

6. Funktionen als Daten, Übersicht

Orthogonales Typsystem: Funktionen sind beliebig mit anderen Typen kombinierbar

Notation für Funktionswerte (Lambda-Ausdruck):

```
fn (z,k) => z*k
```

Datenstrukturen mit Funktionen als Komponenten:

z. B. Suchbaum für Funktionen

Funktionale, Funktionen höherer Ordnung (higher order functions, HOF):

haben **Funktionen als Parameter oder als Ergebnis**

Berechnungsschemata:

Funktion als Parameter abstrahiert Operation im Schema, wird bei Aufruf des Schemas konkretisiert

```
foldl (fn (z,k) => z*k, [2,5,1], 1);
```

 (hier noch ohne Currying)

Schrittweise Parametrisierung (Currying):

Funktion als Ergebnis bindet ersten Parameter, nützliche Programmieretechnik, steigert Wiederverwendbarkeit

```
val chorner = fn l => fn x => foldl (fn (z,k) => z*x+k, 1, 0);
```

nicht-endliche Datenstrukturen (Ströme, lazy evaluation), (Kapitel 7):

Funktionen als Komponenten von Datenstrukturen, z. B. Funktion, die den Rest einer Liste liefert

```
datatype 'a seq = Nil | Cons of 'a * (unit -> 'a seq)
```

Notation von Lambda-Ausdrücken

Auswertung eines Lambda-Ausdruckes

liefert eine Funktion, als Datum, unbenannt.

Notation:

```
fn ParameterMuster => Ausdruck
```

```
fn (z,k) => z*k
```

```
fn (x, y) => Math.sqrt (x*x + y*y)
```

mit Fallunterscheidung:

```
fn Muster1 => Ausdruck1
```

```
| Muster2 => Ausdruck2
```

```
| ...
```

```
| Mustern => Ausdruckn
```

```
fn nil => true
```

```
| (_::_) => false
```

Anwendungen von Lambda-Ausdrücken:

```
linsert (1, fn (z,k) => z*x+k, 0)
```

```
(fn (z,k) => z*k) (a, b)
```

```
if b then fn (z,k) => z*k
```

```
else fn (z,k) => z+k
```

```
[fn (z,k) => z*k, fn (z,k) => z+k]
```

```
val null = fn nil => true
           | (_::_) => false;
```

```
fun Comp (f, g) = fn x => f (g x);
```

Currying

Haskell B. Curry: US-amerikanischer Logiker 1900-1982, Combinatory Logic (1958);

Moses Schönfinkel, ukrainischer Logiker, hat die Idee schon 1924 publiziert:

Funktionen **schrittweise parametrisieren statt vollständig mit einem Parametertupel.**

abstraktes Prinzip für eine n-stellige Funktion:

Tupelform:

Signatur: $gn: ('t_1 * 't_2 * \dots * 't_n) \rightarrow 'r$

Funktion: $fun\ gn\ (p_1, p_2, \dots, p_n) = \text{Ausdruck über } p_1, \dots, p_n$

Aufrufe: $gn\ (a_1, a_2, \dots, a_n)$ liefert Wert vom Typ 'r

ge-curried:

Signatur: $cgn: 't_1 \rightarrow ('t_2 \rightarrow \dots \rightarrow ('t_n \rightarrow 'r) \dots)$

Funktion: $fun\ cgn\ p_1\ p_2 \dots p_n = \text{Ausdruck über } p_1, \dots, p_n$

Aufruf: liefert Wert vom Typ

```
(cgn a1 a2 ... an) 'r
```

```
(cgn a1 a2 ... an-1) 'tn->'r
```

```
...
```

```
(cgn a1) 't2->(...('tn->'r)...) 'r
```

Ergebnisfunktionen tragen die schon gebundenen Parameter in sich.

Funktion voll-parametrisiert entwerfen - teil-parametrisiert benutzen!

Currying: Beispiele

Parametertupel:

```
fun prefix (pre, post) = pre ^ post;
```

Signatur: $string * string \rightarrow string$

ge-curried:

```
lang: fun prefix pre = fn post => pre ^ post;
```

```
kurz: fun prefix pre post = pre ^ post;
```

Signatur: $string \rightarrow (string \rightarrow string)$

gleich: $string \rightarrow string \rightarrow string$

Erster Parameter (**pre**) ist in der Ergebnisfunktion gebunden.

Anwendungen:

```
val knightify = prefix "Sir ";
```

```
val dukify = prefix "The Duke of ";
```

```
knightify "Ratcliff";
```

```
(prefix "Sir ") "Ratcliff";
```

```
prefix "Sir " "Ratcliff";
```

 linksassoziativ

auch rekursiv: **x** oder **n** ist in der Ergebnisfunktion gebunden

```
fun repxlist x n = if n=0 then [] else x :: repxlist x (n-1);
```

```
fun repnlist n x = if n=0 then [] else x :: repnlist (n-1) x;
```

```
(repxlist 7); (repnlist 3);
```

Funktionen in Datenstrukturen

Liste von Funktionen:

```
val titlefns =
  [prefix "Sir ",
   prefix "The Duke of ",
   prefix "Lord "]      :(string -> string) list
```

```
hd (tl titlefns) "Gloucester";
```

Suchbaum mit (string * (real -> real)) Paaren:

```
val fntree =
  Dict.insert
    (Dict.insert
      (Dict.insert
        (Lf, "sin", Math.sin),
         "cos", Math.cos),
       "atan", Math.atan);

Dict.lookup (fntree, "cos") 0.0;
```

Currying als Funktional

Funktional: Funktionen über Funktionen; Funktionen höherer Ordnung (HOF)

`secl, secr` (section):

2-stellige Funktion in Curry-Form wandeln; dabei den linken, rechten **Operanden binden**:

```
fun secl x f y = f (x, y);
'a -> ('a * 'b -> 'c) -> 'b -> 'c
```

```
fun secr f y x = f (x, y);
('a * 'b -> 'c) -> 'b -> 'a -> 'c
```

Anwendungen:

```
fun power (x, k):real =if k = 1 then x else
                        if k mod 2 = 0then   power (x*x, k div 2)
                        else x *power (x*x, k div 2);
```

```
val twoPow = secl 2.0 power;          int -> real
```

```
val pow3 = secr power 3;             real -> real
```

```
map (1, secr power 3);
```

```
val knightify = (secl "Sir " op^);    string -> string
```

`op^` bedeutet infix-Operator `^` als Funktion

Komposition von Funktionen

Funktional `cmp` verknüpft Funktionen `f` und `g` zu deren Hintereinanderausführung:

```
fun cmp (f, g) x = (f (g x));
```

Ausdrücke mit **Operatoren**, die Funktionen zu neuen **Funktionen verknüpfen**,
2-stelliger **Operator** `o` statt 2-stelliger Funktion `cmp`:

```
infix o;
fun (f o g) x = f (g x);      ('b->'c) * ('a->'b) -> 'a -> 'c
```

Funktionen nicht durch **Verknüpfung von Parametern** in Lambda-Ausdrücken definieren:

```
fn x => 2.0 / (x - 1.0)
```

sondern durch **Verknüpfung von Funktionen** (algebraisch) berechnen:

```
(secl 2.0 op/) o (secr op- 1.0)
```

Potenzieren von Funktionen $f^n(x)$:

```
fun repeat f n x = if n > 0 then repeat f (n-1) (f x) else x;
repeat: ('a->'a) -> int -> 'a -> 'a
```

Aufrufe:

```
(repeat (secr op/ 2.0) 3 800.0);      (repeat tl 3 [1,2,3,4]);
(repeat (secr op/ 2.0) 3);           (repeat tl 3);
(repeat (secr op/ 2.0));             (repeat tl);
```

[John Backus: Can Programming Be Liberated from the von Neumann Style? A functional Style and Its Algebra of Programs; 1977 ACM Turing Award Lecture; CACM, vol. 21, no. 8, 1978]

Kombinatoren

Kombinator: Funktion ohne freie Variable

Kombinatorischer Term T:

T ist ein Kombinator oder T hat die Form (T1, T2) und Ti sind kombinatorische Terme

Kombinatorische Terme dienen

zur **Verknüpfung** und zu algebraischer **Transformation** von Funktionen,
zur Analyse und zum **Beweis** von Programmen

David Turner (britischer Informatiker) hat 1976 gezeigt, dass **alle Funktionen des Lambda-Kalküls** durch die klassischen Kombinatoren `S` und `K` darstellbar sind.

klassische Kombinatoren S K I:

```
fun I x = x;          Identitätsfunktion          'a -> 'a
fun K x y = x;       bildet Konstante Fkt.      'a -> 'b -> 'a
fun S x y z = x z (y z); wendet x auf y an, nach Einsetzen von z in beide
('a -> 'b -> 'c) -> ('a -> 'b) -> 'a -> 'c
```

I entspricht `S K K` denn $((S K K) u) = (S K K u) = (K u (K u)) = u$

Beispiel:

Der Lambda-Ausdruck $(\lambda x (\lambda y (x y)))$
kann in $(S (K (S I)) (S (K K) I))$ transformiert werden.

Reihenberechnung als Schema

Allgemeine Formel für eine **endliche Reihe**: $\sum_{i=0}^{m-1} f(i)$

```
fun summation f m =
  let fun sum (i, z):real =          akkumulierende Hilfsfunktion
        if i=m then z else sum (i+1, z + (f i))
      in sum (0, 0.0) end;
```

Signatur: (int->real) -> int -> real

Aufruf `summation (fn k => real(k*k)) 5;` liefert 30

Doppelsumme:

$$\sum_{i=0}^{m-1} \sum_{j=0}^{n-1} g(i,j)$$

summation als Parameter von summation:

```

      Bindung
      |
      v
summation(fn i => summation (fn j => g(i,j)) n) m
int->real          int->real
```

einfacher `h i j` statt `g(i,j)`: `summation (fn i => summation (h i) n) m;`

Kombination von Funktionen, Konversion nach `real`: `summation (Math.sqrt o real);`

Summe konstanter Werte; Kombinator `K` für konstante Funktion: `summation (K 7.0) 10;`

Funktionale für Listen: map

Liste elementweise mit einer Funktion abbilden:

```
map f [x1,...,xn] = [f x1,..., f xn]
```

```
fun map f nil = nil
  | map f (x::xs) = (f x) :: map f xs;
Signatur: ('a -> 'b) -> 'a list -> 'b list
```

Anwendungen:

```
map size ["Hello", "World!"];
map (secl 1.0 op/) [0.1, 1.0, 5.0];
```

für 2-stufige Listen (setzt `map` in Curry-Form voraus!):

```
map (map double) [[1], [2, 3]];
```

statt `map f (map g l)` besser `map (f o g) l`

Matrix transponieren:

```
fun transp (nil::_) = nil
  | transp rows =
    map hd rows :: transp (map tl rows);
```

Funktionale für Listen: Filter

Schema: Prädikatfunktion wählt Listenelemente aus:

```
fun filter pred nil = nil
  | filter pred (x::xs) = if pred x then x :: (filter pred xs)
                        else (filter pred xs);
```

Anwendungen:

```
filter (fn a => (size a) > 3) ["Good", "bye", "world"];
fun isDivisorOf n d = (n mod d) = 0;
filter (isDivisorOf 360) [24, 25, 30];
```

Mengendurchschnitt (`mem` ist auf nächster Folie definiert):

```
fun intersect xs ys = filter (secl (op mem) ys) xs;
```

Variationen des Filterschemas:

```
val select = filter;
fun reject f = filter ((op not) o f);
```

```
fun takewhile pred nil = nil
  | takewhile pred (x::xs) = if pred x then x::(takewhile pred xs)
                            else nil;

takewhile isPos [3, 2, 1, 0, ~1, 0, 1];
fun dropwhile ... entsprechend
```

Funktionale für Listen: Quantoren

Existenz und All-Quantor:

```
fun exists pred nil = false
  | exists pred (x::xs) = (pred x) orelse (exists pred xs);
fun all pred nil = true
  | all pred (x::xs) = (pred x) andalso (all pred xs);
```

Member-Operator:

```
infix mem;
fun x mem xs = exists (secl op= x) xs;
```

Disjunkte Listen?

```
fun disjoint xs ys = all (fn x => all (fn y => y<>x) ys) xs;
oder:
fun disjoint xs ys = all (fn x => (all (secl op<> x) ys)) xs;
```

Quantoren-Funktionale für Listen von Listen:

```
exists (exists pred)          z.B. exists (exists (secl 0 op=))
filter (exists pred)          z.B. filter (exists (secl 0 op=))
takewhile (all pred)          z.B. takewhile (all (secl op> 10))
```

Funktionale verknüpfen Listenwerte

Listenelemente mit 2-stelliger Funktion f verknüpfen:

```
foldl f e [x1, ..., xn] = f(xn, ... f(x2, f(x1, e))...)
```

```
foldr f e [x1, ..., xn] = f(x1, ... f(xn-1, f(xn, e))...)
```

`foldl` verknüpft Elemente sukzessive vom ersten zum letzten.

`foldr` verknüpft Elemente sukzessive vom letzten zum ersten.

```
fun foldl f e nil = e           akk. Parameter
| foldl f e (x::xs) = foldl f (f(x, e)) xs;

fun foldr f e nil = e
| foldr f e (x::xs) = f(x, foldr f e xs);
```

Signatur: ('a * 'b -> 'b) -> 'b -> 'a list -> 'b

Beispiel: `val sum = foldl op+ 0;`

Verknüpfungsreihenfolge bei `foldl` und `foldr`:

```
val difl = foldl op- 0;      difl [1,10]; ergibt 9
val difr = foldr op- 0;      difr [1,10]; ergibt ~9
```

Horner-Schema in Curry-Form:

```
fun horner l x = foldl (fn (h,a) => a*x+h) 0.0 l;
```

Liste umkehren: `fun reverse l = foldl op:: nil l;`

Menge aus Liste erzeugen: `fun setof l = foldr newmem [] l;`
`setof [1,1,2,4,4];`

Werte in binären Bäumen

```
datatype 'a tree = Lf | Br of 'a * 'a tree * 'a tree
```

Schema:

Für jedes Blatt einen Wert e einsetzen und an inneren Knoten Werte mit 3-stelliger Funktion verknüpfen (vergl. `foldr`):

```
fun treefold f e Lf = e
| treefold f e (Br (u,t1,t2)) =
  f (u, treefold f e t1, treefold f e t2);
```

Anwendungen

Anzahl der Knoten:

```
treefold (fn (_, c1, c2) => 1 + c1 + c2) 0 t;
```

Baumtiefe:

```
treefold (fn (_, c1, c2) => 1 + max(c1, c2)) 0 t;
```

Baum spiegeln:

```
treefold (fn (u, t1, t2) => Br (u, t2, t1)) Lf t;
```

Werte als Liste in Preorder (flatten):

```
treefold (fn (u, l1, l2) => [u] @ l1 @ l2) nil t;
```