Plan für die Vorlesung im WS 00/01

Praxis der Funktionalen Programmierung

Johannes Waldmann, Institut für Informatik
Universität Leipzig,
joe@informatik.uni-leipzig.de

27. Juni 2000

Die Vorlesung vermittelt:

- Fähigkeit zum deklarativen Denken bei Problemanalyse und Algorithmenentwurf
- Fertigkeiten zu Implementierung in der funktionalen Programmiersprache *Haskell*

Literatur

Paul Hudak (Yale Haskell Group): *The Haskell School of Expression: Learning Functional Programming Through Multimedia* (2000)

Multimedia: Grafik, Reaktion, Musik

Vorteile des funktionalen Ansatzes:

- ein deklaratives Programm ist seine eigene Spezifikation
- Abstraktion durch Namen (Equational Reasoning)

  $\texttt{let}\ x = t\ \texttt{in}\ A \quad \equiv \quad A[x/t]$
- algorithmische Abstraktion (durch Funktionen höherer Ordnung)
```
foldr f z []     = z
foldr f z (x:xs) = f x (foldr f z xs)
```
- Abstraktion von Daten (durch polymorphe Funktionen)
```
foldr :: (a->b->b) -> b -> [a] -> b
```
- absolute Typ–Sicherheit

Multimedia–Themen

- deklarative 2D–*Grafik*
```
r4 = Shape (Polygon [(-2.5,2.5), (-3.0,0), (-1.7,-1.0),
                     (-1.1,0.2), (-1.5,2.0)] )
reg1 = r3 `xUnion`
       (r1 `Intersect` Complement r2 `Union` r4)
pic1 = Region Blue reg1
reg2 = let circle = Shape (Ellipse 0.5 0.5)
           square = Shape (Rectangle 1 1)
       in (Scale (2,2) circle)
          `Union` (Translate (1,0) square)
          `Union` (Translate (-1,0) square)
pic3 = pic2 `Over` pic1
```
- *Animation* :: Zeit → Grafik (Jonglier–Simulator)
- *Reaktive* Animation :: Ereignisse → (Zeit → Grafik) (Spiele)
- *Musik*

  Haskore :: Partitur → Midi–File

Denk– und Sprach–Elemente (basic)

- Ausdrücke, Funktionsdefinitionen

```
rectangle a b = [(0,0),(a,0),(a,b),(0,b)]
square d      = rectangle d d
```

- einfache Typen (Bool, Char, Integer), Listen

```
polygon_area :: [(Float,Float)] -> Float
polygon_area [(x1,y1),(x2,y2),(x3,y3)] = ...
polygon_area (p1 : p2 : p3 : rest)
    = polygon_area [p1, p2, p3]
    + polygon_area (p1 : p3 : rest)
```

- "naive" IO

```
main = runGraphics $ do
    w <- openWindow "hugs graphics window" (500, 500)
    drawInWindow w $ polygon (rectangle 200 300)
    k <- getKey w; closeWindow w
```


Denk– und Sprach–Elemente (basic)

- list comprehensions

```
mirror :: [(Float,Float)] -> [(Float,Float)]
mirror ps = [(y,x) | (x,y) <- ps]
```

- nutzerdefinierte Typen, pattern matching

```
data Shape = Polygon [(Float,Float)]
           | Ellipse Float Float
area :: Shape -> Float
area s = case s of
    Polygon ps  -> polygon_area ps
    Ellipse a b -> pi * a * b
```

- Polymorphie

```
mirror :: [(a,b)] -> [(b,a)]
```

- Typklassen (Show, Read, Ord, Eq), Instanzen

```
qsort :: Ord a => [a] -> [a]
qsort [] = []; qsort [x] = [x]
qsort (x : xs) =
    let (left, right) = partition ( < x) xs
    in qsort left ++ [x] ++ qsort right
```


Denk– und Sprach–El. (advanced)

- unendliche Listen (streams)

```
primes :: [ Integer ]
primes = sieve [2..] where
  sieve (p : rest)
    = p : sieve (filter ((/= 0) . (flip mod p)) rest)
flip f x y = f y x
```

- Rekursions–Schemata

```
foldl :: (b -> a -> b) -> b -> [a] -> b
foldl f z [] = z
foldl f z (x : xs) = f x : foldl f z xs
reverse = foldl (flip (:)) []
```

- Konstruktorklassen

```
class Functor f where
    fmap :: (a -> b) -> (f a -> f b)
data Maybe a = Nothing | Just a
instance Functor Maybe where
    fmap h Nothing  = Nothing
    fmap h (Just x) = Just (h x)
fmap (g . h) = fmap g . fmap h
```

- Monaden, do–Notation
- State–Monads, Arrays

Denk– und Sprach–El. (advanced)

- time & space profiling
- strictness
- foreign function interface
- Kompilation nach Java
- "a good programmer can write LISP in any language"

Beispiel–Thema: Animation/Simulation

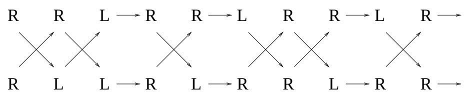
- Mehrpersonen–Jongliermuster
- *wer* wirft *wann wie* hoch zu *wem*?
- simulieren und anzeigen
- vorher: beschreiben

Das ist Simulation und Spezifikation paralleler Prozesse

- Programmier–Sprache

```
<3p 2 3p 3 3p 2 3 3p 3p 2 3 3p 3 |
 4p 4p 2 3 4p 3 4p 2 4p 3 4p 2 3 >
```

- Diagramme



Beispiel–Thema: Spiele

Zwei Personen, Nullsumme, vollst. Inf. (Hex, Go)

- Positionsbewertung
- Zugvorschläge
- *lokal*: Spielbaum (generisch)

Monotonie (Steine werden nur gesetzt, bewegen sich nicht):

- sein guter Zug ist mein guter Zug
- spätere Vorschläge vorziehen

global: "gute Form" erkennen:
Wissen darüber erwerben und fixieren

Beispiel–Thema: Musik

Haskore :: Partitur → Midifile (→ Klangerzeuger)

```
data Cluster = Cl SNote [Cluster]
sim :: Pat -> [Cluster]
sim pat = map mkCluster pat
    where mkCluster notes
      = Cl notes (map (mkCluster . addmult notes) pat)
addmult pds iss = zipWith addmult' pds iss
    where addmult' (p,d) (i,s) = (p+i,d*s)
sim3 n = Instr "vibes"
   (Trans 48
      (Tempo 4 1 (simToHask (simFringe n pat3))))
sim4 n  = (Trans 60
      (Tempo 2 1 (simToHask (simFringe n pat4'))))
sim4s n = let s = sim4 n
              l1 = Instr "flute" s
              l2 = Instr "bass"  (Trans (-36) (revM s))
          in l1 :=: l2
```

- Werk und Interpretation trennen
- . . . → Klangbeeinflussung
- Struktur grafisch darstellen und "live" ändern

Grafik und Graphen

- Daten grafisch darstellen (Bäume, Graphen)
- durch *Layout–Kombinatoren* (nebeneinander, untereinander, ineinander, verbunden durch . . .)
- Ausgabe (Rendering) für PostScript, VRML, X11, . . .
- Ausgabe *editierbar*
- Kombinatoren = Regeln in Graph–Grammatiken

Grafik und Musik

Schicht 1	
Schriftsatz (Font)	Klang (Sound)
Buchstabe	Note
Schicht 2	
Box (minipage)	(Teil–)Partitur
Layout– Kombinator	Musik– Kombinator
Interpretation	
Boxen–Anordnung	Phrasierungen
Seiten–Umbruch	Instrumentierung

Eingebettete Sprachen

anwendungsspezifische Sprache
wird eingebettet in Haskell
(nicht interpretiert oder kompiliert)

Vorteile:

- gesamte Ausdruckskraft von Haskell
- deklarative Semantik
- Sprachen sind kombinierbar

Für und Wider

Algorithmus = Prozeß = imperativ
Spezifikation = deklarativ

imperatives Programmieren:

- intuitiv (von Implementation her)
- Exaktheit nur schwer zu erreichen (Hoare–Kalkül, temporale Logiken)

deklaratives (funktionales) Progr.:

- intuitiv (von Spezifikation her)
- Korrektheitsbeweise einfach (Equational Reasoning)
- muß ins Imperative übersetzt werden (Kompiler, Laufzeitsystem)
- Resourcenverbrauch schwer zu beschreiben

Andere Lehrveranstaltungen

- Lambda–Kalkül (Herre): Reduktion
- Semantik (Gerber, Hartwig): Domains, Typen
- Term–Ersetzung (Hartwig): Reduktion, Termalgebra
- Funktionale Programmierung (Gerber):
  - Implementierung, Kompilation (Graph–Reduktion, abstrakte Maschinen)
  - Parallelisierung
  - Funktional–Logische Sprachen