Zur Rückführung nicht mehr benötigten Speicherplatzes in PASCAL

Dynamic memory recovery in PASCAL

Von S. M. Rump
Institut für Angewandte Mathematik,
Universität Karlsruhe

Elektron. Rechenanl. 22 (1980), H. 2, S. 55–62
Manuskripteingang: 26. Juli 1979

Bei Benutzung der Datenstruktur Pointer in PASCAL tritt selbst bei kleineren Programmen schnell Speicherüberlauf ein, wenn nicht entsprechende Vorkehrungen getroffen werden. Aber gerade die Möglichkeit dynamischer Variablen ist ein entscheidendes Merkmal von PASCAL, so daß es der Verfasser für dringend angeraten hält, Möglichkeiten zur weiteren Verwendung nicht mehr benötigten Speicherplatzes zu schaffen. In der vorliegenden Arbeit werden zunächst Methoden beschrieben, wie diese Rückführung vom Benutzer bewerkstelligt werden kann. Gleichwohl ist es für eine moderne Programmiersprache unzumutbar, daß zum einen etwas derart Fundamentales vom Benutzer implementiert werden soll und zum andern dies ja für jedes Programm und Unterprogramm von neuem geschehen muß. In einem zweiten Teil der Arbeit werden daher Methoden beschrieben, wie die dynamische Speicherplatzverwaltung durch den Compiler geschehen kann. Im Gegensatz zu bekannten Methoden wird ein Kompromiß zwischen möglichst optimaler Rückführung und schneller Rechenzeit gesucht. Der Verfasser hofft dazu beizutragen, daß die unbedingte Notwendigkeit der Speicherplatzrückführung erkannt und in den Report aufgenommen wird und, daß bald mehr Compiler mit dieser Fähigkeit auf den Markt kommen.

When using the data structure pointer even for small programs the memory is filled up quickly if no precautions were made. However, especially the possibility of dynamic variables is a fundamental feature of PASCAL, so that in the authers opinion it is of great importance to manage the further use of not-more-needed memory. In the present paper first methods are described how this can be done by the programmer. However, for a modern programming language it is a tall order to leave this important task to the user and, moreover, this has to be done for every program and subprogram separately. Therefore, in a second chapter methods will be described, how the compiler can manage the dynamic memory recovery in PASCAL. In contrast to known methods here it has been looked for a compromise between optimal recovery and time efficiency. The author hopes, that the fundamental necessity of recovering memory not longer needed will be recognized and entered in the report and, that more compilers with this ability will come into the market.

Einleitung

Nikolaus Wirth schreibt in seinem PASCAL-Report, daß eine der wesentlichen Erweiterungen der Programmiersprache PASCAL in bezug auf ALGOL die Möglichkeiten im Bereich der Datenstrukturen sind. Die Einführung von Record- und File-Strukturen ermöglicht es, in PASCAL auch für kommerzielle Probleme wirksam eingesetzt zu werden. Eine entscheidende Rolle spielt hierbei der Datentyp pointer. Durch ihn wird ermöglicht, Variable während der Abwicklung des Programms laufend zu generieren. Vornehmlich in Verbindung mit Records wird so die Möglichkeit der Listenverarbeitung geschaffen. Aus Erfahrung mit anderen Systemen wie etwa LISP ist jedoch bekannt, daß der sinnvolle Einsatz der Listenverarbeitung in der Praxis erst durch eine Möglichkeit der Wiedereingliederung nicht mehr benötigter Speicherplätze tatsächlich realisiert wird. Dies ist in PASCAL insbesondere deshalb von besonderer Wichtigkeit, da in manchen PASCAL-Compilern bei der Vereinbarung einer File-Variablen gleich 600 oder mehr Speicherplätze von vornherein reserviert werden und so ein sinnvolles Arbeiten mit File-Variablen in großen Programmen u. U. erschwert wird (dies ist z. B. beim PASCAL-Compiler der UNIVAC 1108 an der Universität Karlsruhe der Fall).

Der Verfasser hatte nun ein größeres Programm zu schreiben, bei dem die Verwendung von Files als Datenstruktur durch den gerade beschriebenen Umstand von vornherein ausgeschlossen war. Andererseits wird bei der Verwendung von Pointern der Speicherplatz immer mehr aufgebraucht, und die Erfahrung zeigte, daß selbst bei kleineren Programmen sehr schnell ein Speicherüberlauf eintritt. Im PASCAL-Report ist eine Standardprozedur dispose vorgesehen, um den durch einen Aufruf von new reservierten Speicherplatz wieder zur Verfügung zu stellen. Da die verschiedenen Implementierungen hierbei nicht einheitlich vorgehen, empfiehlt *Grogono* zur Überprüfung ein Testprogramm „testdispose". Im günstigsten Fall wirkt dispose komplementär zu new; d. h., die Anwendung von dispose macht *einen* Speicherplatz (auf den der entsprechende Pointer zeigt) wieder verfügbar. Will man etwa den Ast eines Baumes oder eine ganze Listenstruktur wieder bereitstellen, muß dies mühselig Knoten für Knoten geschehen. Die gesamte Verwaltung wird

dem Programmierer aufgebürdet, und das für jedes einzelne Programm und Unterprogramm. Ganz abgesehen davon, daß die entstehenden Programme lang und unübersichtlich werden, bedeutet dies eine gefährliche Fehlerquelle. An der UNIVAC 1108 der Universität Karlsruhe gibt es die Möglichkeit, die Operatoren mark und release zu verwenden. Wird jedoch ein Zwischenergebnis berechnet und in einer Liste gespeichert, so steht diese Liste ja am Ende einer Folge von Operationen. Um mark und release also überhaupt einsetzen zu können, müssen die in der Liste enthaltenen Informationen zunächst ausgelagert werden, um nicht verlorenzugehen. Gleichzeitig gilt entsprechend die Bemerkung zu dispose. Insgesamt kann man sich durch solche Tricks behelfen, doch es bleibt ein Behelf und im höchsten Maße unbefriedigend. Nicht nur, daß ein solches Unterfangen viel Mühe bereitet, meist wird doch nur ein Teil des nicht mehr benötigten Speicherplatzes wieder freigegeben und allzu leicht passiert es, daß ein Speicherplatz zu früh als nicht mehr benötigt rückgeführt wird, neu beschrieben wird und so ein fast unauffindbarer Fehler entsteht.

Es wurden also Überlegungen angestellt, eine einfachere Methode zur Rückführung nicht mehr benötigten Speicherplatzes zu finden. Die Methode sollte einerseits dem Benutzer möglichst keine Mehrarbeit abverlangen, andererseits aber auch nicht allzu zeitintensiv sein. Die bekannten Methoden sind meist im einen oder anderen Punkt nicht zufriedenstellend.

1. Bekannte Methoden, Vorteile, Nachteile

Betrachtet man sich den Ablauf eines Programmes und das Problem der Speicherplatzrückführung, sieht man sich mit zwei extremen Möglichkeiten konfrontiert: Entweder kümmert man sich um gar nichts und der Speicher füllt sich immer mehr bis kein freier Speicherplatz mehr zur Verfügung steht und mit einer Fehlermeldung abgebrochen wird: Dies ist bis jetzt in den meisten PASCAL-Compilern der Fall. Oder man versucht, jeden einzelnen Speicherplatz sofort wieder verfügbar zu machen, sobald er nicht mehr benötigt wird. Wie wir sehen werden, erfordert die Realisierung des zweiten Extrems viel Zeit und Aufwand, so daß der Gewinn nicht gerechtfertigt erscheint. Als Folge wird man nach dem goldenen Mittelweg suchen, der möglichst viel nicht mehr benötigte Speicherplätze zurückgibt, andererseits aber möglichst rationell arbeitet.

Betrachten wir zunächst manuelle Methoden zur Speicherplatzrückführung, d.h., welche Möglichkeiten hat der Benutzer nicht mehr benötigte Speicherplätze wieder verfügbar zu machen. Wird eine dynamisch angelegte Variable einmal für den Rest des Programms nicht mehr benötigt, kann sie sofort einen anderen Wert bekommen ohne den Programmablauf insgesamt und insbesondere die Endergebnisse zu beeinflussen. Ab diesem Zeitpunkt also kann sie den Status „wieder verfügbar" oder „free" bekommen. Bei der nächsten dynamischen Vereinbarung einer Variablen gleichen Typs könnte demnach auf die erneute Reservierung von Speicherplatz verzichtet und statt dessen auf diese oder eine andere Variable gleichen Typs mit dem Status „free" zugegriffen werden. Zugegriffen heißt einfach, daß der ohnehin bereits lange vorher reservierte Speicherplatz der neu zu vereinbarenden Variablen zugeteilt wird. Hat also etwa die Variable p den Status „free" und soll für q (eine Variable vom gleichen Typ wie p) neuer Speicherplatz (dynamisch) reserviert werden, heißt die Zuweisung

$q := p;$ statt new(q);

Man sieht sofort zwei Probleme auftauchen: Erstens muß von einer Variablen der

- Status free „irgendwie" erkannt werden und
- zweitens sind diese Variablen zu verwalten,

um bei einem Aufruf von new gegebenenfalls statt dessen auf bereits reservierte Speicherplätze zurückgreifen zu können. Lassen wir das erste Problem zunächst einmal beiseite und konzentrieren uns auf das zweite. Eine naheliegende Methode ist, eine lineare Liste der Variablen vom Status „free" anzulegen. Jedesmal, wenn von einer Variablen „irgendwie" der Status „free" erkannt wurde, wird sie in diese Liste aufgenommen. Wird eine Variable (dieses Typs) benötigt, schaut man zunächst in der Liste nach. Ist sie nicht leer, kann von dort bereits reservierter Speicherplatz übergeben werden, andernfalls ist mittels new neuer Speicherplatz anzufordern. Ist t der Typ der zu verwaltenden dynamischen Variablen, sieht ein entsprechender Programmausschnitt etwa wie folgt aus:

```
TYPE link = ↑list;
     list = RECORD
              element : t;
              successor : link
            END;
VAR free : link;
```

Es ist klar, daß für jeden Typ einer dynamischen Variablen eine extra Liste der zu verwaltenden Variablen definiert werden muß, also etwa free1, free2,... Wird ein neuer Speicherplatz benötigt oder soll ein nicht mehr benötigter Speicherplatz als wieder verfügbar eingestuft werden (also den Status „free" bekommen), sind die folgenden Prozeduren create und release zu verwenden:

```
PROCEDURE create(VAR p : t);
  BEGIN IF free = NIL
          THEN new(p)
          ELSE BEGIN p := free↑.element;
                     free := free↑.successor
               END
  END; {create}
PROCEDURE release(p:t);
  VAR d : link;
  BEGIN new(d); d↑.successor := free;
        free := d; free↑.element := p
  END; {release}
```

Die Initialisierung des Hauptprogramms enthält die Anweisung

```
free := NIL;
```

so daß die Liste free zu Beginn der Ausführung leer ist.

Ein entscheidender Nachteil der hier gezeigten Verwaltung der Speicherplätze des Status „free" ist, daß insgesamt der doppelte Speicherplatz für dynamische Variable benötigt wird. Denn bei jedem Aufruf von release wird der Speicherplatz nochmals reserviert. Unbedingte Voraussetzung für ein einigermaßen effizientes Arbeiten dieses Systems ist also, daß die Anzahl der auf einmal benötigten Speicherplätze nicht zu hoch ist und nicht mehr benötigte Speicherplätze sofort mittels release verfügbar gemacht werden. Ein Vorteil des Systems ist allerdings, daß ein vorhandenes Programm ohne größere Änderungen verwendet werden kann. Außer den oben eingeführten Typvereinbarungen und Unterprogrammen sind nur jeweils new durch create zu ersetzen (was vom Editor erledigt werden kann) und an bestimmten Stellen release aufzurufen, worauf wir noch zu sprechen kommen. Für Variablen unterschiedlichen Typs sind allerdings verschiedene Listen free1, free2,... und ebenso verschiedene Prozeduren create1, create2,... und release1, release2,... zu verwenden. Insbesondere sind die Typen zweier Variablen gleichen Typs jedoch mit unterschiedlichem Variantenteil als verschieden anzusehen. Hat ein Record also ineinander geschaltete Varianten, kann eine größere Anzahl von Typen entstehen. Dieser Nachteil wird vermieden, wenn eine dynamische Variable p mit z. B. dem Tag-Feld t statt mit new(p,t) immer mit new(p) vereinbart wird. In diesem Fall wird zwar nur eine Liste free und jeweils nur eine Prozedur create und release benötigt, doch gleichzeitig werden die Vorteile der Records mit Variantenteil aufgegeben. In *Grogono*, Seite 270/71, wird eine andere Möglichkeit der Definition von create und release angegeben. Ein Nachteil des dort eingeführten Typs „cell" ist, daß größere Änderungen des gesamten Programms notwendig werden. Dort wird jeder mögliche Typ einer dynamischen Variablen als Variante in einen Typ „cell" hereingenommen. Dadurch erhält man sozusagen universelle Variable. Daher müssen sämtliche Typvereinbarungen und Variablenvereinbarungen von dynamischen Variablen eines Programms geändert werden. Vorteil der dort gezeigten Methode ist, daß insgesamt nur ein Typ „link" und je eine Prozedur create und release notwendig sind.

In der Prozedur create wird jedoch jedesmal der Speicherplatz für eine Variable vom Typ „link" reserviert. Wir haben vorhin bereits gesehen, daß sämtliche Typen dynamischer Variablen in diesem Typ zusammengefaßt sind. Bei jedem Aufruf von create wird also jeweils Speicherplatz für je eine Variable *jeden* Typs reserviert. Gibt es im Programm insgesamt k Typen t_i und benötigt eine Variable des Typs t_i (und zwar inklusive aller Varianten) je n_i Speicherplätze, werden bei jedem Aufruf von create $n_1 + n_2 + \ldots + n_k =: n$ Speicherplätze reserviert, beim Aufruf von release für eine Variable vom Typ t_i jedoch nur n_i Speicherplätze wieder verfügbar gemacht. Ist l die Anzahl der auf einmal auftretenden Variablen, müssen mindestens $l \cdot n$ Speicherplätze auf dem Heap verfügbar sein. Ein effizientes Arbeiten dieser Version der Verwaltung der Speicherplätze des Status „free" ist also nur in Programmen mit wenigen Typen dynamischer Variablen gewährleistet. In diesem Fall kommen zu den für jede Variable ohnehin benötigten Speicherplätzen noch die für die Variante free hinzu.

Eine andere, jedoch illegale Möglichkeit, die trotzdem an vielen Compilern anwendbar ist und Nachteile der beiden oben beschriebenen Methoden teilweise vermeidet, ist die folgende. Mit der globalen Vereinbarung

```
TYPE list = FILE OF integer;
VAR  free : list;
```

werden die dynamischen Variablen des Status „free" nicht mehr selbst, sondern deren Speicheradressen verwaltet. Diese ist i. a. vom Typ integer. An vielen Compilern ist es möglich, eine Pointervariable in einem Unterprogramm als integer zu vereinbaren. Somit erhalten wir die Prozeduren

```
PROCEDURE create(VAR n : integer; b : boolean);
    BEGIN b := false;
          IF eof(free) THEN b := true
                       ELSE read(free, n)
          END; {create}

PROCEDURE release(n : integer);
    BEGIN write(free,n)
          END; {release}
```

Für eine dynamische Variable p des Status „free" wird die Information der Verfügbarkeit mit

```
release(p);
```

an die Speicherverwaltung übergeben. Ein Aufruf von new ist durch

```
create(p,b); IF b THEN new(p);
```

mit einer logischen Variablen b zu ersetzen. Nachteil des Systems ist, daß für dynamische Variable verschiedenen Typs jeweils ein neues File angelegt und Prozeduren create und release geschrieben werden müssen. Dafür ist hier der zusätzliche Speicheraufwand minimal; denn was nützt eine aufwendige Speicherplatzrückführung, wenn diese selbst relativ viel Speicherplatz benötigt.

Jetzt kommen wir zu dem oben angesprochenen wesentlich schwierigeren Problem des Erkennens, wann eine Variable den Status „free" hat. Eine primitive Lösung des Problems ist, die Sorge hierfür ganz dem Benutzer zu überlassen. Dazu ist nichts weiter zu sagen, denn für jede Variable gibt es einen eindeutig bestimmten Punkt im Programm, wo sie zum letzten Mal verwendet wird, und dieser Punkt ist „im Prinzip" (dem Programmierer) be-

kannt. Mit dieser Bemerkung ist dem Leser natürlich herzlich wenig gedient. Eine systematische Methode zur Erkennung nicht mehr benötigter Speicherplätze, die auch automatisiert werden kann, setzt in irgendeiner Form Strukturiertheit des Programms voraus. Eine naheliegende Methode scheint daher über die Blockstruktur zu laufen. Jede Variable, die lokal in bezug auf einen anderen Block ist (in dem Sinn, daß sie oder ihr Speicherinhalt in keinem globaleren Block verwendet wird), wird dort mit Sicherheit nicht mehr benötigt. Auf diese Weise ist dem Programmierer zumindest ein Hilfsmittel an die Hand gegeben, für eine ganze Reihe von Variablen deren Status zu entscheiden. Ein großes technisches Problem ist hierbei, daß es sich nicht um einfache Variablen, sondern um ganze Listenstrukturen handelt. So ist es nicht trivial in einem Baum zu erkennen, welche Äste lokal und welche global sind. Möglicherweise können einzelne Knoten global sein, weil irgendein Zeiger einer globalen Variablen auf diesen Knoten zeigt. Es ist also nicht möglich, bei jedem Verlassen eines Blocks B_1 in einen globaleren Block B_2 als letzte Anweisungen in B_1 alle in bezug auf B_2 lokalen dynamischen Variablen mittels release wieder verfügbar zu machen. Der Programmierer muß von jeder Variablen wissen, ob ein globalerer Zeiger auf sie weist. Bei der Automatisierung der Methode muß dies auf anderem Wege festgestellt werden. Auf jeden Fall muß bei der manuellen Speicherplatzrückführung dies für jede einzelne Variable geschehen. Dadurch wird ein immenser Aufwand an Programmierung notwendig. Sehr schwerwiegend ist hierbei, daß bei diesen teilweise recht diffizilen Problemen der Programmierer viel Zeit verschwendet und überdies eine äußerst gefährliche Fehlerquelle entsteht. Wird nämlich eine Variable zu früh als wieder verwendbar erklärt, entsteht ein fast unauffindbarer Fehler. Zudem wird das entstehende Programm wesentlich langsamer, da jede Reservierung von Speicherplatz und jede Rückführung von Speicherplatz einen zeitaufwendigen Unterprogrammaufruf bedeutet. All dies spricht dafür, von der manuellen zur automatischen Speicherplatzrückführung überzugehen. Die hierfür existierenden Verfahren werden jetzt vorgestellt.

Die wohl bekannteste Methode ist der Garbagecollector (GC). Er benötigt eine Liste der im Gebrauch befindlichen Variablen. Diese wird vom Compiler sowieso angelegt und mit Hilfe der Unterprogramm-Struktur laufend auf den neuesten Stand gebracht. Diese Liste wird vom GC durchlaufen und alle von diesen Variablen erreichbaren Speicherplätze markiert. Die nach diesem Vorgang unmarkierten Speicherplätze sind nicht erreichbar und werden mit Sicherheit für den weiteren Verlauf des Programms nicht mehr benötigt und werden wieder verfügbar gemacht. In den markierten Speicherplätzen wird sodann die Markierung gelöscht, womit der Vorgang abgeschlossen ist. Der GC setzt ein, sobald kein freier Speicherplatz mehr verfügbar ist. Die Methode wurde erstmals von *McCarthy* 1960 vorgestellt. Für das Auffinden aller in Gebrauch befindlichen Speicherplätze wird jedoch ein rekursiver Suchalgorithmus benötigt, der, wie die Erfahrung zeigte, recht viel Zeit in Anspruch nehmen kann. Insbesondere, wenn fast alle Speicherplätze in Gebrauch sind, macht die Zeit für den GC schon einen größeren Prozentsatz der Gesamtzeit aus. Speziell hierfür wird manchmal vereinbart, daß wenigstens k% des gesamten Speicherplatzes rückgeführt werden muß. Doch auch diese Vereinbarung scheint das Problem nicht befriedigend zu lösen. Denn die Speicherplatzrückführung ist ja gerade dann interessant, wenn fast aller Speicherplatz in Gebrauch ist und nicht *mehr* Speicherplatz angegeben werden kann. In diesem Fall sind jedoch gleich mehrere zehntausend Speicherplätze zu durchforsten, und der GC verschlingt sehr viel Zeit. Ein weiterer Vorschlag ist die maximal collection rule. Sie geht davon aus, daß jede Variable sofort auf Null gesetzt oder irgendwie gekennzeichnet wird, sobald sie nicht mehr benötigt wird. Mit dieser Regel wird das vorhin beschriebene zweite Extrem praktisch erreicht; allerdings nur, wenn sie strikt eingehalten wird. Die Befolgung der Regel würde jedoch für den Benutzer immense Mehrarbeit bedeuten und seine Programme stark vergrößern; ein Aufwand, der in keinem Verhältnis zum Nutzen steht. Zur Verteidigung des GC muß gesagt werden, daß der Benutzer mit ihm kaum Mehrarbeit hat. Er schreibt seine Programme wie gewöhnlich und braucht sich um die Speicherplatzrückführung nicht zu kümmern, sie wird automatisch erledigt. Das „unverhoffte" Einsetzen des GC-Algorithmus hat jedoch noch weitere Folgen. Zum einen wird ein Real-time-Vergleich schwierig. Des weiteren können beim Arbeiten am Terminal zuweilen größere Wartezeiten auftreten, wenn plötzlich der GC aufgerufen wird. Man sollte auch nicht den großen rekursiven Suchalgorithmus als potentielle Fehlerquelle unterschätzen. Schon mancher Algorithmus stellte sich nach langer einwandfreier Tätigkeit als fehlerhaft heraus, wenn an irgendeiner Stelle unvermutet der GC einsetzte.

Eine andere Möglichkeit stellt die reference count-Methode dar. Hier bekommt jedes Listenelement einen Zähler, den reference count, der jeweils angibt, wieviele Zeiger auf das entsprechende Listenelement zeigen. Jedesmal, wenn ein neuer Zeiger auf das Listenelement zeigt, muß der reference count explizit um eins erhöht werden, und jedesmal, wenn eine Referenz nicht mehr benötigt wird, explizit um eins erniedrigt werden. Ist kein Speicherplatz mehr verfügbar, werden alle Zellen tatsächlich gebraucht und das Programm mit einer Fehlermeldung abgebrochen. Die Methode wurde Ende 1960 von Collins eingeführt. Man sieht sofort, daß das besprochene zweite Extrem bei entsprechender Programmierung tatsächlich erreicht wird. Das updating (richtige Programmierung vorausgesetzt) findet laufend statt. Insbesondere wird kein aufwendiger rekursiver Suchalgorithmus benötigt, die Methode arbeitet recht effizient; der Programmierer am Terminal wird diese Tatsache zu würdigen wissen. Dieser Vorteil wird jedoch mit sehr mühsamer Programmierung erkauft. Der Benutzer muß nicht nur die gesamte Buchführung seiner Variablen und Listen im Kopf haben, sondern ihm darf in dieser Hinsicht auch nicht

der kleinste Fehler unterlaufen. Andernfalls wird nämlich ein Listenelement irgendwo benötigt, wo es nicht mehr vorhanden ist. Der Fehler ist jedoch an einer ganz anderen Stelle passiert, so daß die Fehlersuche u. U. recht mühsam werden kann. Das laufende Erhöhen und Erniedrigen des reference count ist auf die Dauer sehr ermüdend und bildet eine brisante Fehlerquelle. Wer einmal mit einem solchen System gearbeitet hat, kennt sicher die Sorgen, wenn nach Ablauf eines Programms nicht mehr alle Speicherplätze verfügbar sind. Ganz abgesehen davon werden die Programme sehr lang, das Programmieren wird träge. Darüber hinaus können rekursive und ringförmige Ketten Schwierigkeiten bei der Rückgliederung bereiten. Bis jetzt wurde immer stillschweigend davon ausgegangen, daß jedes Listenelement einen Zähler, den reference count, zugeordnet bekommt. Hierfür müssen jedoch extra Bits zur Verfügung gestellt werden. Nach einem Vorschlag von Teer könnte man den reference count bis k zählen und alle Elemente, die einmal einen reference count größer als k bekommen, als resident und damit nicht rückführbar erklären. Eine Lösung, die eine solche Regelung vermeiden und den reference count prinzipiell immer weiter zählen soll, erfordert einen komplizierten Verwaltungsaufwand (siehe etwa das SAC-1 System, z. B. in der Arbeit von Lauer-Sämann).

Die vorstehenden Ausführungen machten deutlich, daß der GC und die reference count-Methode sich mehr an unseren vorhin erwähnten Extremstellen bewegen: entweder zu hoher Zeitaufwand oder zu große Mühe für den Benutzer. Eine Art Verbindung von beiden Systemen von Teer ist mehr für seinen speziellen Fall zugeschnitten und ist für unsere Zwecke nicht brauchbar. Ein Mittelweg, der die Vorteile der beiden Systeme verbindet, soll im nächsten § vorgestellt werden.

2. Beschreibung der neuen Methode

Nach dem Prinzip des GC wird ja zunächst immer nur Speicherplatz verbraucht, und das solange, bis kein Platz mehr da ist. Erst dann setzt der GC ein und versucht, alle nicht mehr benötigten Speicherplätze zu finden. Bei dieser Suche muß jedoch jeder Speicherplatz wenigstens einmal angesehen werden, demnach beträgt die Rechenzeit mindestens Anzahl der überhaupt verfügbaren Speicherplätze mal einer Konstante (beim GC werden alle Speicherplätze tatsächlich sogar zweimal angesehen). Hinzu kommen die beschriebenen Effekte, wenn der Speicher nahezu voll ist und mit einer hohen Wahrscheinlichkeit umsonst Rechenzeit verbraucht wird, wenn der Speicher dann im Endeffekt doch nicht ausreicht. Es scheint ohnehin recht sorglos, einfach immer Speicherplätze zu verbrauchen und sich erst, wenn keine mehr vorhanden sind, Gedanken darüber zu machen, wie nicht mehr benötigte Speicherplätze herausgefunden werden können. Man wird also nach einer Methode suchen, die die nicht mehr benötigten Speicherplätze dynamisch, d. h. ständig, zurückgibt. In dieser Hinsicht macht der reference count schon eine gute Figur (wenn richtig programmiert wird), und er wäre bereits die Lösung, wenn nicht seine vorhin beschriebenen Nachteile *so* überwiegend wären. Im Prinzip hätten wir eine Lösung schon gefunden, wenn der reference count automatisiert werden könnte. Nun ist das nicht ganz so einfach, denn wie soll während der Ausführung eines Programms bereits festgestellt werden, was bis zum Ende seiner Ausführung noch alles geschieht und was noch benötigt wird. Eine wertvolle Hilfe wird sicher die Unterprogrammstruktur des Programms sein. Bei „gewöhnlichen" Variablen steht ja bereits vor der Ausführung fest, wo und wann sie benötigt werden. Bei Pointervariablen ist das nun nicht ganz so einfach. Bei genauerem Hinsehen stellt sich jedoch als einzige Schwierigkeit heraus, daß eine „globalere" Variable auf eine „lokale" Variable zeigt und bei Verlassen des „lokaleren" Unterprogramms u. U. nicht mehr richtig wiedergegeben würde. Bevor wir weiter ins Detail gehen, wollen wir diesen Umstand näher präzisieren.

Zunächst führen wir eine globale Variable level ein. Sie wird zu Beginn mit 0 initialisiert, bei jedem Aufruf eines Unterprogrammes um eins erhöht und bei jedem Rücksprung aus einem Unterprogramm wieder um eins erniedrigt. Ihr Wert zu Beginn und nach Ausführung des Programms ist also Null. Betrachten wir nun etwa folgende Situation (Bild 1):

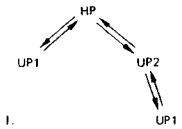


Bild 1.

Das Diagramm ist von links nach rechts zu lesen, d. h., der Programmablauf ist wie folgt (Bild 2):

Zeile 0	Start (Hauptprogramm)	Level 0
1	Aufruf von UP1	1
2	Rücksprung ins HP	0
3	Aufruf von UP2	1
4	Aufruf von UP1	2
5	Rücksprung in UP2	1
6	Rücksprung ins HP	0
7	Ende	

Bild 2.

Angenommen, in UP1 und UP2 sind zwei Pointervariable $p1$ und $p2$ vom gleichen Typ vereinbart, respektive. In Zeile 5 von Figur 2 wird nun der Platz der Variable $p1$ auf dem Stack sowieso gelöscht. Würde nun nach dem Vorschlag von vorhin auch $p11$ auf dem Heap gelöscht, so wäre eine Anweisung

$$p2 := p1; \tag{1}$$

also im UP1 unzulässig, da $p21$ nach dem Rücksprung in UP2 nicht mehr vollständig verfügbar wäre. Die einfachste Lösung des Problems ist natürlich, ein solches Vorgehen kurzerhand zu verbieten. Ganz so einfach liegt die Sache natürlich nicht, denn es müssen schon Alternativen angeboten werden, wie (1) programmiert werden kann (zweifelsohne tritt die Situation laufend auf, z. B.

beim Aufruf einer Prozedur, deren Ergebnis eine Liste ist). Zunächst wollen wir jedoch als Ausgangspunkt festhalten:

> Grundregel: Jede Variable lebt genau so lange, wie der level ihrer Vereinbarung angibt.

Das heißt im Klartext, daß jede Variable sofort gelöscht wird, sobald der nächst globalere level erreicht wird, also in den aufrufenden Block oder einen globaleren Block zurückgesprungen wird. Wie diese Löschung im einzelnen aussieht, werden wir später noch sehen; gehen wir für den Moment davon aus, daß Variable nur in ihrem Vereinbarungslevel und untergeordneten Blöcken vorhanden sind. Gehen wir mehr ins Detail, stellt sich die Frage, was bei der Anweisung (1) überhaupt im Widerspruch zu der eben ausgesprochenen Grundregel steht. Nun, damit $p1\uparrow$ überhaupt definiert ist, sei eine Anweisung new($p1$) vorausgegangen. Nach der Grundregel leben die Speicherplätze von $p1$ auf dem Heap aber nur auf dem level 2 und darunter, oder anders ausgedrückt, sind sie nicht mehr ansprechbar sobald in UP2 zurückgesprungen wird. Der Fehler liegt also darin, daß die Generierung neuer Speicherplätze mittels new auf dem falschen leven stattgefunden hat. Das Problem wäre gelöst, sobald man

new(p2); $p1 := p2$;

geschrieben und dann mit $p1$ normal weitergerechnet hätte. Weiter könnte es sein, daß die Listenelemente von $p2$ für die neue Berechnung noch benötigt werden. In diesem Fall könnte man jedoch im UP1 eine weitere Variable $p3$ gleichen Typs vereinbaren und schreiben

$p3 := p2$; new($p2$); $p1 := p2$;

Im folgenden würde dan mit $p3$ statt $p2$ weitergerechnet. Man beachte, daß mit $p3 := p2$ eine lokalere an eine globalere Variable zugewiesen wurde. Dies ist durchaus zulässig, da $p2$ ja mindestens so lange lebt wie $p3$. Das spezielle Problem (1) ist also gelöst. Es gibt nur noch weitere Fälle, die im folgenden behandelt werden sollen:

a) pointer auf pointer sind nicht zugelassen, da sie offenbar nicht sinnvoll erscheinen.

b) Bei Anwendung von new auf eine Recordkomponente, die ihrerseits wieder pointer ist, wird der level der Muttervariablen übernommen.

z. B. type longint = record element: integer;
nachfolger: ↑longint end;
var i: ↑longint; begin new(i↑.nachfolger);

Ein weiteres Problem stellt die Pointerfunktion dar. Eine Möglichkeit, dieses Problem zu lösen, wäre dem level in einer speziellen Prozedur new1 direkt anzugeben. Mit einer Funktionsprozedur lev könnte man den level einer Pointervariablen direkt angeben. (1) würde jetzt also so aussehen:

new1($p1$,lev($p2$));

Mit einer solchen Regelung wären überhaupt alle Probleme auf einen Schlag gelöst. Andererseits gäbe man dem Benutzer ein gefährliches Mittel in die Hand, den level selbst zu bestimmen und zu setzen. Die Gefahr von unkontrollierten Fehlern wäre sicher sehr groß. Eine zweite Möglichkeit wäre, immer eine extra Variable mitzuführen, auf die jeweils new angewandt wird und die den richtigen level hat. Soll also eine Funktionsprozedur $f(a,b)$ geschrieben werden, und soll nach der Ausführung $f(a,b)$ den level l haben, vereinbart man eine neue Variable d auf dem level l und schreibt statt dessen $f(d,a,b)$. Beim Aufbau der Liste für die Ausgabevariable $f(d,a,b)$ wird new dann immer auf d angewandt. Eine andere Möglichkeit ist, immer den level des ersten Funktionsparameters zu übernehmen. Dabei treten für $c := f(a,b)$ zwei Fälle auf:

a) c ist lokaler oder vom gleichen level wie a.
In diesem Fall schreibt man einfach $c := f(a,b)$

b) c ist globaler als a

Könnte man in diesem Fall einfach

$c := a$; $c := f(c,b)$;

schreiben, so wäre der Fall gelöst. c wird nach Ausführung von f sowieso umbesetzt, kann also vorher einen anderen Wert bekommen. Nach unserer Grundregel ist eine solche Zuweisung aber nicht zulässig. Wir können in diesem Fall jedoch einen Trick anwenden, und zwar wird eine eigentliche Prozedur equal eingeführt. Wir schreiben dann

equal(c,a); $c := f(c,b)$;

Man beachte jedoch, daß equal in Wirklichkeit keine Prozedur ist. Tatsächlich wird intern equal(c,a) wie $c := a$ interpretiert und übersetzt. Der einzige Unterschied zu $c := a$ ist der, daß die level nicht mehr abgeprüft werden, d. h. bei lev(c) < lev(a) kein Abbruch mit Fehlermeldung erfolgt. Da nach Ablauf der Prozedur c sowieso einen anderen Wert bekommt, kann auch equal(c,a) unmittelbar vor dem Aufruf der Prozedur stehen. Der klare Vorteil von equal ist, daß dem Benutzer unmittelbar bewußt wird, daß eine Wertzuweisung mit an sich falschen leveln stattgefunden hat.

Die einzige Mehrarbeit in diesem System für den Benutzer ist, daß er sich über die level der einzelnen Variablen im klaren sein muß. Das heißt aber im Grunde genommen nichts anderes, als daß er wissen muß, was er programmiert. Die einzige Fehlerquelle im letzten Vorschlag ist, daß equal(a,b) unsachgemäß angewandt wird. Doch allein durch den Zwang, equal schreiben zu müssen sind solche Fehler praktisch ausgeschlossen. Eine Vorprüfung der level kann übrigens bereits bei der Übersetzung stattfinden. Im Gegensatz zu „normalen" Variablen darf dann eine Wertzuweisung nur noch von global nach lokal oder gleich geschehen. Der Verfasser hat bereits in vielen Programmen mit diesem System Erfahrungen gesammelt, und es traten keinerlei Schwierigkeiten auf.

Man mag einwenden, daß es schwierig ist, Listen aneinanderzuhängen, die auf unterschiedlichem level generiert wurden. Die Praxis zeigt jedoch, daß solche Fälle so gut wie nicht vorkommen, wie auch die jüngste Arbeit von *Moenck* bestätigt.

Will man jegliche Reglementierung vermeiden und das Programmieren „narrensicher" machen, gibt es die Möglichkeit, bei einer Zuweisung

$c := a$: mit lev(c) < lev(a)

alle levels in a auf lev(c) zu setzen. Dazu ist folgendes zu sagen. Zunächst wird für ein solches Unterfangen natürlich ein rekursiver Algorithmus benötigt, der alle von a aus erreichbaren Zellen findet und ihren level entsprechend neu setzt. Dazu könnte der bei jedem GC ohnehin vorhandene Algorithmus direkt übernommen werden. Die Rechenzeit für diesen Algorithmus ist aber proportional zur Anzahl der von a aus erreichbaren Listenelemente und nicht zur Anzahl der Speicherplätze überhaupt. Im allgemeinen wird die Rechenzeit also sehr gering sein. Hinzu kommt noch, daß die Situation (2) tatsächlich recht selten auftritt, so daß der zusätzliche Zeitaufwand gering erscheint. Das Problem, Listen aneinanderzuhängen, die auf unterschiedlichem level generiert wurden, wäre bei dieser Methode übrigens auch gelöst. Gleichzeitig könnte in naheliegender Weise eine Prozedur dispose eingeführt werden. Wir werden jedoch gleich sehen, daß es bei der Verwaltung der Variablen Schwierigkeiten geben könnte, so daß der Verfasser diese Methode nicht uneingeschränkt empfehlen möchte. In der Tat stellt die aufgestellte Grundregel keine nennenswerte Einschränkung dar.

3. Hinweise zur Implementierung

Zunächst wollen wir, um die Arbeitsweise des Systems noch weiter zu verdeutlichen, eine einfache und übersichtliche Lösung angeben, die jedoch recht Speicherplatz- und Zeit-intensiv ist. Zum besseren Verständnis scheint es jedoch angebracht, zunächst eine solche einfache Version vorzustellen.

Wir gehen davon aus, daß ein stack von folgendem Typ vereinbart ist:

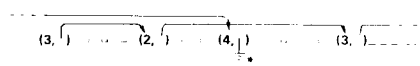
```
type stack: array[1..100] of file of integer;
```

Die Komponenten des stacks sind files, in denen die Adressen von Speicherplätzen stehen. Wird auf dem level l die pointervariable p vereinbart, wird p ja während des Laufs ein Speicherplatz zugewiesen und gleichzeitig l festgehalten. Dies geschieht auf dem normalen Variablen-Stack, der ja üblicherweise als Stapel angelegt wird. Unser stack muß jedoch dynamisch bleiben, da ja ständig vom level l und von allen lokaleren levels neuer Speicherplatz angefordert werden kann. Wird also new(p) aufgerufen, müssen auf dem Heap soviele Speicherplätze reserviert werden, wie p benötigt. Die Adressen dieser Speicherplätze werden nun auf stack[l] geschrieben. Wie man sieht, ist die Vorschrift unabhängig davon, auf welchem level new(p) aufgerufen wird. Vor Beginn der Ausführung des Programms stehen die Adressen aller verfügbaren Speicherplätze auf einem file free1. Außerdem existiert noch ein file free2 mit

```
type free1, free2: file of integer;
```

Weiter seien alle files stack[i], i = 1(1)100 und free2 zu Anfang leer. Die im Hauptprogramm reservierten Speicherplätze sind immer resident und brauchen daher nicht in die Speicherplatzrückführung einbezogen zu werden. Wird ein UP aufgerufen, so wird der level im 1 erhöht. Wird aus einem Unterprogramm des levels l in den Block des levels l' zurückgesprungen, werden alle Speicherplätze, deren Adressen auf stack[$l'+1$], stack[$l'+2$],..., stack[l] stehen, auf free2 geschrieben. Entsprechend werden, sobald irgendwo new(p) aufgerufen wird, die Speicherplätze von free1 genommen. Das geht solange gut, bis irgendwann free1 leer ist. In diesem Fall tauschen free1 und free2 die Plätze. Die Adressen der nicht benötigten Speicherplätze werden fortan auf free1 geschrieben und die verfügbaren werden durch Abruf von new von free2 genommen. Das System verfügt über keine freien Speicherplätze mehr, wenn beide files free1 und free2 leer sind.

Offenbar ist die Anzahl der geschachtelt aufrufbaren Unterprogramme von vornherein auf 100 bei dem beschriebenen System beschränkt. Denkbar ist, alles dynamisch anzulegen, doch dieses Problem ist nicht so gravierend. Das beschriebene System ist noch weitgehend in PASCAL formuliert und daher noch nicht allzu effizient und für den Compilerbauer nicht direkt brauchbar. Eine andere Möglichkeit, die Verwaltung des Heaps vorzunehmen, ist die folgende. Was vorher stack[l] hieß und ein file war, wird zur linearen Liste. Bei jedem Aufruf von new wird ein Kontingent von Speicherplätzen reserviert. Nun wird ein Platz mehr reserviert, in dem die Anzahl der Speicherplätze des nachfolgenden Kontingents und die Adresse des nächsten Kontingents gleichen levels gespeichert wird. Ein Ausschnitt einer typischen Situation im Speicher konnte etwa so aussehen:



Hier haben das erste, zweite und vierte Kontingent gleichen level. Die Liste eines anderen levels hört gerade bei dem dritten Kontingent auf. Wird nun new auf einem pointer dieses levels angewandt, wird der Zeiger (mit * gekennzeichnet) auf einen Speicherplatz gesetzt, auf den mindestens soviele verfügbare Speicherplätze folgen wie benötigt werden. Bei jedem Rücksprung aus einem Unterprogramm wird (nur) die lineare Liste des entsprechenden levels durchgegangen und jedes Element markiert, d.h., die markierten Speicherplätze sind wieder verfügbar. Möglicherweise könnte diese Liste auch einfach an eine Liste von freien Speicherplätzen angehängt wer-

den. Die Information, daß eine Speicherzelle verfügbar ist, kann in einem extra-Bit oder irgendwie sonst gespeichert werden. Das System bricht mit einer Fehlermeldung ab sobald kein Speicherplatz mehr auf dem Heap bei einem Aufruf von new zur Verfügung steht oder Stack und Heap zusammenstoßen. Das System mag dem einen oder anderen noch nicht zusagen, so daß noch eine weitere Variante angegeben werden soll. Der Stack der Variablen wird wie beim normalen Ablauf eines Programms angelegt, nur wird von einem level zum nächsten (d.h. von einem Unterprogramm zum nächsten) immer ein Freiraum gelassen. In diesen Freiraum wird, ähnlich wie vorhin, die Anfangsadresse und die Anzahl der reservierten Speicherplätze bei jedem Aufruf von new eingetragen. Sollte der reservierte Freiraum für die Speicherverwaltung eines bestimmten levels während des Ablaufs irgendwann nicht ausreichen, wird in den letzten Speicherplatz des Freiraums eine Vorwärtsadresse eingetragen, die auf einen neuen Freiraum zeigt, der am Ende des Stacks angelegt wird. Beim Rücksprung aus einem Unterprogramm werden wieder alle Speicherplätze, deren Adressen in dem entsprechenden Freiraum stehen, markiert, d.h. an den verfügbaren Speicherplatz zurückgegeben. Wie vorhin bricht das System mit einer Fehlermeldung ab, wenn bei einem Aufruf von new kein Speicherplatz mehr auf dem Heap zur Verfügung steht oder wenn Stack und Heap zusammenstoßen. Bei beiden Systemen ist zu beachten, daß niemals der ganze Speicher durchlaufen werden oder gar ein rekursiver Suchalgorithmus angewandt werden muß. Die benötigte Rechenzeit wird in beiden Fällen verschwindend klein sein. Bei beiden Systemen ist es möglich, eine dynamische Speicherverwaltung vorzusehen, etwa die nach einem Vorschlag von *Morris*.

Im vorliegenden hat der Verfasser Wert darauf gelegt, daß mehrere verschiedene Möglichkeiten sowohl in der Arbeitsweise als auch in der Realisierung des neuen Algorithmus aufgezeigt wurden. Für den einen oder anderen Anwender ist sicher die eine oder andere Variante interessanter oder besser in sein (evtl. zum Teil schon vorhandenes) Konzept einzubauen.

4. Zusammenfassung und abschließende Bemerkungen

Eine wirklich sinnvolle Anwendung von dynamischen Variablen, insbesondere bei großen Programmen, wird in PASCAL erst durch eine Speicherverwaltung mit Rückgabe nicht mehr benötigten Speicherplatzes möglich. Beim Anlegen von Listen blähen sich diese sehr schnell auf und nach kurzer Zeit ist aller Speicherplatz aufgebraucht. Die bekanntesten Vorschläge zur Lösung des Problems sind der GC und die reference count-Methode, die mit Varianten in gängigen Systemen auch angewandt werden wie SCRATCHPAD, MACSYMA usw. für die erste und SAC-1 für die zweite Methode. Leider ist die erste Methode für den Rechner, die zweite für den Benutzer recht zeitintensiv. Daher wurde nach einer neuen Möglichkeit gesucht, die im Optimum zwischen der Rückgabe aller nicht mehr benötigten Speicherplätze und möglichst geringem Aufwand für Anwender und Rechenanlage liegt. Es wurde ein Verfahren mit verschiedenen Varianten vorgeschlagen, das dem zitierten Optimum recht nahe zu kommen scheint. Tatsächlich wird genau soviel Speicherplatz wie beim GC zurückgegeben, und der Aufwand für den Benutzer ist minimal. Andererseits ist die Rechenzeit vernachlässigbar, so daß auch realtime Betrieb nicht mehr ausgeschlossen wird. Bei sehr kleinen Programmen ist die Rückgabe von Speicherplatz sowieso nicht notwendig. Bei allen anderen Programmen, auch denen, die am Bildschirm gerechnet werden, ist wie beim GC strukturiertes Programmieren Voraussetzung für ein effizientes Arbeiten des Systems. Weiterhin ist das vorgestellte Verfahren für beliebige Listen und Baumstrukturen geeignet. Im Gegensatz zu LISP treten solche Strukturen in PASCAL durch Records auch laufend auf. Diese lassen sich zwar immer auf binäre Bäume zurückführen, bei der Speicherplatzrückführung treten jedoch völlig neue Probleme auf. Beim GC werden auf der Suche nach nicht mehr benötigtem Speicherplatz ja zunächst alle im Gebrauch befindlichen Zeller markiert und sodann die unmarkierten freigegeben. Diese Markierung würde bei allgemeinen Baumstrukturen einen wohl nicht mehr gerechtfertigten Aufwand bedeuten. Beim vorgeschlagenen System entfällt ein solcher Suchalgorithmus gänzlich. Während beim GC bei jedem Aufruf alle Speicherplätze bearbeitet werden müssen und dadurch sehr viel Rechenzeit benötigt wird, wenn fast alle Speicherplätze in Benutzung sind, werden im vorgestellten System die nicht mehr benötigten Speicherplätze angeschaut und zurückgegeben, und zwar nur diese. Selbst bei der Variante, daß automatisch bei einer Zuweisung

$$c := a; \text{ mit } \mathrm{lev}(c) < \mathrm{lev}(a)$$

alle von a aus erreichbaren Speicherplätze den level von c bekommen, ist nur wenig zeitkonsumierend im Vergleich zum GC. Die Anforderungen an den Benutzer und an den Rechner sind beim vorgestellten System minimal.

Durch die verschiedenen in der Beschreibung angegebenen Varianten wurde versucht, auch unterschiedlichen Bedürfnissen gerecht zu werden.

Anerkennung: Für wertvolle Hinweise möchte ich Herrn Dr. *E. Kaucher* danken.

Literatur

Collins, George E.: A method for overlapping and erasure of lists, Comm. ACM 3, 12 (Dez. 1960), 655-657.

Jensen, K., und *Wirth, N.*: PASCAL User Manual und Report, 2. Auflage, Springer Verlag NY, Heidelberg, Berlin 1978.

Gregono, P.: Programming in PASCAL, Addison Wesley Series in Comp. Sc., 1978.

Lauer, M., und *Sämann, M.*: Reference Count Overflow, SIGSAM-Bulletin, 10, 2 (Mai 1976), 24–29.

McCarthy, J.: Recursive Functions of Sybolic Expressions and their Computation by machine, I, Comm. ACM 3, 3 (März 1960), 184–195.

Moenck, R.: Is a Linked List the Best Storage Structure for an Algebra System?, SIGSAM Bulletin 12, 3 (August 1978), 20–24.

Morris, F. L.: A Time- and Space-Efficient Garbage Compaction Algorithm, Comm. ACM 21, 8 (August 1978) 662–665.

Teer, F.: Formula Manipulation and PASCAL, Dr.-Dissertation, Mai 1978.