Ausdrucksmittel funktionaler Programmiersprachen

Programmierparadigmen

Johannes Brauer

4. April 2020

Ziele

- Kennen und Anwenden besonderer Ausdrucksmittel verschiedener funktionaler Sprachen:
  - Mustervergleich (pattern matching) in *SML*
  - Typinferenz in *SML*
  - Partielle Anwendung von Funktionen
  - Curryfizierung
  - Destructuring in *Clojure*

Mustervergleich (pattern matching)

Algebraische Datentypen

Im Modul *Einführung in die Programmierung* wurden algebraische Datentypen behandelt: Produkttypen und Summentypen.

Produkttypen

- auch Verbünde, records genannt
- Die Menge aller Exemplare der Strukturdefinition

  ```
  (define-struct point [x y])
  ```

  kann als Teilmenge des kartesischen Produkts

  $$number \times number$$

  angesehen werden.

Produkttypen in SML

- Tupel (vgl. Zusammengesetzte Typen: Tupel)
  - Komponentenzugriff über Indizes (by position)
- Verbünde (vgl. Zusammengesetzte Typen: Verbünde)
  - Komponentenzugriff über Feldbezeichner (by name)
- Ein Punkt-Exemplar als Verbund:

  ```
  {x=12, y=9}
  ```

- Deklaration eines Typbezeichners:

  ```
  type point = { x : int, y: int }
  ```

- `type` definiert Typsynonyme:

```
- type point = { x : int, y: int };
type point = {x:int, y:int}
- {x=12, y=9};
val it = {x=12,y=9} : {x:int, y:int}
```

Summentypen

- in *Einführung in die Programmierung* als *gemischte Daten* bezeichnet
- Racket kennt keine Syntax für Summentypen.
- Beispiel aus EidP: Flächeninhalt von Figuren

```
;; Eine Figur ist entweder
;; - ein Rechteck oder
;; - ein Kreis
;; Name: shape

;; Flächeninhalt einer Figur berechnen
(check-within (shape-area (make-circle 1.0)) pi 0.01)
(check-within (shape-area (make-rectangle 2.5 3.0)) 7.5 0.01)
(define shape-area
  (lambda [s]
    (cond
      [(circle? s)    (circle-area s)]
      [(rectangle? s) (rect-area s)])))
```

Summentypen in SML

- Beispiel:

```
datatype mytype = TwoInts of int * int
                | Str of string
                | Pizza
```

- Datentyp `mytype` für „gemischte Daten“ mit drei Varianten
- Drei Konstruktoren: `TwoInts`, `Str`, `Pizza`
- Konstruktoren dienen zwei Zwecken:
  1. Er ist entweder eine Funktion für die Erzeugung von Werten des neuen Typs, wenn die Variante eine Komponente besitzt (erkennbar am Schlüsselwort `of`) oder er repräsentiert selbst einen Wert dieses Typs (im Beispiel oben: `Pizza`).
  2. Konstuktoren dienen für Fallunterscheidungen in sog. *case-expressions*, d. h. für Mustervergleiche.

Zugriff auf `datatype`-Werte

- ausschließlich durch Mustervergleich:

```
fun f x = (* f has type mytype->int *)
    case x of
        Pizza => 3
      | TwoInts(i1,i2) => i1 + i2
      | Str s => String.size s
```

- `case`-Ausdrücke stellen ausdrucksmächtigere Fallunterscheidungen dar (verglichen mit `if-then-else` oder `cond`).
- Zwei Teilausdrücke werden ausgewertet:
  1. der Ausdruck zwischen `case` und `of` (hier: `x`)

2. der erste Ausdruck hinter `of`, dessen Muster (der Teil vor `=>`) zu dem Wert von `x` passt. Beispiel: Wenn `TwoInts(7,9)` an `f` übergeben wird, wird der zweite Zweig ausgewählt. `f` liefert in diesem Fall 16.

```
fun f x = (* f has type mytype->int *)
    case x of
        Pizza => 3
      | TwoInts(i1,i2) => i1 + i2
      | Str s => String.size s
```

- Beachte: Da die Variante `TwoInts` aus zwei Integer-Werten besteht, muss (vorläufig) das Muster zwei Variablen (hier `i1, i2`) benutzen/deklarieren.
- Jeder Zweig muss den gleichen Ergebnistyp besitzen.
- Jeder Zweig hat die Form `muster => ausdruck`.
- Die Zweige werden durch `|` voneinander getrennt.
- Jedes Muster benutzt einen anderen Konstruktor.

Übertragung des Figur-Beispiels aus Racket

```
datatype shape =
         Rectangle of real * real
         | Circle of real

fun shape_area s =
    case s of
        Rectangle(w, h) => w * h
      | Circle(r) => 3.14159 * r * r

val ra = shape_area (Rectangle (3.0, 4.0))
val ca = shape_area (Circle 1.0)

(* Repl:*)
datatype shape = Circle of real | Rectangle of real * real
val shape_area = fn : shape -> real
val ra = 12.0 : real
val ca = 3.14159 : real
```

Listen als rekursive algebraische Datentypen

in Racket

```
; Eine Liste ist
; - die leere Liste oder
; - ein Paar
; Name: lst

; die leere Liste
(define empt 'empt)
(define empt? (lambda [l] (equal? l empt)))

; Ein Paar besteht aus
; - einem Wert
; - einer Liste
(define-struct lst [fst rst])

;; ein paar Umbenennungen:
(define cns make-lst)
(define fst lst-fst)
```

```
(define rst lst-rst)

(define list-sum
  (lambda (lis)
    (cond
      [(empt? lis) 0]
      [(lst? lis)
       (+ (fst lis)
          (list-sum (rst lis)))])))

(= 11 (list-sum (cns 1 (cns 7 (cns 3 empt)))))
```

in SML

- `datatype`-Definitionen dürfen rekursiv sein.
- Man beachte die Nutzung des Mustervergleichs in den Funktionen.

```
datatype my_int_list = Empty
                     | Cons of int * my_int_list

val one_two_three = Cons(1,Cons(2,Cons(3,Empty)))

fun append_mylist (xs,ys) =
    case xs of
        Empty => ys
      | Cons(x,xs') => Cons(x, append_mylist(xs',ys))
```

Mustervergleich für in SML eingebaute Listen

- Statt

```
fun sum_list (xs : int list) =
    if null xs
    then 0
    else hd(xs) + sum_list(tl xs)
```

- besser

```
fun sum_list xs =
    case xs of
        [] => 0
      | x::xs' => x + sum_list xs'
```

Vorteile des Mustervergleichs

- Laufzeitfehler wie durch `hd []` werden vermieden.
- Wenn in einem `case`-Ausdruck eine Variante fehlt, erzeugt der Compiler eine Warnung. Zur Laufzeit erzeugte eine fehlende Variante einen Programmabbruch.
- Falls eine Variante mehr als einmal verwendet wird, gibt der Compiler ebenfalls eine Warnung.

Polymorphe Datentypen

- die `append` Funktion für eingebaute Listen:

```
fun append (xs,ys) =
    case xs of
        [] => ys
      | x :: xs' => x :: append(xs',ys)

val append = fn : 'a list * 'a list -> 'a list
```

- Der eingebaute Listentyp ist polymorph (erkennbar am Typparameter `'a`).
- Polymorphe Datentypen erlauben generische Funktionen in statisch getypten Sprachen.
- Polymorph heißt **nicht** heterogen.

Selbsdefinierte polymorphe Datentypen

- Beispiel: Binärbaum, dessen interne Knoten Daten vom Typ `'a` und dessen Blätter Daten vom Typ `'b` enthalten:

```
datatype ('a,'b) tree = Node of 'a * ('a,'b) tree * ('a,'b) tree
                      | Leaf of 'b
```

- Damit sind Konkretisierungen wie `(int,int) tree` oder `(string,bool) tree` möglich.
- Die Art der Verwendung von Konstruktoren und die Nutzung des Mustervergleichs ist dieselbe für polymorphe und reguläre Datentypen.

Mustervergleich für Produkttypen und die Wahrheit über Funktionen

- Grundsätzlich ist der Mustervergleich auch auf Produkttypen anwendbar.
- Für ein Record der Form `{f1=v1,...,fn=vn}` werden durch das Muster `{f1=x1,...,fn=xn}` die Variablen `xi` an die Werte `vi` gebunden.
- Ein Muster der Form `(x1,...,xn)` bindet die Werte eines Tupels `(v1,...,vn)`, da die Schreibweise `(x1,...,xn)` nur syntaktischer Zucker für das Record `{1=x1,...,n=xn}` ist.
- Beispiel: Funktion, die die Elemente eines Tripels `int * int * int` summiert:

```
fun sum_triple (triple : int * int * int) =
    case triple of
        (x,y,z) => z + y + x
```

Andere (bessere) syntaktische Varianten

```
fun sum_triple (triple : int*int*int) =
    let val (x,y,z) = triple
    in x+y+z end

fun sum_triple (x,y,z) = x+y+z
```

- Die letzte Variante akzeptiert ein Tripel als Argument und bindet die Komponenten an die Variablen `x`, `y` und `z`.
- Diese Form unterscheidet sich nicht von der bisherigen Notation für Funktionen mit mehreren (hier drei) Parametern.
- Mit anderen Worten:

  Jede Funktion in ML besitzt nur **einen** Parameter.

Geschachtelte Muster

```
exception BadTriple

fun zip3 list_triple =
    case list_triple of
        ([],[],[]) => []
      | (hd1::tl1,hd2::tl2,hd3::tl3) => (hd1,hd2,hd3)::zip3(tl1,tl2,tl3)
      | _ => raise BadTriple

fun unzip3 lst =
```

```
    case lst of
        [] => ([],[],[])
      | (a,b,c)::tl => let val (l1,l2,l3) = unzip3 tl
                       in
                           (a::l1,b::l2,c::l3)
                       end
```

Exkurs: Destructuring in Clojure

Unter *destructuring* versteht man eine Weise, Werte aus einer Datenstruktur zu extrahieren und dabei an Variablen zu binden. Dabei muss die Datenstruktur nicht traversiert werden. Destructuring wird vorwiegend auf Vektoren und Maps angewendet, ist aber auch für Listen und Zeichenketten möglich.
(vgl. Destructuring in Clojure)

Vektoren

```
(def my-vector [:a :b :c :d])
(def my-nested-vector [:a :b :c :d [:x :y :z]])

(let [[a b c d] my-vector]
  (println a b c d))
;; => :a :b :c :d

(let [[a _ _ d [x y z]] my-nested-vector]
  (println a d x y z))
;; => :a :d :x :y :z

;;; Das Muster muss nicht den ganzen Vektor abdecken:

(let [[a b c] my-vector]
  (println a b c))
;; => :a :b :c
;;; ...

;;; Mit & the-rest kann der restliche Teil des Vektors an the-rest gebunden werden

(let [[a b & the-rest] my-vector]
  (println a b the-rest))
;; => :a :b (:c :d)

;;; Wenn das Muster mehr Elemente "verlangt" als im Vektor vorhanden", werden die
;;; überschüssigen Symbole an nil gebunden.

(let [[a b c d e f g] my-vector]
  (println a b c d e f g))
;; => :a :b :c :d nil nil nil

;;; Mit :as some-symbol als die beiden letzten Einträge im Muster wird der ganze Vektor
;;; an some-symbol gebunden

(let [[:as all] my-vector]
  (println all))
;; => [:a :b :c :d]

(let [[a :as all] my-vector]
  (println a all))
;; => :a [:a :b :c :d]

(let [[a _ _ _ [x y z :as nested] :as all] my-nested-vector]
```

```
  (println a x y z nested all))
;; => :a :x :y :z [:x :y :z] [:a :b :c :d [:x :y :z]]

;;; & the-rest und :as some-symbol können auch zusammen benutzt werden.

(let [[a b & the-rest :as all] my-vector]
  (println a b the-rest all))
;; => :a :b (:c :d) [:a :b :c :d]
```

Beispiel ohne Destructuring ...

```
(def my-line [[5 10] [10 20]])

(let [p1 (first my-line)
      p2 (second my-line)
      x1 (first p1)
      y1 (second p1)
      x2 (first p2)
      y2 (second p2)]
  (println "Line from (" x1 "," y1 ") to (" x2 ", " y2 ")"))
;; => "Line from ( 5 , 10 ) to ( 10 , 20 )"
```

... und mit Destructuring

```
(def my-line [[5 10] [10 20]])

(let [[p1 p2] my-line
      [x1 y1] p1
      [x2 y2] p2]
 (println "Line from (" x1 "," y1 ") to (" x2 ", " y2 ")"))
;= "Line from ( 5 , 10 ) to ( 10 , 20 )"
;; => "Line from ( 5 , 10 ) to ( 10 , 20 )"
```

Maps

```
(def my-hashmap {:a "A" :b "B" :c "C" :d "D"})
(def my-nested-hashmap {:a "A" :b "B" :c "C" :d "D" :q {:x "X" :y "Y" :z "Z"}})

(let [{a :a d :d} my-hashmap]
  (println a d))
;; => A D

(let [{a :a, b :b, {x :x, y :y} :q} my-nested-hashmap]
  (println a b x y))
;; => A B X Y

;;; Wenn ein Schlüssel in der Map nicht existiert, wird die Variable an nil gebunden.

(let [{a :a, not-found :not-found, b :b} my-hashmap]
  (println a not-found b))
;; => A nil B
;;; ...

;;; Für fehlende Schlüssel kann hinter dem Schlüsselwort :or eine Map mit Default-Werten
;;; angegeben werden.

(let [{a :a, not-found :not-found, b :b, :or {not-found ":)"}} my-hashmap]
  (println a not-found b))
;; => A :) B
```

```
;;; Die Form :as some-symbol ist auch für Maps verfügbar.

(let [{a :a, b :b, :as all} my-hashmap]
  (println a b all))
;; => A B {:a A :b B :c C :d D}

;;; Die Kombination von :as und :or ist auch möglich.

(let [{a :a, b :b, not-found :not-found, :or {not-found ":)"}, :as all} my-hashmap]
  (println a b not-found all))
;; => A B :) {:a A :b B :c C :d D}
```

Beispiel

```
(def client {:name "Super Co."
             :location "Philadelphia"
             :description "The worldwide leader in plastic tableware."})

;;; ohne Destructuring
;;;;;;;;;;;;;;;;;;;;;;
(let [name (:name client)
      location (:location client)
      description (:description client)]
  (println name location "-" description))
;; => Super Co. Philadelphia - The worldwide leader in plastic tableware.

;;; mit Destructuring
;;;;;;;;;;;;;;;;;;;;;
 (let [{name :name
       location :location
       description :description} client]
  (println name location "-" description))
;; => Super Co. Philadelphia - The worldwide leader in plastic tableware.

;;; und noch kürzer mit dem :keys Schlüsselwort
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
(let [{:keys [name location description]} client]
  (println name location "-" description))
;; => Super Co. Philadelphia - The worldwide leader in plastic tableware.
```

Funktionsköpfe

- Ein häufiger Anwendungszweck von Destructuring ist die Zerlegung von Argumenten, die an eine Funktion übergeben werden.

```
;;; "klassisch" ohne Destructuring
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
(defn print-coordinates-1 [point]
  (let [x (first point)
        y (second point)
        z (last point)]
    (println "x:" x ", y:" y ", z:" z)))

;;; mit Destructuring unter Verwendung von let
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
(defn print-coordinates-2 [point]
  (let [[x y z] point]
    (println "x:" x ", y:" y ", z:" z)))

;;; Anwendung von Destructuring auf die Parameterliste
```

```
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
(defn print-coordinates-3 [[x y z]]
  (println "x:" x ", y:" y ", z:" z))
```

Curryfizierung – partielle Anwendung von Funktionen

- ML-Funktionen akzeptieren nur ein Argument.
- Wege, um die Übergabe mehrerer Argumente zu „simulieren“:
  - Verwendung von Tupeln (s. diverse Beispiele oben)
  - Verwendung einer Funktion, die das erste Argument verarbeitet und mit einer Funktion antwortet, die dann das nächste Argument verarbeitet und so weiter

    Diese Technik wird nach dem Logiker Haskell Curry als *Currying* bezeichnet. Das Verb (englisch) heißt *currify*, im Deutschen mit *curryfizieren* bezeichnet.

Definition und Benutzung einer curryfizierten Funktion

- Beispiel einer Funktion mit „drei Parametern”:

  ```
  val sorted3 = fn x => fn y => fn z => z >= y andalso y >= x
  ```

- Der Aufruf `sorted3 4` liefert eine Funktion, in deren lokaler Umgebung `x` definiert ist.
- Rufen wir diese Funktion mit dem Argument 5 auf, erhalten wir eine Funktion, in deren lokaler Umgebung `x` und `y` definiert sind.
- Wird diese Funktion schließlich mit 6 aufgerufen, erhalten wir `true`.
- So kann der Ausdruck `((sorted3 4) 5) 6` nahezu wie der Aufruf einer Funktion mit drei Argumenten angesehen werden.
- Die Klammern sind optional: `sorted3 4 5 6`.
- Zum Vergleich die Tupel-Version

  ```
  fun sorted3_tupled (x,y,z) = z >= y andalso y >= x
  ```

  muss so aufgerufen werden:

  ```
  sorted3_tupled(4,5,6)
  ```

- Der Ausdruck `e1 e2 e3 e4` ist äquivalent zu `(((e1 e2) e3) e4)`.

Syntaktischer Zucker für die Definition von curryfizierten Funktionen

- Die Definition von curryfizierten Funktionen mithilfe geschachtelter Lambda-Ausdrücke wie in

  ```
  val sorted3 = fn x => fn y => fn z => z >= y andalso y >= x
  ```

  ist etwas unhandlich und kann ersetzt werden durch:

  ```
  fun sorted3 x y z = z >= y andalso y >= x
  ```

- Man beachte die unterschiedlichen Typen von `sorted3_tupled` und `sorted3`:

  ```
  val sorted3_tupled = fn : int * int * int -> bool
  val sorted3 = fn : int -> int -> int -> bool
  ```

Partielle Anwendung von curryfizierten Funktionen

- In der Regel wird die Funktion `sorted3` wohl mit allen drei Argumenten aufgerufen werden.
- Ein Aufruf mit weniger Argumenten wird als *partielle Anwendung* bezeichnet.
- Ein (nicht besonders nützliches) Beispiel:
  `sorted3 0 0` liefert eine Funktion, die `true` liefert, wenn ihr Argument nicht negativ ist.

Partielle Anwendung und Funktionen höherer Ordnung

- Curryfizierung wird gerne im Zusammenhang mit den klassischen Funktionen höherer Ordnung benutzt.
- Beipiel: Ein curryfizierte Version der Funktion `fold` für Listen

```
fun fold f acc xs =
    case xs of [] => acc
             | x::xs' => fold f (f(acc,x)) xs'
```

- Eine Funktion, die die Summe der Elemente berechnet, könnte nun statt in der bekannten Art

  ```
  fun sum1 xs = fold (fn (x,y) => x+y) 0 xs
  ```

  unter Nutzung von partieller Anwendung so definiert werden:

  ```
  val sum2 = fold (fn (x,y) => x+y) 0
  ```

Diverse Funktionen höherer Ordnung in ML nutzen Curryfizierung

- Beispiele (aus dem Modul `List`):

  ```
  val List.map = fn : ('a -> 'b) -> 'a list -> 'b list
  val List.filter = fn : ('a -> bool) -> 'a list -> 'a list
  val List.foldl = fn : ('a * 'b -> 'b) -> 'b -> 'a list -> 'b
  ```

- Der Ausdruck `List.foldl((fn (x,y) => x+y), 0, [3,4,5])` führt zu einem Typfehler, weil `List.foldl` eine Funktion vom Typ
  `'a * 'b -> 'b` und kein Tripel erwartet.
- Der korrekte Aufruf muss lauten:
  `List.foldl (fn (x,y) => x+y) 0 [3,4,5]`
  Er ruft `List.foldl` mit einer Funktion auf, die eine Funktion liefert und so weiter.
- Ein anderes Beispiel einer nützlichen curryfizierten Funktion ist `List.exists`, die wie folgt implementiert werden kann:

```
fun exists predicate xs =
    case xs of
        [] => false
      | x::xs' => predicate x orelse exists predicate xs'
```

Curryfizierung im Allgemeinen

- Curryfizierung ist nicht nur im Zusammenhang von Funktionen höherer Ordnung von Bedeutung.
- Curryfizierung ist interessant für jede Funktion mit mehr als einem Parameter und führt oft zu prägnanter Ausdrucksweise.
- Im folgenden Beispiel sind `zip` und `range` curryfiziert, `countup` wendet `range` partiell an und `add_numbers` verwandelt die Liste `[v1,v2,...,vn]` in `[(1,v1),(2,v2),...,(n,vn)]`.

```
fun zip xs ys =
    case (xs,ys) of
        ([],[]) => []
      | (x::xs',y::ys') => (x,y) :: (zip xs' ys')
      | _ => raise Empty

fun range i j = if i > j then [] else i :: range (i+1) j
val countup  = range 1
fun add_numbers xs = zip (countup (length xs)) xs
```

Curryfizierung in Lisp-Sprachen

- in Lisp-Sprachen keine syntaktische Unterstützung
- gilt jedenfalls für Racket und Clojure
- Curryfizierung aber durch Nutzung von Funktionen höherer Ordnung möglich

Beispiele in Clojure

Funktion, die eine Funktion erzeugt, die eine Konstante addiert

```
;; make-add: number -> (number -> number)
(def make-add
  (fn [a]
    (fn [b]
      (+ a b))))
```

- Anwendung

```
(def add-5 (make-add 5))
(add-5 7) ;; => 12

;; oder ohne Hilfsdefintion
((make-add 5) 7) ;; => 12
```

- Im Gegensatz zur Standardfunktion `+` akzeptiert die Funktion `make-add` nicht beide Summanden als Argumente, sondern zunächst nur einen. Das Resultat der Anwendung ist dann eine Funktion, die wiederum einen (den zweiten) Summanden akzeptiert und auf den ersten addiert.

Funktion, die eine Funktion erzeugt, die an eine Liste ein Element vorn anhängt

```
;; make-prepend: X -> ((list-of X) -> (list-of X))

(def make-prepend
  (fn [a]
    (fn [b]
      (cons a b))))

(= ((make-prepend 5) (list 1 2 3)) (list 5 1 2 3))
(= ((make-prepend "A") (list "B" "C" "D")) (list "A" "B" "C" "D"))
```

ein ML-Beispiel

```
Standard ML of New Jersey v110.84 [built: Tue Sep 04 08:31:43 2018]
- fun make_mult a = fn b => a * b;
val make_mult = fn : int -> int -> int
- make_mult 3;
val it = fn : int -> int
- it 5;
val it = 15 : int
```

Der Schönfinkel-Isomorphismus

- Die oben gezeigten Funktionen folgen alle dem gleichen Muster.
- Kann dieses Muster verallgemeinert, d. h. in eine Funktion abstrahiert werden?
- Die Funktion `curry` akzeptiert eine Funktion mit zwei Argumenten und liefert eine curryfizierte Variante zurück.
- Die Funktion `uncurry` für die inverse Operation durch.

Die Funktion `curry`

```
;; Prozedur mit zwei Parametern curryfizieren
;; curry: (X Y -> Z) -> (X -> (Y -> Z))

(def curry
  (fn [proc]
    (fn [a]
      (fn [b]
        (proc a b)))))

(= (((curry +) 5) 7) 12)
(= (((curry *) 5) 7) 35)
(= (((curry cons) 5) (list 1 2 3)) (list 5 1 2 3))

;; Definition einer currifyfizierten Funktion
(def make-mult (curry *))
((make-mult 3) 5) ;;=> 15
```

Die Funktion `uncurry`

```
;; curryyfizierte Funktion entcurryfizieren
;; uncurry (X -> (Y -> Z)) -> (X Y -> Z)

(def uncurry
  (fn [proc]
    (fn [a b]
      ((proc a) b))))

(= ((uncurry make-add) 5 7) 12)
(= ((uncurry make-prepend) 5 (list 1 2 3)) (list 5 1 2 3))
```

Der Isomorphismus

- Es gilt folgende Gleichung für Funktionen f mit zwei Parametern:

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

- Z. B. liefert der Ausdruck

```
(= ((uncurry (curry cons)) "A" (list "B" "C" "D")) (list "A" "B" "C" "D"))
```

`true`.