Typklassen: Haskell

Mark Kaminski

Betreuer: Andreas Rossberg

1. April 2004







Zwei Arten von Polymorphismus

- parametrischer Polymorphismus
  - Prozedur definiert über eine Reihe von Typen, gleiches Verhalten für alle Typen
    Beispiel: `foldl` in Standard ML
  - Hindley/Milner-Typsystem (ML, Miranda)

Zwei Arten von Polymorphismus

- parametrischer Polymorphismus
  - Prozedur definiert über eine Reihe von Typen, gleiches Verhalten für alle Typen
    Beispiel: `foldl` in Standard ML
  - Hindley/Milner-Typsystem (ML, Miranda)
- ad-hoc - Polymorphismus (Überladen)
  - Prozedur definiert über mehrere Typen, potenziell unterschiedliches Verhalten für jeden Typ
    Beispiele: +,* in Standard ML, >> in C++
  - wichtig bei der Realisierung von arithmetischen Operatoren, Vergleichsoperatoren
  - unterschiedliche Ansätze in verschiedenen Sprachen







Parametrischer Polymorphismus und Überladen

Arithmetik: Überladen von Operatoren
Beispiel: Standard ML

```
fun square x = x*x
> val square : int -> int
```







Parametrischer Polymorphismus und Überladen

Arithmetik: Überladen von Operatoren
Beispiel: Standard ML

```
fun square x = x*x
> val square : int -> int
```

Problem: `square 3.14` nicht möglich







Parametrischer Polymorphismus und Überladen (2)

Gleichheit

- frühes Standard ML:
  - Gleichheitsoperator überladen







Parametrischer Polymorphismus und Überladen (2)

Gleichheit

- frühes Standard ML:
  - Gleichheitsoperator überladen
  - Problem:

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

  member ist nicht polymorph







Parametrischer Polymorphismus und Überladen (2)

Gleichheit

- frühes Standard ML:
  - Gleichheitsoperator überladen
  - Problem:

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

    member ist nicht polymorph
- Caml, Miranda:
  - Gleichheitstest polymorph





Parametrischer Polymorphismus und Überladen (2)

Gleichheit

- frühes Standard ML:
  - Gleichheitsoperator überladen
  - Problem:

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

    member ist nicht polymorph
- Caml, Miranda:
  - Gleichheitstest polymorph
  - Probleme: Laufzeitfehler beim Test von Prozeduren, inadequater Vergleich auf abstrakten Datentypen







Parametrischer Polymorphismus und Überladen (3)

Gleichheit

- Standard ML:
  - Einschränkung des Polymorphismus
  - Einführung von Typen mit Gleichheit (eqtypes)







Parametrischer Polymorphismus und Überladen (3)

Gleichheit

- Standard ML:
  - Einschränkung des Polymorphismus
  - Einführung von Typen mit Gleichheit (eqtypes)
  - Mängel:
    - Anwendung des Gleichheitsoperators auf abstrakte Datentypen nur möglich, wenn Gleichheit mit Darstellungsgleichheit zusammenfällt
    - Vergleich eines komplexen Datentyps nicht möglich, sobald der Typ mindestens eine nicht vergleichbare Komponente enthält







Typklassen: Grundideen

- Verallgemeinerung des Konzepts von eqtypes







Typklassen: Grundideen

- Verallgemeinerung des Konzepts von eqtypes
- Operationen sind nur auf "passende" Typen anwendbar.
  Information über geeignete Argumenttypen geht als sog. Kontext in den Typ der Operation ein.

  Beispiele:

  – `(==) :: Eq a => a -> a -> Bool`
    – Gleichheitstest anwendbar auf Typen mit Gleichheit, angezeigt durch den Kontext $Eq\ a$
  – `(^)  :: (Num a, Integral b) => a -> b -> a`
    – ganzzahlige Potenzierung fordert ganzzahlige Exponenten

  Eq, Num, $Integral$ sind Typklassen.
  Ein Typschema mit Kontext wird als qualifizierter Typ bezeichnet.

Typklassen: Grundideen

- Verallgemeinerung des Konzepts von eqtypes
- Operationen sind nur auf "passende" Typen anwendbar.
  Information über geeignete Argumenttypen geht als sog. Kontext in den Typ der Operation ein.

  Beispiele:

  – `(==) :: Eq a => a -> a -> Bool`
    – Gleichheitstest anwendbar auf Typen mit Gleichheit, angezeigt durch den Kontext $Eq\ a$
  – `(^)  :: (Num a, Integral b) => a -> b -> a`
    – ganzzahlige Potenzierung fordert ganzzahlige Exponenten

  Eq, Num, $Integral$ sind Typklassen.
  Ein Typschema mit Kontext wird als qualifizierter Typ bezeichnet.
- Der Benutzer kann neue Typklassen einführen und die bestehenden um neue Typen (Instanzen) erweitern.







Klassisches Beispiel: Gleichheit

Definition der Klasse Eq von Typen mit Gleichheit:

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

- Eingeführte Typbindung:

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

- == wird als Methode der Klasse Eq deklariert. Jede Instanz der Klasse Eq muss eine Implementierung für == angeben.







Klassisches Beispiel: Gleichheit (2)

Beispiele für Instanzen der Klasse Eq:

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

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







Klassisches Beispiel: Gleichheit (2)

Beispiele für Instanzen der Klasse Eq:

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

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



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







Arithmetik

Klasse Num von Zahlen (vereinfacht):

```
class Num a where
  (+), (-), (*) :: a -> a -> a
```







Arithmetik

Klasse Num von Zahlen (vereinfacht):

```
class Num a where
  (+), (-), (*) :: a -> a -> a
```

Realisierung von $square$ mit Typklassen:

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

Eingeführte Typbindung:

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







Arithmetik

Klasse Num von Zahlen (vereinfacht):

```
class Num a where
  (+), (-), (*) :: a -> a -> a
```

Realisierung von $square$ mit Typklassen:

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

Eingeführte Typbindung:

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

$square$ ist nun auf alle Instanzen von Num anwendbar.

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







Ableitung von Typklassen

Konzepte:

- Abgeleitete Klasse “erbt” alle Eigenschaften ihrer Basisklasse(n) und erweitert sie um neue.







Ableitung von Typklassen

Konzepte:

- Abgeleitete Klasse “erbt” alle Eigenschaften ihrer Basisklasse(n) und erweitert sie um neue.
- Jede Instanz der abgeleiteten Klasse ist zugleich eine Instanz der Basisklasse.







Ableitung von Typklassen

Konzepte:

- Abgeleitete Klasse “erbt” alle Eigenschaften ihrer Basisklasse(n) und erweitert sie um neue.
- Jede Instanz der abgeleiteten Klasse ist zugleich eine Instanz der Basisklasse.

Beispiel: Definition der Klasse Ord der angeordneten Typen

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

Ein Typ a kann nur dann Instanz der Klasse Ord sein, wenn er gleichzeitig als Instanz von Eq deklariert ist.

Ambiguität

Gegeben seien folgende Signaturen:

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

Welchen Typ hat der Ausdruck `show(read s)`?







Ambiguität

Gegeben seien folgende Signaturen:

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

Welchen Typ hat der Ausdruck `show(read s)`?

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







Ambiguität

Gegeben seien folgende Signaturen:

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

Welchen Typ hat der Ausdruck `show(read s)`?

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

Um welche Instanz von $Read$, $Show$ handelt es sich beim Typ a?







Ambiguität

Gegeben seien folgende Signaturen:

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

Welchen Typ hat der Ausdruck `show(read s)`?

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

Um welche Instanz von $Read$, $Show$ handelt es sich beim Typ a?

- Bei der Verwendung von Typklassen kann Typinferenz versagen.







Ambiguität

Gegeben seien folgende Signaturen:

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

Welchen Typ hat der Ausdruck `show(read s)`?

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

Um welche Instanz von $Read$, $Show$ handelt es sich beim Typ a?

- Bei der Verwendung von Typklassen kann Typinferenz versagen.
- Auflösung von Ambiguitäten durch explizite Typconstraints möglich: `show(read s :: Int)`
  Im Allgemeinen widerspricht dies aber der Idee hinter Typklassen.

Ambiguität

Gegeben seien folgende Signaturen:

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

Welchen Typ hat der Ausdruck `show(read s)`?

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

Um welche Instanz von $Read$, $Show$ handelt es sich beim Typ a?

- Bei der Verwendung von Typklassen kann Typinferenz versagen.
- Auflösung von Ambiguitäten durch explizite Typconstraints möglich: `show(read s :: Int)`
  Im Allgemeinen widerspricht dies aber der Idee hinter Typklassen.
- Sinnvolle Erweiterungen des Typklassensystems von Haskell verschlimmern das Problem der Ambiguität.

Typkonstruktoren

Beispiel: Optionstypen in Haskell

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

Analog zur funktionalen Applikation lassen sich Typen wie $Maybe\ Int$ darstellen als die Anwendung von $Maybe$ auf Int.
Objekte wie $Maybe$, die auf Typen angewandt wiederum Typen ergeben, heißen Typkonstruktoren.
Im Gegensatz zu diesen werden "gewöhnliche" Typen wie Int auch Grundtypen genannt.







Typkonstruktoren (2)

- Funktionen auf Typebene







Typkonstruktoren (2)

- Funktionen auf Typebene
- als höherstufige Typen Bestandteil des Typsystems von Haskell







Typkonstruktoren (2)

- Funktionen auf Typebene
- als höherstufige Typen Bestandteil des Typsystems von Haskell
- Werte haben immer einen Grundtyp







Typkonstruktoren (2)

- Funktionen auf Typebene
- als höherstufige Typen Bestandteil des Typsystems von Haskell
- Werte haben immer einen Grundtyp

  Formalisierung: System von **Arten**

  – Arten (kinds) sind "Typen von Typen".

  – Grundtypen sind von der Art $*$.

  – Konstruktoren sind von der Art $\kappa_1 \longrightarrow \kappa_2$, wobei κ_1, κ_2 selbst Arten sind.

  Beispiele:

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







Konstruktorklassen

Typkonstruktoren können ebenso wie Grundtypen zu Klassen zusammengefasst werden.

Konstruktorklassen

Typkonstruktoren können ebenso wie Grundtypen zu Klassen zusammengefasst werden.

Beispiel: Die **Functor**-Klasse

$Functor$ enthält Typen, für die die Operation map verfügbar ist.

Es gilt:

- *map id* $=$ *id*
- *map (f . g)* $=$ *map f . map g*







Konstruktorklassen (2)


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







Konstruktorklassen (2)


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

Da $f\ a$ ein Grundtyp sein muss, müssen alle Instanzen von Functor von der Art $* \longrightarrow *$ sein.

Konstruktorklassen (2)


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

Da $f\ a$ ein Grundtyp sein muss, müssen alle Instanzen von Functor von der Art $* \longrightarrow *$ sein.

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

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

Multi-Parameter-Typklassen

Eine Klasse wird über Tupeln von Typen/Konstruktoren definiert.

Typische Anwendungen:

- Einschränkung der Anwendbarkeit eines Konstruktors auf bestimmte Typen
- Darstellung von Beziehungen zwischen Typen







Multi-Parameter-Typklassen (2)

Beispiel: Datenstrukturen mit Zugriff durch Indizes

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

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

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

Multi-Parameter-Typklassen (2)

Beispiel: Datenstrukturen mit Zugriff durch Indizes

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

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

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

Je nach Fall können Elementtyp und Indextyp eingeschränkt werden.

Typklassen und OOP

Ähnlichkeiten mit Klassen in C++/Java:

- Standardoperationen können auf vom Benutzer definierte Typen ausgeweitet werden.
- Klassendefinitionen entsprechen Schnittstellen (interfaces) in Java, abstrakten Basisklassen in C++.
- Methoden einer Typklasse entsprechen virtuellen Funktionen in C++.







Typklassen und OOP (2)

Unterschiede:

- universeller Polymorphismus statt Subtyping-Polymorphismus
- Flexibleres Typsystem erlaubt neue Anwendungsmöglichkeiten für ad-hoc - Polymorphismus, z. B. dispatch nach (inferiertem) Ergebnistyp einer Prozedur.
- Trennung von Typdeklaration und Implementierung
- keine Kontrolle über die Sichtbarkeit von Feldern einer Klasse (Dies ist die Aufgabe des Modulsystems.)
- heterogene Datenstrukturen nur über Zusatzmechanismus (existenzielle Typen – eine Erweiterung des Typsystems)

Zusammenfassung

- entwickelt für die Programmiersprache Haskell







Zusammenfassung

- entwickelt für die Programmiersprache Haskell
- generalisieren das Konzept von eqtypes

Zusammenfassung

- entwickelt für die Programmiersprache Haskell
- generalisieren das Konzept von eqtypes
- Erweiterung und Verallgemeinerung des Hindley/Milner-Systems







Zusammenfassung

- entwickelt für die Programmiersprache Haskell
- generalisieren das Konzept von eqtypes
- Erweiterung und Verallgemeinerung des Hindley/Milner-Systems
- ermöglichen die Typinferenz verbunden mit dem Überladen







Zusammenfassung

- entwickelt für die Programmiersprache Haskell
- generalisieren das Konzept von eqtypes
- Erweiterung und Verallgemeinerung des Hindley/Milner-Systems
- ermöglichen die Typinferenz verbunden mit dem Überladen
- verbinden den parametrischen Polymorphismus mit dem ad-hoc-Polymorphismus







Literatur

[1] 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.

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

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

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

[5] 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.

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

Klassisches Beispiel: Gleichheit (3)

Überführung in das Hindley/Milner-System:

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

eq              :: EqDict a -> a -> a -> Bool
eq  (EqDict e) = e

eqDictInt   :: EqDict Int
eqDictInt   = EqDict eqInt
```







Klassisches Beispiel: Gleichheit (4)


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







Klassisches Beispiel: Gleichheit (4)


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

Überführung von Ausdrücken:

```
    1==2    →  eq eqDictInt 1 2
[1,2]==[]   →  eq (eqDictList eqDictInt) [1,2] []
```