Funktionen höherer Ordnung

Einführung in die Programmierung

Johannes Brauer

1. Januar 2020

#q+SETUPFILE: ./theme-bigblow-local.setup

Erweiterung der funktionalen Abstraktion

Historie:

- Infinitesimal-Rechnung (17. Jhdt. Newton, Leibniz) - Funktionen mit Funktionen als Argument
- 1941 Anlonzo Church: Lambda-Kalkül - Daten sind Funktionen
- 1960 John McCarthy: Lisp - Funktionen sind Daten
- 2014 Lambda-Ausdrücke im „Mainstream" (Java) angekommen
- Google, Facebook, Twitter, Netflix - alle stützen sich auf funktionale Programmierung

Funktionen als Argumente – Analogie zur Summenformel der Mathematik

$$\sum_{i=a}^{b} f(i) = f(a) + \ldots + f(b) \tag{1}$$

entspricht

```
(define sum
  (lambda [f a next b]
    (if (> a b)
      0
      (+ (f a)
         (sum f (next a) next b)))))
```

Anwendungsbeispiele:

$$\sum_{x=1}^{5} x^3 \tag{2}$$

entspricht

```
(define cube (lambda [x] (* x x x)))
(sum cube 1 add1 5)
```

$$\sum_{i=1}^{10} i \tag{3}$$

entspricht

```
(sum identity 1 add1 10)
```

Weitere Beispiele: Integrale

Nutzung anonymer Funktionen (Lambda-Ausdrücke)

- Lambda-Ausdrücke bereits eingeführt im Abschnitt *Funktionale Abstraktion*
- Beispiele aus dem vorigen Abschnitt:

  ```
  (sum (lambda [x] (* x x x)) 1 (lambda [i] (+ i 1)) 5)
  ```

  oder

  ```
  (sum (lambda [x] x) 1 (lambda [i] (+ i 1)) 10)
  ```

Funktionen höherer Ordnung

> The Revised[6] Report on the Algorithmic Language Scheme:
> Programming languages should be designed not by piling feature on top of feature, but by removing the weaknesses and restrictions that make additional features appear necessary.
>
> Tony Hoare:
> There are two ways of constructing a software design: One way is to make it so simple that there are obviously no deficiencies and the other is to make it so complicated that there are no obvious deficiencies.
>
> Hoare, C. A. R.: The Emperor's Old Clothes 1980 Turing Award lecture; Comm.ACM vol24-81, 2 (Feb. 1981)

Regelkonforme Entwicklung von Funktionen ...

führt zu ähnlichen Funktionen. Zum Beispiel

```
;; sum: (list-of number) -> number
(define sum
  (lambda [x]
   (cond
     [(empty? x) 0]
     [else (+ (first x) (sum (rest x)))])))

;; prod: (list-of number) -> number
```

```
(define prod
  (lambda [x]
   (cond
     [(empty? x) 1]
     [else (* (first x) (prod (rest x)))])))
```

Noch zwei ähnliche Funktionen

Beispiel: Aus einer ungeordneten Liste von Zahlen werden alle unterhalb/oberhalb eines Schwellwerts heraus gefiltert:

```
;; unterhalb/oberhalb: (list-of number) number -> (list-of number)
(define unterhalb/oberhalb
  (lambda [eineLvz schwelle]
    (cond
      [(empty? eineLvz) empty]
      [else
       (cond
         [(</> (first eineLvz) schwelle)
          (cons (first eineLvz)
                (unterhalb/oberhalb (rest eineLvz) schwelle))]
         [else (unterhalb/oberhalb (rest eineLvz) schwelle)])])))
```

Ähnliche Funktionen vermeiden

- Die beiden Funktionen `unterhalb` und `oberhalb` unterscheiden sich – abgesehen von ihrem Namen – nur in dem Vergleichsoperator.
- Die Funktion `oberhalb` erzeugt man wahrscheinlich durch Kopieren und Anpassen von `unterhalb`.
- Kopieren von Code ist eine „Todsünde" wider die Pflegbarkeit von Programmen.
- Bessere Lösung: Eine allgemein verwendbare Filterfunktion, der die Art der Vergleichsoperation als Argument mitgegeben wird.
- Wir brauchen eine Funktion, die eine Funktion als Argument akzeptiert: *Funktion höherer Ordnung* (Typ 1)

Eine Filterfunktion

```
;; filtere (number number -> boolean) (list-of number) number
;;          -> (list-of number))
(define filtere
  (lambda [vglfkt eineLvz schwelle]
            ;;;;;;
    (cond
      [(empty? eineLvz )empty ]
      [else
       (cond
         [(vglfkt (first eineLvz ) schwelle)
```

```
    ;;;;;;
   (cons (first eineLvz )
         (filtere vglfkt
          (rest eineLvz ) schwelle))]
  [else (filtere vglfkt
         (rest eineLvz ) schwelle)])]))))
```

- Um `unterhalb` zu realisieren ruft man `filtere` so auf:

  ```
  (filtere < '(3 4 5 6) 5) ;;=> (3 4)
  ```

- Man beachte den Vertrag der Funktion!

Funktionen sind Werte erster Ordnung

- Die Möglichkeit, Funktionen, die andere Funktionen als Argument erwarten, zu bauen, beruht auf einem ebenso einfachen wie fundamentalen Prinzip:

  Funktionen sind Werte erster Ordnung

- Das bedeutet im Grunde, dass Funktionen überall dort, wo normale Werte verwendet werden dürfen, ebenfalls benutzt werden können. D.h. sie dürfen als

  - Argumente von Funktionen benutzt,
  - als Resultate von Funktionen geliefert und
  - in Datenstrukturen verpackt

  werden.

- Das Prinzip eröffnet eine breite Palette an Programmiertechniken.

Verallgemeinerung der Filterfunktion

```
;; filter wendet auf jedes Element der Zahlenliste das Prädikat an.
;; In die Ergebnisliste werden die Elemente aufgenommen, für die das
;; Prädikat true liefert.
;; filter : (number -> boolean) (list-of number) -> (list-of number)
(check-expect (filter (lambda [x] (< x 5)) '(3 4 5 6 )) '(3 4))
(check-expect (filter odd? '(3 4 5 6 )) '(3 5))
(define filter
  (lambda [praedikat eineLvz]
    (cond
      [(empty? eineLvz) empty]
      [(praedikat (first eineLvz))
         (cons (first eineLvz ) (filter praedikat (rest eineLvz)))]
      [else (filter praedikat (rest eineLvz))])))
```

Eine Abstraktion für `sum` und `prod`

- Es kommt häufiger vor, dass man z.B. eine Liste von Zahlen zu einem Wert zusammenfalten will.
- Die verschiedenen Faltungsarten unterscheiden sich meist nur in dem Basiswert (Ergebnis für die leere Liste) und der eigentlichen Faltungsfunktion – z.B. Addition für `sum` und Multiplikation für `prod`.
- Man beachte auch hier wieder den Vertrag der Funktion!

```
;; falte (number number -> number) number (list-of number) -> number
(define falte
  (lambda [fltfkt basis aLon]
            ;;;;;; ;;;;;
    (cond
      [(empty? aLon) basis]
                     ;;;;;
      [else
        (fltfkt (first aLon)
                (falte fltfkt basis (rest aLon)))])))
                       ;;;;;; ;;;;;
```

Ähnliche Datenstrukturen ...

... haben ähnliche Funktionen zur Folge.

- Zum Beispiel sind die rekursiven Datenstrukturen *Liste-von-Zahlen* und *Liste-von-Symbolen* isomorph:
  - Eine *Liste-von-Zahlen* ist entweder
    1. `empty` oder
    2. `(cons z lvz)`, wobei z eine **Zahl** und lvz eine *Liste-von-Zahlen* ist.
  - Eine *Liste-von-Symbolen* ist entweder
    1. `empty` oder
    2. `(cons s lvs)`, wobei s eine **Symbol** und lvs eine *Liste-von-Symbolen* ist.
- Der wesentliche Unterschied ist hervorgehoben.
- Will man eine Funktion `enthaelt?` definieren, die prüft, ob eine Zahl bzw. ein Symbol in der Liste enthalten ist, muss man ihr ein Vergleichsprädikat als Argument mitgeben.

Eine generische Funktion `enthaelt?`

```
(define enthaelt?
  (lambda [gleich? liste elem]
    (cond
      [(empty? liste) #false]
      [else
```

```
    (cond
      [(gleich? (first liste) elem) #true]
      [else (enthaelt? gleich? (rest liste) elem)])]))))
```

Anwendungen:

```
(enthaelt? = '(2 3 4 5) 4) ;;=> #true
(enthaelt? equal? '(a b c d) 'b) ;;=> #true
```

- Die Funktion `enthaelt?` kann alle Arten von Listen verarbeiten. Der Typ der Elemente spielt keine Rolle.
- Solche Funktionen heißen *generisch* oder *polymorph*.
- Wie lautet der Vertrag für diese Funktion?

Parametrisierte Datenstruktur-Definitionen

- Da eine generische, Listen verarbeitende Funktion Listen von irgendetwas akzeptiert, brauchen wir eine generische Datenstrukturdefinition.
- Eine Liste von X ist entweder
  1. `empty` oder
  2. `(cons e l)`, wobei e ein X und l eine Liste von X ist.
- X bezeichnet man als Typparameter (Signaturvariable).
- X kann durch jeden eingebauten Typ aber auch durch jede selbst definierte Datenstruktur ersetzt werden.
- In Verträgen benutzen wir die Notation `(list-of X)`, um auszudrücken, dass eine Funktion beliebige Listen akzeptiert.
- Für das generische `enthaelt?`-Prädikat ergibt sich dann der Vertrag:

```
;; enthaelt? (X X -> boolean) (list-of X) X -> boolean
```

Die Funktion `map`

Zunächst zwei Beispiele

1. Quadrieren einer Liste von Zahlen

```
;; quadrieren: (list-of number) -> (list-of number)
(define quadrieren
  (lambda [lvz]
    (cond
      [(empty? lvz) empty]
      [else (cons ((lambda [x] (* x x)) (first lvz))
                  (quadrieren (rest lvz)))])))
```

2. Extrahieren von Produktnamen aus einer Preisliste

```
;; Ein Produkt besteht aus
;; - einem Symbol für den Namen
;; - einer Zahl für den Preis
;; ...
(define-struct produkt [name preis])

;; produkt-namen: (list-of produkt) -> (list-of symbol)
(define produkt-namen
  (lambda [pl]
    (cond
      [(empty? pl) empty]
      [else (cons (produkt-name (first pl))
                  (produkt-namen (rest pl)))])))
```

Verallgemeinerung von `quadrieren` und `produkt-namen`

- Die beiden Funktionen unterscheiden sich – abgesehen von ihren Namen – nur in der Funktion, die auf das erste Listenelement angewendet wird.
- Wann immer zwei (oder mehr) dieser Art ähnliche Funktionen auftreten, gehe man folgendermaßen vor:
  1. Markiere die Stellen in den Funktionen, wo sie sich unterscheiden. Wenn sie sich nur in Werten (auch Funktionen sind Werte) unterscheiden, ist eine Abstraktion möglich.
  2. Führe für jedes Paar markierter Stellen einen neuen Namen ein.
  3. Übernimm diesen Namen in die Liste der formalen Parameter der Funktionen. Nach diesem Schritt müssen die beiden Funktionen außer in ihrem Namen übereinstimmen.
  4. Schließlich ersetzen wir die Funktionsnamen durch den Namen einer neuen abstrakten Funktion.
- So entsteht aus `quadrieren` und `produkt-namen` formal die

  Funktion `map`:

  ```
  ;; map: (X -> Y) (list-of X) -> (list-of Y)
  (define map
    (lambda [fun lst]
      (cond
        [(empty? lst) empty]
        [else (cons (fun (first lst))
                    (map fun (rest lst)))])))
  ```
- Man beachte wieder den Vertrag der Funktion!
- Als letzten Schritt des Abstraktionsprozesses definiert man die Originalfunktionen mithilfe der neuen abstrahierten Funktion:

  ```
  (define quadrieren (lambda [lvz] (map (lambda [x] (* x x)) lvz)))
  (define produkt-namen (lambda [pl] (map produkt-name pl)))
  ```

Verallgemeinerung von `falte`

Ein anderes Verständnis für `falte`

- Die Reduktion einer Liste zu einem Wert ist eine häufig vorkommende Aufgabe.
- Als Funktion höherer Ordnung gehört die Falte-Funktion zum Standard-Repertoire funktionaler Programmiersysteme.
- häufig verwendete Bezeichner: `fold`, `reduce`
- In Racket gibt es `foldr` und `foldl`. Das oben definierte `falte` entspricht `foldr`.
- Die Wirkung von `(falte + 0 '(11 17 5))` kann man sich, wie folgt, veranschaulichen:
  - `falte` ersetzt in der Argumentliste, die dem Ausdruck `(cons 11 (cons 17 (cons 5 empty)))` entspricht, jedes Auftreten von `cons` durch `+` und jedes Auftreten von `empty` durch 0: `(+ 11 (+ 17 (+ 5 0)))`
  - Verallgemeinert: **Jedes Auftreten von `cons` wird durch die Faltungsfunktion (`fltfkt`) und jedes Auftreten von `empty` durch den Basiswert ersetzt**.

Weitere Anwendungen von `falte`

- Was liefern die Ausdrücke
  1. (falte cons empty '(a b c)) ?
  2. (falte cons '(x y z) '(a b c)) ?

Generisches falte in Racket: foldr

- Abstraktion von `falte` hinsichtlich des Typs der Listenelemente
- Aus dem Vertrag

  ```
  ;; falte: (number number -> number) number (list-of number) -> number
  ```

  wird:

  ```
  ;; foldr: (X Y -> Y) Y (list-of X) -> Y
  ```

Die Funktionsdefinition für `foldr`

```
;; foldr: (X Y -> Y) Y (list-of X) -> Y
(define foldr
   (lambda [combine base l]
     (cond
       [(empty? l) base]
       [else (combine
                (first l)
                (foldr combine base (rest l)))])))
```

- Der Parameter `base` steht für den Funktionswert im Fall der leeren Liste.
- Der Parameter `combine` ist die Funktion, die aus dem Zugriff auf das erste Element der Liste und der rekursiven Anwendung der Funktion auf die Restliste den Funktionswert ermittelt, falls die Liste `l` nicht leer ist.

Anwendungen von `foldr`

```
;; sum: (list-of number)  ->  number)
(define sum (lambda [list] (foldr + 0 list)))

;; product: (list-of number)  ->  number
(define product (lambda [list] (foldr * 1 list)))

;; erzeugt eine Liste aus all den Elementen von alox,
;; für die p gilt
;; filterr (X  ->  boolean) (list-of X) -> (list-of X)))
(define filterr
  (lambda [p alox]
    (letrec ((cp (lambda [x y]
                   (cond [(p x) (cons x y)]
                         [else y]))))
      (foldr cp empty alox))))
```

Unterschied zwischen `foldr` und `foldl`

```
(foldl - 0 '(1 2 3 4))
= (- 4 (- 3 (- 2 (- 1 0))))
= (- 4 (- 3 (- 2 1)))
= (- 4 (- 3 1))
= (- 4 2)
= 2

(foldl cons '() '(1 2 3))
= (cons 3 (cons 2 (cons 1 '())))
= (cons 3 (cons 2 '(1)))
= (cons 3 '(2 1))
= '(3 2 1)

(foldr - 0 '(1 2 3 4))
= (- 1 (- 2 (- 3 (- 4 0))))
= (- 1 (- 2 (- 3 4)))
= (- 1 (- 2 -1))
= (- 1 3)
= -2

(foldr cons '() '(1 2 3))
= (cons 1 (cons 2 (cons 3 '())))
= (cons 1 (cons 2 '(3)))
= (cons 1 '(2 3))
= '(1 2 3)
```

Integration von Daten und Funktionen – Einführung des Objektbegriffs

Funktionen mit Funktionen als Resultat

- Was liefert der Ausdruck `(inc 3)`, wenn die Funktion `inc` wie folgt definiert ist?

```
(define inc
   (lambda [n] (lambda [z] (+ z n))))
```

- Welches Resultat liefert eine Anwendung der Funktion `dot`?

```
(define dot
   (lambda [f g] (lambda [x] (f (g x)))))
```

- Anwendung von `inc` und `dot`:

```
(define inclist
   (lambda [x] (map (inc 1) x)))

(define increv
   (lambda [x] ((dot inclist reverse) x)))

(increv '(3 13 23 33)) ==> (34 24 14 4)
```

Was ist ein Objekt?

- Objekte im Sinne der „objektorientierten Programmierung" sind nichts weiter als Funktionen mit „internen" Daten.
- So gesehen liefert ein Ausdruck der Art
  `(define inc-obj (inc 3))`
  ein Objekt (eine Funktion) mit dem internen Datum (Zustand) **3**.
- Die Funktion bezeichnet man in der objektorientierten Programmierung auch als *Methode*.
- Durch den Ausdruck `(inc-obj 7)` wird die Methode aktiviert (die Funktion aufgerufen). Sie verlangt ein Argument.
- `inc-obj` ist ein sehr primitives Objekt mit nur einer (unveränderlichen) Zustandsgrößen und einer Methode.

Ein etwas interessanteres Objekt

```
(define person
  (lambda [name vorname]
    (let* ([get-name     (lambda [] name)]
           [get-vorname  (lambda []vorname)]
           [vollname     (lambda [] (string-append vorname " " name))]
           [gruss        (lambda [g] (string-append g " " (vollname)))])
      (lambda [nachricht]
```

```
    (cond
      [(equal? nachricht 'get-name) get-name]
      [(equal? nachricht 'get-vorname) get-vorname]
      [(equal? nachricht 'get-vollname) vollname]
      [(equal? nachricht 'gruss) gruss]
      [else (error "unbekannte Nachricht")])))))
```

- Ein Aufruf der Funktion `person` liefert ein „Objekt" mit
  - zwei Zustandsgrößen (`name`, `vorname`) und
  - vier Methoden: `get-name`, `get-vorname`, `vollname` und `gruss`
- „In Wirklichkeit" liefert ein Aufruf der Funktion `person` eine Verteilfunktion, die anhand ihres Parameters `nachricht` ermittelt, welche der Methoden aufzurufen ist.

Anwendungen des `person`-Objekts

```
;; Objekterzeugung:
(define p1 (person "Gans" "Gustav"))
((p1 'get-vorname))      ;;=> "Gustav"
((p1 'get-name))         ;;=> "Gans"
((p1 'get-vollname))     ;;=> "Gustav Gans"
((p1 'gruss) "hallo")    ;;=> "hallo Gustav Gans"
(p1 'x)                  ;; error
```

Java-Klasse `Person`

```
 public class Person
{
   String vorname, name;

    public Person (String n, String v)
    {
        vorname = v;
        name = n;
    }

    public String get_name()
    {
        return name;
    }
    public String get_vorname()
    {
        return vorname;
    }
    public String vollname()
    {
        return vorname + " " + name;
    }
```

```
    public String gruss(String gruss)
    {
        return gruss + " " + this.vollname();
    }
}
```

Zusammenfassung

- Für die objektorientierte Programmierung bedarf es keiner speziellen Programmiersprachen.
- Objektorientierung besteht nur aus einer Reihe von Konventionen zum Umgang mit Daten.
- Diese Konventionen werden von objektorientierten Sprachen explizit und implizit unterstützt. Z. B. muss die Verteilfunktion nicht selbst programmiert werden.
- Objekte sind nichts weiter als
  - eine Menge von `Name->Wert`-Abbildungen
  - eine Reihe von Funktionen, die solche Abbildungen als erstes Argument akzeptieren und
  - eine Verteilfunktion, die ermittelt, welche dieser Funtkion aufzurufen ist.