Kapitel 15

Speicherverwaltung

Die virtuelle Maschine aus dem letzten Kapitel kann nur mit ganzzahligen Werten rechnen. Wir erweitern die Maschine jetzt um die Fähigkeit, mit Listen, Bäumen, Referenzen und (dynamischen) Prozeduren zu rechnen. Diese sogenannten *dynamischen Datenstrukturen* werden in einem zusätzlichen Speicherbereich dargestellt, der als *Halde* bezeichnet wird. Dynamische Datenstrukturen werden dabei als Folgen von ganzen Zahlen dargestellt. Das dabei zur Anwendung kommende Schema wird als *verzeigerte Blockdarstellung* bezeichnet.

Im Laufe einer Berechnung sammeln sich auf der Halde Objekte, die für die Fortsetzung der Berechnung nicht mehr benötigt werden. Diese Objekte müssen spätestens dann entfernt werden, wenn der Platz in der Halde erschöpft ist. Diese Aufgabe wird als *Speicherbereinigung* bezeichnet.[1] Wir werden einen raffinierten Algorithmus kennenlernen, der die Halde automatisch bereinigt.

Das in diesem Kapitel beschriebene Speichermodell versetzt uns in die Lage, den Platzbedarf für die durch ein Standard ML Programm berechneten Datenstrukturen abzuschätzen. Die Kenntnis dieses Modells ist eine wichtige Voraussetzung für die Entwicklung effizienter Programme.

15.1 Gerichtete Graphen

Als konzeptuelles Hilfsmittel führen wir den mathematischen Begriff des *gerichteten Graphen* ein. Dabei handelt es sich um eine Formalisierung von bestimmten grafischen Darstellungen, die unter anderem für zusammengesetzte Datenstrukturen verwendet werden.

Ein **gerichteter Graph** ist ein Paar (V, E), das aus einer endlichen Menge V und einer Mege $E \subseteq V \times V$ besteht. Die Elemente von V heißen **Knoten** (englisch

[1] Im Englischen spricht man anschaulich von *garbage collection*.

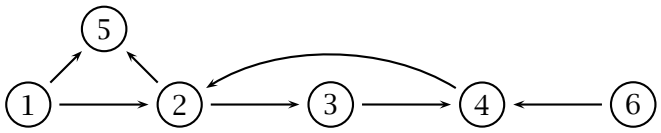


Abbildung 15.1: Ein gerichteter Graph

vertices) und die Elemente von E **Kanten** (englisch edges). Wir sagen, dass eine Kante $(v_1, v_2) \in E$ vom Knoten v_1 zum Knoten v_2 führt.

Gerichtete Graphen lassen sich grafisch darstellen. Dabei wird eine Kante (v_1, v_2) als ein Pfeil dargestellt, der vom Knoten v_1 zum Knoten v_2 führt. Knoten werden meistens durch Kreise dargestellt. Als Beispiel betrachten wir den gerichteten Graphen $G = (V, E)$ mit

$$V = \{1,\ 2,\ 3,\ 4,\ 5,\ 6\}$$
$$E = \{(1,2),\ (1,5),\ (2,3),\ (2,5),\ (3,4),\ (4,2),\ (6,4)\}$$

Abbildung 15.1 zeigt eine mögliche grafische Darstellung dieses Graphens. Abbildung 15.2 zeigt eine zweite grafische Darstellung desselben Grafens, bei der die Knoten und Kanten des Graphens anders angeordnet sind. Man sagt, dass sich die beiden Darstellungen im *Layout* unterscheiden.

Das Konzept des Graphen hat sich aus den anschaulichen grafischen Darstellungen entwickelt. Der gerade eingeführte mathematische Begriff des gerichteten Graphen klärt mit überraschend einfachen Mitteln, was unter dem zunächst intuitiven Konzept eines gerichteten Graphen genau zu verstehen ist. Der mathematische Begriff des gerichteten Graphen gehört zum Standardvokabular der Informatik und bildet eine tragfähige Grundlage für die Formulierung und den Beweis von Grapheigenschaften.

Neben den gerade eingeführten Graphen mit gerichteten Kanten gibt es auch Graphen mit ungerichteten Kanten. Da wir aber nur gerichtete Graphen betrachten werden, sagen wir im Folgenden einfach Graph und meinen stets gerichteter Graph.

Sei $G = (V, E)$ ein Graph. Dann ist E eine binäre Relation auf V. Zu jeder endlichen binäre Relation E erhalten wir einen Graphen indem wir $Dom\, E \cup Ran\, E$ als Knotenmenge wählen.

Sei $G = (V, E)$ ein Graph. Wir vereinbaren die folgenden Sprechweisen:

- Ein Knoten v' heißt **Nachfolger** eines Knotens v genau dann, wenn $(v, v') \in E$.
- Ein Knoten v heißt **Vorgänger** eines Knotens v' genau dann, wenn $(v, v') \in E$.

© G. Smolka, 12.2.2003

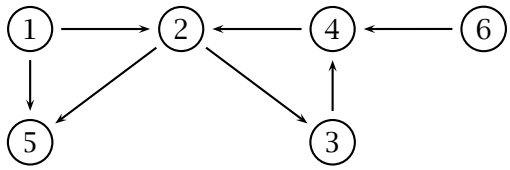


Abbildung 15.2: Eine andere Darstellung des Graphen aus Abbildung 15.1

- Ein Knoten heißt **Quelle** genau dann, wenn er keinen Vorgänger hat.
- Ein Knoten heißt **Senke** genau dann, wenn er keinen Nachfolger hat.
- Der **Eingrad** eines Knotens ist die Anzahl seiner Vorgänger.
- Der **Ausgrad** eines Knotens ist die Anzahl seiner Nachfolger.
- Ein **Pfad** ist ein nichtleeres Tupel $\langle v_1, \ldots, v_n \rangle$, dessen Komponenten durch Kanten verbundene Knoten sind. Genauer:
  1. Für jeden Knoten v ist das Tupel $\langle v \rangle$ ein Pfad.
  2. Für $n \geq 2$ Knoten $v_1, \ldots, v_n$ mit $(v_i, v_{i+1}) \in E$ für alle $i \in \{1, \ldots, n-1\}$ ist das Tupel $\langle v_1, \ldots, v_n \rangle$ ein Pfad.

  Dabei heißt $n-1$ die **Länge**, v_1 der **Ausgangspunkt** und v_n der **Endpunkt** des Pfades. Wir sagen auch, dass der Pfad von v_1 nach v_n geht.
- Ein Pfad heißt **einfach** genau dann, wenn er keinen Knoten mehrfach enthält.
- Ein Pfad $\langle v_1, \ldots, v_n \rangle$ heißt **Zyklus** genau dann, wenn $n \geq 2$ und $v_1 = v_n$.
- Ein Knoten v' ist von einem Knoten v aus **erreichbar** genau dann, wenn ein Pfad von v nach v' existiert.
- Wir schreiben $v \to_G v'$ genau dann, wenn $(v, v') \in E$.
- Wir schreiben $v \to_G^* v'$ genau dann, wenn v' von v aus erreichbar ist.

Für den Graphen in Abbildung 15.1 gilt:

- Der Graph hat die Quellen 1 und 6 und die Senke 5.
- Der Knoten 2 hat den Eingrad 2 und den Ausgrad 2.
- Der Knoten 6 hat den Eingrad 0 und den Ausgrad 1.
- Vom Knoten 2 aus sind die Knoten 2, 5, 3 und 4 erreichbar.
- Vom Knoten 1 aus sind alle Knoten bis auf 6 erreichbar.
- Der Pfad $\langle 2, 3, 4, 2 \rangle$ ist ein Zyklus.

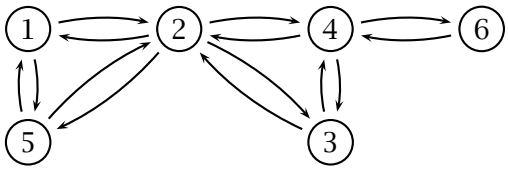


Abbildung 15.3: Der symmetrische Abschluss des Graphen aus Abbildung 15.2

Wir brauchen noch mehr Begriffe für Graphen:

- Die **Größe** eines Graphen ist die Anzahl seiner Knoten.
- Ein Graph heißt **nichtleer** genau dann, wenn er mindestens einen Knoten hat. Der Graph $(\emptyset, \emptyset)$ heißt der **leere Graph**.
- Die **Tiefe** eines nichtleeren Graphen ist die maximale Länge seiner einfachen Pfade.
- Ein Graph heißt **zyklisch** genau dann, wenn er einen Zyklus enthält.
- Ein Graph heißt **azyklisch** genau dann, wenn er keinen Zyklus enthält.
- Der **symmetrische Abschluss** eines Graphen $G = (V, E)$ ist der Graph

  $$\overleftrightarrow{G} = (V, E \cup E^{-1}) \qquad \text{mit} \qquad E^{-1} = \{(v, v') \mid (v', v) \in E\}$$

- Ein Graph heißt **stark zusammenhängend** genau dann, wenn jeder seiner Knoten von jedem seiner Knoten aus erreichbar ist.
- Ein Graph (V, E) heißt **zusammenhängend** genau dann, wenn $\overleftrightarrow{G}$ stark zusammenhängend ist.

Abbildung 15.3 zeigt den symmetrischen Abschluss des Graphen in Abbildung 15.2. Der Graph in Abbildung 15.2 ist nichtleer, zyklisch und zusammenhängend, aber nicht stark zusammenhängend. Er hat die Größe 6 und die Tiefe 3. Der Graph in Abbildung 15.3 ist nichtleer, zyklisch, zusammenhängend und stark zusammenhängend. Er hat die Größe 6 und die Tiefe 5.

Proposition 15.1.1 (Azyklische Graphen) *Für jeden Graph G sind die folgenden Aussagen äquivalent:*

1. *G ist azyklisch.*
2. *G hat nur einfache Pfade.*
3. *G hat nur endlich viele Pfade.*

Weiter gilt für jeden azyklischen Graphen G: Zu jedem Knoten v existiert eine Quelle, von der aus v erreichbar ist.

© G. Smolka, 12.2.2003

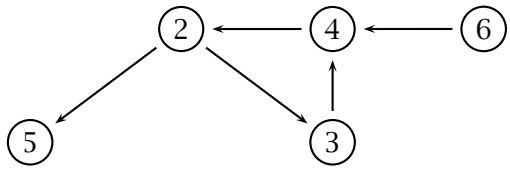


Abbildung 15.4: Der von 6 aus erreichbare Teilgraph des Graphen in Abb. 15.2

Beweis Die Äquivalenzen sind offensichtlich. Sei v ein Knoten in einem azyklischen Graphen G. Wir wählen einen einfachen Pfad maximaler Länge, dessen Endpunkt v ist. Der Ausgangspunkt dieses Pfades muss eine Quelle sein. □

Ein Graph $G = (V, E)$ heißt **Teilgraph** eines Graphen $G' = (V', E')$ genau dann, wenn $V \subseteq V'$ und $E \subseteq E'$.

Sei $G = (V, E)$ ein Graph und $v \in V$. Der von v aus **erreichbare Teilgraph** von G ist wie folgt definiert:

$$G^v = (V',\ E \cap V' \times V') \qquad \text{mit} \qquad V' = \{v' \in V \mid v \to_G^* v'\}$$

Sei G der Graph in Abbildung 15.3. Dann gilt $G^v = G$ für jeden Knoten v von G. Abbildung 15.4 zeigt den von 6 aus erreichbaren Teilgraphen des Graphen in Abbildung 15.2.

Proposition 15.1.2 *Sei G ein Graph mit einem Knoten v. Dann ist der von v aus erreichbare Teilgraph von G zusammenhängend.*

Ein Graph heißt **waldartig**, wenn er azyklisch ist und jeder Knoten höchstens einen Vorgänger hat. Die Quellen eines waldartigen Graphen heißen auch **Wurzeln** und die Senken heißen auch **Blätter**.

Ein Graph heißt **baumartig**, wenn er waldartig ist und höchstens eine Wurzel hat.

Abbildung 15.5 zeigt zwei grafische Darstellungen eines baumartigen Graphen $G = (V, E)$ mit

$$V = \{1,\ 2,\ 3,\ 4,\ 5,\ 6,\ 7,\ 8,\ 9,\ 10,\ 11\}$$
$$E = \{(1,2),\ (1,5),\ (1,7),\ (2,3),\ (2,4),\ (5,6),\ (7,8),\ (7,11),\ (8,9),\ (8,10)\}$$

Bei der linken Darstellung handelt es sich um die übliche Darstellung für Graphen. Die rechte Darstellung stellt den baumartigen Graphen wie einen Baum dar. Die Gegenüberstellung der beiden Darstellungen zeigt die Entsprechungen zwischen baumartigen Graphen und den in Kapitel 5 eingeführten Bäumen. Der wichtigste Unterschied besteht darin, dass die Nachfolgern eines Knotens eines baumartigen Graphen ungeordnet sind, während die Unterbäume eines Baums

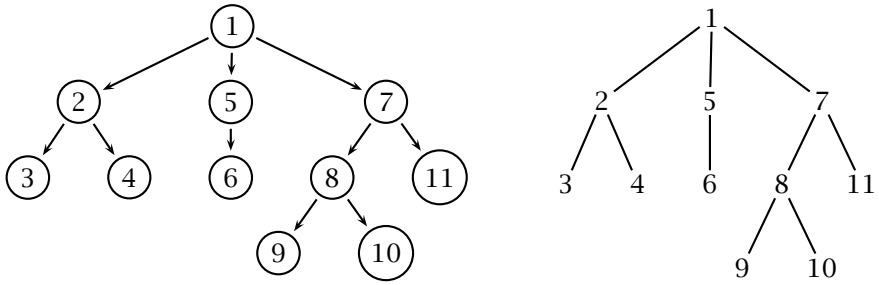


Abbildung 15.5: Ein Baum in Graph- und Baumdarstellung

linear angeordnet sind (man spricht daher auch von *geordneten Bäumen*). Die Begriffe für Graphen und Bäume sind so definiert, dass sie zusammenpassen (z.B. Größe, Tiefe oder Nachfolger).

Wenn man einen waldartigen Grafen grafisch darstellt, bekommt man tatsächlich eine Ansammlung von baumartigen Graphen.

Proposition 15.1.3 (Waldartige Graphen) *Sei G ein azyklischer Graph. Dann sind die folgenden Aussagen äquivalent:*

1. *G ist waldartig.*
2. *Jeder Knoten hat höchstens einen Vorgänger.*
3. *Zwischen zwei Knoten gibt es höchstens einen Pfad.*

Weiter gilt für jeden waldartigen Graphen W: Zu jedem Knoten v existiert genau eine Wurzel, von der aus v erreichbar ist.

Beweis Folgt mit Proposition 15.1.1. □

Proposition 15.1.4 (Baumartige Graphen) *Sei B ein nichtleerer baumartiger Graph. Dann gilt:*

1. *B hat genau eine Wurzel.*
2. *Zu jedem Knoten v gibt es genau einen Pfad von der Wurzel nach v.*
3. *B ist zusammenhängend.*

Beweis Folgt mit Proposition 15.1.3. □

Proposition 15.1.5 *Sei v ein Knoten eines baumartigen Graphen B. Dann ist der von v aus erreichbare Teilgraph von B ein baumartiger Graph mit der Wurzel v.*

© G. Smolka, 12.2.2003

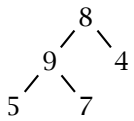
15.2 Verzeigerte Blockdarstellung

Die virtuelle Maschine aus dem letzten Kapitel kann nur mit ganzzahligen Werten rechnen. Wir erweitern die Maschine jetzt um die Fähigkeit, mit Listen, Bäumen und Referenzen zu rechnen. Diese sogenannten **dynamischen Datenstrukturen** werden in einem zusätzlichen Speicherbereich dargestellt, der als **Halde** (englisch heap) bezeichnet wird. Dynamische Datenstrukturen werden dabei als Folgen von ganzen Zahlen dargestellt. Das dabei zur Anwendung kommende Schema wird als **verzeigerte Blockdarstellung** bezeichnet.

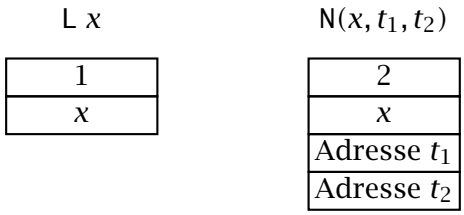
Zunächst stellen wir fest, dass die Halde eine Reihung über `int` ist. Die verzeigerte Blockdarstellung von dynamischen Datenstrukturen in der Halde erklären wir am Beispiel von Bäumen des Typs

```
datatype tree = L of int | N of int * tree * tree
```

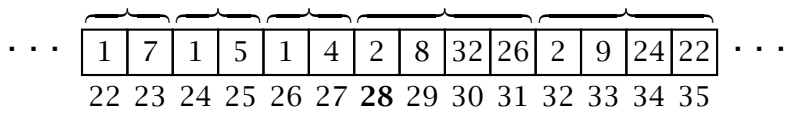
Dazu betrachten wir den Baum



Die Blätter des Baums werden mit `L` und die anderen Knoten mit `N` konstruiert. Für jeden Knoten des Baums allozieren wir in der Halde einen **Block** (eine Folge von aufeinander folgenden Zellen) gemäß dem folgenden Schema:



Die erste Zelle eines Blocks sagt, ob es sich um die Darstellung eines L- oder N-Knotens handelt. Die zweite Zelle enthält die Marke des Knotens. N-Blöcke haben zwei weitere Zellen, in denen die Adressen der Unterbäume stehen. Die Adresse eines Baums ist die Anfangsadresse des Blocks, der die Wurzel des Baums darstellt. Hier ist eine mögliche Darstellung des oben angegebenen Baums:



Die Anfangsadresse dieser Darstellung ist 28. Die die Knoten des Baums darstellenden Blöcke sind mit gescheiften Klammern markiert. Da die N-Blöcke Adres-

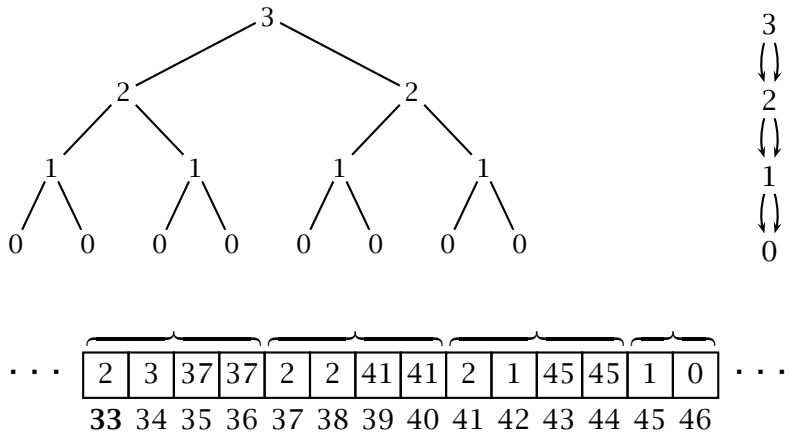


Abbildung 15.6: Darstellung eines Baums durch eine Graphstruktur

sen tragen, die auf weitere Blöcke zeigen, sagt man anschaulich, dass der Baum durch **verzeigerte Blöcke** dargestellt wird. Die Blöcke können in beliebiger Reihenfolge in der Halde liegen, und zwischen den Blöcken kann beliebig viel Zwischenraum eingefügt werden (natürlich müssen dafür die Adressen in den N-Blöcken angepasst werden).

Die Details der Darstellung der Werte des Typs `tree` in der Halde sind auf der Ebene von Standard ML unsichtbar. Beispielsweise ist es unmöglich, die Anfangsadresse der Darstellung eines Wertes zu ermitteln. Das Verbergen von semantisch irrelevanten Implementierungsdetails erlaubt wichtige Optimierungen. Insbesondere besteht die Möglichkeit, Teilbäume, die in einem Baum mehrfach auftreten, nur einmal darzustellen. Abbildung 15.6 zeigt dazu ein Beispiel. Durch die Zusammenlegung gleicher Teilbäume bekommt man eine graphartige Struktur, deren Darstellung weniger Platz als die der baumartigen Struktur erfordert. Die Abbildung zeigt unten eine Darstellung des Baums in der Halde (mit der Anfangsadresse 33), die der graphartigen Struktur entspricht.

Mit verzeigerten Blöcken kann man eine große Klasse von Graphstrukturen darstellen. Als Bespiel betrachten wir die in Abbildung 15.7 gezeigte Darstellung eines gerichteten Gaphen. Jeder Knoten des Graphen wird durch einen Block wie folgt dargestellt:

© G. Smolka, 12.2.2003

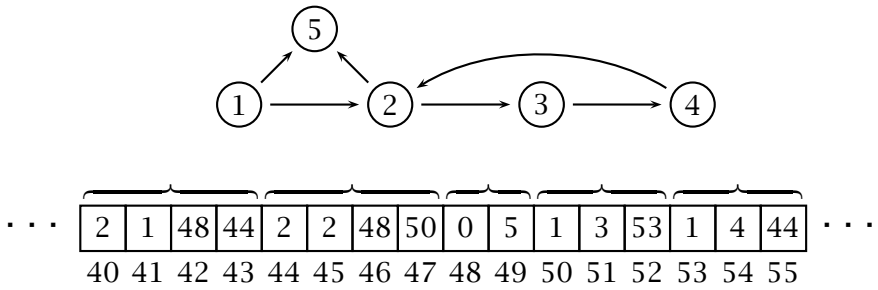
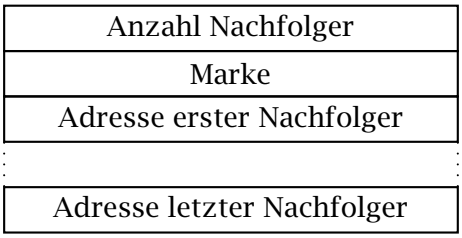


Abbildung 15.7: Verzeigerte Blockdarstellung eines gerichteten Graphen



Der den Knoten 1 darstellende Block hat die Adresse 40.

Die graphartige Struktur

kann übrigens nicht durch einen gerichteten Graphen gemäß unserer Definition beschrieben werden, da jeweils zwei Kanten von einem Knoten zum nächsten gehen. In dieser Hinsicht ist die verzeigerte Blockdarstellung ausdrucksstärker als gerichtete Graphen. Zudem ordnet eine verzeigerte Blockdarstellung die Nachfolger eines Knotens linear an.

15.3 Eine virtuelle Maschine mit Halde

Wir erweitern jetzt die virtuelle Maschine V aus Kapitel 14 um eine Halde. Die erweiterte Maschine nennen wir VH. VH hat vier neue Befehle:

- `new` *i* nimmt *i* Werte vom Stapel und alloziert in der Halde einen neuen

Block mit i Zellen, in die die Werte eingetragen werden (entsprechend der Ordnung, in der sie auf dem Stapel lagen, wobei der oberste Wert in die erste Zelle des neuen Blocks kommt). Die Anfangsadressse des neuen Blocks wird auf den Stapel gelegt.

- `getH` i nimmt die Adresse eines Blocks vom Stapel und legt den Wert in der i-ten Zelle des Blocks auf den Stapel.
- `putH` i nimmt die Adresse eines Blocks und einen Wert vom Stapel und schreibt den Wert in die i-te Zelle des Blocks.
- `eq` nimmt zwei Werte vom Stapel und testet, ob diese gleich sind. Wenn die Werte gleich sind, wird 1 auf den Stapel gelegt, sonst 0.

Jeder der vier Befehle erhöht den Programmzähler um eins. Bei den ersten drei Befehlen muss $i \geq 1$ gelten, da ein Block mindestens eine Zelle hat und die Zellen mit eins beginnend durchnummeriert sind. Bei den Werten auf dem Stapel und in der Halde handelt es sich immer um Werte aus `int`, die entweder als Zahlen oder als Anfangsadressen von Blöcken in der Halde interpretiert werden.

Als Beispiel betrachten wir die Prozedur

```
fun tree n = if n<1 then L 0
             else let val t = tree(n-1) in N(n,t,t) end

val tree : int -> int tree
```

Diese liefert zu $n \in \mathbb{N}$ einen balancierten Binärbaum der Tiefe n:

```
tree 2
N(2, N(1, L 0, L 0), N(1, L 0, L 0)) : int tree
```

Die Prozedur hat lineare Laufzeit, liefert aber, semantisch gesehen, eine exponentiell großen Baum. Das ist kein Widerspruch, da der Baum in der Halde durch eine Graphstruktur linearer Größe dargestellt wird (siehe Abbildung 15.6).

Die Prozedur kann wie folgt in eine Befehlssequenz für VH übersetzt werden (für den rekursiven Aufruf nehmen wir an, dass die Befehlssequenz mit der Adresse 0 beginnend im Programmspeicher abgelegt ist):

```
[proc(1,18),                        (* fun tree n =          *)
 con 0, getF~1, leq, cbranch 5,     (* if n<=0               *)
 con 0, con 1, new 2, return,       (* then L 0 else         *)
 con 1, getF~1, sub, call 0,        (* let val t = tree(n-1) *)
 getF 1, getF~1, con 2, new 4,      (* in N(n,t,t) end       *)
 return]
```

Die Adresse des durch den rekursiven Aufruf gelieferten Baums wird in der Zelle 1 des Aufrufrahmens abgelegt. Um den N-Knoten in der Halde darzustellen, wird sie mit `getF 1` noch einmal auf den Stapel gelegt.

© G. Smolka, 12.2.2003

Als Nächstes betrachten wir die Prozedur

```
fun top (L x)      = x
  | top (N(x,_,_)) = x
val top : tree -> int
```

Da die Marke bei L- und N-Blöcken gleichermaßen in der zweiten Zelle liegt, können wir `top` durch die folgende Befehlssequenz realisieren:

```
[proc(1,4), getF~1, getH 2, return]
```

Schließlich betrachten wir die Prozedur

```
fun sum (L x)       = x
  | sum (N(x,t,t')) = x + sum t + sum t'
val sum : tree -> int
```

die die Summe aller Marken eines Baums liefert. Die Realisierung dieser Prozedur muss zwischen L- und N-Blöcken unterscheiden können. Das ist einfach, da die erste Zelle eines Blocks die Variantennummer des Blocks enthält. Wir realisieren die Prozedur durch die folgende Befehlssequenz (für den rekursiven Aufruf nehmen wir an, dass die Befehlssequenz mit der Adresse 0 beginnend im Programmspeicher abgelegt ist):

```
[proc(1,20),                              (* fun                *)
 getF~1, getH 1, con 1, eq, cbranch 4,    (* sum L x =          *)
 getF~1, getH 2, return,                  (*           x        *)
                                          (* | sum(N(x,t,t')) = *)
 getF~1, getH 4, call 0,                  (*             sum t' *)
 getF~1, getH 3, call 0,                  (*     sum t          *)
 getF~1, getH 2, add, add,                (* x +       +        *)
 return]
```

Mit VH können wir auch Referenzen realisieren. Eine Referenz stellen wir durch einen Block mit einer Zelle dar. Die Befehlsfolge

```
[con 13, new 1]
```

legt eine neue Referenz auf den Stapel, die auf den Wert 13 zeigt. Die Gleichheit von zwei Referenzen können wir mit dem Befehl `eq` testen. Eine Prozedur

```
fun double r = !r before r:= 2 * !r
val double : int ref -> int
```

die den Wert einer Referenz verdoppelt und den alten Wert der Referenz liefert, können wir mit VH wie folgt realisieren:

```
[proc(1,9),           (* fun double r =      *)
 getF~1, getH 1,      (* !r before          *)
 getF 1, con 2, mul,  (*              2* !r  *)
 getF~1, putH 1,      (*          r:=        *)
 return]
```

15.4 Realisierung der virtuellen Maschine

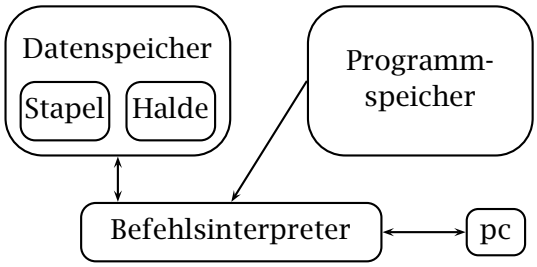
Die Halde wird durch eine Reihung über `int` realisiert. Neue Blöcke werden, beginnend mit der Adresse 0, aufeinanderfolgend in der Halde abgelegt. Wenn ein neuer Block nicht mehr in die Halde passt, gibt es zwei Möglichkeiten: Entweder bricht die Maschine die Ausführung des Programms mit einer Fehlermeldung ab, oder aber sie setzt eine sogenannte **automatische Speicherbereinigung** in Gang, die entscheidet, welche Blöcke noch benötigt werden und diese in der Halde so verschiebt, dass wieder Platz für die Allokation neuer Blöcke entsteht. Platz wird natürlich nur dann frei, wenn die Speicherverwaltung Blöcke findet, die nicht mehr benötigt werden.

Eine Realisierung von VH ohne automatische Speicherbereinigung ist einfach (Übungsaufgabe). Maschinennahe Sprachen wie C und C++ werden in der Tat mit einer Halde ohne automatische Speicherbereinigung realisiert.[2] Diese Sprachen bieten die Möglichkeit, die Halde auf eine bestimmte Adresse zurückzusetzen. Natürlich muss beim **manuellen Rücksetzen der Halde** gewährleistet sein, dass die Blöcke im freigegebenen Bereich nicht mehr zugegriffen werden, da sie durch neue Blöcke überschrieben werden. Manuelles Rücksetzen der Halde erfordert eine wohlüberlegte Programmorganisation. Wenn dabei Fehler gemacht werden, sind diese oft sehr schwer zu lokalisieren. Höhere Programmiersprachen wie Java und Standard ML werden grundsätzlich mit automatischer Speicherbereinigung realisiert.

Wir werden VH mit automatischer Speicherbereinigung realisieren. Der Implementierung von VH geben wir die folgende Architektur:

[2] Da C und C++ die Implementierung der Halde nicht verbergen, kann man sie auch nicht ohne weiteres mit automatischer Speicherbereinigung realisieren.

© G. Smolka, 12.2.2003



Der Stapel und die Halde werden zu einer Struktur zusammengefasst, die wir **Datenspeicher** nennen. Mit dieser Zusammenfassung erreichen wir, dass die Adressen der Blöcke in der Halde innerhalb des Datenspeichers enkapsuliert werden, also nicht nach außen gelangen können. Ein Block in der Halde heißt **erreichbar** genau dann, wenn er von einem Block aus erreichbar ist, dessen Adresse auf dem Stapel liegt (im Sinne eines Graphen, die Blöcke sind ja durch die Verzeigerung zu einem Graphen organisiert). Die automatische Speicherbereinigung entfernt genau die Blöcke aus der Halde, die nicht mehr erreichbar sind.

Damit wir die automatische Speicherbereinigung realisieren können, muss es möglich sein, zwischen Zahlen und Adressen (der Halde) zu unterscheiden. Beide werden jedoch als Werte von `int` dargestellt. Für VH lösen wir das Problem dadurch, dass wir alle Werte von `int`, die kleiner als eine Konstante `fha` (first heap address) sind, als Zahlen interpretieren, und alle anderen Werte als Adressen. Es wird jetzt auch klar, warum die Werte des Typs `int` bei Moscow ML nur das Intervall

$$\{-2^{30}, \ldots, 2^{30}-1\}$$

abdecken, und nicht das durch die Hardware realisierte größere Intervall[3]

$$\{-2^{31}, \ldots, 2^{31}-1\}$$

Die fehlenden Zahlen dienen als Haldenadressen.

Die automatische Speicherbereinigung muss auch wissen, wie lang ein Block ist (also aus wie vielen Zellen er besteht). Daher versehen wir jeden Block mit einer zusätzlichen Zelle, in die wir seine Länge eintragen. Diese zusätzliche Zelle ist außerhalb des Datenspeichers unsichtbar und wird an den Anfang eines Blocks gelegt. Die Länge wird mit einer Zahl dargestellt, die kleiner als `fha` ist.

Es ist jetzt problemlos möglich, die erreichbaren Blöcke zu finden. Als Nächstes stellt sich die Frage, wie die erreichbaren Blöcke so zusammengeschoben werden

[3] Bei 32-Bit-Computern, siehe Abschnitt 10.2.1.

können, dass die dazwischenliegenden, unerreichbaren Blöcke eliminiert werden. Dieses Problem werden wir etwas anders lösen als ursprünglich angedeutet.

Wir unterteilen die Halde in zwei gleich große Hälften. Zuerst allozieren wir nur in der ersten Hälfte Blöcke. Sobald ein Block alloziert werden soll, der nicht mehr in die erste Hälfte passt, starten wir die Speicherbereinigung. Diese kopiert alle erreichbaren Blöcke in die zweite Hälfte. Wenn es es unerreichbare Blöcke gibt, haben wir jetzt in der zweiten Hälfte mehr Platz als vorher in der ersten Hälfte. Wir vertauschen jetzt die Rolle der beiden Hälften und allozieren neue Blöcke in der zweiten Hälfte. Wenn diese wieder erschöpft ist, kopieren wir die dann erreichbaren Blöcke wieder in die erste Hälfte und allozieren neue Blöcke dort. Dieses einfache Verfahren nützt den verfügbaren Haldenplatz zwar nur zur Hälfte aus, aber die verbleibende Hälfte steht voll für erreichbare Blöcke zur Verfügung.

Die Speicherbereinigung könnte man jetzt durch rekursive Prozeduren realisieren. Dabei werden nicht-iterative Rekursionen benötigt, deren Tiefe sehr groß werden kann (wenn sich beispielsweise eine lange Liste in der Halde befindet). Außerdem muss sich der Kopieralgorithmus merken, welche Blöcke bereits an welche Adressen in die neue Hälfte kopiert wurden (da die Blöcke zu einer (zyklischen) Graphstruktur verzeigert sein können). Eine naive rekursive Lösung benötigt also jede Menge Platz in einem Hilfsspeicher.

Mit einer einfachen Technik ist es jedoch möglich, den Kopieralgorithmus iterativ und ohne Hilfsspeicher zu realisieren. Wenn ein Block aus der alten Hälfte in die neue Hälfte kopiert wird, schreiben wir die Adresse der Kopie in die erste Zelle des Blocks in der alten Hälfte. In dieser Zelle befindet sich normalerweise die Länge des Blocks. Da Adressen von Längen unterschieden werden können (Längen sind kleiner als `fha`), kann der Kopieralgorithmus, wenn er erneut auf den bereits kopierten Block stösst, sofort erkennen, dass dieser bereits kopiert wurde. Zudem hat er die Adresse der Kopie zur Verfügung.

Der iterative Kopieralgorithmus arbeitet in zwei Phasen:

1. Zuerst werden alle allozierten Stapelzellen besucht. Wenn eine besuchte Stapelzelle eine Adresse enthält, handelt es sich immer um einen Block in der alten Hälfte. Wenn der Block noch nicht in die neue Hälfte kopiert wurde, wird er dorthin kopiert. In jedem Fall wird in die besuchte Stapelzelle die Adresse der Kopie eingetragen.

2. Danach werden alle allozierten Zellen in der neuen Hälfte der Halde besucht. Wenn eine besuchte Haldenzelle eine Adresse enthält, handelt es sich immer um einen Block in der alten Hälfte. Wenn der Block noch nicht in die neue Hälfte kopiert wurde, wird er dorthin kopiert. In jedem Fall wird in die Adresse der Kopie die besuchte Haldenzelle eingetragen. (Überlegen Sie sich, warum das der Fall ist.)

© G. Smolka, 12.2.2003

```
type index = int
type noa   = int  (* number of arguments *)

signature Store =
sig
  val pushI   : int   -> unit
  val popI    : unit  -> int
  val pushS   : index -> unit
  val updateS : index -> unit
  val pushF   : noa   -> unit
  val pushFF  : index -> unit
  val popF    : noa   -> unit
  val clear   : unit  -> unit
  val pushB   : noa   -> unit
  val pushH   : index -> unit
  val updateH : index -> unit
  val pushEq  : unit  -> unit
end
```

Abbildung 15.8: Signatur des Datenspeichers von VH

Da auch in der zweiten Phase Blöcke in die neue Hälfte kopiert werden, ist diese erst beendet, nachdem auch die Zellen der neu allozierten Blöcke besucht wurden.

Damit sind die wesentlichen Ideen für die Implementierung von VH klar. Wir schreiten jetzt zur Tat. Unser Ausgangspunkt ist die Implementierung von V (siehe Kapitel 14). Die neuen Instruktionen von VH wurden in Abschnitt 15.3 beschrieben. Die Struktur `Stack` von V ersetzen wir durch eine Stuktur `Store`, die den Datenspeicher von VH implementiert. Abbildung 15.8 zeigt die Signatur von `Store`. Die ersten sieben Operationen betreffen den Stapel, der genau wie bei V realisiert wird (bis auf eine kleine Änderung bei `pushI` und `popI`, auf die wir später eingehen). Die Operation `clear` setzt den Stapel und die Halde in den Anfangszustand zurück. Die restlichen vier Operationen betreffen die Halde. Ihre Semantik ergibt sich aus der Semantik der neuen Befehle von VH (siehe Abschnitt 15.3) und der in Abbildung 15.9 gezeigten Erweiterung des Befehlsinterpreters.

Es fehlt jetzt nur noch die Implementierung des Datenspeichers. Den Stapel von VH implementieren wir so wie den von V, mit der kleinen Änderung, dass wir die Prozeduren `popI` und `pushI` jetzt mit `pop` und `push` bezeichnen. Die nach außen sichtbaren Prozeduren `popI` und `pushI` realisieren wir mit den internen Prozeduren `pop` und `push`, wobei wir sicherstellen, dass mit `popI` und `pushI` nur Zahlen kommuniziert werden können (also keine Adressen):

```
fun execute instruction =
  case instruction of
    ...
  | new noa        => (pushB noa ; pc:= !pc+1)
  | getH i         => (pushH i   ; pc:= !pc+1)
  | putH i         => (updateH i ; pc:= !pc+1)
  | eq             => (pushEq()  ; pc:= !pc+1)
```

Abbildung 15.9: Erweiterung des Befehlsinterpreters für VH

```
fun isI v = v<fha

fun pushI v = if isI v then push v
              else raise Error "number overflow"

fun popI () = let val v = pop()
              in  if isI v then v
                  else raise Error "number expected"
              end
```

Jetzt fehlt nur noch die Implementierung der Halde. Diese finden Sie in den Abbildungen 15.10 und 15.11. Für die Halde gibt es zwei Zeiger, `bp` (base pointer) und `hp` (heap pointer). Der Zeiger `bp` zeigt immer auf die erste Zelle der gerade aktuellen Haldenhälfte. Der Zeiger `hp` zeigt immer auf die zuletzt allozierte Zelle in der gerade aktuellen Haldenhälfte. Wenn ein neuer Block in die Halde gelegt wird, bekommt er die Anfangsadresse `!hp` + 1 und `hp` wird entsprechend erhöht.

15.5 Dynamische Prozeduren

15.6 Effiziente Realisierung kaskadierter Prozeduren

© G. Smolka, 12.2.2003

```
val heapSize = 100

val heap = Array.array(2*heapSize, 0)

val fha = valOf(Int.maxInt)-2*heapSize+1  (* first heap address *)

val bp = ref fha     (* first cell in current heap section *)

val hp = ref(fha-1)  (* topmost heap cell allocated *)

fun getH ha = Array.sub(heap, ha-fha)

fun putH(ha,v) = Array.update(heap, ha-fha, v)

fun alloc v = (hp:= !hp+1 ; putH(!hp,v))

fun pushH i = let val a = pop()
                  val l = getH a
              in  if 0<i andalso i<l then push(getH(a+i))
                  else raise Error "illegal block index (get)"
              end

fun updateH i = let val a = pop()
                    val l = getH a
                in  if 0<i andalso i<l then putH(a+i, pop())
                    else raise Error "illegal block index (put)"
                end

val wc  = ref ~1  (* counter for while loop  *)

fun gc () = ···

fun pushB' noa = if !hp-heapSize > !bp-noa-2
                   then raise Error "heap overflow"
                 else (alloc(noa+1) ;
                       wc:= noa ;
                       while !wc>0 do
                         (alloc(pop()) ; wc:= !wc-1) ;
                       push(!hp-noa) )

fun pushB noa = if 0>=noa orelse noa>=heapSize
                  then raise Error "illegal block allocation"
                else pushB' noa handle _ => (gc() ; pushB' noa)
```

Abbildung 15.10: Implementierung der Halde

```
fun isI v = v<fha

val iwc = ref ~1  (* counter for inner while loop  *)

fun heap2heap(ha,n) = (iwc:= ha ;
                       while !iwc < ha+n do
                         (alloc(getH(!iwc)) ; iwc:= !iwc+1) )

fun gcCopyBlock oldba =
  if not(isI(getH oldba)) then getH oldba
  else let
         val newba = !hp+1
       in
         heap2heap(oldba, getH oldba) ;
         putH(oldba,newba) ;
         newba
       end

fun gcCopyStack () =
  (wc:= 0 ;              (* wc points to stack cell *)
   while !wc <= !sp do
     (if isI(getS(!wc)) then ()
      else putS(!wc, gcCopyBlock(getS(!wc))) ;
      wc:= !wc+1 ) )

fun gcCopyHeap () =
  (wc:= !bp ;            (* wc points to heap cell *)
   while !wc <= !hp do
     (if isI(getH(!wc)) then ()
      else putH(!wc, gcCopyBlock(getH(!wc))) ;
      wc:= !wc+1 ) )

fun gc () = (bp:= !bp + (if !bp=fha then 1 else ~1) * heapSize ;
             hp:= !bp-1 ;
             gcCopyStack() ;
             gcCopyHeap() )
```

Abbildung 15.11: Implementierung der Speicherbereinigung

© G. Smolka, 12.2.2003