Eberhard Karls
Universität
Tübingen

Übungsblatt 3

Spezifikation eines Heap

Ausgabe: 14.5.2002
Abgabe: 21.5.2002, 11:45

Inhaltsverzeichnis

1 Vorwort

Diese Woche sollt Ihr ein etwas umfangreicheres Beispiel in VDM-SL spezifizieren, und zwar einen *Heap* und die in den Heap zeigenden *Zeiger*.

Ich habe versucht, dieses Unterfangen in einem separaten Handout *Spezifikation durch Funktionen und die Behandlung von Speicher* (TeachSWT@Tü, 2002b) etwas zu motivieren.

Ich werde im Folgenden eine (hoffentlich detaillierte) Schritt-für-Schritt-Anleitung geben. Wer entsprechend Sportsgeist besitzt, kann gleich bei der konkreten Aufgabenstellung weiterlesen.

2 Modellierung von Heap und Zeigern

Anleitung: Heaps und Zeiger (in der einen oder anderen Verkleidung) sind recht eng miteinander verknüpft. Wenn es darum geht, einen Heap zu spezifizieren, so stehen tatsächlich drei Aspekte dieses Themenkomplexes zur Behandlung an:

1. Dynamische Speicherorganisation: D. h. ein Verfahren, mit der aus einem großen Speicherbereich Abschnitte zur Benutzung reserviert und wieder freigegeben werden.
2. Die Semantik von Zeigern: Was bedeutet es, einen Zeiger lesend oder schreibend zu dereferenzieren? Was ist ein ungültiger Zeiger?
3. Die Semantik von Zeiger-Arithmetik: Was bedeutet es, einen Zeiger zu „verschieben“?

Diese Aspekte können zum größten Teil (und interessanterweise) unabhängig voneinander modelliert werden. Wir wollen uns hier auf die Semantik von Zeigern und ein stark vereinfachtes Modell der Belegung von Speicher beschränken. Von den gerade erwähnten Aspekten können wir den letzten für diese Aufgabe vollkommen beiseite lassen[1]. Beim ersten Teilaspekt können wir es uns für den Anfang recht einfach machen und einen Heap modellieren, bei dem eine Speicherbelegung niemals fehlschlägt. Dies ist zum Einen gar nicht so unrealistisch auf modernen Systemen[2], zum Anderen könnte man – ähnlich, wie ich das im Handout *Von der Spezifikation zur Implementation* demonstriert habe, den unendlichen Heap als Idealisierung des endlichen Heap begreifen, und in einem späteren Spezialisierungsschritt (wie am Beispiel der Zeile durchgeführt) sowohl die Einschränkung auf Endlichkeit, als auch die dadurch notwendig werdende Behandlung von Ausnahmesituationen (kein Speicher mehr verfügbar) einführen.

Weiterhin soll hier vorerst nur die Außenansicht des Heap modelliert werden, also kein Mechanismus oder Algorithmus, mit dem ein größerer Speicherbereich in Stücke unterteilt wird. Die Beziehung der einzelnen im Heap allozierten Speicherstücke zu dem unterliegenden zusammenhängenden Speicher der Maschine kann und soll vernachlässigt werden.

Speicher zu belegen, bedeutet vor allem, die Möglichkeit zu haben, Daten im Speicher abzulegen und diese Daten später wieder auslesen zu können. Der Heap fungiert als Container für Daten, die Zeiger als Zugriffsschlüssel.

[1] Meines Erachtens setzt man dessen Modellierung am zweckmäßigsten auf eine bereits existierende Beschreibung von Zeigern ohne Zeigerarithmetik auf.

[2] Memory-Over-Commit-Strategien, Paging und, im historischen Vergleich, riesige Swap-Bereiche machen es in der Praxis gar nicht einfach, mit Allokationen „vernünftiger“ Größe (für ANSI-C: 32 Kilobyte) den Heap wirklich zu erschöpfen.

Für unsere Zwecke hat ein Heap zwei Operationen, die Bezug zur Reservierung von Heapspeicher haben:

Malloc reserviert einen Speicherblock einer gegebenen Länge (in Byte) und gibt einen Zeiger zurück, der später verwendet werden kann, um diesen Speicher im Heap zu identifizieren bzw. zu referenzieren.

Free gibt einen zu einem Zeiger gehörigen Speicherblock wieder frei.

Ich empfehle, sich nun zuerst über eine als Modell für den Heap geeignete geeignete VDM-SL-Datenstruktur Gedanken zu machen, und darüber, was es bedeutet, dass ein Zeiger gültig ist. Von da ausgehend sollte es verhältnismäßig einfach sein, die Operationen *free* und *malloc* zu spezifizieren.

Die Modellierung des Zugriffs auf den im Heap reservierten Speicher über einen Zeiger bedarf noch einer kurzen Erläuterung. Die einfachsten (prototypischen) Fälle für den Heapzugriff sehen so aus:

1. `x=(*p)`: Vom Heap werden *n* Bytes (abhängig vom Typ von $*p$ gelesen und in die Variable *x* übertragen.
2. `*p=`*expr*: Aus dem Wert des Ausdrucks *expr* werden (abhängig vom Typ von $*p$) *n* Bytes produziert und auf den Heap geschrieben.

Wenn Ausdrücke dieser Form in einem Programm vorkommen, werden natürlich mitnichten spezielle Prozeduren aufgerufen, welche vom Heap lesen, bzw. auf ihn schreiben, d. h. unserer Beschreibung des schreibenden bzw. lesenden Zugriffs über einen Zeiger wird keine zu implementierende Prozedur gegenüberstehen. Stattdessen wird der Compiler an diesen Stellen direkt Maschineninstruktionen erzeugen, die die betreffenden Operationen durchführen. In diesem Sinne hat der Compilerbauer diese Operationen bereits implementiert, und wir müssen sie nur noch modellieren, um zu beschreiben, „was es bedeutet, ein Zeiger zu sein“. Was die Anzahl der referenzierten Bytes angibt – die ja vom Typ des Zeigers abhängt – so haben wir keine (einfache) Möglichkeit, davon zu wissen, oder gar in VDM-SL darüber zu reden. Der Compiler allerdings kann die entsprechende Information bereitstellen (tut dies in der Praxis auch), wir dagegen müssen diese Zahl als Eingangsparameter der Operation behandeln. Auch die Interpretation der gelesenen oder geschriebenen Bytes als Daten eines bestimmten Typs (z. B. als Fließkommazahl), ist nicht unsere Sache: Ihr sollt hier die gelesenen und geschriebenen Daten als Folgen von Bytes beschreiben.

Beschreibe nun die folgenden Zugriffe auf den Heap (als Operationen in VDM-SL):

Access: *N* Bytes werden von einem durch einen Zeiger *p* referenzierten Speicherblock im Heap gelesen.

Store: *N* Bytes werden in einem durch einen Zeiger *p* referenzierten Speicherblock im Heap abgelegt.

Die folgende Prozedur nimmt einen Zeiger auf einen (mindestens) zwei Byte langen Speicherbereich und tauscht die ersten beiden Byte aus.

Demonstriere an ihrem Beispiel den Gebrauch des Heapmodells, indem Du die Leistung dieser Prozedur in VDM-SL spezifizierst, und zwar so, dass sichtbar wird, dass der Parameter der Prozedur ein Zeiger ist:

```
void swap_bytes(unsigned char* p){
  unsigned char c;
```

```
  c    = p[1];
  p[1] = p[0];
  p[0] = c;
}
```

Nimm dabei an, dass der Speicherblock (wenn gültig) vom Heap alloziert wurde. Hier ist es nützlich, wenn einige Heapeigenschaften in geeignete Funktionen abstrahiert wurde.

Konkrete Aufgabenstellung: Hier noch einmal die Aufgabenstellung in Kurzform:

1. Beschreibe einen idealisierten Heap, der zwischen Zeigern und Speicherblöcken eine Beziehung herstellt.
2. Beschreibe die Operationen *malloc* und *free*.

   Malloc reserviert einen Speicherblock einer gegebenen Länge (in Byte) und gibt einen Zeiger zurück, der später verwendet werden kann, um diesen Speicher im Heap zu identifizieren bzw. zu referenzieren.

   Free gibt einen zu einem Zeiger gehörigen Speicherblock wieder frei.
3. Beschreibe den lesenden und schreibenden Zugriff über einen Zeiger: Die Operationen *access* und *store*.

   Access: *N* Bytes werden von einem durch einen Zeiger *p* referenzierten Speicherblock im Heap gelesen.

   Store: *N* Bytes werden in einem durch einen Zeiger *p* referenzierten Speicherblock im Heap abgelegt.
4. Spezifiziere die Prozedur *swap*, deren Implementation im letzten Abschnitt zitiert wurde.

Abzugeben sind: VDM-SL-Spezifikation der Zeiger- bzw. Heap-Operationen *malloc*, *free*, *access* und *store*, sowie der Prozedur *swap* in ASCII- oder mathematischer Notation. Eventuell erläuternde Bemerkungen.

Zum Schluss: Haben wir es uns mit der Idealisierung (unendlicher Heap) zu einfach gemacht? Ich glaube nicht. Stattdessen haben wir die Essenz dessen, was es bedeutet, „ein Heap zu sein“, herausgearbeitet. Diese ist wichtig, um Entscheidungen, die im Hinblick auf die Implementation getroffen werden, sauber von den grundlegenden Konzepten zu trennen. Mit Hilfe von Abstraktionsfunktionen (Jones (1992), S. 118; Bird und Wadler (1997), S. 222, 247; TeachSWT@Tü (2002c)) könnten wir nun in der nächsten Phase auf ein restriktiveres und realistischeres Modell übergehen, das die „realen“ Verhältnisse besser beschreibt. Eine solche Spezifikation könnte beginnen mit

```
types
  pointer = nat

  heap#s ::
    memory : seq of byte
    size   : map from pointer to nat

...
```

3 Null-Zeiger

Wie kann das Modell so erweitert werden, dass es auch einen Null-Zeiger (ein Zeiger, der immer ungültig ist) beschreibt?

Abzugeben ist: Eine kurze Erläuterung oder modifizierte Teile der Spezifikation.

4 Determinismus

Formale Spezifikationen sind natürlich vor allem deshalb nützlich, weil sich aus ihnen Fragen eindeutig klären lassen, an die bei Erstellung der Spezifikation (bzw. später der Implementation) noch niemand gedacht hat. Betrachte beispielsweise die folgende Prozedur:

```
unsigned char foo(){
  unsigned char* p;
  unsigned char  c;

  p=malloc(2); /* assume: never fails */
  c=*p;

  return c;
}
```

Diese Prozedur hat als Ergebnis ein Zeichen (C-Typ *unsigned char*). Ist dieses Ergebnis nach Deiner Spezifikation wohldefiniert, d. h. lässt sich sagen, was das Ergebnis sein wird? Begründe bzw. beweise Deine Behauptung aus Deiner Spezifikation! Wird das Ergebnis dieser Funktion immer das gleiche sein?

Abzugeben ist: Antwort auf die obige Frage, sowie eine kurze Begründung (Beweis).

Viel Spaß!

Literatur

[Bird und Wadler 1997] BIRD, Richard ; WADLER, Philip: *Introduction to functional programming*. Prentice Hall, 1997 (Prentice-Hall International series in computer science). – ISBN 0-13-484197-2, 0-13-484189-1

[Jones 1992] JONES, Cliff B.: *Systematic software development using VDM*. Zweite Auflage. Prentice Hall, 1992 (Prentice-Hall International series in computer science). – ISBN 0-13-880733-7

[TeachSWT@Tü 2002a] Wilhelm Schickart Institut (Veranst.): *Übungsmaterial, Fallbeispiele und Ergänzungen zur Softwaretechnikvorlesung 2002 in Tübingen*. Sand 13, D 72076 Tübingen, 2002. (Materialien zur Softwaretechnik). – URL `http://www-pu.informatik.uni-tuebingen.de/teachswt`, Release `teachswt-tue-2002-a.tar.gz`

[TeachSWT@Tü 2002b] LEYPOLD, M E.: Spezifikation durch Funktionen und die Behandlung von Speicher. In: (TeachSWT@Tü, 2002a). – `handouts/functionale-spec-und-speicher.vdm`

[TeachSWT@Tü 2002c] LEYPOLD, M E.: Von der Spezifikation zur Implementation. In: (TeachSWT@Tü, 2002a). – `handouts/example-specification-to-implementation.vdm`