

Kapitel 5

Datenstrukturen im Speicher

Der Simulator des letzten Kapitels erlaubt es, bei vielen Aufgaben der Assemblerprogrammierung zu „schummeln“, indem Register und Speicherzellen dazu benutzt werden, auch Symbole, Paare und ggf. sogar Prozeduren zu enthalten. In der Realität ist dies nicht möglich: dort können Register und Speicherzellen nur Zahlen aufnehmen, in der Regel sogar nur Zahlen fester Bitbreite. Trotzdem wollen natürlich auch Assemblerprogramme mit Datenstrukturen hantieren.

5.1 Repräsentation von Paaren

Das Repräsentationsproblem für Paare läßt sich auf das der Repräsentation von Paaren zurückführen. Alle anderen Repräsentationen lassen sich analog konstruieren. Dazu können zwei Abschnitte im Speicher dienen — einer für die Aufnahme der `car`-Komponenten der Paare, der andere für die `cdr`-Komponenten. Speicherabschnitte entsprechen zwei Vektoren `*cars*` und `*cdrs*`, die mit folgenden Definitionen beispielsweise Platz für 1000 Paare bieten:

```
(define *cars* (make-vector 1000 0))
(define *cdrs* (make-vector 1000 0))
```

Ein Paar könnte repräsentiert werden als ein Index in diese beiden Felder:

```
(define (new-car p)
  (vector-ref *cars* p))
(define (new-set-car! p v)
  (vector-set! *cars* p v))
(define (new-cdr p)
  (vector-ref *cdrs* p))
(define (new-set-cdr! p v)
  (vector-set! *cdrs* p v))
```

Die Realisierung von `cons` erfordert Buchführung darüber, welche Zellen in `*cars*` und `*cdrs*` schon belegt sind. Am einfachsten geschieht dies durch eine Variable, welche den Index des nächsten freien Pairs enthält:

```
(define *next-pair* 0)

(define (new-cons a b)
  (let ((p *next-pair*))
    (vector-set! *cars* p a)
    (vector-set! *cdrs* p b)
```

```
(set! *next-pair* (+ 1 *next-pair*))
p))
```

Diese Repräsentation ist dann praktikabel, wenn Paare die einzigen Daten sind, die vom Programm verarbeitet werden. In der Regel sind aber noch „echte“ Zahlen (also solche, die keine Paare repräsentieren), Booleans, Symbole etc. im Spiel, die von Paaren unterscheidbar sein müssen. Gleichzeitig sollen alle diese Werte repräsentiert werden durch ein einzelnes Maschinenwort.

Diese Unterscheidung zwischen verschiedenen Sorten Maschinenworten erfordert eine Markierung jedes Worts mit Typinformation. Meistens werden dafür einige Bits innerhalb des Worts reserviert. Diese Markierung (meist genannt *Tag*) erlaubt dem System, zwischen Worten, die Paare repräsentieren, von solchen zu unterscheiden, die Zahlen oder boolesche Werte repräsentieren.

Diese Art der Datenrepräsentation erklärt auch, warum einige Prozeduren und Datentypen in Scheme so funktionieren wie sie es tun. (Der Scheme-Standard läßt andere Repräsentationen zu, begünstigt aber diese hier.)

- `Eq?` vergleicht die Maschinenworte, die seine Parameter repräsentieren direkt miteinander. Dies ist in der Regel eine einzelne Maschineninstruktion und damit sehr schnell.

`Equal?` im Gegensatz dazu muß den Typ des Worts betrachten und abhängig davon handeln: Bei Datenstrukturen wie Paaren und Vektoren müssen die Komponenten einzeln verglichen werden.

- Scheme erlaubt die Benutzung beliebig großer Zahlen. In einem Maschinenwort ist aber nur die Repräsentation begrenzt großer Zahlen möglich — dabei handelt es sich um sogenannte *Fixnums*. Für größere Zahlen (*Fixnums*) müssen Datenstrukturen im Speicher benutzt werden, die durch einen Zeiger darauf repräsentiert werden. Das erklärt, warum `eq?` zwar kleine Zahlen akkurat vergleicht, große aber nicht.
- Ein Symbol wird durch einen Zeiger auf eine Folger der Buchstaben repräsentiert, die seinen Namen bilden. Dabei ist allerdings zu beachten, daß Symbole gesharet werden: Zu jedem Namen gibt es nur ein Symbol im Sinne von `eq?`. Darum muß das Scheme-System, wenn es ein Symbol mit einem gegebenen Namen wiederholt konstruiert, sichergehen, daß jedesmal der gleiche Zeiger vergeben wird. Das Scheme-System verwaltet also eine Tabelle aller bereits erzeugten Symbole. Jedesmal, wenn ein Symbol angelegt wird, wird entweder ein neuer Tabelleneintrag angelegt oder ein alter wiederverwendet. Dieser Prozeß heißt gelegentlich *Interning*.

Die Details für eine effektive Datenrepräsentation mit Tags hängen stark von der verwendeten Programmiersprache und Maschine ab und werden darum hier nicht weiter ausgeführt.

5.2 Die Illusion unendlichen Speichers

Die Repräsentation von Paaren in Vektoren aus Zahlen funktioniert wunderbar für die ersten 1000 Aufrufe von `cons`. Dann ist leider der Speicher voll. Moderne Rechner haben zwar gigantische Hauptspeicher (jedenfalls solche mit mehr als 1000 Zellen), aber auch die sind irgendwann belegt.

Glücklicherweise haben die meisten Datenstrukturen in realen Programmen eine begrenzte Lebensdauer, weil sie nur Zwischenergebnisse der Berechnung halten. Wenn diese Zwischenergebnisse nicht mehr benötigt werden, kann ihr Speicher wiederverwendet werden: Sie heißen dann *Müll* oder *Garbage*.

Manche Programmiersprachen haben spezielle Prozeduren, welche benutzten Speicherplatz wieder freigeben. (Pascal, C und C++ sind Beispiele dafür.) In Scheme müßte eine solche Prozedur `destroy-pair!` oder ähnlich heißen. Solche sogenannte *manuelle Speicherverwaltung* hat allerdings eine Reihe von Nachteilen. Sie hängen damit zusammen, daß der Programmierer sich bei der Beurteilung, ob eine Datenstruktur zu Müll geworden ist, irren kann:

```
(define p (cons 1 2))
(destroy-pair! p)
(car p)
```

Was soll beim letzten Ausdruck als Ergebnis herauskommen? In C führen solche Programme im besten Fall zu Systemabstürzen, in schlechteren zu unerklärlichem Fehlverhalten. In C- und C++-Programm sind in der Tat ca. 40% aller auftretenden Fehler solche „Speicherlecks“. Umgekehrt führt das Vergessen eines Aufrufs von `destroy-pair!` dazu, daß der vom Programm benötigte Speicher monoton wächst und, wenn er länger läuft, irgendwann den verfügbaren Speicher übersteigt.

Moderne Programmiersprachen verzichten darum auf Prozeduren wie `destroy-pair!`. Stattdessen führen sie in periodischen Abständen eine Erhebung darüber durch, welche vom Programm angelegten Datenstrukturen inzwischen zu Müll geworden sind, und recyclet ihren Speicherplatz für zukünftige Aufrufe von `cons` und Konsorten. Dieser Recycling-Vorgang heißt *Garbage Collection* oder kurz *GC*. Systeme mit Garbage Collection sind durchgängig einfacher zu programmieren und verhalten sich robuster. Moderne GC-Techniken sind außerdem meist schneller als die Speicherverwaltungs-Strategien menschlicher Programmierer.

Die fundamentale Einsicht hinter der GC-Technik ist, daß alle Daten, auf die das Program in Zukunft noch zugreifen kann, entweder in den Maschinenregistern steht oder durch eine Folge von `car`- und `cdr`-Operationen erreichbar sein muß, die mit dem Inhalt eines Maschinenregisters beginnt. Diese „Nicht-Müll-Daten“ heißen auch *lebendig*. Jede Speicherzelle, die nicht erreichbar ist, ist Müll oder auch *tot*.

5.3 Realisierung eines Garbage Collectors

Es gibt viele Strategien und Methoden, um GC durchzuführen. Die GC-Methode hängt dabei eng zusammen mit der Methode, Speicherplatz für neue Datenstrukturen zur Verfügung zu stellen.

Eine besonders einfache Technik ist die des *Two-Space Copiers*. Dieses Verfahren teilt den Speicher in zwei Hälften auf: den *Arbeitsspeicher* und den *freien Speicher*. Die Datenstrukturen des Programms befinden sich ausschließlich im Arbeitsspeicher. Wenn GC stattfindet (in der Regel dann, wenn der Mutator Speicher anfordert, der nicht mehr verfügbar ist) werden alle lebendigen Datenstrukturen vom Arbeitsspeicher in den freien Speicher kopiert. Danach werden die Rollen der beiden Speicherhälften vertauscht. Da die GC nur die lebendigen Datenstrukturen kopiert, sollte im freien Speicher nach der GC noch Platz übrig sein für die Bereitsstellung neuer Paare.

Der Two-Space Copier kann beim Kopieren alle Datenstrukturen in aufeinanderfolgenden Speicherzellen ablegen. Dieser Prozeß heißt *Kompaktifizierung*. Damit ist der neue Arbeitsspeicher nach der GC in zwei Bereiche aufgeteilt: den belegten und den freien Speicher. Für die Verwaltung des freien Bereichs reicht also ein Zeiger auf dessen Anfang; dieser muß nur nach der Reservierung von Speicher für eine Datenstruktur entsprechend erhöht werden. Die GC wird dann in der Regel aktiviert, wenn dieser Zeiger das Ende des Arbeitsspeichers bzw. des freien Bereichs erreicht hat.

Zu beachten bei dieser Methode ist, daß die GC die Datenstrukturen im Speicher verschiebt oder *reloziert*. Sie befinden sich also nach der GC an anderen Adressen als vorher. Das heißt aber, daß die GC sicherstellen muß, daß alle Zeiger auf Datenstrukturen so verändert werden müssen, daß sie auf die relozierten Strukturen zeigen.

Die GC benötigt, um arbeiten zu können, Kooperation vom laufenden Programm — in diesem Zusammenhang oft als *Mutator* bezeichnet: Der Mutator muß zum Zeitpunkt der GC einen Satz von Registern und Speicherzellen bei der GC anmelden, welche die Ausgangspunkte für alle lebendigen Daten des Systems darstellen. Diese Ausgangspunkte heißen auch *Root-Set*. Die GC übermittelt nach getaner Arbeit die Werte des Root-Sets zurück an den Mutator, und zwar so justiert, daß diese auf die entsprechenden relozierten Datenstrukturen nach der GC zeigen.

Eine beispielhafte Implementation eines Two-Space-Copiers in Scheme zeigt, wie das Verfahren funktioniert. Dabei sei angenommen, daß es im System nur Zahlen und Paare gibt. Die Variablen `*n-pairs*` und `*free-pair*` enthalten die Anzahl der Paare, für die Platz im Speicher ist, und den Index des nächsten freien Paares respektive:

```
(define *n-pairs* 100)
(define *free-pair* 0)
```

Zwei Sätze Vektoren sind für den Arbeitsspeicher und den freien Speicher zuständig:

```
(define *the-cars* (make-vector *n-pairs* 0))
(define *the-cdrs* (make-vector *n-pairs* 0))

(define *from-cars* (make-vector *n-pairs* 0))
(define *from-cdrs* (make-vector *n-pairs* 0))
```

Repräsentiert werden Zahlen und Paare durch Zahlen mit Tags. Als Tag fungiert dabei das untere Bit: Zahlen werden durch gerade Zahlen, Paare durch ungerade Zahlen repräsentiert. Die neuen Repräsentation bekommen — zwecks Unterscheidung von den alten — den Präfix `new-`.

```
(define (new-number? n)
  (even? n))

(define (number-value n)
  (quotient n 2))

(define (make-number n)
  (* 2 n))

(define (new-pair? n)
  (odd? n))

(define (make-new-pair index)
  (+ 1 (* 2 index)))

(define (pair-index p)
  (quotient (- p 1) 2))
```

`New-cons` gibt `#f` zurück, falls der Speicher erschöpft ist. Ansonsten werden — wie schon oben — die entsprechenden Zellen in `*the-cars*` und `*the-cdr*` belegt und der getagte Index in die Vektoren zurückgegeben:

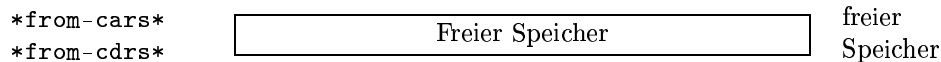
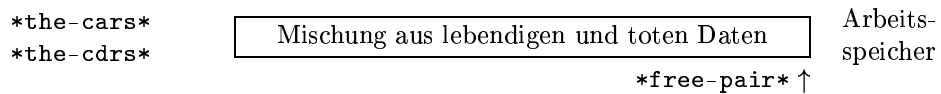


Abbildung 5.1: Speicher vor der GC

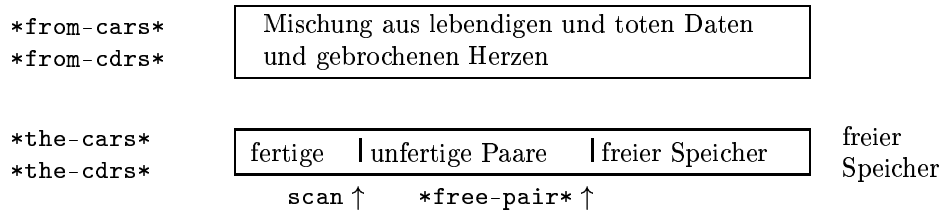


Abbildung 5.2: Speicher während GC

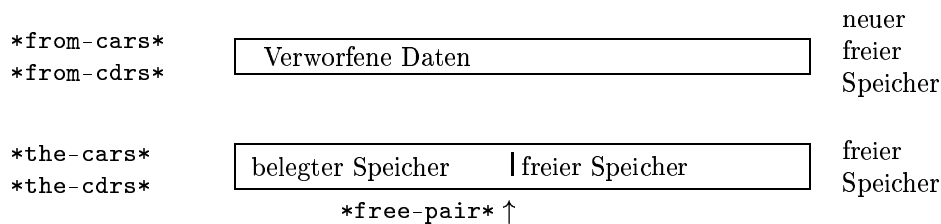


Abbildung 5.3: Speicher nach der GC

```
(define (new-cons a b)
  (if (>= *free-pair* *n-pairs*)
      #f
      (let ((index *free-pair*))
        (vector-set! *the-cars* index a)
        (vector-set! *the-cdrs* index b)
        (set! *free-pair* (+ 1 *free-pair*))
        (make-new-pair index))))
```

New-car und new-cdr bleiben unverändert:

```
(define (new-car p)
  (vector-ref *the-cars* (pair-index p)))

(define (new-cdr p)
  (vector-ref *the-cdrs* (pair-index p)))
```

Der eigentliche Kern des Two-Space-Copiers wendet eine Operation namens *Tracing* auf alle lebendigen Objekte an: Tracing stellt zunächst einmal fest, ob es sich um eine Zahl oder ein Paar handelt. Paare werden — falls dies noch nicht geschehen ist — in den freien Speicher kopiert. Zunächst passiert dies für alle Werte des Root-Sets, deren Paare sich hernach im freien Speicher befinden. *Free-pair* ist dabei ein Index in den freien Speicher, der auf das nächste freie Paar zeigt. Alsdann läuft die GC von Anfang an alle Paare im freien Speicher ab mit Hilfe eines Zeigers *scan* in den freien Speicher. Bei jedem kopierten Paar wird dabei *free-pair* erhöht. Der Prozess ist dann beendet, wenn *scan* den Wert von *free-pair* erreicht hat.

Die zentrale Prozedur des Two-Space-Copiers heißt `collect-garbage` und bekommt das Root-Set als Liste von Werten übergeben. Die Prozedur gibt später eine Liste mit den justierten Werten zurück. Entscheidend ist, daß *alle* Werte übergeben werden, auf die das Programm unmittelbaren Zugriff hat. Ein Aufruf könnte so aussehen:

```
(define *reg1* ...)
(define *reg2* ...)
(define *reg3* ...)
(let ((p (new-cons a b)))
  (if (not p)
      (let ((new-regs (collect-garbage (list *reg1* *reg2* *reg3*)))
            (set! *reg1* (car new-regs))
            (set! *reg2* (car (cdr new-regs))
                  (set! *reg3* (car (cdr (cdr new-regs))))
            (set! p (new-cons a b))))
      ...))
```

`Collect-garbage` vertauscht zunächst die Rollen der beiden Vektorensätze:

```
(define (collect-garbage roots)
  (swap-spaces!))
```

`Swap-spaces!` tut das offensichtliche:

```
(define (swap-spaces!)
  (let ((from-cars *from-cars*)
        (from-cdrs *from-cdrs*)
        (set! *from-cars* *the-cars*)
        (set! *from-cdrs* *the-cdrs*)

        (set! *the-cars* from-cars)
        (set! *the-cdrs* from-cdrs)))
```

Zur Erinnerung: Die „alten“ Paare befinden sich nun in `*from-cars*` und `*from-cdrs*` und die lebendigen unter ihnen müssen in `*the-cars*` und `*the-cdrs*` übertragen werden. Zunächst wird in `collect-garbage` der Zeiger `*free-pair*` auf den Anfang von `*the-cars*` und `*the-cdrs*` gesetzt. Dann werden die Werte des Root-Sets getracet:

```
(set! *free-pair* 0)
(let ((new-roots (map trace-value roots)))
```

Nun kann die Schleife über die Paare in `*the-cars*` und `*the-cdrs*` laufen; jeder `car` und jeder `cdr` wird ebenfalls getracet:

```
(letrec ((loop
          (lambda (scan)
            (if (> *free-pair* scan)
                (begin
                  (vector-set! *the-cars* scan
                              (trace-value (vector-ref *the-cars* scan)))
                  (vector-set! *the-cdrs* scan
                              (trace-value (vector-ref *the-cdrs* scan)))
                  (loop (+ 1 scan))))))
        (loop 0))
```

Schließlich werden die neuen Werte für das Root-Set zurückgegeben:

```
new-roots))
```

Trace-value unterscheidet zunächst zwischen Paaren und Zahlen:

```
(define (trace-value value)
  (if (new-pair? value)
      (copy-pair value)
      value))
```

Die Hauptarbeit des Copiers wird in `copy-pair` verrichtet. Dabei gibt es noch ein Problem zu lösen: Es kann für ein Paar mehrere Zeiger geben, die darauf verweisen. Kopiert werden darf es jedoch nur einmal, um Sharing zu erhalten. `Copy-pair` muß also Buch darüber führen, welche Paare bereits kopiert wurden und wohin. Zu diesem Zweck werden für ein kopiertes Paar entsprechende Informationen in `*from-cars*` und `*from-cdrs*` abgelegt: Eine Markierung, welche besagt, daß das Objekt bereits kopiert wurde, und eine „Nachsendeadresse“, also ein Verweis auf die neue Position des Paares in `*the-cars*` und `*the-cdrs*`. Diese Markierung wird in der Folklore der Speicherverwaltung als *gebrochenes Herz* („broken heart“) bezeichnet. Diese GC benutzt dafür das Symbol `broken-heart`:

```
(define (copy-pair pair)
  (let* ((index (pair-index pair))
        (car-val (vector-ref *from-cars* index)))
    (if (eq? car-val 'broken-heart)
        (vector-ref *from-cdrs* index)
```

Wenn das Paar noch nicht kopiert wurde, muß dies noch geschehen:

```
(let ((new-index *free-pair*))
  (vector-set! *the-cars* new-index car-val)
  (vector-set! *the-cdrs* new-index
              (vector-ref *from-cdrs* index)))
```

Das Herz an der alten Adresse muß gebrochen werden:

```
(vector-set! *from-cars* index 'broken-heart)
(vector-set! *from-cdrs* index (make-new-pair new-index))
```

Zum Schluß wird noch `*free-pair*` inkrementiert und das kopierte Paar zurückgegeben:

```
(set! *free-pair* (+ 1 *free-pair*))
(make-new-pair new-index))))
```