Informatik I

11. Prozeduren als Daten

Jan-Georg Smaus

Albert-Ludwigs-Universität Freiburg

9. Dezember 2010

Informatik I

9. Dezember 2010 — 11. Prozeduren als Daten

11.1 Prozeduren als Eingabe

11.2 Prozedurfabriken

Prozeduren als Daten?

Prozeduren rechnen auf Daten, d.h., sie haben Daten als Ein- und Ausgabe.
Was sind Daten? Zahlen, Booleans, Strings, gemischte Daten, zusammengesetzte Daten, insbesondere Listen ...
Man kann aber auch Prozeduren höherer Ordnung schreiben, die Prozeduren als Daten (Ein- oder Ausgabe) haben. Das nennt man Programmierung höherer Ordnung.
Es ist ein Abstraktionsmittel: wir vermeiden, sehr ähnliche Programmstücke immer wieder schreiben zu müssen, indem das entsprechende Programmstück durch einen Namen ersetzen.

11.1 Prozeduren als Eingabe

- `filter`
- Listen falten

Unterliste aus Liste extrahieren

Beispiel: Gästeliste

```
; Ein Gast ist ein Wert
;  (make-guest name male veggie)
; wobei name   : string
;       male   : boolean (#f f\"ur weiblich)
;       veggie : boolean
(define-record-procedures guest
  make-guest guest?
  (guest-name guest-male? guest-vegetarian?))
(: make-guest (string boolean boolean -> guest))
```

VegetarierInnen suchen

```
; aus einer Gästeliste die Vegetarier extrahieren
(: guests-veg ((list guest) -> (list guest)))
(define guests-veg
  (lambda (guests)
    (cond
     ((empty? guests)
      empty)
     ((cons? guests)
      (let ((guest (first guests))
            (result (guests-veg (rest guests))))
        (if (guest-vegetarian? guest)
            (cons guest result)
            result))))))
```

Männer suchen

```
; aus einer Gästeliste die Männer extrahieren
(: guests-male ((list guest) -> (list guest)))
(define guests-male
  (lambda (guests)
    (cond
     ((empty? guests)
      empty)
     ((cons? guests)
      (let ((guest (first guests))
            (result (guests-male (rest guests))))
        (if (guest-male? guest)
            (cons guest result)
            result))))))
```

Was ist der Unterschied zum vorherigen Programm?
Einmal `guest-veg?`, einmal `guest-male?`.

Nach einem Prädikat suchen

```
; aus einer Gästeliste die Gäste extrahieren,
; die ein Prädikat erfüllen
(: guests-filter
   ((guest -> boolean) (list guest) -> (list guest)))
(define guests-filter
  (lambda (pred? guests)
    (cond
     ((empty? guests)
      empty)
     ((cons? guests)
      (let ((guest (first guests))
            (result (guests-filter pred?
                                   (rest guests))))
        (if (pred? guest)
            (cons guest result)
            result))))))
```

Verwendung von guests-filter

```
; aus einer Gästeliste die Vegetarier extrahieren
(: guests-veg ((list guest) -> (list guest)))
(define guests-veg
  (lambda (guests)
    (guests-filter guest-vegetarian? guests)))

; aus einer Gästeliste die Männer extrahieren
(: guests-male ((list guest) -> (list guest)))
(define guests-male
  (lambda (guests)
    (guests-filter guest-male? guests)))
```

War's das an Allgemeinheit?

Die Signatur von `guests-filter` lautet

```
((guest -> boolean) (list guest) -> (list guest))
```

Geht es nicht noch allgemeiner?
Z.B. könnte man aus einem Zug alle Nichtspeisewagen extrahieren.
Die Sorte `guest` ist zu speziell und sollte durch eine Sortenvariable ersetzt werden.

filter als polymorphe Prozedur

```
; aus einer Liste die Elemente extrahieren,
; die ein Prädikat erfüllen
(: filter ((%a -> boolean) (list %a) -> (list %a)))
(define filter
  (lambda (pred? xs)
    (cond
     ((empty? xs)
      empty)
     ((cons? xs)
      (let ((x (first xs))
            (result (filter pred? (rest xs))))
        (if (pred? x)
            (cons x result)
            result))))))
```

Unterschied zu `guests-filter`? Nur einige Namen!

filter ist Prozedur höherer Ordnung

filter abstrahiert vom Muster der Prozeduren guests-veg und guests-male. Man nennt filter eine Prozedur höherer Ordnung (siehe -> in der Signatur).
Vorteile von Prozeduren höherer Ordnung:

- Verkürzung des Programms
- Verbesserung der Lesbarkeit des Programms
- Verbesserung der Wartbarkeit des Programms
  - Ein Fehler in filter muss nur einmal korrigiert werden.
  - Wenn filter korrekt ist, dann sind auch alle Anwendungen von filter korrekt.

Mantra

Mantra #11 (Abstraktion aus Mustern)

Wenn mehrere Prozeduren bis auf wenige Stellen gleich aussehen, dann schreibe eine allgemeinere Prozedur, die über diese Stellen abstrahiert. Ersetze die ursprünglichen Prozeduren durch Anwendungen der neuen, allgemeineren Prozedur.

Listen falten

Vielen Listenoperationen liegt eine einzige gemeinsame Abstraktion zugrunde: Das Falten einer Liste.

Beispiel 1: Elemente einer Liste aufsummieren

```
; Elemente einer Liste aufsummieren
(: list-sum ((list number) -> number))
(define list-sum
  (lambda (l)
    (cond
     ((empty? l)
      0)
     ((cons? l)
      (+ (first l) (list-sum (rest l)))))))
```

Beispiel 2: Elemente einer Liste aufmultiplizieren

```
; Elemente einer Liste aufmultiplizieren
(: list-product ((list number) -> number))
(define list-product
  (lambda (l)
    (cond
     ((empty? l)
      1)
     ((cons? l)
      (* (first l) (list-product (rest l)))))))
```

Abstraktion des Musters liefert num-list-fold

```
; Elemente einer Liste auffalten
(: num-list-fold
   (number (number number -> number) (list number)
                                     -> number))
(define num-list-fold
  (lambda (e f l)
    (cond
     ((empty? l)
      e)
     ((cons? l)
      (f (first l) (num-list-fold e f (rest l)))))))
```

- `e` : Startwert für leere Liste
- `f` : Kombinationsfunktion für erstes Listenelement und rekursiven Aufruf auf der Restliste

Verwendung von num-list-fold

```
; Elemente einer Liste aufsummieren
(: list-sum ((list number) -> number))
(define list-sum
  (lambda (xs)
    (num-list-fold 0 + xs)))

; Elemente einer Liste aufmultiplizieren
(: list-product ((list number) -> number))
(define list-product
  (lambda (xs)
    (num-list-fold 1 * xs)))
```

num-list-fold ersetzt die Listenkonstruktoren

(num-list-fold *e f l*) ersetzt

- das Vorkommen von empty in *l* durch *e* und
- jedes Vorkommen von cons durch *f*

und wertet den entstehenden Ausdruck aus:

(num-list-fold *e f*
(cons x_1 (cons x_2 ... (cons x_n empty))))

$\Longrightarrow$

(*f* x_1 (*f* x_2 ... (*f* x_n *e*))))

War's das an Allgemeinheit?

Die Signatur von `num-list-fold` lautet

```
(number (number number -> number) (list number)
                                  -> number)
```

Geht es nicht noch allgemeiner?

```
(number (number number -> number) (list number)
                                  -> number)
```

Die Sorte `number` ist zu speziell und sollte durch Sortenvariablen ersetzt werden: `list number` durch `list %b`, `number` durch `%a`; die anderen Sorten ergeben sich zwangsläufig.
Diese Prozedur heißt dann `list-fold`.

Polymorphe Signatur für `list-fold`

```
; Elemente einer Liste auffalten
(: list-fold (%a (%b %a -> %a) (list %b) -> %a))
(define list-fold
  (lambda (e f l)
    (cond
     ((empty? l)
      e)
     ((cons? l)
      (f (first l) (list-fold e f (rest l)))))))
```

Unterschied zu `num-list-fold`? Keiner!

Beispiel: append

Zum Beispiel kann append (Listenkonkatenation) mit `list-fold` implementiert werden:

```
; Listen verketten
(: append ((list %a) (list %a) -> (list %a)))
(define append
  (lambda (xs ys)
    (list-fold ys cons xs)))
```

Anonyme Prozeduren

Listenlänge

```
; Länge einer Liste
(: list-length ((list %a) -> natural))
(define list-length
  (lambda (l)
    (cond
     ((empty? l) 0)
     ((cons? l)  (+ 1 (list-length (rest l)))))))
```

Aufgabe: Schreibe `list-length` mit Hilfe von `fold`

Listenlänge als Faltung

Passende Kombinationsfunktion:

```
; Hilfsfunktion zur Definition von length mit
; Hilfe von list-fold
(: add-1-for-length (%a natural -> natural))
(define add-1-for-length
  (lambda (ignore n)
    (+ 1 n)))

; Länge einer Liste
(: my-length ((list %a) -> natural))
(define my-length
  (lambda (xs)
    (list-fold 0 add-1-for-length xs)))
```

`add-1-for-length` hat nur eine Verwendung im Programm $\Rightarrow$ es lohnt nicht, sie zu benennen!

Listenlänge als Faltung mit anonymer Prozedur

```
; Länge einer Liste
(: my-length ((list %a) -> natural))
(define my-length
  (lambda (xs)
    (list-fold 0 (lambda (ignore n) (+ 1 n)) xs)))
```

Filtern als Faltung

```
(define filter
  (lambda (pred? xs)
     (list-fold empty
                (lambda (elem result)
                   (if (pred? elem)
                       (cons elem result)
                       result))
                xs)))
```

Gültigkeit als Faltung

```
; prüfen, ob ein Prädikat auf alle Elemente einer
; Liste zutrifft
(: every? ((%a -> boolean) (list %a) -> boolean))
(define every?
  (lambda (pred? xs)
    (list-fold #t
               (lambda (elem result)
                 (and (pred? elem) result))
               xs)))
```

11.2 Prozedurfabriken

- Schönfinkel-Isomorphismus

Komposition von Prozeduren

```
; zwei Prozeduren komponieren
; (zusammensetzen, hintereinander ausführen)
(: compose ((%b -> %c) (%a -> %b) -> (%a -> %c)))
(define compose
  (lambda (f g)
    (lambda (x) (f (g x)))))
```

`(compose` f g`)` liefert die Komposition der Funktionen f und g, d.h., eine Funktion, die zuerst g auf ihr Argument anwendet und dann f auf das Ergebnis anwendet.

Dies ist analog zu der Komposition $S \circ R$ von Relationen.

Beispiele für Prozedurkomposition

Eine Funktion, die erst quadriert und dann noch 5 addiert:

```
(define add-5 ;5 addieren
   (lambda (x) (+ 5 x)))
(compose add-5 square) ;(lambda (x) (+ 5 (* x x)))
(define f (compose add-5 square))
(f 0)   ; = 5
(f 1)   ; = 6
(f 2)   ; = 9
```

Eine Funktion, die das zweite Element einer Liste liefert:

```
(define second (compose first rest))
(second (list 1 2 3))   ; = 2
```

Beispiel für die Verwendung

Bestimme Wert von `second`:

```
   (compose first rest)
=> ((lambda (f g)
      (lambda (x) (f (g x))))
    first
    rest)
=> (lambda (x) (first (rest x)))
```

Bestimme nun Wert von `(second (list 1 2 3))`

```
   (second (list 1 2 3))
=> ((lambda (x) (first (rest x))) (list 1 2 3))
=> (first (rest (list 1 2 3)))
=> 2
```

Prozedur wiederholt anwenden

```
; Prozedur mit sich selbst komponieren
(: repeat (natural (%a -> %a) -> (%a -> %a)))
(define repeat
  (lambda (n f)
    (if (zero? n)
        (lambda (x) x)
        (compose f (repeat (- n 1) f)))))
```

Was tut

```
((repeat 6 (lambda (n) (* 2 n))) 1)
```

?

Prozeduren spezialisieren

Manchmal wird eine spezialisierte Version einer mehrstelligen Prozedur benötigt, z.B. `add-5`:

```
(define add-5 ;5 addieren
   (lambda (x) (+ 5 x)))
(compose add-5 square) ;(lambda (x) (+ 5 (* x x)))
(define f (compose add-5 square))
```

Der Schönfinkel-Isomorphismus

Alternative: Definiere die geschönfinkelte (curryfizierte) Version der Addition

```
(define make-add
  (lambda (x)
    (lambda (y)
      (+ x y))))
```

`make-add` erwartet zwei Argumente nacheinander.
Verwendung:

```
(compose (make-add 5) square)
(define f (compose (make-add 5) square))
```

Zwei weitere Prozedurfabriken

Prozedur erzeugen, die mit einer Konstante multipliziert:

```
(: make-mult (number -> (number -> number)))
(define make-mult
  (lambda (x)
    (lambda (y)
      (* x y))))
```

Prozedur erzeugen, die ein Element an eine Liste anfügt:

```
(: make-prepend (%a -> ((list %a) -> (list %a))))
(define make-prepend
  (lambda (x)
    (lambda (y)
      (cons x y))))
```

Die curry-Prozedur

```
; Prozedur mit zwei Parametern staffeln
(: curry ((%a %b -> %c) -> (%a -> (%b -> %c))))
(define curry
  (lambda (f)
    (lambda (x)
      (lambda (y)
        (f x y)))))
```

Damit ergeben sich

```
(define make-add     (curry +))
(define make-mult    (curry *))
(define make-prepend (curry cons))
```

Entstaffeln von Prozeduren

```
; Prozedur mit zwei Parametern entstaffeln
(: uncurry ((%a -> (%b -> %c)) -> (%a %b -> %c)))
(define uncurry
  (lambda (f)
    (lambda (x y)
      ((f x) y))))
```

Es gilt für alle p mit passender Sorte:

$$\texttt{(uncurry (curry } p\texttt{))} \equiv p$$

Zusammenfassung

- Prozeduren abstrahieren von wiederkehrenden Programmstücken.
- Prozeduren höherer Ordnung erlauben ...
  - ...die Abstraktion von Programmiermustern;
  - ...das Zusammensetzen von Programmiermustern.
- Anonyme Prozeduren
- Schönfinkel-Isomorphismus