Eberhard Karls
Universität
Tübingen

Lösungsskizze 3 : Spezifikation eines Heap in VDM-SL

Inhaltsverzeichnis

1 Vorbemerkung

Liest man die Aufgabenstellung (TeachSWT@Tü, 2002a) und die Spezifikation eines assoziativen Arrays (TeachSWT@Tü, 2002d), so fällt auf, dass *Heap* und *assoziatives Array* eine große Ähnlichkeit miteinander haben: Beide Strukturen fungieren als Container, in denen Daten[1] gespeichert werden können und mittels bestimmter Schlüssel[2] wieder extrahiert werden können. Die beiden Hauptunterschiede bestehen darin, dass

1. beim assoziativen Array die Schlüssel vom Klienten gewählt werden, während sie beim Heap durch den Provider (eben die Heapimplementation) festgelegt werden, und
2. die unter einem Schlüssel speicherbaren Daten beim Heap zusätzlichen Beschränkungen unterliegen, die sich nach den bei der Erteilung des Schlüssels (Allokation von Speicher) gegebenen Parametern richten (Größenbeschränkung der Bytefolgen).

Mit dieser Beobachtung sollte die formale Beschreibung eines Heap, dem der Speicher niemals ausgeht, eigentlich recht einfach sein. Wie beim assoziativen Array müssen wir in den Nachbedingungen besonders Sorge tragen, zu fixieren, welche Teile des Heapzustandes sich nicht verändern (nämlich die gerade nicht bearbeiteten Speicherblöcke). Die Vorbedingungen werden (wegen der Größenbeschränkungen) etwas reichhaltiger als beim assoziativen Array.

Ich werde im Folgenden eine Lösung geben, in der syntaktische Abstraktion (d. h. die Ersetzung von Teiltermen durch Funktionen mit sinnvollen Bezeichnern) konsequent und ausführlich angewandt wird, um zu illustrieren, wie lesbar formale Spezifikationen dann sein können. Andererseits geht das über das hinaus, was wir von den Übungsteilnehmern erwartet haben, weshalb ich teilweise auch alternative Formulierungen angebe, welche meist durch simples Einsetzen der Funktionsdefinitionen erhalten werden können.

2 Definition des Modells

types

1.0 $byte = \text{token};$

2.0 $pointer = \text{token};$

3.0 $blockdata = byte^*;$

4.0 $heap = pointer \xrightarrow{m} blockdata$

Über die Struktur von Bytes (*byte*) und die Struktur von Zeigern (*pointer*) müssen keine speziellen Annahmen gemacht werden. Ein Heap leistet im Wesentlichen eine Zuordnung zwischen den gültigen Zeigern, und den Daten in den jeweiligen Speicherblöcken[3].

[1] Beim Heap: Folgen von Bytes.

[2] Beim Heap: Zeiger.

[3] Um ganz genau zu sein: Diese Zuordnung ist eigentlich eine Leistung der Prozessorhardware, aber die hier behauptete Sicht ist natürlich insofern korrekt, als wir diese Leistung durch die Brille eines Heap ADT wahrnehmen wollen. Um zu sagen, wie ein Heap funktioniert, muss man nicht über die unterliegende Hardware reden, aber man muss natürlich sagen, wie Zeiger funktionieren und in diesem Kontext erscheint diese Funktion als eine Eigenschaft der Struktur (des ADT) *Heap*

Die Gültigkeit eines Zeigers ist dabei implizit darin enthalten, ob er in der Definitionsmenge dieser Zuordnung auftaucht,

functions

```
5.0  isValidPointer : heap × pointer → 𝔹
.1   isValidPointer (h, p) ≙
.2     p ∈ (dom h);
```

und die Länge eines Speicherblocks in der Länge der zugeordnete Bytefolge:

```
6.0  blocksize : heap × pointer → ℕ
.1   blocksize (h, p) ≙
.2     len h (p)
.3   pre isValidPointer (h, p) ;
```

3 Zugriff auf den Speicher

In der Regel will man beim Lesen und Beschreiben eines Speicherblocks auf dem Heap nicht den gesamten Inhalt auslesen oder modifizieren. Es ist durchaus erlaubt, einen größeren Speicherblock als benötigt zu allozieren, und nur einen Abschnitt dieses Speicherblocks zu verwenden. Aus diesem Grund ist in der Anleitung für *access* ein expliziter Längenparameter vorgesehen, der bei *store* implizit in der Länge der übergebenen Folge enthalten ist. Für die Gültigkeit der lesenden und der schreibenden Zugriffe gilt dieselbe Regel: Speicherzellen (Bytes), die beschrieben werden, müssen vorher alloziert worden sein. Da wir dieses Prädikat immer wieder benötigen, formulieren wir es vorausschauend bereits jetzt. Die boolesche Funktion *isValidAccess* gibt die Umstände an, unter denen es legitim ist, die ersten *n* Bytes eines Speicherblocks zu lesen oder zu schreiben.

```
7.0  isValidAccess : heap × pointer × ℕ → 𝔹
.1   isValidAccess (h, p, n) ≙
.2     isValidPointer (h, p) ∧
.3     len h (p) ≥ n;
```

Wie bereits in der Anleitung angedeutet, müssen wir in den Nachbedingungen immer explizit fixieren, welche Speicherblöcke des Heaps durch die betreffende Operation nicht verändert werden sollen. Die folgenden beiden Prädikate sind dazu nützlich: *unchanged-but* spezifiziert, dass der einzige Unterschied zwischen Heapzuständen in den Zuordnungen bezüglich des Zeigers *p* besteht (das betrifft sowohl Dateninhalt der Speicherblöcke, als auch Gültigkeit der Zeiger).

```
8.0  unchanged-but : heap × heap × pointer → 𝔹
.1   unchanged-but (h, h′, p) ≙
.2     {p} ⩤ h = {p} ⩤ h′;
```

Das Prädikat *data-unchanged-but* spezifiziert ganz analog, dass zwischen zwei Speicherblöcken (präziser: den Daten in den Speicherblöcken) ein Unterschied nur in den Bytes existiert, deren Indizes in der Menge *s* gegeben sind.

```
9.0  data-unchanged-but : blockdata × blockdata × ℕ-set → 𝔹
.1   data-unchanged-but(d, d′, s) ≙
.2     (len d = len d′ ∧
.3      ∀ i ∈ (inds d′ \ s) ·
.4           d′(i) = d(i));
```

Der lesende Zugriff auf einen Speicherblock (das Auslesen der ersten n Bytes) lässt sich nun ohne große Umstände so spezifizieren:

```
10.0 access(h : heap, p : pointer, n : ℕ) d′ : byte*
.1   pre isValidAccess(h, p, n)
.2   post d′ = (h(p))(1, ..., n) ;
```

Das Schreiben von n Bytes in einen Speicherblock (der durch einen Zeiger identifiziert wird), kann so spezifiziert werden:

```
11.0 store(h : heap, p : pointer, d : byte*) h′ : heap
.1   pre isValidAccess(h, p, len d)
.2   post let n = len d in
.3        isValidPointer(h′, p) ∧
.4        h′(p)(1, ..., n) = d ∧
.5        data-unchanged-but(h(p), h′(p), {1, ..., n}) ∧
.6        unchanged-but(h, h′, p) ;
```

Dabei sei (nochmals) darauf hingewiesen, dass ein Element vom Typ *heap* sowohl als Funktionsargument, als auch als Teil des Ergebnisses auftaucht: Dies bedeutet hier, dass diese Operation den *Zustand* des Heap verändert (und nicht, dass ein neuer Heap erzeugt wird).

Weiterhin macht ein Zeiger nur Sinn, wenn er im Kontext eines bestimmten Heapzustandes interpretiert wird. Der Zeiger verweist auf einen Teil des Heapzustandes – hier: $h'(p)$ – und ist, wenn nicht auf einen Heapzustand bezogen, vollkommen bedeutungslos.

Die gerade gegebene Spezifikation von *store* ist extrem deklarativ in dem Sinn, dass Sie präzise logische Aussagen über die erlaubten Anfangs- und Endzustände macht. Dies ist sehr nützlich, wenn man Beweise führen möchte, und wegen der angewandten syntaktischen Abstraktion – Namen für Prädikate, statt sie auszuschreiben – sogar leicht lesbar. Dabei ist aber nicht unmittelbar ersichtlich, wie sich der Endzustand aus dem Anfangszustand ergibt. Man kann das sichtbar machen, indem man alternativ spezifiziert:

```
12.0 store′(h : heap, p : pointer, d : byte*) h′ : heap
.1   pre isValidAccess(h, p, len d)
.2   post let b = h(p) in
.3        let n = len d in
.4        let b′ = d ↷ b(n + 1, ..., len b) in
.5        h′ = h † {p ↦ b′} ;
```

Die beiden Spezifikationen sind übrigens vollkommen äquivalent, in dem Sinne, dass sie exakt dieselbe datenverarbeitende Leistung beschreiben, d. h. dasselbe Verhältnis zwischen Anfangs- und Endzuständen.

Welche Formulierung man bevorzugt, wird in der Regel vom beabsichtigten Zweck abhängen: Die erste ist wie erwähnt leichter lesbar und nützlicher für Beweise, die die spezifizierte Operation involvieren. Die zweite ist näher an der Implementation. Man kann beide Spezifikationen als aufeinanderfolgende Schritte auf dem Weg zur Implementation sehen:

Zuerst könnte man die logischen Eigenschaften der Operation rein deklarativ festlegen und dann eine zwar schwerer lesbare aber explizitere Formulierung, aus der sich ein Algorithmus ableiten lässt, wählen[4].

4 Belegung und Freigabe von Speicher

13.0 $malloc\,(h : heap, size : \mathbb{N})\ p : pointer, h' : heap$
.1 post $isValidPointer\,(h', p) \wedge$
.2 $\quad \neg\, isValidPointer\,(h, p) \wedge$
.3 $\quad blocksize\,(h', p) = size \wedge$
.4 $\quad unchanged\text{-}but\,(h, h', p)$;

Hier wird spezifiziert, dass der zurückgegebene Zeiger neu ist (die beiden Klauseln mit *isValidPointer*), dass ein neuer Speicherblock mit der Länge *size* belegt wurde, und dass sich weder die Gültigkeit der anderen Zeiger, noch der Speicherinhalt der assoziierten Speicherblöcke ändert.

Ich gebe hier nochmal eine Definition mit expandierten Prädikatdefinitionen wieder, weil dies mehr dem entspricht, was von Teilnehmern der Übungen „spontan“ spezifiziert wurde:

14.0 $malloc'\,(h : heap, size : \mathbb{N})\ p : pointer, h' : heap$
.1 post $p \in (\mathsf{dom}\ h') \wedge$
.2 $\quad \neg\, p \in (\mathsf{dom}\ h) \wedge$
.3 $\quad \mathsf{len}\ h'\,(p) = size \wedge$
.4 $\quad \{p\} \lhd h = \{p\} \lhd h'$;

Bei der Freigabe von Speicher wird ein Zeiger als Zugriffsschlüssel ungültig:

15.0 $free\,(h : heap, p : pointer)\ h' : heap$
.1 pre $isValidPointer\,(h, p)$
.2 post $\neg\, isValidPointer\,(h', p) \wedge$
.3 $\quad unchanged\text{-}but\,(h, h', p)$;

Eine explizite Formulierung macht klar, dass lediglich der Zeiger p aus der Zuordnungsliste entfernt wird.

16.0 $free'\,(h : heap, p : pointer)\ h' : heap$
.1 pre $isValidPointer\,(h, p)$
.2 post $h' = \{p\} \lhd h$;

[4] Aber Vorsicht: Manchmal ist das möglich und dann zweifellos mit interessanten Einsichten verbunden. Meist ist es so trivial, dass man sich selten die Mühe machen wird, die Äquivalenz zwischen beiden Formulierungen auch zu beweisen (und dann hat man eine neue Quelle aus der im Laufe des Entwicklungprozesses Fehler sprudeln). Und manchmal ist es schlicht unmöglich, aus der Spezifikation etwas herzuleiten, was den (oder einen) Algorithmus erkennen lässt. Beispielsweise möchte ich doch bezweifeln, dass man aus der Formulierung des Prädikates *Sortiert* durch geschicktes Expandieren der Teilformeln den *Quicksort* Algorithmus praktisch halbautomatisch erhalten kann.

5 swap

17.0 $swap\,(h : heap, p : pointer)\; h' : heap$
.1 pre $isValidAccess\,(h, p, 2)$
.2 post $(h'\,(p))\,(1) = (h\,(p))\,(2) \wedge$
.3 $(h'\,(p))\,(2) = (h\,(p))\,(1) \wedge$
.4 $isValidPointer\,(h', p) \wedge$
.5 $unchanged\text{-}but\,(h, h', p) \wedge$
.6 $data\text{-}unchanged\text{-}but\,(h\,(p), h'\,(p), \{1,2\})$;

Eine alternative Formulierung leitet Vor- und Nachbedingung direkt aus den Operationen ab, aus denen die Implementation besteht:

18.0 $swap'\,(h : heap, p : pointer)\; h' : heap$
.1 pre pre-$access\,(h, p, 2) \wedge$
.2 let $l = access\,(h, p, 2)$ in
.3 let $l' = [l\,(2), l\,(1)]$ in
.4 pre-$store\,(h, p, l')$
.5 post let $l = access\,(h, p, 2)$ in
.6 let $l' = [l\,(2), l\,(1)]$ in
.7 post-$store\,(h, p, l', h')$;

Selbstverständlich lässt sich die kompliziertere Vorbedingung wieder auf die oben gegebene einfacher Form reduzieren. Am verständlichsten ist wahrscheinlich eine gemischte Formulierung:

19.0 $swap''\,(h : heap, p : pointer)\; h' : heap$
.1 pre $isValidAccess\,(h, p, 2)$
.2 post let $l = access\,(h, p, 2)$ in
.3 let $l' = [l\,(2), l\,(1)]$ in
.4 post-$store\,(h, p, l', h')$

6 Determiniertheit

Die Operation *malloc* ist nicht vollständig determiniert in dem Sinn, dass zwar über die Länge des allozierten Speicherblocks eine Aussage getroffen wird, jedoch nicht über den Inhalt des Speicherblocks. Dieser kann verschieden sein – entweder von Implementation zu Implementation oder gar von Aufruf zu Aufruf [5]. Diese Indeterminiertheit wird am Besten dadurch bewiesen, dass man zu einem gegebenen Input (h, n) die Existenz (mindestens) zwei verschiedener Resultate (p', h') und $(p'', h'',)$ beweist, zum Beispiel dadurch, dass man Beispiele konstruiert. Wollte man die Indeterminiertheit widerlegen, würde man voraussetzen, dass (p', h') und (p'', h'') die Nachbedingungen für die Eingabe (h, n) erfüllen und dann daraus beweisen müssen, dass $h' = h''$ und $p' = p''$.

Identifizieren wir für den Rest dieses Abschnittes den Typ *byte* mit *char* und den Typ *pointer* mit *nat*, nur um zu vermeiden, dass wir ständig *mk-token* schreiben müssen, wenn wir Werte dieser Mengen aufschreiben.

[5] Ob nicht fixiert ist, welche Implementation zu wählen ist, oder ob die Implementation selbst wiederum indeterministisch ist, wird in vielen Spezifikationsverfahren, die auf Vor- und Nachbedingungen (VDM-SL) oder auf Invarianten (Z) beruhen, nicht unterschieden.

Betrachten wir nun einen beliebigen Anfangszustand des Heap:
values

20.0 $h =$ let $m : \mathbb{N} \xrightarrow{m}$ char in
.1 m;

Wir konstruieren daraus zwei weitere Heapzustände:

21.0 $p =$ let $n : \mathbb{N}$ be st $n \notin (\text{dom } h)$ in
.1 n;

22.0 $h' = h \dagger \{p \mapsto \text{"}abc\text{"}\}$;

23.0 $h'' = h \dagger \{p \mapsto \text{"}xyz\text{"}\}$

Nun erfüllen, wie man sich leicht überzeugen kann, sowohl (p, h') als auch (p, h'') die Nachbedingung *post-malloc* für den Anfangszustand h und den Parameter n.

7 Nullpointer

functions

24.0 *nullpointer* () p : *pointer*
.1 post $p = p$

Die Existenz eines beliebigen, aber konstanten Wertes beschreiben wir am günstigsten durch eine nullstellige Funktion (TeachSWT@Tü, 2002c, siehe auch). Der Nullzeigers ist nie ein gültiger Zeiger, d. h. für *keinen* Heapzustand. Da Zeiger sowieso nur im Kontext eines Heapzustandes gültig oder ungültig sind, bedeutet das, dass ein Heap einfach keinen Zustand annehmen darf, in dem *nullpointer*() gültig ist. Man verankert dies am Besten als Invariante in der Definition des Heapzustandes:
types

25.0 $heap' = pointer \xrightarrow{m} blockdata$
.1 inv $h \triangleq nullpointer() \notin (\text{dom } h)$

Wie man sich wiederum leicht überzeugen kann, hat die Substitution der ursprünglichen Definition von *heap* mit dieser den Effekt, dass der Nullpointer nie die Vorbedingungen für *store*, *access* und *free* erfüllen kann und *malloc* nie *nullpointer*() zurückgeben kann, da mit dem modifizierten Heapmodell $heap'$ das Prädikat $isValidPointer(h, nullpointer())$ für alle h falsch ist.

Literatur

[TeachSWT@Tü 2002a] LEYPOLD, M E.: Übungsblatt 3: Modellierung eines Heap in VDM-SL. In: (TeachSWT@Tü, 2002b). – handouts/uebung-03.tex

[TeachSWT@Tü 2002b] 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 2002c] LEYPOLD, M E.: Lösungsskizze 2: Implementation eines FiFo. In: (TeachSWT@Tü, 2002b). – handouts/loesung-02.vdm

[TeachSWT@Tü 2002d] LEYPOLD, M E.: Spezifikation eines assoziativen Arrays in VDM-SL. In: (TeachSWT@Tü, 2002b). – Quelle handouts/example-vdm-spec.vdm