Typklassen: Haskell

Mark Kaminski
Betreuer: Andreas Rossberg

Proseminar Programmiersysteme WS 2003/04
Lehrstuhl Programmiersysteme Prof. Gert Smolka
Universität des Saarlandes

5. April 2004

Zusammenfassung

Typklassen erweitern das polymorphe Hindley/Milner-Typsystem um die Möglichkeit, neben parametrisch polymorphen auch überladene Operationen zu definieren. Der Definitionsbereich solcher Operationen kann dabei den an sie gestellten Anforderungen angepasst werden. Im Folgenden soll einerseits die Motivation und die Grundideen hinter Typklassen, andererseits ein Einblick in Implementierungstechniken des Typklassensystems vermittelt werden. Anschließend werden Konstruktorklassen sowie Multi-Parameter-Typklassen als Beispiele für Erweiterungen des Typklassensystems vorgestellt. Zusammenfassend werden Typklassen mit Konzepten aus der objektorientierten Programmierung verglichen.

1 Einleitung

Akzeptiert eine Operation, etwa eine Prozedur oder ein Operator, für einen ihrer Argumente Werte unterschiedlicher Typen, so nennt man diese Operation polymorph. In gegenwärtig verbreiteten Programmiersprachen lassen sich zwei Arten von Polymorphismus unterscheiden.
Beim parametrischen Polymorphismus ist eine Prozedur unabhängig vom Typ ihrer Argumente definiert. Der Argumenttyp kann daher frei variieren ohne das Verhalten der Prozedur zu beeinflussen. Typischerweise werden parametrisch polymorphe Prozeduren auf einen Behältertyp, etwa eine Liste, angewendet und operieren auf der Behälterstruktur. Da prinzipiell jeder Typ als Elementtyp des Behälters in Frage kommt, bleibt er für die Prozedur unbekannt. Ohne diese Information kann die Prozedur nur indirekt durch prozedurale Argumente auf die Elemente des Behälters zugreifen. Ein weit verbreiteter Ansatz zur Realisierung von parametrischem Polymorphismus in Programmiersprachen ist das Hindley/Milner-Typsystem (siehe [1]). Prominente Vertreter sind Dialekte von ML sowie Miranda. Zu den Vorzügen des Systems zählt die Möglichkeit automatischer Typinferenz.
Beim ad-hoc-Polymorphismus, auch bekannt als Überladen, kennt eine Prozedur den genauen Typ ihrer Argumente, hat somit direkten Zugriff auf die Argumentwerte und bestimmt ihr Verhalten in Abhängigkeit von diesem Wissen. Diese Möglichkeit ist wichtig, wenn sich das Verhalten einer Operation je nach Argumenttyp grundlegend

unterscheiden muss. So muss beispielsweise für die Multiplikation von Gleitkommazahlen ein anderer Algorithmus verwendet werden als für ganze Zahlen. Noch deutlicher ist der Unterschied bei dem <<-Operator in C++, der in Abhängigkeit von seinen Argumenttypen als binärer Schiftoperator oder als Ausgabeoperator fungiert. Das Verhalten einer überladenen Operation ist in der Regel nur auf einer bestimmten Teilmenge aller Typen definiert, die entweder explizit durch entsprechende Deklarationen oder implizit durch Vererbung erweitert werden kann. Im Gegensatz zum parametrischen Polymorphismus existieren für das Überladen unterschiedliche Ansätze in verschiedenen Programmiersprachen.
Ein typisches Einsatzgebiet für das Überladen ist die Realisierung von arithmetischen Operatoren sowie dem Gleichheitsoperator. Deswegen wollen wir anhand dieser Beispiele einige Probleme bei der Verbindung des Hindley/Milner-Typsystems mit dem Überladen aufzeigen.
Unser erstes Beispiel ist die Implementierung von Arithmetik in Standard ML. Hierbei sind Operatoren zwar überladen, Prozeduren, die diese verwenden, sind aber monomorph getypt. So führt die Deklaration

```
fun square x = x*x
```

die Typbindung

```
square : int -> int
```

ein, obwohl der Multiplikationsoperator auch auf Gleitkommazahlen anwendbar ist. Da man *square* natürlich auch auf Gleitkommazahlen anwenden möchte, ist eine Implementierung denkbar, die zu einer Deklaration sämtliche überladenen Versionen einer Prozedur einführt. Man sieht jedoch schnell, dass die Anzahl solcher Versionen exponentiell mit der Anzahl der Prozedurargumente wachsen kann, was den Einsatz der Alternativlösung problematisch macht.
Schauen wir uns nun an, wie Gleichheit in Sprachen mit dem Hindley/Milner-Typsystem realisiert wird. In der ersten Version von Standard ML ist der Gleichheitsoperator wie die arithmetischen Operationen überladen, mit der Folge, dass eine Deklaration wie

```
fun member _ nil     = false
  | member x (y::yr) = x=y orelse member x yr
```

genau wie im Falle der Arithmetik eine monomorphe Prozedur einführt. Dies ist eine sehr starke Einschränkung, denn während es in ML nur zwei arithmetische Typen gibt, gibt es unendlich viele Typen, die den Gleichheitstest erlauben, durch eine monomorphe Implementierung von *member* aber nicht verglichen werden können.
In Caml oder Miranda ist der Gleichheitsoperator daher vollständig polymorph. Das Typsystem erlaubt den Gleichheitstest auf allen Typen. Dies hat zwei negative Folgen. Zum einen kann der Gleichheitstest auf Prozeduren so erst zur Laufzeit abgefangen werden, zum anderen wird beim Vergleich von abstrakten Datentypen nur die Darstellung des Datentyps auf Gleichheit überprüft. Falls ein Wert auf mehrere Arten dargestellt werden kann, berechnet der Gleichheitsoperator somit nicht mehr den Gleichheitstest.
Aktuelle Implementierungen von Standard ML schränken die Anwendbarkeit des Gleichheitsoperators auf eine Teilmenge von Typen ein, die man Typen mit Gleichheit (equality types oder kurz eqtypes) nennt. Auf diesen Typen ist der Operator

polymorph. Die Lösung macht es möglich, die oben beschriebenen ungültigen Gleichheitstests auf Typebene abzufangen. Es bleiben dennoch einige Einschränkungen. Will man abstrakte Datentypen auf Gleichheit testen, so kann man den Gleichheitsoperator nur dann anwenden, wenn Gleichheit mit Darstellungsgleichheit zusammenfällt. Ansonsten muss man den abstrakten Datentyp aus der Menge der eqtypes ausschließen. Hat man einen komplexen Datentyp, der vergleichbare und nicht vergleichbare Komponenten hat, wie etwa

```
datatype a b ComparableProc of unit ref * (a -> b)
```

so kann man den Gleichheitstest auch dann nicht anwenden, wenn Gleichheit allein durch den Test vergleichbarer Komponenten definiert sein soll.

2 Grundideen

Beim Enwurf von Haskell empfand man keinen der oben genannten Ansätze als zufriedenstellend. Dies gab den Anstoss zur Entwicklung einer neuen Lösung, dem System der Typklassen. Typklassen lehnen sich an das Konzept von eqtypes an und erweitern es um die Möglichkeit, neue überladene Operationen auf eingeschränkten Mengen von Typen einzuführen sowie diese Mengen nach Bedarf zu vergrößern. Sie verallgemeinern das Hindley/Milner-System so, dass auf den überladenen Bezeichnern Typinferenz weiterhin möglich ist, und vereinigen damit den parametrischen Polymorphismus mit dem ad-hoc-Polymorphismus in einem einheitlichen Typsystem.
Die Grundidee hinter Typklassen ist die Einsicht, dass bestimmte Operationen nur auf Argumente bestimmter, zu der Semantik der jeweiligen Operation passender Typen anwendbar sind. Eine Menge von Typen kann also durch die auf sie anwendbaren Operationen charakterisiert werden. Gehört ein Typ zu dieser Menge, so sind alle für die Menge charakteristischen Operationen auch auf dem Typ definiert. Will man die Menge um einen neuen Typ erweitern, muss man die Definition aller charakteristischen Operationen der Menge um eine adäquate Behandlung des neuen Typs ergänzen. Die eben beschriebenen Mengen von Typen nennt man Typklassen. Die für eine Typklasse charakteristischen Operationen werden als Methoden der Typklasse bezeichnet. Die Typen, aus denen sich eine Typklasse zusammensetzt, heißen Instanzen der Klasse.
Es ist klar, dass der Typ einer polymorphen Operation Information über geeignete Argument- und Ergebnistypen beinhalten sollte, ebenso wie der Typ einer polymorphen Variablen Aufschluss über erlaubte Werttypen geben sollte. Diese Information bildet den so genannten Kontext eines Typs. Typen mit einem nichtleeren Kontext nennt man qualifiziert. Der Kontext ordnet eine freie Typvariable einer oder mehreren Typklassen zu. Eine Typvariable kann nur durch solche Typen instanziiert werden, welche zur Schnittmenge aller der Variablen durch den Kontext zugeordneten Typklassen gehören. Betrachten wir als Beispiel den Typ des Gleichheitsoperators in Haskell:

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

Der Kontext *Eq a* schränkt den Definitionsbereich des Gleichheitsoperators auf Instanzen der Typklasse *Eq* ein. Wie erwartet, beschreibt die Klasse *Eq* gerade Typen mit Gleichheit. Zum Vergleich der entsprechende Typ in Standard ML:

```
op= : ''a * ''a -> bool
```

Wir sehen, dass wir mit Hilfe von Kontexten die Funktion von eqtype-Variablen in Standard ML leicht nachbilden können. Der Operator für ganzzahlige Potenzierung stellt stärkere Forderungen an seine Argumente:

```
(^) :: (Num a, Integral b) => a -> b -> a
```

Das erste Argument, die Basis, muss zur Typklasse *Num* der Zahlen gehören. Das zweite Argument, der Exponent, muss eine Instanz der Klasse *Integral* der ganzen Zahlen sein. Die Potenz ist vom selben Typ wie die Basis. Da wir auch ganze Zahlen potenzieren wollen, muss die Klasse *Integral* offensichtlich eine Teilmenge der Klasse *Num* beschreiben. Um dafür Sorge zu tragen wäre es sinnvoll, Typklassen als Teilmengen anderer Klassen definieren zu können. Diese Möglichkeit lernen wir in Abschnitt 6 kennen.

3 Beispiel: Gleichheit

Zunächst wollen wir uns aber anschauen, wie man neue, anfangs leere Typklassen einführt und die bestehenden Klassen um neue Instanzen erweitert. Zu diesem Zweck betrachten wir, wie die Klasse *Eq* von Typen mit Gleichheit deklariert ist.

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

  x == y     = not (x /= y)
  x /= y     = not (x == y)
```

Die Deklaration lässt sich in zwei Teile mit unterschiedlichen Funktionen gliedern. Der erste Teil besagt, dass ein Typ a genau dann zur Klasse *Eq* gehört, wenn auf ihm der Gleichheits- und der Ungleichheitsoperator definiert sind, wobei die beiden Operatoren wie folgt getypt sind:

```
(==) :: Eq a => a -> a -> Bool
(/=) :: Eq a => a -> a -> Bool
```

Der Kontext *Eq a* taucht bei der Deklaration der Methoden nicht explizit auf, wird aber durch die Bindung der Typvariable a im Kopf der Klassendeklaration inferiert. Eine Variable, die im Kopf einer Typklassendeklaration auftaucht, heißt Parameter der Typklasse. Der aktuelle Haskell-Standard (Haskell 98, siehe [2]) erlaubt Typklassen mit genau einem Parameter.
Da sowohl der Gleichheits- als auch der Ungleichheitsoperator Methoden der Klasse *Eq* sind, muss jede Instanz für beide Operatoren Implementierungen angeben. Dies erscheint nicht besonders sinnvoll, da wir Ungleichheit in der Regel einfach als logische Umkehrung von Gleichheit implementieren. Daher gibt es die Möglichkeit, bei der Klassendeklaration für Methoden Default-Implementierungen anzugeben. So braucht eine Instanz nicht alle Methoden einer Klasse zu implementieren. Gibt sie für eine Methode mit einer Default-Implementierung keine eigene Implementierung an, wird die Default-Implementierung verwendet. Nun können wir auch die Funktion des zweiten Teils der Deklaration von *Eq* erklären. Es ist eine zirkuläre Deklaration der Default-Implementierungen für Gleichheit und Ungleichheit. Eine Instanz von *Eq* muss also immer nur einen Operator implementieren, der fehlende Operator wird

durch die logische Umkehrung des gegebenen ergänzt. Natürlich muss man beachten, dass wenn eine Instanz überhaupt keine Implementierung angibt, der Gleichheitstest auf dieser Instanz wegen der zirkulären Deklaration divergiert.
Wie kann man nun Typen zu Instanzen einer Typklasse machen? Ein Paar Beispiele helfen diese Frage zu klären:

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

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

Wir sehen, wie die Typen Int und $Bool$ durch so genannte Instanzdeklarationen zur Klasse Eq hinzugefügt werden können. In beiden Fällen braucht man nur den Gleichheitsoperator zu definieren, die Default-Implementierung für `/=` ergänzt die Deklaration. Für dieses Beispiel nehmen wir an, dass $eqInt$ eine primitive Vergleichsprozedur ist, die den Typ $Int \to Int \to Bool$ hat.
Natürlich können wir auch Listen zur Klasse Eq hinzufügen:

```
instance Eq a => Eq [a] where
  []   == []       = True
  x:xs == y:ys     = x==y && xs==ys
  _    == _        = False
```

Man sieht, dass Kontexte auch im Kopf einer Instanzdeklaration auftauchen können. In unserem Fall ist der Kontext $Eq\ a$ verständlich, da wir auf einem Listentyp nur dann Gleichheit definieren wollen, wenn auf dem Elementtyp Gleichheit definiert ist. Die Implementierung des Gleichheitstests nutzt das Wissen um die Zugehörigkeit des Elementtyps zur Klasse Eq, als sie mit dem Ausdruck `x==y` die Listenköpfe auf Gleichheit überprüft. Anschließend macht sie mit `xs==ys` einen rekursiven Aufruf um die Listenrümpfe zu testen.

4 Überführung in das Hindley/Milner-System

Eine wichtige Eigenschaft des Typklassensystems ist die Möglichkeit, Programme, die Typklassen einsetzen, zur Übersetzungszeit in Programme zu überführen, die durch das Hindley/Milner-Typsystem vollständig beschrieben werden können. So lassen sich bestehende Programmiersprachen mit Hindley/Milner-Typen leichter um die Unterstützung von Typklassen erweitern. Wir wollen diese Überführungstechnik näher betrachten, weil sie einige Einblicke in Methoden liefert, mit denen das Überladen von Haskell-Übersetzern tatsächlich realisiert wird. Eine Schlüsselrolle bei der Überführung spielt die Verwendung so genannter Wörterbücher. Zu jeder Instanz einer Typklasse gehört ein Wörterbuch, das die Implementierung der Methoden für diese Instanz enthält. Prozeduren, die auf einer oder mehreren Typklassen überladen sind, erwarten zusammen mit ihren Argumenten ein oder mehrere den Argumenttypen entsprechende Wörterbücher und extrahieren aus diesen die passenden Operationen. Sehen wir uns nun an, wie die Deklaration der Klasse Eq in das Hindley/Milner-System überführt werden kann:

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

```
eq, neq           :: EqDict a -> a -> a -> Bool
eq  (EqDict e _)  = e
neq (EqDict _ ne) = ne

eqDefault, neqDefault  :: EqDict a -> a -> a -> Bool
eqDefault  eqDictA x y = not(neq eqDictA x y)
neqDefault eqDictA x y = not(eq  eqDictA x y)
```

Bei der Deklaration von *Eq* führen wir einen entsprechenden Wörterbuchtyp *EqDict* ein. Ein Wörterbuch vom Typ *EqDict* enthält zwei Felder, in denen Implementierungen für den Gleichheitstest und den Ungleichheitstest gespeichert werden. Beide haben den Typ $a \to a \to Bool$. Ferner werden zwei Prozeduren *eq* und *neq* deklariert, die zu einem Wörterbuch vom Typ *EqDict* die darin gespeicherten Methoden für Gleichheit und Ungleichheit liefern. Diese Prozeduren ersetzen die in *Eq* eingeführten Operatoren. Die Default-Implementierungen nehmen ein zusätzliches Argument *eqDictA*. Dabei handelt es sich um das Wörterbuch für den Typ, auf den die Default-Implementierungen angewandt werden sollen. Die ursprünglichen Operatoranwendungen werden wie angekündigt durch Anwendungen von *eq* und *neq* ersetzt, wobei beide Prozeduren noch das Wörterbuch *eqDictA* übergeben bekommen. Die Instanzdeklarationen für *Int* und *Float* werden dann wie folgt übersetzt:

```
eqDictInt   :: EqDict Int
eqDictInt   = EqDict eqInt (neqDefault eqDictInt)

eqDictFloat :: EqDict Float
eqDictFloat = EqDict eqFloat (neqDefault eqDictFloat)
```

Zu jeder Instanzdeklaration wird ein entsprechendes Wörterbuch erstellt. Die Default-Implementierungen bekommen dabei das sich erstellende Wörterbuch als Argument und liefern bei der ersten Auswertung ihrer Anwendung eine Vergleichsprozedur vom nötigen Typ $a \to a \to Bool$. Diese zirkuläre Abhängigkeit stellt das korrekte Funktionieren der Operationen sicher solange nicht beide Methoden des Wörterbuchs Default-Implementierungen sind. Im Falle von Listen ist die Überführung etwas komplexer:

```
eqDictList         :: EqDict a -> EqDict [a]
eqDictList eqDictA = EqDict (eqList eqDictA)
                            (neqDefault (eqDictList eqDictA))

eqList             :: EqDict a -> [a] -> [a] -> Bool
eqList _       []     []     = True
eqList eqDictA (x:xs) (y:ys) = eq eqDictA x y &&
                               eq (eqDictList eqDictA) xs ys
eqList _       _      _      = False
```

Der wesentliche Unterschied zu vorherigen Instanzdeklarationen besteht darin, dass der Listentyp selbst einen Parametertyp hat. Zu einem Elementtyp mit Gleichheit wird daher das passende Wörterbuch für Listen über diesen Typ generiert, indem die Prozedur *eqDictList* auf das Wörterbuch für den Elementtyp angewendet wird. Auch der Gleichheitstest *eqList* benötigt das Wörterbuch für den Elementtyp als einen zusätzlichen Parameter. Die Anwendung des überladenen Gleichheitsoperators wird durch die Anwendung von *eq* ersetzt, wobei nun das übergebene Wörterbuch die auszuführende Methode bestimmt. Die erste Anwendung vergleicht Werte

des Elementtyps. Die zweite konstruiert zunächst ein Wörterbuch für Listen über dem Elementtyp, was auf einen rekursiven Aufruf hinausläuft. Zusammen mit den Klassen- und Instanzdeklarationen werden alle überladenen Operatoranwendungen zu Anwendungen von *eq* und *neq* überführt. Das zu übergebende Wörterbuch wird dabei während der Typinferenz anhand ermittelter Argumenttypen bestimmt:

```
   1 == 2     →  eq eqDictInt 1 2
 1.0 /= 2.0   →  neq eqDictFloat 1.0 2.0
[1,2] == []   →  eq (eqDictList eqDictInt) [1,2] []
```

In [3] findet man neben weiteren Beispielen auch eine formale Darstellung der Überführungsregeln.

5 Ergebnisse: Arithmetik

Die praktischen Vorteile, die das Typklassensystem mit sich bringt, sollen nun am Beispiel der schon betrachteten Prozedur *square* aufgezeigt werden. Dazu definieren wir *square* noch einmal, diesmal in Haskell:

```
square x = x*x
```

Der Multiplikationsoperator ist auf der Klasse Num der Zahlen definiert, hat also den Typ $Num\ a \Rightarrow a \rightarrow a \rightarrow a$. Für *square* folgt die Typbindung

```
square :: Num a => a -> a
```

square ist auf allen Instanzen der Klasse Num d. h. auf allen Zahlen definiert. Dies bedeutet, dass wir *square* nur einmal definieren müssen, wonach es auf alle Zahlendarstellungen anwendbar ist. Bei jeder Anwendung wird die für die jeweilige Darstellung passende Multiplikationsoperation verwendet:

```
square 5    ⇝ 25 :: Int
square 3.14 ⇝ 9.8596 :: Float
```

Auf der anderen Seite wird jeder Versuch, *square* auf einen Typ anzuwenden, der die Multiplikation nicht unterstützt, als Typfehler erkannt. Führen wir eine neue Zahlendarstellung ein, so können wir diese durch eine entsprechende Instanzdeklaration zur Typklasse Num hinzufügen und *square* auch auf unsere eigene Zahlendarstellung anwenden. Wir können die Wiederverwendbarkeit unserer Programme deutlich verbessern ohne die Kürze und die Lesbarkeit des Codes durch übertriebene Anwendung höherstufiger Prozeduren zu gefährden.

6 Ableitung von Typklassen

Wie wir in Abschnitt 2 am Beispiel ganzzahliger Potenzierung gesehen haben, kann eine Menge von Typen Teilmenge einer anderen Menge von Typen sein. Ein weiteres Beispiel dafür ist das Verhältnis zwischen Typen mit Gleichheit und angeordneten Typen, d. h. Typen, auf denen Vergleichsoperatoren wie `>` oder `<=` definiert sind. Es ist nicht sinnvoll, angeordnete Typen zu haben, auf denen der Gleichheitstest nicht definiert ist. Daher gibt es im Typklassensystem die Möglichkeit, Teilmengenverhältnisse zwischen Klassen auszudrücken. In Analogie zur objektorientierten

Programmierung können können wir diese Möglichkeit als Ableitung von Typklassen bezeichnen. Teilmengen einer Basisklasse werden dabei zu abgeleiteten Typklassen zusammengefasst. Als Teilmengen erweitern sie die Eigenschaften ihrer Basisklasse um neue, für die jeweilige Teilmenge spezifische Operationen. Alle Operationen, die auf der Basisklasse definiert sind, sind auch auf allen abgeleiteten Klassen definiert. Es ist leicht zu sehen, dass diese „Vererbung“ lediglich eine andere Sichtweise auf die ursprüngliche Forderung ist, dass jede Instanz einer abgeleiteten Klasse zugleich Instanz der Basisklasse sein muss. Um die Konzepte zu illustrieren, leiten wir von der Klasse *Eq* die Klasse *Ord* der angeordneten Typen ab:

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

  -- Default-Implementierungen...
```

Alles, was wir zur Darstellung einer Teilmengenrelation brauchen, ist ein Kontext *Eq a* im Kopf der Klassendeklaration. Er bedeutet nichts anderes, als dass ein Typ *a* nur dann zur Typklasse *Ord* gehören kann, wenn er gleichzeitig eine Instanz von *Eq* ist.
Obwohl wir bisher immer von einer Basisklasse ausgegangen sind, kann eine abgeleitete Klasse mehrere Basisklassen haben. Ein Beispiel hierfür ist die Klasse *Real* der reellen Zahlen, die wie folgt deklariert ist:

```
class (Num a, Ord a) => Real a where
  ...
```

Aus der Mengensicht bedeutet die Ableitung einer Typklasse von mehreren Basisklassen einfach die Auszeichnung einer bestimmten Teilmenge im Schnitt der Basisklassen.
Explizite Darstellung von Teilmengenverhältnissen zwischen Typklassen erlaubt eine kürzere Darstellung von Kontexten bei qualifizierten Typen. Schränkt der Kontext die möglichen Belegungen einer Typvariablen auf Instanzen einer abgeleiteten Klasse ein, so können auf den entsprechenden Wert auch alle Basisklassenoperationen angewendet werden ohne den Kontext explizit erweitern zu müssen. Zur Verdeutlichung genügt eine kurze Implementierung der binären Suche in Binärbäumen:

```
data Tree a = L a | N (Tree a) a (Tree a)

binsearch              :: Ord a => Tree a -> a -> Maybe a
binsearch (L y)     x = if x==y then Just y else Nothing
binsearch (N l y r) x = case compare x y of LT -> binsearch l x
                                            EQ -> Just y
                                            GT -> binsearch r x
```

Obwohl *binsearch* den Gleichheitstest verwendet, muss *Eq a* nicht im Kontext des Prozedurtyps auftauchen. Aus der Deklaration der Klasse *Ord* als Teilmenge von *Eq* kann nämlich abgeleitet werden, dass *Ord a* die Einschränkung *Eq a* impliziert.
Ein weiterer Vorteil hierarchischer Typklassenstruktur fällt uns auf, wenn wir die Default-Implementierung der Methode *compare* in *Ord* betrachten:

```
compare x y | x==y      = EQ
            | x<=y      = LT
            | otherwise = GT
```

Man sieht, wie die Implementierung von *compare* den Gleichheitsoperator verwendet. Dies ist nur möglich, weil durch den Ableitungsmechanismus sichergestellt ist, dass *Ord* immer eine Teilmenge von *Eq* beschreibt.

7 Ambiguität

Die Möglichkeit, durch die Einführung qualifizierter Typen die Menge der Belegungen einer Typvariablen nach Belieben einzuschränken, bedeutet einen großen Gewinn an Ausdrucksstärke gegenüber dem Hindley/Milner-Typsystem. Leider kann Verwendung qualifizierter Typen in bestimmten Fällen dazu führen, dass Typinferenz nicht mehr ausreichend Information liefert, um die Semantik eines Ausdrucks vollständig festlegen zu können. Ausdrücke, die von dem Problem betroffen sind, nennt man ambig. Ein prominentes Beispiel für Ambiguität ist die Kombination der Prozeduren *read* und *show*. *show* ist eine Methode der Klasse *Show* von Typen, deren Werte als Zeichenketten dargestellt werden können. Zu einem Wert eines solchen Typs liefert *show* die Darstellung des Werts als Zeichenkette. *read* wiederum liefert zu einer Zeichenkette ihre Interpretation als Wert eines anderen Typs. Typen, deren Werte auf diese Weise aus Zeichenketten eingelesen werden können, gehören zur Typklasse *Read*. Für *read* und *show* ergeben sich folgende Typbindungen:

```
read :: Read a => String -> a
show :: Show a => a -> String
```

Nun soll ein Wert zuerst mit *read* aus einer Zeichenkette s eingelesen und anschließend mit *show* wieder als Zeichenkette dargestellt werden. Der kurze Ausdruck `show(read s)` scheint auf den ersten Blick bestens geeignet, das Gewünschte zu tun. Doch welchen Typ hat dieser Ausdruck? Durch Kombination der Typen seiner Teilausdrücke erhalten wir:

```
show(read s) :: (Read a, Show a) => String
```

Der Gesamtausdruck hat den Typ *String*, seine Teilausdrücke arbeiten aber auf einem Zwischenergebnis vom Typ a, über den wir nur die Information haben, dass er zu Typklassen *Read* und *Show* gehören muss. Somit kommen für a mehrere Typen in Frage. Da die Typvariable a rechts von $\Rightarrow$ nicht vorkommt, hängt der Typ des Endergebnisses nicht vom Typ des Zwischenregebnisses ab. Ausgehend vom Typ des Gesamtausdrucks kann die Menge möglicher Belegungen für a daher nicht weiter eingeschränkt werden. Um den Ausdruck auswerten zu können müssen wir uns aber auf eine Belegung für a festlegen, welche dann die an *read* und *show* übergebenen Wörterbücher und so die Semantik des Gesamtausdrucks bestimmt. Wie wir sehen, sind ambige Ausdrücke nicht auswertbar.
Eine Möglichkeit, Ambiguität aufzulösen, besteht darin, Verwendung bestimmter Typen durch Typconstraints zu erzwingen. So kann für den Ausdruck `show(read s :: Int)` der nicht ambige Typ *String* ermittelt werden. Bedenkt man, dass das Typklassensystem die Möglichkeit von Typinferenz als eines seiner wesentlichen Ziele formuliert, ist solch eine Lösung sicherlich nicht zufriedenstellend.
Während Ambiguität im Typklassensystem, so wie es bisher vorgestellt wurde, eine eher seltene Erscheinung ist, existieren sinnvolle, für viele praktische Anwendungen

als nützlich empfundene Erweiterungen des Typklassensystems, deren Verwendung wegen häufigem Auftreten von Ambiguitäten problematisch ist. Das wohl wichtigste Beispiel einer solchen Erweiterung sind Multi-Parameter-Typklassen. Sie werden in Abschnitt 9 vorgestellt.

8 Typkonstruktoren und Konstruktorklassen

Mit der Deklaration von $Tree\ a$ lernten wir in Abschnitt 6 kennen, wie parametrisierte Typdeklarationen in Haskell aussehen. Der Typ

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

ist ein weiteres Beispiel für einen Datentyp mit einem Parameter. Dieser Typ wird in Haskell verwendet um Optionen darzustellen. Natürlich können Datentypen auch mehr als einen Parameter haben:

```
data Exp a b = Con a
             | Var b
             | Abs b (Exp a b)
             | App (Exp a b) (Exp a b)
```

$Exp\ a\ b$ stellt applikative Ausdrücke dar, wie sie in einem einfachen Interpreter auftauchen könnten. Betrachtet man die Syntax parametrisierter Typen, lassen sich Parallelen zu funktionaler Applikation ziehen. Objekte wie $Tree$, $Maybe$ oder Exp werden dabei auf Grundtypen wie Int oder $Bool$ angewandt und liefern neue Grundtypen wie $Tree\ Bool$ oder $Maybe\ Int$. Solche höherstufigen Typen nennt man Typkonstruktoren. Bei Exp handelt es sich demnach um einen kaskadierten Typkonstruktor. So liefert er bei seiner Anwendung auf den Grundtyp Int einen Typkonstruktor $Exp\ Int$. Wird dieser wiederum auf den Grundtyp $String$ angewandt, erhält man als Ergebnis der kaskadierten Konstruntoranwendung den Grundtyp $Exp\ Int\ String$. Wie wir festgestellt haben, sind Typkonstruktoren nichts anderes als Funktionen auf Typebene.
Obwohl Typkonstruktoren ebenso wie Grundtypen Bestandteil des Typsystems von Haskell sind, bilden Grundtypen insofern die wichtigere Klasse, als Werte immer einen Grundtyp haben. Typconstraints, die einem Wert einen Typkonstruktor zuweisen, müssen daher als nicht wohlgeformt erkannt werden. Andererseits kann auch die Anwendung von Grundtypen auf andere Typen, wie beispielsweise $Int\ Bool$, nicht wohlgeformt sein. Um Typen auf Wohlgeformtheit zu testen bedarf es eines eigenen Typsystems für Typen. Die Art und Weise wie ein Typ mit anderen verknüpft werden kann ohne die Wohlgeformtheit des Ganzen zu verletzen wird dabei formal unter dem Begriff der Art eines Typs zusammengefasst. Das System von Arten lässt sich wie folgt definieren:

- Grundtypen sind von der Art $*$.
- Typkonstruktoren sind von der Art $\kappa_1 \rightarrow \kappa_2$, wobei κ_1, κ_2 selbst Arten sind.

Sehen wir uns an, wie sich einige der von uns bisher betrachteten Typen und Typkonstruktoren nach Arten gliedern lassen:

```
Int, Bool, [Float] :: *
[], Tree, Maybe    :: * -> *
(->), Exp          :: * -> * -> *
```

Bis auf die Syntax ihrer Anwendung unterscheiden sich die Konstruktoren für Listentypen und und für Prozedurtypen nicht von anderen Typkonstruktoren. So wird der Pfeilkonstruktor zuerst auf den Argumenttyp τ_1, danach auf den Ergebnistyp τ_2 angewendet und liefert als Ergebnis seiner Anwendung den Grundtyp $\tau_1 \rightarrow \tau_2$. Um Missverständnisse zu vermeiden sei noch bemerkt, dass die Annotation von Typen mit ihren Arten, so wie in der obigen Auflistung, weder legal noch notwendig ist. Die Art eines Typs ist immer eindeutig und kann anhand entsprechender Typdeklarationen inferiert werden.
Nachdem wir durch die Einführung von Arten Grundtypen und Typkonstruktoren einheitlich behandeln können, wollen wir auch das Typklassensystem von Haskell verallgemeinern. Typkonstruktoren beliebiger Art sollen nun ebenso wie Grundtypen zu Klassen zusammengefasst werden können. Klassen von Typkonstruktoren nennen wir Konstruktorklassen. Um die Wohlgeformtheit von Instanzdeklarationen zu garantieren müssen wir fordern, dass alle Instanzen einer Konstruktorklasse von derselben Art sind. Die Art kann aus der Klassendeklaration abgeleitet werden. Die Verwendung von Konstruktorklassen wollen wir am Beispiel der Klasse *Functor* demonstrieren. *Functor* enthält Typen, auf denen die Operation *map* definiert ist. *map* ist eine wohlbekannte und häufig verwendete Operation, die folgende Gesetze erfüllt:

- $map\ id = id$
- $map\ f \circ map\ g = map\ (f \circ g)$

Wie deklarieren wir *Functor*?

```
class Functor f where
  map :: (a -> b) -> f a -> f b
```

Wir sehen, dass a, b, $f\ a$ und $f\ b$ Grundtypen sein müssen, sonst wäre der Typ von *map* nicht wohlgeformt. Für f folgt die Art $* \rightarrow *$. Da alle Instanzen von *Functor* gültige Belegungen für f sein müssen, ist *Functor* also eine Klasse von Typkonstruktoren der Art $* \rightarrow *$. Bekanntlich kann *map* u. a. auf Listen, Binärbäumen und Optionen definiert werden. Da die entsprechenden Konstruktoren, so wie wir sie deklariert haben, alle von der passenden Art sind, hindert uns nichts daran, sie zu Instanzen von *Functor* zu machen:

```
instance Functor [] where
  map f xs = [f x | x<-xs]

instance Functor Tree where
  map f (L x)     = L (f x)
  map f (N l x r) = N (map f l) (f x) (map f r)

instance Functor Maybe where
  map _ Nothing  = Nothing
  map f (Just x) = Just (f x)
```

Der Vergleich von *Functor* mit Definitionsansätzen für *map*, die nicht auf Konstruktorklassen basieren, sowie weitere motivierende Beispiele für Konstruktorklassen können [4] entnommen werden.

9 Multi-Parameter-Typklassen

Bisher waren alle von uns betrachteten Typklassendeklarationen von der Form

```
class C a where ...
```

wobei C der Name der Typklasse und a eine Parametervariable sind. Von Multi-Parameter-Typklassen spricht man, wenn Typklassendeklarationen mehr als eine Parametervariable haben. Instanzen von Multi-Parameter-Typklassen sind daher nicht einfach Typen sondern Tupel von Typen. Jeder Parametervariablen muss dabei eine Komponente des Tupels entsprechen. Da sowohl die Realisierung als auch die Verwendung von Multi-Parameter-Typklassen mit vielen noch nicht zufriedenstellend gelösten Problemen verbunden ist, wurde diese Erweiterung nicht in Haskell 98 aufgenommen. Dennoch werden Multi-Parameter-Typklassen von den meisten aktuellen Haskell-Implementierungen als Spracherweiterung angeboten, weil sie sich in der Praxis als sehr ausdrucksstark erwiesen haben.
Welche zusätzliche Ausdrucksmöglichkeiten gewinnen wir durch den Einsatz von Multi-Parameter-Typklassen? Wir können Beziehungen aller Art zwischen Typen darstellen, indem wir Tupel aus den zueinander in Relation stehenden Typen zu Instanzen geeigneter Typklassen machen. Als solche Relationen kommen beispielsweise Isomorphismen in Frage. Auch Typen, zwischen denen implizite Umwandlung (coercion) möglich ist, können zu Typklassen zusammengefasst werden. Die Darstellung dieser und anderer Beziehungen zwischen Typen werden ebenso wie einige Fragestellungen zur Realisierung von Multi-Parameter-Typklassen in [5] ausführlich diskutiert.
Ein interessanter Spezialfall sind Typklassen, die Relationen zwischen Typkonstruktoren und Grundtypen darstellen. Im Falle von *Functor* ist die überladene Prozedur *map* unabhängig vom Parametertyp einer Instanz der Typklasse definiert. Will man auf Konstruktorklassen überladene Operationen verwenden, bei denen der Parametertyp eine Rolle spielt, so kann man die Anwendbarkeit des Konstruktors auf passende Typen einschränken, indem man Typklassen über Tupel aus dem Konstruktor und dem Parametertyp definiert. Dann können für die gegebenen Operationen geeignete Parametertypen in Abhängigkeit vom Typkonstruktor durch Instanzdeklarationen festgelegt werden. Dieser Ansatz wird in [6] zur Implementierung von Datenstrukturbibliotheken mit überladenen universellen Zugriffsoperationen vorgeschlagen.
An einem ähnlichen Beispiel wollen wir den Einsatz von Multi-Parameter-Typklassen näher betrachten. Wir wollen allerdings ohne Typkonstruktoren als Parameter auskommen. Nehmen wir an, wir wollen Algorithmen formulieren, die auf allen Datenstrukturen funktionieren, deren Elemente durch Indizes erreichbar sind. Dann können wir eine Typklasse *Indexed* deklarieren, welche die Zugriffsoperationen abstrahiert:

```
class Indexed c a i where
  sub :: c -> i -> a
  idx :: c -> a -> Maybe i
```

Die Prozedur *sub* soll zu einer Datenstruktur vom Typ c und einem Index vom Typ i das an der indizierten Stelle gespeicherte Element vom Typ a liefern. Die Prozedur *idx* soll zu einer Datenstruktur und einem Element entscheiden, ob das Element in der Datenstruktur gespeichert ist und im positiven Fall den zugehörigen Index liefern. Als Instanzen von *Indexed* sind u. a. sortierte Reihungen oder endliche Funktionen

geeignet. Ohne uns mit den entsprechenden Typdeklarationen aufzuhalten, wollen wir uns ansehen, wie man beide Datenstrukturen zu *Indexed* hinzufügen kann:

```
instance Ord a => Indexed (SortedArray a) a Int where ...

instance Eq a => Indexed (Map a b) b a where ...
```

Bei sortierten Reihungen muss man beachten, dass auf den Elementen einer Reihung eine Ordnungsrelation definiert sein muss. Sonst kann die Reihung nicht sortiert werden. Dies wird durch die Einschränkung des Elementtyps auf Instanzen der Klasse *Ord* erreicht. Als Indextyp kommt nur *Int* in Frage. Endliche Funktionen legen auf den Elementtyp keine Einschränkungen auf. Die Indizes müssen aber sinnvollerweise als Mindestanforderung den Gleichheitstest zulassen. Wird Wert auf Effizienz gelegt, sind auch stärkere Forderungen an den Indextyp denkbar. Wie wir sehen, können sowohl der Elementtyp als auch der Indextyp in Abhängigkeit vom Datenstrukturtyp eingeschränkt werden um so die überladenen Zugriffsoperationen auf möglichst viele passende Typkombinationen erweitern zu können. Im Hinblick auf die folgenden Ausführungen halten wir noch fest, dass in beiden Instanzdeklarationen sowohl der Elementtyp als auch der Indextyp eindeutig durch den Datenstrukturtyp festgelegt sind. Haben wir eine Datenstruktur vom Typ *Map Int Bool*, so müssen der Elementtyp *Bool* und der Indextyp *Int* sein.
Unsere Lösung ist leider nicht ganz unproblematisch. Angenommen, es gelten die folgenden Typbindungen:

```
c :: CollType
i :: IdxType
```

Wir wollen mit *sub* zur Datenstruktur *c* das durch *i* indizierte Element finden und anschließend mit *idx* wieder den Index des gefundenen Elements ermitteln. Für den resultierenden Ausdruck erhalten wir die Typbindung

```
idx c (sub c i) :: Indexed CollType a IdxType => Maybe IdxType
```

Wir begegnen wieder dem Problem der Ambiguität. Obwohl der Elementtyp bei allen sinnvollen Instanzen von *Indexed* durch den Datenstrukturtyp eindeutig festgelegt ist, geht diese Information nicht in die Klassendeklaration ein. Wir können problemlos sinnlose aber korrekt getypte Instanzen von *Indexed* deklarieren, bei denen ein und derselbe Datenstrukturtyp in Relation zu jeweils unterschiedlichen Element- und Indextypen auftritt. Daher kann der unbekannte Elementtyp *a* nicht aus *CollType* abgeleitet werden. Um dieses Problem zu lösen hätte man gerne die Möglichkeit, funktionale Abhängigkeiten zwischen den Parametern einer Typklasse explizit auszudrücken. Dann könnte man aus *CollType* den eindeutig zugeordneten Elementtyp *a* inferieren. Eine Erweiterung des Typklassensystems um funktionale Anhängigkeiten wird in [7] vorgeschlagen.

10 Zusammenfassung: Typklassen und OOP

Das Ziel hinter dem Typklassensystem ist die Verbesserung der Wiederverwendbarkeit von Programmcode, das einerseits durch Einführung von Polymorphismus auf eingeschränkten Mengen von Typen, andererseits durch die Möglichkeit diese Mengen durch neue Typen zu erweitern erreicht werden soll. Auf diese Weise lassen sich

bereits vorhandene Operationen und die auf ihnen basierenden Algorithmen mit geringem Aufwand auf neu eingeführte Datenstrukturen ausweiten und so bewährte Lösungsstrategien auf neue Probleme anwenden. Dieselben Prinzipien liegen auch dem Konzept der objektorientierten Programmierung (OOP) zugrunde. Zwischen Typklassen und OOP lässt sich leicht begriffliche wie konzeptuelle Verwandtschaft erkennen. Diese Verwandschaft wollen wir zum Schluss etwas näher untersuchen.
Welche Analogien lassen sich zwischen Typklassen und Klassen erkennen, wie sie etwa in Java oder in C++ vorkommen? Wie wir wissen, legt eine Typklassendeklaration Operationen fest, die in einer Instanzdeklaration für einen Typ implementiert werden müssen, damit der Typ in die Typklasse aufgenommen wird. Diese Funktion entspricht ziemlich genau dem, was in Java durch Schnittstellen (interfaces) oder in C++ durch abstrakte Basisklassen erreicht wird. In beiden Fällen definiert man eine Schnittstelle bestehend aus Methoden, welche eine neue Abstraktionsebene bildet, auf der alle die Schnittstelle erfüllenden Klassen einheitlich behandelt werden können. Besonders groß ist die Ähnlichkeit zu abstrakten Klassen in C++. Denn diese bieten ebenso wie Typklassendeklarationen die Möglichkeit, für Methoden Default-Implementierungen anzugeben. Auch der Methodenbegriff hat im Typklassensystem und in OOP in etwa dieselbe Bedeutung.
Es gibt aber auch eine Reihe von Unterschieden zwischen OOP und Typklassen. Einer davon ist die strikte Trennung von Typdeklarationen, Typklassen- und Instanzdeklarationen, die für das Typklassensystem charakteristisch ist. In OOP ist der Klassenbegriff synonym zum Typbegriff. Die Deklaration einer nicht abstrakten Klasse in C++ entspricht zugleich einer Typdeklaration und einer Instanzdeklaration, deren Typklasse durch die Basisklasse definiert ist. Insofern kann eine Klassendeklaration in C++ oder Java auch als eine Typklassendeklaration angesehen werden, deren Instanzen die Klasse selbst sowie alle von ihr abgeleiteten Klassen sind. Es ist daher nicht verwunderlich, dass der Subtyping-Polymorphismus, d. h. die Definition polymorpher Operationen auf einem Typ zusammen mit allen von ihm abgeleiteten Typen, die ‚‚natürliche“, dominante Art von Polymorphismus in Java oder in C++ ist. Beim Polymorphismus im Typklassensystem handelt es sich dagegen um universellen Polymorphismus, d. h. um Polymorphismus auf Typen, zwischen denen keine hierarchische Ordnung bestehen muss. Dass universeller Polymorphismus sinnvoll ist, sieht man daran, dass er sowohl in C++ als auch in Java durch spezielle Formen von Subtyping-Polymorphismus simuliert wird. In C++ nutzt man dazu abstrakte Basisklassen in Verbindung mit Mehrfachvererbung, in Java den Schnittstellenmechanismus.
Ein Vorteil des Typklassensystems gegenüber OOP besteht in der größeren Leistungsfähigkeit und Präzision der dynamischen Bindung durch Wörterbücher. Operationen, die auf Typklassen überladen sind, können ihr Verhalten vom Typ beliebiger Argumente ebenso wie von dem durch Typinferenz ermittelten Ergebnistyp abhängig machen. Ein Beispiel für die letztere Möglichkeit ist die schon bekannte Prozedur *read*. Wie wir uns erinnern, kann sie zu einer Zeichenkette Ergebnisse von unterschiedlichen Typen liefern. Der Typ des Ergebnisses wird durch dessen Verwendung im weiteren Verlauf der Auswertung bestimmt. Besteht in OOP eine grundsätzliche Trennung zwischen dynamischer und statischer Bindung, die besonders deutlich in C++ durch die Unterscheidung virtueller und nicht virtueller Funktionen sichtbar ist, so tritt im Typklassensystem statische Bindung vielmehr als eine Optimierung des ursprünglich dynamischen Bindungssystems auf. Können bei einer überladenen Operation einige Typen statisch ermittelt werden, so können Zugriffe auf entsprechende Wörterbücher durch Beta-Reduktion in monomorphe Prozeduranwendungen

übersetzt werden.
Eine Funktionalität objektorientierter Systeme, die man im Typklassensystem auf den ersten Blick vermissen mag, ist die Kontrolle über die Sichtbarkeit von Feldern einer Klasse. Während in OOP Datenabstraktion und Modularisierung von Programmen als Aufgaben des Klassensystems angesehen werden, werden diese Funktionen in Haskell vom Modulsystem, nicht vom Typklassensystem übernommen. Abstrakte Datentypen können in Haskell somit nur auf Modulebene eingeführt werden.
Eine wesentliche Einschränkung des Typsystems von Haskell gegenüber objektorientierten Typsystemen ist, dass die so eingeführten Implementierungen abstrakter Datentypen keine Objekte erster Klasse sind. Ein Algorithmus kann nicht zur Laufzeit eine Implementierung eines abstrakten Datentyps gegen eine andere austauschen. Ebensowenig können heterogene Datenstrukturen verarbeitet werden, etwa Listen, deren Elemente unterschiedlichen Implementierungen eines abstrakten Datentyps angehören. Durch die Verschmelzung der Funktionalität des Typsystems und des Modulsystems im Klassensystem wird dieses Problem in OOP elegant gelöst. In Haskell kann es durch eine Erweiterung des Typsystems um so genannte existenzielle Typen gelöst werden. Als ein von Typklassen ursprünglich unabhängiges Sprachkonstrukt können existenzielle Typen, wie in [8] gezeigt, mit Typklassen zu einem Typsystem kombiniert werden, in dem sich durch die Interaktion beider Mechanismen zusätzliche Ausdrucksmöglichkeiten eröffnen.

Literatur

[1] R. Milner. A theory of type polymorphism in programming. Journal of Computer and System Sciences 17, 348-375, 1978.

[2] S.P. Jones (editor). Haskell 98 Language and Libraries. The Revised Report. December 2002. http://www.haskell.org/definition/

[3] P. Wadler and S. Blott. How to make ad-hoc polymorphism less ad-hoc. In 16th ACM Symposium on Principles of Programming Languages, Austin, Texas, January 1989.

[4] M.P. Jones. A system of constructor classes: overloading and implicit higher-order polymorphism. Journal of Functional Programming 5, 1-36, January 1995.

[5] S.P. Jones, M. Jones and E. Meijer. Type classes: Exploring the design space. In proceedings of the Second Haskell Workshop, Amsterdam, June 1997.

[6] S.P. Jones. Bulk type with class. In Proceedings of the Second Haskell Workshop, Amsterdam, June 1997.

[7] M.P. Jones. Type Classes with Functional Dependencies. In Proceedings of the 9th European Symposium on Programming, ESOP 2000, Berlin, Germany, March 2000. Springer-Verlag.

[8] K. Läufer. Type classes with existential types. Journal of Functional Programming 6(3), 485-517, May 1996.