

11. Funktionen höherer Ordnung

Beispiel: insert und lookup in BST

$(\text{'a} \times \text{'a} \rightarrow \text{bool}) \times \text{'a} \text{ bintree} \times \text{'a} \rightarrow \text{'a} \text{ bintree}$
 funktionaler Typ

Funktionen höherer Ordnung haben Funktionen als Argumente und/oder Ergebnis.
 Funktionen sind **Werte**, „first.class citizens“

11.1 Funktionen als Argumente und Resultate

Beispiel: Anwendung einer Funktion auf alle Elemente einer Liste

```
fun map' (f,nil) = nil
    | map' (f,x::xs) = (f x) :: map' (f,xs)
val l = [1,2,4,8]
map' ((fn x => x+1), l) ↓ [2,3,5,9]
map' ((fn x => x*x), l) ↓ [1,4,16,64]
```

Häufige Verwendung höherer Funktionen:

1. induktiv definiert über den Aufbau einer Datenstruktur
 wendet ihr funktionales Argument auf die Komponenten der Datenstruktur an.
 Beispiel war map' für Listen
2. kaskadierte Funktionen: Funktion bekommt einen Teil der Argumente früher als den Rest
 und produziert als Ergebnis eine Funktion in den restlichen Argumenten

Beispiel:

```
val konstant = fn k => (fn x => k);
```

Das Ergebnis der Anwendung von konstant auf 1 ist die Funktion, der für alle Argumente 1 liefert.

Wie passiert das?

1. Funktionsanwendung konstant 1 bindet den Funktionsparameter k an 1.
 Wertumgebung $\{k \mapsto (val,1)\}$
2. Definierender Ausdruck $(fn x => k)$ wird mit dieser Wertumgebung in einen **Abschluss** (closure) gepackt $((fn x => k), \{k \mapsto (val,1)\})$

11.2 Kaskadierte Funktionen (curried functions)

Die Funktion $\text{map}' : (\text{'a} \rightarrow \text{'b}) * \text{'a list} \rightarrow \text{'b list}$

hat einen Paartyp als Argumenttyp, braucht immer beide Argumente gleichzeitig.

Wenn man map' mit identischem ersten Argument auf viele Listen anwenden möchte, muss man immer wieder das erste Argument angeben.

Alternative ist **Kaskadierung**: „Umwandlung Tupeltyp in Pfeiltyp“

```
fun map f nil = nil
  | map f (x::xs) = f x :: map f xs
```

Allgemein:

gegeben: $f: t_1 \times t_2 \times \dots \times t_n \rightarrow t$
 Kaskadierung: $f': t_1 \rightarrow t_2 \rightarrow \dots \rightarrow t_n \rightarrow t$

f' nimmt Argument a_1 vom Typ t_1 und produziert eine Funktion f_2
 die nimmt Argument a_2 vom Typ t_2 und produziert eine Funktion f_3

...

f'_n nimmt Argument a_n vom Typ t_n und produziert einen Wert vom Typ t
 mit $f' a_1 \dots a_n = f(a_1, \dots, a_n)$

11.3 Reduktion

map lässt die Struktur des Listenarguments unverändert.

Reduktionen berechnen durch Anwendung einer (meist assoziativen binären Operation einen (Nicht-Listen-) Typ.

```
fun auf_add nil = 0
  | auf_add (x::xs) = x + auf_add xs
```

```
fun auf_mul nil = 1
  | auf_mul (x::xs) = x * auf_mul xs
```

Abstraktion der Gemeinsamkeiten:

- Gemeinsame Kontrolle: reduce
- neutrales Element
- binäre Operation

Beispiel:

```
fun reduce (ne, opn, nil) = ne
  | reduce (ne, opn, x::xs) = opn(x, reduce(ne, opn, xs))
```

Nachteil von reduce:

Die ersten beiden Argumente ne, opn werden unverändert durchgereicht.
 Wie kann man das verhindern?

```
fun better_reduce (ne, opn, l) =
  let fun red nil = ne
      | red (x::xs) = opn(x, red xs)
  in red l
  end
```

Bei Anwendung von `better_reduce` wird ein Abschluss für `red` gebildet mit Wertbindung für `ne`, `opn`.

Aber: dieser Abschluss wird erst gebildet, wenn alle drei Argumente gegeben sind.

Phasierung (staging)

Wenn die ersten beiden Argumente gegeben sind, sollten tatsächlich schon Auswertungsschritte passieren. Verwandt mit „partieller Auswertung“.

```
fun phas_reduce (ne, opn) =  
  let fun red nil = ne  
    | red (x::xs) = opn(x, red xs)  
  in red  
end
```

`phas_reduce` konstruiert Abschluss für `red`, sobald die beiden Argumente für `ne`, `opn` gegeben sind.