Eberhard Karls
Universität
Tübingen

Lösungsskizze 2 : Implementation eines FIFO

Inhaltsverzeichnis

1 Schnittstellen

1.1 *jobs.hh*

```
15  #ifndef HH_INCLUDED_jobs
16  #define HH_INCLUDED_jobs

18  namespace jobs {                /* DEFINE jobs       */

20    typedef struct DESC (* desc);

22    void
23    init   (desc* d, const char* title, const char* file);

25    const char*
26    title_of (desc* d);

28    const char*
29    file_of (desc* d);

31    void
32    copy (desc* s, desc* d);

34    void
35    move (desc* s, desc* d);

37    void
38    deinit (desc* d);
39  }

41  #endif /* HH_INCLUDED_jobs defined */
```

1.2 *queues.hh*

```
15  #ifndef HH_INCLUDED_queues
16  #define HH_INCLUDED_queues

18  #include <sys/types.h>

20  #include "jobs.hh"               /* IMPORT jobs       */

22  namespace queues {               /* DEFINE queues     */

24    const size_t len_max = 5;

26    typedef struct {       /* treat as ABSTRACT !         */
27      size_t   len;             /*INVARIANT: len<=len_max.*/
28      jobs::desc data[len_max]; /*data[0] ... data[len-1] */
29                                  are the content         */
30    } queue;
```

```
33    void
34    init (queue* q);

36    int
37    is_empty(queue* q);

39    int
40    insert (queue* q, /* in/move  */  jobs::desc* jp);

42    void
43    next   (queue* q, /* out/move */  jobs::desc* jp);

45  }

47  #endif /* HH_INCLUDED_queues defined */
```

2 Implementationen

2.1 *jobs.cc*

```
14  #include "jobs.hh"      /* IMPLEMENT jobs            */
16  #include <string.h>     /* USING strdup() + friends  */
17  #include <stdlib.h>     /* USING malloc() + friends  */
19  namespace jobs {

21    struct DESC {   /* treat as ABSTRACT !           */
22      char* title;  /* this is a dummy module anyway */
23      char* file;
24    };

26    void
27    init   (desc* d, const char* title, const char* file)
28    {
29      (*d)=(struct DESC*)malloc(sizeof(*d));
30      (*d)->title=strdup(title);
31      (*d)->file=strdup(file);
32    };

34    const char*
35    title_of (desc* d)
36    {
37      return (*d)->title;
38    };

40    const char*
41    filename_of (desc* d)
42    {
43      return (*d)->file;
44    };

46    void
47    copy (desc* s, desc* d)
48    {
49      init(d, (*s)->title, (*s)->file);
50    };

52    void
53    move (desc* s, desc* d){
54      (*d)=(*s); (*s)=0;
55    };

57    void
58    deinit (desc* d){
59      free((*d)->file);
60      free((*d)->title);
61      free(*d);
62    };
63  }
```

2.2 *queues.cc*

```
13  #include "queues.hh" /* IMPLEMENT queue */
14  namespace queues {

16    void
17    init (queue* q)
18    {
19      q->len=0;
20    };

22    int
23    is_empty(queue* q)
24    {
25      return q->len==0;
26    };

28    int
29    insert(queue* q, jobs::desc* jp)
30    {
31      if (q->len>=len_max) return 0;
32      jobs::move(jp,q->data+q->len);
33      q->len++;
34      return -1;                /* return TRUE on success */
35    };

37    void
38    next(queue* q, jobs::desc* jp)
39    {
40      size_t        i;

42      jobs::move(q->data+0,jp);

44      for (i=1; i<(q->len); i++){
45        jobs::move(q->data+i,q->data+(i-1));
46      }
47      q->len--;
48    };

50  }
```

3 Testtreiber *testschedule.cc*

```
13  #include "jobs.hh"
14  #include "queues.hh"

16  #include <stdio.h>

20  const char* titles[] = {
21    "Diplomarbeit Illec",
22    "Testdruck 4Farben",
23    "Buch der 1000erlei Kräuter",
24    "Power C++ in 2 Tagen",
25    "Schattenparker",
26    "Schattenparker 2",
27    "Die Rückkehr der Schattenparker"
28  };

31  const char* files[] = {
32    "cutomers/illec.ps",
33    "testing/4cols.ps",
34    "customers/1000-kraeuter.ps",
35    "customer/powerc++.ps",
36    "archive/schattenp1.ps",
37    "archive/schattenp2.ps",
38    "archive/schattenp3.ps"
39  };

41  #define NR_ELEMNS(x) (sizeof(x)/sizeof(*x))

43  const int no_of_jobs = NR_ELEMNS(titles);

46  int
47  main(int argc, char* argv[])
48  {
49    int i;
50    int success;

52    jobs::desc       jd;
53    queues::queue     q;

55    queues::init(&q);

57    printf("\nstarting test: %s.\n","queuing.");

59    for (i=0; i<no_of_jobs; i++){

61      printf("queuing: %s.\n",titles[i]);

63      jobs::init(&jd,titles[i],files[i]);

65      success=queues::insert(&q,&jd);

67      if (!success) {
```

```
68        printf("could not be queued: %s.\n",
                              jobs::title_of(&jd));
69        jobs::deinit(&jd);
70      }
71    };

73    printf("\ncontinuing test: %s.\n","dequeuing.");

75    while (! queues::is_empty(&q)){
76      queues::next(&q,&jd);
77      printf("dequeued: %s.\n",jobs::title_of(&jd));
78      jobs::deinit(&jd);
79    };

81    printf("\ntest completed.\n");
82    return 0;
83  };
```

4 Abstraktion und Fehlerbehandlung

4.1 Abstraktionsfunktion

Sei q ein Lvalue (Ausdruck an den eine Zuweisung erfolgen kann) des Typs *queue* an einer Stelle (Quelltextzeile) eines C-Programms, dann soll, beim Ausführen dieser Stelle, der Speicherzustand der Maschine als abstrakter Wert f vom Typ *fifo* mit Hilfe der Abstraktionsfunktion *abstr-fifo*$\langle\,\rangle$ folgendermassen[1] interpretiert werden:

$$
\begin{aligned}
&\textit{abstr-fifo}\langle q\rangle = f\\
&\quad\Leftrightarrow\quad \textit{len}\, f \le \lceil q.\textit{len}\rfloor\\
&\quad\wedge\quad \forall i \in 0\ldots(\lceil q.\textit{len}\rfloor\text{-}1)\ \bullet\\
&\qquad\qquad \textit{abstr-job}\langle q.\textit{data}(\!|\, i\,|\!)\rangle = f(i)
\end{aligned}
$$

Die Darstellungsinvariante *reprinv-fifo*$\langle\,\rangle$ ist eine Integritätsbedingung an den Speicherzustand des C-Programms, damit q noch als Wert vom Typ *fifo* interpretiert werden kann. Insbesondere müssen alle Lvalues (Speicherreferenzen) in den C-Ausdrücken der Definition von *abstr-fifo*$\langle\,\rangle$ gültige Speicherreferenzen sein, wenn *reprinv-fifo*$\langle\,\rangle$ für den Laufzeitzustand des C-Programms gilt.

Im vorliegenden Fall benötigen wir die folgende Darstellungsinvariante:

$$
\begin{aligned}
&\textit{reprinv-fifo}\langle q\rangle =\\
&\qquad (0 \le \lceil q.\textit{len}\rfloor \le \lceil \textit{max\_len}\rfloor)\\
&\quad\wedge\quad \forall i \in 0\ldots(\lceil q.\textit{len}\rfloor\text{-}1)\ \bullet\\
&\qquad\qquad \textit{reprinv-job}\langle q.\textit{data}(\!|\, i\,|\!)\rangle
\end{aligned}
$$

Dabei darf $\lceil q.\textit{len}\rfloor$ nie größer als $\lceil \textit{max\_len}\rfloor$ werden, damit $\lceil q.\textit{data}(\!|\, i\,|\!)\rfloor$ in der Definition von *abstr-fifo*$\langle\,\rangle$ eine gültige Speicherreferenz ist[2]. Gleichzeitig müssen alle im Array gespeicherten Werte eine Interpretation als Jobdeskriptoren besitzen.

Für all das müssen dann schon eine Interpretation *abstr-job*$\langle\,\rangle$ und eine Darstellungsinvariante *reprinv-job*$\langle\,\rangle$ für Variablen des Typs *job* vorliegen.

4.2 Einschränkungen der Implementation

Wie man unschwer erkennen kann, ist *abstr-fifo*$\langle\,\rangle$ nicht surjektiv: Die Funktion ist nur auf Darstellungen q (C-Variablen vom Typ *queue*) definiert, in denen das Feld *q.len* kleiner als $\lceil \textit{max\_len}\rfloor$ ist (diese Bedingung ist die Darstellungsinvariante und wurde durch die Notwendigkeit erzwungen nur gültigen Speicher zu referenzieren).

Der ursprüngliche abstrakte Typ *fifo* enthält jedoch Sequenzen beliebiger Länge, diejenigen, deren Länge größer $\lceil \textit{max\_len}\rfloor$ ist, können also nicht im Bildbereich von *abstr-fifo*$\langle\,\rangle$ – *rng abstr-fifo*$\langle\,\rangle$ – liegen.

Wenn wir nun Operationen spezifiziert haben, die bei bestimmten Eingaben Werte außerhalb *rng abstr-fifo*$\langle\,\rangle$ produzieren sollen, so kann eine Implementation auf der Basis der gewählten Darstellung den von der Spezifikation vorgegebenen Kontrakt gar nicht erfüllen,

[1] Zur Bedeutung der verwendeten Notation und eine oberflächliche Herleitung siehe Anhang A (Seite 14).

[2] Meines Erachtens lassen sich diese Überlegungen recht gut formalisieren, das möchte ich allerdings nicht in dieser Lösungsskizze tun, schon allein um der Verständlichkeit willen.

weil sie *keine Möglichkeit hat* die Resultatwerte in einer Variable vom Typ *queue darzustellen*.

Da unser nächster Schritt sein wird, den abstrakten Typ zu beschränken, um alle Werte des abstrakten (und wie sich herausgestellt hat: zu idealen) Typs darstellbar zu machen, sollten wir nun untersuchen, ob die Operationen überhaupt die gesamte Wertemenge des Datentyps *fifo* erreichen, wenn man mit einem einer leeren Warteschlange beginnt und dann beliebige Operationen ausführt. In unserem Fall ist dies tatsächlich so: Durch sukzessives Einstellen geeigneter Jobdeskriptoren in die Warteschlange lässt sich jeder Zustand des *fifo* erzeugen[3].

Wäre dem nicht so gewesen, wäre unser Modell *fifo* für den Zustand eines Fifo unangemessen groß gewesen. Wir hätten uns fragen müssen, ob wir nicht wichtige Operationen vergessen haben, oder wir hätten mittels einer Invariante einschränken müssen, um Werte, die nie auftreten können, explizit aus dem Modell zu nehmen.

Definieren wir nun eine Untermenge *fifo* des ursprünglichen Typs *fifo*, welche wir tatsächlich mit C-Variablen des Typs *queue* darstellen können.

functions

1.0 $max\text{-}len() \; m : \mathbb{N}$

.1 post $m \geq 2$

Wir postulieren die Existenz eines bestimmten Wertes (der gewissermaßen Parameter unserer Spezifikation sein wird), über den wir noch keine genauen Aussagen machen wollen.

Für eine rein axiomatische Charakterisierung eines mathematischen Objekts muss man in VDM-SL etwas tricksen, da dort Axiome nicht einfach freistehend aufgeschrieben werden können (Nach dem Motto: Sei *max-len* eine Zahl größer 0). Aussagen lassen sich nur in die Vor- und Nachbedingungen von Funktionen schreiben. Wir nutzen das hier aus, indem wir bemerken, dass konstante Funktionen (mit 0 Argumenten) vollkommen äquivalent zu Konstanten sind, und formulieren unsere Axiome über die Konstante, welche die Funktion repräsentiert in der Nachbedingung als Aussage über das Resultat der Funktion.

Die Bedingung $m \geq 2$ wurde hier deswegen gewählt, weil ich keine über das Nötige hinausgehenden Einschränkungen machen wollte. Fordert der Kunde später eine Mindestlänge der Warteschlange von, sagen wir, 100, so ist $m \geq 100$ die stärkere Bedingung, da $m \geq 2$ daraus gefolgert werden kann, und somit alle auf der Basis von $m \geq 2$ getroffenen Aussagen gültig bleiben. Die Wahl der 2 als Untergrenze ist etwas willkürlich, rührt aber daher, dass ich davon ausgehe, dass sich die (hier nur angedeuteten) Beweise auf der Basis eine Länge von 1 nicht führen lassen[4].

Nun schränken wir den Wertebereich des ursprünglichen *fifo* mittels einer Invarianten ein.

types

2.0 $job = \text{token};$

3.0 $fifo' = job^*$

.1 inv $f \triangleq \text{len } f \leq max\text{-}len()$

Für die Implementation müssen wir nun fordern $max\text{-}len() = \lceil max\_len \rfloor$, dann ist die Ab-

[3] Das lässt sich leicht durch Induktion beweisen.

[4] Der Induktionsschritt wird sich nicht immer durchführen lassen, zudem sind viele Eigenschaften eines FiFo – wie die, dass die Reihenfolge des Herausnehmens genau die Reihenfolge des Einstellens ist – mit $n = 1$ gar nicht sichtbar.

straktionsfunktion surjektiv. [5]

Wie bereits erläutert ist es nun möglich, dass die Nachbedingungen nicht mehr für alle Operationen unter allen Umständen erfüllbar sind.

Dies ist in der Tat der Fall für die Operation *enqueue*. Ich gebe hier nochmal die alte Spezifikation von *enqueue*:

functions

4.0 $enqueue''\,(f : fifo', j : job)\; f' : fifo'$

.1 post $f' = f \frown [j]$

Wie man aus der Nachbedingung leicht zeigt, ist immer $len\, f' = 1 + len\, f$, das heisst also, die Nachbedingung ist nicht erfüllbar für alle f mit $len\, f = max\text{-}len()$, da es kein f' in *fifo'* mit $len\, f = 1 + max\text{-}len$ gibt.

Nichtmathematisch ausgedrückt: Die Warteschlange ist voll. Für diese Fälle müssen wir uns eine Sonderbehandlung einfallen lassen.

Die Nachbedingungen aller anderen Funktionen sind übrigens auch mit dem eingeschränkten Typ *fifo'* erfüllbar[6], diese müssen also nicht korrigiert werden.

Wir wollen nun eine möglichst minimale Abwandlung von *enqueue* erstellen, die auch wirklich erfüllbar ist. Möglichst minimal heisst hier, dass wir überall, wo es möglich ist, die ursprünglich mit *enqueue* ins Auge gefasste Leistung erbringen – da wir ja wirklich an dieser Leistung interessiert sind – aber für die Stellen (eine Untermenge A von *fifo'*) wo das nicht möglich ist, eine Ausnahmebehandlung vorsehen.

Wir haben hier prinzipiell zwei Möglichkeiten: Wir können die Vorbedingung geeignet verschärfen, so dass wir die Eingaben, für die die Nachbedingung nicht erbracht werden kann, ausschließen. Die Verantwortung, die Ausnahmen A zu erkennen, wird damit den Client-Prozeduren aufgebürdet.

Alternativ können wir die *enqueue* um einen Mechanismus erweitern, mit dem die Operation selbst signalisieren kann, wenn die Leistung nicht erbracht werden kann. Dazu muss die Ergebnismenge von *enqueue* um entsprechende Fehlerwerte ergänzt werden[7].

Ich möchte nun die letztere Möglichkeit umsetzen. Ein Flag vom Typ *success-flag* wird von jeder Operation mit zurückgegeben werden und zeigt an, ob die (ursprünglich intendierte) Operation durchgeführt werden konnte oder nicht.

types

5.0 $success\text{-}flag = \mathbb{B}$

Mit dem bereits formulierten Desideratum, überall, wo möglich die ursprünglich spezifizierte Leistung zu erbringen, ergibt sich die folgende modifizierte Operation, wobei wir auf die Definition von *enqueue* rekurrieren.

functions

6.0 $enqueue'\,(f : fifo', j : job)\; f' : fifo', OK : success\text{-}flag$

.1 post if $\exists f'' : fifo' \cdot$ post-$enqueue''\,(f, j, f'')$

.2 then post-$enqueue''\,(f, j, f') \wedge OK$

.3 else $f = f' \wedge \neg OK$;

[5] Auch dies lässt sich bei freier Handhabung der Notation mittels Induktion zeigen.

[6] Um das zu zeigen muss man einen gewöhnlichen mathematischen Existenzbeweis für Werte, die die Nachbedingung erfüllen, unter Voraussetzung der jeweiligen Vorbedingungen, führen.

[7] Es gibt hierbei wiederum verschiedene Möglichkeiten: Einmal die Erweiterung um einige ausgewählte Fehlerkodes, das andere Mal die Bildung des kartesischen Produkts der ursprünglichen Wertemenge mit einer Menge von Fehlerkodes. Ich möchte das hier nicht vertiefen. Beide Verfahren haben ihren Platz und ihren Anwendungsbereich.

Wie man sieht ist dies nichts weiter als die mathematische Notation der Forderung, wo möglich, die Leistung von *enqueue* zu erbringen, aber wo nicht, einen Fehler zu signalisieren ($OK = false$) und dabei den Zustand der Warteschlange unverändert zu lassen.

Geschicktes Umformen und Expandieren verschiedener Definitionen ergibt eine eigenständige Formulierung der modifizierten Operation:

$$
\begin{array}{ll}
7.0 & \mathit{enqueue}\,(f : \mathit{fifo}', j : \mathit{job})\ f' : \mathit{fifo}', \mathit{OK} : \mathit{success\text{-}flag} \\
.1 & \text{post if len } f < \mathit{max\text{-}len}\,() \\
.2 & \qquad \text{then } f' = f \overset{\frown}{} [j] \wedge \mathit{OK} \\
.3 & \qquad \text{else } f = f' \wedge \neg \mathit{OK}\ ;
\end{array}
$$

Etwas syntaktische Abstraktion zum Abschluß ist nicht verkehrt, da sie die Definition lesbarer macht:

$$
\begin{array}{ll}
8.0 & \mathit{isFull} : \mathit{fifo}' \to \mathbb{B} \\
.1 & \mathit{isFull}\,(f) \ \triangleq \\
.2 & \quad \text{len } f = \mathit{max\text{-}len}\,();
\end{array}
$$

$$
\begin{array}{ll}
9.0 & \mathit{enqueue}'''\,(f : \mathit{fifo}', j : \mathit{job})\ f' : \mathit{fifo}', \mathit{OK} : \mathit{success\text{-}flag} \\
.1 & \text{post if } \neg\, \mathit{isFull}\,(f) \\
.2 & \qquad \text{then } f' = f \overset{\frown}{} [j] \wedge \mathit{OK} \\
.3 & \qquad \text{else } f = f' \wedge \neg \mathit{OK}
\end{array}
$$

Die in den vorherigen Kapiteln gegebene Implementation entspricht bereits dieser Spezifikation.

5 Die Gefahren des pauschalen unqualifizierten Imports

Am wahrscheinlichsten ist etwas in der folgenden Art geschehen: Die Entwickler der Firma B haben im Package *wonderapp* eine Klasse mit einem einem recht unspezifischen Namen, wie z. B. *Slider* definiert.

Diese befindet sich – da *public* – automatisch im Namensraum, der beim Übersetzen von *B.wonderapp.q* gilt, und zwar unter dem verkürzten Namen *Slider*.

Ab Version 0.98 enthält auch das Package *A.superbib* eine Klasse *Slider*. Durch den pauschalen[8], unqualifizierten[9] Import "`importA.superbib.*;`" wird auch diese Klasse unter ihrem verkürzten Namen *Slider* im Namensraum bekannt gemacht, in dem *B.wonderapp.q* übersetzt wird. Damit sollte es zwei *verschiedene* Konstrukte (hier: Klassen) geben, die unter *demselben* Namen bekannt sind.

Wenn wir das allgemeinere Bild betrachten und uns nicht nur auf Java beschränken, kommt es in solchen Situationen auf die Sprachdefinition an, wie der Compiler auf dieses Ansinnen reagiert. Die Definition des C++-Namespace-Mechanismus sieht vor, dass der – im Verlauf der Übersetzung von *B.wonderapp.q* – später vorgenommene Import den zuerst existierende stillschweigend überdeckt. Ein konformer Java-Compiler andererseits muss hier ein Problem melden.

Betrachtet man den ganzen Vorfall unter dem Gesichtspunkt des *Design by Contract* so ist eigentlich nicht auszumachen, was eine Direktive wie "`importA.superbib.*;`" ausdrücken soll.

[8] Darunter verstehe ich, dass alle exportierten Namen unterschiedslos importiert werden.

[9] Darunter verstehe ich, dass die Namen ohne den Paketpfad verfügbar gemacht werden, also nicht durch den Paketpfad *qualifiziert* sind.

Bei den Import-Deklaration in Modula-2 ist verhältnismäßig klar, welcher Kontrakt sich hier wie konstituiert: Ein Identifier wird explizit exportiert: Hier liegt ein *Angebot* vor, das implizit (begleitende Dokumentation) von einem entsprechenden Leistungsversprechen begleitet ist. Wird der Identifier ebenso explizit (mit Namen) importiert, wird das Angebot *angenommen*, und man verlässt sich darauf, dass die Leistung erbracht wird. Dies gilt ebenso, wenn das komplette Modulinterface importiert wird. In beiden Fällen wird gleichzeitig ein Teil des Namensraums im Clientmodul an "fremde" Konstrukte abgetreten, und in beiden Fällen lässt sich deutlich sagen und vor allem abgrenzen, welcher Teil des Namensraums das ist (beim Import des vollständigen Interface bleiben die Namen ja qualifiziert). Vor allem ist klar welche Namen nicht verändert, bzw. definiert werden.

Anders sieht die Sache beim *-Import in Java aus: Der Provider (Firma A) hat kein formales (d. h. sich im Programmtext von *A.superbib* niederschlagendes) Angebot gemacht, welchen Teil des Namensraums das Paket im Client benötigt, noch kann der Autor des Clientmodul kontrollieren (solange er den *-Import verwendet), welche Namen definiert werden. Die beiden Kontrakte, von denen man sich vorstellen könnte, dass sie durch den *-Import in Anspruch genommen werden, wären

- Ich bin bereit, alle Namen aus dem Paket *A.suberbib* zu akzeptieren, entweder auch alle zukünftigen – dann reserviere ich keinen Teil des Namensraums im Clientmodul für mich selbst – oder ich werde nie auf eine neue Version von *A.suberbib* upgraden.
- A hat versprochen nie neue öffentliche Klassen in *A.suberbib* einzufügen.

Beide sind im Grunde vollkommen unsinnig, so dass sich erübrigt darauf grob einzugehen.

Es bleibt das Fazit, dass der pauschale unqualifizierte Import – übrigens in jeder Sprache, die ihn besitzt – eine im Sinne der Softwarewartung riskante Situation schafft, und deshalb schlicht unterbleiben sollte: Konstrukte, die unter kurzen (unqualifizierten) Namen verfügbar werden sollen, müssen einzeln importiert werden.

A Einführung einer Notation zur Verbindung eines VDM-SL-Modells mit einer C-Implementation

Dieser Anhang sollte eigentlich ein eigenes Handout sein, oder noch besser, mit dem Handout Von der Spezifikation zur Implementation *(TeachSWT@Tü, 2002d) zusammengeführt werden.*

A.1 Vorrede und Motivation

Spezifikationen, in unserem Fall VDM-SL-Modelle, beschreiben die gewünschten Eigenschaften einer Implementation – in unserem Fall die datenverarbeitenden Leistungen des Programms. In den modellbasierten Spezifikationsverfahren geschieht dies weitgehend durch Analogie: Die Implementation soll sich „so wie“ das Modell verhalten. Dieses „verhält sich so wie“ bedarf aber einer genaueren Präzisierung: Es muss eine Verbindung zwischen Modell und Implementation hergestellt werden.

Hierfür benötigen wir zum einen ein Verfahren (d. h. eine Herangehensweise, die für eine breite Klasse von Modellen und Implementationen anwendbar ist), zum anderen eine Notation, die es ermöglicht, den Zusammenhang zwischen Modell und Implementation zu verschriftlichen und in formaler (symbolischer) Weise über Eigenschaften der Implementation zu argumentieren. Bei Sprachen, für die bereits eine formale Semantik erstellt wurde, kann der semantischen Bereich (d. h. das mathematische Modell, in das die Leistung von Programmen dieser Sprache abgebildet wird), als Ansatzpunkt dienen, diesen Zusammenhang herzustellen.

C jedoch, die Sprache, um die es mir hier hauptsächlich geht, besitzt keine formal festgelegte Semantik. Eine solche zu entwickeln, wäre auch – zieht man dabei die historisch gewachsenen Eigenheiten der Sprache in Betracht – keine leichte Aufgabe. Zudem halte ich ein auf eine formale Semantik aufsetzendes Verfahren für zu aufwendig für den Alltagsgebrauch. Es geht mir ausdrücklich nicht darum, vollständige Regeln zum Schlussfolgern über Programme zur Verfügung zu stellen, sondern darum, so schnell als möglich zu einer Notation zu kommen, mit der Fakten über Programme in unmissverständlicher und dem Problem angemessener Weise beschrieben werden können.

Interessanterweise lässt sich das von der Frage nach der Semantik einzelner Anweisungen verhältnismäßig weit abtrennen, so daß man auch frei wäre, später mit verschiedenen Semantiken von, sagen wir, Untermengen von C innerhalb des im Folgenden skizzierten Frameworks zu experimentieren.

Ich möchte damit beginnen, herauszustellen, was als eine Verbindung zwischen Modell und Spezifikation *nicht* genügt: Dies ist die *syntaktische* Entsprechung von Datendefinitionen bzw. von Prozedurköpfen und Funktionssignaturen, also z. B. gleiche oder ähnliche Benennungen von Parametern und Datenkomponenten, und gleiche Parameterzahl von Konstrukten der Spezifikation und der Implementation. Dies genügt nicht aus zwei Gründen: Zum einen existieren, wie der „syntaktische Ringelreihen“ im Handout *Spezifikation durch Funktionen und die Behandlung von Speicher* (TeachSWT@Tü, 2002c) hoffentlich überzeugend demonstriert, zu viele Freiheiten, ein und dieselbe datenverarbeitende Leistung mit einer Schnittstelle (einem Prozedurkopf, bzw. Übergabekonventionen für Parameter und Resultate) zu versehen. Zum anderen weist der Implementationbereich (C) Charakteristika auf, die der Modellbereich (VDM-SL) schlicht nicht besitzt – etwa Zeiger und ein System von Variablen. Diese könnten zwar (in mühseliger Weise) im Modellbereich beschrieben (modelliert) werden – damit ist man jedoch bereits auf dem halben Weg zur Erstellung einer Semantik, ein Unterfangen, das man nach meinem Dafürhalten am besten solange und

soweit als möglich vermeidet[10], wenn man den Spezifikationsprozess „lightweight“ halten möchte.[11].

Syntaktische Entsprechung kommt also nicht in Frage. Stattdessen müssen wir eine Äquivalenz – in einem gewissen Sinn einen Homomorphismus – finden, zwischen den in der Spezifikation aufgeschriebenen *Datenzusammenhängen* (also etwa der Relation zwischen Ein- und Ausgabe einer Operation), und den Abläufen in der Implementation. Mit einem Wort: Wir müssen die in der Spezifikation beschriebenen Daten in den Variablen, allgemeiner in den Laufzeitzuständen des Programms wiederfinden.

Das Verfahren, das ich hier skizzieren möchte, ist im Wesentlichen das in *Von der Spezifikation zur Implementation* (TeachSWT@Tü, 2002d) mit einer verbesserten Notation, und, wie ich hoffe, einem genaueren konzeptionellem Fundament. Ich werde das Verfahren in diesem Anhang soweit ausführen, wie ursprünglich im genannten Handout skizziert, und dann darlegen, wie es noch weiter präzisiert werden kann – denn einige Sachverhalte werde noch absichtlich offen lassen – und weiterhin, wie das Verfahren ausgebaut werden muss, um auch universell anwendbar zu sein[12].

A.2 Laufzeitzustände

Der einfachste Fall ist der einer Operation (nennen wir sie *foo*), die durch geeignete Vor- und Nachbedingungen charakterisiert wird, und in C durch einen Block von Anweisungen realisiert werden soll. Während der Ausführung eines C-Programms liegt ein Laufzeitzustand z_1, wenn der Anweisungsblock betreten wird, ein anderer, z_2, wenn der Block schließlich wieder verlassen wird. In dieser Veränderung – von z_1 nach z_2 – muss die datenverarbeitende Leistung der Implementation, d. h. des Anweisungsblocks, liegen. Die Frage ist: Welche Veränderungen (also welche Übergänge von z_1 nach z_2) entsprechen der Spezifikation und welche nicht? Und weiter: Welche Laufzeitzustände dürfen zu Beginn des Anweisungsblocks vorliegen, d. h. was muss der in der Ausführung zeitlich vorangegangene Programmcode sicherstellen?

Es bietet sich an, anzustreben, dass wir die Vorbedingung in eine Aussage über z_1 umwandeln und die Nachbedingung in eine Aussage über z_1 und z_2. Dies kann geschehen, indem wir die Eingabedaten aus z_1 herausprojizieren und darauf die Vorbedingung anwenden, und analog die Ausgabedaten aus z_2 herausprojizieren und darauf die Nachbedingung anwenden. Ganz allgemein gesprochen benötigen wir also Werkzeug, um Funktionen der folgenden Art zu konstruieren:

$$\begin{aligned} &\textit{proj-foo-input} &&: \textit{LZZ} \to \textit{input-data} \\ &\textit{proj-foo-output} &&: \textit{LZZ} \to \textit{output-data}\,. \end{aligned}$$

Dabei ist *LZZ* die Menge der Laufzeitzustände. Nun muss für z_1 gelten

$$\textit{pre-foo}(\textit{proj-foo-input}(z_1))$$

damit z_1 ein zulässiger Zustand ist, mit dem der Anweisungsblock betreten werden darf (initialer Zustand) und für alle zulässigen initialen Zustände z_1 muss die Implementation einen Zustand z_2 beim Verlassen des Anweisungsblock herstellen, für den gilt

$$\textit{post-foo}(\textit{proj-foo-input}(z_1), \textit{proj-foo-output}(z_2))\,,$$

[10] Insbesondere, wenn die Sprache, um die es geht, über gut 30 Jahre empirisch gewachsen ist.

[11] Der Preis, den man dafür zahlt, ist natürlich eingeschränkte Präzision des Verfahrens – d. h. an vielen Stellen ist dennoch freies Argumentieren in natürlicher Sprache nötig, und auch „scharfes Hinsehen“, um bestimmte Probleme zu erkennen – schließlich auch, dass die Verifikation nicht automatisierbar ist. Jedoch erhält man hier, wie wir sehen werden, sehr viel für sehr wenig Aufwand, was, glaube ich, trotzdem ein gutes Geschäft ist, vor allem, wenn die Alternative darin bestünde, wegen des Aufwandes eine aufwändigere, wenn auch präzisere, Methode gar nicht anzuwenden.

[12] Diese Erweiterung wird die am Ende angesprochene Projektion der Zeiger in den Modellbereich sein

damit man sagen darf, dass der Anweisungsblock der Spezifikation entspricht.

Wir müssen uns nun also fragen, wie wir die Laufzeitzustände eines C-Programms charakterisieren können und welche Mittel uns zur Verfügung stehen, die nötigen Projektionsfunktionen zu definieren.

Jeder Prozess, der ein C-Programm ausführt, durchläuft sequentiell eine Unzahl von Laufzeitzuständen. Diese entsprechen Zuständen des Speichers, der Variablen, des Stacks, des Callstacks der Funktionen, der Register, des ausführenden Prozessors, und so weiter. Wir sind aus Gründen, die bald klar werden werden, nur an Laufzeitzuständen interessiert, die sich eindeutig einer Position im Quelltext zuordnen lassen.

Diese Aussage soll im Sinne eines *Ausführungsmodells* aufgefasst werden, mit dem man sich die Funktion des Programmes erklären kann. Unbenommen bleibt, dass die Instruktionen durch einen optimierenden Compiler umgeordnet werden können. Dies muss jedoch in einer Weise geschehen, dass die datenverarbeitende Leistung des Codes gleich bleibt, soll also unsere Überlegungen nicht hindern.

Um was es hier geht, ist, dass der C-Standard selbst angibt, dass sich in vielen Fällen keine Ausführungsreihenfolge für Teilanweisungen festlegen lassen: Wir könnten deshalb nur schwer sagen, in welcher Reihenfolge die Laufzeitzustände auftreten, wieviele Zwischenergebnisse noch nicht vollständig verwertet wurden und welche Effekte bereits stattgefunden haben. Auch die Zuordnung zu einer Quelltextposition ist schwierig.

Dagegen kann für einige Stellen des Quelltextes die Vorstellung aufrechterhalten werden, dass zuerst die davor stehenden Anweisungen vollständig evaluiert werden, ehe die Operationen des folgenden Ausdrucks begonnen werden.[13] Es handelt sich dabei unter anderem um die Trennstelle zwischen Anweisungen [1,2], um das Ende des Argumentes eine *return*-Anweisung [3] bzw. einer Bedingung einer bedingten Anweisung [4] oder die booleschen Operatoren von Ausdrücken, die selbst nicht als echte Unterausdrücke anderer Ausdrücke auftreten [5,6]:

```
v = a + MINFOO * b; /*1*/
if (((*(p+OFFSET))==MAXFOO && /*5*/ (OFFSET==0)) /*4*/ {

   ...

} /*2*/

bar(&v) || /*6*/ getfoo(&c);
return v*c /*3*/ ;
```

Es ist nicht nur so, dass es Laufzeitzustände gibt, welche einem Überqueren von Quelltextposition der angedeuteten Art entsprechen, sondern diese Laufzeitzustände sind sogar noch besonders einfach: Da alle vorangehenden Operationen vollständig abgeschlossen sind, genügt es, sich unter dem Laufzeitzustand jeweils die Inhalte der lokalen Variablen (aller aktivierten Prozeduren), den Inhalt der globalen Variablen, den Zustand des Heap und des Aufrufstacks der Prozeduren vorzustellen. Dazu kommt noch an einigen dieser Stellen [3,4,5] das Resultat aus der Evaluierung des unmittelbar vorangehenden Ausdrucks, das in den folgenden Operationschritten weiter ausgewertet werden wird.

[13] Hier gibt es fast sicher eine Querverbindung zu den im C-Standard erwähnten *sequence points*. Was mich jedoch vorerst abschreckt, diese Verbindung weiter zu verfolgen, ist der folgende Satz aus der C-FAQ:

> A sequence point is a point in time (at the end of the evaluation of a full expression, or at the „||“, „&&“, „?:“, or comma operators, or just before a function call) at which the dust has settled and all side effects are guaranteed to be complete.

Ich bin hier nicht an einer „Abfolge der Zeit“ interessiert, sondern an einer Lokalisierung des Laufzeitzustandes im Quelltext und einer statischen Charakterisierung der Quelltextpunkte, für die das möglich ist.

Entsprechend dem anfänglich skizzierten Programm will ich es vermeiden, explizite, detaillierte Modelle der Menge der Laufzeitzustände zu entwerfen. Wie sich gleich herausstellen wird, müssen wir, um über den Effekt von Anweisungsblöcken zu reden, die Laufzeitzustände nicht vollständig charakterisieren können.

Fassen wir jedoch zuvor kurz zusammen:

- Im Laufe der Abarbeitung eines C-Programms durchläuft ein Prozess eine Reihe von Laufzeitzuständen.
- Wir sind nur an solchen Zuständen interessiert, die sich auch eindeutig einer Quelltextposition zuordnen lassen. Wir bezeichnen diese (vom konkret gegebenen Programm abhängige Menge) als *LZZ*.
- Die einem $z \in LZZ$ zugeordnete Quelltextposition soll als $pos(z)$ bezeichnet werden.

Die Aufgabe einer Semantik[14] der Programmiersprache wäre es nun, anzugeben, was zu einem gegebenen Laufzeitzustand z_1 der jeweils nächste (aus *LZZ*) sein wird, den der Prozess annehmen wird. Dies lässt sich in zwei Schritte zerlegen, und zwar

- In eine Spezifikation, was zu einer gegebenen Quelltextposition p_1 die Position des nächsten interessierenden Laufzeitzustandes sein wird.[15] Dies wird meistens einfach die, in der durch den Quelltext gegebenen Reihenfolge, nächste Position sein, manchmal jedoch (bei Schleifen und bedingten Anweisungen) auch vom augenblicklichen Laufzeitzustand abhängen, und zwar vom Ergebnis des letzten evaluierten Ausdrucks. Wir können über diesen Teil der Semantik als eine *Spezifikation des Kontrollflusses* denken.
- Eine Funktion, welche alle z_1 (die in p_1 auftreten können) auf die zugehörigen z_2 abbildet. Diese Funktion ist abhängig von der zwischen p_1 und p_2 liegenden Anweisung. Wir können sie als die *Semantik der Anweisung* bezeichnen.

Wir sind jedoch, wie bereits erläutert, nicht daran interessiert, eine Semantik von C aufzustellen, oder auch nur zu skizzieren. Vielmehr interessiert uns nun, einzuschränken, was dem Laufzeitzustand auf dem Weg von p_1 nach p_2 (die eventuell mehrere Schritte auseinanderliegen) überhaupt widerfahren kann, d. h. wieweit sich dieser überhaupt ändern kann (welche Effekte die Anweisung überhaupt bewirken kann), und wie der Laufzeitzustand in p_1 und p_2 beschrieben werden muss, um ihn mit der Spezifikation in Verbindung zu bringen.

Dabei wollen wir uns bezüglich des Quelltextes, der zwischen p_1 und p_2 liegen soll, auf Anweisungen und Anweisungsblöcke beschränken. Wie man sich später – wenn wir fertig sind – leicht überzeugen kann, ist es dank des der Tatsache, dass C Prozeduren „by value“ ruft, recht einfach, die Beschreibung von Prozeduraufrufen und -rückkehr ad hoc in das Verfahren einzupassen.

A.3 Kanonische Interpretation integraler Typen

Ganz allgemein müssen wir uns im Sinne des einleitend skizzierten Programms fragen, wo in z_1 die Eingabewerte für die implementierte Operation stecken und welche Teile von z_1

[14] Wir wollen – noch immer – keine Semantik von C entwerfen, es ist aber ganz nützlich, das, was wir hier tun, im Verhältnis zu dem, was man tun könnte, zu positionieren.

[15] Mit anderen Worten: Welche Anweisungen bzw. welcher Anweisungsblock als nächstes abgearbeitet werden wird.

die Ausgangswerte der Operation darstellen. Es ist klar, dass diese letzten Endes in Variablen abgelegt sein müssen, und das soll nun genauer beleuchtet werden. Da das Verfahren Abstraktionsfunktionen zu definieren für z_1 und z_2, d. h. Ein- und Ausgangszustand der Implementation vollkommen gleich ist, kümmern wir uns vorerst nicht um den Verarbeitungsprozess und untersuchen die Verhältnisse an einer beliebigen, aber festen Quelltextstelle (aus denen, die die vorgenannten Kriterien erfüllen), welche wir p nennen wollen. Jeder solchen Quelltextposition lässt sich eine Anzahl Terme aus der Programmiersprache C zuordnen, die nach den Sichtbarkeits- und Typregeln der Sprache *Lvalues* sind, d. h., Speicherstellen notieren, denen Werte zugewiesen werden können, und aus denen Werte ausgelesen werden können. Wir sind nur an einer Untermenge interessiert, die ich hier nicht vollständig charakterisieren will[16], was ich jedoch explizit ausschließen möchte, sind *Lvalues* in denen Arithmetik irgendeiner Art betrieben wird[17].

Nennen wir die betreffende syntaktische Untermenge von Ausdrücken, aus der wir schöpfen *LEXPR*, und die in p nach den Regeln der Sprache gültige Untermenge, welche *Lvalues* notieren, nennen wir $lexpr_p$. Die Menge der Laufzeitzustände aus der Menge *LZZ* aller Laufzeitzustände, welche sich p zuordnen lassen, wollen wir lzz_p nennen:

$$lzz_p = \{\, z \in LZZ \mid pos(z) = p \,\} .$$

Was wir nun suchen, ist ein Verfahren, Funktionen zu definieren, welche VDM-SL-Werte aus den Laufzeitzuständen lzz_p extrahieren, z. B.

$$proj : lzz_p \to \textit{vdm-sl-val} .$$

Dabei ist *vdm-sl-val* eine geeignete Menge aus dem Modell, beispielsweise die Wertemenge des Typs eines Parameters der spezifizierten Operation.

Die Darstellung der Werte, die hier herausprojiziert werden, müssen prinzipiell irgendwo in den Speicherinhalten der *Lvalues* aus $lexpr_p$ abgelegt sein. Unser Verfahren muss dem Rechnung tragen, dass die Darstellung über mehrere Speicherstücke verteilt sein kann, wie etwa in der bereits in *Von der Spezifikation zur Implementation* (TeachSWT@Tü, 2002d) zitierte Darstellung einer Textzeile:

```
typedef struct{
 char*  store;
 size_t dlen;
 size_t slen; } line  ;
```

Wir beginnen damit, dass wir den Inhalten von Lvalues mit integralem Typ (char, int, long, usw.) eine kanonische Interpretation geben, und zwar diejenige ganze Zahl, welche dem Variableninhalt entspricht. Diese Interpretation ist eine Funktion auf dem Laufzeitzustand z und dem betreffenden Ausdruck der den Lvalue designiert:

$$\begin{array}{rcl} \textit{abstr-canon} : LZZ \times LEXPR \to & \mathbb{Z} \\ (z, l) & \to & \ldots \quad , \end{array}$$

wobei $l \in expr_p$ sein muss und l im Kontext von $pos(z)$ einen Lvalue darstellen muss. Der Ausdruck l wählt gewissermaßen einen Teil des Laufzeitzustandes aus – der selektierte Speicherinhalt wird in eine ganze Zahl übersetzt.

Um die Notation elegant und übersichtlich zu halten, führen wir eine besondere Schreibweise für *abstr-canon* ein:

$$\lceil l \rfloor_z = \textit{abstr-canon}(z, l) .$$

[16] Hier hat man noch einen gewissen Spielraum, den vorgestellten Formalismus zu erweitern oder einzuschränken

[17] Also $a[i+15]$ oder $a[trunc(z*71,4)]]$

Den Ausdruck l notieren wir dabei in der in C üblichen Syntax. Die Klammern „$\lceil \ldots \rfloor$“ markieren also einen Bereich, in dem eine besondere Syntax der Notation gilt[18]. Unsere mathematische Notation außerhalb der Klammern soll im Wesentlichen gültiges VDM-SL sein.

Sei beispielsweise z ein Laufzeitzustand, *lp* ein Zeiger auf eine Struktur vom Typ *line*, dann ist

$$\lceil lp \triangleright dlen \rfloor_z$$

ein gültiger Ausdruck in der erweiterten Notation. Er gibt, nebenbei bemerkt, die Länge der Zeile an, auf die *lp* zu dem Zeitpunkt verweist, zu dem der Prozess sich im Zustand z befindet.

A.4 Abstraktionsfunktionen

Auf den kanonischen Interpretationen aufbauend lassen sich beliebig Abstraktionsfunktionen definieren. Diese interpretieren den Speicherinhalt, der von einem Lvalue – beschreiben durch einen Term $l \in lexpr_{pos(z)}$ – aus erreicht werden kann, indem sie den Term und den Laufzeitzustand auf den Wertebereich eines VDM-SL-Typen *val* abbilden.

$$\begin{array}{rll} abstr_{T,val} : LZZ \times LEXPR \to & val \\ (z, l) & \to & \ldots \end{array} ,$$

wobei $l \in lexpr_{pos(z)}$ sein muss und im Kontext von $pos(z)$ einen Lvalue vom Typ T notiert.

Dabei sind T und *val* hier nur Subskripte, mit denen wir verschiedene Funktionen voneinander unterscheiden, in der Anwendung wird man den Abstraktionsfunktionen dann doch eher problemspezifische Namen geben, wie *abstr-foo* für eine Funktion, die aus dem Laufzeitzustand VDM-SL-Werte des Typs *foo* (Elemente der Wertemenge des Typs *foo*) extrahiert.

Der Typ der Ausdrücke, auf denen die Abstraktionsfunktion operiert, ist dabei implizit – man wird darüber etwas bei der Definition der Funktion sagen müssen – es besteht jedoch keine Notwendigkeit, ihn im Funktionsnamen zu kodieren: Meistens wird es eine 1 : 1-Entsprechung der C-Typen, welche die Darstellung übernehmen, und der VDM-SL-Typen, die den Wertebereich der Abstraktion stellen, geben.

Wichtig ist allerdings, dass Abstraktionsfunktionen nur auf C-Ausdrücken $l \in lexpr_{pos(z)}$ agieren können, welche einen ganz bestimmten Typ haben, da alle aus dem Argument l der Abstraktionsfunktion abgeleiteten Ausdrücke einen bestimmten Typ haben müssen, wie man anhand der später folgenden Definitionsbeispiele noch sehen wird.

Zuvor jedoch wollen wir auch für die Abstraktionsfunktionen eine besondere Syntax einführen:

$$abstr_{T,val}\langle l \rangle_z = abstr_{T,val}(z, l) \; .$$

Wie schon bei den besonderen Klammern der kanonischen Interpretation markieren die spitzen Klammern „$\langle \ldots \rangle$“ Anfang und Ende eines Bereichs, in dem eine abweichende Syntax gilt: Zwischen diesen Klammern wird l in der üblichen C-Syntax geschrieben.

Abstraktionsfunktionen lassen sich nun mittels der bereits vorhandenen Notation definieren, indem sie auf zuvor definierte Abstraktionsfunktionen zurückgeführt oder letztlich in kanonische Interpretation aufgelöst werden. Dabei wird entweder der Speicherblock, auf den der Ausdruck $l \in lexpr_p$ verweist, in Teilblöcke zerlegt, indem l mit Selektoren (Namen von *struct*-Komponenten) erweitert wird (siehe dazu das folgende Beispiel zur Implementation von *fraction*), oder aber man folgt einem eingebetteten Zeiger zu anderen

[18] Die Bedeutung dieser anderen Syntax ist ein Term aus der Termmenge *LEXPR*. Ich will hier ausdrücklich betonen, dass hier keine Zeichenketten „zitiert“ werden.

Speicherstellen (siehe hierzu das übernächste Beispiel). Ersteres bedeutet, dass die Darstellung entlang ihrer Typstruktur rekursiv zerlegt wird, letzteres löst das Problem, über verschiedene Speicherblöcke verteilte Darstellungen zu einer Abstraktion zu sammeln.

Wir illustrieren dies an zwei Beispielen. Sei einmal definiert

```
typedef struct{
   long enumerator;
   long denominator;  }  fraction;
```

so können wir eine Interpretation (Abstraktion) der Darstellung folgendermaßen definieren: Für alle $z \in LZZ$ und $fr \in lexpr_{pos(z)}$ mit Typ *fraction* im Kontext von $pos(z)$ sei die gewünschte Abstraktion

$$\begin{array}{lll} abstr\text{-}fraction : LZZ \times LEXPR \to & \mathbb{N} \\ abstr\text{-}fraction \quad (z, fr) & \to & \lceil fr.enumerator \rfloor_z / \lceil fr.denominator \rfloor_z \end{array}$$

also

$$abstr\text{-}fraction\langle fr \rangle_z = \lceil fr.enumerator \rfloor_z / \lceil fr.denominator \rfloor_z \, .$$

Zum anderen Mal sei die Darstellung eines linearen, inhomogenen Gleichungssystems definiert als ein Kompositum aus einer Matrix und einem Vektor:

```
typedef struct{
     *matrix mt;
     *vector vc;  }  eqnsys;
```

Hier wurde, um die Darstellung zwischen verschiedenen Gleichungssystemen zu teilen, was an sich problematisch, aber nicht verboten ist, entschieden, lediglich durch eine Referenz auf die Darstellung einer Matrix bzw. eines Vektors zu verweisen. Für die Definition der Abstraktionsfunktion ergibt sich daraus kein Problem: Für alle $z \in LZZ$ und $e \in lexpr_{pos(z)}$ mit Typ *eqnsys* im Kontext von $pos(z)$ sei

$$\begin{array}{lll} abstr\text{-}eqnsys : LZZ \times LEXPR \to & linear\text{-}equation\text{-}system \\ abstr\text{-}eqnsys \quad (z, e) & \to & les \end{array}$$

mit

$$les.mat = abstract.matrix\langle e \rightarrowtriangle mt \rangle_z \quad \text{und} \quad les.vec = abstract.vector\langle e \rightarrowtriangle vc \rangle_z \, ,$$

was äquivalent ist zu

$$\begin{array}{rcl} abstr\text{-}eqnsys\langle e \rangle_z.mat & = & abstract.matrix\langle e \rightarrowtriangle mt \rangle_z \\ abstr\text{-}eqnsys\langle e \rangle_z.vec & = & abstract.vector\langle e \rightarrowtriangle vc \rangle_z \, . \end{array}$$

Dabei wird vorausgesetzt, dass *linear-equation-system* zuvor als zusammengesetzter Typ in VDM-SL definiert wurde, z. B.

$$\begin{array}{ll} 10.0 & linear\text{-}equation\text{-}system :: mat : model\text{-}matrix \\ .1 & \qquad\qquad\qquad\qquad vec : model\text{-}vector \end{array}$$

und *abstr-matrix* und *abstr-vector* bereits bekannt sind.

A.5 Implizite Quantifikation über Zustände

Ehe wir uns weitere, notwendige Notationselemente erarbeiten, wollen wir jedoch noch einige Vereinfachungen für das Aufschreiben von Abstraktionsfunktionen und kanonischen Interpretationen festlegen. Meist wird es nämlich so sein, dass die Aussagen, die mit Hilfe der Funktionen spezifiziert werden, sich nicht auf einen bestimmten, einzelnen Laufzeitzustand beziehen, sondern eine ganze Klasse äquivalenter Zustände betreffen – also beispielweise immer für die Zustände gelten (sollen), welche einem Punkt p_1 unmittelbar vor Betreten eines Anweisungsblocks entsprechen, etwa so:

$$\forall\, z \in LZZ \mid pos(z) = p_1 \bullet \mathit{bar\text{-}count}(\mathit{abstr\text{-}foo}\langle e \triangleright x \rangle_z) < 0$$

Dieser Fall ist geradezu typisch, da wir die Aussagen, die wir mit Hilfe der Abstraktionsfunktion machen wollen, wie zuvor schon erwähnt, an bestimmte Positionen des Quelltextes binden wollen. Die Quantifikation über Laufzeitzustände ist jedoch nachgerade lästig, so dass wir statt der obigen Aussagen etwas im folgenden Stil schreiben

$$\mathit{bar\text{-}count}(\mathit{abstr\text{-}foo}\langle e \triangleright x \rangle_{label}) < 0\,,$$

wobei *label* eine beliebige symbolische Schreibweise ist, die per Konvention eine Menge $M_{label} \subset LZZ$ von Laufzeitzuständen identifiziert. Der Allquantor wird impliziert, ausgeschrieben müsste die obige Aussage

$$\forall\, z \in M_{label} \bullet \mathit{bar\text{-}count}(\mathit{abstr\text{-}foo}\langle e \triangleright x \rangle_z) < 0$$

lauten. Ist die Menge der Laufzeitzustände aus dem Kontext klar, sprechen wir z. B. über alle Zustände, die in Abarbeitung einer bestimmten Prozedur auftreten können, dann können wir den *label* auch vollkommen weglassen:

$$\mathit{bar\text{-}count}(\mathit{abstr\text{-}foo}\langle e \triangleright x \rangle) < 0\,.$$

Für die kanonischen Interpretationen der integralen Typen sollen von nun an dieselben Konventionen gelten.

A.6 Quantifikation über Lvalues

Wenden wir uns wieder der Definition von Abstraktionsfunktionen zu. Die bis hierher erarbeitete Syntax genügt noch nicht, um für den anfänglich definierten C-Datentyp *line* eine Abstraktionsfunktion zu konstruieren. Der Punkt, an dem dies scheitert, ist der, dass wir im Kern eine Aussage formulieren müssen, welche darauf hinausläuft, dass *alle* im Array **store* abgelegten Zeichen einer Zeile entsprechen, etwa

$$\forall\, i \in 1 \ldots \lceil q.len \rfloor \bullet \mathit{abstr\text{-}char}\langle \mathit{store}[i\text{-}1] \rangle = (\mathit{abstr\text{-}line}\langle l \rangle)(i)$$

Diese Aussage ist aber syntaktisch nicht ganz sauber, da $(i\text{-}1)$ ein VDM-SL Ausdruck ist, wir aber innerhalb der Spitzklammern in einer an C angelehnten Syntax Terme[19] notieren.

Was wir nun benötigen, ist eine Möglichkeit, über C-Ausdrücke (Lvalues aus *LEXPR*) zu quantifizieren, bzw. diese mit Hilfe von VDM-SL-Ausdrücken, die über bestimmte Bereich laufen, zu generieren. Das eigentliche Problem hierbei ist, dass wir bisher nur Möglichkeiten besitzen, *literale* Elemente aus *LEXPR* zu auszudrücken. Da wir kein Modell der Menge

[19] Diese Terme können sehr wohl eine Repräsentation in der VDM-SL-Welt haben, und zwar in Gestalt der schon öfter zitierten Menge *LEXPR*. Diese definiert die Sprache, in der wir durch den Datenbestand des C-Programms navigieren.

LEXPR angegeben haben, können wir mit VDM-SL-Mitteln bis jetzt keine Ausdrücke aus dieser Menge angeben.

Eine solche Syntax ist jedoch leicht zu schaffen: Wir legen fest, dass an allen Stellen innerhalb der Spitzklammern, an denen in C eine *Ganzzahlkonstante* zulässig ist, es auch zulässig sein soll, einen VDM-SL-Ausdruck des Typs $\mathbb{Z}$ einzusetzen, der in Bananenklammern „$(\!| \ldots |\!)$“ eingefaßt ist:

$$\ldots \mathit{abstr\text{-}char}\langle \mathit{store}[(\!| i\text{-}1 |\!)]\rangle = \ldots .$$

In einem spezifischen Kontext, in dem der VDM-SL-Ausdruck einen bestimmten Wert *n* hat, interpretiert, soll die Bedeutung dieser erweiterten Ausdrücke der sein, dass anstelle des VDM-SL-Ausdrucks für den Term in Bananenklammern der Wert *n* eingesetzt wird.[20]

Das bedeutet in Wirklichkeit lediglich, dass wir, um Terme aus *LEXPR* zu *notieren*, die ja schon immer Ganzzahlen als 0-stellige Operationssymbole enthielten, nun auch VDM-SL-Ausdrücke verwenden können, was es uns ermöglicht, nun auch Funktionen aufzuschreiben, bzw. Ausdrücke vom Typ *LEXPR* mit einer variablen Bedeutung. Die Bananenklammer, das muss betont werden, ist ein rein syntaktisches Mittel. Sie ist *keine Funktion*, welche die VDM-SL-Werte wieder zurück nach C transportiert.

An Stellen, an denen die Bananenklammer direkt von eckigen Klammern (Arrayklammern) umfasst ist, wollen wir – um der Glätte der Notation willen – die Eckenklammern weglassen:

$$\ldots \mathit{abstr\text{-}char}\langle \mathit{store}(\!| i\text{-}1 |\!)\rangle = \ldots .$$

Mit dieser Erweiterung der Syntax gerüstet, können wir nun, als ein vollständiges Beispiel für den Einsatz aller drei bis hierher erarbeiteten Notationselemente, eine Definition für ein Abstraktionsfunktion des bereits definierten C-Typs *line* geben.

Sei der VDM-SL-Typ *zeile* definiert als[21]

11.0 $\quad \mathit{zeile} = \mathit{appchar}^*$

und $l \in \mathit{LEXPR}$ vom Typ *line*, dann sei

$$\begin{aligned} \mathit{abstr\text{-}line}\langle l\rangle \quad &= \quad z \quad \text{mit} \quad z : \mathit{zeile} \\ &\Leftrightarrow \quad \lceil l.\mathit{slen}\rceil \geq n \\ &\qquad \textbf{and} \ \ \forall\, i \in 1\ldots n \bullet z(i) = \mathit{abstr\text{-}char}\langle l \triangleright \mathit{store}[i\text{-}1]\rangle \\ &\qquad\quad \text{wobei} \quad n = \lceil l.\mathit{dlen}\rceil \end{aligned}$$

A.7 Zusammenfassung und Projektionsfunktionen

Fassen wir nochmals kurz zusammen, was wir bis zu diesem Punkt erarbeitet haben: Motiviert davon, dass wir Aussagen über die in einem Laufzeitzustand dargestellten Werte treffen wollen, haben wir zusätzliche Syntax definiert, mit welcher wir mittels C-Ausdrücken die Variablen, allgemeiner Lvalues, notieren, Werte aus Laufzeitzuständen „herausprojizieren“ können.

Die anfänglich gesuchten Projektionsfunktionen stellen einen direkten Zusammenhang her zwischen einem Parameter oder dem Ergebnis einer in VDM-SL spezifizierten Operation und den Variablen bzw. Lvalues in denen diese Werte in der Implementation gespeichert

[20] Wenn man möchte, kann man solche Ausdrücke als Funktionen von einer Potenzmenge von $\mathbb{Z}$ nach der Menge *LEXPR* auffassen

[21] Wir weichen hier etwas von dem in *Von der Spezifikation zur Implementation* (TeachSWT@Tü, 2002d) gegebenen Beispiel ab.

sind. Diese Projektionsfunktionen lassen sich nun durch ein einfaches Currying mit einem Term, der die Variable identifiziert, aus den Abstraktionsfunktionen erhalten. Sei zum Beispiel

```
12.0  op (f : foo) b : bar
 .1   pre ...
 .2   post ... ;
```

und werden die Typen *foo* und *bar* durch die C-Typen *FOO* und *BAR* dargestellt und die Darstellung durch die Abstraktionsfunktionen *abstr-foo* bzw. *abstr-bar* vermittelt.

Wollen wir nun zum Ausdruck bringen, dass f beim Betreten der Prozedur in der Variable $x : FOO$ und b beim Verlassen der Prozedur in der Variable $y : BAR$ gespeichert (dargestellt) sein wird, so schreiben wir:

$$\textit{proj-f}_{\textit{procedure-entry}} = \textit{abstr-foo}\langle x\rangle_{\textit{procedure-entry}}$$

$$\textit{proj-b}_{\textit{procedure-exit}} = \textit{abstr-bar}\langle x\rangle_{\textit{procedure-exit}}$$

Die vorgestellte Notation lässt sich übrigens auch dazu verwenden, über die nötigen Invarianten *der Darstellung* zu sprechen, also Strukturregeln, die für einen „korrekten“ Inhalt eines Teils des Speichers zu Laufzeit gelten müssen, damit dieser im Sinne des Modells interpretiert werden kann[22]. Ein Beispiel dafür befindet sich in dieser Lösungskizze im Abschnitt *Abstraktion und Fehlerbehandlung* (Seite 9). Dem soll hier nicht näher nachgegangen werden.

A.8 Vollständigkeit

Der Zweck der gezeigten Notation ist es, es zu ermöglichen, Funktionen zu definieren, welche Laufzeitzustände eines Prozesses, der ein C-Programm ausführt, auf Werte aus den Wertemengen beliebiger VDM-SL-Typen projizieren, und zu ermöglichen, Aussagen über Laufzeitzustände (Vor- und Nachbedingungen, Invarianten, allgemeine Zusicherungen) auszudrücken. Für sehr viele Zwecke genügen die vorgestellten Notationselemente, aber nicht für alle. Beispielweise sind Darstellungen, in denen die Verzeigerung der Datenblöcke Zyklen oder alternative Wege enthält, damit nicht zu beschreiben, da keines der bisher vorgestellten Notationselemente es ermöglicht, über die Gleichheit zweier Speicheradressen zu reden.

Ich möchte es für die Zwecke dieses Anhangs dennoch bei den bisher eingeführten Elementen belassen und werde erst zum Schluss Hinweise geben, wie die Erweiterung der Notation meines Erachtens weitergeführt werden kann, damit auch diese Fälle behandelt werden können.

Für’s Weitere möchte ich mich dagegen der Frage zuwenden, ob die Beschreibung der Laufzeitzustände zu Beginn und Ende eines Anweisungsblocks mittels des Herausproji-

[22] Verletzt der tatsächliche Laufzeitzustand in einem Kontext, in dem eine solche Invariante gelten sollte, diese, können Zustand und Berechnungen des Programms nicht im Sinne des Modells interpretiert werden. Das Programm ist dann im wahrsten Sinn des Wortes in einem undefinierten Zustand: Alles kann geschehen, und es gibt keine Garantien mehr über Leistungen und Verhalten des Programms. In der Praxis wird diese Situation durch schlampigen Entwurf enstehen (ein Programmteil produziert Daten, mit denen ein anderer, im Sinne des für ihn geltenden Modells und der für ihn geltenden Abstraktion, nichts anfangen kann) oder durch Überschreiben von Arraygrenzen oder Zugriff über uninitialisierte Zeiger, was zu Veränderung (Zerstörung) von Daten führt, mit denen der schreibende Programmteil eigentlich nichts zu tun haben sollte. In der Praxis wird es auf modernen Systemen mit virtuellem, geschütztem Speicher recht schnell zu einem *Segmentation Fault* als direkte oder indirekte Folge eines solchen Arbeitens in einem undefinierten Zustand kommen, allerdings vielleicht nicht, ehe nicht größerer Schaden angerichtet ist. Oft genug ist allerdings der Programmabbruch Schaden genug, hier muss man nur an die Kontrollsoftware in Luft- und Raumfahrt denken.

zierens von Werten über Lvalues überhaupt prinzipiell genügt, um die datenverarbeitende Leistung eines Anweisungsblocks, d. h. Veränderungen, die der Laufzeitzustand des Prozesses auf dem Weg durch die Anweisung erfährt, ausreichend zu beschreiben.

Ich will diese Frage präzisieren: Wenn der Anweisungsblock B betreten wird, liegt ein Laufzeitzustand z_1 aus einer Menge möglicher Zustände lzz_1 vor. Wird der Anweisungsblock verlassen, so liegt ein anderer Zustand z_2 aus einer möglichen Menge von Zuständen lzz_2 vor. Der Zustand z_2 ist eine kausale Folge des Zustands z_1, d. h. demselben Zustand z_1 folgt immer derselbe Zustand z_2[23] . Der Anweisungsblock bildet z_1 auf z_2 ab, wobei die Abbildung selbst vom Inhalt des Anweisungsblocks B abhängt. Diese Abbildung

$$\begin{aligned} S_B : lzz_1 &\to lzz_2 \\ z_1 &\to z_2 = S_B(z_1) \end{aligned}$$

ist nichts anderes als die *Semantik* des Anweisungsblocks.

Wir haben nun Mittel an der Hand, Aussagen über Zustände zu formulieren. Unser Bestreben ist es, eine Aussage $a_1(z_1)$ zu finden, deren Lösungsmenge genau die Definitionsmenge lzz_1 von S_B ist:

$$\{z \in LZZ \mid a_1(z) \text{ ist wahr}\} = \mathbf{dom}\, S_B = lzz_1$$

und weiterhin eine Aussage $a_2(z_1, z_2)$, welche $z_2 = S_A(z_1)$ eindeutig bestimmt:

$$\forall\, z_1 \in \mathbf{dom}\, S_B \bullet \exists_1\, z_2 \in lzz_2 \bullet a_2(z_1, z_2) \text{ ist wahr}$$

Die Frage ist nun die, ob sich in den Aussagen, die wir formulieren können, solche Aussagen finden lassen bzw. ob dies zumindest in jedem interessierenden Fall möglich ist. Dies könnte aus folgenden Gründen scheitern:

- Es gibt Zustandsteile, die durch die Lvalues in $lzz_{pos(z_1)}$, also zu Beginn des Anweisungsblocks, nicht erreicht werden können, aber in die Berechnung von Ergebnissen, d. h. Zustandsteile, die vom Ende des Anweisungsblocks aus erreicht werden können, einfließen.
- Durch den Anweisungsblock werden Zustandsteile verändert (Ergebnisse) erzeugt, die mittels der Lvalues, welche an $pos(z_2)$ gelten, nämlich die Menge $lzz_{pos(z2)}$, nicht erreicht werden können.
- Die formulierbaren Aussagen über z_2 können z_2 nicht ausreichend einschränken, d. h. z_2 kann abhängig vom z_1 nicht festgelegt werden.

Die erste Befürchtung lässt sich entkräften, d. h. es lassen sich Funktionen angeben, welche alle Projektionen am Ende des Anweisungsblock B (potentiellen Ergebnisse, Projektionen von z_2) als Bilder (Resultate) von Projektionen am Anfang des Anweisungsblocks (Projektionen von z_1) darstellen. Voraussetzung dafür ist, dass die Interpretation der vorgestellten Notation noch geringfügig variiert wird, wie ich im Folgenden darstellen werde.

Die zweite und die dritte Befürchtung dagegen sind durchaus berechtigt: Zum einen sind über Lvalues in $pos(z_1)$ Zustandsteile erreichbar, die durch Lvalues in z_2 nicht mehr erreicht werden können. Dies ist zum Beispiel der Fall, wenn innerhalb des Anweisungsblocks Zeiger „verloren“ gehen, d. h. überschrieben werden. Dies bedeutet nicht notwendig ein Speicherleck. Zwar kann man sich über die Meriten dieses Programmierstils streiten[24], jedoch bleibt, dass es mit den bis hierher vorgestellten Notationselementen nicht möglich ist,

[23] Ich gehe hier für's Weitere davon aus, dass das Programm vollständig deterministisch ist. Ansonsten lassen sich gleichartige Überlegungen anstellen, dabei müssen aber einige Funktionen durch Relationen ersetzt werden.

[24] Ich persönlich glaube, dass Zeiger nur an Abstraktionsgrenzen, d. h. beim Ein- und Austritt aus Prozeduren dupliziert werden oder verloren gehen sollten.

die Effekte solcher durchaus legitimer Programmteile zu beschreiben. Zum anderen fixiert, wie sich herausstellen wird, die vorgestellte Notation z_2 nicht eindeutig: Der verbleibende Freiheitsgrad betrifft Aussagen darüber, ob Zeiger identisch oder gar neu sind. Dies ist nicht sehr verwunderlich, nachdem ja, wie bereits angesprochen, in den bisher existierenden Notationselementen keine Möglichkeit besteht, über die Identität oder Verschiedenheit von Zeigern zu sprechen. Interessant ist jedoch, dass es sehr wohl möglich ist, die datenverarbeitende Leistung im Sinne einer Abbildung von Werten auf Werte vollständig zu charakterisieren[25] und dass die Spezifikation der Zeigerdisziplin (und damit die Festlegung einer Speicherdisziplin) davon derart unabhängig ist.

Ich werde darauf später noch unvollständig eingehen. Kümmern wir uns zuerst um die Widerlegung des ersten Einwands und um die Modifikationen die wir dafür an der Notation vornehmen müssen.

Der Grundgedanke dahinter, sich auf Aussagen über die in Lvalues gespeicherten Daten zu konzentrieren, ist der, dass wir zum einen annahmen, dass Programmzustand in Lvalues gespeichert ist, zumindest solange wir uns nur um die Verhältnisse in den von uns ausgezeichneten Quelltextpositionen kümmern, zum anderen, dass die Anweisungen nur Zustand manipulieren können, den sie über entsprechende Ausdrücke, die an dieser Stelle des Programms Lvalues bedeuten, adressieren können. Dieser Grundgedanke ist zwar im Wesentlichen richtig, aber nicht ganz.

Die erste Komplikation entsteht dadurch, dass Zustand in Lvalues l abgelegt sein könnte, die einen opaken Typ T_l haben[26]. Den Inhalt von Instanzen solcher Typen können wir nicht durch strukturelle Zerlegung entlang der Typstruktur (durch Erweiterung des Ausdrucks mit Selektoren, d. h. Feldnamen, oder Dereferenzierung) in der erläuterten Methode beschreiben. Wir gingen ja bisher davon aus, dass die Ausdrücke, welche die Daten adressieren, im Kontext der Quelltextposition, die wir betrachten, gültige Lvalues sein sollen. Die erweiterten Ausdrücke $l \mathbin{-\!\!\triangleright} x$ oder $l.f$ können jedoch keine solchen gültigen Ausdrücke sein, da ja der Typ von l opak ist.

Das Problem entsteht hier dadurch, dass, obwohl wir noch keine Mittel haben, über die Inhalte solcher Variablen zu reden, der in Rede stehende Anweisungsblock trotzdem die Möglichkeit hat, diese Inhalte zu manipulieren, indem er Prozeduren eines Moduls aufruft, das über die vollständige Typinformation verfügt, und innerhalb dessen dann sehr wohl ein Zugriff auf die Darstellung von T_l möglich ist.

Es (das Problem) lässt sich leicht lösen, indem wir eine Abstraktionsfunktion für Lvalues des Typs T_l vorlegen, deren *Definition* im Sichtbarkeitsbereich (Kontext, insbesondere Typkontext) des Moduls, welches T exportiert, verstanden werden muss, die aber außerhalb des betreffenden Moduls (d. h. für die Argumentation über Quelltextstellen außerhalb des betreffenden Moduls) *verwendet* werden kann.

Dies ist im Grunde keine Erweiterung der bisherigen Notation, nur eine Klärung der Frage, in welchem Kontext der Typ eines der erweiterten Ausdrücke aus *LEXPR* in der Definition der Abstraktionsfunktion zu bestimmen ist. Hier wird offenbar, dass wir Abstraktionsfunktionen in einem Typkontext definieren können sollen und unter Umständen in einem anderen anwenden wollen. Wir lösen also das Problem des im Kontext einer Quelltextposition p

[25] Vorausgesetzt man hält strenge Typdisziplin und verwandelt Zeiger nicht in Ganzzahlen oder umgekehrt.

[26] C besitzt keine durchgehende Unterstützung für die Implementierung opaker Typen – stattdessen wird diese durch Vereinbarung von Konventionen, Typumwandlungen (z. B. den bekannten *void*-Zeiger-Trick) etc. simuliert, siehe auch TeachSWT@Tü (2002b). Wir wollen hier trotzdem so tun, als ob es opake Typen gäbe, da es für Zwecke dieser Diskussion nicht interessiert, ob der Compiler die Interna des betreffenden Typs kennt und es nur der Programmierer ist, der sich für die Darstellung per Konvention nicht interessieren darf. So oder so darf Wissen über Interna von T_l nicht in Argumentation über die Klientencode des Moduls einfließen, das T_l opak exportiert.

Zudem können wir, wenn wir die Existenz opaker Typen vorgeben und sei es auch nur als Konvention, die Ergebnisse der Diskussion unmittelbar auf andere imperative Sprachen mit einem Modulsystem, wie Ada, Modula-2 oder Oberon übertragen.

opaken Typs T_l indem wir, wenn wir über den Inhalt von l reden wollen, auf eine geeignete Abstraktionsfunktion von T_l zurückgreifen.[27]

Weiterhin kann Zustand auch in anderen Modulen eingekapselt sein. Auch dieser kann durch einen Aufruf entsprechender Prozeduren verändert oder ausgelesen werden, ist jedoch überhaupt nicht über einen Lvalue beschreibbar, der an einem Aufrufort dieser Prozeduren gilt (was für die opaken Typen noch möglich war, in die wir nicht hineinsehen konnten).

Wenn wir annehmen, dass der Zustand innerhalb des betreffenden Moduls – nennen wir es M – in einer statischen Variablen v gespeichert sein muss, erweist sich dieses Problem jedoch als eine Variante des vorangegangenen mit den opaken Typen. Wir können es auf analoge Art lösen, indem wir eine Abstraktionsfunktion *state-M*$\langle\rangle$ definieren, die in den Spitzklammern kein Argument nimmt[28], deren Definition wieder im Typkontext von M zu verstehen ist, auf v zurückgreift und so eine Abstraktion des Modulzustandes liefert.

Bibliotheksfunktionen aus der *libc*, die intern Speicherplatz zur Statusspeicherung allozieren (wie etwa *getpwent*() und Genossen) können analog beschrieben werden. Ebenso kann über externen Zustand abstrahiert werden, indem nullstellige Abstraktionsfunktionen eingesetzt werden (die dann jedoch mit anderen Mitteln, etwa auf einem Modell des externen Zustandes, definiert werden müssen).

Wovon wir uns bis hierher überzeugt haben, ist, dass wir alle Daten, die der Anweisungsblock lesen oder manipulieren kann, mittels Abstraktionsfunktionen beschreiben können (d. h. in das Modell projizieren können), die

- entweder anderweitig definiert wurden und die Inhalte opaker Typen beschreiben oder über Modulzustände und externe Zustände abstrahieren,
- oder die wir unter Verwendung von Lvalues, die *innerhalb* des Anweisungsblocks gültig sind, definieren können.

Die ersteren Abstraktionsfunktionen repräsentieren Daten, welche nur über Prozeduraufrufe manipuliert werden können, während die letzteren Daten repräsentieren, die direkt als Variableninhalte zur Verfügung stehen und durch Zuweisung manipuliert werden können.

Entscheidend ist hier aber die Formulierung „innerhalb des Anweisungsblocks“. Genügt es nun, sich auf Lvalues zu beschränken, die am Anfang p_1 oder am Ende p_2 des Anweisungsblocks gültig sind, d. h. Lvalues aus lzz_{p_1} und lzz_{p_2}?

Das hauptsächliche Problem, das sich hier stellt, ist, dass Abstraktionsfunktionen ja nur einen Blick auf einen *Ausschnitt* des vollständigen Laufzeitzustandes gewähren. Man könnte vielleicht vermuten, dass Programmteile innerhalb des Anweisungsblocks Zugriff auf einen weit größeren Teil des Laufzeitzustandes haben, als er in den Quelltextpositionen p_1 und p_2 sichtbar ist, und entweder dieser Zustand verändert wird – also Effekte auftreten, die wir mit Aussagen über den in p_2 sichtbaren Zustandsausschnitt nicht beschreiben können, oder dieser Zustand in die Berechnungen einfließt, also der Zustand in p_2 die kausale Folge von Zustandsanteilen ist, die wir mit Projektionen aus dem in p_1 sichtbaren Ausschnitt des Zustands nicht beschreiben können.

Dies ist jedoch im Wesentlichen nur ein Scheinproblem. Betrachten wir, wie es sein kann, dass innerhalb des Anweisungsblocks weitere Lvalues gültig werden. Zum könnte der Anweisungsblock einen Unterblock mit lokalen Variablen enthalten, zum anderen könnten

[27] Im Grunde ist des geradezu programmatisch: Um außerhalb des definierten Moduls über einen *opaken* Typ reden zu können, müssen wir auf seine *Abstraktions*funktion zurückgreifen, d. h. über die Darstellung des Typen *abstrahieren.* Opake Typen haben also irgendwie eine tiefere Bedeutung als die eines rein technischen Zugriffsschutzes, der den schlampigen Programmierer stolpern lassen soll.

[28] Eigentlich ist *state-M*$\langle\rangle$ also einstellig, da sie natürlich – in unserer Schreibweise implizit – einen Laufzeitzustand als Argument nimmt.

weitere Zeiger beschafft werde, entweder durch die Allokation von Speicher oder durch die Extraktion der Zeiger aus opaken Strukturen.

Im ersteren Fall werden die Lvalues mit dem Ende des lokalen Unterblocks wieder ungültig. Sie sind also kein in p_2 unbeschreibbarer Zustandsteil, der durch den Anweisungsblock verändert werden könnte. Solange die lokalen Variablen korrekt initialisiert werden – und dies kann nur aus Zustandsteilen geschehen, die vorher zugänglich waren, oder mit Konstanten – handelt es sich um eine Datenfluss von den in p_1 beschreibbaren Werten zu den in p_2 beschreibbaren Werten: Es fließen also auch keine in p_1 unzugänglichen Daten in das Resultat der Berechnung mit ein.

Komplizierter werden die Verhältnisse, wenn innerhalb des Anweisungsblock weitere Zeiger beschafft werden. Hier müssen wir zwei Fälle unterscheiden: Zum einen kann es sein, dass die Referenz, über die wir gerade reden, mittels eines Prozeduraufrufs aus dem gekapselten Zustand eines Moduls oder aus einem opaken Typen extrahiert wird. Dabei müssen Werte, die über $lexpr_{pos(z_1)}$ erreichbar sind, als Parameter einfließen. Wenn wir Abstraktionsfunktionen haben, die den gekapselten Zustand oder den opaken Typ auf ein Modell abbilden, so ist dieser Prozess prinzipiell beschreibbar, d. h. im Modell nachvollziehbar. Das heisst, es genügt, von den in $pos(z_1)$ zugänglichen Zustandsteilen auszugehen, um die Verhältnisse in $pos(z_2)$ angeben zu können. Zum anderen kann es sein, dass eine Referenz innerhalb des Anweisungsblocks z. B. mittels *malloc*() alloziert wird. Wird der dazugehörige Speicher wiederum vor Gebrauch korrekt initialisiert, so nimmt der dazugehörige Datenfluss, der in die aus z_2 herausprojizierten Werte einfließt, noch immer in den aus z_1 herausprojizierbaren Werten seinen Anfang. Daraus entsteht also kein Problem, unabhängig davon, ob der allozierte Block wieder freigegeben wird, oder ob der erhaltene Zeiger in die Datenstruktur, die das Resultat der Berechnung darstellt, integriert wird.

A.9 Kanonische Interpretation für Zeiger

In den vorangegangenen Abschnitten wurden schon kurz einige Situationen angedeutet, die mit dem bisher verfolgten Herausprojizieren von Werten nur sehr unvollständig zu beschreiben sind: Dies betrifft Situationen in den es nötig ist, explizit über Zeiger zu reden, statt sie nur zu Navigation innerhalb des Laufzeitzustandes zu verwenden. Ich will nun eine kurze Liste solcher Situationen geben und im Anschluß andeuten, wie das bis hierher vorgestellte Verfahren erweitert werden muss, um die entsprechenden Ausdrucksmittel zu erhalten.

Es handelt sich übrigens bei den Aussagen, die mit diesen Mitteln formuliert werden können, um zusätzliche Anforderungen, die auf dem Weg von einer funktionalen Spezifikation zu einer imperativen Implementation hinzukommen: Funktionale Spezifikation redet eigentlich nur über Werte und ihre Zusammenhänge, aber nicht über Speicherplatz und die Überlappung von Darstellungen (auch wenn z. B. ein Heap explizit modelliert werden kann). Insbesondere der Effekt eines Anweisungsblocks, der, wie im vorangegangenen Abschnitt kurz angesprochen, Referenzen verliert, ist auf funktionale Art sehr schwierig zu beschreiben.

Welche Situationen können nun auftreten, in denen es notwendig wird, explizit über Zeiger zu reden?

- Bei der Abarbeitung des Anweisungsblocks wird ein Zeiger, der am Anfang des Anweisungsblocks p_1 erreichbar war, überschrieben, nachdem der dahinterstehende Speicher modifiziert wurde. Mittels Lvalues, die am Ende des Anweisungsblocks gelten, bei p_2, sind diese Änderungen des Laufzeitzustandes nicht zu beschreiben. Dieses Situation wurde zuvor schon als „verlorener Zeiger bezeichnet“. Was man hier formulieren können müsste, aber mit der zur Verfügung stehenden Notation nicht

kann, ist eine Aussage über die Daten, an die man gelangt, wenn man einen Zeiger aus lzz_{p_1} zu dem Zeitpunkt, zu dem die Ausführung p_2 erreicht hat, dereferenziert. Mit der bisher erarbeiteten Notation können wir über den Zustand beim Erreichen von p_2 nur Aussagen über Zustandsteile machen, die auch von z_2 aus erreichbar sind.

- Mit der bisher erarbeiteten Notation kann kein Unterschied formuliert werden zwischen den Leistungen der beiden folgenden Quelltextfragmente:

```
int* a;
int* b;

    /* ... */

a=b;
```

oder

```
int* a;
int* b;

    /* ... */

(*a)=(*b);
```

Das bedeutet, mit den bisherigen Mitteln können keine Aussagen über *Representation Sharing* getroffen werden. Auch der Unterschied zwischen *deep copy* und *shallow copy* kann nicht zum Ausdruck gebracht werden. Dies wäre jedoch möglich, wenn es ein Mittel gäbe, die Gleichheit oder Verschiedenheit von Zeigern zum Ausdruck zu bringen.

- Die Tatsache, dass ein Zeiger frisch alloziert wurde (d. h. innerhalb des Anweisungsblocks Speicher belegt wurde, anstatt die Ergebnisse in bereits zugänglichem Speicher abzulegen), kann bisher nicht zum Ausdruck gebracht werden. Um dies zu tun, ist zumindest ansatzweise eine Beschreibung des Heapzustandes nötig.

- Zirkuläre Datenstrukturen, z. B. bei einem Ringpuffer eine Liste von Elementen, die am Ende wieder auf den Anfang zurückverweist, oder allgemeiner Datenstrukturen mit alternativen Dereferenzierungspfaden, die dann doch auf denselben Speicherblock verweisen, können nicht beschrieben werden, wenn kein Möglichkeit besteht, über die Gleichheit der Werte zu reden, die in verschiedenen Lvalues mit einem Zeigertyp gespeichert sind.

Die meisten dieser Defizite können behoben werden, indem man ein Mittel schafft, über die Gleichheit von Zeigern zu reden. Dies kann geschehen, ohne dass man ein explizites Modell für den Speicher entwirft. Hierfür muss eine kanonische Interpretation für Zeiger (die ja selbst kein strukturierter Typ sind) eingeführt werden. Diese kann und soll aber keine Abbildung in die natürlichen Zahlen sein, wie bei den integralen Typen. Vielmehr wird man einen gesonderten semantischen Bereich einführen müssen:

$$\begin{array}{rlll} \textit{abstr-pointer} : & \textit{LZZ} \times \textit{LEXPR} & \to & \textit{ADDR} \\ & (z, l) & \to & \ldots \end{array} ,$$

wobei $l \in \textit{LEXPR}$ im Kontext von $pos(z)$ einen Zeigertyp hat. Um Gleichheit von Zeigern ausdrücken zu können, genügt, wenn *ADDR* ein Token-Typ ist, aber wenn noch weitere

Ausdrucksmöglichkeiten benötigt werden, kann *ADDR* mit einer geeigneten Struktur versehen werden, also z. B. als Paare von Basiszeigern (Implikation: auf Speicherblöcke) und Offsets definiert werden.

Um über den Zustand hinter Zeigern bei Laufzeitzuständen z zu reden, in denen diese Zeiger nicht mehr in $lzz_{pos(z)}$ zugänglich sind, wird weitere Syntax benötigt, mit der Elemente der Menge *ADDR* wieder in Lvalues $l \in$ *LEXPR* „zurückverwandelt“ werden können, analog der bereits besprochenen Bananenklammernotation.

Und schließlich muss für einige Zwecke, wie bereits angedeutet, zumindest ein teilweises Modell für den Heapzustand entworfen werden, das mindestens erlaubt, die „Neuheit“ und die „Gültigkeit“ von Zeigern zum Ausdruck zu bringen.

Dieses Programm soll im Folgenden nicht umgesetzt werden. Es sei noch angemerkt, dass es selbst mit diesen Mitteln schwer werden wird, die Abwesenheit von Speicherlecks auszudrücken: Dazu wäre es nötig, Methoden zu entwickeln, Verantwortlichkeiten für Verwaltung, insbesondere Freigabe von Zeigern (und eventuell auch anderen Ressourcereferenzen) zum Ausdruck zu bringen. Dies ist, allgemein gesprochen, nichts, was so einfach aus dem Laufzeitzustand geschlossen werden könnte, und alle Versuche, die Notation in dieser Richtung zu erweitern, gehen weit über den ursprünglich Ansatz – durch Projektion von Werten aus dem Laufzeitzustand in das Modell über die Leistungen der Implementation zu reden – hinaus.

Da überall, wo explizit über Zeiger geredet werden muss – also Projektionen auf Werte nicht genügen – Schnittstellenkontrakte dazu tendieren, recht kompliziert zu werden, und damit auch die Wahrscheinlichkeit für Missverständnisse und Fehler steigt, empfiehlt es sich, die Implementation so zu gestalten, dass die Notwendigkeit mit nichtlokalen Zeigern umzugehen (d. h. Zeigern, die Schnittstellen in die eine oder andere Richtung überqueren und die nicht in einem opaken Typen verborgen sind), soweit als möglich vermieden wird (siehe dazu auch die Ausführungen in Lösungsskizze 5 zu den opaken Typen: TeachSWT@Tü, 2002b).

Wo es doch nicht zu vermeiden, kann man die zusätzlich nötigen Aussagen über Zeiger – anstatt sich der in diesem Abschnitt angedeuteten Erweiterung der Notation zu bedienen, auch in natürlicher Sprache zum Ausdruck bringen. Ist dies nicht mehr möglich, so ist das – meines Erachtens – ein recht guter Indikator für einen falschen Implementationsansatz, den man dann wohl nochmal überdenken sollte.

A.10 Verbesserungen und Ergänzungen

Im vorangegangenen Text habe ich einige Probleme und Feinheiten zuweilen vernachlässigt, um einen Überblick über das zugrundeliegende Konzept zu geben. Ich möchte nun in Kurzform andeuten, wo das Verfahren noch ergänzt werden muss oder präzisiert werden kann.

- Zuweilen ist das Ergebnis der Auswertung des unmittelbar vorangegangen Ausdrucks Bestandteil des Laufzeitzustandes, da davon der weitere Kontrollfluss (bedingte Anweisung) oder das Ergebnis einer Funktion abhängt, beispielsweise an den im Folgenden mit [1,2] markierten Stellen.

  ```
  if (b*b<4*a*c) /*1*/ { ... } else { ... };
     ...
  return -b+r/(2*a) /*2*/ ;
  ```

  Über dieses Ergebnis kann mit der eingeführten Notation nicht gesprochen werden, da der Speicher, in dem es vorliegt (ein Register oder eine Scratch-Area auf dem

Stack) keinen Namen (Identifier) hat, ihm also kein $l \in \mathit{LEXPR}$ entspricht. Ich schlage vor, hierfür *LEXPR* um ein spezielles Symbol – sagen wir „□“ – zu erweitern, so dass eine kanonische Interpretation oder eine Projektion dieses Ergebnisses aus dem Laufzeitzustand *z* durch eine Abstraktionsfunktion *abstr-foo* mit den Ausdrücken

$$\lceil \square \rfloor$$

oder

$$\mathit{abstr\text{-}foo}\langle \square \rangle$$

notiert werden kann. Mann kann über □ als eine anonyme lokale Variable denken und es zum Beispiel als *Ergebnisregister* bezeichnen.

- Nicht alle syntaktisch korrekten und im Kontext eine Quelltextstelle *p* wohlgetypten Lvalues $l \in \mathit{LEXPR}$ verweisen zur Laufzeit auch auf gültigen Speicher. Ein solcher Fall ist z. B. wenn über einen Nullzeiger dereferenziert wird, oder Arraygrenzen überschritten werden. Hier stellt sich dann die Frage, welche Bedeutung (d. h. welches Ergebnis) dann die angegebenen Ausdrücke, die Laufzeitzustand und Lvalue *l* in eine Menge von VDM-SL-Werten abbilden, besitzen sollen.

  Eine Möglichkeit bestünde darin, einen speziellen Wert einzuführen, sagen wir **sigsegv**, der dazu dient, einen solchen Lvalue anzuzeigen. Beispielsweise wäre dann

  $$\mathit{abstr\text{-}foo}\langle p \triangleright x \rangle = \mathbf{sigsegv},$$

  wenn *p* ein Nullzeiger wäre. Damit stünde dann auch eine sehr explizite Möglichkeit zur Verfügung, über illegitime Speicherreferenzen zu reden.

  Mathematisch (und damit konzeptuell) eleganter wäre es allerdings, sich darauf zurückzuziehen, dass es sich bei den Abstraktionsfunktionen sowieso um *partielle Funktionen* handelt, und zu fordern, dass Paare (l,z) mit $l \in \mathit{LEXPR}$ und $z \in \mathit{LZZ}$, in denen *l* auf ungültigen Speicher verweist, nicht im Definitionsbereich der Abstraktionsfunktionen liegen, die in $\mathit{pos}(z)$ definiert werden können. Daraus würden dann natürlich in bestimmten Anwendungen auch gewisse Beweisverpflichtungen über die Definitionsbereiche der verwendeten Ausdrücke (Abstraktionsfunktionen, Aussagen) resultieren, da man ja bekanntlich aus einer Aussage, deren Anwendungsbereich leer ist, alles beweisen kann.

- Benötigt man einen speziellen Wert *undefined*, auf den die Ergebnisse von Rechenoperationen abgebildet werden können, die laut C-Standard undefiniert sind, also z. B. in

  ```
  /*1*/ a = INT_MAX + 42 /*2*/; ?
  ```

  Nach meinem Dafürhalten ist das nicht notwendig, jedoch sollte man die Regeln, nach denen man anhand der Programmanweisung aus den Verhältnissen in [1] auf die Verhältnisse in [2] schließt, so gestalten, dass sich keinerlei Aussagen über die Verhältnisse in [2] ableiten lassen, wenn die Anweisung undefiniertes Verhalten bewirkt.

- Im vorangegangenen Text wurden die Menge *LEXPR* und *LZZ* informell eingeführt. Diese Mengen können aber auch als VDM-SL-Typen definiert werden.

- Wir haben uns im vorliegenden Text hauptsächlich mit der Frage beschäftigt, wie man Anforderungen an die Implementation formuliert. Letzten Endes wird man aber die Implementation auch prüfen wollen, d. h. sich davon überzeugen wollen, dass sie den Anforderungen entspricht. Dazu will man aus dem Verhältnissen welche vor dem Abarbeiten eines Anweisungsblocks gelten, anhand der im Block enthaltenen

Anweisungen Schlussfolgerungen ziehen über die Verhältnisse, die nach dem Abarbeiten des Anweisungsblocks gelten. Damit entsteht nun doch das Bedürfnis, eine Semantik für die Programmiersprache (in unserem Fall C) zur Verfügung zu haben, da es eben die Aufgabe der Semantik ist, die Beweisregeln zu Verfügung zu stellen, nach denen diese Folgerungen gezogen werden können.

In der Praxis kann jedoch die Semantik vieler Anweisungen ad hoc aufgeschrieben werden, und diejenigen, bei denen dies nicht möglich ist, sollte man vermutlich im Interesse einer Verständlichkeit des Programms nochmal überdenken.

A.11 Zusammenfassung

Was haben wir erreicht? Wir haben eine einfache, erweiterbare Notation vorgestellt, welche uns gestattet, die Entsprechung zwischen einer funktionalen, modellbasierten Spezifikation, und einer Implementation in einer imperativen Sprache auszudrücken. Dies geschieht, indem wir mit Hilfe der Notation Abstraktionsfunktionen definieren, die aus hypothetischen Laufzeitzuständen des Programms Werte des Modellbereichs extrahieren, und Aussagen über diese Werte in der Modellierungssprache (hier: VDM-SL) definieren.

Dabei vermeiden wir, ein explizites Modell der Laufzeitzustände zu entwerfen, indem wir Teile des Laufzeitzustandes mit syntaktischen Mitteln der Implementationssprache selektieren (*LEXPR*), eine kanonische Interpretation nur für sehr einfache Typen geben ($\lceil \ldots \rfloor$) und damit die Mittel stellen, Interpretationen komplizierter Darstellungen durch Rekursion entlang der Typstruktur zu definieren.

Das gezeigte Verfahren muss als halbformal bezeichnet werden, da eine explizite Semantik der Implementationsprache fehlt und der Effekt von Anweisungen für Argumentationszwecke ad hoc beschrieben werden muss. Gerade dies senkt jedoch die Einstiegsschwelle für praktische und didaktische Anwendung erheblich, so dass man hier sehr viel Nutzen für sehr wenig Aufwand erhält: Ein präzises Vokabular für das Sprechen über Programmzustände.

Es ist offensichtlich, dass analoge Verbindungen (Brückennotationen) auch zwischen anderen Modellierungssprachen und anderen imperativen Sprachen geschaffen werden können.

Literatur

[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.: Lösungsskizze 5: Entwurf eines Disassembler. In: (TeachSWT@Tü, 2002a). – `handouts/loesung-05.vdm`

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

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