GOETHE UNIVERSITÄT FRANKFURT AM MAIN

www.uni-frankfurt.de

Polymorphe Typen und Typklassen

David Sabel and Manfred Schmidt-Schauß

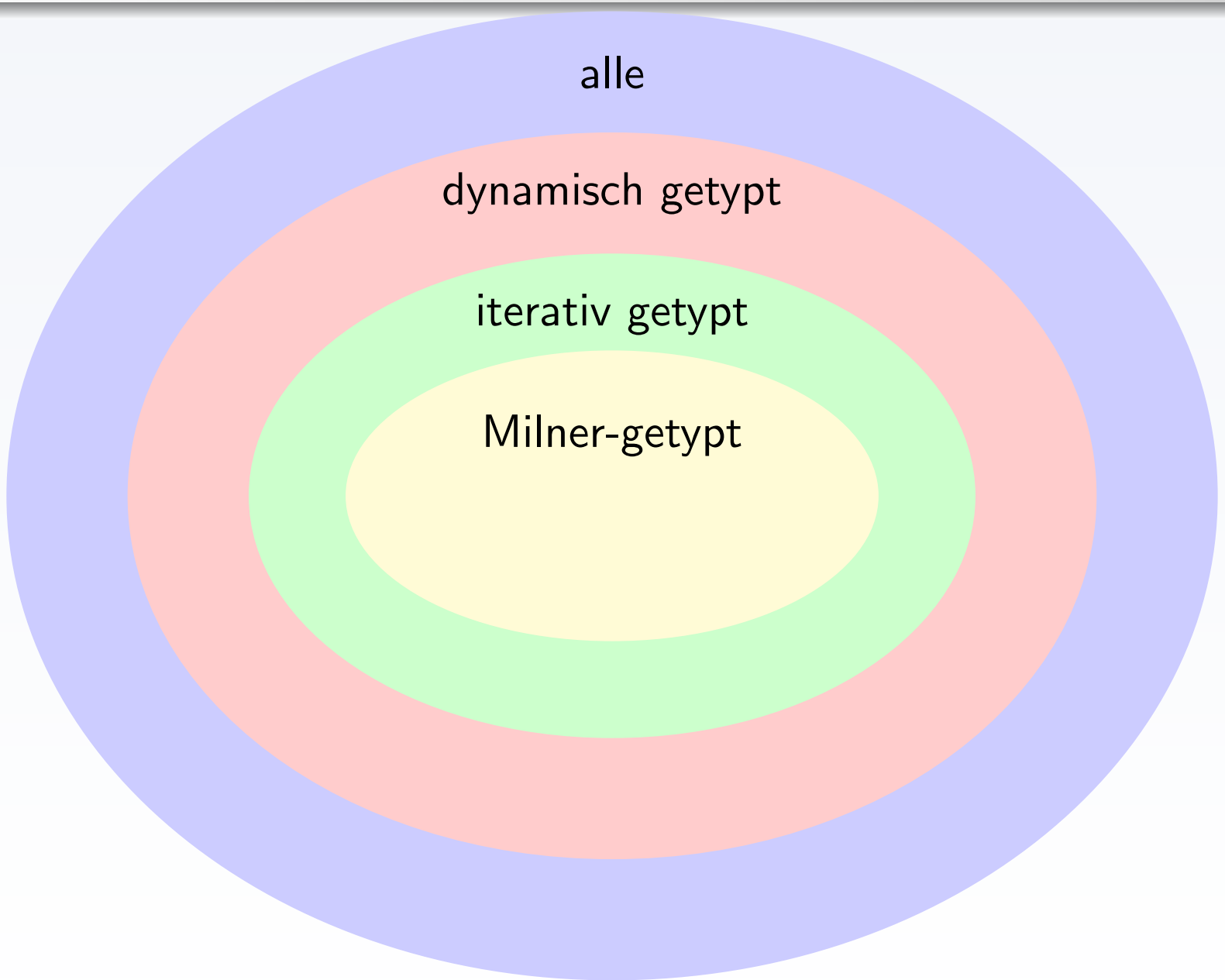
29. September 2015

Last update: 30. September 2015

Polymorphe Typen und Typklassen

1. (Parametrisch) Polymorphe Typen von Funktionen
2. Typklassen: Verwendung und Typfehler

Ausdrücke, getypt und ungetypt



Sinn der Typsysteme

- Für ungetypte Programme können dynamische Typfehler auftreten
- Fehler zur Laufzeit sind Programmierfehler
- Starkes und statisches Typsystem
  $\Longrightarrow$ keine Typfehler zu Laufzeit
- Typen als Dokumentation
- Typen bewirken besser strukturierte Programme
- Typen als Spezifikation in der Entwurfsphase

Sinn der Typsysteme

Minimalanforderungen:

- Die Typisierung sollte zur Compilezeit entschieden werden.
- Korrekt getypte Programme erzeugen keine Typfehler zur Laufzeit.

Sinn der Typsysteme

Minimalanforderungen:

- Die Typisierung sollte zur Compilezeit entschieden werden.
- Korrekt getypte Programme erzeugen keine Typfehler zur Laufzeit.

Wünschenswerte Eigenschaften:

- Typsystem schränkt wenig oder gar nicht beim Programmieren ein
- Compiler kann selbst Typen berechnen = Typinferenz

Typsysteme, allgemein

Haskell mit der polymorphen Milner-Typisierung erfüllt alle guten Eigenschaften

Es gibt Typsysteme, die diese Eigenschaften nicht erfüllen:

- Z.B. Simply-typed Lambda-Calculus ohne Fixpunkt-Kombinator:
  Diese Sprache ist nicht mehr Turing-mächtig, da dieses Typsystem erzwingt, dass alle Programme terminieren
- Erweiterungen in Haskells Typsystem:
  Typisierung / Typinferenz ist unentscheidbar.
  D.h. Manchmal terminiert der Compiler nicht!.

Polymorphe Typen

Definition

Die Syntax von polymorphen Typen kann durch die folgende Grammatik beschrieben werden:

$$\mathbf{T} ::= TV \mid TC\ \mathbf{T}_1\ \ldots\ \mathbf{T}_n \mid \mathbf{T}_1 \to \mathbf{T}_2$$

wobei TV für eine Typvariable steht und TC ein Typkonstruktor mit Stelligkeit n ist.

- polymorph: Typen haben Typvariablen
- z.B. `fakultaet` :: `Int` $\to$ `Int`
- z.B. `map` :: (`a` $\to$ `b`) $\to$ (`List a`) $\to$ (`List b`)

Polymorphe Typen; Haskell Konventionen

- $a \to b \to c$ bedeutet $a \to (b \to c)$.
- `->` statt $\to$
- `[a]` statt `(List a)`.
- Namen die mit Kleinbuchstaben beginnen in Typen sind **Typvariablen**;
  Namen die mit Großbuchstaben beginnen in Typen sind **Typkonstruktoren**.

Beispiele

`True`	::	`Bool`
`False`	::	`Bool`
`not`	::	$\texttt{Bool} \to \texttt{Bool}$
`map`	::	$(a \to b) \to [a] \to [b]$
$(\lambda x.x)$	::	$(a \to a)$
`length`	::	$[a] \to \texttt{Int}$
`genericLength`	::	$\texttt{Num}\ i \Rightarrow [a] \to i$??

Einfache Typregeln

- Für die Anwendung:

$$\frac{s :: T_1 \to T_2, t :: T_1}{(s\ t) :: T_2}$$

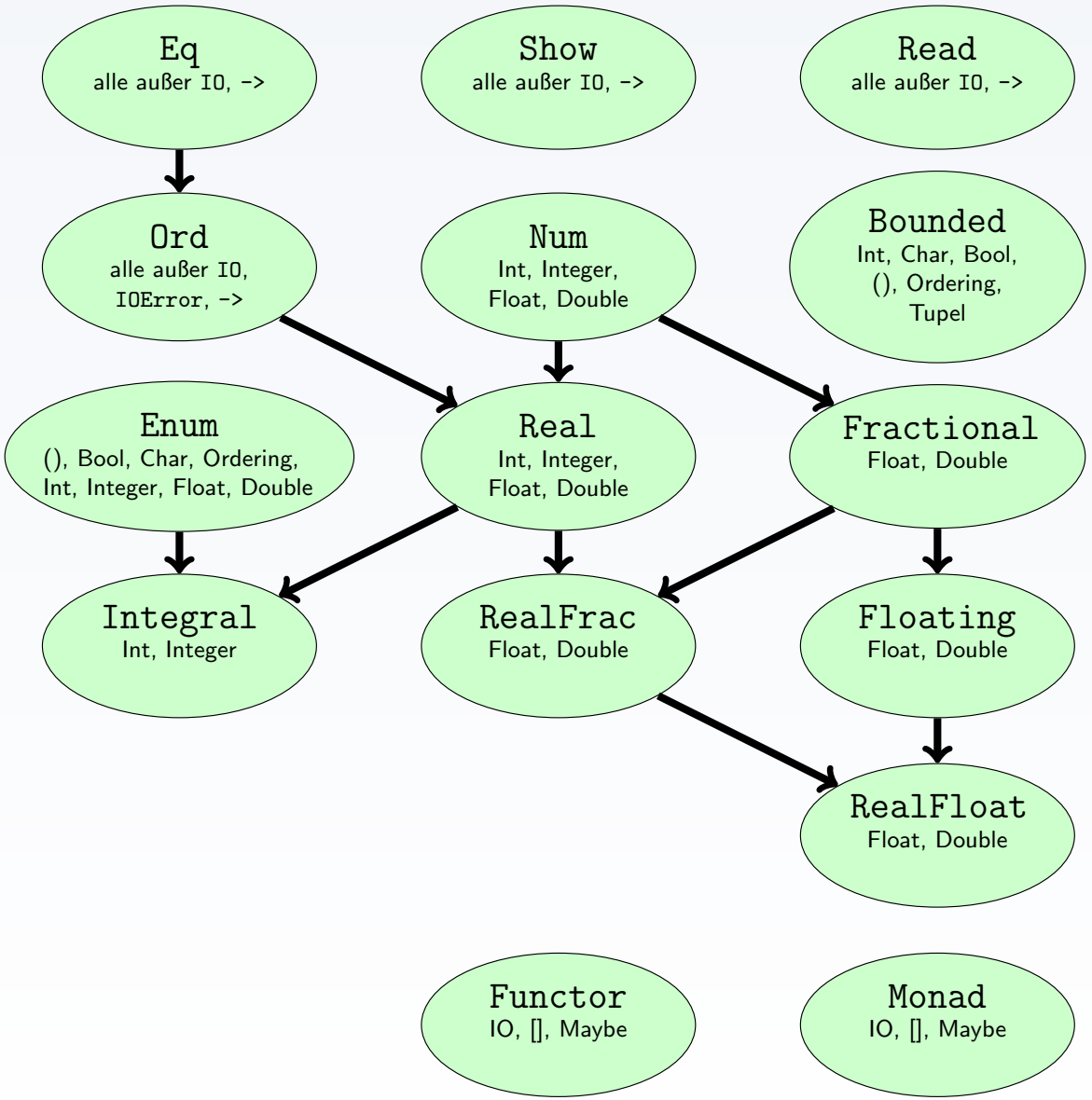
- Instantiierung

$$\frac{s :: T}{s :: T'}$$ wenn $T' = \sigma(T)$, wobei σ eine Typsubstitution ist, die Typen für Typvariablen ersetzt.

Beispiel

$$\frac{\dfrac{\texttt{map} :: (a \to b) \to [a] \to [b]}{\texttt{map} :: (\texttt{Bool} \to \texttt{Bool}) \to [\texttt{Bool}] \to [\texttt{Bool}]}, \texttt{not} :: \texttt{Bool} \to \texttt{Bool}}{(\texttt{map not}) :: [\texttt{Bool}] \to [\texttt{Bool}]}$$

Übersicht über die vordefinierten Typklassen



Typklassen in Haskell

- Typklassen **verbessern** die Ausdrucksfähigkeit des polymorphen Typsystems und haben Ähnlichkeit mit dem Klassensystem der objektorientierten Systeme
- Sie erlauben **Überladung von Symbolen**: z.B. gleiches Zeichen + für Addition von Ganzzahlen und rellen Zahlen, ...
- Sie sind unvermeidlich enthalten in den Typfehler-Meldungen des GHC.

Ad hoc Polymorphismus

- Eine Funktion (bzw. Funktionsname) f wird mehrfach für verschiedene Datentypen definiert.
- Implementierungen von f für verschiedene Typen sind unterschiedlich.
- Ad hoc-Polymorphismus nennt man auch Überladung.
- Beispiele
  - `+`, für `Integer`- und für `Double`-Werte.
  - `==` für `Bool` und `Integer`
  - `map` für Listen und Bäume

Haskells Typklassen implementieren Ad hoc Polymorphismus
$map : (a \to b) \to [a] \to [b]$ ist parametrischer Polymorphismus.

Typ-Syntax Erweiterung

GOETHE UNIVERSITÄT FRANKFURT AM MAIN

Constraints $=>$ Polymorpher Typ

Beispiel: `genericLength` :: `Num` $b => [a] \to b$

D.h. für alle Typen a
und alle Typen b, die in der Typklasse `Num` sind,
hat `genericLength` den Typ $[a] \to b$;

bzw. `genericLength` kann auf alle Listen angewendet werden;
und ergibt ein Objekt von numerischem Typ
(`Int`, `Integer`, `FLoat`, `Double`,...)

Typklassen

In der Klassendefinition wird festgelegt:

- Typ der Klassenfunktionen
- Optional: Default-Implementierungen der Klassenfunktionen

Typklassen

In der Klassendefinition wird festgelegt:

- Typ der Klassenfunktionen
- Optional: Default-Implementierungen der Klassenfunktionen

Pseudo-Syntax für den Kopf:

```
class [OBERKLASSE =>] Klassenname a where
  ... Typdeklarationen und Default-Implementierungen ...
```

- definiert die Klasse `Klassenname`
- a ist der Parameter für den Typ.
  Es ist nur eine solche Variable erlaubt
- `OBERKLASSE` ist eine Klassenbedingung (optional)
- Einrückung beachten!

Die Klasse Eq

Definition der Typklasse `Eq`:

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

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

- Hat keine Klassenbedingung
- Klassenmethoden sind == und /=
- Es gibt Default-Implementierungen für beide Klassenmethoden
- Instanzen müssen mindestens == oder /= definieren

Typklasseninstanz (1)

- Instanzen definieren die Klassenmethoden für einen konkreten Typ
- Instanzen können Default-Implementierungen überschreiben

Syntax für Instanzen:

```
instance [KLASSENBEDINGUNGEN => ] KLASSENINSTANZ where
  ...Implementierung der Klassenmethoden ...
```

- `KLASSENINSTANZ` besteht aus Klasse und der Instanz, z.B. `Eq [a]` oder `Eq Int`
- `KLASSENBEDINGUNGEN` optional

Typklasseninstanz (2)

Beispiele:

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

Typklasseninstanz (2)

Beispiele:

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

```
instance Eq Bool where
   x == y    =  (x && y) || (not x && not y)
   x /= y    =  not (x == y)
```

Typklasseninstanz (2)

Beispiele:

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

```
instance Eq Bool where
   x == y    =  (x && y) || (not x && not y)
   x /= y    =  not (x == y)
```

```
instance Eq Wochentag where
 Montag     == Montag     = True
 Dienstag   == Dienstag   = True
 Mittwoch   == Mittwoch   = True
 Donnerstag == Donnerstag = True
 Freitag    == Freitag    = True
 Samstag    == Samstag    = True
 Sonntag    == Sonntag    = True
 _          == _          = False
```

Typklasseninstanz (3)

Beispiel mit Klassenbedingung:

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

- Für die Abfrage `x == y` muss der Gleichheitstest auf Listenelementen vorhanden sein
- `Eq a => Eq [a]` drückt diese Bedingung gerade aus.
- „Typ `[a]` ist nur dann eine Instanz von `Eq`, wenn Typ `a` bereits Instanz von `Eq` ist“

Fehlermeldungen: Erklärung

```
> (\x->x) == (\y->y)
No instance for (Eq (t0 -> t0)) arising from a use of '=='
```

==-Methode ist nicht für Funktionstypen definiert.

Weitere Beispiele: Eingabe in den Interpreter GHCI

```
3+[]
seq (3 + []) 0
3 < []
(\x -> x)
data Mydata = MyCons Int
MyCons 1
data Mydata = MyCons Int deriving (Show,Eq,Read)
MyCons 0
MyCons 1 == MyCons 0
```