FACHHOCHSCHULE WEDEL

Diplomarbeit

in der Fachrichtung

Technische Informatik

Thema:

Formale Spezifikation, Entwurf und Realisierung
einer automatischen Speicherverwaltung für
die C++-Schnittstelle der VDM Class Library

Eingereicht von:	Thorsten Tews Hinter der Linah 42 21614 Buxtehude Tel. (04161) 61706 Matr.-Nr. 8359
Abgegeben am:	22. Februar 1994
Referent (FH Wedel):	Prof. Dr. Uwe Schmidt

Inhaltsverzeichnis

Abbildungsverzeichnis

1 Einleitung

1.1 Vienna Development Method

Die *Vienna Development Method* (VDM) ist eine formale Methode für die Spezifikation und das Design von Programmen, die in den frühen siebziger Jahren von IBM-Mitarbeitern in Wien entwickelt wurde. Sie basiert auf Konzepten der diskreten Mathematik. Aufgrund dieses Ursprungs zeichnet sich VDM[1] durch die folgenden prinzipiellen Vorteile gegenüber anderen, nicht formalen Spezifikations- und Designmethoden aus:

1. **Eindeutigkeit:** im Gegensatz zu den in anderen Methoden häufig verwendeten umgangssprachlichen Beschreibungen, Grafiken oder Pseudocodes treten bei formalen Methoden keine Mehrdeutigkeiten bei der Interpretation der Aussagen auf, da die Semantik der einzelnen Elemente einer Spezifikation oder eines Designs eindeutig ist.

2. **Beweisbarkeit:** formale Methoden erlauben es Korrektheitsbeweise zu führen. Es kann z.B. streng formal bewiesen werden, daß ein Programm seine Spezifikation erfüllt (siehe [AI91, Kap. 11.2]); hierfür ist jedoch Voraussetzung, daß auch die Semantik der verwendeten Programmiersprache formal spezifiziert wurde[2]. Diese Möglichkeit ist besonders beim Einsatz von Programmen in sicherheitsrelevanten Bereichen wichtig. In der Praxis sind solche Beweise häufig sehr schwierig; eine maschinelle Unterstützung des Beweisprozesses ist jedoch prinzipiell möglich.

3. **Implementationsunabhängigkeit:** die Spezifikation von Daten und Operationen erfolgt auf einem sehr abstrakten Niveau und ist somit völlig frei von Implementationsdetails.

4. **Prägnanz:** durch Verwendung von mathematischer Notation sind Spezifikationen und Designs in formalen Methoden im Vergleich zu anderen Methoden kurz und prägnant. Dies macht sie — ausreichende Erfahrung mit der Notation vorausgesetzt — lesbarer und verständlicher.

Aufgrund dieser Vorteile haben formale Methoden wie VDM sowohl im akademischen als auch im industriellen Bereich ein reges Interesse hervorgerufen.

Auf den eigentlichen Ablauf der Spezifikation und des Designs mit VDM soll hier nicht weiter eingegangen werden. Es sei vielmehr auf die entsprechende Literatur verwiesen (siehe [AI91],[JM91]).

[1] und natürlich auch alle anderen formalen Methoden, die auf diesen Konzepten basieren

[2] eine solche formale Spezifikation wurde schon für viele Sprachen erstellt (z.B. PL/I, Modula-2 oder Ada); die Entwicklung der Methode VDM erfolgte sogar vor dem Hintergrund der formalen Spezifikation von PL/I

1.2 Die VDM Class Library

I'd rather write programs to write programs than write programs.
— Dick Sites, Digital Equipment Corporation

Ein wichtiger Teil der Spezifikationssprache von VDM — früher Meta-IV, heute VDM-SL genannt — sind *Typgeneratoren*, die zur Modellierung der Daten in der Spezifikation und im Design verwendet werden. Unter einem solchen Typgenerator, der auch als generischer abstrakter Datentyp bezeichnet wird, versteht man einen abstrakten Datentyp (ADT), dessen Elemente beliebig gewählt werden können; durch Festlegung der Elementtypen kann also ein bestimmter ADT generiert werden. VDM-SL stellt zur Modellierung der Daten folgende Typgeneratoren zur Verfügung:

> Mengen, Tuple, Abbildungen (maps), Records (trees), Unions und Optionale

Bei der Codierung eines mit VDM entwickelten Programmes stellen diese Typgeneratoren jedoch häufig ein Problem dar, da sie in den üblichen Programmiersprachen (C, Pascal, C++, Ada etc.) keine Entsprechung haben[3]. Ein möglicher Ansatz zur Lösung dieses Problems ist die *VDM Class Library*, die seit 1990 an der FH Wedel als Hilfsmittel zur Verfügung steht.

Die VDM Class Library besteht aus zwei Teilen: einer Datenbank mit Codefragmenten und dem *VDM-Compiler*. Der VDM-Compiler ist ein Codegenerator, der aus der Beschreibung abstrakter Datentypen in der *Domain Specification Language* (DSL) Programmodule zur Implementierung dieser ADT in einer Programmiersprache erzeugt. Zur Erzeugung dieser Programmodule verwendet der VDM-Compiler die Codefragmente aus der Datenbank der VDM Class Library.

Die DSL entspricht im Prinzip der Notation, die zur Spezifikation der Daten in VDM verwendet wird; insbesondere werden zur Modellierung der Daten die gleichen Typgeneratoren zur Verfügung gestellt. Das Erstellen einer entsprechenden DSL-Beschreibung aus einer VDM-Typgleichung ist somit sehr einfach möglich.

Erweitert wurde die DSL jedoch um eine Implementationsbeschreibung. Hier werden zum einen die vom Anwender benötigten Operationen ausgewählt; dies verringert die Größe der erzeugten Programmodule, da so nur für die vom Anwender tatsächlich benötigten Operationen Code erzeugt werden muß. Die zur Auswahl stehenen Operationen entsprechend weitgehend den in VDM-SL für den jeweiligen Typgenerator zur Verfügung stehenden Operationen.

[3]der funktionelle Teil einer VDM-Spezifikation bzw. eines Designs läßt sich hingegen meistens sehr einfach umsetzen — vorausgesetzt es stehen die benötigten Datentypen zur Verfügung

Zum anderen wird in der Implementationsbeschreibung die konkrete Implementation des erzeugten ADT ausgewählt. In der VDM Class Library stehen für jeden Typgenerator mehrere Implementationen zur Auswahl, die sich hinsichtlich ihres Laufzeitverhaltens und ihres Speicherbedarfs unterscheiden. Der Anwender kann also eine seinen Anforderungen entsprechende Implementation wählen. Besonders hervorzuheben ist die Möglichkeit, eine Implementation später ohne Änderungen des restlichen Programmes auszutauschen, z.B. wenn die VDM Class Library um eine effizientere Implementation für einen Typgenerator erweitert wird !

Der VDM-Compiler unterstützt zur Zeit die Sprachen C, Pascal, Modula-2 und C++ (hier stehen jedoch nicht alle Operationen zur Verfügung).

Ausführlicher soll hier nicht auf die Arbeit mit der VDM Class Library oder deren Realisierung eingegangen werden. Diese Informationen sind in der entsprechenden Literatur schon ausführlich vorhanden (siehe [SH90], [SH91], [JM91] und [KS93]).

1.3 Aufgabenstellung

Bei der täglichen Arbeit mit der VDM Class Library hat sich gezeigt, daß zwei Eigenschaften der erzeugten ADT die Anwendung in eigenen Programmen erschweren und eine Quelle häufiger Fehler darstellen:

- die Verwendung von Referenzsemantik
- die manuelle Speicherverwaltung

Eine ausführliche Beschreibung der auftretenden Probleme, ihrer Ursachen sowie alternativer Vorgehensweisen folgt in den Abschnitten 2 und 3.

Im Rahmen dieser Diplomarbeit soll die VDM Class Library deshalb um eine automatische Speicherverwaltung, die die Probleme der manuellen Speicherverwaltung vermeidet, sowie die Verwendung von Wertsemantik anstelle der jetzt verwendeten Referenzsemantik erweitert werden.

Hierbei sind folgende Rahmenbedingungen zu beachten:

- die neu zu implementierende Speicherverwaltung und die Wertsemantik müssen nur für die Zielsprache C++ realisiert werden
- die Erweiterungen sollen — soweit möglich — für alle vorhandenen Typgeneratoren und Implementationen der VDM Class Library nutzbar sein
- die Erweiterungen sollen optional wählbar sein
- notwendige Änderungen am VDM-Compiler und den bestehenden Modulen der Datenbank sind auf ein Minimum zu beschränken

1.4 Aufbau der Diplomarbeit

Die weitere Diplomarbeit hat folgenden Aufbau:

1. es werden die theoretischen Grundlagen der Wert- und Referenzsemantik sowie verschiedener Ansätze zur Speicherverwaltung beschrieben (Abschnitt 2 und 3).
2. das grundlegende Konzept zur Realisierung der neuen Speicherverwaltung und der Wertsemantik wird beschrieben und die Wahl begründet; die Vorgehensweise zur Integration der Erweiterungen in das bestehende System wird erläutert (Abschnitt 4).
3. es folgt eine Spezifikation der notwendigen Erweiterungen der Datenbank sowie des eigentlichen Compilers unter Verwendung von VDM (Abschnitt 5)
4. die eigentliche Implementierung der Erweiterungen wird in Abschitt 6 beschrieben
5. in Abschnitt 7 werden die durchgeführten Tests der implementierten Erweiterungen vorgestellt; hierbei werden auch deren Laufzeiteigenschaften ermittelt
6. den Abschluß der eigentlichen Diplomarbeit bildet eine kurze zusammenfassende Bewertung der durchgeführten Erweiterungen (Abschnitt 8)

2 Wert- und Referenzsemantik

Durch die Wahl der Implementierungssemantik[1] — unter diesem Begriff werden Wert- und Referenzsemantik zusammengefaßt — wird festgelegt, wie ein im Programm verwendeter Name einer Variablen auf den zu dieser Variablen gehörenden Wert abgebildet wird. Diese Unterscheidung ist unabhängig von der tatsächlichen Implementation auf der Maschinenebene.

Bei Verwendung von Wertsemantik erfolgt eine direkte, einstufige Abbildung des Namens auf den Wert. In VDM-Notation kann die Menge der Variablen in diesem Fall folgendermaßen modelliert werden:

$$Variables = Name \xrightarrow[\mathrm{m}]{} Value$$

Auf jeden Wert kann zu jedem Zeitpunkt also nur über genau einen Namen zugegriffen werden. Was hier genau ein *Name* oder ein *Value* ist, soll nicht weiter interessieren.

Bei Verwendung von Referenzsemantik erfolgt hingegen eine zweistufige Abbildung. Dem Namen selbst wird nur eine Referenz zugeordnet und erst über diese Referenz erhält man dann Zugriff auf den eigentlichen Wert. In VDM-Notation kann dies folgendermaßen modelliert werden:

$$\begin{aligned} Variables &= Name \xrightarrow[\mathrm{m}]{} Reference \\ References &= Reference \xrightarrow[\mathrm{m}]{} Value \end{aligned}$$

Dieses für die Referenzsemantik verwendete Modell entspricht im Prinzip dem Modell des Arbeitsspeichers eines konventionellen Computers auf der Maschinenebene: alle Werte werden in Speicherzellen gespeichert, auf diese Speicherzellen wird über Adressen (entspricht den Referenzen) zugegriffen, Adressen kann ein symbolischer Name zugeordnet werden.

In den meisten Programmiersprachen finden sowohl Referenz- als auch Wertsemantik Verwendung. Die Auswahl der Implementierungssemantik ist in der Regel vom Datentyp abhängig[2]. Teilweise kann der Anwender die Referenzsemantik aber auch explizit auswählen; z.B. durch die in C++ eingeführten Referenzen oder in Pascal durch Verwendung des Schlüsselwortes `VAR` in der Definition der formalen Parameter einer Prozedur oder Funktion.

[1] siehe [SB93, S. 78]

[2] In der Sprache C wird z.B. für den Typ `int` Wertsemantik, für Arrays hingegen Referenzsemantik verwendet; in Eiffel wird für alle einfachen Typen Wertsemantik, für Klassentypen jedoch Referenzsemantik verwendet

Für die vom VDM-Compiler erzeugten abstrakten Datentypen wird bis auf wenige Ausnahmen die Referenzsemantik verwendet und zwar unabhängig von der gewählten Zielsprache.

2.1 Nachteile der Referenzsemantik

Die Verwendung von Referenzsemantik hat einen entscheidenden Nachteil gegenüber der Wertsemantik: es kann eine *dynamische Mehrfachbenennung* bzw. *Aliasbildung* auftreten[3]. Hierunter versteht man, daß über mehr als einen Namen auf einen Wert zugegriffen werden kann. In dem oben für die Referenzsemantik beschriebenen Modell der Variablen würde dies so aussehen:

$$\begin{aligned} variables &= [name_1 \mapsto ref, name_2 \mapsto ref] \\ references &= [ref \mapsto value] \end{aligned}$$

Auf den Wert $value$ kann hier sowohl über den Namen $name_1$ als auch über den Namen $name_2$ zugegriffen werden. Diese dynamische Mehrfachbenennung entsteht in der Regel durch Zuweisung oder durch Parameterübergabe an Prozeduren und Funktionen, da im Gegensatz zur Wertsemantik nicht der eigentliche Wert sondern nur die Referenz übergeben bzw. zugewiesen wird[4].

Durch die dynamische Mehrfachbenennung oder Aliasbildung tritt nun folgendes Problem auf: eine Operation kann den einem Namen $name_1$ zugeordneten Wert ändern, obwohl dieser Name im Kontext der Operation nicht auftritt, sondern nur ein anderer Name $name_2$, dem aber derselbe Wert zugeordnet ist !

Einige Beispiele sollen dieses Problem verdeutlichen. Der Typ `BAG`, definiert durch die DSL-Spezifikation `Bag = Nat0 -bag`, sei hierbei vom VDM-Compiler erzeugt.

```
BAG x = BAG(42);
BAG y;
y = x;
x.sub1d(42);
cout << x << y;
```

In diesem Fall erfolgt die Aliasbildung durch Zuweisung; auf den Bag, der die Zahl 42 enthält, kann nun sowohl über den Namen x als auch über den Namen y zugegriffen werden. Die für x aufgerufene Operation `sub1d` verändert somit auch den Wert von y.

[3]dynamisch, da die Aliasbildung zur Laufzeit erfolgt; es gibt auch eine statische Aliasbildung, bei der die Aliase zur Übersetzungszeit erzeugt werden (ein Beispiel ist die berüchtigte FORTRAN-Anweisung `EQUIVALENCE`, die festlegt, daß zwei Variablen den gleichen Speicherbereich verwenden)

[4]je nach Sprache mögen weitere Möglichkeiten zur Erzeugung eines Alias vorhanden sein

```
void proc(BAG b)
{
  if (!b.is_empty())
     b.sub1d(b.sel());
}
...
BAG x = BAG(42);
proc(x);
cout << x;
```

Hier erfolgt die Aliasbildung durch Parameterübergabe. In der Prozedur `proc` wird über den Namen b auf denselben Bag zugegriffen, der in der aufrufenden Umgebung über den Namen x erreichbar ist. Die lokale Änderung von b hat somit auch Einfluß auf x; man spricht hier auch von einem *Seiteneffekt*. Diese Eigenschaft wird häufig ausgenutzt um Funktionen, die mehr als einen Rückgabewert haben, zu realisieren (z.B. in der Operation `is_in_dom` für Maps).

```
void swap(int&x,int&y)
{
   cout << x << y;
   x = x + y;
   y = x - y;
   x = x - y;
   cout << x << y;
}
...
int a = 5, b = 3;
swap(a,b);
```

Die Prozedur `swap` soll zwei Zahlen mit einem etwas ungewöhnlichen Algorithmus vertauschen und ausgeben. Funktioniert sie korrekt ? Die Funktion ist tatsächlich *im allgemeinen* korrekt ! Eine Ausnahme bildet der Fall, in dem für beide aktuelle Parameter dieselbe Variable übergeben wird (`swap(a,a)`). Aufgrund der Aliasbildung wird durch die Namen x und y dann derselbe Wert bezeichnet; der aktuelle Parameter wird auf 0 gesetzt.

Diese Beispiele zeigen, daß der Programmierer bei Verwendung von Referenzsemantik aufgrund der möglichen Aliasbildung immer mehr Informationen besitzen muß, als eigentlich für ihn wichtig sein sollte:

- bevor der Programmierer eine Operation ausführt, die einen Wert verändert, muß er wissen, ob aufgrund der Aliasbildung noch über andere Namen auf diesen Wert zugegriffen werden kann; ist dies der Fall, so müssen evtl. erst weitere Aktionen (s.u.) durchgeführt werden, um eine unerwünschte Auswirkung dieser Veränderung auf andere Namen zu verhindern (siehe 1. Beispiel).
- beim Aufruf einer Prozedur oder Funktion ist es nicht ausreichend zu wissen, *was* gemacht wird, sondern es muß auch bekannt sein, *wie* dies gemacht wird (siehe 2. und 3. Beispiel)

Die Beschaffung und ständige Bereithaltung dieser zusätzlichen Informationen macht das Arbeiten mit Referenzsemantik zu einer kniffeligen und fehlerträchtigen Angelegenheit.

Warum wird die Referenzsemantik trotz dieses schwerwiegenden Nachteils nun überhaupt in so vielen Programmiersprachen unterstützt ? Dies hat einen sehr einfachen Grund: die Ausführung eines Programmes auf einem heute typisch verwendeten Rechner mit Von Neumann-Architektur ist effizienter.

Wie weiter oben bereits beschrieben wurde, entspricht das für die Referenzsemantik verwendete Modell ja weitgehend dem Modell des Speichers in der Von Neumann-Architektur. Die Referenzen entsprechen den Adressen der eigentlichen Werte im Speicher. Bei Verwendung von Referenzsemantik muß also bei einer Zuweisung oder Parameterübergabe nur diese Adresse im System bewegt werden. Wird hingegen die Wertsemantik verwendet, so muß immer der ganze Speicherbereich, in dem der Wert gespeichert ist, kopiert werden. Dies ist natürlich viel zeitaufwendiger.

Um die oben beschriebenen Probleme zu vermeiden, werden solche Kopieroperationen, die die eigentlichen Werte und nicht nur die Referenzen kopieren, oder zumindest Hilfsmittel zu ihrer Implementation von vielen Systemen, die Referenzsemantik verwenden, bereitgestellt. Der VDM-Compiler stellt z.B. für alle von ihm erzeugten Typen die Kopieroperationen `copy()` und `copy1()` zur Verfügung. In C++ besteht die Möglichkeit eigene Routinen zur Durchführung der Parameterübergabe/Initialisierung und der Zuweisung für Klassentypen zu definieren. Durch „geschickten“ Einsatz dieser Operationen ist ein problemfreie Verwendung der Referenzsemantik im Prinzip möglich; daß dieser „geschickte“ Einsatz nicht einfach ist und auch nicht immer korrekt gelingt, kann wohl jeder Programmierer aus eigener Erfahrung bestätigen.

Der Programmierer bzw. Sprachentwickler steht hier also vor einer schwierigen Entscheidung: der theoretisch überlegenen Wertsemantik, die das Programmieren vereinfacht und zu einer Verminderung von Fehlern führt, steht die problematische Referenzsemantik gegenüber, die eine weit effizientere Implementierung auf heutigen Rechnern erlaubt. Die Entscheidung fällt heute in der Regel noch zu Gunsten der Effizienz.

3 Speicherverwaltung

In diesem Abschnitt sollen die verschiedenen Möglichkeiten zur Freigabe von dynamisch allokiertem Speicher diskutiert werden. Dieses ist für die hier bearbeitete Aufgabe wichtig, da — bis auf wenige Ausnahmen — alle von der VDM Class Library bereitgestellten Typgeneratoren mit Hilfe von dynamischen Datenstrukturen implementiert werden.

Bevor jedoch genauer auf die Speicherverwaltung eingegangen wird, möchte ich zwei im weiteren verwendete Begriffe erläutern (siehe auch [Me90, Kapitel 16.1.1]):

- **Objekt:** unter einen Objekt soll ein zur Laufzeit eines Programmes existierendes Element verstanden werden; ein dynamisches Objekt wird zur Laufzeit durch einen expliziten Aufruf einer Operation, z.B. `new` in C++, erzeugt

- **Größe:** unter einer Größe soll ein Element eines Programmtextes verstanden werden, z.B. eine Variable in einem C-Programm; einer Größe ist zu einem bestimmten Zeitpunkt immer genau ein Objekt zugeordnet, dieses Objekt kann sich während der Laufzeit des Programmes jedoch ändern (z.B. bei einer lokalen Variable in einer rekursiven Funktion)

3.1 Die Aufgaben

Bei der Freigabe des von dynamischen Objekten belegten Speicherplatzes sind zwei zentrale Aufgaben zu lösen:

- **Entdeckung:** unter Entdeckung sei die Bestimmung des Zeitpunktes, an dem ein bestimmtes Objekt gelöscht[1] werden darf, verstanden

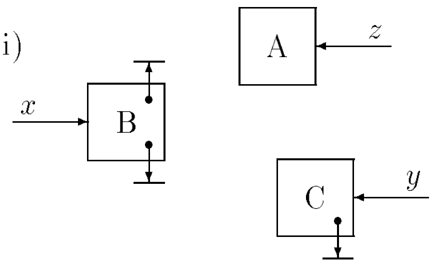
- **Rückgewinnung:** unter Rückgewinnung sei die eigentliche Durchführung der Freigabe des von einem oder mehreren Objekten belegten Speicherplatzes verstanden

Diese zwei Aufgaben erscheinen auf dem ersten Blick recht einfach; sie könnnen aber, insbesondere durch die bei Verwendung von Referenzsemantik möglicherweise auftretende dynamische Mehrfachbenennung, sehr kompliziert werden. Die Schwierigkeiten sollen an mehreren einfachen Beispielen aufgezeigt werden: der Verweis x auf das dynamische Objekt B wird nicht mehr benötigt; es muß nun

[1] Löschen soll im weiteren abkürzend für Freigeben des von einem Objekt belegten Speicherplatzes verwendet werden

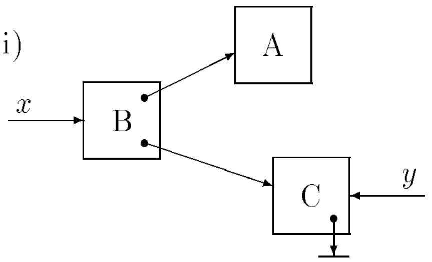
entschieden werden, ob das Objekt B und evtl. weitere Objekte gelöscht werden können.

i)



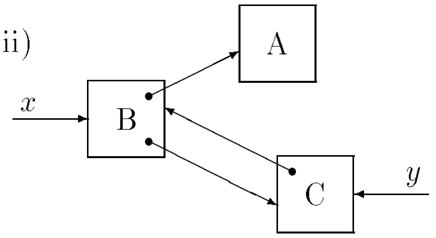
Dies ist der einfachste Fall. Wird der Verweis x nicht mehr benötigt, so kann das Objekt B gelöscht werden. Weitere Objekte sind nicht betroffen.

ii)



Auch in diesem Fall darf das Objekt B gelöscht werden. B enthält jetzt jedoch noch Verweise auf A und C; diese Objekte können also evtl. auch noch gelöscht werden. In diesem Fall darf jedoch nur A gelöscht werden, da auf C noch ein weiterer Verweis (y) vorhanden ist.

iii)



In diesem dritten Fall darf schließlich keines der Objekte gelöscht werden, denn über den noch vorhandenen Verweis y ist ein Zugriff auf alle drei Objekte möglich !

In diesen Beispielen wird die besondere Schwierigkeit deutlich: man kann nicht allein anhand des zu löschenden Objektes entscheiden, ob dieses Objekt auch tatsächlich gelöscht werden darf. Dies ist immer auch von allen anderen Objekten abhängig. In Fall ii und iii hat z.B. B beidesmal den selben Inhalt (jeweils einen Verweis auf A und C), jedoch nur in Fall ii darf B gelöscht werden. Außerdem müssen evtl. weitere Objekte gelöscht werden, so z.B. Objekt A in Fall ii.

In der Praxis findet man die folgenden vier Lösungsansätze für die beschriebenen Aufgaben; das Unterscheidungskriterium ist die Zuordnung der Zuständigkeit für die Aufgaben (siehe [Me90, Kap. 16]):

1. „Gleichgültiger“ Ansatz
2. manuelle Speicherverwaltung
3. Selbstverwaltungsansatz
4. Garbage Collection

3.2 „Gleichgültiger“ Ansatz

Der „Gleichgültige“ Ansatz stellt die einfachste Lösung dar, denn hier ist *keiner* für die oben beschriebenen Aufgaben zuständig. Der für dynamische Objekte benötigte Speicherplatz wird bei Bedarf allokiert, er wird jedoch *nie* wieder freigegeben !

Dieser Ansatz stellt natürlich keine allgemein anwendbare Lösung dar, denn es wird sehr viel Speicher verschwendet. Sinnvoll einsetzbar ist dieser Ansatz jedoch, wenn

- ein Programm nicht sehr viele dynamische Objekte erzeugt, so daß trotz Speicherverschwendung ausreichend Speicher vorhanden ist
- ein Programm zum größten Teil dynamische Objekte erzeugt, die bis zum Ende der Programmlaufzeit benötigt werden; eine aufwendige Strategie zur Speicherfreigabe ist in diesem Fall überflüssig

Kann dieser Ansatz verwendet werden, so bietet er einen großen Vorteil: die Allokation von Speicher für dynamische Objekte wird sehr einfach[2] und somit sehr schnell !

[2] der freie Speicher liegt immer als zusammenhängender Bereich vor; für die ganze Speicherverwaltung wird deshalb nur ein Zeiger benötigt, der auf den Anfang des freien Speichers zeigt; bei einer Allokation muß nur die Größe des allokierten Blockes zu diesem Zeiger addiert werden

3.3 Manuelle Speicherverwaltung

Bei der manuellen Speicherverwaltung wird die Aufgabe der Erkennung und Rückgewinnung völlig dem Anwender überlassen. Vom System/der Sprache wird lediglich eine Operation zur Freigabe des von *einem* Objekt belegten Speichers zur Verfügung gestellt (z.B. `free` in C, `delete` in C++). Diese sehr einfache Lösung — für den Implementator der Sprache, nicht für den Anwender — wird in den meisten imperativen Sprachen, wie z.B. C, Pascal und Modula-2, aber auch in C++ realisiert.

Diese Lösung hat jedoch entscheidende Nachteile:

- **Dangling Pointer:** wird vom Programmierer fehlerhafterweise ein dynamisches Objekt gelöscht, auf welches noch weitere Verweise vorhanden sind, entstehen die sogenannten *dangling pointer* (hängende Zeiger). Diese ins „nichts" zeigenden Verweise führen zu fatalen Programmfehlern, wenn der Programmierer über sie auf das nicht mehr vorhandene Objekt zugreift. Im Fall iii des vorherigen Beispiels entsteht ein solcher dangling pointer aus dem im Objekt C vorhandenen Verweis auf B, wenn fehlerhafterweise B gelöscht wird.

- **rekursive Freigabe:** die von der Sprache bereitgestellte Freigabeoperation löscht immer nur ein dynamisches Objekt. Ein Objekt kann aber selbst wieder Verweise auf andere Objekte enthalten, die eventuell auch gelöscht werden müssen. Diese Objekte können nun ebenfalls wieder Verweise enthalten ... Dieses Problem wird auch als das Problem der rekursiven Freigabe bezeichnet. Für jeden Typ eines dynamischen Objektes, das selbst wieder Verweise enthält, muß der Anwender also eine eigene Freigabegabeoperation schreiben; die von der Sprache bereitgestellte Freigabeoperation ist hier nicht mehr ausreichend. Dies trifft z.B. auf das Objekt B im vorherigen Beispiel zu, daß selbst wieder zwei Verweise enthält.

- **„Speicherleichen":** vergißt der Anwender ein dynamisches Objekt zu löschen, bevor alle Verweise auf dieses Objekt entfernt werden, so entsteht ein Objekt im Speicher, auf das nicht mehr zugegriffen und somit auch nicht mehr gelöscht werden kann; es ist eine „Speicherleiche" entstanden! Dies soll an einem Beispiel verdeutlicht werden:

```
main()
{
  { char * array = new char[1000]; }
   ...hier kein Zugriff mehr auf das Array !...
}
```

Entsteht eine solche „Speicherleiche“ in einer Funktion, die in einem langlaufenden Programm sehr häufig aufgerufen wird, kann durch solche Objekte schließlich der gesamte Speicher verbraucht werden; das Programm muß wegen Speichermangel abgebrochen werden. Während eines Tests bleibt dieser Fehler häufig unentdeckt, da Auswirkungen erst nach langer Laufzeit des Programmes auftreten. In Fall ii des vorherigen Beispiels entsteht eine solche „Speicherleiche“ aus dem Objekt A, wenn es beim Löschen von B nicht ebenfalls gelöscht wird.

Durch diese Nachteile der manuellen Speicherverwaltung erhöht sich die Komplexität eines Programmes erheblich. Der Programmierer muß nicht nur das eigentliche Anwendungsproblem lösen, sondern sich auch mit den Problemen der Speicherverwaltung beschäftigen; er wird zum „Müllsammler“ ([Me90, S. 387]). Eine erhöhte Komplexität hat wiederum negative Einflüsse auf die Sicherheit eines Programmes. Denn je komplexer ein Programm ist, desto höher ist die Wahrscheinlichkeit von Fehlern. Aus diesen Gründen wird diese Lösung in [Me90, S. 387ff.] als „unannehmbar“ oder zumindest als „recht fragwürdig“ bezeichnet !

Da in allen von der VDM Class Library unterstützen Sprachen eben dieser Lösungsansatz gewählt wurde, ist die Speicherverwaltung für die vom VDM-Compiler erzeugten abstrakten Datentypen ebenfalls so konzipiert, daß der Anwender selbst für die Freigabe des von diesen belegten Speicherplatzes verantwortlich ist. Das Entwickeln eigener Freigabeoperationen ist jedoch nicht notwendig; alle Typgeneratoren stellen die Operationen `del()` und `del1()` für die Freigabe des von einem ADT belegten Speicherplatzes zur Verfügung. Zusätzlich sind Kombinatoroperationen vorhanden, die die Operation `del()` implizit zum Löschen von Elementen aufrufen; diese sind durch ein an den Namen angehängtes '`d`' zu erkennen.

Da das gleiche Konzept verwendet wird, können auch die gleichen Fehler auftreten. Dies soll anhand einiger einfacher Beispiele mit den ADT

```
Tuple = Set *
Set = Nat0 -set
```

verdeutlicht werden:

`SET   s = SET(empty);`
`TUPLE t = TUPLE(s);`
 	
`s.del();`
	Durch die Operation del wird die Menge, auf die s verweist, gelöscht. Da diese Menge aber auch Element von t ist, entsteht in t ein dangling pointer. Weitere Zugriffe auf t führen zu Fehlern.

```
SET   s = SET(empty);
TUPLE t = TUPLE(s);

t.tld();
```

Durch die tld-Operation wird das im Tuple gespeicherte Set und damit auch das Set, auf das s verweist, gelöscht. s enthält jetzt einen dangling pointer.

```
SET   s = SET(empty);
TUPLE t = TUPLE(s);

s = SET(100);
t.tl();
```

Durch die tl-Operation wird das im Tuple gespeicherte Set aus dem Tuple entfernt aber nicht gelöscht. Da s inzwischen auf ein neues Set verweist, exitiert kein weiterer Verweis auf das entfernte Set. Es ist eine „Speicherleiche" entstanden !

Bei der Entwicklung der neuen Speicherverwaltung für die VDM Class Library wurden diese beiden Fehlerarten auch tatsächlich in der Datenbank des Compilers sowie in einem Beispielprogramm gefunden; besonders oft traten dabei die „Speicherleichen" auf. Dies zeigt, daß die Probleme der manuellen Speicherverwaltung nicht nur theoretischer Natur sind, sondern auch in der praktischen Anwendung häufig auftreten !

3.4 Selbstverwaltungsansatz

Der Selbstverwaltungsansatz stellt keine allgemeingültige Lösung der vorgestellten Aufgaben dar. Er kann nur dann verwendet werden, wenn allgemein wiederverwendbare abstrakte Datentypen in Form von Bibliotheken zur Verfügung gestellt werden sollen und die Implementationssprache das Konzept der manuellen Speicherverwaltung verwendet. Solche allgemein wiederverwendbaren ADT sind z.B. die Typgeneratoren der VDM Class Library.

Der Selbstverwaltungsansatz unterscheidet sich weniger in technischen als vielmehr in organisatorischen Aspekten von der manuellen Speicherverwaltung. Neben dem Anwendungsprogrammierer, der ein Programm zur Lösung eines bestimmten Anwendungsproblemes entwickelt, gibt es jetzt einen weiteren Programmierer[3], der abstrakte Datentypen entwickelt und als allgemein verwendbare Bibliotheken bereitstellt. Dieser Programmierer ist nun für die Erkennung und Rückgewinnung aller *in der Bibliothek verwendeten* dynamischen Objekte zuständig.

Der Programmierer der Bibliotheken steht natürlich vor den gleichen Problemen wie der Anwendungsprogrammierer bei der manuellen Speicherverwaltung; zur

[3]in der Praxis kann dieser mit dem Anwendungsprogrammierer übereinstimmen, aufgrund der unterschiedlichen Aufgaben ist eine Unterscheidung trotzdem sinnvoll

Realisierung der Bibliothek werden ihm von der Sprache ja die gleichen Hilfsmittel zur Verfügung gestellt. Da er jedoch die volle Kontrolle über die Benutzung des ADT hat, kann er in der Regel leichter eine allgemeine und effiziente Lösung zur Erkennung und Rückgewinnung der benutzten dynamischen Objekte entwickeln. Ein Ausgleich für den hierbei notwendigen höheren Entwicklungsaufwand ist durch die stark vereinfachte Wiederverwendung der erstellten ADT gegeben: der Anwendungsprogrammierer kann die ADT in vielen Programmen verwenden, ohne daß er sich mit Problemen der Speicherverwaltung beschäftigen muß !

Obwohl es sich nur um eine partielle Lösung[4] handelt, stellt der Selbstverwaltungsansatz aus dem oben genannten Grund der stark vereinfachten Wiederverwendung eine zu bevorzugende Vorgehensweise gegenüber der manuellen Speicherverwaltung dar.

3.5 Garbage Collection

Erfolgt die Bestimmung des Zeitpunkts sowie die Durchführung der Freigabe des von einem dynamischen Objekt belegten Speicherplatzes völlig automatisch und ohne Eingriffe des Programmierers durch das Laufzeitsystem, so spricht man von einem System mit *Garbage Collection*[5] . Der für die Erkennung und Rückgewinnung zuständige Teil des Laufzeitsystems wird als *Garbage Collector* bezeichnet.

Im Gegensatz zu den schon vorgestellten Verfahren, die zur Realisierung der Speicherverwaltung nur von der Sprache bereitgestellte Operationen verwenden, sind für die Implementierung eines Garbage Collectors immer auch Arbeiten auf der Sprachimplementierungsebene, d.h. am Compiler oder Interpreter, notwendig.

Durch die Verwendung eines Garbage Collectors lassen sich alle in Abschnitt 3.3 beschriebenen Nachteile der manuellen Speicherverwaltung vermeiden. Dies führt zu einer Verringerung der Komplexität und somit zu einer Erhöhung der Sicherheit ! Trotz dieses wichtigen Vorteils werden Garbage Collectoren aber noch vorwiegend in eher „exotischen“ Sprachen wie z.B. Lisp, ML oder Smalltalk eingesetzt. Die Verwendung der manuellen Speicherverwaltung in anderen Sprachen wird in der Regel durch folgende Nachteile der Garbage Collection begründet:

- Schwierigkeiten bei der Implementierung
- Verlängerung der Laufzeit des Programmes

[4] sind benötigte Datentypen nicht in einer bereitgestellten Bibliothek vorhanden, so muß der Anwendungsprogrammierer diese unter Verwendung der manuellen Speicherverwaltung selbst realisieren

[5] dt. Müllsammlung

Sind die Schwierigkeiten der Implementierung eines Garbage Collector für den Anwender einer Sprache auch irrelevant, so kann die Laufzeitverlängerung für viele Anwendungen nicht akzeptiert werden. Insbesondere die frühen Garbage Collectoren unterbrachen die laufende Anwendung oft für mehrere Sekunden[6], um die nicht mehr benötigten dynamischen Objekte zu finden und zu löschen. Dies schließt eine Verwendung für Echtzeitanwendungen aus; aber auch in vielen interaktiven Anwendungen ist eine solche Unterbrechung für den Anwender sehr unangenehm. Der durch den Einsatz eines Garbage Collectors gewonnenen Sicherheit und der Verringerung der Komplexität steht also ein Verlust an Effizienz gegenüber. Dieser Widerspruch wird in [St92, S. 510] folgendermaßen beschrieben:

> Man sagt, daß Lisp-Programmierer wissen, daß die Speicherverwaltung zu wichtig ist, um sie dem Anwender zu überlassen, während C++-Programmierer wissen, daß die Speicherverwaltung zu wichtig ist, um sie dem System zu überlassen.

Durch Fortschritte in der Technik der Garbage Collectoren kann der notwendige Zusatzaufwand heute jedoch auf ein solches Maß beschränkt werden, daß die Vorteile den noch vorhandenen Zusatzaufwand rechtfertigen. Aus diesem Grund wird die automatische Speicherverwaltung durch Garbage Collectoren immer mehr als notwendige Eigenschaft einer modernen Programmiersprache angesehen ([Ap91, S. 89]). Selbst in neu entwickelten Sprachen zur Systemprogrammierung — bei denen Effizienz also sehr wichtig ist — verwendet man heute schon Garbage Collectoren; als Beispiele seien hier Eiffel und Oberon genannt.

Bevor einige Verfahren zur Implementierung eines Garbage Collectors vorgestellt werden, soll der Begriff des *erreichbaren* dynamischen Objektes kurz erläutert werden, da er in diesem Zusammenhang oft verwendet wird.

Ein dynamisches Objekt ist *erreichbar*, wenn man ausgehend von einer lokalen oder globalen Variable direkt durch einen Verweis oder durch eine Folge von Verweisen auf dieses Objekt zugreifen kann. Ist dies nicht möglich, so bezeichnet man dieses Objekt als *nicht erreichbar* (siehe auch Abb. 1; erreichbare Objekte sind durch normale, nicht erreichbare Objekte durch gestrichelte Rechtecke dargestellt). Nicht erreichbare Objekte können nie wieder erreichbar werden ! Der von ihnen belegte Speicherplatz kann deshalb freigegeben und wiederverwendet werden.

3.5.1 Referenzzähler

Die Implementation eines Garbage Collectors mit Hilfe von Referenzzählern ist auf der Sprachimplementierungsebene sehr einfach. Hierbei muß jedes dynamische

[6] diese Unterbrechung kann z.B. bei Verwendung des weitverbreiteten Editors Emacs beobachtet werden, der zum größten Teil in einem Lisp-Dialekt geschrieben wurde

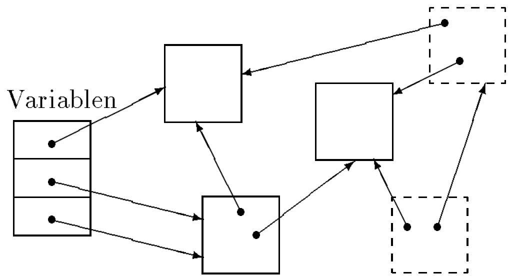


Abbildung 1: zur Definition *erreichbarer* Objekte

Objekt lediglich um einen Zähler erweitert werden, der die Anzahl der Verweise (Referenzen) auf dieses Objekt angibt.

Für ein neu erzeugtes Objekt wird dieser Referenzzähler mit eins initialisiert. Für jeden neu auf dieses Objekt eingerichteten Verweis wird er um eins erhöht; für jeden entfernten Verweis wird er um eins verringert. Erreicht der Referenzzähler hierbei den Wert 0, so ist das entsprechende dynamische Objekt nicht mehr erreichbar und der von dem Objekt belegte Speicherplatz kann wieder freigegeben werden.

In der Praxis werden Referenzzähler jedoch kaum zur Realisierung eines Garbage Collectors verwendet, da sie zwei entscheidende Nachteile haben:

- die Verwaltung des Referenzzählers ist kostenintensiv[7]: bei jeder Operation, die Verweise auf Objekte erzeugt oder entfernt, sind nun zusätzliche arithmetische und Vergleichsoperationen zur Verwaltung des Referenzzählers notwendig. Diese zusätzlich notwendigen Operationen können z.T. aufwendiger als die eigentliche durchzuführende Operation sein (z.B. bei einer Zuweisung). Außerdem wird für jedes Objekt zusätzlicher Speicher für den Referenzzähler benötigt; bei sehr vielen kleinen Objekten kann hierfür ein großer Teil des Speichers verbraucht werden.

- der von zyklischen Strukturen — die dynamischen Objekte enthalten direkt oder indirekt Verweise auf sich selbst — belegte Speicher wird nie wieder freigegeben, da die Referenzzähler in einer solchen Struktur nie den Wert 0 erreichen ! Ein Beispiel hierfür ist in Abb. 2 gegeben. Wird der Verweis x auf das Objekt A entfernt, so sind alle Objekte in der zyklischen Struktur nicht mehr erreichbar. Sie müßten also gelöscht werden. Dies erfolgt jedoch nicht, da immer noch ein Verweis auf jedes dieser Objekte vorhanden ist und somit alle Referenzzähler den Wert 1 haben.

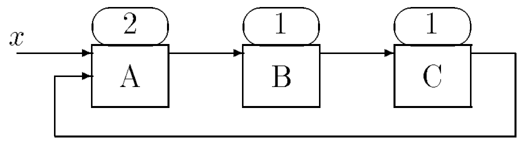


Abbildung 2: zyklische Strukturen werden nie freigegeben

Obwohl Referenzzähler aus den genannten Gründen kaum noch für Garbage Collectoren verwendet werden, können sie in anderen Bereichen sinnvoll eingesetzt werden. So werden sie z.B. häufig im Zusammenhang mit dem Selbstverwaltungsansatz verwendet (siehe [St92, S. 266ff. und 505ff.]).

3.5.2 Mark and Sweep Collection

Die „Mark & Sweep"-Collection[8] ist ein schon sehr lange bekanntes Verfahren zur Realisierung einer Garbage Collection; es wurde bereits 1960 zur Implementierung des ersten Garbage Collectors in einem Lisp-System verwendet.

Bei der „Mark & Sweep"-Collection wird der Garbage Collector erst dann aktiviert, wenn im System ein Speichermangel auftritt, d.h. wenn die Anforderung, ein dynamisches Objekt zu erzeugen, fehlgeschlagen ist. Dies hat den Vorteil (z.B. gegenüber dem Referenzzähler-Verfahren), daß kein Zusatzaufwand durch die automatische Speicherverwaltung entsteht, solange im System ausreichend Speicher vorhanden ist. Weniger vorteilhaft ist die Tatsache, daß die Ausführung des Anwendungsprogrammes für eine recht lange Zeit unterbrochen werden muß, während die Garbage Collection durchgeführt wird.

Die Garbage Collection selbst besteht aus zwei getrennten Phasen:

1. **Mark:** in dieser ersten Phase werden alle erreichbaren dynamischen Objekte markiert. Auch hier muß also jedes Objekt um eine Komponente (für die Markierung) erweitert werden; der benötigte Platz kann jedoch im Gegensatz zum Referenzzählerverfahren vernachlässigt werden, da ein einzelnes Bit ausreichend ist.

2. **Sweep:** in dieser zweiten Phase wird der *gesamte* Speicher sequentiell durchsucht und der von allen nicht markierten Objekten belegte Speicherplatz freigegeben.

[7] in Bezug auf die Rechenzeit und den Speicherplatzbedarf

[8] in [Me90, S. 396] wird Markierungs-Putz-Bereiniger/Bereinigung als deutsche Übersetzung verwendet; da sich dies *etwas* komisch anhört, möchte ich lieber die englischen Begriffe weiterverwenden

Die Markierung der erreichbaren Objekte stellt einen Spezialfall eines allgemeineren Problems dar: die Bestimmung der Menge $\mathcal{M}$ aller direkt oder indirekt von einem Knoten k (oder einer Menge $\mathcal{K}$ von Knoten) aus erreicharen Knoten in einem möglicherweise zyklischen Graphen G. Die aus der Arbeit mit Graphen bekannten Algorithmen können also übernommen werden[9].

Für die Laufzeit der „Mark & Sweep“-Collection gilt im besten Fall $t_{ms} = c_1a + c_2m$. Hierbei ist a die Anzahl der erreichbaren Objekte und m die Gesamtanzahl der Objekte im Speicher (c_i seien Konstanten). Ein Nachteil wird nun deutlich: die Laufzeit des Garbage Collectors ist umso länger, je größer der Arbeitsspeicher ist, da dann die Gesamtzahl m der Objekte im Speicher zunimmt! Die Anzahl der erreichbaren Objekte a ist hingegen unabhängig von der Speichergröße; ihre Anzahl ist nur von dem verwendeten Anwenderprogramm und den aktuellen Eingabedaten abhängig.

3.5.3 Copying Collection

Bei Einsatz der Copying Collection (ca. 1969) wird der für dynamische Objekte vorgesehene Arbeitspeicher in zwei gleich große Bereiche aufgeteilt:

- *from-space*: in diesem Bereich kann das Anwenderprogramm Speicherplatz für dynamische Objekte allokieren
- *to-space*: dieser Bereich ist leer, das Anwenderprogramm hat auf diesen Bereich keinen Zugriff.

Die Garbage Collection wird gestartet, wenn der Speicherplatz im from-space vollständig belegt ist. Entsprechend dem „Mark & Sweep“-Verfahren entsteht also kein Zusatzaufwand durch die automatische Speicherverwaltung solange ausreichend Speicherplatz zur Verfügung steht.

Der Garbage Collector kopiert nun alle erreichbaren Objekte aus dem from-space in den to-space. Beim Kopieren der Objekte aus dem from-space in den to-space muß man jedoch aufpassen: auf Objekte, die vom Garbage Collector kopiert werden, können noch viele Verweise in anderen Objekten oder Variablen existieren; nachdem das Objekt kopiert wurde, sind alle diese Verweise nicht mehr korrekt ! Sie müssen also angepaßt werden.

Da mit hoher Wahrscheinlichkeit ein Teil des Speichers im from-space von nicht erreichbaren Objekten belegt wird, ist der to-space durch die Kopien nicht vollständig belegt. Die Bedeutung der beiden Bereiche ändert sich nun: aus dem

[9] die Knoten entsprechen hierbei den dynamischen Objekten bzw. Variablen, die Kanten entsprechen den Verweisen

from-space wird der neue to-space und aus dem to-space wird der neue from-space. Im from-space ist jetzt also wieder freier Speicher vorhanden und das Anwendungsprogramm kann fortgesetzt werden.

Für die Laufzeit eines Copying Collectors gilt $t_{cc} = c_3 * a$, wobei a die Anzahl der erreichbaren Objekte im from-space ist. Die unangenehme Abhängigkeit der Laufzeit von der Speichergröße entfällt hier also !

Ein weiterer Vorteil der Copying Collection ist die durch das Kopieren hervorgerufene Kompaktierung der Objekte im Speicher. Dies führt dazu, daß der gesamte freie Speicher immer als ein zusammenhängender Bereich vorliegt. Die Allokation von dynamischen Objekten wird hierdurch stark vereinfacht und beschleunigt, insbesondere bei Objekten unterschiedlicher Größe (siehe [Kn73, S. 439ff.]). Aufgrund dieser beiden Vorteile wird dieses Verfahren heute vorwiegend eingesetzt.

3.5.4 Incremental/Concurrent Collection

Auch wenn der Zusatzaufwand für die Garbage Collection in Bezug auf die Gesamtlaufzeit des Programmes relativ klein ist, kann die durch den Garbage Collector erzwungene Pause des Anwenderprogrammes zu lang sein. Dies trifft insbesondere auf Echtzeitsysteme, aber auch auf interaktive Anwendungen zu.

Für solche Anwendungen könnte es dann sinnvoll sein, die Garbage Collection nicht in einem großen Schritt sondern in vielen kleinen Schritten durchzuführen. Ein solches Vorgehen wird dann als Incremental Collection oder Concurrent Collection bezeichnet; hierbei handelt es sich immer um eine Erweiterung einer „Mark & Sweep“ oder Copying Collection.

Durch diese Vorgehensweise nimmt die Dauer der einzelnen Unterbrechungen des Anwenderprogrammes stark ab; die Häufigkeit der Unterbrechungen steigt dafür aber auch stark an. Die Gesamtlaufzeit des Programmes verringert sich durch diese Vorgehensweise also nicht.

Besonders effektiv können diese Verfahren jedoch auf Mehrprozessorsystemen implementiert werden, denn dort kann das Anwenderprogramm teilweise parallel zum Garbage Collector weiterlaufen; dies führt dann auch zu einer Verringerung der Gesamtlaufzeit ([De91] beschreibt einen auf diesem Prinzip basierenden Garbage Collector).

4 Konzept der neuen Speicherverwaltung und Wertsemantik

"Don't ask me how it works, or i'll start to whimper."
— Arthur Dent

Dieses Kapitel beschreibt das grundlegende Konzept der zu realisierenden neuen automatischen Speicherverwaltung und Wertsemantik sowie deren Integration in die bereits existierende VDM Class Library. Soweit dies sinnvoll ist, werden nicht nur das tatsächlich verwendete Konzept sondern auch mögliche Alternativen vorgestellt und die getroffene Auswahl begründet.

4.1 Wertsemantik

Wie schon vorher beschrieben, wird in der aktuellen[1] Version der VDM Class Library für die meisten ADT aufgrund ihrer Implementierung durch dynamische Datenstrukturen die Referenzsemantik verwendet.

Trotz der Implementation durch dynamische Datenstrukturen ist es relativ einfach möglich eine Wertsemantik zu realisieren. Es muß lediglich überall dort, wo ein Alias entstehen würde, eine Kopieroperation aufgerufen werden, die den Wert und nicht nur die Referenz auf den Wert kopiert; eine solche Operation wird vom VDM-Compiler für jeden erzeugten ADT zur Verfügung gestellt: `copy`. So wird verhindert, daß überhaupt Aliase entstehen; auf jeden Wert kann zu jedem Zeitpunkt also nur über genau einen Namen zugegriffen werden.

Diese Lösung ist jedoch nicht sehr effizient, da unnötige Kopien erzeugt werden. Ein Beispiel soll dies verdeutlichen:

```
void func1(TUPLE t) { t = t.tl(); }
void func2(TUPLE t) { cout << t.hash(); }
...
TUPLE my_tuple(an_element);
func1(my_tuple);
func2(my_tuple);
```

Bei der Parameterübergabe würde ein Alias entstehen: in den Funktionen `func1` und `func2` wird dann über den Namen `t` auf den selben Wert zugegriffen, auf den

[1] vor dieser Diplomarbeit

in der aufrufenden Funktion auch über den Namen `my_tuple` zugegriffen wird. Also muß durch Aufruf der Kopieroperation jeweils eine Kopie dieses Wertes erzeugt und dem Namen `t` zugeordnet werden, um eben dies zu verhindern. Für die Funktion `func1` ist dies auch tatsächlich notwendig, da sonst der `my_tuple` zugeordnete Wert durch die `tl`-Operation verändert würde. Für die Funktion `func2` ist diese Kopie jedoch überflüssig, da hier der Wert, auf den `t` und `my_tuple` verweisen, nicht verändert wird !

Um diese unnötigen Kopien zu vermeiden, soll hier ein Ansatz zur Realisierung der Wertsemantik gewählt werden, der in [St92, S. 271] als *Wertsemantik mit „aufgeschobener Kopie“ (delayed copy)* bezeichnet wird. Im Gegensatz zu dem oben beschriebenen Ansatz wird hier die Aliasbildung geduldet. Dies ist möglich, da solange ohne Probleme über zwei oder mehr Namen auf denselben Wert zugegriffen werden kann, bis dieser Wert über einen der Namen verändert werden soll. Erst jetzt muß eine Kopie des Wertes erzeugt werden, um zu verhindern, daß diese Änderung auch Auswirkungen auf die anderen Namen hat ! Findet jedoch keine Veränderung des Wertes statt, wie dies in `func2` des oberen Beispiels der Fall ist, so muß auch keine Kopie erzeugt werden.

Von allen Operationen, die die VDM Class Library bereitstellt, verändern nur die Kombinatoroperationen den Wert eines ADT. Um diesen Ansatz zu realisieren, müssen deshalb alle Kombinatoroperationen so erweitert werden, daß diese eine Kopie der Werte erzeugen, die sie verändern.

Auch diese Lösung ist in Bezug auf die erreichbare Effizienz noch nicht völlig befriedigend. Ein weiteres Beispiel soll dies verdeutlichen:

```
...
while (tuple1.len() < 100)
  tuple1 = tuple1.conc(tuple2);
...
```

In dieser `while`-Schleife muß in jedem Durchlauf eine Kopie der `tuple1` und `tuple2` zugeordneten Werte erzeugt werden, da die Kombinatoroperation `conc` beide Werte verändert. Da das Ergebnis der Konkatenation aber sofort wieder `tuple1` zugewiesen wird, ist die Kopie des Wertes von `tuple1` eigentlich überflüssig, denn die Veränderung dieses Wertes ist ja erwünscht ! Da dieser Fall sehr häufig in Programmen auftritt, bedeutet dies eine große Anzahl überflüssiger Kopien und somit einen hohen Verlust an Effektivität. Aus diesem Grund wird folgendes festgelegt:

> Der Wert eines C++-Objektes, auf welches — durch Aufruf der entsprechenden Memberfunktion — eine Kombinatoroperation angewendet wird, darf durch diese Operation verändert werden; es hat nach

> der Operation denselben Wert wie das Ergebnis dieser Operation. Die Werte der C++-Objekte, die als zusätzliche Parameter an die Kombinatoroperation übergeben werden, dürfen jedoch nie verändert werden.

Aus Effizienzgründen werden also Kombinatoroperationen zugelassen, die keine neuen Werte erzeugen, sondern über einen Seiteneffekt einen bereits existierenden Wert verändern. Ein solche Abweichung von der „reinen Lehre" findet man mit derselben Begründung sogar in funktionalen Sprachen wie Lisp; dort wurde z.B. eine Operation `RPLACA`[2] eingeführt, die den Wert einer Liste verändert, indem das erste Element einer Liste durch ein anderes ersetzt wird. Das letzte Beispiel kann nun folgendermaßen realisiert werden:

```
while (tuple1.len() < 100)
  (void) tuple1.conc(tuple2);
```

Die Anzahl der zu erzeugenden Kopien verringert sich durch diese Vorgehensweise erheblich. Für das C++-Objekt, auf welches eine Kombinatoroperation angewendet wird, muß nur noch dann eine Kopie des zugeordneten Wertes erzeugt werden, wenn mehr als ein Verweis auf diesen Wert vorhanden ist — eine Auswirkung auf andere Namen soll natürlich weiterhin vermieden werden. Zum Problem der Bestimmung der Anzahl der auf einen Wert vorhandenen Verweise sei auf 4.2.2 verwiesen. Für das letzte Beispiel bedeutet dies, daß in der Schleife nur max. eine Kopie des `tuple1` zugeordneten Wertes erzeugt werden muß.

Diese Vorgehensweise hat jedoch auch einen Nachteil: wird der ursprüngliche Wert eines C++-Objektes, auf welches eine Kombinatoroperation angewendet werden soll, nach dieser Operation noch benötigt, so muß dieser Wert in einem temporären Objekt gespeichert werden:

```
TUPLE tmp = tuple1;
cout << tuple1.tl();
tuple1 = tmp;
```

Dieser Nachteil wird aufgrund der erheblich verbesserten Effektivität jedoch akzeptiert !

Aufgrund der Syntax des Memberfunktionsaufrufes ist die Unterscheidung zwischen dem C++-Objekt, auf welches eine Operation angewendet und dessen zugeordneter Wert bei einer Kombinatoroperation auch verändert wird, und den als zusätzlichen Parametern übergebenen C++-Objekten, deren Wert nie verändert

[2] replace CAR

wird, sehr einfach. Dies erhöht deshalb die Lesbarkeit und Verständlichkeit des Programmes. Bei Friend-Funktionen, die der VDM-Compiler ebenfalls für die C++-Schnittstelle generieren kann, ist diese Unterscheidung aufgrund der Syntax nicht möglich. Aus diesem Grund sollte man auf die Verwendung von Friend-Funktionen verzichten.

4.2 Speicherverwaltung

4.2.1 Auswahl eines Lösungsansatzes

Das Ziel der neuen automatischen Speicherverwaltung für die vom VDM-Compiler erzeugten abstrakten Datentypen ist, den Anwender von der fehlerträchtigen Aufgabe der manuellen Speicherverwaltung zu befreien. Der Anwender erzeugt also nur noch die von ihm benötigten abstrakten Datentypen mit Hilfe der bereitgestellten Konstruktoroperationen; für das Löschen ist er jedoch nicht mehr zuständig. Mit Ausnahme der manuellen Speicherverwaltung, die in der aktuellen Version ja verwendet wird, erfüllen im Prinzip alle in Abschnitt 3 beschriebenen Speicherverwaltungsstrategien diese Anforderung:

- *„Gleichgültiger" Ansatz:* auch bei diesem Ansatz wird der Anwender von der Notwendigkeit, nicht mehr benötigte ADT manuell zu löschen, befreit. Vom Standpunkt des notwendigen Implementierungsaufwandes aus gesehen, wäre dies sogar die „optimale" Lösung, da lediglich die Operationen `del` und `del1` aus der C++-Schnittstelle entfernt werden müßten. Aufgrund der auftretenden Speicherverschwendung ist dieser Ansatz jedoch nicht für eine allgemein verwendbare Klassenbibliothek geeignet.

- *Garbage Collection:* vom Standpunkt des Anwenders aus gesehen, wäre dies die optimale Lösung, da die Speicherverwaltung nicht nur für die vom VDM-Compiler erzeugten ADT sondern für alle dynamischen Objekte automatisiert würde. Diese Lösung hat jedoch den Nachteil, daß zur Realisierung erhebliche Arbeiten auf der Sprachimplementierungsebene, d.h. am C/C++-Compiler, notwendig sind. Somit wären die vom VDM-Compiler erzeugten ADT von einem bestimmten Compiler abhängig, da nur die wenigsten Compiler über einen solchen Garbage Collector verfügen. Für eine portable Klassenbibliothek ist dies also nicht sinnvoll. Ein weiterer Punkt, der gegen diese Lösung spricht, soll nicht verschwiegen werden: die Implementation eines Garbage Collectors für einen C/C++-Compiler ist aufgrund semantischer Eigenschaften der Sprache sehr aufwendig[3].

[3] die ausführliche Beschreibung einer solchen Implementation findet sich in [De91]

- *Selbstverwaltungsansatz:* dieser Lösungsansatz stimmt in seiner Zielrichtung (siehe 3.4) genau mit dem hier zu lösenden Problem überein: die VDM Class Library ist eine Klassenbibliothek, die allgemein wiederverwendbare abstrakte Datentypen für den Anwendungsprogrammierer bereitstellt; die Implementationssprache C/C++ verwendet die manuelle Speicherverwaltung.

Nach dieser Gegenüberstellung der verschiedenen möglichen Lösungsansätze ist die Auswahl des zu verwendenden Ansatzes recht einfach: es wird der Selbstverwaltungsansatz verwendet !

4.2.2 Selbstverwaltung mit Referenzzählern

Die Realisierung des Selbstverwaltungsansatzes für die vom VDM-Compiler erzeugten abstrakten Datentypen erfolgt mit Hilfe von Referenzzählern. Im Gegensatz zu dem in 3.5.1 beschriebenen Referenzzählerverfahren zur Realisierung eines Garbage Collectors wird jedoch nicht für jedes einzelne dynamische Objekt, sondern nur für jeden ADT[4] ein eigener Referenzzähler verwaltet. Ein solcher Ansatz ist häufig in der Literatur beschrieben, so z.B. in [St92, Kap. 7.11 und Kap. 13.9].

Der Referenzzähler gibt an, wieviele Verweise auf einen ADT vorhanden sind. Die Verwaltung des Referenzzählers erfolgt entsprechend der Beschreibung in 3.5.1. Für jeden neu eingerichteten Verweis wird der Referenzzähler um eins erhöht, für jeden entfernten Verweis wird er um eins verringert. Erreicht der Referenzzähler hierbei den Wert null, so kann der der ADT gelöscht werden.

Wann wird nun ein neuer Verweis auf einen ADT eingerichtet und wann wird ein bereits bestehender Verweis wieder entfernt ? Die verschiedenen Möglichkeiten sollen im folgenden beschrieben werden[5]. Für die Beispiele werden durch die DSL-Spezifikation

```
DOMAINS
   Natset = Nat0 -set
   Setset = Natset -set
IMPLEMENTATIONS
   Natset = ...
   Setset = ...
```

erzeugte ADT verwendet.

[4]der ADT selbst kann aus sehr vielen dynamischen Objekten bestehen

[5]diese Beschreibung stellt nur eine grobe Übersicht dar, eine formale Spezifikation der verschiedenen Abläufe folgt in 5.1

1. Ein ADT wird mit Hilfe einer Konstruktoroperation erzeugt und einem C++-Objekt zugeordnet. Es ist nun genau ein Verweis von diesem C++-Objekt auf den ADT vorhanden; der Referenzzähler wird also mit eins initialisiert. Beispiel:

   ```
   NATSET n(empty); // n verweist auf leeren Set
   ```

2. Durch eine Kombinatoroperation wird ein bereits existierender ADT als Element in einen anderen ADT eingefügt oder, wenn er schon Element eines anderen ADT ist, aus diesem entfernt. Auch durch eine Konstruktoroperation kann ein ADT als Element in einen ADT — den durch den Konstruktor gerade erzeugten — eingefügt werden. Es wird ein neuer Verweis auf den eingefügten ADT eingerichtet; ein bestehender Verweis auf den entfernten ADT wird gelöscht. Beispiel:

   ```
   NATSET n(42);
   SETSET s(n); // einfügen, neuer Verweis auf {42}
   s.sub1(n); // entfernen, Verweis auf {42} gelöscht
   ```

3. Durch eine Selektoroperation wird ein temporäres C++-Objekt erzeugt, dem ein als Element in einem anderen ADT bereits existierender ADT zugeordnet wird. Es wird ein neuer Verweis auf den selektierten ADT eingerichtet. Beispiel:

   ```
   NATSET n(42);
   SETSET s1(n);
   SETSET s2(s1.sel()); // selektieren und einfügen
   ```

4. Ein C++-Objekt wird durch ein bereits existierendes C++-Objekt initialisiert. Dem initialisierten C++-Objekt ist nun der gleiche ADT zugeordnet wie dem initialisierenden; ein neuer Verweis auf diesen ADT wurde eingerichtet. Beispiel:

   ```
   NATSET n1(42);
   NATSET n2 = n1; // neuer Verweis auf {42}
   ```

   Dieser Fall muß von der weiter unten aufgeführten Zuweisung unterschieden werden, da hier n2 — im Gegensatz zur Zuweisung — noch keinen definierten Wert hat.

5. Ein C++-Objekt wird als aktueller Parameter an eine Funktion übergeben. Dem in der Funktion verwendeten formalen Parameter wird hierbei derselbe ADT zugeordnet, der auch dem aktuellen Parameter zugeordnet ist. Ein neuer Verweis auf diesen ADT wird eingerichtet. Beispiel:

```
void func(NATSET formal) {...}
...
NATSET actual(42);
func(actual);
```

Die Parameterübergabe kann auch als Initialisierung verstanden werden. Durch die Parameterübergabe wird ein bisher nicht initialisiertes C++-Objekt — der formale Parameter — durch den übergebenen aktuellen Parameter initalisiert.

6. Es findet eine Zuweisung zwischen zwei C++-Objekten statt. Nach der Zuweisung ist beiden C++-Objekten der selbe ADT zugeordnet. Beispiel:

```
NATSET n1(42);
NATSET n2(empty);
n2 = n1; // neuer Verweis auf 42, Verweis auf {} entfernt
```

Der ursprüngliche Verweis von n2 auf den zugeordneten ADT wird entfernt und einer neuer Verweis auf den n1 zugeordneten ADT eingerichtet. Anmerkung: bei manueller Speicherverwaltung wird hier eine „Speicherleiche" erzeugt, da nach der Zuweisung der ursprünglich n2 zugeordnete ADT nicht mehr erreichbar ist und somit auch nicht mehr gelöscht werden kann.

7. Wird im Programmablauf der Gültigkeitsbereich eines C++-Objektes verlassen, so wird dieses C++-Objekt gelöscht und der Verweis auf den zugeordneten ADT entfernt. Beispiel:

```
NATSET n1(42);
{
   NATSET n2 = n1;
} n2 wird gelöscht, der Verweis auf {42} entfernt
```

8. Wird ein ADT gelöscht, da sein Referenzzähler den Wert 0 erreicht hat, so werden alle in diesem ADT vorhandenen Verweise auf Elemente entfernt.

Da bei der Einrichtung eines neuen bzw. Entfernung eines bestehenden Verweises immer ein Referenzzähler manipuliert werden muß, stellt sich nun eine wichtige Frage: wie kann der Programmierer die hierfür notwendigen Aktionen in die beschriebenen Vorgänge einbinden ?

Einfach ist dies in den Fällen, in denen Verweise durch vom VDM-Compiler erzeugte Operationen neu eingerichtet oder entfernt werden. Diese Operationen liegen im Quellcode in der Datenbank der VDM Class Library vor und können dort bei Bedarf leicht erweitert werden. Auf die Zuweisung, die Initialisierung, die Parameterübergabe und das Verlassen des Gültigkeitsbereiches trifft dies jedoch nicht zu. Die hier ausgeführten Aktionen werden durch die jeweilige Sprache definiert.

Während C, Modula-2 und Pascal in diesem Zusammenhang keine weitere Unterstützung bieten, ist es in C++ möglich, die standardmäßigen Aktionen dieser vier Vorgänge durch eigene Funktionen zu erstetzen. Diese stellen dann die notwendige Funktionalität zur Verwaltung des Referenzzählers bereit. Dies ist auch der Grund dafür, daß die neue Speicherverwaltung nur für die Sprache C++ und nicht auch für die anderen von der VDM Class Library unterstützen Sprachen realisiert werden kann.

Die hierfür von C++ bereitgestellten Sprachmittel sollen kurz erwähnt werden; für eine ausführliche Beschreibung sei auf [St92] verwiesen.

- der *Zuweisungoperator '='* kann — wie auch alle anderen Operatoren — für Klassentypen redefiniert werden.

- für Klassentypen kann eine spezielle Memberfunktion, der *Destruktor*, definiert werden, die immer dann aufgerufen wird, wenn der Gültigkeitsbereich eines Objektes dieser Klasse verlassen wird.

- für Klassentypen kann eine spezielle Memberfunktion, der *Kopierkonstruktor* (copy constructor), definiert werden, die für die Initialisierung eines Objektes dieser Klasse mittels eines anderen Objektes aufgerufen wird. Eine solche Initialisierung findet auch bei der Parameterübergabe und bei der Rückgabe eines Funktionswertes statt.

Mit Hilfe dieser Sprachmittel ist es recht einfach, die notwendige Funktionalität zur Manipulation des Referenzzählers bereitzustellen.

4.2.3 Kritik an Referenzzählern

In Abschnitt 3.5.1 wurden zwei gewichtige Nachteile von Referenzzählern beschrieben, die dazu führen, daß Referenzzähler nicht zur Realisierung eines eines Garbage Collectors verwendet werden. Wie sind diese Nachteile nun im Zusammenhang mit dem hier gewählten Selbstverwaltungsansatz mit Referenzzählern zu bewerten ?

Die hohen Kosten — in Bezug auf Rechenzeit und Speicherbedarf — der Referenzzähler sind der erste dort beschriebene Nachteil. Dieser verliert hier an Bedeutung, da nicht für jedes dynamische Objekt, sondern nur für jeden ADT ein Referenzzähler verwaltet wird. Somit verringert sich sowohl der für die Referenzzähler benötigte Speicherplatz, als auch der zur Verwaltung benötigte Zeitaufwand.

Der zweite Nachteil der Referenzzählerverfahren ist die fehlerhafte Behandlung zyklischer Datenstrukturen; der von diesen belegte Speicherplatz wird nie wieder freigegeben. Könnte man bei Verwendung der manuellen Speicherverwaltung problemlos mit solchen zyklischen Datenstrukturen arbeiten, so wäre dies hier tatsächlich ein schwerwiegender Nachteil, da dann eine mächtige Klasse von Datenstrukturen aufgrund der automatischen Speicherverwaltung nicht mehr zur Verfügung stände. Dies ist aber nicht der Fall! Selbst bei Verwendung der manuellen Speicherverwaltung ist die Arbeit mit zyklischen Datenstrukturen nur sehr eingeschränkt möglich. Ein Beispiel mit einer zyklischen Liste soll dies verdeutlichen:

```
DOMAINS
   List = Node *
   Node = Info | List
   Info = ...
IMPLEMENTATIONS
   ...
```

Mit diesen ADT kann leicht eine zyklische Struktur erzeugt werden:

```
INFO my_info = ...
LIST my_list = empty;
my_list.appr(NODE(my_info));
my_list.appr(NODE(my_info));
my_list.appr(NODE(my_list));
```

Die im Speicher entstehende zyklische Datenstruktur ist für das Beispiel einer `linklist`-Implementation in Abb. 3 vereinfacht dargestellt; die Probleme treten jedoch genauso bei anderen Implementationen des Tuples auf.

Auf das Tuple `my_list` kann eine Vielzahl der bereitgestellten Operationen nicht mehr angewendet werden. Dies betrifft sowohl komplexe Operationen wie `fpr`, aber auch solch elementare Operationen wie `copy`, `del` und `is_eq`. Allen diesen Operationen ist gemein, daß sie für jedes Element des Tuples eine entsprechende Operation des Elementtyps aufrufen. Aufgrund der zyklischen Struktur führt dies hier zu einer Endlosrekursion !

Die fehlerhafte Speicherfreigabe der Referenzzählerverfahren bei zyklischen Strukturen kann *hier* also nicht als schwerwiegender Nachteil gewertet werden. Denn wie das Beispiel gezeigt hat, ist die Verarbeitung solcher zyklischen Strukturen durch die bereitgestellten Operationen selbst bei manueller Speicherverwaltung nur stark eingeschränkt möglich.

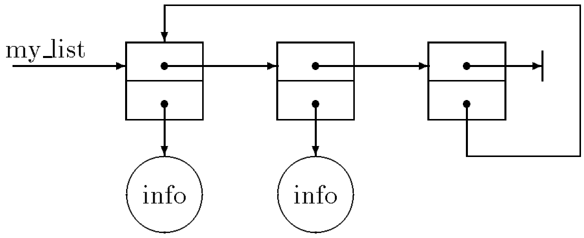


Abbildung 3: zyklische Strukturen mit ADT

4.3 Integration in das System

Nachdem nun das grundlegende Konzept der neuen automatischen Speicherverwaltung und der Wertsemantik festgelegt ist, muß überlegt werden, wie die notwendige zusätzliche Funktionalität in das bestehende System integriert werden soll. Zwei grundsätzliche Alternativen stehen zur Auswahl:

1. die in der Datenbank der VDM Class Library zur Realisierung der Operationen der ADT in den verschiedenen Implementationen benutzten Codefragemente werden direkt erweitert, um die notwendige Funktionalität bereitzustellen. Die Auswahl dieser Erweiterungen kann mit Hilfe der bedingten Kompilierung erfolgen, da diese Codefragmente mit einem C-Präprozessor bearbeitet werden.

2. für jeden Typgenerator wird eine neue Implementation `automem` bereitgestellt, die einen ADT mit automatischer Speicherverwaltung und Wertsemantik realisiert. Diese Implementation verwendet selbst wieder einen Hilfs-ADT des gleichen Typs und erweitert diesen lediglich um die zusätzlich benötigte Funktionalität; die eigentlichen Operationen werden durch Aufruf der entsprechenden Routinen des Hilfs-ADT implementiert. Die `automem`-Implementation bildet also eine „schützende“ Hülle um die bereits bestehenden Implementationen und schirmt den Anwender so von deren unangenehmen Eigenschaften ab. Die Implementation des intern verwendeten Hilfs-ADT muß der Anwender durch die Übergabe eines Parameters an die `automem`-Implementation auswählen. Die sich aus dieser Vorgehensweise ergebende Struktur eines ADT in der Implementation `automem` ist in Abb. 4 dargestellt.

Die Wahl der ersten Alternative ist in Bezug auf den entstehenden Overhead vorteilhafter. Leider ist diese Alternative aber auch mit einer fast kompletten Neucodierung der Datenbank verbunden und somit sehr aufwendig. Im Gegensatz

Anwenderprogramm in C++

C++-Schnittstelle

automem-Implementation
(stellt automat. Speicherverwaltung/Wertsemantik bereit)

interner Hilfs-ADT
(mit man. Speicherverwaltung/Referenzsemantik)

Abbildung 4: Struktur der erzeugten abstrakten Datentypen

dazu ist die zweite Alternative in dieser Hinsicht viel einfacher und eleganter, denn eine neue Implementierung pro Typgenerator ist hier ausreichend. Auch kann auf diese Weise auf eine Änderung bereits existierender Codefragmente verzichtet werden.

Aufgrund der Eleganz und Einfachheit wird — trotz eines zu erwartenden höheren Overheads — hier diese zweite Alternative gewählt !

Diese Wahl legt zugleich die für den Anwender notwendige Vorgehensweise zur Auswahl der automatischen Speicherverwaltung und der Wertsemantik fest. Um z.B. ein Tuple von natürlichen Zahlen zu erzeugen, muß der Anwender folgende DSL-Spezifikation erstellen:

```
DOMAINS
   Tuple = Nat0 *
IMPLEMENTATIONS
   Tuple = automem(BASEIMPL="array(NOCHECK)")
         + mk_empty + appr + hd + tl
```

Durch den an die Implementation `automem` übergebenen Parameter `BASEIMPL` wird die Implementation des intern verwendeten Hilfs-ADT, hier `array`, ausgewählt.

Wie schon dieses einfache Beispiel zeigt, ist diese Vorgehensweise doch recht umständlich. Für den Anwender wäre es einfacher, wenn er die automatische Speicherverwaltung und die Wertsemantik für alle ADT einer DSL-Spezifikation einfach durch die Angabe eines zusätzlichen Kommandozeilenparameters beim

Aufruf des VDM-Compilers auswählen könnte. Doch auch die manuelle Auswahl über eine spezielle Implementation hat ihre Vorteile: es können gleichzeitig ADT mit manueller Speicherverwaltung/Referenzsemantik und der automatischer Speicherverwaltung/Wertsemantik verwendet werden. Dies kann sinnvoll sein, wenn die Laufzeiteigenschaften eines Programmes, das ADT verwendet, optimiert werden sollen (durch den Overhead der neuen Speicherverwaltung und der Wertsemantik sind Laufzeitverschlechterungen zu erwarten).

Um beide Vorteil nutzen zu können, sollen auch beide Benutzungsmöglichkeiten für den Anwender zur Verfügung gestellt werden !

Zur Erleichterung der Implementierung wird jedoch ein kleiner Trick verwendet: der VDM-Compiler wird so erweitert, daß beim Aufruf mit dem entsprechenden Kommandozeilenparameter die intern erzeugte Repräsentation der DSL-Spezifikation — der vom Parser erzeugte Ableitungsbaum — so transformiert wird, daß er die selbe Struktur erhält, die er bei manueller Auswahl über die Implementation `automem` hätte. Wird also die DSL-Spezifikation

```
DOMAINS
   Tuple = Nat0 *
IMPLEMENTATIONS
   Tuple = array(NOCHECK,SIMPLEELEMS)
         + mk_empty + appr + hd + tl
```

gelesen, so wird die interne Repräsentation so transformiert, als wäre die DSL-Spezifikation

```
DOMAINS
   Tuple = Nat0 *
IMPLEMENTATIONS
   Tuple = automem(BASEIMPL="array(NOCHECK,SIMPLEELEMS)")
         + mk_empty + appr + hd + tl
```

gelesen worden. Dies vereinfacht die Erweiterung der Datenbank der VDM Class Library sehr, da keine zwei Möglichkeiten der Auswahl unterschieden werden müssen.

4.4 Bereitgestellte Operationen

4.4.1 Entfernte Operationen

Aufgrund der automatischen Speicherverwaltung und der Wertsemantik werden eine Anzahl der normalerweise bereitgestellten Operationen überflüssig, da deren Funktionen in der Implementation `automem` nun bei Bedarf automatisch ausgeführt werden. Aus diesem Grund ist es sinnvoll solche Operationen in der Implementation `automem` erst gar nicht zur Verfügung zu stellen. Folgende Operationen sind hiervon betroffen:

- `del`, `del1`: ADT werden durch die neue Speicherverwaltung automatisch gelöscht, wenn sie nicht mehr benötigt werden
- Kombinatoren mit implizitem Löschen: die neue Speicherverwaltung erkennt und löscht nicht mehr benötigte Elemente automatisch
- `copy`, `copy1`: zur Realisierung der Wertsemantik werden automatisch Kopien erzeugt, wenn dies notwendig ist

4.4.2 Operatoren

In Abschnitt 4.1 wurde aus Effizienzgründen erlaubt, daß durch eine Kombinatoroperation der Wert des C++-Objekts, auf den diese Operation angewendet wird, verändert werden darf. Durch die Syntax des Aufrufs einer Memberfunktion wird dieses veränderte Objekt deutlich hervorgehoben.

In der C++-Schnittstelle werden für einige Kombinatoroperationen zusätzlich zu den normalen Memberfunktionen auch Infix-Operatoren definiert. So sind z.B. die beiden folgenden Anweisungen äquivalent:

```
result = a.diff(b);
result = a - b;
```

Die Infix-Operatoren werden in der C++-Schnittstelle so definiert, daß sie zur Ausführung der Operation dieselbe C-Funktion aufrufen, wie die entsprechende Memberfunktion. Dies hat zur Folge, daß hier ein Operand durch die ausgeführte Operation verändert wird; im obigen Beispiel hat `a` nach Ausführung der Anweisung den selben Wert wie `result`. Im Gegensatz zum Memberfunktionsaufruf wird hier jedoch der veränderte Operand durch die Syntax nicht besonders hervorgehoben.

Dies kann zu erstaunlichen Ergebnissen bei der Anwendung der Operatoren in komplexeren Ausdrücken führen, da man aus dem gewohnten Umgang mit Operatoren in der Mathematik ein solches Verhalten nicht erwartet. So liefert z.B. der eigentlich zur Berechnung der Schnittmenge zweier Mengen a und b gedachte Ausdruck

```
cout << a-(a-b);
```

aus diesem Grund immer ein falsches Ergebnis !

Um dies zu verhindern, werden die Infix-Operatoren in ihrer Semantik an die gewohnte Semantik der Operatoren in der Mathematik angepaßt, d.h. die Operanden des Infix-Operators werden durch die Operation nicht verändert !

Häufig will man mit Hilfe eines Operators aber auch den Wert eines Operanden verändern; man verwendet dann z.B. die Anweisung

```
a = a - b;
```

Durch die oben festgelegte Änderung der Semantik der Infix-Operationen werden solche Anweisung recht ineffizient (es muß eine unnötige Kopie von `a` erzeugt werden). Um auch diese Vorgänge effizient zu realisieren, werden deshalb zusätzlich zu den bereits vorhandenen Infix-Operatoren die entsprechenden Zuweisungs-Operatoren in der C++-Schnittstelle bereitgestellt !

Anstelle des Infix-Operators würde man für das letzte Beispiel also den entsprechenden Zuweisungsoperator verwenden:

```
a -= b;
```

Die entspricht dann dem Aufruf der entsprechenden Memberfunktion `a.diff(b)`; durch die Syntax des Zuweisungsoperators wird der veränderte Operand hier auch wieder deutlich hervorgehoben.

5 Spezifikation

5.1 Die `automem`-Implementation

In diesem Abschnitt sollen die vom VDM-Compiler für die Zielsprache C++ erzeugten Operationen in der Implementation `automem` unter Verwendung von VDM spezifiziert werden.

5.1.1 Das Datenmodell

Die Spezifikation verwendet die folgende abstrakte Syntax:

$$
\begin{aligned}
Data &= Var \xrightarrow[\text{m}]{} [Handle] \\
Adts &= Handle \xrightarrow[\text{m}]{} Adt \\
Adt &:: \; rc : \mathbf{N}_0 \quad elems : Elements \\
Elements &= Handle \text{-}\mathsf{bag} \\
Var &= \mathsf{Token}_1 \\
Handle &= \mathsf{Token}_2
\end{aligned}
$$

Data repräsentiert die Menge der C++-Variablen, die vom Typ einer vom VDM-Compiler erzeugten Klasse[1] sind. Jeder Variablen ist genau ein Handle, der die Verbindung zu einem ADT herstellt, oder der Wert nil, der die sogenannten nicht-initialisierten Variablen[2] kennzeichnet , zugeordnet. *Adts* repräsentiert die Menge aller abstrakter Datentypen im System. Auf einen solchen abstrakten Datentyp kann über einen Handle zugegriffen werden.

Ein *Adt* selbst besteht in diesem Modell aus einem Referenzzähler *rc*, der die Anzahl der Verweise auf diesen ADT angibt, und einem Bag *elems* von Elementen. Elemente sind in diesem Modell nur Verweise auf andere ADT (also Handle), da nur diese für die Speicherverwaltung interessant sind. Handelt es sich bei den Elementen des ADT also um elementare Typen, z.B. bei einem Tuple von natürlichen Zahlen, so ist der Bag *elems* immer leer, denn dieser ADT enthält ja keine Verweise auf andere ADT ! Auch eine evtl. vorhandene Ordnung der Elemente (z.B. in einem Tuple oder die Paarbildung bei einer Map) wird in diesem Modell nicht betrachtet, da sie nicht relevant ist.

[1] gemeint sind hier nur Klassen die einen abstrakten Datentyp implementieren; die den elementaren Datentypen (Nat0, Bool etc.) des VDM-Compilers entsprechenden Klassen werden nicht berücksichtigt. Die ist möglich da für diese Typen eine automatische Speicherfreigabe sowie eine Wertsemantik schon standardmäßig vorhanden ist (diese Typen basieren ja nicht auf Zeigern).

[2] siehe hierzu auch 5.1.2

Ein Wert des Typs *Data* und ein Wert des Typs *Adts* beschreiben zusammen alle hier interessierenden Aspekte eines C++-Programmes; diese beiden Werte werden deshalb als Zustand *Program* zusammengefaßt:

$$
\begin{array}{ll}
\mathsf{state}\ \ \mathit{Program}\ \mathsf{of} & \\
\quad \mathit{data} \ \ : \mathit{Data} & \\
\quad \mathit{adts} \ \ : \mathit{Adts} & \\
\mathsf{end} & \\
\mathsf{inv}\ \ \mathit{mk\text{-}Program(data,adts)} \ \widehat{=} & \\
\quad \mathsf{rng}\ \mathit{data} - \{\mathsf{nil}\} \subseteq \mathsf{dom}\ \mathit{adts} \quad \wedge & (1) \\
\quad \forall \mathit{adt} \in \mathsf{rng}\ \mathit{adts} \cdot \mathit{bag\text{-}to\text{-}set}(\mathit{adt.elems}) \subseteq \mathsf{dom}\ \mathit{adts} \quad \wedge & (2) \\
\quad \forall h \in \mathsf{dom}\ \mathit{adts} \cdot (\mathit{adts}(h).\mathit{rc} = \mathit{cnt\text{-}references}(h, \mathit{data}, \mathit{adts}) \wedge & (3) \\
\qquad\qquad\qquad\qquad \mathit{is\text{-}accessible}(h, \mathit{data}, \mathit{adts})) \quad \wedge & (4) \\
\quad \mathit{is\text{-}acyclic}(\mathit{adts}) & (5) \\
\mathsf{init}\ \ \mathit{mk\text{-}Program(data,adts)} \ \widehat{=} & \\
\quad \mathit{data} = [\,] \quad \wedge & \\
\quad \mathit{adts} = [\,] &
\end{array}
$$

Alle im weiteren spezifizierten Operationen arbeiten auf diesem Zustand ! Die in der Definition des Zustandes *Program* verwendeten Hilfsfunktionen, wie auch alle weiteren Hilfsfunktionen, werden in Abschnitt 5.1.9 definiert.

Die Invariante des Zustandes *Program* ist recht interessant; sie legt fest, daß ...

- jede C++-Variable auf einen existierenden ADT verweist oder nicht-initialisiert ist; eine C++-Variable enthält also nie einen *dangling pointer* (Term 1)
- die Elemente eines ADT stets elementare Typen (entspricht einem leeren Bag *elems*) oder Verweise auf existierende ADT sind; ein ADT enthält als Element also ebenfalls nie einen *dangling pointer* (Term 2)
- der Wert des Referenzzähler eines ADT immer genau der Anzahl der Verweise auf diesen ADT entspricht (Term 3)
- jeder existierende ADT erreichbar[3] ist, d.h. der Anwender kann direkt (über eine C++-Variable) oder indirekt (durch eine Folge von Selektoroperationen) auf jeden ADT zugreifen; es existieren also keine „Speicherleichen“ (Term 4)
- die Menge aller ADT stets zyklenfrei[4] ist (Term 5)

[3]siehe hierzu auch 3.5
[4]siehe hierzu auch 4.2.3

5.1.2 Konstruktoren

Die Operation *constructor* ist die Verallgemeinerung der von allen Typgeneratoren bereitgestellten Konstruktoren (z.B. mk_empty, mk1, mk2, ...). Es wird eine neue Variable *var* erzeugt, die auf einen ebenfalls neu erzeugten ADT verweist. Elemente des neu erzeugten ADT können durch den Bag *with* ausgewählt werden. Handelt es sich bei den Elementen nicht um andere ADT sondern um elementare Typen, so wird dies durch einen leeren Bag dargestellt (siehe Beschreibung des Datenmodells in 5.1.1).

$$
\begin{array}{l}
constructor(var : Var, with : Var\text{-bag}) \\
\mathsf{ext}\;\; \mathsf{wr}\; data : Data \\
\qquad \mathsf{wr}\; adts : Adts \\
\\
\mathsf{pre}\;\; var \notin \mathsf{dom}\; data \wedge \textit{is-init-vars}(data, with) \\
\\
\mathsf{post}\; \mathsf{let}\; elems = \{\overline{data}(v) \mid v \in with\}\text{-bag}\;\; \mathsf{in} \\
\qquad \mathsf{let}\; hdl = \textit{new-handle}\,(\overline{adts})\;\; \mathsf{in} \\
\qquad\quad data = \overline{data} \cup [var \mapsto hdl] \quad \wedge \\
\qquad\quad adts = \textit{inc-ref-cnt}\,(\overline{adts} \cup [hdl \mapsto \textit{mk-adt}(1, elems)], elems)
\end{array}
$$

Die Operation *nil-constructor* erzeugt eine sogenannte nicht-initialisierte Variable *var*, der bei ihrer Erzeugung kein ADT sondern der Wert nil zugeordnet wird. Einer solchen nicht-initialisierten Variablen kann jedoch später durch eine Zuweisung (Operation *assign*) ein ADT zugeordnet werden. An andere Operationen darf eine nicht-initialisierte Variable nicht als Parameter übergeben werden (Ausnahme ist *destructor*, siehe jeweils die Vorbedingungen).

$$
\begin{array}{l}
\textit{nil-constructor}\,(var : Var) \\
\mathsf{ext}\;\; \mathsf{wr}\; data : Data \\
\\
\mathsf{pre}\;\; var \notin \mathsf{dom}\; data \\
\\
\mathsf{post}\; data = \overline{data} \cup [var \mapsto \mathsf{nil}]
\end{array}
$$

5.1.3 Attribute und Prädikate

Die Operation *attribut* ist die Verallgemeinerung der von allen Typgeneratoren bereitgestellten Attribute und Prädikate (ein Prädikat kann im Prinzip als Spezialfall eines Attributes betrachtet werden, bei dem der Ergebniswert vom Typ

B ist). Die eigentliche Berechnung eines Ergebniswertes wird hier nicht mit dargestellt, da sie für die Speicherverwaltung irrelevant ist. Wichtig ist hier nur, daß der Zustand *Program* durch die Operation *attribut* nicht verändert wird. Aufgrund der Zugriffsrechte ist die Nachbedingung eigentlich redundant, sie ist nur zur Verdeutlichung nochmals aufgeführt.

$$
\begin{array}{ll}
\multicolumn{2}{l}{attribut(vars : Var\text{–}\mathsf{bag})} \\
\mathsf{ext} & \mathsf{rd}\ data : Data \\
 & \mathsf{rd}\ adts : Adts \\
\mathsf{pre} & \textit{is-init-vars}\,(data, vars) \\
 & \\
\multicolumn{2}{l}{\mathsf{post}\ data = \overline{data} \wedge adts = \overline{adts}}
\end{array}
$$

5.1.4 Selektoren

Die Operation *selector* ist die Verallgemeinerung der von allen Typgeneratoren bereitgestellten Selektoren. Es wird eine neue Variable *var* erzeugt, die auf einen bereits existierenden ADT verweist, der Element des durch die Variable *from* ausgewählten ADT ist. Die Art und Weise, wie das selektierte Element aus der Menge aller Elemente ausgewählt wird, ist für die Speicherverwaltung nicht relevant und wird daher nicht dargestellt.

$$
\begin{array}{ll}
\multicolumn{2}{l}{selector(var : Var, from : Var)} \\
\mathsf{ext} & \mathsf{wr}\ data : Data \\
 & \mathsf{wr}\ adts : Adts \\
\mathsf{pre} & var \notin \mathsf{dom}\ data \wedge \textit{is-init-vars}\,(data, \{from\}) \wedge \\
 & adts(data(from)).elems \neq \{\} \\
\mathsf{post} & \mathsf{let}\ hdl \in \overline{adts}(\overline{data}(from)).elems\ \ \mathsf{in} \\
 & data = \overline{data} \cup [var \mapsto hdl] \quad \wedge \\
 & adts = \textit{inc-ref-cnt}\,(\overline{adts}, \{hdl\})
\end{array}
$$

5.1.5 Kombinatoren

Die Operation *update* ist die Verallgemeinerung der von allen Typgeneratoren bereitgestellten Kombinatoren. Durch diese Operation wird der ADT, auf den die Variable *var* verweist, verändert. Um zu verhindern, daß die Änderung dieses ausgewählten ADT Auswirkungen auf andere Variablen, die ebenfalls auf dieses ADT verweisen, oder andere ADT, in denen dieser ADT Element ist, hat, wird durch die Funktion *cond-clone* eine Kopie dieses ADT erzeugt, wenn mehr als ein Verweis auf ihn vorhanden ist. Die durch die Variablen im Bag *with* ausgewählten ADT werden nicht verändert, sie dienen lediglich als „Lieferant“ der für die

Operation benötigten Elemente. Sind diese Elemente, die zur Veränderung des ADT benötigt werden, elementare Datentypen, so wird dies durch einen leeren Bag modelliert (siehe Beschreibung zum Datenmodell in 5.1.1).

$$\begin{array}{l} update(var : Var, with : Var\text{-}\mathsf{bag}, op : Operation) \\ \mathsf{ext}\ \ \mathsf{wr}\ data : Data \\ \qquad \mathsf{wr}\ adts : Adts \\ \\ \mathsf{pre}\ \ is\text{-}init\text{-}vars\,(data, with \cup \{var\}) \quad \wedge \\ \qquad \mathsf{let}\ (\ , a_1) = do\text{-}update\,(data, adts, var, with, op)\ \ \mathsf{in} \\ \qquad\quad is\text{-}acyclic\,(a_1) \\ \\ \mathsf{post}\ (data, adts) = do\text{-}update\,(\overline{data}, \overline{adts}, var, with, op) \end{array}$$

Die Operation *update* ist die einzige Operation, die Zyklen in *adts* erzeugen kann; hierdurch würde die Invariante des Zustandes *Program* verletzt. Also muß durch eine entsprechende Vorbedingung erreicht werden, das die Operation nur für solche Aufrufe definiert ist, die keine Zyklen erzeugen. Ob durch einen Aufruf aber Zyklen entstehen, kann erst geprüft werden, *nachdem* die eigentliche Berechnung des Ergebnisses durchgeführt wurde. Dies führt zu der auf den ersten Blick seltsam erscheinenden Vorbedingung, in der derselbe Wert berechnet wird, der auch in der Nachbedingung das Ergebnis dieser Operation definiert.

$$\begin{array}{l} \mathsf{type}\ do\text{-}update : Data \times Adts \times Var \times Var\text{-}\mathsf{bag} \times Operation \\ \qquad\qquad\qquad\quad \longrightarrow Data \times Adts \\ \\ \mathsf{pre\text{-}}\ do\text{-}update\,(d, a, v, w, o) \mathrel{\widehat{=}} is\text{-}init\text{-}vars\,(d, w \cup \{v\}) \\ \\ do\text{-}update\,(data, adts, var, with, op) \ \mathrel{\widehat{=}} \\ \quad \mathsf{let}\ (new\text{-}data\,, a_1) = cond\text{-}clone\,(data, adts, var)\ \ \mathsf{in} \\ \quad \mathsf{let}\ elems = \{new\text{-}data\,(v) \mid v \in with\}\text{-}\mathsf{bag}\ \ \mathsf{in} \\ \quad \mathsf{let}\ h = new\text{-}data\,(var)\ \ \mathsf{in} \\ \quad \mathsf{let}\ (ins, rmv) = op(h, elems, a_1)\ \ \mathsf{in} \\ \quad \mathsf{let}\ a_2 = a_1 + [h \mapsto mk\text{-}adt\,(a_1(h).rc, a_1(h).elems \cup ins - rmv)]\ \ \mathsf{in} \\ \quad \mathsf{let}\ new\text{-}adts\ = dec\text{-}ref\text{-}cnt\,(inc\text{-}ref\text{-}cnt\,(a_2, ins), rmv)\ \ \mathsf{in} \\ \qquad (new\text{-}data\,, new\text{-}adts\,) \end{array}$$

Die eigentlich durchzuführende Kombinatoroperation (`union`, `sub1`, `tl` etc.) wird durch die als Parameter übergebene Funktion *op* bestimmt. Diese ermittelt in Abhängigkeit der ausgewählten ADT die tatsächlich einzufügenden bzw. zu entfernenden Elemente. Der Typ *Operation* ist folgendermaßen definiert:

$$Operation = Handle \times Elements \times Adts \longrightarrow Elements \times Elements$$

Beispiele für Funktionen, die als Parameter *op* für eine bestimmte Kombinatoroperation an *update* übergeben werden, lassen sich in diesem Modell nicht spezifizieren. Diese Funktionen benötigen in der Regel einen Zugriff auf die in den ADT gespeicherten elementaren Daten (Nat0, Bool etc.), die hier ja nicht im Modell enthalten sind !

5.1.6 Destruktor

Die Operation *destructor* spezifiziert die beim Verlassen des Gültigkeitsbereichs einer C++-Variablen *var* durchzuführenden Aktionen. Die Variable *var* wird gelöscht und der evtl. vorher vorhandene Verweis auf einen ADT entfernt.

$destructor(var : Var)$
ext wr $data : Data$
wr $adts : adts$

pre $var \in \mathsf{dom}\ data$

post $data = \overline{data} - \{var\} \quad \wedge$
$adts = \mathsf{if}\ \overline{data}(var) = \mathsf{nil}$
then $\overline{adts}$
else $dec\text{-}ref\text{-}cnt(\overline{adts}, \{\overline{data}(var)\})$

5.1.7 Zuweisung

Die Operation *assign* spezifiziert die bei einer Zuweisung in der Zielsprache C++ durchzuführenden Aktionen. Der Zuweisung `left=right` entspricht die Operation $assign(left, right)$. Der Verweis auf den ursprünglich der Variablen *lval* zugeordneten ADT wird entfernt (falls *lval* initialisiert ist) und es wird ihr derselbe ADT zugeordnet, der auch der Variablen *rval* zugeordnet ist.

$assign(lval : Var, rval : Var)$
ext wr $data : Data$
wr $adts : adts$

pre $lval \in \mathsf{dom}\ data \wedge is\text{-}init\text{-}vars(data, \{rval\})$

$$\begin{array}{l}
\mathsf{post\ let}\ adts_1 = \mathit{inc\text{-}ref\text{-}cnt}\,(\overline{adts}, \{\overline{data}(rval)\})\ \ \mathsf{in} \\
\qquad data = \overline{data} + [lval \mapsto \overline{data}(rval)] \quad \wedge \\
\qquad adts = \mathsf{if}\ \overline{data}(lval) = \mathsf{nil} \\
\qquad\qquad \mathsf{then}\ adts_1 \\
\qquad\qquad \mathsf{else}\ \mathit{dec\text{-}ref\text{-}cnt}\,(adts_1, \{\overline{data}(lval)\})
\end{array}$$

5.1.8 Kopierkonstruktor

Die Operation *copy-constr* spezifiziert die bei der Initialisierung einer neu erzeugten C++-Variablen var_1 durch eine bereits existierende C++-Variable var_2 durchzuführenden Aktionen. Trotz des Names wird keine Kopie des ADT, auf den var_2 verweist, erstellt, sondern die neu erzeugte Variable var_1 enthält lediglich einen weiteren Verweis auf diesen ADT.

$$\begin{array}{l}
\mathit{copy\text{-}constr}\,(var_1 : Var, var_2 : Var) \\
\mathsf{ext}\ \ \mathsf{wr}\ data : Data \\
\qquad \mathsf{wr}\ adts : Adts \\
\\
\mathsf{pre}\ \ var_1 \notin \mathsf{dom}\ data \wedge \mathit{is\text{-}init\text{-}vars}\,(data, \{var_2\}) \\
\\
\mathsf{post}\ data = \overline{data} \cup [var_1 \mapsto \overline{data}(var_2)] \quad \wedge \\
\qquad adts = \mathit{inc\text{-}ref\text{-}cnt}\,(\overline{adts}, \{\overline{data}(var_2)\})
\end{array}$$

5.1.9 Hilfsfunktionen

Die Funktion *bag-to-set* wandelt einen Bag von Handles in einen Set, der dieselben Handles — jedoch ohne Duplikate — enthält.

$$\mathsf{type}\ \mathit{bag\text{-}to\text{-}set}\ : Handle\ \mathsf{-bag} \longrightarrow Handle\ \mathsf{-set}$$

$$\mathit{bag\text{-}to\text{-}set(bag)}\ \mathrel{\widehat{=}}\ \{h \mid h \in bag\}$$

Die Funktion *cnt-references* bestimmt die Gesamtanzahl der Verweise auf einen ADT, der durch den als Parameter übergebenen Handle ausgewählt wird.

$$\begin{array}{ll}
\mathsf{type}\ \mathit{cnt\text{-}references} & : Handle \times Data \times Adts \longrightarrow \mathbf{N}_0 \\
\mathsf{type}\ \mathit{cnt\text{-}var\text{-}refs} & : Handle \times Data \longrightarrow \mathbf{N}_0 \\
\mathsf{type}\ \mathit{cnt\text{-}elem\text{-}refs} & : Handle \times Adts \longrightarrow \mathbf{N}_0 \\
\mathsf{type}\ \mathit{occurs} & : Handle \times Elements \longrightarrow \mathbf{N}_0
\end{array}$$

$$
\begin{array}{l}
\textit{cnt-references}\,(hdl, data, adts) \;\; \widehat{=} \\
\quad \textit{cnt-var-refs}\,(hdl, data) + \textit{cnt-elem-refs}\,(hdl, adts)
\end{array}
$$

$$
\begin{array}{l}
\textit{cnt-var-refs}\,(hdl, data) \;\; \widehat{=} \\
\quad \mathsf{if}\ data = [\,] \\
\quad \mathsf{then}\ 0 \\
\quad \mathsf{else}\ \ \mathsf{let}\ var \in \mathsf{dom}\ data\ \ \mathsf{in} \\
\qquad\quad \mathsf{let}\ n = \ \mathsf{if}\ hdl = data(var)\ \mathsf{then}\ 1\ \mathsf{else}\ 0\ \ \mathsf{in} \\
\qquad\qquad n + \textit{cnt-var-refs}\,(hdl, data - \{var\})
\end{array}
$$

$$
\begin{array}{l}
\textit{cnt-elem-refs}\,(hdl, adts) \;\; \widehat{=} \\
\quad \mathsf{if}\ adts = [\,] \\
\quad \mathsf{then}\ 0 \\
\quad \mathsf{else}\ \ \mathsf{let}\ h \in \mathsf{dom}\ adts\ \ \mathsf{in} \\
\qquad\quad \mathsf{let}\ n = occurs(hdl, adts(h).elems)\ \ \mathsf{in} \\
\qquad\qquad n + \textit{cnt-elem-refs}\,(hdl, adts - \{h\})
\end{array}
$$

$$
\begin{array}{l}
occurs\,(hdl, elems) \;\; \widehat{=} \\
\quad \mathsf{if}\ elems = \{\ \} \\
\quad \mathsf{then}\ 0 \\
\quad \mathsf{else}\ \ \mathsf{let}\ h \in elems\ \ \mathsf{in} \\
\qquad\quad \mathsf{let}\ n = \ \mathsf{if}\ hdl = h\ \mathsf{then}\ 1\ \mathsf{else}\ 0\ \ \mathsf{in} \\
\qquad\qquad n + occurs\,(hdl, elems - \{h\})
\end{array}
$$

Die Funktion *is-accessible* prüft, ob der durch einen Handle ausgewählte ADT von einer zur Zeit existierenden C++-Variablen aus erreichbar ist. Dies ist der Fall, wenn der ADT selbst einer Variablen zugeordnet ist oder von einem ADT aus erreichbar ist, der einer C++-Variablen zugeordnet ist.

$$\mathsf{type}\ \textit{is-accessible} : Handle \times Data \times Adts \longrightarrow \mathbf{B}$$

$$
\begin{array}{l}
\textit{is-accessible(hdl,data,adts)} \;\; \widehat{=} \\
\quad \exists h \in (\mathsf{rng}\ data - \{\mathsf{nil}\}) \cdot hdl = h \vee hdl \in reachable(h, adts)
\end{array}
$$

Das Funktion *is-acyclic* prüft, ob die Menge aller ADT zyklenfrei ist. Dies ist der Fall, wenn für jeden ADT gilt, daß dieser ADT nicht in der Menge der von ihm selber aus erreichbaren ADT enthalten ist.

$$\mathsf{type}\ \textit{is-acyclic} : Adts \longrightarrow \mathbf{B}$$

$$is\text{-}acyclic(adts) \mathrel{\hat{=}} \forall h \in \mathsf{dom}\ adts \cdot h \notin reachable(h, adts)$$

Die Funktion *reachable* bestimmt die Menge aller ADT, die von einem ADT aus erreichbar ist.

$$\begin{array}{l} \mathsf{type}\ reachable\ \ : Handle \times Adts \longrightarrow Handle\ \mathsf{-set} \\ \mathsf{type}\ reachable_1 : Handle\ \mathsf{-set} \times Adts \longrightarrow Handle\ \mathsf{-set} \end{array}$$

$$reachable(h, adts) \mathrel{\hat{=}} reachable_1(bag\text{-}to\text{-}set\,(adts(h).elems), adts)$$

$$\begin{array}{l} reachable_1(hs, adts) \mathrel{\hat{=}} \\ \quad \mathsf{let}\ hs_1 = hs \cup \mathsf{union}\ \{bag\text{-}to\text{-}set\,(adts(h).elems) \mid h \in hs\}\ \ \mathsf{in} \\ \quad\quad \mathsf{if}\ hs = hs_1 \\ \quad\quad \mathsf{then}\ hs \\ \quad\quad \mathsf{else}\ reachable_1(hs_1, adts) \end{array}$$

Die Funktion *is-init-vars* prüft, ob alle als Parameter übergebenen Variablen existieren und initialisiert sind, d.h. es ist ihnen ein ADT zugeordnet.

$$\mathsf{type}\ is\text{-}init\text{-}vars\ : Data \times Var\ \mathsf{-bag} \longrightarrow \mathbf{B}$$

$$is\text{-}init\text{-}vars\,(d, vars) \mathrel{\hat{=}} \forall v \in vars \cdot v \in \mathsf{dom}\ data \wedge d(v) \neq \mathsf{nil}$$

Die Funktion *new-handle* gibt einen bisher noch nicht verwendeten Handle als Resultat zurück.

$$\mathsf{type}\ new\text{-}handle\ : Adts \longrightarrow Handle$$

$$\mathsf{post\text{-}}\ new\text{-}handle\,(adts, hdl) \mathrel{\hat{=}} hdl \notin \mathsf{dom}\ adts$$

Die Funktion *inc-ref-cnt* inkrementiert die Referenzzähler der durch durch den Parameter *elems* ausgewählten ADT.

$$\mathsf{type}\ inc\text{-}ref\text{-}cnt\ : Adts \times Elements \longrightarrow Adts$$

$$\mathsf{pre\text{-}}\ inc\text{-}ref\text{-}cnt\,(adts, elems) \mathrel{\hat{=}} \forall h \in elems \cdot h \in \mathsf{dom}\ adts$$

$$\begin{array}{l} inc\text{-}ref\text{-}cnt\,(a, elems) \mathrel{\hat{=}} \\ \quad [h \mapsto mk\text{-}adt\,(a(h).rc + occurs(h, elems), a(h).elems) \mid h \in \mathsf{dom}\ a] \end{array}$$

Die Funktion *dec-ref-cnt* dekrementiert die Referenzzähler der durch die Parameter *elems* ausgewählten ADT. Erreicht hierbei der Referenzzähler eines ADT den Wert 0, so wird dieser ADT gelöscht (er wird garantiert nicht mehr benötigt). Die Referenzzähler der Elemente des gelöschten ADT müssen nun ebenfalls dekrementiert werden, da ja jeweils ein Verweis auf diese Elemente entfernt wurde.

$$\mathsf{type}\ \textit{dec-ref-cnt} : Adts \times Elements \longrightarrow Adts$$

$$\mathsf{pre\text{-}}\,\textit{dec-ref-cnt}\,(adts, elems) \mathrel{\widehat{=}} \forall h \in elems \cdot h \in \mathsf{dom}\ adts$$

$$\begin{array}{l}
\textit{dec-ref-cnt}\,(adts, elems) \mathrel{\widehat{=}} \\
\quad \mathsf{if}\ elems = \{\,\} \\
\quad \mathsf{then}\ adts \\
\quad \mathsf{else}\ \mathsf{let}\ h \in elems\ \ \mathsf{in} \\
\qquad\quad \mathsf{let}\ rc_1 = adts(h).rc - 1\ \ \mathsf{in} \\
\qquad\qquad \mathsf{if}\ rc_1 = 0 \\
\qquad\qquad \mathsf{then}\ \textit{dec-rec-cnt}\,(adts - \{h\}, elems - \{h\} \cup adts(h).elems) \\
\qquad\qquad \mathsf{else}\ \textit{dec-ref-cnt}\,(adts + [h \mapsto \textit{mk-adt}\,(rc_1, adts(h).elems)], \\
\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad elems - \{h\})
\end{array}$$

Die Funktion *cond-clone* erzeugt eine Kopie des ADT, auf den *var* verweist, wenn mehr als ein Verweis auf diesen ADT vorhanden ist. Es wird jedoch lediglich eine *flache Kopie (shallow copy)* erzeugt; dies entspricht somit der Vorgehensweise der in der VDM Class Library für alle Typen bereitgestellten Operation `copy1`. Nach dem Kopieren verweist allein *var* auf die erzeugte Kopie.

$$\mathsf{type}\ \textit{cond-clone} : Data \times Adts \times Var \longrightarrow Data \times Adts$$

$$\mathsf{pre\text{-}}\,\textit{cond-clone}\,(data, adts, var) \mathrel{\widehat{=}} var \in \mathsf{dom}\ data \wedge data(var) \neq \mathsf{nil}$$

$$\begin{array}{l}
\textit{cond-clone}\,(data, adts, var) \mathrel{\widehat{=}} \\
\quad \mathsf{let}\ hdl = data(var)\ \ \mathsf{in} \\
\qquad \mathsf{if}\ adts(hdl).rc = 1 \\
\qquad \mathsf{then}\ (data, adts) \\
\qquad \mathsf{else}\ \mathsf{let}\ hdl_1 = \textit{new-handle}\,(adts)\ \ \mathsf{in} \\
\qquad\qquad \mathsf{let}\ elems = adts(hdl).elems\ \ \mathsf{in} \\
\qquad\qquad \mathsf{let}\ data_1 = data + [var \mapsto hdl_1]\ \ \mathsf{in} \\
\qquad\qquad \mathsf{let}\ adts_1 = adts + [hdl \mapsto \textit{mk-adt}\,(adts(hdl).rc - 1, elems)]\ \ \mathsf{in} \\
\qquad\qquad \mathsf{let}\ adts_2 = adts_1 \cup [hdl_1 \mapsto \textit{mk-adt}\,(1, elems)]\ \ \mathsf{in} \\
\qquad\qquad\quad (data_1, \textit{inc-ref-cnt}\,(adts_2, elems))
\end{array}$$

5.1.10 Modell und Programm

In diesem Abschnitt soll durch ein Beispiel verdeutlicht werden, wie ein C++-Programm mit Hilfe der hier vorgestellten Operationen im Modell dargestellt werden kann. Das Beispiel verwendet die durch folgende DSL-Spezifikation definierten abstrakten Datentypen:

```
DOMAINS
  Nbag = Nat0 -bag
  Bbag = Nbag -bag
IMPLEMENTATIONS
  Nbag = automem(BASEIMPL=...) + mk1 + union1
  Bbag = automem(BASEIMPL=...) + mk1 + card + union
```

Dem C++-Programm ist hier die entsprechende Sequenz von Operationen auf den Zustand *Program* gegenübergestellt.

```
void proc1(BBAG l1);
main()
{
  NBAG n1 = NBAG(42);
  BBAG b1 = BBAG(n1);
  BBAG b2 = b1;

  b1.sel();
  n1.plus1(4242);
  proc1(b1);
  b1.plus(b2);
  b2 = b1;
}
void proc1(BBAG l1)
{
 cout << l1.card();
}
```

$constructor(n1, \{\})$
$constructor(b1, \{n1\})$
$copy\text{-}constr(b2, b1)$

$selector(temp, b1)$
$destructor(temp)$
$update(n1, \{\}, union1\text{-}bag\text{-}op)$
$copy\text{-}constr(l1, b1)$
$attribut(\{l1\})$
$destructor(l1)$
$update(b1, \{b2\}, union\text{-}bag\text{-}op)$
$assign(b2, b1)$

$destructor(b2)$
$destructor(b1)$
$destructor(n1)$

5.2 Erweiterung des VDM-Compilers

In Abschnitt 4.3 wurde eine Transformation der vom VDM-Compiler intern erzeugten Repräsentation einer DSL-Spezifikation beschrieben. Diese soll auf einfache Weise die Auswahl der automatischen Speicherverwaltung/Wertsemantik für alle ADT einer DSL-Spezifikation durch das Setzen eines Kommandozeilenparameters beim Aufruf des VDM-Compilers ermöglichen. Diese Transformation soll hier nun genau spezifiziert werden.

Die interne Repräsentation einer DSL-Spezifikation im VDM-Compiler wird durch folgende abstrakte Syntax beschrieben (siehe [KS93, Kapitel 5.1.1]):

$$
\begin{array}{rcl}
Dominfo & = & \textit{Domain-Name} \xrightarrow[\mathrm{m}]{} Description \\
Description & :: & domain : Domain \\
 & & impl : Implementation \\
Domain & :: & constr : \textit{Type-Constr-Name} \\
 & & elems : Elems \\
 & & depends : Depend \\
Elems & = & \textit{Domain-Name}\ * \\
Depend & = & \textit{Domain-Name}\ \textsf{-set} \\
Implementation & :: & implid : Implid \\
 & & params : Params \\
 & & operations : Operations \\
Implid & = & [\textit{Implementation-Name}] \\
Params & = & \textit{Param-Name} \xrightarrow[\mathrm{m}]{} \textit{Param-Value} \\
Operations & = & \textit{Operation-Name} \xrightarrow[\mathrm{m}]{} Params \\
Str & = & Char\ *
\end{array}
$$

Alle verwendeten Typen, deren Name auf *-Name* endet, sowie *Param-Value*, sind Synonyme für den Typ *Str*. Sie wurden lediglich eingeführt, um die Verständlichkeit zu erhöhen.

Der Parser des VDM-Compilers erzeugt aus der gelesenen DSL-Spezifikation einen Wert des Typs *Dominfo*; dieser Wert muß also transformiert werden, um das beschriebene Ziel zu erreichen. Hierzu wird die Funktion *automem-transform* verwendet.

$$
\begin{array}{l}
\textsf{type}\ \textit{automem-transform} : Dominfo \longrightarrow Dominfo \\
\textit{automem-transform}(di)\ \widehat{=} \\
\quad [dn \mapsto \textit{automem-trans}_1(di(dn)) \mid dn \in \textsf{dom}\ di]
\end{array}
$$

Es darf jedoch nicht jeder Wert aus dem Bildbereich der vom Parser gelieferten Abbildung transformiert werden. In Abhängigkeit der gewählten Implementation (*Implid*) müssen folgende Fälle unterschieden werden:

- es wurde eine Implementation gewählt, für die eine Transformation nicht notwendig ist; dies trifft für alle Implementationen der elementaren Datentypen[5] sowie für alle `automem`-Implementationen zu
- es wurde eine Implementation gewählt, für die eine Transformation in eine `automem`-Implementation durchgeführt werden kann
- es wurde eine Implementation gewählt, für die die automatische Speicherverwaltung/Wertsemantik nicht mit Hilfe einer `automem`-Implementation realisiert werden kann; dies trifft z.B. auf die Implementationen `heap` und `extern` des Typgenerators Map zu; es darf keine Transformation durchgeführt werden
- es wurde keine Implementation angegeben; dies wird z.B. verwendet, um für Typen, die in anderen Dateien definiert wurden, weitere Operationen anzufordern; eine Transformation ist hier nicht notwendig

Der letzte Fall kann leicht erkannt werden, da hier *Implid* den Wert **nil** hat. Um die anderen Fälle unterscheiden zu können, wird ein Eintrag in der Datei *typgen*/*impl*/`basic/param` der Datenbank der VDM Class Library verwendet. Eine solche Datei ist für jede Implementation und jeden Typgenerator vorhanden. In dieser Datei werden eigentlich die Parameter aufgezählt, die in der Implementationsbeschreibung der DSL-Spezifikation an die Implementation *impl* übergeben werden dürfen; sie wird jedoch bereits für ähnliche Aufgaben „mißbraucht“ (so z.B. durch den Eintrag `FORWARD_ALLOWED`).

Ist in dieser Datei die Stringkonstante OK_AUTOMEM[6] eingetragen, so darf für diese Implementation eine Transformation durchgeführt werden; ist die Stringkonstante NO_AUTOMEM eingetragen, so darf für diese Implementation keine Transformation durchgeführt werden. Ist keine der beiden Stringkonstanten vorhanden, so ist eine Transformation nicht notwendig.

Diese Fallunterscheidung wird von der Funktion $\mathit{automem\text{-}trans}_1$ unter Verwendung einiger Hilfsfunktionen durchgeführt.

[5] die elementaren Datentypen werden alle durch C-Datentypen, die nicht auf Pointern basieren, implementiert; Wertsemantik sowie automatische Speicherfreigabe sind deshalb standardmäßig vorhanden

[6] der Wert dieser Konstante ist hier nicht weiter interessant; dies gilt auch für NO-AUTOMEM

$$
\begin{array}{ll}
\mathsf{type}\ \mathit{automem\text{-}trans}_1 & : \mathit{Description} \longrightarrow \mathit{Description} \\
\mathsf{type}\ \mathit{ok\text{-}automem} & : \mathit{Domain} \times \mathit{Implementation} \longrightarrow \mathbf{B} \\
\mathsf{type}\ \mathit{no\text{-}automem} & : \mathit{Domain} \times \mathit{Implementation} \longrightarrow \mathbf{B} \\
\mathsf{type}\ \mathit{get\text{-}params} & : \mathit{Str} \times \mathit{Str} \times \mathit{Str} \longrightarrow \mathit{Params}
\end{array}
$$

$$
\begin{array}{l}
\mathit{automem\text{-}trans}_1(\mathit{descr}) \ \widehat{=} \\
\quad \mathsf{let}\ \mathit{mk\text{-}description}(\mathit{dom}, \mathit{impl}) = \mathit{descr}\ \ \mathsf{in} \\
\quad \mathsf{let}\ \mathit{impl}_1 = (\mathit{impl.implid} = \mathsf{nil} \rightarrow \mathit{impl} \\
\qquad\qquad\quad , \mathit{ok\text{-}automem}(\mathit{dom}, \mathit{impl}) \rightarrow \mathit{mk\text{-}automem\text{-}impl}(\mathit{impl}) \\
\qquad\qquad\quad , \mathit{no\text{-}automem}(\mathit{dom}, \mathit{impl}) \rightarrow \mathit{impl} \\
\qquad\qquad\quad , \mathsf{true} \rightarrow \mathit{impl} \\
\qquad\qquad\quad)\ \ \mathsf{in} \\
\qquad \mathit{mk\text{-}description}(\mathit{dom}, \mathit{impl}_1)
\end{array}
$$

$$
\begin{array}{l}
\mathsf{pre\text{-}}\ \mathit{ok\text{-}automem}(\mathrm{d,i}) \ \widehat{=}\ \mathit{i.implid} \neq \mathsf{nil} \\
\mathit{ok\text{-}automem}(d, i) \ \widehat{=} \\
\quad \mathrm{OK\text{-}AUTOMEM} \in \mathsf{dom}\ \mathit{get\text{-}params}(\mathit{d.constr}, \mathit{i.implid}, \text{“}\mathtt{basic}\text{``})
\end{array}
$$

$$
\begin{array}{l}
\mathsf{pre\text{-}}\ \mathit{no\text{-}automem}(\mathrm{d,i}) \ \widehat{=}\ \mathit{i.implid} \neq \mathsf{nil} \\
\mathit{no\text{-}automem}(d, i) \ \widehat{=} \\
\quad \mathrm{NO\text{-}AUTOMEM} \in \mathsf{dom}\ \mathit{get\text{-}params}(\mathit{d.constr}, \mathit{i.implid}, \text{“}\mathtt{basic}\text{``})
\end{array}
$$

Die Funktion *get-params* dient zum Zugriff auf die oben beschriebene `param`-Datei; sie wird bereits vom VDM-Compiler bereitsgestellt und soll deshalb hier nicht weiter definiert werden. In der Codierung der Transformation sollte für den Fall, daß in der `param`-Datei die Stringkonstante NO-AUTOMEM eingetragen und somit eine Transformation nicht erlaubt ist, zusätzlich eine Warnung für den Anwender ausgegeben werden.

Die eigentliche Transformation einer Implementationsbeschreibung wird von der Funktion *mk-automem-impl* durchgeführt, wenn in der entsprechenden `param`-Datei die Stringkonstante OK-AUTOMEM gefunden wurde.

$$
\begin{array}{ll}
\mathsf{type}\ \mathit{mk\text{-}automem\text{-}impl} & : \mathit{Implementation} \longrightarrow \mathit{Implementation} \\
\mathsf{type}\ \mathit{params\text{-}to\text{-}str} & : \mathit{Params} \longrightarrow \mathit{Str}
\end{array}
$$

$$
\begin{array}{l}
\mathsf{pre\text{-}}\ \mathit{mk\text{-}automem\text{-}impl}(\mathrm{i}) \ \widehat{=}\ \mathit{i.implid} \neq \mathsf{nil} \\
\mathit{mk\text{-}automem\text{-}impl}(i) \ \widehat{=} \\
\quad \mathsf{let}\ \mathit{par\text{-}str} = \mathit{params\text{-}to\text{-}str}(\mathit{i.params})\ \ \mathsf{in} \\
\quad \mathsf{let}\ \mathit{baseimpl\text{-}pval} = \mathit{i.implid} \frown < \text{“}\mathtt{(}\text{“} > \frown \mathit{par\text{-}str} \frown < \text{“}\mathtt{)}\text{“} >\ \ \mathsf{in} \\
\qquad \mathit{mk\text{-}Implementation}(\text{“}\mathtt{automem}\text{“}, \\
\qquad\qquad\qquad [\text{“}\mathtt{BASEIMPL}\text{“} \mapsto \mathit{baseimpl\text{-}pval}], \mathit{i.operations})
\end{array}
$$

$params\text{-}to\text{-}str(p) \;\hat{=}$
  if $p = [\,]$
  then $<>$
  else let $pn \in \mathsf{dom}\; p$ in
    let $str =<$ "`"`" $> \frown pn \frown <$ "`"="`" $> \frown p(pn) \frown <$ "`"`" $>$ in
      if $p - \{pn\} = [\,]$
      then str
      else $str \frown <$ "`,`" $> \frown param\text{-}to\text{-}str(p - \{pn\})$)

Ein Beispiel soll die hier durchgeführte Transformation noch einmal verdeutlichen. Für die DSL-Spezifikation

```
DOMAINS
   String = Char *
IMPLEMENTATIONS
   Tuple = array(NOCHECK,ISCHAR) + hd + tl
```

wird vom Parser des VDM-Compilers der folgende *Dominfo*-Wert geliefert:

["`String`" $\mapsto$
  mk-Description(
    mk-Domain("`tupl`",<"`Char`">,{}),
    mk-Implementation(
      "`array`",
      ["`ISCHAR`" $\mapsto$ "`1`","`NOCHECK`" $\mapsto$ "`1`"],
      ["`hd`" $\mapsto$ [ ], "`tl`" $\mapsto$ [ ]]
)
)
]

Wenn in der Datei `tupl/array/basic/param` die Stringkonstante OK-AUTOMEM eingetragen ist, wird die Funktion *automem-transform* hierfür den folgenden Wert als Ergebnis liefern:

["`String`" $\mapsto$
  mk-Description(
    mk-Domain("`tupl`",<"`Char`">,{}),
    mk-Implementation(
      "`automem`",
      ["`BASEIMPL`" $\mapsto$ "`array(``NOCHECK``=``1``,``ISCHAR-``1``)`"],
      ["`hd`" $\mapsto$ [ ], "`tl`" $\mapsto$ [ ]]
)
)
]

Dies entspricht genau dem Wert, der vom Parser des VDM-Compilers für die DSL-Spezifikation

```
DOMAINS
   String = Char *
IMPLEMENTATIONS
   Tuple = automem(BASEIMPL="array(NOCHECK,ISCHAR)")
         + hd + tl
```

geliefert wird.

6 Implementierung

Whenever possible, steal code.
— Tom Duff, Bell Labs

Dieses Kapitel beschreibt die Implementierung der Operationen der `automem`-Implementation sowie der Erweiterung des VDM-Compilers.

6.1 Datenstrukturen der `automem`-Implementation

Zur Realisierung der für die Implementation `automem` benötigten Datenstrukturen wird eine Technik verwendet, bei welcher das konkrete Objekt — der ADT mit automatischer Speicherverwaltung/Wertsemantik — in zwei Teile zerlegt wird[1]:

- einen *Handle*, der die Benutzerschnittstelle bereitstellt
- eine *Repräsentation*, die den größten Teil oder sogar den ganzen Objektzustand enthält

In diesem Fall ist die Repräsentation der intern verwendete Hilfs-ADT mit manueller Speicherverwaltung und Referenzsemantik. Die Verbindung zwischen dem Handle und der Repräsentation wird in der Regel durch einen Zeiger hergestellt[2]. Neben dem Verweis auf die Repräsentation enthält der Handle hier nur noch den Referenzzähler, der die Anzahl der Verweise auf den ADT in der `automem`-Implementation angibt. In Abb. 5 sind die aus dieser Vorgehensweise folgenden Datenstrukturen dargestellt (`Domain` sei hierbei der vom Anwender in der DSL-Spezifikation gewählte Name des ADT).

Diese Datenstrukturen werden durch folgende DSL-Spezifikation erzeugt, die sich jeweils in der Datei *typgen*`/automem/basic/vdm`[3] der Datenbank der VDM Class Library befindet (hier etwas verkürzt wiedergegeben):

[1] siehe [St92, Kap. 13.9]

[2] eine Ausnahme dieser Regel bilden der Typgenerator Union in der Implementation `union` sowie der Typgenerator Tree in der Implementation `struct`

[3] *Hinweis:* Da die meisten der im folgenden beschriebenen Definitionen unabhängig von einem speziellen Typgenerator sind, werden sie jeweils nur einmal im Verzeichnis `aux/automem` gespeichert; dies erleichtert die Wartung, die jetzt zentral an einer Stelle vorgenommen werden kann. Im `automem`-Verzeichnis des jeweiligen Typgenerators befindet sich in der jeweiligen Dateien dann nur noch eine `#include`-Anweisung, die die echte Datei aus dem `aux/automem`-Verzeichnis einbindet. Ausnahmen werden explizit gekennzeichnet.

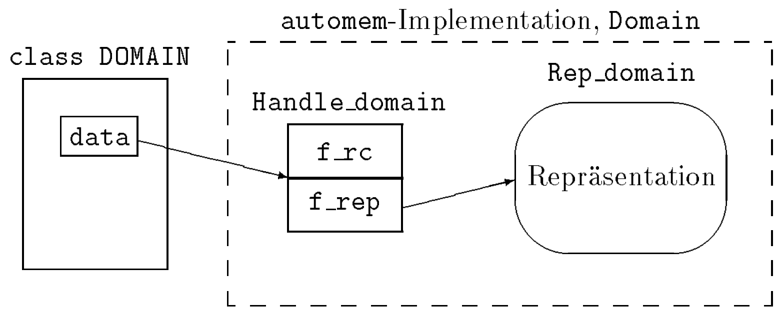


Abbildung 5: Datenstrukturen in der `automem`-Implementation

```
DOMAINS
   Rep##_DOMAIN = Wiederholung der Definition von DOMAIN
   Heap##_DOMAIN = (DOMAIN) -heap
   DOMAIN -> Rep##_DOMAIN
   DOMAIN -> Heap##_DOMAIN
IMPLEMENTATIONS
   Rep##_DOMAIN = BASEIMPL
   Heap##_DOMAIN = freelist(MAXLEN=100,PROFILE)
           + alloc(STRUCT=``struct Handle``#_DOMAIN)
```

`DOMAIN` und `_DOMAIN` sind vom VDM-Compiler definierte Parameter, die immer den Namen des gerade bearbeiteten ADT angeben (`_DOMAIN` in Kleinbuchstaben und mit führenden Unterstrich). `BASEIMPL` ist der an die Implementation `automem` übergebene Parameter, der zur Auswahl der Implementation des intern verwendeten Hilfs-ADT, der Repräsentation `Rep##_DOMAIN`, dient. Der Typgenerator `heap` stellt Operationen zur Allokation und Deallokation von dynamischen Objekten bereit; der Typ der dynamischen Objekte wird durch den Parameter `STRUCT` festgelegt, der an die angeforderte Operation `alloc` übergeben wird. Der Name `Heap##_DOMAIN` ist hier irrelevant: die erzeugten Funktionen haben die Endung `_DOMAIN` entsprechend dem gewählten Element; es wird auch kein Typ dieses Namens erzeugt.

In der Datei *typgen*`/automem/basic/gtyp` wird der Typ `struct Handle##_DOMAIN` folgendermaßen definiert:

```
START_RECORD_TYP(Handle##_DOMAIN)
   RECFIELD(f_rep,Rep##_DOMAIN);
   RECFIELD(f_rc,Nat0);
END_RECORD_TYP(Handle##_DOMAIN)
```

`f_rep` ist der Verweis auf die Repräsentation, `f_rc` der Referenzzähler. Für jeden Referenzzähler werden also vier Byte benötigt (`Nat0` entspricht dem Typ `unsigned long` in C/C++). Eine Einschränkung des Wertebereichs der Referenzzähler mit dem Ziel Speicherplatz zu sparen, z.B. durch Verwendung des Typs `Char` (benötigt nur ein Byte) anstelle von `Nat0`, ist hier nicht erfolgreich. Da auf fast allen Rechnern dynamische Objekte nur an durch 4, 8 oder 16 teilbaren Adressen erzeugt werden und der Zeiger auf die Repräsentation 4 Byte belegt, stehen aufgrund dieses Alignments also mindestens 4 Byte für den Referenzzähler zur Verfügung.

Damit auf den ADT auch unter dem vom Anwender festgelegten Namen zugegriffen kann, muß dem C/C++-Compiler noch angezeigt werden, daß ein Zeiger auf `Handle##_DOMAIN` und `DOMAIN` den selben Typ bezeichnen. Dies wird durch die Typdefinition

```
TYPE_DEF(DOMAIN,POINTER_TO_STRUCT(Handle##_DOMAIN))
```

in der Datei *typgen*`/automem/basic/gptr` erreicht.

6.2 Operationen der `automem`-Implementation

Bei der Implementierung der in 5.1 spezifizierten Operationen müssen zwei Arten von Operationen unterschieden werden:

1. Operationen, für die Funktionen in C erzeugt werden; diese Funktionen werden dann in der C++-Schnittstelle von entsprechenden Memberfunktionen aufgerufen

2. Operationen, für die *nur* Memberfunktionen in der C++-Schnittstelle erzeugt werden. Hierbei handelt es sich um Operationen, die keine Entsprechung in C haben: *destructor*, *assign*, *copy-constr* und *nil-constructor*. Diese Operationen werden außerdem nicht vom Anwender in der Implementationsbeschreibung ausgewählt, sondern immer automatisch erzeugt.

Für Operationen der ersten Art werden die einzelnen Komponenten auch in der Implementation `automem` wie gewohnt in einem Verzeichnis *typgen/impl/opname* der Datenbank gespeichert[4]. Die schon vorhandene C++-Schnittstelle dieser Operationen — eine C++-Schnittstelle gilt ja immer für alle Implementationen eines Typgenerators — kann bis auf wenige Ausnahmen unverändert übernommen werden.

[4]siehe [SH91, Kap. 5.3]

Operationen der zweiten Art müssen hiervon abweichend in einem sogenannten *Klassenkopf*[5] definiert werden, der nur einmal pro Typgenerator in der Datei *typgen*`/sign/basic/cpp` gespeichert wird. Alle hier relevanten Typgeneratoren definieren jedoch keinen eigenen Klassenkopf, sondern verwenden gemeinsam den in `com/sign/basic/cpp` definierten Klassenkopf. Die Operationen der zweiten Art können in der Implementation `automem` ebenfalls in diesem allgemeinen Klassenkopf definiert werden, da sie unabhängig von einem bestimmten Typgenerator sind. Die Unterscheidung, ob die „normalen“ Operationen oder die speziellen Operationen der Implementation `automem` erzeugt werden sollen, kann anhand des Parameters `BASEIMPL` erfolgen, der nur in der Implementation `automem` definiert ist. Somit ergibt sich für diesen Klassenkopf folgende Struktur:

> *allgemeingültige Definitionen*
> `#if defined(BASEIMPL)`
>   *Operationen für Implementation* `automem`
> `#else`
>   *Operationen für alle anderen Implementation*
> `#endif`

Die Implementation der Operationen im Klassenkopf wird später dargestellt.

In der Spezifikation mußte nicht jede Operation der ersten Art für jeden Typgenerator einzeln spezifiziert werden, sondern alle Operationen konnten — unabhängig vom jeweiligen Typgenerator — in wenige Gruppen eingeteilt werden: Konstruktoren, Prädikate/Attribute, Selektoren und Kombinatoren. Aus Sicht der Speicherverwaltung/Wertsemantik verhalten sich alle Operationen einer Gruppe gleichartig und vereinfachen so die Spezifikation.

Auch bei der Implementierung soll zur Vereinfachung diese Einteilung in Gruppen wieder angewendet werden. Leider ist die Implementierung der Operationen einer Gruppe nicht gleich sondern nur ähnlich (Name der Funktionen, Anzahl und Typ der Parameter etc. sind unterschiedlich). Um die Implementierung trotzdem möglichst einfach zu gestalten, wurde für jede Gruppe ein Makro für den C-Präprozessor entwickelt, welches jeweils die Operationen einer Gruppe erzeugen kann; die von diesen „Operationsgeneratoren“ (OG) erzeugten Operationen werden durch Makroparameter genauer spezifiziert[6]. Alle OG sind jeweils in einem eigenem Unterverzeichnis von `aux/automem` in der Datenbank des VDM Class Library gespeichert. In den Verzeichnissen *typgen*`/automem/`*opname* werden in den

[5] siehe [WM92, 2.1.1 in Abschnitt 5]

[6] z.T. werden pro Gruppe auch zwei OG verwendet, damit die Makros nicht zu kompliziert sind

entsprechenden Dateien dann nur noch die jeweiligen Makroparameter definiert und der entsprechende OG durch eine `#include`-Anweisung eingebunden.

Diese OG sollen hier nicht weiter erläutert werden, vielmehr wird für jede Gruppe eine von dem entsprechenden OG erzeugte C-Funktion für eine Operation beispielhaft vorgestellt und die jeweiligen Eigenschaften erläutert. Eine ausführliche Beschreibung der OG und ihrer Makroparameter befindet sich in der Datenbank selbst (siehe `aux/automem/macros.README`).

Bevor jedoch mit dieser Beschreibung der Implementierung der spezifizierten Operationen begonnen werden kann, müssen zuerst noch zwei andere Operationen erläutert werden, die man hier eigentlich nicht mehr erwartet hätte: die Operationen `copy` und `del`.

6.2.1 copy und del

In Abschnitt 4.4.1 wurde beschrieben, daß die Operationen `copy` und `del` in der Implementation `automem` überflüssig sind und deshalb nicht mehr bereitgestellt werden müssen. Warum sind diese Operationen hier nun wieder interessant ?

In der C++-Schnittstelle — also in der Schnittstelle zum Anwender — sind diese Operationen tatsächlich überflüssig und brauchen daher nicht mehr bereitgestellt werden. Die zu diesen Operationen gehörenden C-Funktionen, `copy##_DOMAIN` und `del##_DOMAIN`, haben jedoch eine sehr wichtige Aufgabe: sie sind für die Verwaltung des Referenzzählers zuständig, wenn Elemente in einen ADT eingefügt bzw. aus diesem entfernt werden und wenn ein ADT gelöscht wird.

Im folgenden soll anhand zweier Beispiele gezeigt werden, wie diese Funktionen für diese Aufgabe verwendet werden und warum diese „Zweckentfremdung“ sinnvoll ist.

Entfernen von Elementen aus einen ADT
Die Anweisung `m1.diff(m2)` wird verwendet, um alle Elemente aus der Menge `m1` zu entfernen, die auch in `m2` enthalten sind (Berechnung der Differenzmenge). Wird ein Element aus der Menge `m1` entfernt, so fällt ein Verweis auf dieses Element weg und der Referenzzähler des Elementes muß dekrementiert werden.

Hier stößt man nun auf ein Problem: die C-Funktion `diff_menge` der `automem`-Implementation, die von der Memberfunktion `MENGE::diff` in der C++-Schnittstelle aufgerufen wird, kennt die Elemente, die entfernt werden, nicht. Denn zur Durchführung der eigentlichen Operation wird ja die entsprechende C-Funktion der Repräsentation, hier `diff_rep_menge`, aufgerufen. In der Spezifikation wurde dieses bei allen Kombinatoren auftretende Problem durch die „mystische“, als Parameter übergebene Funktion *op* gelöst, die genau die Elemente bestimmt, die in den bearbeiteten ADT eingefügt bzw. aus diesem entfernt werden.

Ein solche Funktion könnte man hier nachbilden: für jedes Element der Menge `m1` wird geprüft, ob es auch in `m2` enthalten ist und wenn dies der Fall ist, dekrementiert man den Referenzzähler dieses Elementes. Diese Vorgehensweise ist jedoch sehr ineffizient, denn man führt hier im Prinzip nochmal genau die Aufgabe durch, die auch die aufgerufene Funktion der Repräsentation durchführt.

Hier kommt nun die Operation `del` ins Spiel. Denn für alle Kombinatoroperationen[7], die Elemente in einen ADT einfügen oder aus diesem entfernen, existiert auch eine entsprechende Operation, welche die gleiche Funktion ausführt, aber zusätzlich für alle nicht mehr benötigten Elemente implizit die Funktion `del()` aufruft; diese Operationen mit impliziten Löschen sind durch ein an den normalen Namen angehängtes '`d`' gekennzeichnet. Die nicht mehr benötigten Elemente sind im Falle der Operation `diff` genau die Elemente, die aus `m1` entfernt werden. Wenn nun anstelle der Funktion `diff_rep_menge` die entsprechende Funktion mit implizitem Löschen `diffd_rep_menge` für die Repräsentation aufgerufen wird, so kann die für die entfernten Elemente aufgerufene Funktion `del_element` zum Dekrementieren des Referenzzählers dieser Elemente verwendet werden:

```
void del_element(Element x1)
{
  if ((x1!=0) && (--x1->f_rc==0))
  {
    del_rep_element(x1->f_rep);
    free_element(x1);
  }
}
```

Der Referenzzähler des entfernten Elementes wird durch diese Funktion dekrementiert und, falls dieser hierbei den Wert 0 erreicht, wird das Element — bestehend aus Repräsentation und Handle — gelöscht.

Ein weiterer Vorteil dieser Vorgehensweise wird hier deutlich. Wenn die Repräsentation des Elementes durch Aufruf der Funktion `del_rep_element` gelöscht wird, wird für alle Elemente des gelöschten Elementes ebenfalls die entsprechende Operation `del` aufgerufen und somit der Referenzzähler dieser Elemente, wie es dies Spezifikation vorschreibt, ebenfalls dekrementiert[8].

Einfügen von Elementen in einen ADT

Beim Einfügen von Elementen in einen ADT treten ähnliche Probleme auf. Die

[7]eine Ausnahme bilden Kombinatoren, die Elemente in einen Tuple einfügen

[8]siehe 5.1.9, Funktion *dec-ref-cnt*

Anweisung `m1.union(m2)` wird verwendet, um alle Elemente, die in `m2` aber nicht in `m1` enthalten sind, in die Menge `m1` einzufügen (Berechnung der Vereinigungsmenge). Wird ein Element aus `m2` in `m1` eingefügt, so entsteht ein neuer Verweis auf dieses Element und der Referenzzähler des Elementes muß inkrementiert werden.

Auch hier müssen jetzt die Elemente, die tatsächlich in `m1` aufgenommen werden, bestimmt werden, denn nur deren Referenzzähler darf inkrementiert werden. Hierzu wird zunächst eine Kopie der Repräsentation der Menge `m2` durch Aufruf der Funktion `copy_rep_menge` erzeugt — diese Kopie ist aufgrund der zu realisierenden Wertsemantik sowieso notwendig, da die aufgerufene `union`-Operation sonst die Repräsentation von `m2` zerstören würde. Diese Funktion ruft nun wiederum für jedes Element die Funktion `copy_element` auf, welche zum Inkrementieren des Referenzzählers verwendet wird:

```
Element copy_element(Element x1)
{
  check_handle_element(x1);
  x1->f_rc++;
  return x1;
}
```

Durch das Erzeugen dieser Kopie werden also die Referenzzähler *aller* Elemente von `m2` inkrementiert. Dies ist natürlich noch nicht korrekt; die Referenzzähler der Elemente, die nicht in `m1` aufgenommen werden, müssen wieder dekrementiert werden. Die erreicht man, indem zur Ausführung der eigentlichen Operation wieder die entsprechende Funktion mit impliziten Löschen, hier `uniond_rep_menge`, für die Repräsentation aufgerufen wird. Diese ruft für alle Elemente von `m2`, die nicht in `m1` aufgenommen werden, die Funktion `del_element` auf und dekrementiert so deren Referenzzähler wieder.

Durch das Zusammenspiel der copy- und del-Funktionen wird hier also das gewünschte Ziel erreicht: genau für die Elemente aus `m2`, die auch in `m1` aufgenommen werden, wird der Referenzzähler um eins erhöht.

Weitere Vorteile

Diese Verwendung der `copy`- und `del`-Funktionen für die Verwaltung des Referenzzählers hat jedoch noch weitere Vorteile:

- elementare Datentypen (`Bool`, `Nat0` etc.) als Elemente eines ADT müssen nicht als Sonderfall behandelt werden, obwohl für solche Elemente ja kein Referenzzähler verwaltet wird (siehe 5.1.1). Denn auch für diese Typen stehen die Operationen `copy` und `del` zur Verfügung. Da die entsprechenden C-Funktionen keine echten Aktionen ausführen (für `del` wird eine leere Funktion erzeugt; die für `copy` erzeugte Funktion gibt lediglich den als Parameter übergebenen Wert zurück), entsteht durch diese eigentlich unnötigen Funktionsaufrufe auch kein Overhead, denn jeder gute C-Compiler kann diese „wegoptimieren“.

- ADT mit manueller Speicherverwaltung / Referenzsemantik können ohne Probleme als Elemente eines ADT mit automatischer Speicherverwaltung/Wertsemantik verwendet werden. Aus Gründen der Effizienz sollte dies aber vermieden werden. Im Gegensatz zu der hier vorgestellten `copy`- und `del`-Operationen der Implementation `automem`, die nur den Referenzzähler manipulieren, werden in anderen Implementationen durch diese Operationen echte Kopien erzeugt und auch wieder gelöscht. Dies ist natürlich entsprechend aufwendiger.

Zusammenfassung
Die sich aus der Verwendung der Operationen `del` und `copy` zur Verwaltung des Referenzzählers ergebende Vorgehensweise für die Implementierung sei hier nocheinmal kurz zusammengefaßt:

- für jede `automem`-Implementation werden die Operationen `del` und `copy` als *C-Funktionen* entsprechend den vorhergehenden Beispielen bereitgestellt
- eine C++-Schnittstelle für die Operationen `del` und `copy` wird nicht erzeugt
- bevor ein Element in einen ADT eingefügt wird, muß es durch Aufruf der Operation `copy` „kopiert“ werden, um den Referenzzähler zu inkrementieren
- bei der Implementierung der Kombinatoren der Implementation `automem` wird für die Repräsentation immer die entsprechende Operation mit impliziten Löschen aufgerufen; wird ein Element aus dem ADT entfernt bzw. ein vorher kopiertes nicht eingetragen, so wird der Referenzzähler des Elementes durch die aufgerufene Operation `del` dekrementiert.

Für die Beispiele, der in den folgenden Abschnitten vorgestellten Operationen, werden die ADT

```
DOMAINS
  Map = Nat0 -m-> Elem
  Elem = Nat0 -set
IMPLEMENTATIONS
  Map = automem(BASEIMPL=...) + ...
  Elem = automem(BASEIMPL=...) + ...
```

verwendet.

6.2.2 Default-Konstruktor

Der Default-Konstruktor[9] ist die Implementierung der Operation *nil-constructor* aus der Spezifikation und dient zur Erzeugung nicht-initialisierter Variablen, d.h. Variablen denen kein ADT zugeordnet ist.

```
MAP::MAP() { data = 0; }
```

Der Wert 0 repräsentiert hier den Wert `nil` aus der Spezifikation.

Um zu prüfen, ob eine Variable initialisiert ist, kann das Makro `check_handle##_DOMAIN`[10] benutzt werden. Ist die geprüfte Variable nicht initialisiert, so wird das Programm mit einer Fehlermeldung abgebrochen.

Der Default-Konstruktor wird im Klassenkopf definiert.

6.2.3 Konstruktoren

Die für Konstruktoren (Operation *constructor* in der Spezifikation) erzeugten C-Funktionen haben folgenden prinzipiellen Aufbau:

- erzeugen eines neuen Handle, Referenzzähler mit 1 initialisieren
- jedes in den neu erzeugten ADT einzufügende Element kopieren
- aufrufen des entsprechenden Konstruktors der Repräsentation

Beispiel: Konstruktor `mk1` für den ADT Map

```
Map mk1_map(Nat0 x1, Elem x2)
{
  Map l = alloc_map(); // erzeugt neuen Handle
  l->f_rc = 1;
  x1 = copy_nat0(x1);
  x2 = copy_elem(x2); // inkrementiert Referenzzähler
  l->f_rep = mk1_rep_map(x1,x2);
  return l;
}
```

Konstruktoren werden durch die Operationsgeneratoren `mk` und `mk_special` erzeugt. In der C++-Schnittstelle werden die Konstruktoren durch echte C++-Konstruktoren realisiert.

[9] zur Bedeutung des Default-Konstruktors, siehe [WM92, 4.4 in Abschnitt 3, S. 53f]

[10] wird an die Implementation `automem` der Parameter `NOCHECK` übergeben, so wird diese Prüfung nicht durchgeführt; die erzeugten Makros sind in diesem Fall nur Dummies

6.2.4 Attribute und Prädikate

Die für Prädikate und Attribute (Operation *attribut* in der Spezifikation) erzeugten C-Funktionen haben folgenden prinzipiellen Aufbau:

- prüfen, ob die Operation für nicht-initialisierte Variablen aufgerufen wurde
- aufrufen der entsprechenden Funktion der Repräsentation

Weitere Aktionen sind nicht notwendig, da die aufgerufenen Funktionen den Zustand der Repräsentation nicht verändern. Beispiel: Prädikat `is_submap` für den ADT Map

```
Bool is_submap_map(Map x1, Map x2)
{
   check_handle_map(x1);
   check_handle_map(x2);
   return is_submap_rep_map(x1->f_rep,x2->f_rep);
}
```

6.2.5 Selektoren

Die für Selektoren (Operation *selector* in der Spezifikation) erzeugten C-Funktionen haben folgenden prinzipiellen Aufbau:

- prüfen, ob die Operation für nicht-initialisierte Variable aufgerufen wurde
- aufrufen der entsprechenden Funktion der Repräsentation
- kopieren des selektierten Elementes und Rückgabe als Funktionswert

Beispiel: Selektor `sel` für den ADT Map

```
Elem sel_map(Map x1, Nat0 x2)
{
   check_handle_map(x1);
   // copy inkrementiert Referenzzähler von Elem
   return copy_elem(sel_rep_map(x1->f_rep,x2));
}
```

Die Umwandlung des von dieser C-Funktion gelieferten C-Datentyps, hier `Elem`, in den entsprechenden C++-Klassentyp, hier `ELEM`, erfolgt durch einen immer erzeugten Konvertierungskonstruktor[11].

Eine Besonderheit stellen die Operationen `is_in_dom`, `is_in_rng`, `is_in_min` und `is_in_dom` für den Typgenerator Map dar, bei denen es sich um eine Kombination aus Selektor und Prädikat handelt. Die erzeugten Funktionen liefern ein Boolschen Wert als Funktionsergebnis (Prädikat) und, wenn das Prädikat erfüllt ist, ein Element der Map über einen Referenzparameter (Selektor).

Obwohl bei Verwendung der Implementation `automem` eigentlich eine Wertsemantik realisiert werden soll, werden diese Funktionen mit Referenzparameter aus Gründen der Effizienz hier unterstützt. Im Prinzip sind diese Operationen jedoch überflüssig; das gleiche Ergebnis läßt sich auch durch vorhandene echte Prädikate und Selektoren erreichen, nur muß die Map dann zweimal durchsucht werden.

In der Spezifikation wurden diese Operationen nicht explizit aufgeführt, da sie sich durch eine Sequenz elementarer Operationen darstellen lassen. Der Operation *is-in-dom* (*elem*, *map*, *ref-elem*) entspricht z.B die Sequenz

$$attribut(\{elem, map\})$$
$$selector(temp, map)$$
$$assign(\textit{ref-elem}, temp)$$
$$destructor(temp)$$

Diese Funktionen werden durch den OG `map_is_in_xxx` erzeugt.

Im Gegensatz zu anderen Implementationen ist der Wert des Referenzparameters hier nicht undefiniert, wenn das Prädikat nicht erfüllt ist, sondern dieser hat den definierten Wert nicht-initialisiert.

6.2.6 Kombinatoren

Die für Kombinatoren (Operation *update* in der Spezifikation) erzeugten C-Funktionen haben folgenden prinzipiellen Aufbau:

- prüfen, ob die Operation für nicht-initialisierte Variablen aufgerufen wurde
- Kopie des zu verändernden ADT erzeugen, wenn mehr als eine Referenz auf dieses ADT vorhanden ist; nur diese Kopie wird verändert
- eine Kopie der Elemente, die in den ADT eingefügt oder durch die aufgerufene Funktion zerstört werden, erzeugen

[11] siehe [WM92, 3 in Kap. 3, S.35ff.]

- die entsprechende Funktion der Repräsentation mit impliziten Löschen aufrufen

Beispiel: Operation `union` für den ADT Map

```
Map union_map(Map x1, Map x2)
{
  check_handle_map(x1);
  check_handle_map(x2);
  cond_clone_map(&x1); // erzeugt Kopie, falls f_rc>1
  lrep2 = copy_rep_map(x2->f_rep);
  x1->f_rep = uniond_rep_map(x1->f_rep,lrep2);
  return x1;
}
```

Kombinatoren werden durch die OG `update_D` und `update_DD` erzeugt.

Beim Vergleich mit der Spezifikation der Operation *update* fällt auf, daß die Vorbedingung hier nicht vollständig überprüft wird. Die erzeugten Funktionen können nämlich ohne Probleme auch mit solchen Parametern aufgerufen werden, die zu Zyklen führen und somit die Invariante verletzen. Der Grund für diese „Nachlässigkeit“ ist, wie schon so oft, die Effizienz; die notwendige Prüfung wäre hier extrem aufwendig. Die Verantwortung für die Vermeidung von Zyklen liegt also in den Händen des Anwenders[12].

Für Kombintoroperationen, die neben den normalen Memberfunktionen auch Operatoren unterstützen, müssen in der Implementation `automem` Infix-Operatoren, die die Operanden nicht verändern, sowie Zuweisungsoperatoren definiert werden[13]. Dies erfordert eine Erweiterung der C++-Schnittstelle dieser Operationen.

Die Definition der Infix- und Zuweisungsoperatoren wird hier am Beispiel des Kombinators `union` für den ADT Map vorgestellt:

```
MAP MAP::operator | (const MAP& x2) const
{
  MAP temp = *this; // Kopier-Konstruktor, siehe 6.2.8
  temp.data = union_map(temp.data,x2.data);
  return temp;
}
MAP& MAP::operator |= (const MAP& x2)
```

[12] dort liegt sie ja bereits bei Verwendung der manuellen Speicherverwaltung, siehe 4.2.3

[13] siehe hierzu 4.4.2

```
{
  data = union_map(data,x2.data);
  return *this;
}
```

Bei Verwendung des Infix-Operators wird also nur der Wert einer temporären Variablen verändert und als Funktionsresultat zurückgegeben. Die Definition des Zuweisungsoperators entspricht der Definition des Infix-Operators in den anderen Implementationen.

Da die C++-Schnittstelle einer Operation immer für alle Implementationen eines Typgenerators gültig ist, müssen die Definitionen der Operatoren für die Implementation `automem` und die der anderen Implementationen zusammen in einer Datei gespeichert werden. Die Unterscheidung, welche Operatoren nun erzeugt werden sollen, kann anhand des Parameters `BASEIMPL` erfolgen, der nur bei gewählter Implementation `automem` definiert ist. Für die C++-Schnittstelle der betroffenen Kombinatoren ergibt sich somit die folgende Struktur:

```
Definition der Member- bzw. Friendfunktionen
#ifdef(OPERATOR)
#if defined(BASEIMPL)
  Infix-/Zuweisungsoperator für Implementation automem
#else
  Infix-Operator für andere Implementationen
#endif
#endif
```

6.2.7 Destruktor

Die Operation *destructor* aus der Spezifikation wird durch den Destruktor der in der C++-Schnittstelle erzeugten Klassen implementiert:

```
MAP::~MAP() { del_map(data); }
```

Die Verwendung der Funktion `del` zur Dekrementation des Referenzzählers wurde bereits in 6.2.1 erläutert. Der Destruktor ist im Klassenkopf definiert.

6.2.8 Kopier-Konstruktor

Die Operation *copy-constr* aus der Spezifikation wird durch den Kopier-Konstruktor der in der C++-Schnittstelle erzeugten Klassen implementiert:

```
MAP::MAP(const MAP& x1)
{
  check_handle_map(x1);
  data = x1.data;
  data->f_rc++;
}
```

Der Kopierkonstruktor wird im Klassenkopf definiert.

6.2.9 Zuweisung

Die Operation `assign` aus der Spezifikation wird durch Überladen des Zuweisungsoperators der in der C++-Schnittstelle erzeugten Klassen implementiert:

```
MAP& MAP::operator =(const MAP& x1)
{
  check_handle_map(x1);
  (x1.data)->f_rc++;
  if (data!=0) del_map(data);
  data = x1.data;
  return *this;
}
```

Die Zuweisungsoperator wird im Klassenkopf definiert.

6.3 Der ADT Str

Zwei Besonderheiten des ADT Str führen dazu, daß im Zusammenhang mit der Implementation `automem` Probleme auftreten:

- der ADT Str ist vordefiniert
- der ADT Str wird in der Datenbank zur Implementierung der Ausgabeoperationen (`fpr`, `pr`, etc.) genutzt

Da der ADT Str in der Datenbank vordefiniert ist, kann der Anwender in der DSL-Spezifikation keine Implementation für diesen ADT wählen; der Versuch führt zu einer Fehlermeldung wg. doppelter Definition des ADT Str. Die Auswahl der automatischen Speicherverwaltung und Wertsemantik durch die Implementation `automem` ist auf diese Weise also auch nicht möglich.

Zur Auswahl der automatischen Speicherverwaltung und Wertsemantik für den ADT Str wird deshalb ein anderer Weg eingeschlagen: die Definition des ADT Str in der Datei `stdlib/com/basic/vdm` der Datenbank wird in Abhängigkeit eines Parameters geändert.

```
#if defined(AUTOMEM_SEL)
   Str = automem(BASEIMPL="array(\"ISCHAR\",\"SIMPLEELEMS\",
                                  \"INTERNALDOMAIN\")")
#else
   Str = array("ISCHAR"´,"SIMPLEELEMS")
#endif
```

Ist der Parameter `AUTOMEM_SEL` definiert, so wird der ADT Str in der Implementation `automem` erzeugt, ansonsten wird die normale Implementation gewählt. Bei Aufruf des VDM-Compiler mit dem Kommandozeilenparameter zur Auswahl der automatischen Speicherverwaltung wird dieser Parameter automatisch definiert. Ansonsten muß er beim Aufruf des VDM-Compilers explizit mit Hilfe der Option `-DAUTOMEM_SEL` definiert werden.

Die Verwendung des ADT Str in der Implementation `automem` innerhalb der Datenbank führt ebenfalls zu Problemen. Denn die in diesem Fall für Str erzeugten C-Funktionen zeigen ein zum Teil vom Erwartetem abweichendes Verhalten. Dies führt jedoch bis auf zwei Ausnahmen nicht zu Fehlern, da die Elemente des ADT Str elementare C-Datentypen (`char`) sind.

Die Ausnahmen bilden die für die Kombinatoren `conc` und `conc_path` erzeugten C-Funktionen. Diese Funktionen zerstören normalerweise die beiden als Parameter übergebenen Strings. Bei der Anwendung dieser Funktionen in der Datenbank werden diese Strings deshalb auch nicht mehr durch einen expliziten Aufruf von `del` gelöscht. In der Implementation `automem` wird jedoch *nur einer* der Strings durch diese Funktionen zerstört; der zweite String bleibt also als „Speicherleiche" zurück !

Um dieses Problem zu verhindern, werden in der Implementation `automem` des ADT Str deshalb automatisch zwei Versionen der Operation `conc`- bzw. `conc_path` erzeugt:

1. `conc_str` : diese C-Funktion zerstört beide Parameter, wie dies auch in allen anderen Implementationen der Fall ist. Aufgrund ihres Namens wird diese Funktion intern in der Datenbank verwendet.

2. `conc_auto_str` : diese C-Funktion zerstört nur einen Parameter, wie es für die speziellen Anforderungen der `automem`-Implementation notwendig ist. Diese Operation wird aus der C++-Schnittstelle — also auch vom Anwenderprogramm aus — aufgerufen.

Dieses geschieht jedoch unsichtbar für den Anwender, d.h. in der Implementationsbeschreibung braucht weiterhin nur die Operation `conc` bzw. `conc_path` angefordert werden. Auch in der C++-Schnittstelle wird nur eine Memberfunktion mit diesem Namen erzeugt.

6.4 Nicht verfügbare Implementationen

Es ist leider nicht immer möglich, alle für einen bestimmten Typgenerator vorhandenen Implementationen auch als Implementation des internen Hilfs-ADT der Implementation `automem` zu verwenden. Bei folgenden Typgeneratoren treten in diesem Zusammenhang Einschränkungen auf:

- Tuple: für die Implementationen `readfile` und `writefile` sind aufgrund ihres speziellen Anwendungszweckes nur sehr wenige Operationen verfügbar; insbesondere fehlt die unbedingt benötigte Operation `copy`

- Optional: der in der Implementation `ptelem` zur Repräsentation von `nil` verwendete Wert hat in der Implementation `automem` schon eine andere Bedeutung (nämlich nicht-initialisiertes Objekt)

- Map: in den Implementationen `scanfile` und `heap` steht die benötigte Operation `copy` nicht zur Verfügung; in der Implementation `extern` sind keine Operationen definiert, diese muß der Anwender selber bereitstellen, es kann also nicht davon ausgegegangen werden, daß die benötigten Operationen `copy` und `del` auch vorhanden sind

Wie man sieht, sind nur wenige, eher exotische Implementationen nicht verfügbar; alle wichtigen Implementationen können hingegen ohne Probleme verwendet werden.

Benötigt man jedoch unbedingt eine dieser nicht unterstützten Implementationen, so besteht immer noch die Möglichkeit, diese direkt, d.h. ohne automatische Speicherverwaltung und Wertsemantik, zu realisieren. Dies ist dann zwar nicht sehr bequem, aber ohne Probleme möglich.

6.5 Nicht verfügbare Operationen

Viele Operationen stehen für einen ADT in der Implementation `automem` nicht mehr zur Verfügung, da sie dort nicht benötigt werden (siehe 4.4.1)

Wählt der Anwender in der Implementationsbeschreibung der DSL-Spezifikation aber eine solche nicht verfügbare Operation aus, können Probleme auftreten. Der VDM-Compiler sucht eine benötige Operation nämlich nicht nur im jeweiligen Verzeichnis der Implementation, sondern auch — falls er sie dort nicht findet — in mehreren `com`-Verzeichnissen [14]. Das Einbinden der Operationen aus diesen Verzeichnissen ist aber unerwünscht, da der Aufruf dieser Funktionen zu Programmfehlern führen kann. Es wäre besser, wenn bei der Auswahl einer solchen nicht unterstützten Operation eine Fehlermeldung ausgegeben wird.

Um dies zu erreichen, wird für jede nicht verfügbare Operation eine Dummy-Funktion implementiert. Diese besteht lediglich aus einer `#error`-Direktive des C-Präprozessors:

`#error === automem: Operation` *typgen*`/`*op* `not supported ===`

Die Auswahl einer nicht verfügbaren Operation führt deshalb zur erwünschten Ausgabe dieser Fehlermeldung beim Übersetzen der DSL-Spezifikation durch den VDM-Compiler.

6.6 Erweiterung des VDM-Compilers

Die Erweiterung des VDM-Compilers um die Transformation der internen Repräsentation einer DSL-Spezifikation gestaltet sich sehr einfach, da im Programm des VDM-Compilers tatsächlich die in der Spezifikation definierten ADT verwendet werden — diese wurden wiederum vom VDM-Compiler selbst erzeugt! Aus diesem Grund soll hier nicht besonders ausführlich auf die Erweiterung eingegangen werden.

Geändert wurden folgende Module (alle Änderungen sind durch `/* TT */` gekennzeichnet):

- `domgen.c` : in diesem Modul wird die eigentliche Transformation in den Funktionen `automem_transform` und `automem_impl` durchgeführt
- `domsrc.c` : Erweiterung zur Verarbeitung des neuen Kommandozeilenparameters, gewählt wurde `-M`

[14] zur Suchstrategie des VDM-Compilers siehe [SH91, Kapitel 5.4]

- `domuse.c` : Erweiterung des ausgegebenen Hilfetextes um neuen Kommandozeilenparameter
- `domfct.c` : Defintion neuer Stringkonstanten, die zur Transformation benötigt werden

Die Transformation des vom Parser erzeugten Ableitungsbaumes darf jedoch nicht für jede gelesene Eingabedatei erfolgen, sondern nur für Dateien mit der Extension `.vdm`. Für die sogenannten Enrichment-Dateien[15], die die Extension `.enr` haben, darf die Transformation hingegen nicht durchgeführt werden. Für diese Dateien erfolgte eine Transformation nämlich bereits bei ihrer Erzeugung durch den VDM-Compiler. Eine erneute Transformation würde dazu führen, daß für die schon vorhandenen Hilfs-ADT ebenfalls eine `automem`-Implementation erzeugt wird. Dies ist natürlich nicht sinnvoll.

In der Implementierung der Transformation wurde eine Erweiterung gegenüber der Spezifikation durchgeführt. Diese ist für die korrekte Funktion nicht notwendig, führt aber dazu, daß die erzeugten C++-Module kürzer und verständlicher werden. Normalerweise würde auch für den in der `automem`-Implementation verwendeten Hilfs-ADT eine eigene Klasse in der C++-Schnittstelle erzeugt werden. Um die Erzeugung dieser unnötigen Klassen zu verhindern, wird an die für den Hilfs-ADT ausgewählte Implementation zusätzlich der Parameter `INTERNALDOMAIN` übergeben.

[15] zur Anwendung der Enrichment-Dateien siehe [JM91, Kap. 3.3.3]

7 Test

All men are liable to error; and most men are,
in many points, by passion or by interest, under temptation to it.
— John Locke (1690)

Zum Test der `automem`-Implementationen kann der für die VDM Class Library vorhandene Testfallgenerator BB[1] nicht verwendet werden. Dieser Testfallgenerator erzeugt nur Testprogramme in C, ein ADT in der Implementation `automem` kann jedoch nur in C++ korrekt genutzt werden. Aus diesem Grund wurden eigene Tests entworfen:

- ein Funktionstest
- ein Spezifikationstest
- ein Test der Laufzeiteigenschaften

Diese Tests und ihre Ergebnisse sollen im folgenden vorgestellt werden. (Hinweis: alle in den folgenden Abschnitten beschriebenen Testprogramme befinden sich im Verzeichnis `Test/automem` des VDM-Systems; pro Test ist ein weiteres Unterverzeichnis vorhanden)

7.1 Funktionstest

Der Funktionstest ist als reiner Black-Box-Test realisiert. Für jeden Typgenerator, der die Implementation `automem` unterstützt, und für jede bereitgestellte Operation wird geprüft, ob bei definierten Eingabewerten der erwartete Ergebniswert geliefert wird. Weiterhin wird geprüft, ob die Parameter der Operation in Bezug auf ihre Zerstörung korrekt behandelt werden.

Bei einem normalen Black-Box-Test wird jede zu testende Operation mit einer Menge von Eingabewerten getestet. Dies ist hier nicht notwendig, da die Operationen in der Implementation `automem` diese Eingabewerte nicht verarbeiten, sondern nur an die entsprechende Operation der Repräsentation — diese sind durch den Testfallgenerator BB bereits ausführlich getestet — weiterreichen. Aus diesem Grund ist für diesen Funktionstest ein Testfall pro Operation ausreichend.

Allein der Funktionstest ist für einen vollständigen Test jedoch unzureichend, da die eigentliche Speicherverwaltung und die Wertsemantik mit verzögerter Kopie hier nicht hinreichend geprüft werden können. Hierfür ist der im nächsten Abschnitt beschriebene Spezifikationstest vorgesehen.

[1]siehe [Wi92]

7.2 Spezifikationstest

Der Name dieses Abschnittes ist hier etwas übertrieben, denn es wird kein Testprogramm vorgestellt, welches automatisch überprüft, ob die erstellte Implementation korrekt aus der Spezifikation in Abschnitt 5.1 abgeleitet wurde. Vielmehr wird ein Hilfsmittel bereitgestellt, um diese Überprüfung sinnvoll „von Hand" durchführen zu können.

Die Idee ist nun folgende: Jedes C++-Programm, welches ADT in der Implementation `automem` verwendet, kann durch eine Sequenz von Operationen auf den Zustand *Program* modelliert werden[2]; ein Beispiel wurde in Abschnitt 5.1.10 vorgestellt. Anhand der Spezifikation der Operationen kann der Tester den Wert des Zustandes *Program* nach jeder Operation berechnen. Im C++-Programm wird ebenfalls nach jeder Anweisung der Wert des Zustandes *Program* aus den aktuellen Programmdaten ermittelt. Wurde die Implementation korrekt aus der Spezifikation hergeleitet, so müssen der „von Hand" berechnete und der aus den tatsächlichen Programmdaten ermittelte Wert immer übereinstimmen.

Um auf diese Weise vorgehen zu können, wurde die Möglichkeit geschaffen, den Zustand *Program* zu jedem Zeitpunkt im C++-Programm zu ermitteln.

Um diese Testmöglichkeiten nutzen zu können, müssen Testklassen von den vom VDM-Compiler erzeugten Klassen sowie einer speziellen Hilfsklasse `CHECKED_ADT` abgeleitet werden (Mehrfachvererbung), die dann anstelle der vom VDM-Compiler erzeugten Klassen im C++-Testprogramm verwendet werden. Alle Operationen, die in den vom VDM-Compiler erzeugten Klassen bereitstehen, stehen auch in diesen Testklassen zur Verfügung. Die Testklassen sind notwendig, um zur Berechnung des Zustandes *Program* zusätzlich benötigte Informationen[3] zu verwalten. Diese Verwaltung geschieht unsichtbar für den Anwender; im Testprogramm sind keine zusätzlichen Aktivitäten durch den Tester erforderlich.

Um die Anwendung dieser Testmöglichkeit zu vereinfachen, wurde die Datei `specification_test/test_template` erstellt. In dieser Datei befindet sich neben ausführlichen Erläuterungen auch ein fertiges Programmgerüst für Testprogramme. Der Tester muß hier im Prinzip nur noch die Namen der von ihm verwendeten ADT eintragen. Als Beispiel für die Anwendung dieses Templates kann die sich im selben Verzeichnis befindende Datei `spec2_test.cc` verwendet werden.

In dem Programm `spec2_test.cc` wurde dieses Testkonzept angewendet. Dieses Testprogramm gibt die jeweils ausgeführte Anweisung, die entsprechende Operation sowie den „von Hand" berechneten und den aus den aktuellen Programmdaten ermittelten Zustand *Program* nach Ausführung der Anweisung aus.

[2] modelliert werden natürlich nur Speicherverwaltung/Wertsemantik, nicht das vollständige Verhalten des Programmes

[3] es wird eine Menge von Zeigern auf alle zum aktuellen Zeitpunkt existierenden Instanzen der von `CHECKED_ADT` abgeleiteten Klassen verwaltet

7.3 Test der Laufzeiteigenschaften

After all, if a program does not work correctly,
it does not matter how efficient it is.
— Edward Yourdon

On the other hand, we cannot ignore efficiency.
— John Bentley

Für die Anwendung der im Rahmen dieser Diplomarbeit entwickelten `automem`-Implementationen ist neben der Korrektheit der Implementierung auch die Laufzeiteffizienz dieser Implementierung von Bedeutung. Um die Laufzeiteffizienz beurteilen zu können, wurden insgesamt sechs Testprogramme einmal mit automatischer Speicherverwaltung/Wertsemantik und einmal mit manueller Speicherverwaltung/Referenzsemantik implementiert; bei Programmen mit manueller Speicherverwaltung wurde darauf geachtet, daß auch hier immer der gesamte angeforderte Speicherplatz wieder freigegeben wird.

Aus den Laufzeitunterschieden dieser sechs Programmpaare läßt sich dann der Overhead, der durch die automatische Speicherverwaltung und die Wertsemantik entsteht, ermitteln. Bevor die Ergebnisse der Messungen bekanntgegeben werden, sollen die verschiedenen Testprogramme und ihre speziellen Eigenschaften kurz vorgestellt werden. Dies ist wichtig, um die Testergebnisse richtig bewerten zu können.

Hinweise:

- die Tests wurden auf einem IBM-AT kompatiblen Rechner (386/33 Mhz/8 MB RAM) unter den Betriebssystem Linux 0.99.9 durchgeführt, als C/C++-Compiler wurde g++ 2.3.3 verwendet
- alle Testprogramme wurde mit eingeschalteter Prüfung auf nicht-initialisierte Variablen, ohne Nutzung von Makros und ohne Optimierungen durch den C/C++-Compiler übersetzt; in Bezug auf den entstehenden Overhead stellt dies also den worst-case-Fall dar
- bei Tests mit Dateizugriff befanden sich die Daten stets im Cache des Betriebssystems
- alle Testprogramme befinden sich im Unterverzeichnis `performance_test`; die Testdurchführung und -auswertung erfolgt automatisch durch das Shell/Awk-Script `perf-test.sh`

Testprogramm 1: Coarsest Partioning mit Endrekursion
Dieses Programm berechnet für beliebige nicht-leere Mengen, die nicht überschneidungsfrei sind, die „gröbste Zerlegung“ (coarsest partioning). Hierunter versteht man die Zerlegung dieser Mengen in die minimale Anzahl nicht-leerer, aber überschneidungsfreier Mengen (siehe [JM91, Kap. 5.4.4]).

Dieses Programm implementiert den Algorithmus unter Verwendung von Endrekursion. Für die Effektivität der `automem`-Implementation ist dies nachteilig, da aufgrund der Wertsemantik bei jedem rekursiven Aufruf der Funktionen Kopien der übergebenen Parameter erzeugt werden müssen (die Parameter werden in der Funktion verändert). Bei Verwendung von Referenzsemantik sind diese Kopien nicht notwendig, da die Zerstörung der übergebenen Parameter akzeptiert wird.

Testprogramm 2: Coarsest Partioning mit While-Schleife
Dieses Testprogramm entspricht dem ersten Programm; die Endrekursion wurde jedoch durch eine while-Schleife ersetzt. Die Anzahl der zu erstellenden Kopien verringert sich deshalb stark.

Testprogramm 3: Operationen auf einem binären Baum
Dieses Testprogramm trägt eine Menge von natürlichen Zahlen in einen binären Baum ein und prüft dann für jede Zahl aus dieser Menge, ob sie auch tatsächlich im Baum vorhanden ist. Der binäre Baum wird hierbei durch folgende ADT implementiert:

$$\begin{array}{rcl} Btree & = & [Node] \\ Node & :: & s\_left : Btree \\ & & s\_right : Btree \\ & & s\_info : \mathbf{N}_0 \end{array}$$

In diesem Testprogramm sind fast alle für die Repräsentation der `automem`-Implementation von Btree und Node aufgerufenen Funktionen sehr einfach (so wird z.B. die Selektoroperation des Trees lediglich durch den Zugriff auf die Komponente eines `struct` implementiert). Aus diesem Grund tritt hier der Overhead durch die in der Implemenation `automem` durchgeführten Prüfungen sowie den zusätzlichen Funktionsaufruf besonders deutlich hervor.

Testprogramm 4: Ackermann-Funktion mit Ergebnis-Cache
Dieses Testprogramm berechnet die Ackermann-Funktion

$$a(n,m) = \begin{cases} m+1, \text{ falls } n = 0 \\ a(n-1,1), \text{ falls } m = 0 \\ a(n-1, a(n, m-1)), \text{ sonst} \end{cases}$$

für $n, m \in \mathbf{N}_0$ unter Verwendung eines Ergebnis-Caches. Dieses Programm ist bereits als Beispielprogramm im VDM-System enthalten und wurde lediglich nach C++ portiert. Der Cache wird durch folgende ADT implementiert:

$$\begin{aligned} Cache &= \mathbf{N}_0 \xrightarrow[\mathrm{m}]{} Map_1 \\ Map_1 &= \mathbf{N}_0 \xrightarrow[\mathrm{m}]{} \mathbf{N}_0 \end{aligned}$$

Dieses Testprogramm demonstriert in hervorragender Weise, daß durch die Verwendung der Implementation `automem` auch ein extrem hoher Overhead entstehen kann. Das problematische Einfügen des 3-Tupel $(n, m, a(n,m))$ in den Ergebnis-Cache ist hier auszugweise dargestellt:

```
MAP map1;
if (cache.is_in_dom(n,map1))
{
  map1.ovwrt1(m,result)
  cache.ovwrt1(n,map1);
}
else ...
```

Bei Einfügen des Paares $[m \mapsto result]$ in den `map1` zugeordneten ADT muß eine Kopie dieses ADT erzeugt werden, da dieser ADT ebenfalls noch Element von `cache` ist und somit mehr als ein Verweis auf ihn vorhanden ist[4]. Wird dieser veränderte ADT nun wieder in `cache` eingetragen, so muß das hier noch vorhandene Original gelöscht werden. Da diese zu kopierenden und löschenden ADT sehr groß werden können — mehrere 100 KByte sind leicht möglich[5] —, ist dies natürlich extrem zeitaufwendig.

Testprogramm 5: Ackerman-Funktion mit Optimierung
Dieses Testprogramm entspricht im Prinzip dem Testprogramm 4. Beim Einfügen in den Cache wird jedoch ein kleiner Trick angewendet, der das oben beschriebene aufwendige Kopieren und Löschen vermeidet:

[4] siehe Spezifikation der Operation *update* in 5.1.5

[5] man berechne zum Beispiel den Wert $acker(4, 2)$ und schaue sich während der Berechnung die Speicherauslastung an; man sollte jedoch nicht auf das Ergebis warten, denn dieses ist eine 19729-stellige Zahl

```
MAP map1;
if (cache.is_in_dom(n,map1))
{
  cache -= map1;
  map1.ovwrt1(m,result)
  cache.plus1(n,map1);
}
else ...
```

Vor der Veränderung des `map1` zugeordneten ADT wird dieser zunächst aus `cache` entfernt. Dies führt dazu, daß nur noch genau *ein* Verweis auf diesen ADT existiert und somit keine Kopie beim Aufruf der Kombinatoroperation `ovwrt1` erzeugt werden muß. Bei Einfügen dieses ADT in `cache` muß auch nichts mehr gelöscht werden, da ja kein bereits vorhandenes Element überschrieben, sondern nur ein neues Element eingefügt wird.

Testprogramm 6: Profiler
Dieses letzte Testprogramm soll etwas realitätsnäher gestaltet werden. Neben der Arbeit mit den vom VDM-Compiler erzeugten ADT findet hier auch Datei-I/O statt.

Das Programm stellt den Rumpf eines Programmes zur Auswertung von Profiling-Informationen dar. Die Profiling-Information — dies sind im Augenblick nur Name der aufrufenden und der aufgerufenen Prozedur/Funktion — wird aus einer Eingabedatei gelesen und intern im Programm als Graph mit markierten Kanten gespeichert; der einer Kante zugeordnete Wert ist hier die Aufrufhäufigkeit. Dieser Graph wird durch die folgenden ADT realisiert:

$$\begin{aligned} Calls &= Procid \xrightarrow[\mathrm{m}]{} Called \\ Called &= Procid \xrightarrow[\mathrm{m}]{} \mathbf{N}_1 \\ Procs &= Procid \xrightarrow[\mathrm{m}]{} Str \\ Procid &= \mathsf{Token} \end{aligned}$$

Aus diesem Graphen können dann alle weiteren interessierenden Informationen abgeleitet werden. Im Testprogramm wird lediglich bestimmt, wie oft jede Prozedur/Funktion insgesamt aufgerufen wird, und das Ergebnis in eine Ausgabedatei geschrieben.

Testergebnisse:

Testprogram	Overhead
Coarsest Partioning (Rekursion)	55.7%
Coarsest Partioning (Schleife)	30.6%
Operationen auf binärem Baum	61.9%
Ackermann-Funktion ($a(3, 8)$)	9492.8%
Ackermann-Funktion (optimiert)	57.2%
Profiler	25.6%

Bei Verwendung der Implementation `automem` ist also im Normalfall ein Overhead zwischen 30% und 60% gegenüber einer entsprechenden Implementation ohne automatische Speicherverwaltung und Wertsemantik zu erwarten. In besonderen Fällen kann der Overhead jedoch auch extrem ansteigen, hier bis auf fast 10000%.

Bemerkenswert ist hier noch, daß der Overhead durch relativ kleine Änderungen des Programmes stark vermindert werden kann. Dies wird bei der Berechnung der Coarsest Partioning und insbesondere bei der Berechnung der Ackermann-Funktion deutlich.

8 Fazit

Die Erweiterung der VDM Class Library um eine automatische Speicherverwaltung und eine Wertsemantik konnte erfolgreich durchgeführt werden.

Bei der Programmierung mit abstrakten Datentypen in C++ stellt die automatischen Speicherverwaltung und die Wertsemantik eine große Erleichterung dar und führt zu einer erheblichen Verminderung von Fehlern. Dies konnte insbesondere bei der Erstellung der Testprogramme für die Laufzeituntersuchungen festgestellt werden, denn hier mußte ja für dieselbe Aufgabe einmal ein Programm mit und einmal ein Programm ohne automatische Speicherverwaltung und Wertsemantik erstellt werden.

Nachteilig sind jedoch die recht hohen Laufzeiteinbußen. Hier hängt es von der speziellen Aufgabe ab, ob diese aufgrund der Vorteile bei der Programmerstellung akzeptiert werden können.

In Bezug auf die Laufzeiteinbußen könnten sich Verbesserungen ergeben, wenn die automatische Speicherverwaltung nicht, wie hier geschehen, als zusätzliche Schicht zwischen C++-Schnittstelle und den eigentlichen C-Funktionen implementiert, sondern direkt in die C-Funktionen integriert wird. Dies wäre aber mit einer fast vollständigen Neuentwicklung der Datenbank verbunden. Bei der geplanten Entwicklung echter C++-Implementierungen — es wird also nicht mehr einfach eine entsprechende C-Funktion aufgerufen — sollte dieser Aspekt jedoch unbedingt beachtet werden.

A Auswahl der automatischen Speicherverwaltung/Wertsemantik

A.1 Auswahl beim Aufruf des VDM-Compilers

Beim Aufruf des VDM-Compilers muß in der Kommandozeile die Option

```
-M
```

angegeben werden. Die automatische Speicherverwaltung und Wertsemantik wird für *alle* in der Eingabedatei definierten ADT sowie den vordefinierten Typ Str gewählt. Beispiel:

```
VDM -C++ -M my_file.vdm
```

Bei Verwendung von Implementationen, die nicht korrekt mit der automatischen Speicherverwaltung/Wertsemantik zusammenarbeiten, wird eine Warnung ausgegeben und der entsprechende ADT mit manueller Speicherverwaltung und Referenzsemantik implementiert.

A.2 Auswahl in der Implementationsbeschreibung

Die Auswahl der automatischen Speichenverwaltung/Wertsemantik muß durch Auswahl der speziellen Implementation `automem` für jeden ADT einzeln erfolgen. Beispiel:

```
DOMAINS
  My_adt = ...
IMPLEMENTATIONS
  My_adt = automem(...) + Enrichment
```

Die Implementation `automem` unterstützt folgende Implementationsparameter:

`BASEIMPL` Auswahl der „echten“ Implementation des ADT; die `automem`-Implementation bildet lediglich eine Hülle um diese Operationen. Dieser Parameter *muß* immer angegeben werden.

`NOCHECK` ist dieser Parameter definiert, wird die normalerweise durchgeführte Prüfung auf die Verwendung nicht-initialisierter Variablen abgeschaltet

`MACROS` ist dieser Parameter definiert, werden Funktionen, soweit möglich, durch Makros ersetzt

Werden durch den Parameter `BASEIMPL` Implementationen ausgewählt, die nicht korrekt mit der automatischen Speicherverwaltung/Wertsemantik zusammenarbeiten, so erfolgt hier keine Warnung !

Zur Auswahl der automatischen Speicherverwaltung und Wertsemantik für den vordefinierten ADT Str muß das Symbol `AUTOMEM_SEL` mit der Option `-D` beim Aufruf des VDM-Compilers definiert werden. Beispiel:

```
VDM -C++ -DAUTOMEM_SEL my_file.vdm
```

Bei der Auswahl durch die `automem`-Implementation werden in der C++-Schnittstelle auch Klassen für intern verwendete Hilfs-ADT erzeugt. Dies führt zu einer unnötigen Vergrößerung der erzeugten Datei sowie zu einer Verschlechterung der Übersichtlichkeit. Um dies zu verhindern, kann der Implementationsparameter `INTERNALDOMAIN` an die durch `BASEIMPL` ausgewählte „echte" Implementation übergeben werden. Beispiel:

```
DOMAINS
   My_adt = Nat0 -set
IMPLEMENTATIONS
   My_adt = automem(BASEIMPL="linklist(\"INTERNALDOMAIN\")")
          + ...
```

Bei der Übergabe der Parameter an die Implementation `linklist` ist insbesondere auf die korrekte Anwendung der Anführungszeichen zu achten !

Da diese Auswahlmöglichkeit recht umständlich ist, sollte man sie nur dann verwenden, wenn zur Optimierung des Laufzeitverhaltens die gleichzeitige Verwendung von ADT mit automatischer Speicherverwaltung/Wertsemantik und ADT mit manueller Speicherverwaltung/Referenzsemantik notwendig ist.

B Unterstützte Implementationen und Operationen

Dieser Abschnitt enthält für jeden Typgenerator eine Liste der bei Verwendung der automatischen Speicherverwaltung/Wertsemantik erlaubten Implementationen sowie der zur Verfügung stehenden Operationen (die Anzeige der vorhandenen Operationen mit dem Hilfsprogramm `vdminfo` ist nicht korrekt, da viele Operationen nur als Dummy definiert sind !)

Hinweise zu den erzeugten Memberfunktionen:

- die Parameter der erzeugten Memberfunktionen haben sich gegenüber der Beschreibung in [WM92, Anhang A] nicht verändert und werden deshalb hier nicht wiederholt
- beim Aufruf einer Memberfunktion werden die an diese Funktion übergebenen Parameter grundsätzlich nicht zerstört (auf fünf Ausnahmen wird explizit hingewiesen)
- werden für eine Kombinatoroperation neben Memberfunktionen auch überladene Operatoren erzeugt, so gilt: die Operanden eines Infix-Operators werden nie zerstört; zusätzlich zu den Infix-Operatoren werden immer auch die entsprechenden Zuweisungsoperatoren erzeugt.
- nicht-initialisierte Variablen (erzeugt durch den Default-Konstruktor) dürfen nur als Referenzparameter (siehe B.3) oder auf der linken Seite eines Zuweisungsoperators verwendet werden. Eine fehlerhafte Verwendung wird erkannt und führt zum Abbruch des Programmes.
- in der aktuellen Version der Datenbank werden Funktionale und Iteratoren in C++ nicht mehr unterstützt !

B.1 Generelle Operationen

Die folgenden Operationen stehen für alle Typgeneratoren zur Verfügung:

- `fpr`, auch als Operator `<<`
- `finput`, auch als Operator `>>`
- `hash`
- `cvtotupl`
- `cvfromtupl`, *Achtung:* Das als Parameter übergebene Tuple wird durch diese Operation zerstört ! Das Tuple wird als nicht-initialisiert gekennzeichnet, so daß eine spätere fehlerhafte Verwendung erkannt wird.

B.2 Typgenerator Bag

Für den Typgenerator Bag können folgende Implementationen verwendet werden:

- `mapdom`
- `linklist`

Folgende Operationen stehen zur Verfügung:

`card`	`diff`	`inters`	`is_empty`
`is_eq`	`is_in`	`is_subbag`	`mk1`
`mk1n`	`mk_empty`	`occurs`	`sel`
`sub1`	`sub1all`	`sub1n`	`union1`
`union1n`	`union`		

B.3 Typgenerator Map

Für den Typgenerator Map können folgende Implementationen verwendet werden:

- `linklist`
- `scaldom`
- `hashtab`
- `hashlist`
- `genbt`
- `heapmap`
- `dir`

Folgende Operationen stehen zur Verfügung:

`card`	`fread`	`fwrite`	`inters`
`is_empty`	`is_eq`	`is_in`	`is_in_dom`
`is_in_max`	`is_in_min`	`is_in_rng`	`is_submap`
`max_dom`	`min_dom`	`mk1`	`mk_empty`
`ovwrt1`	`ovwrt`	`sel`	`sel_dom`
`sub1`	`sub`	`union1`	`union`

Achtung: Die Funktionen `is_in_???` verwenden einen Referenzparameter, um das jeweils in der Map gesuchte Element — soweit vorhanden — zu übergeben. Der entsprechende Parameter der Funktion wird immer verändert; wird das entsprechende Element nicht in der Map gefunden, so wird der Parameter als nicht-initialsierte Variable gekennzeichnet.

B.4 Typgenerator Optional

Nur die Implementation `elemonil` darf verwendet werden; der durch den Parameter `NILVAL` ausgewählte Wert für `nil` muß ungleich 0 sein !

Folgende Operationen stehen zur Verfügung:

`is`	`is_eq`	`is_gr`	`is_nil`
`mk_opt`	`mk_nil`	`s`	

B.5 Typgenerator Set

Für den Typgenerator Set können folgende Implementationen verwendet werden:

- `bitmap`
- `hashlist`
- `linklist`

Folgende Operationen stehen zur Verfügung:

`card`	`diff`	`inters`	`is_empty`
`is_eq`	`is_in`	`is_subset`	`mk1`
`mk_empty`	`sel`	`sub1`	`union1`
`union`			

B.6 Typgenerator Tree

Für den Typgenerator Tree können folgende Implementationen verwendet werden:

- `struct`
- `ptstruct`
- `small`

Folgende Operationen stehen zur Verfügung:

`chg`	`is_eq`	`is_ge`	`is_gr`
`mk`	`s`	`subst`	

B.7 Typgenerator Tuple

Für den Typgenerator Tuple können folgende Implementationen verwendet werden:

- `array`
- `linklist`
- `larray`

Die Implementation `bitset` wird durch einen elementaren C-Typ realisiert, daher ist die Wertsemantik sowie die automatische Speicherverwaltung schon standardmäßig vorhanden !

Folgende Operationen stehen zur Verfügung:

appl	appr	conc	conc-path
cpc	fapp	fread	fwrite
hd	index	insert	is_empty
is_eq	is_ge	is_gr	last
len	mk1	mk2	mk3
mk_empty	mk_subtup	mkc	mklc
mkn	ovwrt1	puts	rindex
rmv1	sel	sub_tup	tl

B.8 Typgenerator Union

Für den Typgenerator Union können folgende Implementationen verwendet werden:

- `union`
- `ptunion`

Die Implementation `scalbin` wird durch einen elementaren C-Typ realisiert, daher ist die Wertsemantik sowie die automatische Speicherverwaltung schon standardmäßig vorhanden !

Folgende Operationen stehen zur Verfügung:

chg	discr	is	is_eq
is_ge	is_gr	mk	ovwrt
s			

Literaturverzeichnis

Books are a triviality. Life alone is great.
— *Thomas Carlyle*

[AI91] Derek Andrews, Darrel Ince (1991), *Practical Formal Methods with VDM*, McGraw-Hill

[St92] Bjarne Stroustrup (1992), *Die C++ Programmiersprache, 2. überarbeitete Auflage*, Addison-Wesley

[Ap91] Andrew W. Appel (1991), *Garbage Collection*, erschienen in : Peter Lee (Hrsg.), *Topics in Advanced Language Implementation*, The MIT Press

[De91] David L. Detlefs (1991), *Concurrent Garbage Collection in C++*, erschienen in : Peter Lee (Hrsg.), *Topics in Advanced Language Implementation*, The MIT Press

[Kn73] Donald E. Knuth (1973), *The Art of Computer Programming, Vol. 1: Fundamental Algorithms*, Addison-Wesley

[Me90] Bertrand Meyer (1990), *Objektorientierte Softwareentwicklung*, Prentice-Hall International/Verlag Carl Hanser

[GJ89] C. Ghezzi/M. Jazayeri (1989),*Konzepte der Programmiersprachen — Begriffliche Grundlagen, Analyse und Bewertung*, R. Oldenbourg Verlag

[SH91] Uwe Schmidt/Hans-Martin Hörcher (1991), *The VDM Class Library Generator Reference Guide*, FH Wedel/Deutsche System Technik, Kiel

[SH90] Uwe Schmidt/Hans-Martin Hörcher (1990), *Programming with VDÄOMAINS*, Norsk Data Gmbh, Kiel

[SB93] Michael Stal/Wolfgang Berger (1993), *Wettstreit der Bibliomanen — C++-Klassenbibliotheken: Booch Components 1.4.7, Tools.h++ 5.1, USL Standard Components 2.0, NIHCL 3.0*, erschienen in iX 12/93, Heise-Verlag, Hannover

[JM91] Jan Peter Mohr (1991), *Programmierung mit abstrakten Datentypen unter Verwendung von VDM*, FH Wedel

[WM92] Wiebcke Mohrdieck (1992), *Entwurf und Realisierung einer C++-Schnittstelle für eine Klassenbibliothek von Implementierungen zur Unterstützung der abstrakten Datentypen aus der formalen Software-Entwicklungsmethode VDM*, Diplomarbeit FH Wedel

[KS93] Kai Steinau (1993), *Entwicklung eines Generators zur Erzeugung von Quellcodemodulen und C++-Klassen aus einer abstrakten Syntax mit der Software-Entwicklungsmethode VDM*, Diplomarbeit FH Wedel

[Wi92] Matthias Winzer (1992), *Entwurf und Implementation eines Testfallgenerators zur systematischen Erzeugung von Testfällen aus der Definition abstrakter Datentypen am Beispiel der Spezifikationsmethode VDM*, Diplomarbeit FH Wedel

Danksagung

Ich möchte meinen Dank aussprechen an

- *Prof. Dr. Uwe Schmidt* : für die sehr gute und (hoffentlich) erfolgreiche Betreuung der Diplomarbeit
- *allen Entwicklern der freien Betriebssystems Linux* : für eine hervorragende Arbeitsumgebung
- *Emma* : für den ganzen Rest

Eidesstattliche Erklärung

Ich erkläre hiermit an Eides Statt, daß ich die vorliegende Arbeit selbständig und ohne Benutzung anderer als der angegebenen Hilfsmittel angefertigt habe; die aus fremden Quellen direkt oder indirekt übernommenen Gedanken sind als solche kenntlich gemacht.

Die Arbeit wurde bisher in gleicher oder ähnlicher Form keiner anderen Prüfungskommission vorgelegt und auch nicht veröffentlicht.

Ort, Datum Unterschrift