

## 8.5 Mutierbare Datenstrukturen

`set!` verändert Bindungen der Umgebungsstruktur. Manchmal ist es jedoch auch wünschenswert, Datenstrukturen zu modifizieren. Für Paare stelle Scheme dafür die Prozeduren `set-car!` und `set-cdr!` zur Verfügung:

```
(define p1 (cons 23 42))
> p1
(23 . 42)
(set-car! p1 19)
> p1
(19 . 42)
(set-cdr! p1 22)
> p1
(19 . 22)
```

Diese Prozeduren erlauben es, Datenstrukturen mit Paaren zu konstruieren, die sich ohne sie nicht herstellen lassen:

```
(define p2 (list 'a 'b 'c))
(set-cdr! (cdr (cdr p2)) p2)
> p2
(a b c a b c a b c a b c a ...)
```

Damit enthält `p2` einen *Zyklus*.

Paare haben damit, aufgefaßt als abstrakter Datentyp, neben dem Konstruktor `cons` und den Selektoren `car` und `cdr` auch noch zwei sogenannte *Mutatoren* — `set-car!` und `set-cdr!`.

Tatsächlich bieten `set-car!` und `set-cdr!` jedoch nichts neues, lassen sich doch Paare inklusive dieser beiden Operationen auch in Message-Passing Style realisieren: Ein Paar wird als Prozedur repräsentiert, die eine Nachricht — eins der Symbole `car`, `cdr`, `set-car!`, `set-cdr!` — als Parameter akzeptiert und eine Prozedur zurückgibt, welche die entsprechende Operation ausführt:

```
(define make-pair
  (lambda (car cdr)
    (lambda (message)
      (cond
        ((equal? 'car message) (lambda () car))
        ((equal? 'cdr message) (lambda () cdr))
        ((equal? 'set-car! message)
         (lambda (new-car)
           (set! car new-car)))
        ((equal? 'set-cdr! message)
         (lambda (new-cdr)
           (set! cdr new-cdr)))))))

(define p3 (make-pair 5 17))
> ((p3 'car))
5
> ((p3 'cdr))
17
> ((p3 'set-car!) 23)
> ((p3 'car))
23
```

Mutation läßt sich also durch Zuweisung modellieren.

## 8.6 Sharing und Identität

Abschnitt 8.4 äußert sich in einer Frage nicht ganz präzise: Wenn eine Variable an eine Prozedur gebunden ist, was steht dann eigentlich auf der rechten Seite der Bindung? Die naheliegende Interpretation ist, daß sich dort die Closure befindet. Die Diagramme suggerieren allerdings, daß sich nicht die Closure selbst im Frame befindet, sondern nur ein Pfeil, der auf die Closure zeigt — ein sogenannter *Zeiger*. Dies ist ganz entscheidend beim Verständnis des Unterschiedes zwischen den Definitionspaaren

```
(define w1 (make-withdraw 100))
(define w2 (make-withdraw 100))
```

und

```
(define w1 (make-withdraw 100))
(define w2 w1)
```

Im ersten Fall werden zwei Closures angelegt, im zweiten Fall wird nur eine Closure angelegt, auf die dann zwei Zeiger existieren, gebunden an `w1` und `w2`. Da mehrere Zeiger auf das gleiche Objekt zeigen, wird die Closure zwischen `w1` und `w2` *gesharet*, das Phänomen heißt *Sharing*. (Auch hier hat sich leider kein deutsches Wort durchsetzen können.)

Sharing ist manchmal ein nützliches Werkzeug, birgt aber — wie alles, was mit Zuweisungen zu tun hat — Gefahren: Änderungen an geshareten Datenstrukturen haben Auswirkungen, die u.U. weit entfernt vom Ursprungsort auftreten und damit schwer nachvollziehbar sind.

Wie ist es mit anderen Datentypen in Scheme, was das Sharing betrifft? Paare sind wie Closures, es werden also ebenfalls ausschließlich Zeiger herumgereicht:

```
(define f
  (lambda (x)
    (set-car! x 27)))
(define p1 (cons 13 23))
(f p1)
> p1
(27 . 23)
```

Wenn also manche Werte Zeiger sind, wie lassen sich dann Werte daraufhin vergleichen, daß es Zeiger auf dieselben Objekte sind? `Equal?` reicht dafür offenbar nicht:

```
(define p2 (cons 13 17))
(define p3 (cons 13 17))
> (equal? p2 p3)
#t
(set-car! p2 23)
> p2
(23 . 17)
> p3
(13 . 17)
```

Scheme bietet für diesen Zweck („Zeiger-Gleichheit“) die Prozedur `eq?`:

```
(define p4 (cons 13 17))
(define p5 (cons 13 17))
> (equal? p4 p5)
```

```
#t
> (eq? p4 p5)
#f
(define p6 p5)
> (eq? p5 p6)
#t
```

Interessanterweise ist Sharing bei Zahlen gleichen Werts nicht garantiert:

```
> (eq? 23742374234 23742374234)
#f
```

Dafür sind aber Symbole mit gleichem Namen garantiert gesharet:

```
> (eq? 'anna-frieda 'anna-frieda)
#t
```

Dieser Unterschied ist — außer bei Effizienzüberlegungen — allerdings nicht besonders relevant, da es bei Symbolen keine Mutatoren gibt. (Dies unterscheidet Symbole von Zeichenketten.)

## 8.7 Eindeutige Repräsentationen für Typen

Zuweisungen und `eq?` können uns dabei dienen, ein altes, lästiges Problem zu lösen: Die Abstraktion für Typen aus Kapitel 5 hatte die Aufgabe, verschiedene Sorten von Werten durch einen Typ, der in den Werten mitgeführt wurde, zu unterscheiden. Dazu wurden getypte Werte durch Paare aus dem Typ und dem eigentlichen Wert repräsentiert.

Leider hat die Abstraktion aus Kapitel 5 ein wesentliches Problem: Typen werden dort lediglich durch ihren Namen (ein Symbol) repräsentiert. Damit sind verschiedene Typen gleichen Namens nicht unterscheidbar, und damit ist die Abstraktion anfällig für Namensverwechslungen und böswillige Attacken. Herzstück des Problems war der Konstruktor `make-type`:

```
(define make-type
  (lambda (name)
    name))
```

`make-type` gibt bei gleichem Namen auch immer das gleiche Ergebnis zurück. (Sogar dasselbe im Sinne von `eq?`, da Symbole eindeutig bestimmt sind.) Für eine Lösung der Aufgabe muß sich das ändern:

```
(define make-type
  (let ((type-id 0))
    (lambda (name)
      (set! type-id (+ 1 type-id))
      (cons type-id name))))
```

Damit ist die Typ-Abstraktion dagegen geschützt, daß jemand versehentlich für zwei verschiedene Typen `make-type` mit demselben Namen aufruft. Leider sind Typen immer noch „von Hand“ fälschbar:

```
(define t1 (make-type 'ernie))
(define t2 (cons 1 'ernie))
(define p1 (typed-object-predicate t1))
(define o1 (make-typed-object t2 'attaque!))
> (p1 o1)
#t
```

Auch dieser Defekt läßt sich durch Ersetzung von `equal?` durch `eq?` in `typed-object-value` und `typed-object-predicate` beheben:

```
(define typed-object-value
  (lambda (type object)
    (if (eq? type (car object))
        (cdr object)
        (error "type mismatch"))))

(define typed-object-predicate
  (lambda (type)
    (lambda (value)
      (and (pair? value)
           (eq? type (car value))))))
```

Diese Prozeduren erkennen den Unterschied zwischen Original und Fälschung:

```
(define t1 (make-type 'ernie))
(define t2 (cons 1 'ernie))
(define p1 (typed-object-predicate t1))
(define o1 (make-typed-object t2 'attacke!))
> (p1 o1)
#f
```