

Fortgeschrittene Techniken der Funktionalen Programmierung
Vorlesung vom 05.01.10:
Fancy Types

Christoph Lüth, Dennis Walter

Universität Bremen

Wintersemester 2009/10

Fahrplan

- ▶ Teil I: Monaden und fortgeschrittene Typen
- ▶ Teil II: Fortgeschrittene Datenstrukturen
- ▶ Teil III: Nebenläufigkeit
- ▶ Teil IV: The Future of Programming
 - ▶ Fancy Types
 - ▶ Domain-Specific Languages
 - ▶ The Next Big Thing: F#, Scala
 - ▶ Rückblick, Ausblick

Heute

- ▶ “Sexy Types”:
- ▶ Existentielle Typen
 - ▶ Objektorientierung in Haskell?
- ▶ Polymorphie Höherer Ordnung
- ▶ Typentheorie
- ▶ Entscheidbarkeit der Typüberprüfung

Funktional vs. Objektorientiert

- ▶ Gemeinsame Konzepte:
 - ▶ Polymorphie
 - ▶ Verkapselung (Klassen vs. ADTs/Module)
 - ▶ Klassenhierarchie
 - ▶ abstrakte Klassen (interface/class) vs. Instanzen (class/instance)
- ▶ nur OO: Zustandsbasiertheit, dynamische Bindung
- ▶ nur Funktional: algebraische Typen, Fkt. höherer Ordnung. strengere Typisierung

Dynamische Bindung

- Java: Auflösung der Methoden zur **Laufzeit**:

```
class Shape
{ void draw()
  { System.out.println("*"); }}
class Triangle extends Shape
{ void draw()
  { System.out.println("/\\"); }}
class Circle extends Shape
{ void draw()
  { System.out.println("()"); }}
```

Dynamische Bindung

- Java: Auflösung der Methoden zur **Laufzeit**:

```
class test {  
    public static void main(String [] a) {  
        Shape s;  
        if (a[0].equals("t")) { s= new Triangle(); }  
        else if (a[0].equals("s")) { s= new Circle(); }  
        else { s= new Shape(); }  
  
        s.draw(); } }
```

Dynamische Bindung in Haskell?

```
type Point = (Double, Double)

class Shape s where draw :: s → String
                    draw _ = "*"

data Triangle = Triangle Point Point Point
instance Shape Triangle where
    draw Triangle {} = "/\\\"

data Circle = Circle Point Double
instance Shape Circle where
    draw Circle {} = "()"
```

► So nicht ...

Dynamische Bindung in Haskell?

- ▶ Typ wird zur Übersetzungszeit berechnet.
 - ▶ Obwohl erst zur Laufzeit gebunden!
 - ▶ Implementierung von Klassen durch dictionaries
 - ▶ Typisierung verdeckt dynamische Bindung
- ▶ Warum dynamische Bindung?
 - ▶ Vorteil: Erweiterbarkeit eingebaut
 - ▶ Nachteil: Erweiterbarkeit nicht immer erwünscht

Existentielle Typen

- ▶ Idee aus der Logik: Curry-Howard-Isomorphie
- ▶ Getypter Lambda-Kalkül $\cong$ Intuitionistische Prädikatenlogik
- ▶ Beweis $\leftrightarrow$ Programm
- ▶ Allquantoren (höherer Ordnung) $\leftrightarrow$ Typvariablen
- ▶ Existenzquantoren $\leftrightarrow$

Existentielle Typen

- ▶ Idee aus der Logik: Curry-Howard-Isomorphie
- ▶ Getypter Lambda-Kalkül $\cong$ Intuitionistische Prädikatenlogik
- ▶ Beweis $\leftrightarrow$ Programm
- ▶ Allquantoren (höherer Ordnung) $\leftrightarrow$ Typvariablen
- ▶ Existenzquantoren $\leftrightarrow$ ADTs!

Abstract Data Types have Existential Type

- **Polymorphie**: allquantifizierte Typvariablen

```
forall a. data List a = Nil | Cons a (List a)
```

- Typkonstruktor für alle Typen instanzierbar

- **ADT**: existenzquantifizierte Typvariablen

```
data T = forall a. App a (a  $\rightarrow$  Int)
```

- Typkonstruktor beschreibt **einen** unbestimmten Typ
- NB. Nicht mehr **Haskell98** (nur `ghc`, `hugs`).

Ein einfaches Beispiel

- Ein existenzieller Typ:

```
data T = forall a. App a (a → Int)
```

```
ap :: T → Int
```

```
ap (App x f) = f x
```

- Anwendung:

```
map ap [App [1,2,3] length ,  
        App 3 (5 +),  
        App getLine (λ_ → 0)]
```

- Nützlich?
- Benötigt: **Signatur** für a

Heterogen Listen

- ▶ Signaturen im großen: Module
- ▶ Signaturen im kleinen: Typklassen
- ▶ Klassen für Typvariablen

```
data S = forall a. Show a  $\Rightarrow$  Cons a (a  $\rightarrow$  Int)  
  
instance Show S where  
    show (Cons a _) = show a
```

Dynamische Bindung in Haskell?

- **Beispiel:** Shapes revisited

```
class Shape s where draw :: s → String  
                    draw _ = "*"
```

- Damit heterogene Listen von Shapes (**selbstgemacht**)

```
data ShapeList =  
    forall s. Shape s ⇒ Cons s ShapeList  
  | Empty
```

- Obertyp **aller** Shapes:

```
data ShapeT = forall s. Shape s ⇒ Shape s  
instance Shape ShapeT where  
    draw (Shape s) = draw s
```

Dynamische Bindung in Haskell?

- ▶ Dreiecke und Kreise:

```
data Triangle = Triangle Point Point Point
instance Shape Triangle where draw Triangle {} = "/\

data Circle = Circle Point Double
instance Shape Circle where draw Circle {} = "()"
```

- ▶ Davon **unabhängig** Quadrate:

```
data Square = Square Point Point
instance Shape Square where draw Square {} = "[]"
```

- ▶ Zusammenfügen:

```
shapelist1 = [Shape tri, Shape sq, Shape circle]
shapelist2 = Cons tri (Cons sq (Cons circle Empty))
shapelist2' = foldr (\ (Shape x) l → Cons x l) Empty
```

Dynamische Bindung in Haskell?

- ▶ Auflösung der Bindung zur Laufzeit
- ▶ Damit: heterogene Datenstrukturen
- ▶ Keine echte Vererbung
- ▶ Datentypen müssen erweiterbar angelegt werden.

Polymorphie Höherer Ordnung

- ▶ Normale Polymorphie: allquantifizierte Typvariablen

```
data forall a. Maybe a = Nothing | Just a  
fromJust :: forall a. Maybe a → a  
loop :: forall a. (a → a) → Int → Int
```

- ▶ Allquantor immer **außen**.
- ▶ **Rang-n** Polymorphie: Allquantor **innen**.

```
loop :: (forall a. a → a) → Int → Int
```

- ▶ **Rang 1**: allquantifizierte Typvariablen
- ▶ **Rang $n + 1$** : Rang n auf der **linken** Seite eines Funktionstyps

Erstes Beispiel

- ▶ Zustandsübergangsmonaden, parametrisiert über Zustand s

```
type ST s a = s → (a, s)
```

- ▶ Dazu: Zustandsbehaftete Berechnung ausführen

```
runST :: ST s a → a
```

- ▶ Aber: Zustand **sichtbar** — a hängt von s ab.

```
let v = runST (newRef True) in runST (readVar v)
```

- ▶ **Autsch** — deshalb:

```
runST :: forall a. (forall s. ST s a) → a
```

- ▶ Allquantor links $\cong$ Existenzquantor
- ▶ Zustand kann nicht entkommen, a von s unabhängig.

Generische Programmierung

- ▶ Generischer Fixpunktoperator:

```
forall f. (forall a. (a  $\rightarrow$  a)  $\rightarrow$   
              (f a  $\rightarrow$  f a))  $\rightarrow$  (Fix f  $\rightarrow$  Fix f)
```

- ▶ Definiert wie folgt:

```
data Fix f = Fix (f (Fix f))
```

- ▶ Mit dem **Kind** $(* \rightarrow *) \rightarrow *$
- ▶ Monomorphe Typen haben Kind $*$
- ▶ Polymorphe Typen haben Kind $* \rightarrow *$
- ▶ vgl. **Konstruktorklassen**

```
class Monad m where  
  ( $\gg=$ ) :: m a  $\rightarrow$  (a  $\rightarrow$  m b)  $\rightarrow$  m b
```

Modellierung algebraischer Datentypen

- ▶ Listen als **Rang-2** Datentyp:

```
type List a = (forall l. l  $\rightarrow$  (a  $\rightarrow$  l  $\rightarrow$  l)  $\rightarrow$  l)
```

- ▶ `foldr` instantiiert `l`
- ▶ Initialer Morphismus
- ▶ Führt zur **Typentheorie** — System F (Girard)
 - ▶ Polymorphie als Grundkonzept,
 - ▶ alg. Datentypen abgeleitet.

$$\mathbb{N} = \prod X. X \rightarrow (X \rightarrow X) \rightarrow X$$

Aufwand der Typüberprüfung

- ▶ Mit existentiellen Typen, Rang- n -Polymorphie etc Typüberprüfung **unentscheidbar**.
 - ▶ D.h. Typcheck kann **divergieren**!
- ▶ Aber: **wen kümmert's?**
- ▶ Hindley-Milner (Haskell) ist **exponentiell**.
 - ▶ Kann **so gut wie** divergieren
 - ▶ Beispiel

Zusammenfassung

- ▶ “Abstract types have **existential type**”
- ▶ (Limitierte) Modellierung von Objektorientierung
 - ▶ Keine Vererbung, Erweiterbarkeit
- ▶ **Rang- n Polymorphie**
 - ▶ Beispiele: `runST`, generische Programmierung, ...
- ▶ Typüberprüfung wird **unentscheidbar**
 - ▶ ... aber schon Hindley-Milner-Typcheck ist exponentiell!
- ▶ Nächste Woche: **Keine Vorlesung!**
- ▶ Danach: DSLs (Domain-Specific Languages)