Kapitel 8

Zustand, Objekte und Zuweisungen

Die Programme der vergangenen Kapitel haben sich damit beschäftigt, Bestandteile des Modells eines Programms durch Werte abgebildet werden können, und wie diese Werte durch Prozeduren in Beziehung gesetzt werden können. Werte wandern dabei durch das laufende Programm und werden in der Berechnung und Konstruktion neuer Werte verwendet.

Das Hauptaugenmerk bei der Programmierung liegt dabei darauf, Programme *modular* zu gestalten: Programmteile müssen, so weit wie möglich, unabhängig voneinander sein. Das heißt, daß sie sich unabhängig voneinander verstehen und vor allen Dingen ändern lassen. Das Mittel zur Erreichung dieses Ziel ist die *Abstraktion*: Bestandteile des Modells werden durch Abstraktionen dargestellt, die verstecken, wie die darunterliegenden Programmteile funktionieren.

Ein Weg zu effektiven Abstraktionen ist, sie so zu konstruieren, daß sie tatsächlichen Bestandteilen des Modells entsprechen. Das ist inbesondere bei Programmen sinnvoll, die physikalische Modelle realisieren: Jedem Objekt des physikalischen Modells entspricht ein Wert des laufenden Programms. Nun ist es so, daß Objekte der physikalischen Realität einen *Zustand* haben: sie können sich über die Zeit hinweg verändern.

In den bisherigen Programmen gab es keine Werte, die sich während des Ablaufs eines Programms verändern konnten: sie konnten nur für die Konstruktion neuer Werte dienen. Es fehlte bisher in der Tat ein sprachliches Mittel, um existierende Werte verändern zu können. Dieses Mittel — genannt *Zuweisung* — wird in diesem Kapitel eingeführt. Die Einführung der Zuweisung, so einfach sie aussehen mag, hat tiefgreifende Auswirkungen auf den Begriff des Programms an sich: das bisheriges Substitutionsmodell für die Programmausführung funktioniert nicht mehr, und Beweise über die Korrektheit von Programmen werden um eine Größenordnung schwieriger. Die Einführung des Zustandsbegriffs zieht also einen ganzen Rattenschwanz an Komplikationen mit sich. Gelegentlich ist er diesen Preis wert, oft jedoch auch nicht.

8.1 Zustandsvariablen

Ein klassisches Beispiel für ein Objekt der realen Welt, das seinen Zustand verändert, ist ein Bankkonto. Das Konto soll repräsentiert werden durch eine Prozedur `withdraw`, die Geld vom Konto abhebt, und den noch verbleibenden Geldbetrag auf dem Konto zurückgibt. Das Konto habe für den Anfang 100 Mark Guthaben:

```
> (withdraw 25)
75
> (withdraw 25)
50
> (withdraw 60)
not-enough-money
> (withdraw 15)
35
```

Entscheidend ist, daß `withdraw` in identischem Kontext, mit identischen Werten für den Parameter, *unterschiedliche* Werte zurückgeben kann: `(withdraw 25)` liefert einmal 75, ein anderes Mal 50. Das ist neu: bisher verhielten sich Prozeduren wie mathematische Funktionen, was heißt: gleiche Eingabe, gleiche Ausgabe. Dieses Prinzip wird von `withdraw` verletzt.

Für die Realisierung von `withdraw` wird zunächst eine Variable `balance` definiert, welche an den Geldbetrag auf dem Konto gebunden ist:

```
(define balance 100)
```

Der Wert von `balance` läßt sich mit einer neuen syntaktischen Form verändern:

```
(set! balance 75)
> balance
75
```

(Der Rückgabewert von `set!` ist unsichtbar.) `Set!` hat folgende allgemeine Form:

⟨expression⟩ ::= ⟨assignment⟩
⟨assignment⟩ ::= `(set!` ⟨variable⟩ ⟨expression⟩`)`

Bei der Auswertung von `set!` wird der Ausdruck ausgewertet, und ihr Wert wird zum neuen Wert der Variablen. Dieser Vorgang heißt *Zuweisung*. Damit hat `set!` (ähnlich wie `display` und `newline`) neben einem Wert auch einen *Effekt*. `Set!` läßt sich nun verwenden, um `withdraw` zu realisieren:

```
(define withdraw
  (lambda (amount)
    (if (>= balance amount)
        (begin
          (set! balance (- balance amount))
          balance)
        'not-enough-money)))
```

Der Ausdruck

```
(set! balance (- balance amount))
```

berechnet also den neuen Kontostand `(- balance amount)` und weist dann diesen an `balance` zu.

`Withdraw` hat ein pragmatisches Problem: `balance` ist eine globale Variable und damit von überall zugänglich. Das ist das Äquivalent dazu, das eigene Konto in einer offenen Kiste außen an der Haustür zu verwalten. Mit

```
(set! balance 1000000)
```

könnte sich jeder selbst reich machen, oder jemand anders könnte den Kontoinhaber mit:

```
(set! balance 0)
```

arm machen. Besser wäre es also, `balance` vor Zugriff von außerhalb `withdraw` zu schützen. `Balance` wird dann zu einer lokalen Variable von `withdraw`:

```
(define withdraw
  (let ((balance 100))
    (lambda (amount)
      (if (>= balance amount)
          (begin
            (set! balance (- balance amount))
            balance)
          'not-enough-money))))
```

Damit sitzt der Kontostand — der Zustand des Kontos — in einer lokalen Variable von `withdraw` und ist damit von außen nicht mehr direkt zugänglich. Etwas unbefriedigend an `withdraw` ist, daß es nur ein einzelnes Konto verwaltet. Es wäre schön, wenn sich die Erzeugung von Konten automatisieren ließe. Auch das ist kein Problem:

```
(define make-withdraw
  (lambda (balance)
    (lambda (amount)
      (if (>= balance amount)
          (begin
            (set! balance (- balance amount))
            balance)
          'not-enough-money))))
```

Nun lassen sich mehrere voneinander unabhängige Konten erzeugen:

```
(define w1 (make-withdraw 100))
(define w2 (make-withdraw 100))
> (w1 50)
50
> (w2 70)
30
> (w2 40)
not-enough-money
> (w1 40)
10
```

Diese Beispiele sind ein Indiz dafür, daß das Substitutionsmodell nicht mehr ausreichend ist, um das Verhalten von `make-withdraw` zu erklären: Das grundlegende Prinzip des Substitutionsmodells ist, daß gleiches sich durch gleiches ersetzen läßt. Nun sind `w1` und `w2` Werte des gleichen Ausdrucks, haben aber offensichtlich voneinander verschiedene Identitäten. Damit wird auch klar, daß ein neuer Begriff der Identität mit Einführung der Zuweisung fällig wird.

8.2 Zuweisungen und das Substitutionsmodell

Zuweisungen erlauben es, viele Programmierprobleme effektiv zu lösen. Aber die Benutzung von Zuweisungen bringt eine ganze Reihe von Probleme mit sich: Bisher haben zwei Anwendungen einer Prozedur mit den gleichen Werten für die Parameter immer die gleichen Werte ergeben. Diese Eigenschaft heißt *referentielle Transparenz*, der resultierende Programmierstil heißt *rein funktionale Programmierung*, eben weil sich dann alle Prozeduren wie mathematische Funktionen verhalten. (In

einigen neueren Programmiersprachen lassen sich gar keine anderen Prozeduren ausdrücken.)

Der Unterschied wird an einer vereinfachten Version von `make-withdraw` klar, die sich von der „echten“ nur dadurch unterscheidet, daß sie beim Abheben nicht überprüft, ob das Konto leer ist:

```
(define simple-make-withdraw
  (lambda (balance)
    (lambda (amount)
      (set! balance (- balance amount))
      balance)))
```

Zum Vergleich dient eine ähnliche, referentiell transparente Prozedur:

```
(define make-decrementer
  (lambda (balance)
    (lambda (amount)
      (- balance amount))))
```

`Make-decrementer` erzeugt Prozeduren, die von einem vorgegeben Betrag etwas abziehen:

```
(define d (make-decrementer 100))
> (d 75)
25
> (d 25)
75
```

Das Substitutionsmodell erklärt, wie `make-decrementer` funktioniert:

```
((make-decrementer 100) 75)
⟹ ((lambda (amount) (- 100 amount)) 75)
⟹ (- 100 75)
⟹ 25
```

Dies funktioniert leider nicht für `simple-make-withdraw`:

```
((simple-make-withdraw 100) 75)
⟹ ((lambda (amount) (set! balance (- 100 amount)) 100) 75)
⟹ (set! balance (- 100 75)) 100
```

Damit wäre das Resultat des Ausdrucks nach dem Substitutionsmodell 100, was offensichtlich nicht der Realität entspricht. (Und ihr auch nicht entsprechen sollte.)

Was ist das fundamentale Problem? Das Substitutionsmodell basiert auf der Idee, daß Namen bzw. Variablen für *Werte* stehen. Diese Vorstellung wird mit der Einführung von `set!` offensichtlich falsch:

Eine Variable steht nun für einen *Ort* (eine Kiste, eine Speicherzelle ...), an dem sich ein Wert befindet. Dieser Wert läßt sich mit `set!` durch einen anderen austauschen.

Dementsprechend ergibt sich ein neuer Begriff für Identität:

```
(define d1 (make-decrementer 100))
(define d2 (make-decrementer 100))
```

Sind `d1` und `d2` gleich? Nun, sie verhalten sich auf jeden Fall gleich: Beides sind Prozeduren mit einem Parameter, und liefern das gleiche Resultat, wenn sie auf den gleichen Wert angewendet werden. Das heißt, daß in jeder Berechnung `d1` durch `d2` ersetzt werden kann, ohne das Ergebnis zu beeinflussen.

Bei `simple-make-withdraw` ist das anders:

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

Diese beiden sind offenbar spätestens nach der Auswertung von `(w1 20)` nicht mehr gleich:

```
> (w1 20)
60
> (w2 20)
80
```

Das heißt aber, daß das Substitutionsprinzip („gleiches läßt sich durch gleiches ersetzen“) nicht mehr funktioniert. Das Dilemma wird an einem anderen Beispiel noch offensichtlicher:

```
(define w1 (make-withdraw 100))
(define w2 w1)
> (w1 25)
75
> (w2 25)
50
```

Das heißt, das Änderungen an `w1` auch `w2` betreffen: `w1` und `w2` stehen für denselben Ort im Speicher. (Das ist umgangssprachlich die Definition von „dasselbe“: Wenn zwei Dinge dasselbe sind, betreffen Änderungen an einem auch das andere.) Damit aber ist ein Wert, der von einem solchen Programm manipuliert wird, nicht einfach die Summe seiner Teile: er hat auch eine Identität abseits dieser Konstruktion.

Diese Komplikationen entstehen durch eine Eigenheit unseres Modells. (Ist ein Bankkonto, nachdem 20 Mark abgehoben worde, dasselbe wie vorher? Ist es das gleiche?) Unsere Modellierungsmöglichkeiten sind durch die Einführung von Zustand erweitert worden, aber diese Erweiterungen bringen einige Probleme mit sich.

8.3 Imperative Programmierung

Ein Programmierstil, der sich vieler Zuweisungen bedient, wird als *imperative Programmierung* bezeichnet. Dementsprechend gibt es *imperative Programmiersprachen*, die einen solchen Programmierstil fördern oder sogar erzwingen.

Bei der Auswertung imperativer Programme gegenüber der rein funktionaler Programme ist eine zusätzliche Komponente hinzugekommen: die *Reihenfolge*. `Set!`-Ausdrücke können tief in der Auswertung von Prozeduren verbergen und damit bei Veränderung der Auswertungsreihenfolge subtile Fehler hervorrufen. Noch schlimmer wird es bei der Einführung von Parallelität, in dem sich die Auswertungsreihenfolge von Programmdurchlauf zu Programmdurchlauf verändern kann.

Ein weiteres Problem ist, daß sich bei Verwendung von `set!` das Verhalten einer Prozedur nicht mehr dadurch dokumentieren läßt, daß das Verhältnis von Ein- und Ausgaben beschrieben wird: Abhängigkeiten vom und Effekt der Prozedur auf den Zustand, muß ebenfalls spezifiziert werden. ADTs reichen als Beschreibungsmechanismus nicht mehr aus.

8.4 Das Umgebungsmodell für die Programmauswertung

Da das Substitutionsmodell nicht mehr ausreichend ist, um das Verhalten imperativer Programme zu erklären, muß ein neues Modell her: das sogenannte *Umgebungsmodell.*

Das Problem beim Substitutionsmodell, was die Erklärung von imperativen Programmen betrifft, ist die Regel für die Prozeduranwendung. Hier ist sie noch einmal, zur Erinnerung:

- Bei der Auswertung eines Prozeduraufrufs werden zunächst die Werte des Operanden und der Operatoren festgestellt. Der Operator muß eine Prozedur mit ebensoviel Parametern sein, wie der Prozeduraufruf Operanden hat. Der Wert des Prozeduraufrufs ist dabei der Wert des Rumpfes der Prozedur, wobei die Vorkommen der Parameter im Rumpf durch die Werte der Operanden des Ausdrucks ersetzt werden.

Dies ist die Essenz der Vorstellung, daß ein Name für einen Wert steht. Nach der Einführung von Zuweisungen steht aber ein Name für einen Ort, der sich irgendwo befinden muß. Im Umgebungsmodell befinden sich die Orte für die Werte von Variablen in Strukturen, die *Umgebungen* heißen. Eine Umgebung besteht aus einer Folge sogenannter *Frames* (ein passender deutscher Ausdruck hat sich leider noch nicht eingebürgert), und jedes Frame ist eine Tabelle von *Bindungen*, die Variablen mit ihren Werten assoziieren. Jedes Frame besitzt außerdem einen Zeiger auf die *umschließende Umgebung* bis eine spezielle letzte, ausgezeichnete Umgebung, die *globale Umgebung*, von der es nur eine gibt. Der *Wert einer Variable* in bezug auf eine Umgebung ist der Wert der ersten Bindung in der Frame-Folge, die von der Umgebung ausgeht. Gibt es keine solche Bindung, ist die Variable *ungebunden*.

Die globale Umgebung besteht anfangs aus den Bindungen für die eingebauten Prozeduren und kann durch `define` erweitert werden.

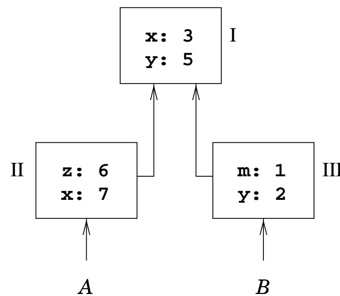

Abbildung 8.1: Eine einfache Umgebungsstruktur

Abbildung 8.1 zeigt ein Beispiel für eine Umgebungsstruktur aus drei Frames. Dabei stehen A und B für die Umgebungen, die von den Frames II und III ausgehen. Zum Beispiel ist der Wert von `x` in bezug auf Umgebung A 7, in bezug auf Umgebung B aber 3. Der Wert von `y` in bezug auf A ist 5, in bezug auf B aber 2.

Die Umgebung stellt einen Kontext für die Auswertung eines Ausdrucks dar: ein Ausdruck bekommt im Umgebungsmodell erst durch eine begleitende Umgebung einen Wert. Inbesondere wird eine Umgebung benötigt, um den Rumpf eines λ-Ausdrucks auszuwerten. Die vollständigen Zutaten für eine Prozedur — den Wert eines `lambda`-Ausdrucks — sind damit:

- die Parameter
- der Rumpf
- die Umgebung

Prozeduren sind also Tripel aus diesen Komponenten. Diese Tripel heißen *Closures*. Dementsprechend ändern sich die Auswertungsregeln für Variablen, Abstraktionen

und Anwendungen gegenüber dem Substitutionsmodell. Alle beziehen sich nun auf eine *momentane Umgebung*:

- Der Wert einer Variablen ist ihr Wert in der momentanen Umgebung.
- Der Wert einer `lambda`-Abstraktion ist ein Tripel aus den Parametern, dem Rumpf der Abstraktion und der momentanen Umgebung.
- Bei der Auswertung eines Prozeduraufrufs werden zunächst Prozedur und Operanden in bezug auf die momentane Umgebung ausgewertet. Der Operator muß eine Prozedur mit ebensovielen Parametern sein, wie der Prozeduraufruf Operanden hat. Nun wird ein neues Frame erzeugt, dessen umschließende Umgebung die Umgebung in der Closure ist, und in der Bindungen der Parameter an die Werte der Operanden angelegt werden. Der Wert des Prozeduraufrufs ist dann der Wert des Prozedurrumpfes in bezug auf die Umgebung, die vom neuen Frame ausgeht.
- Eine Zuweisung führt dazu, daß die Bindung der angegeben Variable verändert wird, so daß sie die Variable an den Wert des Ausdrucks der Zuweisung bindet.

Ein einfaches Beispiel für die Anwendung des Umgebungsmodells ist die folgende Prozedur:

```
(define square
  (lambda (x)
    (* x x)))
```

Die Umgebungsstruktur, die durch Auswertung dieser Definition entsteht, ist in Abbildung 8.2 zu sehen. Dabei gibt es zunächst nur die eine globale Umgebung, in

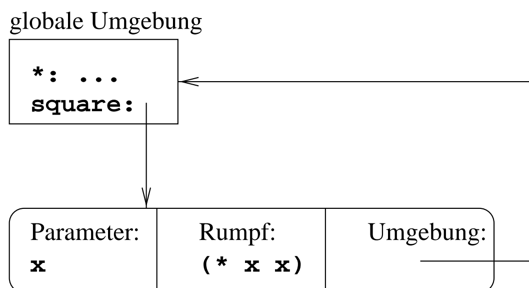

Abbildung 8.2: Die Umgebungsstruktur von `square`

deren Kontext der `lambda`-Ausdruck von `square` ausgewertet wird. Demnach ist die Umgebung der entstehenden Prozedur auch die globale Umgebung; diese Prozedur wird durch `define` in der globalen Umgebung an `square` gebunden.

Durch nachfolgende Anwendung von `square` wird die Umgebungsstruktur erweitert: Abbildung 8.3 zeigt das die Umgebungsstruktur, die bei der Auswertung von `(square 23)` entsteht.

Ein etwas umfangreicheres Beispiel entsteht durch die Betrachtung der Konto-Fabrik `make-withdraw`. Betrachtet werden soll die Umgebungsstruktur, die zunächst durch die Definition selbst erzeugt wird:

```
(define make-withdraw
  (lambda (balance)
    (lambda (amount)
      (if (>= balance amount)
```

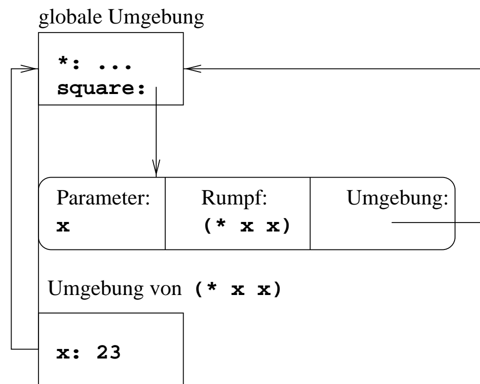

Abbildung 8.3: Die Umgebungsstruktur bei der Anwendung von `(square 23)`

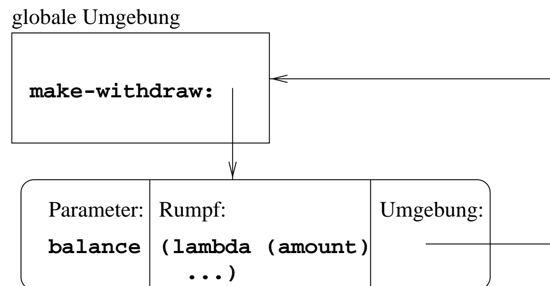

Abbildung 8.4: Die Umgebungsstruktur nach der Auswertung der Definition von `make-withdraw`

```
          (begin
            (set! balance (- balance amount))
            balance)
          'not-enough-money))))
```

Interessant wird es bei der Auswertung von:

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

gefolgt von:

```
(w1 50)
```

Die Umgebungsstruktur nach der Auswertung der Definition von `make-withdraw` ist in Abbildung 8.4 zu sehen. Die einzige Neuerung gegenüber dem letzten Beispiel ist höchstens, daß der Rumpf der Prozedur selbst eine Abstraktion ist. Der Auswertung der Definition von `w1` erzeugt ein eine neue Prozedur und damit ein weiteres Frame für die Bindung von `balance`, das Teil des Tripels wird, das die Prozedur repräsentiert. Abbildung 8.5 zeigt das Resultat.

Die Auswertung von `(w1 50)` erzeugt zunächst ein weiteres Frame mit der Bindung für den Parameter `amount`. Abbildung 8.6 zeigt die erweiterte Umgebungsstruktur.

Die Auswertung des Rumpfes von `w1` benutzt dann `set!`, verändert also die Bindung von `balance` im entsprechenden Frame. Nach der Auswertung verschwindet das Frame für `amount` wieder, und das Ergebnis sieht aus wie in Abbildung 8.7 gezeigt.

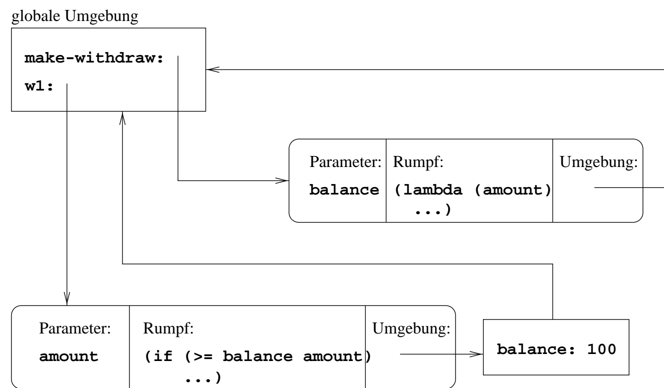

Abbildung 8.5: Die Umgebungsstruktur nach der Auswertung der Definition von `w1`

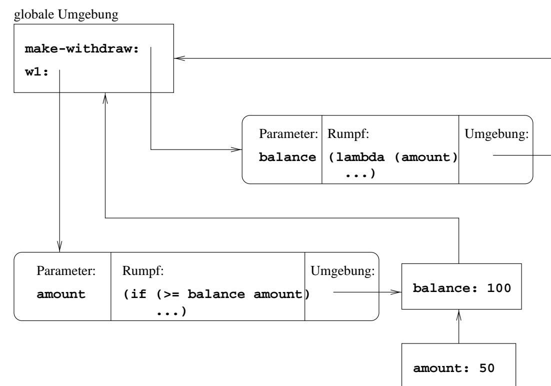

Abbildung 8.6: Die Umgebungsstruktur nach der Anwendung von `w1` auf 50

8.5 Mutierbare Datenstrukturen

`Set!` verändert Bindungen der Umgebungsstruktur. Manchmal ist es jedoch auch wünschenswert, Datenstrukturen zu modifizieren. Für Paare stellt 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 ...
```

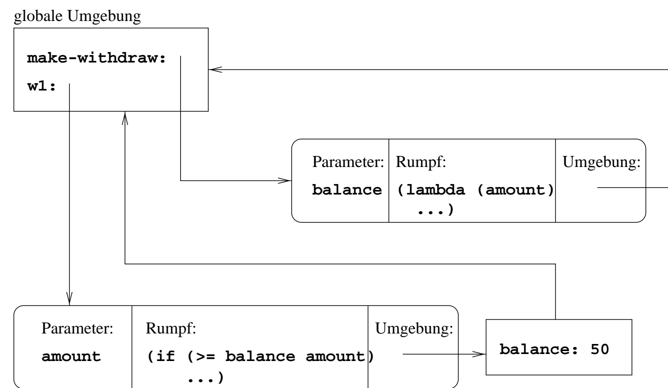

Abbildung 8.7: Die Umgebungsstruktur nach der Auswertung von `(w1 50)`

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` *geshare-ret*, 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. Einem Scheme-System ist es freigestellt, bei der Anwendung von `eq?` auf zwei Zahlen `#t` oder `#f` zurückzugeben. Bei den meisten Scheme-Systeme zum Beispiel funktioniert `eq?` zwar auf kleinen, nicht jedoch auf großen Zahlen:

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

Moral: `eq?` ist für den Vergleich von Zahlenwerten als Ersatz von `=` oder `equal?` ungeeignet.

Interessanterweise 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 bestimt 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 c2 (typed-value-maker t2))
(define p1 (typed-value-predicate t1))
(define o1 (c2 'attacke!))
> (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-value-selector
  (lambda (type)
    (lambda (value)
      (if (eq? type (car value))
          (cdr value)
          (error "type mismatch")))))

(define typed-value-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 c2 (typed-value-maker t2))
(define p1 (typed-value-predicate t1))
(define o1 (c2 'attacke!))
> (p1 o1)
#f
```