus den zuvor genannten Gründen ignorieren.

Damit wir eine Funktion ableiten können, müssen wir zunächst herausfinden, welche Operationen die Funktion anwendet. Wenn wir einen Wert an die Funktion übergeben und untersuchen, was diese zurückgibt, können wir nicht ableiten, um welche Funktion es sich handelt. Gibt uns die Funktion beispielsweise für $f 2$ den Wert 4 zurück, gibt es unendlich viele Möglichkeiten, wie die Funktion aussehen kann. Hier nur eine kleine Auswahl:

$$f x = 2x, \quad f x = x^2, \quad f x = x^3 - 2x, \quad \dots$$

7.7.1 Operatorbaum

Aus diesem Grund wenden wir einen Trick an. Wir wissen aufgrund der Polymorphie der Funktion, dass sie nur eine kleine Menge von Funktionen verwenden kann, um ihr Ergebnis zu berechnen. Außerdem arbeitet die Funktion auf allen Werten aus der Klasse Num. Wir können uns also einen neuen Datentypen schreiben, der alle möglichen Funktionen als Operatorbaum abbilden kann und ihn zu einer Instanz der Klasse Num machen. Das können wir machen, obwohl der Typ eigentlich keine Zahl ist, denn die Semantik der Operationen wird von uns bestimmt.

Wenn wir ein Exemplar dieses Typs in die Funktion stecken, gehen keine Informationen mehr verloren. Die Funktionsweise der Funktion wird greifbar als Operatorbaum abgebildet, auf dem wir dann die nötigen Transformationen durchführen können, um die Ableitung zu bilden.

Der Datentyp benötigt einen Konstruktor für Konstanten, einen für Variablen und je einen für jede Operation, die von der Klasse Num zur Verfügung gestellt wird.

Unser Datentyp könnte also beispielsweise so aussehen:

```
data Comp a =
  X |           -- Variablen
  Z a |         -- Konstanten
  (Comp a) :+: (Comp a) |
  (Comp a) :*: (Comp a) |
  (Comp a) :-: (Comp a)
deriving (Show, Eq)
```

An diesem Beispiel sehen wir im Übrigen auch, dass Datenkonstruktoren, wenn diese aus Symbolen bestehen, auch wie Operatoren infix geschrieben werden können. Durch Umschließen mit accent grave können, analog zu Funktionen, beliebige Datenkonstruktoren in Infixschreibweise angegeben werden.

Diesen Datentypen können wir jetzt ganz einfach zu einer Instanz der Klasse Num machen. Funktionen, die sich nicht ableiten lassen, bleiben dabei undefiniert:

```
instance Num a => Num (Comp a) where
  (+) = (:+:)
  (*) = ( :*: )
  (-) = ( :-: )
  fromInteger = Z . fromInteger
```

Bisher noch unbekannt ist die Funktion fromInteger. Sie dient dazu, eine Instanz eines Zahlentypen aus einem Integer heraus zu erzeugen und hat den Typ:

```
fromInteger :: (Num a) => Integer -> a
```

Um zu verstehen, was die Zeile `fromInteger = Z . fromInteger` macht, müssen wir uns bewusst werden, dass `fromInteger`, je nachdem in welchem Typkontext es aufgerufen wird, unterschiedlich arbeitet.

Schreiben wir zum Beispiel `fromInteger 5` in einem Kontext, der einen Double Wert erwartet, wird 5 zu einem Double umgewandelt. In einem Kontext, der einen Int erwartet, erhalten wir hingegen eine Ganzzahl:

```
Hugs> (fromInteger 5)::Double
5.0
```

```
Hugs> (fromInteger 5)::Int
5
```

In unserem Fall wollen wir sagen, dass unser Typ `Comp a` immer dann zu einer Instanz von `Num` gemacht werden kann, wenn `a` etwas aus `Num` ist. Das verwenden wir um `fromInteger` zu definieren. Da `a` aus der Klasse `Num` ist, muss es eine `fromInteger` Funktion für `a` geben. Die wenden wir einfach auf den `Integer` an, den wir als Eingabe bekommen und schreiben das Ergebnis hinter den Konstruktor `Z`. So haben wir dann ein Exemplar eines `Comp a` aus einem `Integer` erstellt und die Funktionalität von `fromInteger` abgebildet.

7.7.2 Polynome berechnen

Jetzt definieren wir uns eine Funktion `f`, die Polynome ausrechnet, um unseren neuen Datentypen gleich auszuprobieren. Sie ist wie folgt definiert:

```
-- in c sind die Koeffizienten aufsteigend angegeben
-- f = c_0 * x^0 + c_1 * x^1 + ... + c_n * x^n
f c x =
  sum $
  zipWith (*) c $
  zipWith (^) (repeat x) [0..length c]
```

Das wollen wir gleich mal auf der Konsole testen:

```
Hugs> :t f [1,2,3]
f [1,2,3] :: Num a => a -> a
```

```
Hugs> f [1,2,3] x
((z 0 :+: (z 1 :*: z 1)) :+: (z 2 :*: x)) :+: (z 3 :*: (x :*: x))
```

Wie erwartet ist $f[1,2,3] = 0 + 1 \cdot x^0 + 2 \cdot x^1 + 3 \cdot x^2$.

7.7.3 Ableitungsregeln

Jetzt haben wir die Funktion konkret als Wert vom Typen `Comp a` vorliegen und können damit machen, was wir wollen. Insbesondere können wir die bekannten Ableitungsregel darauf anwenden.

Eine kleine Übersicht zur Erinnerung:

$$\begin{aligned}\frac{dc}{dx} &= 0 \\ \frac{dx}{dx} &= 1 \\ \frac{df(x) + g(x)}{dx} &= \frac{df(x)}{dx} + \frac{dg(x)}{dx} \\ \frac{df(x) - g(x)}{dx} &= \frac{df(x)}{dx} - \frac{dg(x)}{dx} \\ \frac{df(x) * g(x)}{dx} &= \frac{df(x)}{dx} * g(x) + \frac{dg(x)}{dx} * f(x)\end{aligned}$$

Das können wir einfach abschreiben und schnell in Haskell übertragen:

```
diff' (Z n)      = Z 0
diff' X          = Z 1
diff' (c :+: c2) = diff' c :+: diff' c2
diff' (c :-: c2) = diff' c :-: diff' c2
diff' (u :*: v)   = (diff' u :*: v) :+: (u * diff' v)
```

7.7.4 Automatisches Auswerten

Jetzt müssen wir nur noch den konkreten Operatorbaum wieder zurücktransformieren. Dazu schreiben wir uns eine eval-Funktion, die den Ausdruck wieder auswertet. Dann können wir die Ableitung auch gleich hinschreiben:

```
eval (Z n) _      = n
eval X x          = x
eval (c :+: c2) x = eval c x + eval c2 x
eval (c :-: c2) x = eval c x - eval c2 x
eval (c :*: c2) x = eval c x * eval c2 x

diff f = eval (diff' (f X))
```

Die Funktionen diff' und eval hätten auch in eine Funktion zusammengeführt werden können. Mit der getrennten Definition können wir aber dazwischen noch symbolische Vereinfachungen des entstehenden Terms vornehmen. Das wird Teil einer Übungsaufgabe sein.

Um uns zu überzeugen, dass alles korrekt funktioniert, können wir uns die entstehenden Funktionen ja nochmal als Operatorbaum ansehen:

```
Hugs> let f x = 2*x in diff f X
(Z 0 :*: X) :+: (Z 2 :*: Z 1)
```

```
Hugs> let f x = 2*x*x in diff f X
((z 0 :*: x) :+: (z 2 :*: z 1)) :*: x) :+: ((z 2 :*: x) :*: z 1)
```

```
Hugs> let f x = 2*x*x+3*x+2 in diff f X
(((z 0 :*: x) :+: (z 2 :*: z 1)) :*: x) :+: ((z 2 :*: x) :*: z 1))
:+: ((z 0 :*: x) :+: (z 3 :*: z 1))) :+: z 0
```

Die entstehenden Ausdrücke sind kompliziert, aber richtig. Es lohnt sich an dieser Stelle innezuhalten und die Ergebnisse selbst nachzuvollziehen.

7.8 Eigene Klassen definieren

Jetzt können wir unsere Datentypen zu Instanzen von vordefinierten Klassen machen. Natürlich können Typklassen auch selbst definiert werden, wenn die vorhandenen nicht das Benötigte abdecken.

Glücklicherweise ist es ganz leicht, Typklassen in Haskell zu vereinbaren. Dazu verwenden wir das Schlüsselwort `class`. Es folgt der Name der Typklasse und ein Platzhalter für den Typen. Dann werden die Typen der geforderten Funktionen und falls gewünscht noch Defaultimplementierungen angegeben. Bei der Definition der Defaultimplementierungen können bereits alle Funktionen der Klasse verwendet werden. Dadurch ist es etwa in der Typklasse `Eq` möglich (`==`) mit (`/=`) und (`/=`) mit (`==`) zu definieren.

Optional können vor dem Klassennamen noch andere Klassen stehen. Die Typen, die zu einer Instanz der neuen Klasse gemacht werden sollen, müssen bereits Instanzen dieser Klassen sein:

```
class (B,C) => Name platzhalter where
  funktion1 :: a -> platzhalter
  funktion2 :: platzhalter -> Bool
  funktion3 :: platzhalter -> platzhalter -> Int

  -- Defaultimplementierungen
  funktion2 _ = False
  funktion3 _ _ = 42
```

Eigene Typklassen zu schreiben, ist eine gute Möglichkeit, um zwischen der Implementierung eines Datentyps und seiner Verwendung zu unterscheiden. Die Repräsentation des Datentyps kann leicht geändert werden, ohne dass der gesamte Code der ihn verwendet mitgeändert werden muss. Das ist sehr wichtig, da in den seltens-ten Fällen zu Beginn alle Anforderungen für ein Programm bekannt sind.