Fancy Types I: Existentielle Typen und Polymorphie höher Ordnung

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, *late binding*
- nur Funktional: algebraische Typen, Fkt. höherer Ordnung. strengere Typisierung

Dynamische Bindung

- Java: Auflösung der Methoden zur Laufzeit
- Hier ist ein Beispiel:

```
class Shape
  { void draw() { System.out.println("Shape"); }}
class Triangle extends Shape
  { void draw() { System.out.println("Triangle"); }}
class Square extends Shape
  { void draw() { System.out.println("Square"); }}
class test {
  public static void main(String [] a)
  ...
  s.draw();
  }
```

Späte Bindung in Haskell?

- Die Klassenhierarchie:

```
class Shape s where draw :: s-> String
                    draw _ = "Shape"
class Shape s=> Triangle s
class Shape s=> Circle s

data TriangleT = Triangle
instance Shape TriangleT where draw Triangle = "Triangle"
instance Triangle TriangleT

data CircleT   = Circle
instance Shape CircleT where   draw Circle   = "Circle"
instance Circle CircleT
```

- aber . . . :

```
main :: IO ()
main = do
  a<- getArgs
  let s= case (head a) of
           "t" -> Triangle; "c"-> Circle
  putStrLn (draw s)
```

- Haskell's Typisierung verdeckt dynamische Bindung
- Typ wird zur Übersetzungszeit berechnet.
  - Obwohl erst zur Laufzeit gebunden!
  - Implementierung von Klassen durch dictionaries

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. Cons a (a-> Int)
```

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

- Beispiel: Emulation von dynamischer Bindung:

```
data T = forall a. Cons a (a-> Int)
ap (Cons x f) = f x
map ap [Cons [1,2,3] length,
        Cons 3 (5 +),
        Cons getLine (\_-> 0)]
```

- Heterogen Listen!
- Erweiterung: Klassenbeschränkung der Typvariablen:

```
data T = forall a. Show a=> Cons a (a-> Int)

instance Show T where
   show (Cons a _ ) = show a
```

Dynamische Bindung in Haskell?

- Beispiel: Klassen wie vorher, aber:

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

main :: IO ()
main = do
  a<- getArgs
  let s= case (head a) of
            "t" -> Shape Triangle; "c"-> Shape Circle

  putStrLn (draw s)
```

Dynamische Bindung in Haskell?

- Auflösung der Bindung zur Laufzeit
- Damit: heterogene Datenstrukturen
- Klassenhierarchie unnötig
- Keine echte Vererbung
- Erweiterung nur a priori, nicht a posteriori
  - D.h. 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
```

- Zustand kann nicht entkommen, `a` von `s` unabhängig.

Generische Programmierung

- Generischer Fixpunktoperator:

```
forall f. (forall a. (a-> a)->
                  (f a-> f a))-> (Fix f-> Fix f)
```

- Definiert wie folgt:

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

- Mit dem Kind `(* -> *)-> *`
- Monomorphe Typen haben Kind `*`
- Polymorphe Typen haben Kind `*-> *`
- vgl. Konstruktorklassen

```
class Monad m where
  (>>=) :: m a-> (a-> m b)-> m b
```

Modellierung algebraischer Datentypen

- Listen als Rang-2 Datentyp:

```
type List a = (forall l. l-> (a-> l-> l)-> l)
```

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

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

Aufwand der Typüberprüfung

- Mit existentiellen Typen, Rang-n-Polymorphie:
  Typcheck wird unentscheidbar.
  - D.h. Typcheck kann divergieren!
- Aber: wen kümmert's?
- Auch Hindley-Milner (Haskell) ist exponentiell.
  - Kann divergieren.
  - Beispiel.

Zusammenfassung

- “Abstract types have existential type”
- (Limitierte) Modellierung von Objektorientierung
  - Fehlt: a posteriori Verfeinerung
- Rang-n Polymorphie
  - Beispiele: `runST`, generische Programmierung, ... $\rightsquigarrow$ Typentheorie
- Typüberprüfung wird unentscheidbar
  - ... aber schon Hindley-Milner-Typcheck ist exponentiell!