TYPKLASSEN UND OPERATOREN IN HASKELL

YUYUAN LIU UND LUCA OELJEKLAUS

1. Einführung

In dieser Arbeit werden zwei Konzepte von Haskell vorgestellt, zum einen Operatoren, zum anderen Typklassen. Im ersten Abschnitt werden die Grundlagen von Haskells Operatoren eingeführt und anschließend die Probleme diskutiert, die bei der Verwendung dieser entstehen können.

Inhalt des zweiten Abschnitts ist die Vorstellung von Haskells parametrischer Polymorphie, welche die Grundlage für Haskells Typklassen bildet. Diese werden anschließend eingeführt und ihre Feinheiten an Beispielen erläutert. Abschließend wird die Kombination der vorgestellten Konzepte betrachtet.

2. Operatoren

Haskell stellt gängige Funktionen als *Operatoren* zur Verfügung. Diese werden in *Infix-Schreibweise* verwendet, d. h. sie werden zwischen ihre Argumente gesetzt. Operatoren werden mit Sonderzeichen[1] notiert und sind vom Typ $\underline{type}_1 \to \underline{type}_2 \to \underline{type}_3$.

Haskell unterscheidet zwischen Konstruktoroperatoren und Variablenoperatoren. Konstruktoren beginnen mit einem Doppelpunkt, wie zum Beispiel der Listenkonstruktor ":". Variablenoperatoren beginnen mit einem beliebigen anderen Sonderzeichen.

```
(+) :: Num a => a -> a -> a
(:) :: a -> [a] -> [a]
(==) :: Eq a => a -> a -> Bool
```

Abbildung 1. Signaturen von (+), (:) und (==)

Durch die Infix-Schreibweise und die Notation durch Sonderzeichen unterscheiden Operatoren sich von anderen Funktionen, die in *Präfix-Schreibweise* verwendet werden und aus Buchstaben und Zahlen bestehen. Durch Klammerung bzw. Backticks können Operatoren in Präfix- bzw. Funktionen[2] in Infix-Schreibweise verwendet werden. (s. Abb. 3).

[1] Jedes Unicode-Satzzeichen und Symbol außer : ( ) , ; [ ] ‘ { } _ ” ′

[2] *zweistellige* Funktionen, d.h. vom Typ $\underline{type}_1 \to \underline{type}_2 \to \underline{type}_3$

[3] Wir nehmen die Funktion ‘addInt‘ als Addition von Integers an

```
plus, (+) :: Int -> Int -> Int
plus x y = x `addInt` y
(+) x y = x `addInt` y
```

Abbildung 2. Die Addition zweier Integers als Funktion/Operator[3]

```
39 + 3
(+) 39 3
plus 39 3
39 `plus` 3
```

Abbildung 3. Äquivalente Ausdrücke der Addition

3. Assoziation und Bindungspriorität

Falls für den Typ eines Operators $\underline{type}_1 = \underline{type}_3$ oder $\underline{type}_2 = \underline{type}_3$ gilt, kann er wiederholt verwendet werden. Dabei kann es zu Problemen kommen, da die Bindungsreihenfolge Einfluss auf das Ergebnis haben kann.

```
(/) :: Int -> Int -> Int
(/) x y = x `divInt` y

> 12 / 6 / 2
?

> (/) ((/) 12 6) 2
1

> (/) 12 ((/) 6 2)
4
```

Abbildung 4. Assoziation von Operatoren

Haskell löst dieses Problem durch *Assoziationsregeln*. Einem Operator wird in seiner Definition **infixl**, **infixr** oder **infix** vorangestellt. Ein mit **infixl** deklarierter Operator wird von links nach rechts, ein mit **infixr** deklarierter von rechts nach links ausgewertet. Mit **infix** deklarierte Operatoren lassen sich nicht assoziieren. So werden Ausdrücke in eindeutige Präfix-Schreibweise konvertiert.

Die Verwendung verschiedener Operatoren in einem Ausdruck kann ebenfalls zu Problemen führen. Die Bindungsreihenfolge kann das Ergebnis verändern.

```
infixl 7 /

> 12 / 6 / 2
1
```

ABBILDUNG 5. Assoziation der Division

```
> 6 + 12 / 3
?

> (/) ((+) 6 12) 3
6

> (+) 6 ((/) 12 3)
10
```

ABBILDUNG 6. Bindungspriorität von Operatoren

Gelöst wird das Problem durch die *Bindungspriorität*[4] eines Operators. Die Priorität ist eine Zahl zwischen 0 und 9 und wird dem Operator als Attribut vorangestellt. Ein Operator mit Priorität 9 bindet am stärksten, einer mit Priorität 0 bindet am schwächsten. Stärker bindende Operatoren werden vor schwächer bindenden Operatoren ausgewertet.

```
infixl 6 +

> 6 + 12 / 3
10
```

ABBILDUNG 7. Priorität der Addition

In Infix-Schreibweise verwendete Funktionen werden als linksassoziierend mit Bindungspriorität 10 ausgewertet.

Für vordefinierte Operatoren sind Bindungspriorität und Assoziation vorgegeben, um z.B. Punktrechnung vor Strichrechnung zu erfüllen und Ausdrücke wie `4:5:[]`[5] auswerten zu können (s. TAB. 1), obwohl für ":" nicht $\underline{type}_1 = \underline{type}_2 = \underline{type}_3$ gilt.

In Haskell ist es möglich, benutzerdefinierte Operatoren einzuführen. Neue Operatoren haben standardmäßig die Bindungspriorität 9 und sind mit **infixl** linksassoziierend. Der Benutzer hat die Möglichkeit, diese Attribute zu modifizieren.

[4] oft auch nur einfach Priorität genannt

[5] `4:5:[]` ⇒ `4:[5]` ⇒ `[4,5]`

Prio.	l.assoziierend	nicht-assoziierend	r.assoziierend
9	!!		.
8			^, ^^, **
7	*, /, ‘div‘, ‘mod‘ ‘rem‘, ‘quot‘		
6	+, −		
5			:, ++
4		==, / =, <, <=, > >=, ’elem’, ’notElem’	
3			&&
2			||
1	>>, >>=		
0			$, $!, ’seq’

TABELLE 1. Die Prioritäten der vordefinierten Operatoren[3]

Operatorbezeichner bestehen aus Sonderzeichen, daher geben sie oft wenig Aufschluss bzgl. der Arbeitsweise ihrer Funktionen. In Paketen die viele Operatoren verwenden ist das problematisch, da sie leicht zu verwechseln sind und Fehlersuchen erschweren.

```
(</>~) :: ASetter s t FilePath FilePath
    -> FilePath -> s -> t

(<</>~) :: LensLike ((,) FilePath) s a
    FilePath FilePath -> FilePath -> s
        -> (FilePath, a)

(<</>=) :: MonadState s m => LensLike'
    ((,) FilePath) s FilePath -> FilePath
        -> m FilePath
```

ABBILDUNG 8. Auswahl der im Lens-Paket definierten Operatoren

4. PARAMETRISCHE POLYMORPHIE

Parametrische Polymorphie wird in Haskell verwendet, um einzelne Funktionen für alle Typen zur Verfügung zu stellen. Statt für jeden Typen dieselbe Funktion zu implementieren, ersetzt man in der Deklaration Typen durch Typvariablen. Falls die Deklaration einer Funktion eine Typvariable enthält kann die Funktion an dieser Stelle jeden beliebigen Typ entgegennehmen.

```
id :: a -> a
id a = a

> id 42
42

> id "HelloWorld"
"HelloWorld"
```

ABBILDUNG 9. Die mathematische Identitätsfunktion mit Typvariablen

5. TYPKLASSEN

5.1. **Entstehung.** Die parametrische Polymorphie ist nur für sehr allgemeine Funktionen geeignet, also überall dort, wo der Typ eines Elements nicht von Relevanz ist. Viele Funktionen können zwar mehrere Typen entgegennehmen, aber nicht alle. Wir sprechen von quasi-polymorphen Funktionen.

```
sort :: [a] -> [a]
show :: a -> String
square :: a -> a
```

ABBILDUNG 10. Quasi-polymorphe Funktionen

Quasi-polymorphe Funktionen können nicht beliebige Typen entgegennehmen, sondern nur solche, die gewisse Bedingungen erfüllen. Zum Beispiel kann eine Liste sortiert werden, solange ihre Elemente verglichen[6] werden können. Jeder Typ kann ausgegeben werden, solange er in einen String konvertiert werden kann. Jeder Typ kann quadriert werden, solange für ihn die Multiplikation definiert ist.

In Standard ML (SML) gab es ein ähnliches Problem. Der Gleichheitsoperator war polymorph[5]. Er durfte aber nicht für Typen definiert sein, die nicht auf Gleichheit prüfbar waren, z. B. Funktionen. Die Lösung war eine Erweiterung des Typsystems um "*equality types*", also wortwörtlich "Gleichheitstypen". Die Gleichheit wurde für alle Typen instanziiert, für die sie sinnvoll und nachprüfbar war. Das verwendete Symbol = blieb für alle Typen gleich.

Falls die Gleichheit mit einem bestimmten Typ verwendet wurde, konnte per Typinferenz entschieden werden, welche Instanz zu verwenden war. In SML konnte die Gleichheit allerdings nicht für neue Typen definiert werden.

[6]Wir sagen dass zwei Elemente vergleichbar sind, falls sich feststellen lässt, ob das eine größer als das andere ist

5.2. **Funktionsweise von Typklassen.** Das Konzept der Gleichheitstypen aus SML wurde erweitert auf alle Funktionen, die für mehrere Typen existieren sollen. Man spricht von *Funktionsüberladung*.

```
(+~) :: Int -> Int -> Int
(+') :: Float -> Float -> Float
(+*) :: Double -> Double -> Double
```

ABBILDUNG 11. *Mögliche* Signaturen der einfachen Addition ohne Funktionsüberladung

Eine *Typklasse*[7] ist eine Menge von Funktionssignaturen, die polymorph deklariert werden. Man kann sie sich ähnlich wie Interfaces in objektorientierten Sprachen vorstellen.

In Haskell ist die Klasse `Eq` vordefiniert. Sie stellt zwei Funktionen bzw. Operatoren zur Verfügung, den Gleichheitsoperator und den Ungleichheitsoperator.

```
class Eq a where
    (==),(/=) :: a -> a -> Bool
    (/=) x y = not (x == y)
    (==) x y = not (x /= y)
```

ABBILDUNG 12. Signatur der Klasse Eq

In Typklassen enthaltene Funktionen werden meistens nicht sofort implementiert. Falls sie sich durch andere Funktionen beschreiben lassen, ist dies aber möglich[8].

Eine Typklasse enthält anfangs keine Typen. Diese müssen zunächst *Instanzen* der Typklasse werden. Ein Typ ist Instanz einer Klasse, wenn alle ihre Funktionen für ihn implementiert sind. In der Klasse `Eq` sind `(==)` und `(/=)` als Negation von `(/=)` bzw. `(==)` implementiert. Damit ein Typ für `Eq` instanziiert wird reicht es also, einen der beiden Operatoren zu implementieren.

Benutzerdefinierte Typen können auch als Mitglieder von Typklassen instanziiert werden.

Es kann mehrere Möglichkeiten geben, um einen Typ als Instanz einer Klasse zu deklarieren. In ABB. 14 wird `Farbe` einmal über `(==)` und einmal über `(/=)` als Mitglied von `Eq` instanziiert.

Typklassen erlauben es, Funktionen für verschiedene Typen unterschiedlich zu definieren. Alle Instanzen einer Klasse besitzen *sinnvolle* Implementierungen der deklarierten Funktionen. Dadurch haben

[7]Im Weiteren bezeichnen wir Typklassen auch einfach als Klassen

[8](s. ABB. 12), die Ungleichheit wird als Negation der Gleichheit definiert, bzw. umgekehrt

```
instance Eq Bool where
    (==) x y = if x then y else not y

instance Eq Int where
    (==) x y = x `eqInt` y
```

ABBILDUNG 13. **Bool** und **Int** sind Instanzen von Eq

```
--neuer Datentyp, Farbe, mit Werten
--Rot, Gruen und Blau
    data Farbe = Rot | Gruen | Blau

instance Eq Farbe where
        (==) Rot   Rot   = True
        (==) Gruen Gruen = True
        (==) Blau  Blau  = True
        (==) _     _     = False

-- Alternative

instance Eq Farbe where
    (/=) Rot   Gruen        = True
    (/=) Rot   Blau         = True
    (/=) Gruen Rot          = True
    (/=) Gruen Blau         = True
    (/=) Blau  Rot          = True
    (/=) Blau  Gruen        = True
    (/=) _     _            = False
```

ABBILDUNG 14. Selbst definierter Datentyp als Instanz von Eq

verschiedenen Instanzen einer Typklasse gemeinsame Eigenschaften. Die in Eq enthaltenen Typen besitzen die Eigenschaft, auf Gleichheit prüfbar zu sein.

```
elemOfList :: Eq a => a -> [a] -> Bool
elemOfList x [] = False
elemOfList x y:ys = (x == y) || (elemOfList x ys)
```

ABBILDUNG 15. Suchfunktion für Listen mit einem auf Gleichheit prüfbaren Typ

Funktionen können Bedingungen an Typvariablen stellen. Wenn in einer Funktion zwei Variablen auf `(==)` geprüft werden, muss der geprüfte Typ Instanz von `Eq` sein. Diese Bedingungen werden in der Signatur gestellt. "`Eq a =>`"[9] fordert z. B. dass `a` eine Instanz von `Eq` ist. Eine Funktion kann nur mit Typen aufgerufen werden, die diese Bedingung erfüllen.

```
sort :: Ord a => [a] -> [a]
show :: Show a => a -> String
square :: Num a => a -> a
```

Abbildung 16. Vollständige Signaturen für `sort`, `show` und `square`

6. Klassenhierarchie

Ein Typ, der Instanz einer Klasse ist, besitzt die entsprechenden Eigenschaften. Einige Eigenschaften können Voraussetzungen für andere Eigenschaften sein. Dass zwei Elemente auf Gleichheit geprüft werden können, ist eine Voraussetzung um zu bestimmen, ob das eine größer oder gleich dem anderen ist.

```
class (Eq a) => Ord a where
    (<), (<=), (>), (>=) :: a -> a -> Bool
    (<) x y = not ((>=) x y)
    (<=) x y = x == y || x < y
    (>) x y = not ((<=) x y)
    (>=) x y = x == y || x > y
```

Abbildung 17. Die Typklasse Ord

"`(Eq a) => Ord a`"[10] bedeutet, dass `a` Instanz von `Eq` sein muss, bevor es Instanz von `Ord` wird.

Es bedeutet nicht, dass jeder Typ der Klasse `Eq` zur Instanz von `Ord` werden muss. Der Typ `Farbe` wird für die Klasse `Ord` nicht instanziiert. Dadurch ist sie nicht sortierbar.

Das System aus Typklassen wird als *Klassenhierarchie* bezeichnet. Sie kann um benutzerdefinierte Klassen erweitert werden. Neue Typklassen müssen nicht von existierenden Klassen abhängen.

In Abb. 19 wird die benutzerdefinierte Klasse `MonoidKlasse` deklariert. Sie modelliert Monoide in Haskell. Um eine Instanz über

[9] `Eq a =>` wird gelesen als *für alle Typen a mit a ist Instanz von* Eq

[10] `(Eq a) => Ord a` wird gelesen als *"Vorrausgesetzt, dass* `a` *eine Instanz von* `Eq` *ist, kann* `a` *Instanz von* `Ord` *werden"*

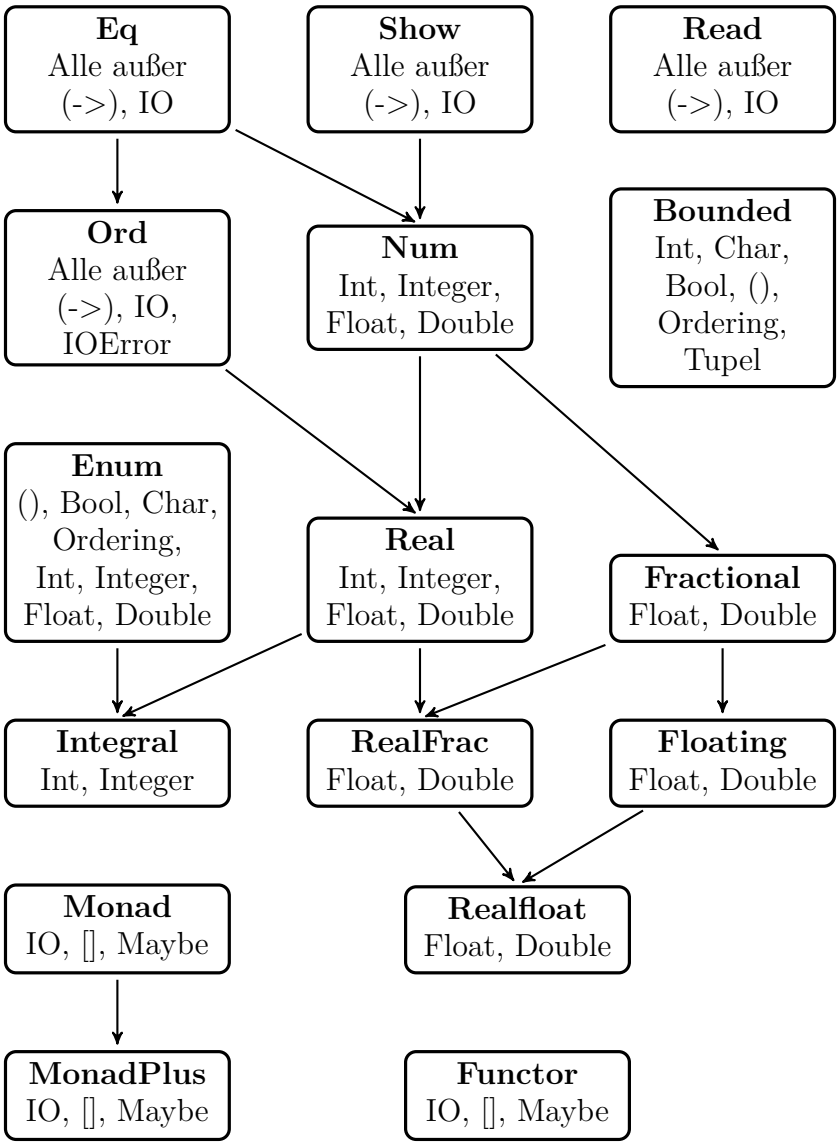

ABBILDUNG 18. Die integrierte Klassenhierarchie in Haskell[4]

einem Typ zu erstellen, muss eine Operation und ein neutrales Element bezüglich dieser Operation definiert werden.

Der Operator `(<|>)` wird während der Instanziierungen von **`Int`** und **`Bool`** einmal als Addition und einmal als logische Verundung definiert. Man spricht von *Operatorüberladung*. Alternativ ist es möglich, $(\mathbb{Z}, *, 1)$ und $(\mathbb{B}, \vee, False)$ zu instanziieren[11].

[11]Um sowohl den additiven als auch den multiplikativen Monoiden zu erhalten, lassen sich über `newtype` die Typen `Add Int` und `Mult Int` definieren, die dann instanziiert werden. Ähnlich für die boolschen Monoide.

```
class MonoidKlasse a where
    (<|>)     :: a -> a -> a
    e         :: a

instance MonoidKlasse Int where
    e = 0
    x <|> y = x + y

instance MonoidKlasse Bool where
    e = True
    x <|> y = x && y
```

Abbildung 19. Typklasse der Monoide, Instanziierungen von $(\mathbb{Z}, +, 0)$ und $(\mathbb{B}, \wedge, True)$

7. Fazit

In dieser Arbeit wurden Haskells Operatoren eingeführt. Nach einer Erklärung ihrer Funktionsweise wurden die Begriffe der Assoziation und der Bindungspriorität eingeführt und erklärt, wie diese für benutzerdefinierte Operatoren bearbeitet werden können.

Danach haben wir gesehen, dass das Konzept der Gleichheitstypen aus SML die Grundlage für Haskells Typklassen bildet. Es wurde verallgemeinert, um Überladung für beliebige Funktionen zuzulassen und auch Instanziierungen von benutzerdefinierten Typen zu ermöglichen.

Zuletzt wurden beide Konzepte kombiniert und Operatorüberladung eingeführt.

Literatur

[1] R. Bird, *Thinking Functionally with Haskell*, Cambridge: Cambridge University Press, 2015
[2] E. Kmett
https://hackage.haskell.org/package/lens
[3] S. Marlow
https://www.haskell.org/onlinereport/haskell2010/haskellch4.html#x10-820061
[4] S. Peyton Jones
https://www.haskell.org/onlinereport/basic.html#standard-classes
[5] P. Wadler
http://homepages.inf.ed.ac.uk/wadler/papers/class-letter/class-letter.txt