

## Kapitel 7

# Polymorphie und Überladung

Der Wechsel der Repräsentation ist bei größeren Programmen eine häufige Operation: Vorläufige, schnell „hingehackte“ Datenstrukturen werden durch endgültige ersetzt, oder während der Programmentwicklung stellt sich heraus, daß eine andere Repräsentation schneller, kürzer, weiter oder höher funktioniert als die vorhandene. Es ist dabei wünschenswert, daß der Repräsentationswechsel möglichst schmerzfrei vor sich geht: Im Idealfall sollte bestehender Code nicht geändert werden müssen, um mit der neuen Repräsentation auszukommen. Handwerkszeug zum Schreiben von Code, der repräsentationsunabhängig ist, sind *Polymorphie* und *Überladung*.

### 7.1 Quote

Bevor es zum eigentlichen Thema des Kapitels geht sei hier noch ein weiteres Sprachelement von Scheme „außer der Reihe“ dargestellt. Bisher gab es grob zwei Sorten von Literalen: Die erste Gruppe wird von den Werten gebildet, bei denen Literal und externe Repräsentation übereinstimmen:

```
> 5
5
> #t
#t
```

Die andere Gruppe wird von Werten gebildet, bei denen Literal und externe Repräsentation sich durch ein Apostroph unterscheiden:

```
> '()
()
> 'foo
foo
```

Zumindest beim letzten Beispiel ist klar, warum das Apostroph (das *Quote*) sein muß: `foo` hat eine andere Bedeutung als `'foo`. (Ersteres bezeichnet den Wert der Variablen `foo`, letzteres das Symbol `foo`.) Interessanterweise funktioniert das *Quote* auch für die erste Gruppe:

```
> '5
5
> '#t
#t
```

Das heißt, daß das *Quote* eigentlich immer bei Literalen stehen muß, bis auf einige Fälle, in denen keine Mißverständnisse durch Weglassen des *Quote* möglich

sind. (Manchmal werden diese Literale auch *selbst-quotierend* genannt.) Nun gibt es aber noch andere Werte, welche Repräsentationen haben, die bisher noch kein Gegenstück als Literale hatten:

```
> (cons 5 7)
(5 . 7)
> (cons 1 (cons 2 (cons 3 '())))
(1 2 3)
```

Aber auch für diese funktioniert Quote:

```
> '(5 . 7)
(5 . 7)
> '(1 2 3)
(1 2 3)
```

Allgemein erlaubt Quote die Konstruktion von Literalen für die sogenannten *repräsentierbaren Werte*. Die repräsentierbaren Werte bilden eine induktiv definierte Menge:

- Zahlen und Booleans sind repräsentierbar.
- Die leere Liste und Symbole sind repräsentierbar.
- Ein Paar aus zwei repräsentierbaren Werten ist repräsentierbar.
- Nichts sonst ist (zumindest nach dem momentanen Kenntnisstand) repräsentierbar.

Vor die Repräsentation eines repräsentierbaren Wertes wird durch Voranstellung eines Quote ein Literal, das diesen Wert produziert.

Das Quote hat noch eine weitere Eigenheit:

```
> '()'
'()
```

Dieses Literal bezeichnet nicht die leere Liste (dann käme nur () heraus), sondern ein anderes Biest:

```
> (pair? '())
#t
> (car '())
quote
> (cdr '())
(())
```

Das heißt aber, das '(thing) äquivalent ist zu (quote (thing)):

```
> (equal? (quote ()) '())
#t
> (equal? (quote (quote ())) '())
#t
```

Diese Übersetzung von ' nach (quote ...) wird schon direkt beim Einlesen eines Programms in das Scheme-System vorgenommen.

Und noch eine Bemerkung: read wendet beim Umwandeln einer Repräsentation in einen Wert dieselben Regeln an wie quote.

## 7.2 Polymorphie

Zurück zum Thema: Im Nim-Spiel ist der Repräsentationswechsel einfach, da um die eigentlichen Repräsentationen noch eine zusätzliche Abstraktionsschicht gewickelt wurde:

```
(define move-type (make-type 'move))

(define make-move
  (lambda (pile n-coins)
    (make-typed-object move-type (cons pile n-coins))))

(define move? (typed-object-predicate move-type))

(define move-pile
  (lambda (move)
    (car (typed-object-value move-type move))))

(define move-n-coins
  (lambda (move)
    (cdr (typed-object-value move-type move))))
```

Durch das kleine selbstprogrammierte Typsystem wird dafür gesorgt, daß Werte verschiedener Typen nicht durcheinander kommen können: Die Operation `move-pile` beispielsweise funktioniert nur auf Werten des Typs `move`. Eine solche Prozedur heißt *monomorph*. Das andere Extrem wird durch Prozeduren wie die `identity` repräsentiert:

```
(define identity
  (lambda (x)
    x))
```

`Identity` funktioniert auf *beliebigen* Werten und ist damit eine *polymorphe* Prozedur. In anderen Prozeduren versteckt sich die Polymorphie in Datenstrukturen. So funktioniert z.B. `length` auf Listen, aber die Typen der Elemente der Listen sind `length` völlig gleichgültig. (In diesem Sinne sind strenggenommen auch `move-pile` und `move-n-coins` polymorph, da nirgendwo garantiert wird, daß `move`-Werte tatsächlich aus zwei Zahlen bestehen.)

Polymorphie in diesen Beispielen bedeutet einerseits, daß polymorphe Prozeduren Werte verschiedener Typen akzeptieren, aber auch, daß sie Werte verschiedener Typen gleichartig behandeln. Manchmal aber ist es wünschenswert, das eine ohne das andere zu haben, also Prozeduren, die zwar auf Werten verschiedener Typen funktionieren, aber die sich je nach Typ unterschiedlich verhalten. Solche Prozeduren, die also ggf. unter einem einzigen Namen mehrere Operationen verstecken, heißen *überladen*.

## 7.3 Überladung

Ein einfaches Beispiel für Überladung (in manchen Kreisen auch als *Ad-hoc-Polymorphie*, in anderen als *Generizität* bekannt) sind Mengen — für Mengen gibt es eine ganze Reihe von Repräsentationen, die sich anbieten: Listen, sortierte Folgen und Suchbäume. Schön wäre es, wenn es nicht notwendig wäre, daß der Programmierer sich die nicht die ganzen speziellen Namen für die Mengenoperationen merken müßte.

Hier also Gerippe der Prozeduren für einen „Wunsch-Mengen-ADT“:

```

(define make-empty-set
  (lambda (< =)
    ...))

(define set?
  (lambda (thing)
    ...))

(define adjoin-set
  (lambda (element set)
    ...))

(define remove-set
  (lambda (element set)
    ...))

(define set-member?
  (lambda (element set)
    ...))

```

Nun sind Prozeduren denkbar, welche Mengen-ADTs mit unterschiedlichen Repräsentationen realisieren: Konstruktoren `make-empty-list-set`, `make-empty-sorted-sequence-set` und `make-empty-search-tree-set`, Prädikate `list-set?`, `sorted-sequence-set?`, `search-tree-set?` sowie Prozeduren `adjoin-sorted-sequence-set`, etc. (Die Namen dieser Prozeduren entsprechen nicht den bisher verwendeten, lassen sich aber trivial in diese überführen.)

In einem Programm mögen jetzt viele Mengen manipuliert werden. Die Repräsentation kann sich dabei während der Entwicklungszeit des Programms ändern. Es ist sogar möglich, daß Mengen unterschiedlicher Repräsentation in einem Programm gleichzeitig vorkommen. Die Entscheidung, welche Repräsentation eine Menge hat, muß dabei bei der Erzeugung fallen. Ab da wäre es allerdings gut, wenn die gleichen Prozeduren für Mengen unterschiedlicher Repräsentation funktionieren würden. Die offensichtliche Art dies zu realisieren ist es, die Prädikate zu benutzen, um die Mengen unterschiedlicher Repräsentation zu unterscheiden:

```

(define set?
  (lambda (thing)
    (or (list-set? thing)
        (sorted-sequence-set? thing)
        (search-tree-set? thing))))

(define adjoin-set
  (lambda (element set)
    ((cond
      ((list-set? set) adjoin-list-set)
      ((sorted-sequence-set? set) adjoin-sorted-sequence-set)
      ((search-tree-set? set) adjoin-search-tree-set))
     element set)))

(define remove-set
  (lambda (element set)
    ((cond
      ((list-set? set) remove-list-set)
      ((sorted-sequence-set? set) remove-sorted-sequence-set)
      ((search-tree-set? set) remove-search-tree-set))
     element set)))

```

```

    element set)))

(define set-member?
  (lambda (element set)
    ((cond
      ((list-set? set) list-set-member?)
      ((sorted-sequence-set? set) sorted-sequence-set-member?)
      ((search-tree-set? set) search-tree-set-member?)
      element set))))

```

Diese Art der Programmierung heißt *datengesteuerte* Programmierung und erlaubt die Konstruktion mächtiger Abstraktionen. Besonders elegant erscheint sie aber nicht aus mehreren Gründen:

- Jede dieser Prozeduren enthält letztlich den gleichen `cond`-Ausdruck.
- Die Erweiterung des ADT um weitere Repräsentationen erfordert die Änderung existierender Prozeduren.

Das erste Problem läßt sich die Einführung einer Abstraktion wie der folgenden mildern, aber nicht wirklich lösen:

```

(define choose-set-op
  (lambda (set
          list-set-op sorted-sequence-set-op search-tree-set-op)
    (cond
      ((list-set? set) list-set-op)
      ((sorted-sequence-set? set) sorted-sequence-set-op)
      ((search-tree-set? set) search-tree-set-op))))

```

Mit der Hilfe von `choose-set-op` lassen sich Operationen wie `set-member?` geringfügig weniger geschwätzig gestalten:

```

(define set-member?
  (lambda (element set)
    ((choose-set-op set
                    list-set-member?
                    sorted-sequence-set-member?
                    search-tree-set-member?)
     element set)))

```

Das zweite Problem bleibt bestehen. Die Ursache des Problems entspringt folgender Beobachtung: *Die Prozeduren bestimmen die Operationen, nicht die Daten*. Glücklicherweise sind Prozeduren in Scheme auch Daten, es läßt sich also ein Rollentausch vornehmen:

```

(define list->list-set
  (lambda (= list)
    (lambda (message)
      (cond
        ((equal? 'adjoin message)
         (lambda (element)
           (if (member? = element list)
               (list->list-set = list)
               (list->list-set = (cons element list))))))
        ((equal? 'remove message)
         (lambda (element)

```

```

      (list->list-set =
        (remove = element list))))
    ((equal? 'member? message)
     (lambda (element)
       (member? = element list))))))

(define member?
  (lambda (= element list)
    (cond
      ((null? list) #f)
      ((= element (car list)) #t)
      (else (member? = element (cdr list))))))

(define remove
  (lambda (= element list)
    (cond
      ((null? list) '())
      ((= element (car list)) (cdr list))
      (else (cons (car list)
                   (remove = element (cdr list))))))

(define make-empty-list-set
  (lambda (=)
    (list->list-set = '())))

```

Diese Mengen benutzen immer noch eine Liste für die Repräsentation, verstecken diese aber in einer Prozedur. Diese Prozedur akzeptiert einen einzelnen Parameter, eine *Nachricht*, welche eine Operation auswählt und diese als Prozedur zurückliefert. Dieser Programmierstil nennt sich *Message-Passing Style*. Die ursprünglichen list-set-Prozeduren lassen sich nun rekonstruieren:

```

(define adjoin-list-set
  (lambda (element set)
    ((set 'adjoin) element)))

(define remove-list-set
  (lambda (element set)
    ((set 'remove) element)))

(define list-set-member?
  (lambda (element set)
    ((set 'member?) element)))

```

Interessanterweise werden bei der Realisierung dieser Prozeduren gerade Operator und Operand vertauscht.

Das Prinzip läßt sich andere Repräsentationen übertragen. Eine weitere Repräsentation für Mengen basiert auf der charakteristischen Funktion (in Gestalt eines Prädikats) der Menge:

```

(define make-predicate-set
  (lambda (= predicate)
    (lambda (message)
      (cond
        ((equal? 'adjoin message)
         (lambda (element)

```

```

      (make-predicate-set
      =
      (lambda (thing)
        (or (= element thing)
            (predicate thing))))))
    ((equal? 'remove message)
     (lambda (element)
       (make-predicate-set
        =
        (lambda (thing)
          (if (= element thing)
              #f
              (predicate thing))))))
    ((equal? 'member? message)
     (lambda (element)
       (predicate element))))))

(define make-empty-predicate-set
  (lambda (=)
    (make-predicate-set = (lambda (thing) #f))))

```

Die einzelnen Operationen sehen nun so aus:

```

(define adjoin-predicate-set
  (lambda (element set)
    ((set 'adjoin) element)))

(define remove-predicate-set
  (lambda (element set)
    ((set 'remove) element)))

(define predicate-set-member?
  (lambda (element set)
    ((set 'member?) element)))

```

Aber die Definitionen dieser Prozeduren sind *identisch* mit denen von `list-set` weiter oben! Wären also alle Repräsentationen von Mengen durch derartige Prozeduren realisiert, die Nachrichten `adjoin`, `remove` und `member?` akzeptieren, so ließe sich einfach schreiben:

```

(define adjoin-set
  (lambda (element set)
    ((set 'adjoin) element)))

(define remove-set
  (lambda (element set)
    ((set 'remove) element)))

(define set-member?
  (lambda (element set)
    ((set 'member?) element)))

```

Es bleibt ein kleines Problem, nämlich das des Prädikats. Dieses läßt sich nicht durch eine Nachricht realisieren, da folgende naheliegende Definition nicht funktionieren kann:

```
(define set?
  (lambda (thing)
    ((thing 'set?))))
```

Diese Fassung von `set?` muß für beliebige Werte funktionieren, produziert aber bei allem, was keine einstellige Prozedur ist, eine Fehlermeldung und beendet das Programm. Das Problem läßt sich aber dadurch lösen, daß die `set`-Werte mit dem `typed-object`-Mechanismus mit einem Typ `set` versehen werden.