Programmierung und Modellierung

Funktionen höherer Ordnung

Martin Wirsing

in Zusammenarbeit mit
Moritz Hammer

Inhalt

- Kap. 6 Funktionen höherer Ordnung
  1. Funktionen als Parameter und Wert von Funktionen
  2. Currying
  3. Funktionskomposition
  4. Grundlegende Funktionen höherer Ordnung

6.1 Funktionen als Parameter und Wert von Funktionen

- In SML und anderen funktionalen Programmiersprachen sind Funktionen Werte.
  - Funktionen können wie andere Werte auch als aktuelle Parameter vonvon Funktionen, auftreten
- Funktionen, die als Parameter oder als Wert Funktionen haben, werden Funktionen höherer Ordnung genannt.
- Die Bezeichnung **Funktionen höherer Ordnung** basiert auf folgender Hierarchie:
  - Die Ordnung 0 umfasst die Konstanten.
  - Die Ordnung 1 umfasst die Prozeduren, deren Parameter und Werte Objekte der Ordnung 0 sind.
  - Die Ordnung n + 1 umfasst die Prozeduren, deren Parameter und Werte Objekte der Ordnung n sind.
- Funktionen höherer Ordnung werden manchmal **Funktionale** (engl. Functionals) oder **Operatoren** genannt.

Funktionen als Parameter und Wert von Funktionen

- Funktionen höherer Ordnung stellen eine nützliche Abstraktion dar.
- **Beispiel:**

  Ist `f : real -> real` eine (ableitbare) mathematische Funktion, so kann die Ableitung `f‘` von `f` wie folgt geschätzt werden (wobei Δ eine kleine Zahl ist):

  `f'(x) = (f(x+Δ) - f(x)) / Δ`

- Dies lässt sich wie folgt in SML implementieren:

```
- val delta = 1e~5;
val delta = 1E~5 : real
- fun abl(f) =
    let
      fun f'(x) = (f(x+delta) - f(x)) / delta
    in
        f'
  end;
val abl = fn : (real -> real) -> real -> real
```

Funktionen als Parameter und Wert von Funktionen

- Die Funktion `abl` hat als Parameter eine Funktion `f`.
  - Innerhalb von `abl` wird eine neue Funktion `f'` mit Hilfe von `f` und `delta` definiert.
  - Das Ergebnis von `abl(f)` ist selbst eine Funktion (`f'`), d.h., dass `abl(f)` wiederum auf reelle Zahl angewandt werden kann.
- Der Typausdruck `(real -> real) -> real -> real` bezeichnet wegen der Rechtsassoziativität von „->“ den Typ

  ```
  (real -> real) -> (real -> real):
  ```

  - `abl` hat also als Parameter eine Funktion vom Typ `real -> real` und liefert als Ergebnis ebenfalls eine Funktion vom Typ `real -> real`.

Funktionen als Parameter und Wert von Funktionen

- **Anwendungsbeispiele**:

```
- abl( fn x => x*x )(5.0);
val it = 10.0000099994 : real
- abl( fn x => x*x*x )(5.0);
val it = 75.0001499966 : real
```

  - Ableitung der Funktion `fn x =>` x^2 ist die Funktion `fn x => 2*x` (und `2*5 = 10`).
  - Ableitung der Funktion `fn x =>` x^3 ist die Funktion `fn x => 3*` x^2 (und $3*5^2 = 75$).
- Die mit SML berechneten Zahlen sind **Schätzungen**:
  - Mit anderen Werten für `delta` kann man die Schätzung verbessern.
  - Da die Schätzung somit auch von `delta` abhängt, bietet es sich an, delta zu einem zweiten Parameter von `abl` zu machen:

```
- fun abl(f, delta) = let
        fun f'(x) = (f(x+delta) - f(x)) / delta
        in f' end;
```

Funktionen als Parameter und Wert von Funktionen

- Unabhängig von der Form der Deklaration kann man mit `abl` erzeugte Funktionen selbst als Parameter von `abl` verwenden:

```
- abl( abl(fn x => x*x, 1e~5), 1e~5 )(5.0);
val it = 2.00003569262 : real
- abl( abl(fn x => x*x*x, 1e~5), 1e~5 )(5.0);
val it = 30.0002511722 : real
```

- Die Ableitung der Ableitung von `fn x =>` x^2 ist die Funktion

  `fn x => 2.`

- Die Ableitung der Ableitung von `fn x =>` x^3 ist die Funktion

  `fn x => 6*x.`

Funktionen als Parameter und Wert von Funktionen

- Ein weiteres Beispiel einer Funktionen höherer Ordnung ist die **Identitätsfunktion**:

```
- val id = fn x => x;
val id = fn : 'a -> 'a
- id(2);
val it = 2 : int
- id(id)(2);
val it = 2 : int
```

- Im Teilausdruck `id(id)` hat die Funktion `id` sogar sich selbst als Parameter und liefert auch sich selbst als Ergebnis.
- `id` ist eine Funktion höherer Ordnung (da Funktionen als Parameter erlaubt sind).
- `id` ist polymorph, da `id` auf Objekte unterschiedlicher Typen anwendbar ist.
- `id` ist eine Funktion der Ordnung n für jedes n > 0, da `id` auch auf Funktionen beliebiger Ordnung angewendet werden darf.

6.2 Currying

- Gegeben sei die folgende mathematische Funktion:

  f : Z * Z * Z -> Z,

  f(n1, n2, n3) : int = n1 + n2 + n3;

- f kann wie folgt in SML implementiert werden:

```
- fun f(n1, n2, n3) : int = n1 + n2 + n3;
val f = fn : int * int * int -> int
```

- Im Folgenden bezeichnen wir mit F(A,B) die Menge der Funktionen von A nach B (A –> B).

Currying

- Aus der Funktion f lässt sich die folgende Funktion definieren:

  f1: Z -> F(Z*Z , Z),

  f1(n1) = n1 + n2 + n3

- Die Funktion f1 kann wie folgt in SML implementiert werden:

```
- fun f1(n1)(n2, n3) = f(n1, n2, n3);
val f1 = fn : int -> int * int -> int
- f1(1)(1, 1);
val it = 3 : int
- f1(1);
val it = fn : int * int -> int
```

Currying

- `int -> int * int -> int` entspricht

  `int -> ((int * int) -> int)`

  wegen der Rechtsassoziativität von "–>“ und der Präzedenzen zwischen "*“ und "–>“
- Die Funktion `f1` bildet also eine ganze Zahl auf eine Funktion vom Typ `((int * int) -> int)` ab.
- In ähnlicher Weise lässt sich die folgende Funktion definieren:

  f11(n1) : Z –> Z –> Z,

  f11(n1) n2 n3 = f1(n1)(n2, n3) = f(n1, n2, n3) = n1 + n2 + n3
- Die Funktion f11 lässt sich wie folgt in SML implementieren:

```
- fun f11(n1)(n2)(n3) = f1(n1)(n2, n3);
val f11 = fn : int -> int -> int -> int
```

Currying

- `int -> int -> int -> int` entspricht

  `int -> (int -> (int -> int))`

- **Beispiele**

```
- f11(1)(1)(1);
 val it = 3 : int
 - f11(1)(1);
 val it = fn : int -> int
 - f11(1);
 val it = fn : int -> int -> int
```

- So lässt sich jede n-stellige Funktion durch eine einstellige (unäre) Funktion höherer Ordnung darstellen.
- Dies ermöglicht oft eine wesentlich kompaktere und übersichtlichere Schreibweisen von Funktionsdefinitionen.

Currying (auch Schönfinkeln)

- Unter Verwendung des `fn`-Konstrukts werden `f1` und `f11` wie folgt deklariert:

```
- val f1 = fn n1 =>
    fn (n2, n3) => f(n1, n2, n3);
val f1 = fn : int -> int * int -> int
- val f11 = fn n1 => fn n2 =>
    fn n3 => f(n1, n2, n3);
val f11 = fn : int -> int -> int -> int
```

- Das Prinzip, das der Bildung von f1 aus f (und f11 aus f) zugrunde liegt, wird **Currying** genannt (nach dem Logiker Haskell B. Curry).
- Allerdings wurde dieses Prinzip schon vorher von **Moses Schönfinkel** beschrieben.
- Die n-stellige Funktion, die sich durch Currying aus einer (n+1)-stelligen Funktion f ergibt, wird "**curried“ Form** von f genannt.

- **Moses Schönfinkel**

  1889-1942

  Russ. Logiker

  1924 Erfinder der kombinatorischen Logik

  (nach Aufenthalt in Göttingen bei Bernays)

- **Haskell B. Curry**

1900-1982

Amerik. Logiker

1930 Diss. über komb. Logik (bei Hilbert)

[Wikipedia/ www-history.mcs.st-andrews.ac.uk]

Currying

- Curried Funktionen können auch rekursiv sein; z.B.:

```
- fun cggT a b =
    if a < b then cggT b a
    else if b = 0 then a
         else cggT b (a mod b);
val cggT = fn : int -> int -> int
- cggT 150 60;
val it = 30 : int
- cggT 150;
val it = fn : int -> int
```

- **Bemerkung:**

  Die Schreibweise `cggT b a mod b` statt `cggT b (a mod b)` wäre **inkorrekt**:

  Wegen der Präzedenzen bezeichnet sie `((cggT b) a) mod b`.

Die Funktion höherer Ordnung `curry`

- **Die Funktion `curry`**

```
- fun curry(f) = fn x => fn y => f(x, y);
val curry = fn : ('a * 'b -> 'c) -> 'a -> 'b -> 'c
```

oder

```
- val curry = fn f => fn x => fn y => f(x, y);
val curry = fn : ('a * 'b -> 'c) -> 'a -> 'b -> 'c
```

- Man beachte, dass `curry` eine polymorphe Funktion ist:

```
- curry(fn (a,b) => 2*a + b);
val it = fn : int -> int -> int
- curry(fn (a,b) => 2*a + b) 3;
val it = fn : int -> int
- curry(fn (a,b) => 2*a + b) 3 1;
val it = 7 : int
```

Umkehrung der Funktion `curry`

- **Die Funktion `uncurry`**

```
- fun uncurry(f) = fn (x, y) => f x y;
val uncurry = fn : ('a -> 'b -> 'c) -> 'a * 'b -> 'c
oder
- val uncurry = fn f => fn (x, y) => f x y;
val uncurry = fn : ('a -> 'b -> 'c) -> 'a * 'b -> 'c
```

- `uncurry` ist eine polymorphe Funktion:

```
- fun f x y = 2 + x + y;
val f = fn : int -> int -> int
- fun g x y = 2.0 + x + y;
val g = fn : real -> real -> real
- uncurry f;
val it = fn : int * int -> int
- uncurry f (1,1);
val it = 4 : int
- uncurry g;
val it = fn : real * real -> real
```

Umkehrung der Funktion `curry`

- **`uncurry` ist die Umkehrung von `curry`:**

```
- curry(uncurry(f));
val it = fn : int -> int -> int
- curry(uncurry(f)) 1 1;
val it = 4 : int
- uncurry(curry(uncurry(f)))(1,1);
val it = 4 : int
```

Nicht-curried und curried Funktionen im Vergleich

- Darstellung der Unterschiede am Beispiel einer einfachen Funktion:
  (Annahme:
  Die vordefinierte Funktion "*“ hat den Typ `int * int -> int`)

	nicht-curried	**curried**

mögliche Deklaration	`fun mal(x, y) = x * y`
	`val mal = (fn (x, y) => x * y)`
		`fun mal x y = x * y`
		`val mal = (fn x y => x * y)`
		`fun mal x = (fn y => x * y)`
		`val mal = (fn x => (fn y => x * y))`

Typ	`int * int -> int`	`int -> int -> int`
Aufruf	`mal(2,3)` (hat Wert 6)	`mal 2 3` (hat Wert 6)
zu wenige Aufrufparameter	—	`mal 2` (hat eine Funktion als Wert)
	`val doppelt = fn y => mal(2, y)`	`val doppelt = mal 2`

6.3 Funktionskomposition und die Kombinatoren I, S und K

- Die SML-Standardbibliothek enthält eine Funktion höherer Ordnung zur Funktionskomposition, die wie folgt definiert werden kann:

```
- infix o;
infix o
- fun (f o g)(x) = f(g(x));
val o = fn : ('a -> 'b) * ('c -> 'a) -> 'c -> 'b
- ((fn x => 2*x) o (fn x => x+1))(3);
val it = 8 : int
- Math.sqrt o ~;
val it = fn : real -> real
- val f = Math.sqrt o ~;
val f = fn : real -> real
- f(~4.0);
val it = 2.0 : real
```

- Die Infixfunktion o leistet eine Funktionskomposition in sogenannter "**Anwendungsreihenfolge**“ (oder Funktionalreihenfolge), d.h.:

(f o g)(x) = f(g(x))

Die Kombinatoren I, S und K

- Die polymorphe Identitätsfunktion `id` wird auch als `I` notiert und **Identitätskombinator** genannt:

```
- fun I x = x;
val I = fn : 'a -> 'a
- I 5;
val it = 5 : int
- I [1,2];
val it = [1,2] : int list
- I (fn x => 2 * x);
val it = fn : int -> int
```

Die Kombinatoren I, S und K

- Der **Kompositionskombinator** S verallgemeinert die Funktionskomposition mit o:

```
- fun S x y z = x z (y z);
val S = fn : ('a -> 'b -> 'c) -> ('a -> 'b) -> 'a -> 'c
```

- Der **Konstantenkombinator** K entspricht der Projektion auf die erste Komponente:

```
- fun K x y = x;
val K = fn : 'a -> 'b -> 'a
```

- Mittels S und K lässt sich der Identitätskombinator `I` als `S K K` definieren:

```
- S K K 5;
val it = 5 : int
- S K K [1,2];
val it = [1,2] : int list
```

Die Kombinatoren I, S und K

- Bedeutung dieser Kombinatoren:
  - Alleine mittels S und K können alle Funktionen des sogenannten Lambda-Kalküls ausgedrückt werden (der Lambda-Kalkül ist eine/die formale Grundlage der rein funktionalen Programmierung).
  - In anderen Worten: Man kann jeden Algorithmus nur unter Verwendung von S und K ausdrücken.
- Diese Kombinatoren spielen eine Rolle in der theoretischen Informatik.
  - Wenn alle Konzepte einer funktionalen Programmiersprache aus S und K abgeleitet werden können, reicht es Eigenschaften dieser Sprache nur für S und K zu beweisen.

6.4. Grundlegende Funktionen höherer Ordnung

- `map`
- `filter`
- `foldl und foldr`
- `exists und all`
- `repeat`

`map`: Anwenden einer Funktion auf alle Elemente einer Liste

- Die Funktion höherer Ordnung **`map`** wendet eine unäre Funktion auf alle Elemente einer Liste an und liefert als Wert die Liste der (Werte dieser) Funktionsanwendungen:

```
- fun quadrat(x : int) = x * x;
val quadrat = fn : int -> int
- map quadrat [2, 3, 5];
val it = [4, 9, 25]
- map Math.sqrt [4.0, 9.0, 25.0];
val it = [2.0, 3.0, 5.0] : real list
```

- Die Funktion map kann wie folgt in SML definiert werden:

```
- fun map f nil = nil
    | map f (h :: t) = f(h) :: map f t;
val map = fn : ('a -> 'b) -> 'a list -> 'b list
- map (fn x:int => x*x) [1,2,3,4,5];
val it = [1,4,9,16,25] : int list
- map (map quadrat) [[1,2], [3,4], [5,6]];
val it = [[1,4], [9,16], [25,36]] : int list list
```

`filter`: Herausfiltern aller Elemente einer Liste, die ein Prädikat erfüllen

- Die Funktion höherer Ordnung `filter` filtert aus einer Liste alle Elemente heraus, die ein Prädikat erfüllen:

```
- fun filter pred nil = nil
    | filter pred (h :: t) =
          if pred(h)
             then h :: filter pred t
             else filter pred t;
val filter = fn : ('a -> bool) -> 'a list -> 'a list
```

- Beispiele

```
- fun ist_gerade x = ((x mod 2) = 0);
val ist_gerade = fn : int -> bool
- filter ist_gerade [1,2,3,4,5];
val it = [2,4] : int list
- filter (fn x => (x mod 2) = 0) [1,2,3,4,5,6,7,8,9];
val it = [2,4,6,8] : int list
- filter (not o (fn x => (x mod 2) = 0)) [1,2,3,4,5,6,7,8,9];
val it = [1,3,5,7,9] : int list
```

`foldl` und `foldr`: Falten einer Liste von links und rechts

- Die Funktionen höherer Ordnung **`foldl`** und **`foldr`** wenden wie folgt eine Funktion auf die Elemente einer Liste an:
  - **`foldl f z [x1,x2,...,xn]` entspricht `f(xn,...,f(x2, f(x1, z))...)`**
  - **`foldr f z [x1,x2,...,xn]` entspricht `f(x1, f(x2, ..., f(xn, z)...))`**
- **In Sml**.

```
- fun foldl f z nil = z
    | foldl f z (x::L) = foldl f (f(x,z)) L;
val foldl = fn : ('a * 'b -> 'b) -> 'b -> 'a list -> 'b
- fun foldr f z nil = z
    | foldr f z (x::L) = f(x, foldr f z L);
val foldr = fn : ('a * 'b -> 'b) -> 'b -> 'a list -> 'b
```

- **Beispiele** (op erlaubt Infix-Operatoren als Präfix-Operatoren zu verwenden):

```
- foldl (op +) 0 [2,3,5]; (* entspricht 5 + (3 + (2 + 0)) *)
val it = 10 : int
- foldr (op +) 0 [2,3,5]; (* entspricht 2 + (3 + (5 + 0)) *)
val it = 10 : int
- foldl (op -) 0 [7,10];  (* entspricht 10 - (7 - 0) *)
val it = 3 : int
- foldr (op -) 0 [7,10];  (* entspricht 7 - (10 - 0) *)
val it = ~3 : int
```

`foldl` und `foldr`: Falten einer Liste von links und rechts

- Viele binäre Funktionen haben neutrale Elemente, z.B. ist `0` das neutrale Element für `+`; `1` für `*`; `nil` für `@` (Listenkonkatenation) und `""` für `^` (Stringkonkatenation).
- Die Funktionen `foldl` und `foldr` werden typischerweise so verwendet, dass das zweite Argument `z` das neutrale Element des ersten Arguments `f` ist.
- Die wichtigste Anwendung von `foldl` und `foldr` ist die Definition von neuen Funktionen auf Listen mit Hilfe von binären Funktionen auf den Elementen der Listen.
- **Beispiele**

```
- val listsum = foldl (op +) 0;
val listsum = fn : int list -> int
- val listprod = foldl (op * ) 1;
val listprod = fn : int list -> int
- val listconc = foldr (op ^) "";
val listconc = fn : string list -> string
- listsum [1,2,3,4];
val it = 10 : int
```

`exists`: Existenz eines Elements, das ein Prädikat erfüllt

- Die Funktion höherer Ordnung `exists` überprüft, ob ein Prädikat für manche Elemente einer Liste erfüllt ist:

```
- fun ist_gerade x = ((x mod 2) = 0);
val ist_gerade = fn : int -> bool
- exists ist_gerade [1,2,3,4,5];
val it = true : bool
- exists ist_gerade [1,3,5];
val it = false : bool
```

- Die Funktion `exists` kann wie folgt in SML implementiert werden:

```
- fun exists pred nil = false
    | exists pred (h::t) = (pred h) orelse exists pred t;
val exists = fn : ('a -> bool) -> 'a list -> bool
```

`all`: Prüfung, ob alle Elemente ein Prädikat erfüllen

- Die Funktion höherer Ordnung `all` überprüft, ob ein Prädikat für alle Elemente einer Liste erfüllt ist:

```
- all ist_gerade [1,2,3,4,5];
val it = false : bool
- all ist_gerade [2,4,6];
val it = true : bool
```

- Die Funktion `all` kann wie folgt in SML implementiert werden:

```
- fun all pred nil = true
    | all pred (h::t) = (pred h) andalso all pred t;
val all = fn : ('a -> bool) -> 'a list -> bool
```

`repeat`: Wiederholte Funktionsanwendung

- Angewandt auf eine Funktion `f` und eine natürliche Zahl n ist die Funktion höherer Ordnung `repeat` eine Funktion, die n-mal die Funktion `f` anwendet:

  $$\texttt{repeat f n x} = f^n(x)$$

- **In SML**

```
- fun repeat f 0 x = x
    | repeat f n x = repeat f (n-1) (f x);
val repeat = fn : ('a -> 'a) -> int -> 'a -> 'a
- repeat (fn x => x+1) 3 4;
val it = 7 : int
- repeat (fn x:int => x*x) 3 2;
val it = 256 : int
```

Zusammenfassung

- Funktionen, die als Parameter oder als Wert Funktionen haben, werden **Funktionen höherer Ordnung** genannt.
- Durch **Currying** lässt sich jede n-stellige Funktion durch eine einstellige (unäre) Funktion (höherer Ordnung) darstellen. Die Umkehrung **Uncurrying** konvertiert eine Funktion höherer Ordnung in eine mehrstellige Funktion.
- **Funktionskomposition, `map`, `filter`, `foldl`, `foldr`, `exists`, `all` und `repeat`** sind weitere nützliche Funktionen höherer Ordnung.
- Die **Kombinatoren** `I,S,K` erlauben es alle berechenbaren Funktionen rein funktional (ohne explizite Argumente) darzustellen.