Typklassen und Operatoren in Haskell

Leo Mommers
Vincent Hilla

1 Einleitung

In der objektorientierten Programmiersprache C# ist es möglich, die Addition mit dem Zeichen `+`, Vergleiche mit `==` sowie andere Sonderzeichen für eine beliebige Klasse zu überschreiben. Schreiben wir beispielsweise eine Klasse, die einen Spaltenvektor repräsentiert, so können wir für diese Vektoren die komponentenweise Addition, die Skalarmultiplikation sowie andere mathematische Operationen definieren und sie mit Symbolen wie `+` oder `*` in Infixschreibweise verwenden. Diese Funktionalität beruht auf zwei Konzepten: Zum Ersten auf dem Konzept von Klassen und ihren zugehörigen Methoden; zum Zweiten auf der Möglichkeit, diese Sonderzeichen überhaupt als Methoden zu definieren.

In Haskell sind uns beide Konzepte bisher nicht bekannt. Das Einzige, was annähernd an das Konzept von Klassen herankommt, sind Datentypen, die meist mit `data` definiert werden. Doch auch hier können wir keine Funktionen schreiben, die explizit zu diesem Typen gehören.

Diese sogenannte *Operatorüberladung* und mehr können wir jedoch auch in Haskell umsetzen. Dafür benötigen wir zwei Funktionalitäten dieser Sprache: Typklassen und Operatoren. Wie wir diese geschickt verwenden, ist das Thema dieser Ausarbeitung.

Fangen wir zunächst mit dem grundlegenderen der beiden Themen an, den Operatoren.

2 Operatoren

In Haskell werden Funktionen standardmäßig in der Präfixschreibweise verwendet. Dies kann jedoch in einigen Fällen zu schwer lesbarem und verwirrendem Code führen. Beispielsweise ist `13 + 5` vor allem in komplexeren Ausdrücken für eine Addition leichter verständlich als `plus 13 5`. Bei dem in Infixnotation verwendeten `+` handelt es sich um einen sogenannten *Operator*. Wie wir solche Operatoren in Haskell implementieren, soll das Thema dieses Abschnittes sein.

2.1 Konstruktoren

Es wird zwischen zwei verschiedenen Typen von Operatoren unterschieden: *Konstruktoren* und *Variablen*. Wir wollen zunächst auf erstere eingehen, bevor wir uns ausführlicher mit Variablen beschäftigen.

Das Konzept von Konstruktoren ist uns bereits bekannt. Es handelt sich dabei um die von uns definierten Zeichenfolgen, die bei der Erstellung eines Datentyps mithilfe des Schlüsselwortes `data` verwendet werden. Wir können zum Beispiel $\mathbb{N}_0$ durch

```
1 data Nats = Zero | Succ Nats
```

beschreiben. Hier sind `Zero` und `Succ` die beiden Konstruktoren, mit deren Hilfe wir eine Instanz vom Typ `Nats` beschreiben.

Bei `Zero` und `Succ` handelt es sich jedoch nicht um Operatoren. Operatoren zeichnen sich nämlich durch zwei Eigenschaften aus: Sie werden standardmäßig in Infixschreibweise

verwendet und ihre Namen bestehen ausschließlich aus Sonderzeichen. Bei `Zero` und `Succ` ist dies beides nicht der Fall.

Um einen Operator als Konstruktor zu verwenden, schreiben wir diesen einfach zwischen zwei Parameter. Eine Menge von ganzen Zahlen lässt sich so zum Beispiel mit

```
data Set = Empty | Set :+ Int
```

darstellen. Die Menge $\{1, 2, 3\}$ entspräche in Haskell dann dem Ausdruck `Empty :+ 1 :+ 2 :+ 3`. Der Datentyp `Set` weist hier zwei Konstruktoren auf: `Empty` und `:+`. Bei letzterem handelt es sich um einen Operator, da er die oben genannten Eigenschaften aufweist.

Jeder Operator, der gleichzeitig ein Konstruktor ist, beginnt in Haskell mit `:`, gefolgt von beliebigen Sonderzeichen. Ein bekanntes Beispiel für einen derartigen Operator ist der Listenkonstruktor `:`, der ebenfalls in Infixnotation verwendet wird.

2.2 Präfix- und Infixnotation

Mathematisch gesehen ist jedes Element in einer Menge nur einmal vorhanden. In unserer Implementierung ist es jedoch möglich, dass eine Menge ein Element zweimal beinhaltet. Dadurch ist der Datentyp `Set` anfälliger für Fehler, beispielsweise bei der Implementierung eines Vergleichsoperators. Um den Typen intuitiver zu gestalten, führen wir die Funktion `add` ein, die ein Element nur dann in eine Menge einfügt, wenn diese es noch nicht enthält.

```
add :: Set -> Int -> Set
add s i = if contains s i then s else s :+ i
```

mit

```
contains :: Set -> Int -> Bool
contains Empty _ = False
contains (s :+ e) i = if e == i then True else contains s i.
```

Nun können wir, um die Menge $\{1, 2, 3\}$ zu erzeugen, auch `add (add (add Empty 1) 2) 3` schreiben. Diese Schreibweise ist allerdings deutlich unübersichtlicher und aufwendiger als `Empty :+ 1 :+ 2 :+ 3`. Dies liegt hauptsächlich daran, dass `add` in Präfixnotation verwendet wird.

Glücklicherweise ist es in Haskell möglich, jede zweistellige Funktion auch in Infixschreibweise zu verwenden, indem wir sogenannte *Backquotes* ` um den Funktionsnamen setzen. Statt `add Empty 1` ist es dann möglich, `Empty `add` 1` zu schreiben. $\{1, 2, 3\}$ kann also mit `Empty `add` 1 `add` 2 `add` 3` erzeugt werden.

Doch auch diese Schreibweise hat ihre Nachteile: Sie ist äußerst mühsam und kann vor allem in komplexeren Ausdrücken zu Verwirrung führen, wenn Präfix- und Infixnotationen gemischt auftreten. Daher schreiben wir zusätzlich folgende Funktion, die nichts anderes tut, als `add` aufzurufen:

```
(<<) :: Set -> Int -> Set
(<<) s i = add s i.
```

`add Empty 1` ist nun also äquivalent zu `(<<) Empty 1`. Dies ist jedoch nicht die Verwendung, für die die Funktion `(<<)` vorgesehen ist. Hat eine Funktion nämlich einen Namen, der ausschließlich aus Sonderzeichen besteht, handelt es sich dabei um einen Operator. Wie in Abschnitt 2.1 erläutert, werden Operatoren standardmäßig in Infixschreibweise verwendet - so ist es auch bei `(<<)` der Fall. Dazu werden die runden Klammern weggelassen, die ohnehin nur besagen, dass der Operator hier entgegen seiner Natur in Präfixnotation verwendet wird - ähnlich wie die Backquotes bei Funktionen besagen, dass sie in Infixnotation verwendet werden. Statt `Empty :+ 1 :+ 2 :+ 3` können wir nun also auch `Empty << 1 << 2 << 3` schreiben, wobei letzteres die Funktion `add` zum Einfügen von Elementen

in die Menge benutzt. `Empty << 1 << 2 << 3 << 1` wertet allerdings im Gegensatz zur Schreibweise mit `:+` auch zur Menge $\{1, 2, 3\}$ aus, da `add` vor dem Einfügen auf doppelt vorkommende Elemente überprüft.

Während Konstruktoroperatoren immer mit einem `:` beginnen, müssen Variablenoperatoren mit einem beliebigen anderen Sonderzeichen anfangen. Ansonsten müssen Variablenoperatoren ausschließlich aus Sonderzeichen bestehen. Bekannte Beispiele für vordefinierte Variablenoperatoren sind der Gleichheitsoperator `==` und die mathematischen Operatoren `+`, `-`, `*` und `/`.

Es sei angemerkt, dass die Sonderzeichen, aus denen Operatornamen bestehen dürfen, gewissen Einschränkungen unterliegen. Es dürfen nur die Zeichen `!#$%&*+./<=>?@^|-~` verwendet werden.

2.3 Assoziation

Wir führen einen neuen Operator ein, der ein Element aus einer Menge entfernt:

```
(-:) :: Set -> Int -> Set
Empty -: _ = Empty
(s :+ e) -: i = if e == i then s else (s -: i) << e.
```

Zusätzlich schreiben wir einen Operator, der die Differenzmenge zweier Mengen berechnet:

```
(\\) :: Set -> Set -> Set
Empty \\ _ = Empty
s \\ Empty = s
sa \\ (sb :+ eb) = (sa -: eb) \\ sb
```

Diese Deklarationen werfen jedoch einige Probleme auf. Das erste dieser Probleme ist, in welche Richtung die Operatoren assoziieren, das heißt wie die Klammersetzung aussehen soll. Die mathematischen Ausdrücke $(\{1, 2, 3\} \setminus \{1\}) \setminus \{2\} = \{3\}$ und $\{1, 2, 3\} \setminus (\{1\} \setminus \{2\}) = \{2, 3\}$ sind offensichtlich nicht identisch. Haskell assoziiert standardmäßig von links nach rechts, weshalb es den Aufruf

```
(Empty << 1 << 2 << 3) \\ (Empty << 1) \\ (Empty << 2)
```

als

```
((Empty << 1 << 2 << 3) \\ (Empty << 1)) \\ (Empty << 2)
```

interpretiert. Zu Zwecken der Deutlichkeit kann es dennoch sinnvoll sein, die Zeile

```
infixl \\
```

in den Quellcode einzufügen. Diese sagt dem Compiler, dass der Operator `\\` von links nach rechts assoziiert, so, wie es auch Standard ist. Möchten wir eine Assoziation von rechts nach links festlegen, können wir dies mit `infixr \\` tun. Mit `infix \\` wirft Haskell eine Fehlermeldung auf, wenn der Operator in einem Ausdruck mehr als einmal hintereinander vorkommt.

Nun zum zweiten der beiden Probleme. Wir betrachten den Ausdruck

```
Empty << 1 -: 1 << 2 \\ Empty << 2.
```

Wenn Haskell diesen Ausdruck auswertet, geht es wie üblich von links nach rechts vor. Er steht also für

$$(((\{1\} \setminus \{1\}) \cup \{2\}) \setminus \emptyset) \cup \{2\} = \{2\}.$$

Wir möchten jedoch, dass `<<` und `-:` eine höhere Bindungspriorität besitzen als `\\`, so wie `*` und `/` stärker binden als `+`. So werden die Mengen auf beiden Seiten des Differenzoperators `\\` zuerst instantiiert und erst dann wird die Differenz gebildet. Mathematisch gesehen evaluiert obiger Ausdruck also zu $\{2\} \setminus \{2\} = \emptyset$.

Um diese Bindungsprioritäten festzulegen, können wir bei den Infixdeklarationen mit `infixl`, `infixr` und `infix` eine Zahl zwischen einschließlich 0 und 9 angeben, welche eben diese Priorität repräsentiert. Im Code könnte das zum Beispiel so aussehen:

```
infixl 8 \\
infixl 9 <<, -:.
```

Damit werden `<<` und `-:` vor `\\` ausgeführt und Haskell berechnet aus dem obigen Ausdruck die leere Menge $\emptyset$ beziehungsweise `Empty`. `infixl 9` kann allerdings auch weggelassen werden, da Haskell diese Werte als Standard nimmt, wenn nichts angegeben wird.

2.4 Partiell ausgewertete Operatoren

Es sei eine Funktion `mult :: Int -> Int -> Int` gegeben, die das Produkt ihrer beiden Argumente zurückgibt. Mit dem Ausdruck `mult 3` erhalten wir eine Funktion vom Typ `Int -> Int`, die ihr Argument mit 3 multipliziert. Solche partiellen Auswertungen sind auch bei Operatoren möglich. Mit `Empty << 1 << 2 << 3 -:` erhalten wir die Funktion, die ihr Argument vom Typ `Int` aus der Menge $\{1, 2, 3\}$ entfernt. Hingegen entfernt `-: 2` die Zahl 2 aus seinem Argument vom Typ `Set`.

3 Typklassen

Der folgende Abschnitt thematisiert nun *Typklassen* in Haskell, welche vergleichbar mit Klassen in der Objektorientierung sind und spezifische Funktionen für eine festgelegte Auswahl von Typen anbieten sollen. In diesem Kapitel wollen wir erläutern, wie solche angelegt sowie instantiiert werden und welche neuen Möglichkeiten sich mithilfe dieser Programmiertechnik erschließen.

Im letzten Kapitel haben wir als Beispiel Mengen vom Typ `Int` verwendet. Nun wäre es aber wünschenswert, Mengen als allgemeine Datenstruktur zu verwenden, wir ändern also die Definition folgendermaßen ab:

```
data Set a = Empty | (Set a) :+ a
```

Natürlich ändern sich damit dann auch die Typdeklarationen aller Funktionen, wobei neue Probleme auftreten. Beispielsweise benötigt die Funktion `contains` den Operator `==`, welcher jedoch nicht für jegliche Typen definiert ist. Die Typdeklaration muss also im Beispiel `contains` folgendermaßen angepasst werden:

```
contains :: Eq a => Set a -> a -> Bool
```

Der Term `Eq a =>` steht hier für einen *Kontext* und legt fest, dass nur Typen für `a` eingesetzt werden dürfen, die zur Typklasse `Eq` gehören und somit den Operator `==` definiert haben.

3.1 Syntax

3.1.1 Typklassendeklaration

Um in Haskell Typklassen anzulegen wird das Schlüsselwort `class` verwendet, dabei erfolgen *Typklassendeklarationen* immer auf oberster Programmebene. Auf dieses Schlüsselwort folgt der mit einem Großbuchstaben beginnende Name der Typklasse und anschließend eine kleingeschriebene Typvariable.

Um nun ein genauere Definition zu ermöglichen kann das Schlüsselwort `where` verwendet werden, auf welches wiederum ein Körper aus mindestens einer Typdeklaration oder Funktionsdeklaration folgen muss. Letztere sind dabei jedoch optional, die Deklaration darf auch in der später thematisierten Instanzendeklaration folgen. Dieser Körper wird

entweder durch Einrücken und Zeilenumbrüche oder durch `{`, `}`, sowie `;` gekennzeichnet. Die Funktionen innerhalb einer Typklasse heißen dabei *Methoden*.

Eine Typklassendeklaration hat also beispielsweise eine der folgenden Formen:

```
class KlasseA a where
    beispielfunktion :: a -> Bool
class KlasseB a where {func :: a -> Bool;}
class KlasseC var
```

3.1.2 Instanzendeklaration

Zur *Instanzendeklaration* wird hingegen das Schlüsselwort `instance`, ebenfalls auf oberster Programmebene, verwendet. Auf dieses folgt der Name der Typklasse und statt einer Typvariable nun der Name des Typen, wie `Int` oder `Bool`, der als Instanz dieser Klasse definiert werden soll. Hierauf folgt wieder das Schlüsselwort `where` und alle nötigen Funktionsdeklarationen.

Dabei müssen alle Funktionen der Typklasse, die aus der Typklassendeklaration keine Standardimplementierung besitzen, hier implementiert werden. Jedoch ist es ebenso möglich eine Funktionsdeklaration anzugeben, trotz einer bereits in der Typklassendeklaraion bestehenden. Dadurch wird die originale Funktionalität überschrieben.

Eine Instanzendeklaration hat also beispielsweise eine der folgenden Formen:

```
instance KlasseA Int where
    beispielfunktion a = True
instance KlasseB Int where {func 0 = False;func a = True;}
instance KlasseC Int
```

3.1.3 Mengenbeispiel

Um zum Mengenbeispiel aus Kapitel 2 zurückzukommen, wollen wir nun den Gleichheitsoperator für den Datentyp `Set` implementieren.

Mithilfe des folgenden Codes wird nun eine Typklasse `Eq` erstellt, zu welcher der Operator `==` gehört und `Set` als Instanz dieser Typklasse festgelegt. Wie später thematisiert wird, ist diese Typklasse in Haskell bereits standardmäßig implementiert.

```
class Eq a where
    (==) :: a -> a -> Bool
instance Eq a => Eq (Set a) where
    Empty == Empty    = True
    Empty == _        = False
    (m1 :+ e) == m2   = contains m2 e && m1 == (m2 -: e)
```

Zwei Mengen A, B werden dabei hier verglichen, indem für $n \in A$ überprüft wird, ob $n \in B$ gilt und rekursiv $A \setminus \{n\} = B \setminus \{n\}$ untersucht wird. Über `Eq a =>` wird wieder ein Kontext angegeben, worauf wir später zurückkommen wollen.

3.2 Überladung

Im letzten Abschnitt haben wir den Operator `==` für den Typ `Set` implementiert und diesen somit überladen. Implementierungen für andere Typen, wie `Int` bleiben also erhalten. In Haskell erfolgt Überladung immer mithilfe von Typklassen, indem die Funktion als Methode einer solchen implementiert wird, um Uneindeutigkeit, wie im folgenden Beispiel zu vermeiden.

```
(==) :: Eq a => Set a -> Set a -> Bool
Empty == Empty = True
(a :+ n) == b  = contains b n && a == (b -: n)
_ == _ = False
```

Hier wurde der bereits für andere Typen, wie `Int` definierte Operator `==` für `Set` nicht als Methode sondern auf oberster Programmebene als Funktion implementiert, nur ist für den Compiler nun nicht klar, ob die letzte Zeile sich auf die Implementierung für `Set` oder doch auf eine für einen anderen Typen bezieht.

3.3 Vererbung und Kontext

Bereits am Anfang diese Kapitels haben wir gesehen, dass ein Kontext bei Funktionen nötig sein kann. Diese können neben Funktionen auch bei Typklassen-, sowie Instanzendeklaration Anwendung finden.

Betrachten wir zuerst noch einmal genauer Kontexte bei Funktionen, nun am folgenden Operator zum Entfernen von Elementen aus Mengen beliebiger Typen. Für Mengen vom Typ `Int` ist dieser Operator bereits in Kapitel 2.3 vorgestellt worden.

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

Durch den Kontext `Eq a =>` wird festgelegt, dass für die Typvariable `a` nur Typen eingesetzt werden dürfen, die der Typklasse `Eq` angehören und somit den Operator `==` besitzen. Eine Funktion kann jedoch auch mehr als einen Kontext besitzen (sowohl für verschiedene als auch für einzelne Variablen), dazu werden Klammern verwendet: `...(Eq a, Eq b) =>...`

In Fällen, wo Methoden von Typklassen verwendet werden, wie beispielsweise `==`, `<`, `show`, `abs` muss also immer der zugehörige Kontext angegeben werden. Andernfalls könnte es beispielsweise zum Vergleich `(x->2*x) < (x->(-2)*x)` kommen, wenn eine Methode `<` auf zwei Variablen anwendet. Ausnahme bilden nur Funktionen innerhalb von Typklassendeklarationen:

```
class Eq a where
    /= :: a -> a -> Bool
```

Die Typvariable `a` hat hier bereits durch die Typklassendeklaration den Kontext `Eq`, also ist es nicht nötig diesen bei der Typdeklaration von `/=` anzugeben. Es wird auch von einem *impliziten* Kontext gesprochen.

In Abschnitt 3.1.3 wurde der Typ `Set` als Instanz der Klasse `Eq` unter Verwendung eines Kontextes `Eq a =>` definiert. Dies ist ein gutes Beispiel für Kontext bei Instanzendeklarationen, denn die dort verwendete Implementierung von `==` verwendet die Funktion

```
contains Eq a => Set a -> a -> Bool
```

Es wird also gefordert, dass die Menge nur mit `==` vergleichbare Typen enthält. Wird diese Funktion nun in einer Instanzendeklaration verwendet, so muss diese Forderung erfüllt, also in der folgenden Form umgesetzt werden:

```
instance Eq a => Eq (Set a) where...
```

Sehr ähnlich dazu sehen Kontexte in Typklassendeklarationen aus, nun am Beispiel einer Typklasse `Ord` für Operatoren, wie `<` und `>`:

```
class Eq a => Ord a where...
```

Hier realisiert der Kontext eine gewisse Vererbung. Da $x \leq y \wedge x \geq y$ auch $x = y$ impliziert, macht es Sinn zu fordern, dass alle Typen der Typklasse `Ord`, also für die `<=`,`>=`,... definiert sind, auch Typen der Typklasse `Eq` sind, also die Methoden `==` und

`/=` besitzen. Zwar muss in Haskell für jeden Typ dennoch eine Instanzendeklaration erst für die Typklasse `Eq` und dann darauf folgend für `Ord` angegeben werden, aber nun ist sichergestellt, dass jeder Typ der Typklasse `Ord` auch ein Typ der Typklasse `Eq` ist. Ein Kontext wie `(Eq a, Ord a) =>` kann daher auf `Ord a =>` verkürzt werden.

3.4 Vordefinierte Typklassen

Die in Abschnitt 3.1.3 implementierte Typklasse `Eq` ist in Haskell tatsächlich vordefiniert. Sie ist dabei als eine von vielen bereits im Haskell Prelude enthalten. Im folgenden wollen wir auf die Typklassen `Eq`, `Ord`, `Num` und `Show` eingehen, literarische Grundlage bildet dafür die *Haskell Prelude Documentation* [Pre].

3.4.1 Eq

```
1 class Eq a where
2     (==), (/=) :: a -> a -> Bool
```

Diese Typklasse besitzt die zwei Operatoren `==` und `/=` für Gleichheit beziehungsweise Ungleichheit zweier Werte. Logischerweise reicht es hier bei einer Instanzendeklaration eine der beiden Operatoren zu definieren, da der jeweils andere automatisch als Negation implementiert wird.

3.4.2 Ord

```
1 class Eq a => Ord a where
2     (<), (<=), (>), (>=) :: a -> a -> Bool
3     min, max :: a -> a -> a
4     compare :: a -> a -> Ordering
```

Diese Typklasse realisiert Vergleichsoperatoren, wobei aufgrund des angegebenen Kontextes `a` auch Instanz der Typklasse `Eq` sein muss. Minimal benötigt wird entweder die Methode `compare` oder `<=`, da aus diesen zusammen mit den Operatoren der Typklasse `Eq` alle anderen Operatoren hergeleitet werden können. Die Funktionen `min`, `max` geben den minimalen bzw. maximalen zweier Werte zurück und die Funktion `compare` liefert ein `Ordering`. Dies ist ein Datentyp, der die numerische Ordnung zweier Werte repräsentiert.

3.4.3 Num

```
1 class Num a where
2     (+), (-), (*) :: a -> a -> a
3     negate, abs, signum :: a -> a
4     fromInteger :: Integer -> a
```

Diese Typklasse steht übergeordnet für die numerischen Typen in Haskell, also u.a. `Int` und `Double`. Bei einer Instanzendeklaration müssen alle Funktionen angegeben werden, abgesehen von `negate` und `-`, da diese ineinander umwandelbar sind, es reicht also eine der beiden. Die Funktion `negate` soll dabei das Vorzeichen invertieren, `abs` dieses entfernen und `signum` das Vorzeichen beispielsweise als -1, 0 oder 1 angeben. Die Methode `fromInteger` soll ein Element dieses Typen aus einem `Integer` bilden. Dies ist ein Datentyp für beliebig große Ganzzahlen und Instanz der Typklasse `Num`, die Operatoren `-`, `+`, `*` sind also verwendbar.

3.4.4 Show

```
class Show a where
    show :: a -> String
    showList :: [a] -> String
```

Diese Klasse stellt Funktionen zur Darstellung von Datentypen als Text bereit, wodurch somit auch die Ausgabe dieser in der Konsole ermöglicht wird. Implementiert werden muss dabei nur die Funktion `show`, denn für die Methode `showList` existiert eine Standardimplementierung aufbauend auf `show`.

Für unser Mengenbeispiel wäre folgende Implementierung möglich:

```
class Show a => Show (Set a) where
    show Empty = "{}"
    show s = "{" ++ elemente s ++ "}"
        where
            elemente (Empty :+ e) = (show e)
            elemente (s :+ e) = (show e) ++ ", " ++ (elemente s)
```

Diese Implementierung wandelt den Datentyp `Set` mithilfe von Rekursion in die typische Mengenschreibweise um. Dabei werden die einzelnen Elemente in Text umgewandelt, durch Komma getrennt und am Ende von geschweiften Klammern umschlossen.

3.5 Sonderfall deriving

Der folgende Teil basiert auf dem *Haskell Online Report* [HDI].

Mithilfe des Operand `deriving` kann bei der Deklaration eines Datentypen dieser direkt Typklassen als Instanz zugewiesen werden. Der Compiler leitet dann selbstständig Implementierung für die Methoden dieser Typklassen her, dies funktioniert jedoch nur mit einer kleinen Auswahl von vordefinierten Typklassen, u.a. denen aus Kapitel 3.4.

Die Funktionen werden dabei aus der Struktur der `data` Deklaration abgeleitet, welche folgendermaßen formalisiert aufgefasst wird:

$$data\ T\ a_1...a_n = K_1\ p_{11}...p_{1k_1} \mid ... \mid K_m\ p_{m1}...p_{mk_m}\ deriving\ (C_1, ..., C_o)$$

Hierbei steht `T` für den Typnamen, `a` für die Typvariablen, `K` für die Typkonstruktoren, `p` für deren Parameter und `C` für die Typklassen, welchen dieser Datentyp angehören soll.

Der Compiler generiert dann darauf aufbauend Instanzen- und Methodendeklarationen, deren Logik auf dem Syntax der `data` Deklaration basiert. Die Limitationen dessen lassen sich an folgenden Beispiel beobachten.

Aus `data Set a = Empty | (Set a) :+ a deriving Eq` würde der Compiler Instanzen- und Methodendeklarationen ableiten, die folgende Aufrufe ermöglichen würden, wobei das zweite Ergebnis aufgrund der rein syntaktischen Interpretation bezogen auf Mengenäquivalenz falsch ist:

```
*Main> (Empty :+ 1 :+ 2) == (Empty :+ 1 :+ 2)
True
*Main> (Empty :+ 1 :+ 2) == (Empty :+ 2 :+ 1)
False
```

3.6 Beispiel: Allgemeine Datenstrukturen

Zum Ende diese Kapitels wollen wir Datenstrukturen, wie Mengen allgemeiner betrachten. Dabei soll dieses Beispiel die wichtigsten Inhalte dieses Kapitel nochmals aufgreifen.

Wir beginnen mit der Definition der Typklasse `DataStruc`, wobei wir für die Funktion `isEmpty` direkt eine Standardimplementierung angeben:

```haskell
class DataStruc s where
    add, remove :: a -> s a -> s a
    first :: s a -> Maybe a
    isEmpty :: s a -> Bool

    isEmpty s = isNothing (first s)
        where isNothing Nothing = True
              isNothing _ = False
```

Die Methoden `add` und `remove` sollen nun Elemente zur Datenstruktur hinzufügen beziehungsweise entfernen während die Methode `first` das erste Element oder `Nothing` zurückgeben soll. Die Methode `isEmpty` prüft, ob die Datenstruktur leer ist, indem überprüft wird, ob `first Nothing` zurück gibt. Die Methoden können dabei für verschiedene Datenstrukturen unterschiedlich implementiert, also überladen sein.

`Maybe` ist ein in Haskell vordefiniertes Konzept um das Fehlen eines Wertes darstellen zu können und ähnlich zu einem Typ mit den Konstruktoren `Nothing` sowie `Just a` implementiert.

Interessant an diesem Beispiel ist neben der ermöglichten Abstraktion, dass die Variable `s` hier eine Stelligkeit von 1 hat, wie an `s a` zu erkennen ist.

Mithilfe des folgenden Codes wollen wir nun den Typ `Set` der Typklasse `DataStruc` als Instanz zuweisen:

```haskell
instance DataStruc Set where
    add = (<<)
    remove = (-:)
    first (s :+ a) = Just a
    first Empty = Nothing
```

Hier ist hervorzuheben, dass keine Implementierung für `isEmpty` angegeben werden muss und in der Instanzendeklaration `Set` statt `Set a` als Typ angegeben wird.

Als mögliche Erweiterung dieser Typklasse könnten nun die Methoden `map` und `filter` eingeführt werden, wie wir sie bereits für Listen kennen - nun jedoch allgemein auch für Bäume, Mengen, Tupel etc.

4 Zusammenfassung

Wir haben gesehen, dass Typklassen und Operatoren vor allem im gegenseitigen Zusammenspiel mächtige Werkzeuge sind.

Operatoren bilden zwar nur eine andere Schreibweise für bereits vorhandene Sprachkonzepte, doch ihre Verwendung in Infixnotation und die Möglichkeit, Assoziierungsprioritäten und -richtungen festzulegen, ermöglichen es, viele Ausdrücke weitaus übersichtlicher und eleganter zu implementieren.

Mithilfe von Typklassen können wir Operatoren und Funktionen überladen, einfachste Vererbung realisieren und Standardimplementierungen für Methoden angeben. Dies spart nicht nur Aufwand, sondern schafft auch viele neue Möglichkeiten. So können wir beispielsweise mithilfe der Typklasse `Show` beliebige Datentypen in der Konsole ausgeben lassen oder durch die Verwendung von Kontexten sicherstellen, dass eine geschriebene Funktion keinen Fehler aufwirft, weil eine bestimmte Funktionalität von einem Datentyp nicht unterstützt wird.

Verwenden wir beide Konzepte zusammen, können wir beispielsweise selbst geschriebene und bekannte Operatoren wie `==` oder `+` für verschiedene Datentypen definieren.

5 EBNF Übersicht

Zur Übersicht über die vorgestellten Themen hier noch ein Ausschnitt aus der EBNF-Grammatik aus der Vorlesung *Funktionale Programmierung* [Gie16], um die Syntax für die Deklarationen von Operatoren und Typklassen zusammenzufassen:

$$
\begin{aligned}
\underline{topdecl} \rightarrow\ & \underline{decl} \\
| \ & data\ [\underline{context} \Rightarrow]\ \ldots \\
| \ & class\ [\underline{context} \Rightarrow]\ \underline{tyconstr}\ \underline{var} \\
& \quad [where\ \{\underline{decl}_1; \ldots; \underline{decl}_n\}], \text{ wobei } n \geq 1 \\
| \ & instance\ [\underline{context} \Rightarrow]\ \underline{tyconstr}\ \underline{instype} \\
& \quad [where\ \{\underline{idecl}_1\ \ldots; \underline{idecl}_n\}], \text{ wobei } n \geq 1 \\
\underline{tyconstr} \rightarrow\ & \text{String von Buchstaben und Zahlen mit Großbuchstaben am Anfang} \\
\underline{decl} \rightarrow\ & \underline{typdecl} \mid \underline{fundecl} \mid \underline{infixdecl} \mid \underline{var} = \underline{expr} \\
\underline{idecl} \rightarrow\ & \underline{fundecl} \mid \underline{var} = \underline{expr} \\
\underline{typdecl} \rightarrow\ & \underline{var}_1, \ldots, \underline{var}_n\ ::\ [\underline{context} \Rightarrow]\ \underline{type} \\
\underline{instype} \rightarrow\ & (\underline{tyconstr}\ \underline{var}_1 \ldots \underline{var}_n), \text{ wobei } n \geq 0 \\
| \ & [\underline{var}] \\
| \ & (\underline{var}_1 \rightarrow \underline{var}_2) \\
| \ & (\underline{var}_1, \ldots, \underline{var}_n), \text{ wobei } n \geq 2 \\
\underline{context} \rightarrow\ & (\underline{tyconstr}_1\ \underline{var}_1, \ldots, \underline{tyconstr}_n\ \underline{var}_n), \text{ wobei } n \geq 1 \\
\underline{infixdecl} \rightarrow\ & (\text{ infix} \mid \text{infixl} \mid \text{infixr }) [0|1| \ldots |9]\ \underline{op}_1, \ldots, \underline{op}_n \text{ wobei } n \geq 1 \\
\underline{op} \rightarrow\ & \underline{varop} \mid \underline{constrop} \\
\underline{varop} \rightarrow\ & \text{String von Sonderzeichen, der nicht mit : beginnt} \\
\underline{constrop} \rightarrow\ & \text{String von Sonderzeichen, der mit : beginnt}
\end{aligned}
$$

Literatur

[Fra] *Haskell Wiki Frageseite*. `https://wiki.haskell.org/Questions_and_answers`. – Zuletzt aufgerufen am 02.05.2019

[Gie16] Giesl, Jürgen: *Skript zur Vorlesung Funktionale Programmierung*. `https://verify.rwth-aachen.de/fp16/FP16.pdf`. Version: SS 2016. – Zuletzt aufgerufen am 28.04.2019

[HDI] *Specification of Derived Instances*. `https://www.haskell.org/onlinereport/derived.html`. – Zuletzt aufgerufen am 29.04.2019

[HPF99] Hudak, Paul ; Peterson, John ; Fasel, Joseph H.: *A Gentle Introduction to Haskell*. `https://www.haskell.org/tutorial/`. Version: Oct 1999

[Pre] *Standard types, classes and related functions*. `https://hackage.haskell.org/package/base-4.12.0.0/docs/Prelude.html`. – Zuletzt aufgerufen am 29.04.2019