Softwaretechnik

Funktionale Programmierung

Christian Lindig

Lehrstuhl für Softwaretechnik
Universität des Saarlandes

23. Januar 2006

Quicksort in Java

```
static void sort(int a[], int lo0, int hi0) {
  int lo = lo0;
  int hi = hi0;
  if (lo >= hi) { return; }
  int mid = a[(lo + hi) / 2];
  while (lo < hi) {
    while (lo<hi && a[lo] < mid)  { lo++; }
    while (lo<hi && a[hi] >= mid) { hi--; }
    if (lo < hi) {int T=a[lo]; a[lo]=a[hi]; a[hi]=T;}
  }
  if (hi < lo) { int T=hi; hi=lo; lo=T;}
  sort(a, lo0, lo);
  sort(a, lo == lo0 ? lo+1 : lo, hi0);
}
```

Quicksort in Haskell

```
qsort []     = []
qsort (x:xs) = qsort [y | y <- xs, y <= x] ++ [x] ++
               qsort [y | y <- xs, y >  x]
```

Quicksort in Haskell

```
qsort []     = []
qsort (x:xs) = qsort [y | y <- xs, y <= x] ++ [x] ++
               qsort [y | y <- xs, y >  x]
```

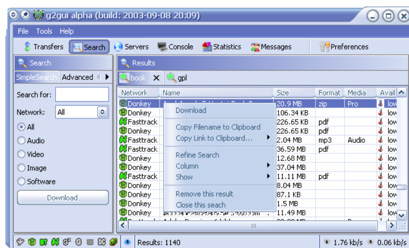
- Definition von `qsort` ist *deklarativ*.
- *Pattern-Matching* vereinfacht die Fallunterscheidung.
- `qsort` ist *polymorph*:
  `qsort [3,6,7,3,9,6,1]` $\Longrightarrow$ `[1,3,3,6,6,7,9]`
  `qsort [('h',5),('b',2),('b',1)]` $\Longrightarrow$ `[('b',1),('b',2),('h',5)]`
- `qsort` besitzt einen Typ:
  `qsort :: Ord a => [a] -> [a]`
- Auf den zu sortierenden Elementen muss eine Ordnung definiert sein.

Anwendungen funktionaler Konzepte

- Spreadsheets: der Wert einer Zelle ist (1) eine Konstante oder (2) eine Funktion über den Wert anderer Zellen.
- C++ Templates bilden eine funktionale Programmiersprache.
- XSLT ist eine funktionale Sprache zur Transformation von XML-Dokumenten.
- XQuery ist eine Anfrage-Sprache (ähnlich SQL) zur Extraktion von Daten aus XML-Dokumenten.

In FP implementierte Anwendungen

- MLdonkey – eDonkey File-Sharing



- Unison – zur Synchronization von Dateien zwischen Laptop und Desktop
- Ericsson Telefonanlagen
- DARCS – dezentrale Versionskontrollsystem
- Compiler: SML/NJ, GHC, OCaml, Skala, XDuce, Galax, . . .

Haskell Syntax

www.haskell.org

`(1, 3.1415, 'c')`	Tripel mit Literalen
`[1, 4, 6]`	Liste mit Literalen
`1:4:6:[]`	Liste mit `(:)` Operator
`length [1,2,3]`	Funktionsaufruf
`max 2 x`	zwei Argumente
`max 2 (x*x)`	zwei Argumente
`2 == x`	Gleichheits-Operator ==
``2 `max` x``	`max` als Infix-Operator
`(==) 2 x`	== als (Prefix-)Funktion
`double x = 2*x`	Funktionsdefinition
`[ x | x <- [1..], even x]`	*list comprehension*

Rekursion

Das Fehlen von Variablen erzwingt es, Iteration durch Rekursion auszudrücken.

```
length :: [a] -> Int
length []         = 0
length (x:xs)     = 1 + len xs

take :: Int -> [a] -> [a]
take _ []         = []
take 0 xs         = []
take n (x:xs)     = x : take (n-1) xs
```

Fortgeschrittene Programmiertechniken verlagern Rekursion in spezielle Kombinatoren (`foldl`), so dass Rekursion nicht explizit ist:

```
length            = foldl (\n _ -> n + 1) 0
```

Funktionen sind Werte

Funktionen als Parameter

In Haskell (oder ML, Scheme, ...) ist eine Funktion ein Wert, der genauso wie ein Integer oder String übergeben werden kann. Hier ist p ein Prädikat:

```
filter :: (a -> Bool) -> [a] -> [a]
filter p []       = []
filter p (x:xs) = if p x then x : filter p xs
                         else filter p xs
```

filter even [1..10] $\Longrightarrow$ [2,4,6,8,10]

Genauso, wie man ein Integer 4 verwenden kann, ohne ihn zu benennen, kann man eine anonyme Funktion verwenden:

filter (\x -> x `mod` 2 == 0) [1..10] $\Longrightarrow$ [2,4,6,8,10]

Funktionen sind Werte

Funktionen als Ergebnis

Wird eine Funktion *unterversorgt*, entsteht eine Funktion als Ergebnis:

```
filter_even = filter even
```

Der Typ der neuen Funktion kann leicht aus der unterversorgten Funktion abgeleitet werden:

```
filter      :: (a -> Bool) -> [a] -> [a]
even        :: Int -> Bool
filter_even :: [Int] -> [Int]
```

Unterversorgung kann als Spezialisierung allgemeiner (also wiederverwendbarer) Funktionen aufgefasst werden.

Typinferenz

Moderne funktionale Programmiersprachen besitzen ein *statisches* Typsystem. Jeder Name und jeder Teil-Ausdruck besitzt einen Typ, der zur *Übersetzungszeit* bestimmt werden kann.
Mit Ausnahme von Datentypen müssen Typen nicht deklariert werden. Ausschnitt aus einer interaktiven Sitzung mit Hugs, einem Haskell-Interpreter:

```
Main> :t filter even
filter even :: Integral a => [a] -> [a]

Main> :t (\f g x -> f (g x))
\f g x -> f (g x) :: (a -> b) -> (c -> a) -> c -> b
```

Datentypen

Summen- und Produkttypen

Produkttyp $\equiv$ Record, Tupel – enthält alle Daten gleichzeitig
Summentyp $\equiv$ varianter Record – enthält eine Alternative

```
data Time     = Time (Int, Int)
data Shape    = Circle | Triangle | Square | Pentagon
```

Datentypen können parametrisiert werden:

```
data Result a = Ok a | Wrong

apply f (Ok a) = Ok (f a)
apply f Wrong  = Wrong
```

Datentypen und Pattern-Matching

Primitive und neu definierte Datentypen können mit Pattern-Matching zerlegt werden:

```
data Tree a = Empty | Node (Tree a) a (Tree a)

height :: Tree a -> Int
height Empty                = 0
height (Node left _ right) = 1 + max (height left)
                                     (height right)
```

Rekursive Funktionen sind die Entsprechung zu rekursiven Datentypen.

Persistenz

Da es keine Seiteneffekte gibt, berechnen eine Funktion (konzeptionell) immer eine neue Datenstruktur.

```
insert x Empty   = Node Empty x Empty
insert x (Node left y right)
    | x == y     = Node left y right
    | x <  y     = Node (insert x left) y right
    | otherwise = Node left y (insert x right)
```

```
x = Empty
y = insert "hallo" x
```

Das Einfügen von `"hallo"` in `x` verändert `x` nicht:

$$\texttt{height x} \Longrightarrow 0$$

Polymorphismus

Funktionen und Datenstrukturen in FP sind oft polymorph: sie funktionieren uniform für *alle* Datentypen.

- `length [1,2]` $\Longrightarrow 2$
- `length [(1,2),(2,3),(4,5)]` $\Longrightarrow 3$

Der Polymorphismus einer Funktion oder eines Datentyps spiegelt sich in seinem Typ wieder, der *Typvariablen* enthält:

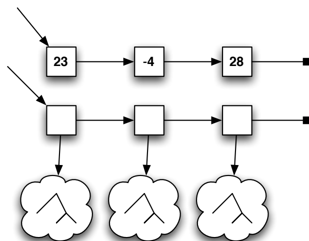
- `length :: [a] -> Int`

```
data Tree a = Empty | Node (Tree a) a (Tree a)
```

Der Polymorphismmus in FP wird deswegen *parametrischer Polymorphismus* genannt.

Implementierung von Polymorphismus

- Daten werden in FP intern uniform repräsentiert – deswegen existiert eine polymorphe Funktion im Maschinencode nur einmal.
- Eine polymorphe Funktion ignoriert bestimmte Teile einer Datenstruktur – diese Teile werden durch Typvariablen beschrieben.
- Beispiel: `length` ignoriert den Inhalt einer Liste.



Grenzen von Polymorphismus

```
qsort []     = []
qsort (x:xs) = qsort [y | y <- xs, y <= x] ++ [x] ++
               qsort [y | y <- xs, y >  x]
```

- `(++)` ist polymorph: `[a] -> [a] -> [a]`
- `<=` und `>` können nicht polymorph sein: sie hängen vom Typ der zu sortierenden Liste ab – es kann keine einheitliche Implementierung für `(<=)` existieren.

Wir benötigen eine typ-spezifische Implementierung für `(<=)`. Dies wäre kein Problem in Java, aber in Haskell oder ML?

Polymorphismus

Typ-Klassen und Ad-Hoc Polymorphismus

Haskell erlaubt es, eine typ-spezifische Implementierung für Funktionen wie `(==)` oder `(<=)` anzugeben. Der Typ von `qsort` zeigt an, dass wir nur solche Liste sortiert werden können, für die eine Ordnung definiert ist:

```
qsort :: Ord a => [a] -> [a]
```

Die Ordung `(<=)` ist typ-spezifisch definiert; ähnlich wie in OOP wird die richtige Implementierung zur Laufzeit ausgewählt.
Diese Form von Polymorphismus heißt *Ad-Hoc Polymorphismus* oder *Überladung*.

Vergleich: Polymorphismus in OO

Eine Variable vom Typ A kann auch jeden Wert vom Typ $B \leq A$ aufnehmen. Dabei werden allerdings alle über A hinausgehenden Eigenschaften "'vergessen"':

```
Integer i = new Integer(123);   // i.intValue() == 123
Object  o = new Integer(123);
System.out.println(o.intValue); // intValue unbekannt
```

Subtyping-Polymorphismus erlaubt "'inhomogene"' Container (Listen, Maps, etc.), erfordert allerdings Typ-Konvertierungen zur Laufzeit.

```
Vector v = new Vector();
v.addElement(new Integer(1));   // addElement(Object o)
v.addElement(new String("two"));
Integer x = (Integer) v.get(0); // Object get(int)
String  y = (String)  v.get(1);
```

Dynamische Bindung in Java

```
public class EvenOdd {
    public EvenOdd () {}
    public boolean even(int n) {
        if      (n == 0) { return true  ;}
        else if (n == 1) { return false ;}
        else { return this.odd(n-1); }
    }
    public boolean odd(int n) {
        if      (n == 0) { return false ;}
        else if (n == 1) { return true  ;}
        else { return this.even(n-1); }
    }
}
public class FastEvenOdd extends EvenOdd {
    public FastEvenOdd() {}
    public boolean even(int n) {
        return (n%2 == 0);
    }
} // FastEvenOdd x = new FastEvenOdd(); x.even(1000)
```

Statische Binding in FP

```
even 0 = True
even 1 = False
even n = odd (n-1)

odd 0  = False
odd 1  = True
odd n  = even (n-1)
```

Die Definitionen von `even` und `odd` können nachträglich nicht mehr ergänzt werden. Eine dynamischer Bindung vergleichbare Ausdruckskraft entsteht durch Funktionen höherer Ordnung. Diese müssen aber geplant werden.

```
map f []     = []
map f (x:xs) = f x : map f xs
```

Evolution von Software

Klassen, Module und Polymorphismus sind nicht unbedingt erforderlich für Software, die einmalig entwickelt und dann eingesetzt wird.
Diese Konzepte aber sind wichtig für Software, die

- erweiterbar bleiben soll;
- als wiederverwendbare Basis als Teil größerer Projekte dienen soll.

Programmiersprachen wie Java, C++, ML und Haskell unterstützen diese Konzepte syntaktisch, während sie in Programmiersprachen wie C oder Pascal nur durch Konventionen simuliert werden können.

Sind OO-Systeme einfacher erweiterbar?

Betrachte *Datenstruktur* mit *Operationen*;
$\oplus$: einfach, $\ominus$: aufwändig

	FP	**OO**
	Eine Funktion pro Operation, ein Summand per Alternative	Eine Klasse pro Alternative, eine Methode pro Operation
Alternative hinzufügen	In allen Funktionen: Pattern-Matching ergänzen $\ominus$	Eine Klasse hinzufügen $\oplus$
Operation hinzufügen	Eine Funktion hinzufügen $\oplus$	Eine Methode in allen Klassen hinzufügen $\ominus$

OO oder FP?

Wenn Datenstrukturen häufiger ergänzt werden als Operationen, ist eine OO Modellierung einfacher:

Daten	≡	*GUI-Widgets (Button, Menu, Radio, ...)*
Operationen	≡	*fest definierter Satz von Methoden wie* `draw()`, `select()`.

Andernfalls ist eine funktionale Modellierung einfacher:

Daten	≡	*abstrakter Syntax-Baum*
Operationen	≡	`eval()`, `type_check()`, `transform()`, ...

OO oder FP?

Einfacher in FP und mit Modulen:

- Handhabung von Baumstrukturen (Pattern-Matching)
- Parametrisierung-im-Großen (ML Funktoren, Typklassen)
- Sicherheit durch Typsystem

Einfacher mit OO:

- Kapselung von Zustand (in Objekten)
- Inkrementelle Programmentwicklung (Vererbung)
- Flexibilität auf Kosten von Sicherheit

Diverse Programmiertechniken helfen, den Schwächen eines gewählten Paradigmas zu begegnen: Design-Patterns, Continuations, Monaden.

Ausblick

In modernen Sprachen verschwimmen die Grenzen von FP und OO.

- OO-Sprachen übernehmen FP-Konzepte: Pyhton und Lua kennen anonyme Funktionen, Java hat innere Klassen, Generics führen parametrischen Polymorphismus ein.
- Objective Caml hat Klassen und Objekte.
- In Scala ist jede Funktion zugleich ein Objekt.
- Themen in der Forschung: generalisierte Summentypen, Nebenläufigkeit, Verteilung, vertrauenswürdiger Binärcode, Integration von XML als Datentyp, *staged programming*.