Paradigmen
3. funktionales Programmieren

Inhalt

Geschichte

Anfang: Lisp (1957, rekursive Funktionen und Listen, Interpretation)
Abweg: APL (Matritzen und Kryptologie)
Wiederbelebung: ML (höhere Funktionen, polymorphe Typen, Moduln)
Entfaltung: FP, Clean, Hope, Miranda (*pattern matching*, *lazy evaluation*)
Konsolidierung: Haskell (Typklassen, Konstruktorklassen, Monaden)

Literatur

Alonzo Church: *Lambda-Kalkül*. 193X
Peter Landin: *The next 700 Programming Languages*. 1966
John Backus: *Can Programming be Liberated from the von-Neumann Style?* 1978

Ausgangspunkt

state variables considered harmful!
verifiable programming

Kernstücke

- *Variablen* sind mathematisch, d.h. nicht überschreibbar
- *Datentypen* sind rekursiv und algebraisch (Listen, Bäume)
- *Funktionen* sind erstklassig (Argument, Resultat, Curry, Kombinator)

Nützliche Ergänzungen

- *referential transparency* (das Reinheitsgebot)
- *polymorphe Typisierung* für sichere und mächtige Funktionen
- *pattern matching* für einfache Fallunterscheidung
- *lazy evaluation* für "unendliche" Datenstukturen (Zyklen)

Funktionen ohne Zustandsvariablen

Fakultät rekursiv (ML)

```
fun fac (n) = if n > 0 then n * fac (n - 1) else 1
```

Auswertung durch Einsetzen der Definition

```
  fac (12)
= let n1 = 12 in if n1>0 then n1*fac (n1-1) else 1
= 12*fac (11)
= 12* let n2 = 11 in if n2>0 then n2 * fac (n2-1) else 1
= 12*11* fac (10)
= let n3 = 10 in if n3>0 then n3 * fac (n3-1) else 1
= 12*11*10* fac (9)
...
= 12*11*10*9*8*7*6*5*4*3*2*1
```


Funktionen mit Zustandsvariablen

```
function factorial (in n: Integer) : Integer;
    f : Integer := 1;
  begin
  while n > 0
  loop
    f := f * n;
    n := n - 1
  end loop;
  return f;
  end factorial;
```

Vergleich: imperativ / funktional

Schleife / Rekursion
2 Zustandsvariablen / 12 Konstanten

Algebraische Typen und *pattern matching*

algebraische Typen

```
data tree ::= Empty
            | Node tree int tree
```

Funktionssdefinition mit *pattern matching* (Haskell)

```
insert :: int->tree->tree
insert new Empty = Node Empty new Empty
insert new (Node left old right)
  = Node (insert new left) old right, if new <= old
  = Node left old (insert new right), otherwise
```


Notiz: pattern

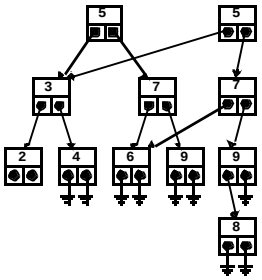
pattern sind Ausdrücke über *Konstruktoren*, z. B. `circle r`
die Gleichung ist nur anwendbar, wenn ihr Argument auf das *pattern paßt*
die Gleichungen werden (z. B.) von oben nach unten geprüft
für nicht vorkommende Argumente ist die Funktion *undefiniert* (Fehler)
die Variablen des *pattern* (wie `r`) werden beim *matchen*
  an Komponenten des Arguments *gebunden*
  und werden auf der rechten Seite *benutzt*

Vorteile von *pattern matching*

jede Gleichung ist eine wahre Aussage über die Funktion
Selektorfunktionen und Variantentest werden nicht (immer) benötigt:

Notiz: Speicherbedarf

können nicht *selektiv überschieben* werden
wegen der referenziellen Transparenz
kann Kopieren durch *Zeiger-Kopieren* realisiert werden (*sharing*)



Notiz: insert imperativ

```
type Tree is access to record
                entry: Integer;
                left, right: Tree;
             end reord;
procedure insert (in i: Integer; inout t:Tree) is
  var s, b: Tree;
begin s:= t; b := null;
while s <> null andthen s.val <> i
loop if i < s.val then s:= s.left; s.left:= b; b:= s;
else s:= s.right; s.right:= b; b:= s; b
end loop;
if s = null then s:= Tree(i, null, null)
end if;
end insert;
```


Listen

DIE zusammengesetzte Datenstruktur funktionaler Sprachen...

```
list ? ::= Nil | Cons ? (list ?)
```

syntaktischer Zucker

```
[ ? ]   = list ?
[]      = Nil
:       = Cons
[a,b,c] = a:b:c:[]
```

vordefinierte Operationen

```
hd      :: [?]->?                    Selektorfunktionen
tl      :: [?]->[?]                  (partiell)
length  :: [?]->Integral             Länge einer Liste
++      :: [?]->[?]->[?]             Listenverkettung
```


Kombinatoren

Zahlenlisten

```
through :: (int,int)->[int]
through (m,n) = [],                    if m > n
              = m: (through (m+1,n)),  otherwise
product :: [int]->int
product  []  = 1
product (h:t) = h ? (product t)
```

Programmieren mit Kombinatoren

```
fac:: int->int
fac (n) = product (through (1,n))
```


Currying

Motto

Lieber *einstellige Funktionale* (die Funktionen liefern)
statt *mehrstelliger Funktionen*

F: *X* ? (*Y* ? *Z*) ? *F*: *X* ? *Y* ? *Z*
F(*x*,*y*) = (*F*'(*x*)) (*y*) (vom Typ *Z*, und *F*'(*x*) :: *Y* ? *Z*)

Beispiel (map)

```
map'':: ((a->b),[a])->[b]
map'(f,[])    = []
map'(f, h:t)  = f(h) :map'(f, t)

map:: (a->b)->[a]->[a]
map f []    = []
map f (h:t) = f h : (map f t)
```


Notiz: partielle Parametrisierung ist nützlich

```
evens :: [Integral]->[Integral]
evens =  filter (\x ->x mod 2 = 0)
```


weitere Listenkombinatoren

homomorphe Erweiterungen

```
foldl :: (?->? ->?)->? ->[?]->?
foldl op one [] = one
foldl op one (h:t) = op h (reduce op one t)

sum,product :: [Integral]->Integral
sum     = foldl (+)0
product = foldl (*) 1
```


Verkettung und Umkehrung von Listen

```
++ :: [?]->[?]->[?]
[] ++ l = l
(h:t) ++ l = h:(t++l)
reverse :: [?]->[?]
reverse [] = []
reverse (h:t) = (reverse t) ++ [h]

implode:: [[?]]->[?]
implode = reduce ((++),[])
```

effiziente Listenverkettung

Listen sind Funktionen rest -> Liste ++ rest

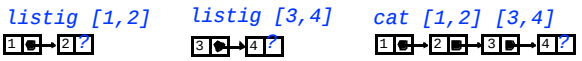
```
type funlist ? = [?]->[??
listig :: [?]-> funlist ?
listig l = \t -> l ++ t

cat :: (funlist ?) -> (funlist ?) -> (funlist ?)
cat l l' = l . l'

rev :: [a]->[a]->[a]
rev [] = id
rev (h:t) = (rev t) . (\l -> h:l)
```


Notiz: Beispiel für Auswertung

```
  cat(listig [1,2]) (listig [3,4]) []
= (listig [1,2]) . (listig [3,4]) []
= (\x ->[1, 2] ++ x) . (\y -> [3, 4] ++ y) []
= (\z -> (\x ->[1, 2] ++ x) (\y -> [3, 4] ++ y) z)[]
= (\x ->[1, 2] ++ x) (\y -> [3, 4] ++ y) []
= (\x ->[1, 2] ++ x) ([3, 4] ++ [])
= [1,2] ++ [3,4] ++ []
```



Notiz: Beispiel für Auswertung

```
  rev [1,2,3] []
= (rev [2,3]) . (\x -> 1:x) []
= (rev [3]) . (\y -> 2:y) . (\x -> 1:x) []
= (rev []) . (\z -> 3:z) . (\y => 2:y) . (\x => 1:x) []
= id . (\z => 3:z) . (\y => 2:y) . (\x => 1:x) []
= id ((\z => 3:z) ((\y => 2:y) ((\x => 1:x) [])))
= id ((\z => 3:z) ((\y => 2:y) (1:[])))
= id ((\z => 3:z) (2:1:[])
= id (3:2:1:[])
= 3:2:1:[]
= [3,2,1]
```


verzögerte Auswertung (*lazy evaluation*)

```
from n = n : from (n+1)

firstprime (h:t)= h,if prime h
  =firstprime t,otherwise

firstprime (from 1000)
```

Erinnerung

Bei *lazy evaluation*
- werden Parameter *unausgewertet* an Funktionen übergeben,
- und in ihrem Rumpf nur *bei Bedarf* ausgewertet (aber dann nur *einmal*)

Konsequenzen
- nicht benötigte Parameter “kosten nichts”
- man kann unendliche rekursive Werte “denotieren”

Listenumschreibungen

mengentheoretische Schreibweise für Listen

z. B. die geraden Zahlen von 1 bis 1000

```
[ n | n <- 1..1000; n mod 2 = 0 ]
```

hierbei ist
- `n <- 1..1000` ein *Generator* (entsprechend *through*)
- `n mod 2 = 0` ein *Prädikat* oder *Filter*

Vergleiche die Ähnlichkeit zur Mengenschreibweise

{ n |n ? {1..1000}, n mod 2 = 0 }

jedoch Vorsicht!
- Listen können Wiederholungen enthalten
- die Reihenfolge der Elemente ist relevant

Beispiel Quicksort

```
sort :: [?]->[?]
sort []    =  []
sort (h:t) =     sort [ i | i <- t; i <  h]
              ++ [h]
              ++ sort [ i | i <- t; i >= h]
```

Zwischenergebnisse können automatisch weg-transformiert werden

Nebenläufigkeit

harmoniert gut mit funktionalen Programmen
führt nicht zu Nichtdeterminismus,
- sondern (hoffentlich) zu mehr Effizienz

Paralelisierung bei Listen wird durch *pattern matching* eher behindert
- (besser wäre reine Bearbeitung mit Kombinatoren)

Praktikum: einige funktionale Sprachen

eine konzeptionelle Übersicht

Konzepte	Lisp	ML	Miranda	Haskell
Zuweisung	+	o	–	–
Funktionen erstklassig	o	+	+	+
algebraische Typen	–	+	+	+
Typisierung	–	+	+	+
Polymorphie, Inferenz	–	+	+	+
Klassen, Überladen	–	o	-	+
pattern matching	–	+	+	+
lazy evaluation	–	–	+	+
Moduln	–	+	+	+