FACHHOCHSCHULE WEDEL

Diplomarbeit

in der Fachrichtung

Technische Informatik

Thema:

Erweiterung des DSL-Compilers
der Spezifikationsmethode VDM
um eine ADT-Klasse «Bag»

Eingereicht von:	Jens Wulf Meister Francke Str. 7 W2000 Hamburg 60 Tel. (040) 630 21 45 Matrikelnr. 8327
Erarbeitet im:	8. Semester
Abgegeben am:	10. September 1996
Referent (FH Wedel):	Prof. Dr. Uwe Schmidt

Inhaltsverzeichnis

Abkürzungsverzeichnis **ii**

1 Einführung **1**

2 Aufgabenstellung **4**

2.1 Anforderungen und Teilaufgaben 5

2.2 Ablauf der Diplomarbeit 6

3 DSL-Syntax und VDM-Compiler **7**

3.1 DSL-Syntax 7

3.1.1 Typdefinitionen 7

Einbindung des neuen ADT in die DSL-Syntax 9

Wahl des ADT-Namens 10

3.1.2 Implementationsbeschreibungen 10

3.1.3 Namenskonventionen 11

3.2 VDM-Compiler 12

3.3 Aufbau der VDM-Datenbank 13

3.4 Suchstragegien 16

4 Definition des ADT Bag **17**

4.1 Generelle Operationen 17

4.2 Von Set übernommene Operationen 19

4.2.1 Konstruktoren 19

4.2.2 Kombinatoren 19

4.2.3 Attribute und Prädikate 20

4.2.4 Iteratoren 20

4.2.5 Funktionale 21

4.3 Neue Operationen 21

5 Implementation des ADT Bag **23**

5.1 linklist 23

5.1.1 Referenzimplementierung 23

5.1.2 Eigenschaften 24

5.1.3 Nachteile 24

5.2 mapdom 25

5.2.1 Grundgedanke 25

5.2.2 Realisierung 25

5.2.3 Referenzzähler 27

5.2.4 Typdefinition 27

6 Implementierung der Operationen 28

6.1 Implementationsunabhängige Operationen 29

6.1.1 Elementare Operationen 29

6.1.2 Unveränderte Set-Operationen 30

6.1.3 Konstruktoren 31

6.1.4 Destruktoren 31

6.1.5 Kombinatoren 32

6.1.6 Attribute und Prädikate 34

6.2 linklist-Implementierungen 35

6.3 mapdom-Implementierungen 36

6.3.1 Konstruktoren und Destruktoren 36

6.3.2 Iterator 38

6.3.3 Kombinatoren 39

6.3.4 Attribute und Prädikate 40

6.4 C++ Schnittstelle 42

7 Testgenerator 44

7.1 Test von mk1n und occurs 44

7.2 Test der Vereinigungsoperation 45

7.3 Test der Durchschnittsoperation 45

7.4 Test der Differenzoperation 46

7.5 Änderung der Löschmakros 46

8 Fazit 48

Anhang 50

A Bag-Operationen 50

A.1 Konstruktoren und Destruktoren 51

A.2 Attribute und Prädikate 51

A.3 Kombinatoren 52

A.4 Funktionale und Iteratoren 52

A.5 Konvertierungs- und Druckoperationen 53

B Bag-Implementation 54

B.1 DSL-Domaingleichung 54

B.2 Implementierungen 54

Literaturverzeichnis 57

Danksagung 59

Eidesstattliche Erklärung 61

Abkürzungsverzeichnis

ADT ... Abstrakt(e) Datentyp(en)
DSL .... Domain Specification Language
FH ..... Fachhochschule
IBM .... International Business Machines Inc.
N_0 .... Menge der natürlichen Zahlen inkl. 0
N_1 .... Menge der natürlichen Zahlen
Bool ... Boolean, Wahrheitswerte
VDM ... Vienna Development Method

Kapitel 1

Einführung

1974 wurde im Wiener IBM Labor unter Mitwirkung der Forscher **Denis Bjørner** und **Cliff B. Jones** die Programmiersprache **PL/1** formal mit Hilfe der funktionalen Sprache **META-IV** definiert. Die dabei angewandte formale Spezifikationsmethode bekam den Namen **Vienna Development Method** (VDM).

Inzwischen wird VDM nicht nur bei der Definition von Programmiersprachen, sondern allgemein zur Erstellung von Softwaresystemen (Programme, Datenbanken usw.) eingesetzt.

Die grundsätzliche Idee von VDM ist die Verwendung von **abstrakten Datentypen** (ADT) und Prozeduren bzw. Funktionen, die mit Hilfe von Typgleichungen, Invarianten (bei den ADT), Signaturen, Vor- und Nachbedingungen (bei den Routinen) definiert werden, ohne daß Überlegungen über die Implementierung gemacht werden.

In einem zweiten Schritt werden dann die Algorithmen der Routinen formal definiert, so daß sich ihre Korrektheit mit Hilfe der Invarianten, Vor- und Nachbedingungen (im Prinzip auch maschinell) beweisen läßt. Erst nach dieser Verifizierung werden die Programme codiert, wobei natürlich möglichst Programmgeneratoren verwendet werden sollten, um hier nicht Fehler einzubauen.

Da hier nicht weiter auf die Anwendung von VDM und der Arbeit mit ADT eingehen möchte, verweise ich auf die Literatur, insbesondere auf [BJ78] und [JM91].

An der Fachhochschule Wedel existiert seit 1990 ein Tool, mit dessen Hilfe die Typgleichungen der ADT einer VDM-Spezifikation durch eine **Domain Specification Language** (DSL) in Typgleichungen einer „normalen“ Programmiersprache wie z.B. **C** oder **PASCAL** übersetzt werden können. Außerdem wird die vom Anwender gewählte Implementierung des ADT, d.h. dessen Datenstruktur, sowie die (benötigten) elementaren Operationen (Funktionen und Prozeduren) auf den ADT in der jeweiligen Programmiersprache aus einer Bibliothek (Datenbank) bereitgestellt.

Dies ist der **VDM-Compiler**. Die Namensgebung mag zunächst verwirren, da die Eingabesprache DSL und nicht VDM heißt. VDM ist aber zunächst eine formale und abstrakte Entwicklungsmethode und keine Sprache! Die DSL dient nur dazu, die VDM-Spezifikation mit Hilfe dieses Codegenerators in eine „übliche“ Programmiersprache zu übersetzen, und zwar unabhängig von der gewählten Zielsprache! Auch hier möchte ich zur weiteren Vertiefung auf die Literatur ([JM91] und [SH90]) verweisen.

Die verfügbaren ADT sind[1)]:

- vordefinierte elementare Datentypen Char, Nat0, Nat1, Intg und Bool (**ELEMENTARY**)
- selbstdefinierte elementare Datentypen (Unterbereiche der vordefinierten elementaren Datentypen und der Sonderfall der Token[2)])
- Str
- Set
- Tuple

[1)] Vor dieser Diplomarbeit!

[2)] siehe [JM91, Seite 55]

- Map
- Optional
- Tree
- Union

Zur Zeit wird hauptsächlich **C** (Kernigham & Ritchie, ANSI) und **C++** unterstützt, aber auch **PASCAL**- und **MODULA-2**-Schnittstellen können generiert werden.

Kapitel 2

Aufgabenstellung

Zu den bereits bekannten ADT, die vom VDM-Compiler generiert werden können, ist ein neuer Typgenerator hinzuzufügen, dessen generierte ADT folgende Eigenschaften besitzen:

- Sie speichern mehrere Objekte.
- Diese Objekte sind alle vom gleichen Typ.
- Die Anzahl der Objekte ist unbeschränkt[1], aber nicht unendlich.
- Die Anzahl der Objekte kann auch Null sein.
- Die Objekte sind nicht geordnet gespeichert.
- Jedes Objekt kann mehrfach gespeichert sein.

(Ändert man den letzten Punkt in „Jedes Objekt ist höchsten einmal vorhanden“, hat man die Definition des ADT **Set**[2].)

---

[1] Abgesehen von physikalischen Grenzen wie Speicher- bzw. Plattengröße

[2] siehe [JM91, Seite 57]

2.1 Anforderungen und Teilaufgaben

Folgende Anforderungen und Teilaufgaben sind dabei zu erfüllen:

- Begründete Wahl der
  - ADT-Operationen
  - ADT-Implementierungen
  - DSL-Syntax
- Die Änderungen am VDM-Compiler sind auf ein Minimum zu beschränken, d.h. es soll möglichst nur die Datenbank erweitert werden. Dies erfordert:
  - Analyse der Datenbankstruktur
  - Analyse der Vorgehensweise des VDM-Compilers
- Die in der Implementation der FH Wedel unterstützten Zielsprachen sind auch durch diesen ADT zu unterstützen:
  - Kernigham & Ritchie **C**
  - **ANSI-C**
  - **C++**

  Die Unterstützung von **PASCAL** und **MODULA-2** ist während der Diplomarbeit festzulegen und soll im Rahmen des bisher implementierten Anteils bleiben.
- Bestehender Code ist soweit als möglich wiederzuverwenden.
- Der ADT ist in das Testsystem (siehe [Wi92]) einzufügen.
- Die Online-Dokumentation (siehe [JM91] und [SH91]) ist zu ergänzen.

2.2 Ablauf der Diplomarbeit

Der Ablauf der Diplomarbeit stellt sich dann wie folgt dar:

1. Es wird der Aufbau der Datenbank und der Ablauf einer Übersetzung durch den VDM-Compiler dargestellt und die DSL-Syntax ausgewählt (Kapitel 3).

2. Im Kapitel 4 wird dann der ADT definiert, d.h. es werden seine Operationen definiert.

3. Danach werden die hier die zu realisierenden Implementationen des ADT Bag festgelegt (siehe Kapitel 5).

4. Die Implementierung der Operationen erfolgt im 6. Abschnitt. Dabei wird auch die C++ Schnittstelle realisiert.

5. Anschließend wird der ADT in das Testsystem (siehe [Wi92]) eingebunden und der geschriebene Code überprüft (Kapitel 7).

6. Abschließend folgt in Kapitel 8 ein Fazit über die Diplomarbeit.

Kapitel 3

DSL-Syntax und VDM-Compiler

3.1 DSL-Syntax

Ich möchte hier nicht auf alle Details der DSL-Syntax und ihrer Anwendung bei einer VDM-Spezifikation eingehen, da in [JM91] (Kapitel 4 und Anhang A) als auch in [SH91] (Kapitel 3 und Anhang A) sich vollständige Dokumentationen befinden.

Eine DSL-Spezifikation besteht an den beiden Anteilen *Typdefinition*, eingeleitet durch das Schlüsselwort **DOMAINS**, und der *Implementierungsbeschreibung*, eingeleitet durch das Schlüsselwort **IMPLEMENTATIONS**.

3.1.1 Typdefinitionen

Die Typdefinitionen sind die Typgleichungen der VDM-Spezifikation in einer maschinenlesbaren Form. Die dabei definierten ADT nennt man *Domains*, die von ihnen gespeicherten Objekte *Elemente* oder *Komponenten*. (Ausnahme: Elementare Domains haben natürlich keine Komponeneten!). Die Syntax dieser Typgleichungen sind soweit als möglich der Syntax der VDM-Typgleichungen angepaßt mit folgenden Ausnahmen:

- Anonyme Typen sind nicht erlaubt. Diese entstehen bei z.B. geschachtelten Typdefinitionen, bei denen die Komponenten eines Domains selbst eine Typdefinition enthalten:

$$Werteliste = (Real \times Real) \quad -\mathbf{set}$$

muß ersetzt werden durch:

$$\begin{aligned} Werteliste &= Paar \quad -\mathbf{set} \\ Paar &:: \\ &s\_X : Real \\ &s\_Y : Real \end{aligned}$$

- Alle Komponenten eines Trees müssen einen Namen haben. Bei einem Tree ist in VDM nur die Angabe des Typs der Komponente notwendig, wenn im folgenden keine explizite Bezugnahme auf die Treekomponente mittels des Punktoperators erfolgt, sondern dies durch die Position bei der Angabe eines Trees in expliziter Notation geschieht:

$$Point :: Real \times Real$$
$$\mathbf{type} \quad swap : Point \to Point$$
$$swap(p) \quad \widehat{=} \quad \mathbf{let} \ (\ x \ , \ y \ ) \ = p \ \ \mathbf{in} \ (\ y \ , \ x \ )$$

In diesem Fall sind $(\ x \ , \ y \ )$ bzw. $(\ y \ , \ x \ )$ die Tree-Objekte mit anonymen Komponenten.)

- Bei Unions ist dies nicht erforderlich, aber sinnvoll, da möglicherweise mehrere Union-Domains definiert werden und dann sich die generierten (siehe 3.1.3) Komponentennamen nicht unterscheiden! (Siehe auch [JM91, Seite 43].)

Fazit:

> Die DSL-Syntax erlaubt keine anonyme Typen, sodaß der Typ der Elemente, die im neuen ADT gespeichert werden sollen, eindeutig bekannt ist.

Dies ist wichtig, da dadurch

- die Konstruktion der Datenstruktur des neuen ADT in der Zielsprache einfach(er) ist
- der Zugriff auf Elementoperationen aufgrund der Namenskonventionen (siehe 3.1.3) nur so möglich ist (Die Elemente sind schließlich ebenfalls ADT, auf die nur die definierten Operationen angewendet werden sollten, auch wenn die Zielsprache (z.B. **C**) (leider) auch den direkten Zugriff erlaubt)
- dem VDM-Compiler die gewünschte Implementierung des Element- ADT mitgeteilt werden kann (siehe 3.1.2) und er die Elementoperationen codieren kann

Einbindung des neuen ADT in die DSL-Syntax

Es ist nun zu überlegen, wie der neue ADT in die bestehende Syntax einzubinden ist.

> Forderung: Der neue ADT hat ähnliche Eigenschaften wie eine Menge, die mit dem ADT **Set** dargestellt werden kann. Daher sollte die Syntax der Typdefinition ähnlich zu **Set** sein:

$$\text{Adtdomain} \quad = \quad \text{Elementdomain} \quad \textbf{-adtname}$$

Dies ist in sofern eine günstige Wahl, da, wie in [SH91, Seite 17] nachzulesen ist, mit dieser Syntax benutzerdefinierte ADT konstruiert werden können, da

≪*adtname*≫ kein Schlüsselwort (wie **set**) zu sein braucht. Der entsprechende Name wird als Suchschlüssel in der Datenbasis (siehe 3.2) verwendet.

Fazit: Die Änderung der DSL-Syntax ist hier nicht notwendig.

Wahl des ADT-Namens

Als ADT-Name kam aufgrund der Funktion die Namen ≪*multiset*≫ und ≪*bag*≫ in Betracht, wobei schließlich der Name ≪*bag*≫ gewählt wurde[1)].

3.1.2 Implementationsbeschreibungen

Die Implementationsbeschreibungen geben an, welche Implementation für den jeweiligen ADT gewählt wird (z.B. *linklist, array, hashlist*) und welche Operationen auf diesen ADT benötigt werden.

Dieser Teil der DSL-Syntax wird für BAG unverändert übernommen, da

1. der Name der Implementierung mit dem entsprechendem Datenbasiseintrag übereinstimmt (siehe 3.2)

2. neue Namen relativ einfach hinzugefügt werden können, da diese keine Schlüsselwörter sind

3. die Anforderung der BAG-Operationen mit der bestehenden Syntax erfolgen kann, da auch hier die Namen Einträge in der Datenbasis und keine Schlüsselwörter sind

4. und es daher hier keinen Grund für eine Modifizierung der Syntax gibt

[1)]Dieser Name wird mehrmals in [SH91] erwähnt, und auch BORLAND verwendet ihn für eine seine Containerklassen

3.1.3 Namenskonventionen

Da in den Domaingleichungen (Typdefinitionen) ein ADT meist für mehrere Domains mit unterschiedlichen Elementen definiert werden, müssen die Operationsnamen erweitert werden, da z.B. ein Konstruktor füer einen Set für natürliche Zahlen mit dem Domainnamen *NSet* und einem Set für Strings (*Strset*) unterschiedliche Operationen sind, aber den gleichen generischen Namen **mk_one** besitzen und auch durch (bei gleicher Implementation des Domains, siehe 3.1.2) den gleichen Algorithmus implementiert werden. (Siehe dazu [JM91, Seite 25ff].) Daher wird an den generischen Namen

1. ein Unterstrich und

2. der Name des Domains in Kleinbuchstaben

angefügt. Im obigen Beispiel würden die Konstruktoren die Namen **mk_one_nset** und **mk_one_strset** bekommen. Die Elementoperationen bekämen die Endung **_nat0** bzw. **_str**.

Erst in neueren Sprachen (z.B. **C++**) darf man mehrere Funktionen mit gleichen Namen definieren, sofern sie sich in ihrer Signatur (Parametertypen und Ergebnistyp(en) [letzteres nicht in **C++**!]) unterscheiden. Das gleiche gilt für Mitglieder verschiedener Klassen. In diesem Zusammenhang möchte ich auf [WM92] hinweisen, wo dies am Beispiel von VDM und seiner **C++**-Schnittstelle erläutert wird. (Intern wird bei den Compilern dieser Sprachen allerdings der gleiche Trick verwendet, wobei meist die Erweiterung aus der Signatur (durch eine Verschlüsselung) gebildet wird. Dies bleibt aber dem Benutzer (meist[2]) verborgen!)

[2] Versuchen Sie mal in **Assembler** eine solche Funktion zu codieren oder auch nur aufzurufen, wenn Sie z.B. zeitkritische Module in **Assembler** und den Rest in **C++** programmieren!

3.2 VDM-Compiler

Der VDM-Compiler ist, wie in [SH91] und [JM91] dargestellt, ein Compiler, der Sourcecode generiert.

Er übersetzt dabei die DSL-Typ-Spezifikation in die Typdefinitionen der Zielsprache und generiert Code für die ADT-Operationen, die in der DSL-Implementierungs-Spezifikation angeforderten ADT-Operationen.

Um diesen Sourcecode generieren zu können, muß dieser in irgendeiner Form vorliegen, und zwar so, daß beliebige ADT Elemente von anderen ADT sein können, und daß ein ADT für mehrere (unterschiedliche) Domains verwendet werden kann. Dies gelingt erstens durch die Namenskonvention (siehe 3.1.3) und zweitens durch die Verwendung von Sourcecodemustern[3)], die in einer Datenbasis (synonym Datenbank) gespeichert sind.

Der Compiler verwendet als Musterbezeichner:

DOMAIN Dies ist der Bezeichner für den ADT-Domain

ELEM1 Bezeichner für das (erste) Element des ADT

ELEM2 dito für das zweite Element des ADT, wenn vorhanden

Ein Beispiel:

```
DOMAINS
   Iset = Intg -set
   Amap = Nat0 -m-> Iset

IMPLEMENTATION
   Iset = ...
   Amap = ...
```

[3)]neudeutsch auch Templates genannt

- Bearbeitet der VDM-Compiler den Domain `Iset`, so enthält **DOMAIN** den Namen „Iset“ und **ELEM1** den Namen „Intg“, während **ELEM2** undefiniert ist.
- Bei der Bearbeitung des Domains `Amap` enthält **DOMAIN** den Namen „Amap“, **ELEM1** den Namen „Nat0“ und **ELEM2** den Namen „Iset“.

Im Mustertext wird also mit Hilfe dieser drei Bezeichner Bezug genommen auf den ADT und seine Elemente, und damit kann der Compiler die jeweiligen Operationen gemäß der Namenskonvention generieren.

Den Mustertext entnimmt der Compiler einer Datenbank, die aus einem hierarchisch aufgebauten Verzeichnissystem unter **UNIX** besteht. Aufgrund ihrer Struktur und deren Auswertung durch den Compiler ist es möglich, neue ADT in das System aufzunehmen, ohne den Compiler ändern zu müssen!

3.3 Aufbau der VDM-Datenbank

Laut [SH91, Kapitel 5] kan die Datenbasis wie folgt modelliert werden:

$$\begin{array}{lll}
Vdmdb & = Adtname & \xrightarrow[\mathrm{m}]{} Adt \\
Adt & = Implname & \xrightarrow[\mathrm{m}]{} Impl \\
Impl & = Opname & \xrightarrow[\mathrm{m}]{} Operations \\
Operation & = Cmpname & \xrightarrow[\mathrm{m}]{} Component \\
Adtname & = Str & \\
Implname & = Str & \\
Opname & = Str & \\
Cmpname & = Str & \\
Component & = Str* &
\end{array}$$

VDMdb ist die Datenbasis, *Adt* ein ADT, *Impl* eine ADT-Implementation und *Operation* eine ADT-Operation, die alle UNIX-Verzeichnisse sind.

Alle ADT werden im *Vdmdb*-Verzeichnis unter ihrem Namen gespeichert, was im Fall von Bag bedeutet, hier den Namen „bag“[4)] zu verwenden.

Einige Einträge in *Vdmdb* haben eine Sonderbedeutung:

„aux“ enthält gemeinsame Einträge, die mittels einer *#include*-Präprozessoranweisung eingebunden werden. (Dazu später mehr!)

„stdlib“ enthält die vordefinierten Domains wie z.B. *Nat0* oder *Bool*.

„vdmlib“ enthält Hilfs- und Fehlerroutinen sowie betriebssystemspezifische Funktionen und Headerdateien

„com“ wird weiter unten erklärt!

Im *Adt*-Verzeichnis — hier also das dem *Adtnamen* „bag“ zugeordnete Verzeichnis — befinden sich die Implementationen des ADT. (Um einen „kleinen“ Vorgriff auf 5 zu machen: Die Einträge haben beim ADT Bag die Namen „linklist“ bzw. „mapdom“)

Auch hier gibt es spezielle Einträge:

„aux“ hat die gleiche Bedeutung wie oben und wurde bei Bag nicht verwendet.

„com“ enthält die Operationen des ADT (hier Bag), die unabhängig von der Implementierung funktionieren.

„sign“ enthält die Signaturen und die Basisdokumentation der Operationen, die spezifisch für den ADT sind. (Also nicht für die Operationen, die für alle ADT definiert sind. (s.u.))

[4)]In Kleinbuchstaben, da der Konstruktor in der DSL-Syntax ebenfalls klein geschrieben ist, denn sonst findet der Compiler den ADT nicht!

Jetzt kann man auch den Spezial-„ADT“-Namen „com“ erklären: In ihm gibt es ebenfalls „Implementationsnamen“ „com“ und „sign“, die die gleiche Aufgabe wie oben beschrieben haben, aber diesmal für die Operationen gelten, die für alle ADT definiert sind, z.B. für die Operation **copy**. Wenn diese Operationen abhängig sind von der Implementierung, wird die ADT-unabhängige Operation auf eine Fehlerroutine abgebildet!

Im *Impl*-Verzeichnis sind die Operationen, die in der entsprechenden Implementierung definiert sind, enthalten. (Entsprechendes gilt für die „com“- und „sign“-„Implementierungen“)

Hier gibt es den Sonderfall „basic“, dem keine Operation sondern die den ADT betreffenden Typdefinitionen zugeordnet sind.

Bleiben noch die Komponenten:

1. Im „sign“-Teil sind dies

   „cpp“ enthält die Definitionen für die **C++** Schnittstelle

   „gsig“ enthält die Signatur der Operation in Form eines Funktionsprototypen

   „vsig“ enthält die grundlegende Dokumentation der Operation

2. ansonsten sind dies u.a.:

   „ccod“ enthält die Operation als C-Funktion

   „cmcr“ : dito als C-Makro

   „pmcr“ : dito als **PASCAL**-Makro

   „vdm“ enthält die DSL-Spezifikation der Operation, d.h. es werden die Operationen angefordert, die zur Implemention dieser Operation verwendet werden. (Ein Teil des Enrichments)

   „docu“ enthält die implementationsabhängige Dokumentation, insbesondere das Zeitverhalten

Alle Komponenten werden durch einen C-Präprozessor behandelt, so daß z.B. interne Makrodefinitionen oder *#include*-Aufrufe möglich sind.

3.4 Suchstragegien

Wie findet nun der VDM-Compiler die Komponenten einer Operation? Dies sei an einem Beispiel des ADT *Set*, Implementierung *linklist* und der Operation *cvtotupl* gezeigt:

1. Existiert der Pfad *vdmdb–c/set/linklist/cvtutupl* ? Wenn ja, dann gefunden und stop, ansonsten:
2. Existiert der Pfad *vdmdb–c/set/com/cvtutupl* ? Wenn ja, dann gefunden und stop, ansonsten:
3. Existiert der Pfad *vdmdb–c/com/linklist/cvtutupl* ? Wenn ja, dann gefunden und stop, ansonsten:
4. Existiert der Pfad *vdmdb–c/com/com/cvtutupl* ? Wenn ja, dann gefunden und stop, ansonsten:
5. Operation existiert nicht!

Die implementationsabhängigen Operationsdefinitionen haben also immer Vorrang vor den allgemeineren Definitionen.

Kapitel 4

Definition des ADT Bag

Ein ADT wird durch seine Schnittstelle, d.h. durch die auf ihn anwendbaren Operationen, definiert. (Siehe Kapitel 1.)

Bag soll (siehe Kapitel 2) eine Menge sein, die ihre Elemente mehrfach enthalten kann. Da dies eine Erweiterung der Definition einer Menge ist, die durch den ADT Set abgebildet wird, ist es logisch, dessen Operationen für Bag zu übernehmen, auch wenn diese durch die Erweiterung der Mengedefinition eine etwas andere Auswirkung haben können.

Dazu werden jetzt diese Operationen in den folgenden Abschnitten vorgestellt und die eventuellen Änderungen der Definition u.a. durch ein Beispiel dargestellt:

4.1 Generelle Operationen

Dies sind Operationen, die für *alle* ADT definiert sein müssen und daher hier nicht genauer erläutert werden[1]:

- die (Kopier-) Konstruktoren

[1] Eine genaue Darstellung findet man in [JM91, Anhang B].

- **copy**
- **copy1**

- die Destruktoren
  - **del**
  - **del1**

- der Vergleichsoperation **is_eq**, der bei Bag so definiert[2)3)] ist:

$$
\begin{aligned}
&\textbf{type } is\_eq : Bag \times Bag \longrightarrow Bool \\
&is\_eq(Bag1, Bag2) \mathrel{\widehat{=}} \\
&\quad \forall\ (elem \in Bag1)(elem \in Bag2\ \wedge \\
&\qquad\qquad\qquad is\_eq(Bag1 - \{elem\}, Bag2 - \{elem\}) \\
&\quad \wedge\ is\_eq(Bag2, Bag1)
\end{aligned}
$$

- das Attribut **hash**

- die Druckoperationen
  - **pr**
  - **fpr** (wie **pr**, nur auf einen beliebigen Stream)
  - **print**

- die Konvertieroperationen
  - **cvlines** (Hilfsroutine für die Druckoperationen)
  - **cvtotupl** (Konvertiert einen Bag zu einen ADT **Tuple**, um dessen Fileoperationen benutzen zu können.)
  - **cvtotup1** (Hilfsroutine für **cvtotupl**)
  - **cvfromtupl** (Gegenstück zu **cvtotupl**.)
  - **cvfromtup1** (Hilfsroutine für **cvfromtupl**)

---

2) Die Operationen $\forall$, $\in$ und $-$ sind genauso definiert wie bei normalen Mengen bzw. wie bei Set, daher verwende ich sie ohne vorherige Definition!

3) Die Operation wird anders als hier dargestellt implementiert, aber dazu später mehr!

4.2 Von Set übernommene Operationen

Dies sind die Operationen, die den gleichnamigen Operationen auf den ADT Set entsprechen. Ist eine Operation in ihrer Bedeutung geändert worden, so wird die neue Bedeutung entweder an einem Beispiel erläutert oder mit Hilfe der abstrakten Syntax definiert. In allen anderen Fällen wird nur der Name und eine kurze Erläuterung angegeben[4)]. Die Signatur ist immer gleich der Signatur der entsprechenden Set-Operation und wird daher hier nicht angegeben. Die Operationen mit einem „d“ am Ende des Namens handelt es um Varianten der Operation, die implizit einen **del**-Destruktor für nicht mehr benötigte Elemente bzw. Bags aufruft, ansonsten aber die gleiche Funktionalität haben.

4.2.1 Konstruktoren

- **mk_empty**: Konstruktion eines leeren Bags
- **mk_one**: Konstruktion eines Bags mit einem Element

4.2.2 Kombinatoren

- **diff** und **diffd**: Bag-Differenz. Enthält alle Elemente, die in Bag1 enthalten sind, aber nicht in Bag2 und entspricht den Operationen $(Bag1 \setminus Bag2)$ bzw. $(Bag1 - Bag2)$.
  Beispiel: $\{1, 2, 1, 5, 6, 6, 6\} - \{3, 2, 1, 6\} = \{1, 5, 6, 6\}$
- **sub1** und **sub1d**: Wie **diff**, wobei nur ein Element abgezogen wird. Entspricht der Operation $(Bag - \{Element\})$.
- **inters** und **intersd**: Durchschnitt zweier Bags. Enthält nur die Elemente, die in beiden Bags enthalten sind, wobei berücksichtigt wird, daß die Elemente mehrfach vorkommen können. Entspricht der Operation $(Bag1 \cap$

[4)]Siehe auch [JM91, Kapitel 5.4 und Anhang B].

$Bag2$).
Beispiel: $\{1,2,1,5,6,6,6\} \cap \{3,2,1,6\} = \{1,2,6\}$

- **union** und **uniond**: Vereinigung zweier Bags. Im Gegensatz zum ADT Set, bei dem die Vereinigung keine Elemente mehrfach enthält, werden hier alle Elemente, die in den beiden Bags enthalten sind, auch in dem Vereinigungs-Bag enthalten sein. (Operation ($Bag1 \cup Bag2$))
  Beispiel: $\{1,2,6,6\} \cup \{3,2,1\} = \{1,2,1,2,6,3,6\}$

- **union1** und **union1d**: Wie **union**, wobei ein Bag mit einem Element vereinigt wird, was der Operation ($Bag \cup \{Element\}$) entspricht.

4.2.3 Attribute und Prädikate

- **card**: liefert Anzahl der Elemente eines Bags.

- **is_empty**: Prüft, ob ein Bag leer ist. (Operation ($Bag = \{\}$).)

- **is_in**: Prüft, ob ein Element in einem Bag enthalten ist. (Operation ($Element \in Bag$).)

- **is_subbag**: Prüft, ob ein Bag ein unechter Teilbag eines anderen Bags ist. (Die entsprechende Operation heißt **is_subset** und wurde nur wegen des ADT-Namens in **is_subbag** umbenannt. Entspricht der Operation ($Bag1 \subseteq Bag2$).
  Beispiel: $\{1,3,2,1\} \subseteq \{1,4,2,1,4,3,3\} =$ **true**

- **sel**: Liefert ein beliebiges Element aus einem nichtleeren Bag $\Longrightarrow$ **sel**($\{\}$) ergibt einen Laufzeitfehler!

4.2.4 Iteratoren

- **forall**: Ausführung einer Prozedur für alle Elemente eines Bags. (Dient z.B. für die Implementierung des $\forall$-Operators.)

- **forsome**: Ausführung einee Prozedur für alle die Elemente eines Bags, die ein Prädikat erfüllen.

4.2.5 Funktionale

- **fold**: Generiert eine Funktion, die einen Bag auf einen Resultatstyp abbildet, indem alle Bag-Elemente auf den Resultatstyp abgebildet werden und alle so erzeugten Resultate (rekursiv) paarweise zu einem Resultat zusammengefaßt werden[5)].

- **binary**: Generiert eine Funktion, die einen Resultatstyp und einen Bag auf den Resultatstyp abbilden, indem für alle Bag-Elemente eine entsprechende Operation ausgeführt wird[6)].

4.3 Neue Operationen

Durch die Eigenschaft des ADT Bag, seine Elemente mehrfach speichern zu können, habe ich einige neue Operationen eingeführt, die den „Umgang“ mit mehreren gleichen Elementen erleichtern.

Diese sind:

- der Konstruktor **mk1n**: Konstruktions eines Bags, der ein Element n-mal enthält, mit $n \geq 0$.

  $$\textbf{type}\ mk1n : Element \times\ N_0 \longrightarrow Bag$$

  Diese Operation entspricht in etwa der **Tuple**-Operation **mkn**. Der Name **mkn** erscheint mir allerdings unpassend, da die **Tuple**-Operationen **mk2** und **mk3** zwei bzw. drei Parameter vom Typ *Element* haben, während **mkn**

[5)] Die vollständige Dokumentation befindet sich in [JM91, Seite 61ff].

[6)] Auch hier siehe [JM91, Seite 60f]!

bzw. **mk1n** nur einen solchen Parameter sowie einen N_0 -Parameter besitzen. (Die Signatur sollte sich meiner Meinung nach auch im Namen ausdrücken, so daß man den Namen **mk_one** durch **mk1** ersetzen sollte. Dies sollte aber für alle ADT gelten, sodaß ich den Namen **mk_one** beibehalten habe.)

- das Attribut **occurs**: Er gibt zurück, wie oft ein Element im Bag gespeichert ist.

  type $occurs : Element \times Bag \longrightarrow N_0$

- die Kombinatoren

  - **sub1n** und **sub1nd**:
    Entsprechen der Operation $diff(Bag, mk1n(Element, n))$.

    type $sub1n : Bag \times Element \times N_0 \longrightarrow Bag$

  - **sub1all** und **sub1alld**: Entsprechen der Operation $diff(Bag, mk1n(Element, occurs(Element, Bag))$, d.h. das Resultat enthält dieses *Element* nicht.

    type $sub1all : Bag \times Element \longrightarrow Bag$

  - **union1n** und **union1nd**:
    Sie entsprechen der Operation $union(Bag, mk1n(Element, n))$, d.h. die Vereinigung enthält neben den Elementen aus *Bag* zusätzlich das *Element* n-mal.

    type $union1n : Bag \times Element \times N_0 \longrightarrow Bag$

Kapitel 5

Implementation des ADT Bag

In diesem Kapitel werden die beiden Implementierungen des ADT Bag vorgestellt, die im Laufe der Diplomarbeit entstanden sind:

- **linklist**
- **mapdom**

5.1 linklist

5.1.1 Referenzimplementierung

Die Implementierung als **linklist** wurde von Set als Referenzimplementierung übernommen, da sie die älteste und damit am besten getestete Implementierung von Set ist. Durch die Verwendung der bereits getesteten und bewährten Set-Operationen, die unverändert von Bag verwendet werden, ist also eine große Sicherheit gegeben, denn Fehler können dann nur in den neuen und den geänderten Operationen auftreten.

Da außerdem die Typdefinition von Set für Bag unverändert weiterverwendet wird, kann hier ebenfalls kein Fehler entstehen.

Ferner erlaubt es die **linklist**-Implementierung, *jeden* ADT als Elementtyp zu verwenden. Dies erlaubt z.B. auch rekursive Datendefinitionen, z.B.

```
DOMAINS
   ABag = Abag -bag

IMPLEMENTATIONS
   Abag = linklist
```

5.1.2 Eigenschaften

Jeder Eintrag (Element) des Bags wird in einer Struktur gespeichert, die das Element selbst und einen Zeiger auf die nächste Struktur enthält. Diese Strukturen werden auf dem Heap gespeichert. Die lineare Liste ist also (nur) einfach verkettet. Das Ende der Liste wird durch einen **NULL**-Zeiger markiert.

Das, was der Anwender als ADT-Objekt benutzt, ist der Anker dieser linearen Liste und damit vom Typ Zeiger. Aus diesem Grund sind obige rekursive Typdefinitionen möglich.

Jedes Element des Bags wird auf dem Heap gespeichert, und dies so oft wie das Element im Bag vorkommt. Dadurch können Elemente, die ebenfalls als Zeiger auf eine Heapstruktur implementiert sind, niemals den gleichen Heap-Bereich belegen, auch wenn sie gleich sind.

Dies erlaubt das beliebige Löschen eines Elements mit Hilfe der Elementoperation **del**, also auch bei Verwendung der **forall**-Routine, was sonst nicht erlaubt wäre wegen der Gefahr, einen Heap-Bereich mehrmals freizugeben.

5.1.3 Nachteile

1. Die **linklist**-Implementierung ist langsam, da für jeden Zugriff auf die Elemente im Regelfall eine Operation der Ordnung $O(N)$ benötigt wird. Ope-

rationen auf zwei Bags (Kombinatoren, z.B. **union**) haben damit fast automatisch eine Ordnung $O(N^2)$[1]! (Bei einem Array hat der Elementzugriff die Ordnung $O(1)$, sodaß entsprechende Kombinatoren im Regelfalle eine Ordnung $O(N)$ haben.)

2. Elemente, die mehrfach im Bag vorkommen, werden einzeln auf dem Heap gespeichert, was Speicherplatz kostet.

3. Durch die ungeordnete Speicherung der Elemente wird der Zugriff auf gleiche Elemente im Zusammenhang mit der Zahl ihres Vorkommens — das wäre für die Implementierung von einigen Bag-Operationen wertvoll — nur unter hohen Kosten[2] möglich.

5.2 mapdom

Aufgrund der der oben angeführten Nachteile der **linklist**-Implementierung wurde eine zweite Implementierung geschaffen, die diese Nachteile nicht hat oder zumindest mildern kann.

5.2.1 Grundgedanke

Wenn man gleiche Elemente nur noch einmal speichert und ihnen einen Referenzzähler zuordnet, der angibt, wie oft das jeweilige Element in dem Bag gespeichert, hat man alle drei Nachteile kompensiert (siehe oben).

5.2.2 Realisierung

Die Zuordnung ($Element \mapsto Zähler$) entspricht aber einer Map-Abbildung, sodaß als zweite Implementierung die Abbildung eines Bags durch eine Map gewählt

[1] siehe dazu [RS91, Kapitel 6, Analyse von Algorithmen]

[2] in Form von Rechenzeit!

wurde und den Namen **mapdom** bekam.

Da es bereits einen ADT Map gibt, ergibt sich außerdem der Vorteil, sämtliche (sinnvolle) Implementierungen des ADT Map nutzen zu können, sodaß man also nicht nur eine neue Implementierung bekommt.

Zur Typdefinition wurde folgende Domaingleichung verwendet:

```
DOMAINS
   Map_!DOMAIN = ELEM1 -m-> Nat1

IMPLEMENTATIONS
   Map_!DOMAIN = MAPIMPL
```

Es wird also ein neuer Domain vom Typ Map erzeugt, dessen Name mit dem Namen des aktuellen Bag-Domains erweitert wird (siehe 3.2). Einen konstanten Namen darf man nicht wählen, da dann nur höchsten ein ADT Bag als Map implementiert werden darf!

Der Bezeichner `MAPIMPL` ist ein Parameter der Implementationsgleichung für den aktuellen Bag-Domain. Ein Beispiel:

```
DOMAINS
   Element = ...
   Abag    = Element -bag

IMPLEMENTATIONS
   Element = ...
   Abag    = mapdom(MAPIMPL="linklist")
```

Damit wird für die dem Domain `Abag` unterliegenden Map die **linklist**-Implementierung ausgewählt. Diese bekommt den Namen `Map_abag`. Damit man diesen Parameter auch weglassen kann, wurde er mit einem Defaultwert belegt.

Als Defaultwert wurde **linklist** genommen, denn dies erlaubt für `Element` jeden beliebigen Typ[3)].

Leider gibt es so keine Möglichkeit, zu verhindern, daß eine Implemetierung ausgewählt wird, die entweder nicht zulässig ist[4)], oder nicht zu `Element` passen[5)].

5.2.3 Referenzzähler

Der Referenzzähler — der Mapwert in der Map — hat den Typ **Nat1**. Damit möchte ich ausdrücken, das Elemente, deren Referenzzähler durch irgendwelche Operationen den Wert Null bekommen, aus der Map gelöscht werden, d.h. es wird der zugehörige Speicherplatz freigegeben.

5.2.4 Typdefinition

Damit nun der Map-Domain nun auch den Bag modellieren kann, muß für den Compiler der Zielsprache — hier **C** — noch anzeigt werden, daß `Map_!DOMAIN` und `DOMAIN` den gleichen ADT bezeichen. Dies erreicht man z.B. in **C** durch folgende Typdefinition:

```
typedef Map_!DOMAIN DOMAIN;
```

Damit dies unabhängig von der Zielsprache erfolgen kann, wird dazu das Makro **TYPE_DEF** verwendet, welches bei der Codegenerierung durch den VDM-Compiler abhängig von der Zielsprache definiert wird.

[3)]siehe auch 5.1!

[4)]unzulässig sind die Map-Implementationen **dir**, **heap** und **heapmap**

[5)]z.B. **scaldom** erlaubt normalerweise nur numerische bzw. Zeichen-Typen

Kapitel 6

Implementierung der Operationen

In diesem Kapitel wird die Implementierungen der Bag-Operationen dargestellt, indem sie, aufbauend auf elementaren Operationen, in einer VDM-Notation beschrieben werden. Dabei wird jedoch nicht immer die mathematische Notation für eine Operation verwendet, sondern die entsprechende ADT-Operation benutzt, um deutlich zu machen, daß dabei Funktionen aufgerufen werden!

Diese Form der Notation wurde gewählt, um von der Zielsprache unabhängig zu sein. Falls dann irgendwann eine **PASCAL**-Implementierung — in Form von echten **PASCAL**-Code — ergänzt wird, ist man nicht darauf angewiesen, **C** zu kennen bzw. zu verstehen, um die Operationen „abzuschreiben“.

Dazu sollte man wissen, das Operationen, die Hilfe einer Rekursion alle Elemente eines Bags bearbeiten, in der Regel bei Programmieren durch eine **forall**-Operation implementiert werden.

Bei allen Operationen ist darauf zu achten, daß nur die Operationen auf den Element-ADT zugelassen sind, die für *jeden* ADT definiert sind[1)], da sonst nicht mehr beliebige ADT als Bag-Elemente zugelassen sind, was den Wert des ADT Bag unnötig einschränken würde!

[1)] Siehe die Liste bei 4.1!

6.1 Implementationsunabhängige Operationen

6.1.1 Elementare Operationen

Die Operationen im Verzeichnis *vdmdb–c/bag/com* bilden eine Basis, die bei der Erweiterung von Bag um weitere Implementationen genutzt werden kann. Diese **com**-Operationen — auch generelle Operationen genannt — stehen für alle Implementationen des ADT Bag zur Verfügung.

Aufgrund der Suchstrategie (siehe 3.2) können sie leicht durch implementationsabhängige Operationen überschrieben werden, falls es einen effizienteren Algorithmus für diese Operation gibt.

Sie werden mit anderen **com**-Operationen und nur wenigen *elementaren* Operationen, die implementationsabhängig sind, definiert. Nur diese *elementaren* Operationen *müssen* dann in *jeder* Implementation neu definiert und codiert werden.

Es muß also die Menge der elementaren Operationen bestimmt werden, damit dann die restlichen Operationen mit ihnen implementiert werden können. Da ich sowohl die **com**- als auch die **linklist**-Implementierungen[2)] von Set kopiert habe, blieb mir dafür eignetlich keine andere Wahl als die elementaren Set-Operationen auch beim ADT Bag als elementare Operationen zu definieren.

Wie bestimmt man nun die Menge der elementaren Operation von Set bzw. jetzt Bag?

Man darf nicht davon ausgehen, daß alle die Operationen, die nicht im **com**-Teil von Bag definiert sind, elementar sind, da es schließlich ja noch den globalen ADT-Konstruktor **com** gibt, in dem alle die Operationen definiert sind, die für *alle* ADT funktionieren.

Für Bag sind das die Operationen **cvfromtupl**, **cvtotupl**, **pr** und **fpr**.

Die elementaren Operationen sind damit:

[2)]siehe 6.2

- **mk_empty**
- **del1**
- **union1** und **union1d**
- **sub1** und **sub1d**
- **sel**
- **forall**

Die Operation **is_in** ist zwar eine elementare Operation für Set, sie kann aber durch die **forall**-Operation und der Vergleichsoperation für Elemente implemetiert werden!

6.1.2 Unveränderte Set-Operationen

Die **com**-Operationen **copy**, **copy1**, **mk_one**, **is_eq**[3)], **is_empty**, **card**, **hash**, **forsome**, **fold**, **binary**, **print**, **cvlines**, **cvtotup1** und **cvfromtup1** wurden vom ADT Set kopiert[4)] und blieben damit unverändert. Sie werden daher hier *nicht* dokumentiert.

Die Operationen **diff**, **diffd** und **union**, die alle mit dem Funktional **binary** und den Operationen **sub1**, **sub1d** und **union1** konstruiert werden, habe ich vom Prinzip her unverändert gelassen. Da aber **binary** zur Konstruktion die **forall**-Operation verwendet, müssen die aufgerufenen Operationen **sub1**, **sub1d** und **union1** Funktionen sein, denn **forall** erwartet als Parameter einen Zeiger auf eine Funktion. Dies kann aber nicht garantiert werden, da eventuell die Operationen durch Makros abgebildet werden. Vom Makro aber läßt sich schlecht ein Zeiger auf die Funktion bilden.

[3)] Zusätzlich als Makro implementiert

[4)] Durch das UNIX-Kommando *cp -r set/com bag/com*!

Lösung: Es werden lokale Funktionen codiert, die nichts weiter machen als die ursprüngliche Operation aufzurufen, d.h. es wird eine „Klammer“ um das (eventuelle) Makro gelegt. So kann garantiert werden, daß immer ein Zeiger auf eine *Funktion* vorhanden ist.

6.1.3 Konstruktoren

$$\textbf{type } mk\_one : Element \longrightarrow Bag$$
$$\textbf{type } mk1n : Element \times N_0 \longrightarrow Bag$$

$$mk\_one(elem) \mathrel{\widehat{=}} union1(\{\}, elem)$$
$$mk1n(elem, n) \mathrel{\widehat{=}} union1n(\{\}, elem, n)$$

6.1.4 Destruktoren

Die **del**-Operation konnte nicht von Set übernommen werden, da diese mit Hilfe der **forall**-Operation implementiert wurde. Dies verletzt eine Vorbedingung der **forall**-Operation, da die von ihr gelieferten Elemente gelöscht werden. Dies kann nämlich dann zu schweren Fehlern führen, wenn Elemente, die mehrfach im Bag gespeichert werden, nur einen gemeinsamen Heapblock belegen (zusammen mit einen Zähler für die logische Anzahl), wie dies bei der Implementierung **mapdom** der Fall ist. Eine mehrfache Freigabe eines Heap-Blocks wäre die Folge, was einer der schlimmsten Fehler bei der Programmierung ist!

$$\textbf{type } del : Bag \longrightarrow$$

$$\begin{array}{l} del(bag) \mathrel{\widehat{=}} \\ \quad \textbf{if } bag = \{\} \textbf{ then } del1(bag) \\ \quad \phantom{\textbf{if } bag = \{\}} \textbf{ else } \textbf{let } elem = \textbf{sel } bag \textbf{ in} \\ \quad \phantom{\textbf{if } bag = \{\} \textbf{ else } } del(sub1alld(bag, elem)) \end{array}$$

6.1.5 Kombinatoren

Es werden nur die Versionen ohne implizites Löschen dokumentiert, denn die Versionen mit impliziten Löschen — ihr Name endet mit einem angehängten „d“ — arbeiten auf die gleiche Weise. Allerdings keine Regel ohne Ausnahme: Die Operationen **uniond** und **intersd** müssen auf eine andere Weise implementiert werden, da bei ihnen die Anwendung der **forall**-Operation aus dem gleichen Grund wie bei **del** nicht möglich ist. (Zur Erinnerung: die Rekursionen werden bei der Programmierung durch eine **forall**-Operation implementiert!)

$$
\begin{array}{lll}
\textbf{type } inters & : Bag \times Bag & \longrightarrow Bag \\
\textbf{type } intersd & : Bag \times Bag & \longrightarrow Bag \\
\textbf{type } sub1n & : Bag \times Element \times N_0 & \longrightarrow Bag \\
\textbf{type } sub1all & : Bag \times Element & \longrightarrow Bag \\
\textbf{type } uniond & : Bag \times Bag & \longrightarrow Bag \\
\textbf{type } union1n & : Bag \times Element \times N_0 & \longrightarrow Bag
\end{array}
$$

Für *inters* und *intersd* benötigen wir jeweils noch eine Hilfsoperation:

$$
\begin{array}{lll}
\textbf{type } do\_inters & : Bag \times Bag \times Bag & \longrightarrow Bag \\
\textbf{type } do\_intersd & : Bag \times Bag \times Bag & \longrightarrow Bag
\end{array}
$$

$$
\begin{array}{l}
inters(bag_1, bag_2) \mathrel{\widehat{=}} do\_inters(bag_1, bag_2, \ \{\ \}\ ) \\
do\_inters(bag_1, bag_2, res) \mathrel{\widehat{=}} \\
\quad \textbf{if } \ bag_1 = \{\ \} \\
\qquad \textbf{then } \ res \\
\qquad \textbf{else } \quad \textbf{let } elem = \ \textbf{sel } bag_1 \textbf{ in} \\
\qquad\qquad\quad \textbf{let } result = \ \textbf{if } \ elem \in bag_2 \\
\qquad\qquad\qquad\qquad\qquad \textbf{then } \ union1(res, elem) \\
\qquad\qquad\qquad\qquad\qquad \textbf{else } \ res \\
\qquad\qquad\quad \textbf{in } do\_inters(sub1(bag_1, elem), sub1(bag_2, elem), result)
\end{array}
$$

$intersd(bag_1, bag_2) \mathrel{\widehat{=}} do\_intersd(bag_1, bag_2, \ \{\ \}\ )$

$do\_intersd(bag_1, bag_2, res) \mathrel{\widehat{=}}$

$\quad$ **if** $bag_2 = \{\ \}$ **then** $(del(bag_1)), res$

$\qquad$ **else** **let** $elem = copy($ **sel** $bag_2)$ **in**

$\qquad\quad$ **let** $n_1 = occurs(bag_1, elem)$ **in**

$\qquad\quad$ **let** $n_2 = occurs(bag_2, elem)$ **in**

$\qquad\quad$ **let** $n \ = minimum(n_1, n_2)$ **in**

$\qquad\quad$ **let** $b_2 \ = sub1alld(bag_2, elem)$ **in**

$\qquad\quad$ **let** $result = union1nd(res, elem, n)$ **in**

$\qquad\quad do\_inters(bag_1, b_2, result)$

$sub1n(bag, elem, n) \mathrel{\widehat{=}}$

$\quad$ **if** $n = 0$ **then** bag

$\qquad$ **else** $sub1n(sub1(bag, elem), elem, n-1)$

$sub1all(bag, elem) \mathrel{\widehat{=}} sub1n(bag, elem, occurs(elem, bag))$

$uniond(bag_1, bag_2) \mathrel{\widehat{=}}$

$\quad$ **if** $bag_2 = \{\ \}$ **then** bag_1

$\qquad$ **else** **let** $elem = copy($ **sel** $bag_2)$ **in**

$\qquad\quad$ **let** $n_2 = occurs(elem, bag_2)$ **in**

$\qquad\quad$ **let** $b_2 = sub1alld(bag_2, elem)$ **in**

$\qquad\quad$ **let** $b_1 = union1nd(bag_1, elem, n_2)$ **in**

$\qquad\quad uniond(b_1, b_2)$

$union1n(bag, elem, n) \mathrel{\widehat{=}}$

$\quad$ **if** $n = 0$

$\qquad$ **then** bag

$\qquad$ **else** **if** $n = 1$

$\qquad\quad$ **then** $union1(bag, elem)$

$\qquad\quad$ **else** $union1n(union1d(bag, copy(elem)), elem, n-1)$

Bei der Operation **union1n** ist zu beachten, daß das Element mehrfach im Bag gespeichert wird. Daher müssen $n-1$ Kopien auf dem Heap erzeugt werden — falls es sich nicht um ein sehr einfaches Element handelt — und diese Kopien mittels **union1d** mit dem Bag vereinigt wird. **union1d** sorgt dafür, daß, wenn die Kopie nicht benötigt wird, diese wieder gelöscht werden. Die Kopien sind notwendig, da eventuell später einzelne Elemente wieder gelöscht werden und dabei der Heap-Bereich freigegeben wird. Ohne diese Kopien würden mehrere Elemente sich einen Heap-Bereich teilen, der dann beim Löschen mehrfach freigegeben werden würde!

6.1.6 Attribute und Prädikate

$$
\begin{array}{lll}
\textbf{type } is\_in & : Element \times Bag \longrightarrow & Bool \\
\textbf{type } occurs & : Element \times Bag \longrightarrow & N_0 \\
\textbf{type } is\_eq & : Bag \times Bag \longrightarrow & Bool \\
\textbf{type } is\_subbag & : Bag \times Bag \longrightarrow & Bool
\end{array}
$$

$$
\begin{array}{l}
is\_in(elem, bag) \;\widehat{=} \\
\quad \textbf{if } bag = \{\ \} \quad \textbf{then} \quad \textbf{false} \\
\qquad\qquad\qquad\quad \textbf{else} \quad \textbf{let } e = \ \textbf{sel } bag \textbf{ in} \\
\qquad\qquad\qquad\qquad\quad e = elem \ \vee \ is\_in(elem, bag - \{\ elem\ \}\ )
\end{array}
$$

$$
\begin{array}{l}
occurs(elem, bag) \;\widehat{=} \\
\quad \textbf{if } bag = \{\ \} \quad \textbf{then} \quad 0 \\
\qquad\qquad\qquad\quad \textbf{else} \quad \textbf{let } e = \ \textbf{sel } bag \textbf{ in} \\
\qquad\qquad\qquad\qquad\quad \textbf{let } n = \ \textbf{if } e = elem \textbf{ then } 1 \textbf{ else } 0 \\
\qquad\qquad\qquad\qquad\quad n + occurs(elem, bag - \{\ elem\ \}\ )
\end{array}
$$

$$is\_eq(bag_1, bag_2) \;\widehat{=}\; card(bag_1) = card(bag_2) \ \wedge \ is\_subbag(bag_1, bag_2)$$

$$
\begin{array}{l}
is\_subbag(bag_1, bag_2) \;\widehat{=} \\
\quad \forall\ (elem \in bag_1)\ (occurs(elem, bag_1) \leq occurs(elem, bag_2))
\end{array}
$$

6.2 linklist-Implementierungen

Bei den **linklist**-Implementierungen der Operation handelt es zum größten Teil um Operationen, die aus der **linklist**-Implementierung des ADT Set 1 : 1 übernommen wurden[5)]. Dies betrifft die Operationen **card**, **copy**, **copy1**, **del1**, **fold**, **forall**, **is_empty**, **is_in**, **mk_empty**, **mk_one**, **sel**, **sub1** und **sub1d**.

Die **intersd**-Operation arbeitet genauso wie die **com**-Implementierung von **inters**, so daß sie hier nicht noch einmal dokumentiert wird. (Sie ist effizienter, da keine occurs-Aufrufe benötigt werden). Dabei werden allerdings einige Elemente, die von den **forall**-Iterator geliefert werden, gelöscht. Dies ist aber in der **linklist**-Implementierung erlaubt, da hier alle Elemente eines Bags einen unterschiedlichen Speicherplatz besitzen. (Bei der **mapdom**-Implementierung des Bag werden die Elemente (siehe 5.2), die gleich sind, nur einmal gespeichert und ihnen ein Referenzzähler zugeordnet.)

Aus dem selben Grund kann die **com**-Implementierung der **del**-Operation von Set übernommen werden, denn diese ist effizienter als die **com**-Implementierung von Bag, da keine sub1alld-Aufrufe benötigt werden.

Da alle Elemente in der **linklist**-Implementierung einen unterschiedlichen Speicherplatz besitzen und die Vereinigungsoperationen den Bag immer nur vergrößern können, werden die Versionen mit impliziten Löschen (**union1d**, **union1nd** und **uniond**) durch die entsprechenden Operationen ohne implizites Löschen abgebildet[6)], da eben keine Löschoperationen nötig sind. Dies sorgt auch für ein bißchen Speicherplatzersparnis.

Zur Darstellung der restlichen Operationen — es ist die elementare Operation **union1** sowie die Operationen **occurs** und **union** — sei ein Gedankenmodell erlaubt, bei dem der Bag durch einen ADT Tuple implementiert wird.

[5)]mit Hilfe eines UNIX-Kommandos *cp -r set/linklist bag/linklist*

[6)]u.a. als Makro

Dies geschieht vor allem deswegen, weil die Operationen tatsächlich aus dem Teil der Bibliothek geholt werden, aus dem ähnliche Tuple-Operationen der Tuple-Implementierung **linklist** aufgebaut werden[7]. (Zum Beispiel wird **card** mit der gleichen Routine berechnet wie die Tuple-Operation **len**.)

$$
\begin{array}{ll}
\textbf{type } occurs & : Element \times Bagtuple \longrightarrow N_0 \\
\textbf{type } union1 & : Bagtuple \times Element \longrightarrow Bagtuple \\
\textbf{type } union & : Bagtuple \times Bagtuple \longrightarrow Bagtuple
\end{array}
$$

$$
\begin{array}{l}
occurs(elem, bagtupl) \mathrel{\widehat{=}} \\
\quad \textbf{if } bagtupl = <> \\
\qquad \textbf{then } 0 \\
\qquad \textbf{else } \quad \textbf{let } x = \textbf{ if } elem = \textbf{ hd } bagtupl \textbf{ then } 1 \textbf{ else } 0 \textbf{ in} \\
\qquad\qquad\quad x + occurs(elem, \textbf{ tl } bagtupl)
\end{array}
$$

$$union1(bagtupl, elem) \mathrel{\widehat{=}} < elem > \frown bagtupl$$

$$union(bt_1, bt_2) \mathrel{\widehat{=}} bt_1 \frown bt_2$$

(Der Operator „$\frown$“ konkateniert zwei Tuple zu einem Tuple)

6.3 mapdom-Implementierungen

6.3.1 Konstruktoren und Destruktoren

Die Konstruktoren einschließlich der Kopierkonstruktoren sowie die Destruktoren werden in der **mapdom**-Implementierung durch direkte Aufrufe (bzw. Makroaufrufe) der entsprechenden Operatoren der den Bag modellierenden Map implementiert.

[7] aus dem Verzeichnis *vdmdb–c/aux/linklist*

Dies hat folgende Gründe:

1. Wenn man die Sache im Sinne der objektorientierten Programmierung (siehe [Me88]) betrachtet, handelt es sich bei dieser Bag-Implementierung um eine private Ableitung vom ADT Map.

   Wenn man ein Objekt (bzw. Klasse) beerbt, so hat man in den meisten Fällen dafür zu sorgen (bzw. man wird sogar dazu gezwungen), vor den eigenen Konstruktor den Konstruktor der beerbten Klasse aufzurufen und nach Aufruf des eigenen Destruktors den Destruktor der beerbten Klasse aufzurufen.

   Ich halte es für einen guten Stil, hier genauso zu verfahren, auch wenn man nicht dazu gezwungen ist.

2. Da die Implementierung des benutzten ADT Map nicht bekannt ist, sollte man diese Operationen der Map benutzen, um die möglichst beste (schnellste) Implementierung der Operation zu bekommen, zumal die generellen Operationen (Stichwort **com**) nicht unbedingt die effizientesten sind.

Hier sei kurz noch mal die Implementierung der **mk**-Konstruktoren in Form von Map-Operationen dargestellt:

$$\begin{array}{lll} \textbf{type } mk\_empty : & & \longrightarrow Bagmap \\ \textbf{type } mk\_one & : Element & \longrightarrow Bagmap \\ \textbf{type } mk1n & : Element \times \ N_0 & \longrightarrow Bagmap \end{array}$$

$$mk\_empty \mathrel{\widehat{=}} [\,]$$

$$mk\_one(elem) \mathrel{\widehat{=}} [\ elem \mapsto 1\ ]$$

$$\begin{array}{l} mk1n(elem, n) \mathrel{\widehat{=}} \\ \quad \textbf{if } n = 0 \ \ \textbf{then} \ \ [\,] \\ \quad \phantom{\textbf{if } n = 0 \ \ } \textbf{else} \ \ [\ elem \mapsto n\ ] \end{array}$$

6.3.2 Iterator

Da die **forall**-Operation als einzige der Iteratoren (und auch der Funktionale, die mit ihr implementiert sind) eine elementare Operation ist, mu sie in der **mapdom**-Implementierung codiert sein.

Dies ist relativ einfach, da die **forall**-Operation der Map verwendet wird. Das Problem ist dabei, daß diese Operation eine Prozedur erwartet, die als Parameter ein Element und seinen Mapwert (hier also die Anzahl, die angibt, wie oft das Element im Bag vorkommt) hat. Die Prozeduren, die für die Bag-**forall**-Operation gedacht sind, haben aber als Parameter nur das Element, dafür soll aber diese sooft aufgerufen werden, wie das Element vorkommt.

Dazu kann man folgende Konstruktion verwenden::

$$\begin{array}{ll} \textbf{type } forall & : Bagmap \times (Element \longrightarrow) \longrightarrow \\ \textbf{type } f1 & : Element \times \; N_0 \; \times (Element \longrightarrow) \longrightarrow \end{array}$$

$$\begin{array}{l} forall(bagmap, proc) \mathrel{\widehat{=}} \\ \quad \forall \; (elem \in \; \textbf{dom} \; bagmap) \; \textbf{do} \; f1(elem, bagmap(elem), proc) \end{array}$$

$$\begin{array}{l} f1(elem, n, proc) \mathrel{\widehat{=}} \\ \quad \textbf{if } n > 0 \textbf{ then} \quad \textbf{do } proc(elem) \\ \qquad\qquad\qquad\qquad f1(elem, n-1, proc) \end{array}$$

Der Parameter *proc* von f1 muß bei der Codierung in einer globalen Variablen übergeben werden, da sonst die Signatur der Hilfsroutine für Map-**forall** nicht stimmt, und um Rekursionen zuzulassen, die bei rekursiven Typdefinitionen bei gewissen Operationen vorkommen können, muß diese Variable bei jedem Aufruf von **forall** auf den Stack (mit Hilfe einer lokalen Variable) gesichert werden und nach Beendigung wieder rekonstruiert werden.

6.3.3 Kombinatoren

Angesichts der Implementierung der **forall**-Operation, die nicht gerade begeistern kann, wurden sämtliche Kombinatoren, die zwei Bags zu einem Resultat-Bag abbilden, in der **mapdom**-Implementierung neu codiert, indem jeweils für alle Paare $[Element \mapsto Anzahl]$ einer Map eine entsprechende Teilabbildung erfolgt.

Daß auch die restlichen Kombinatoren — also die, die einen Bag und ein Element (und evtl. eine natürliche Zahl) auf einen Bag abbilden — liegt an der Tatsache, daß die Operationen mit einem Element auf eine Map effizienter codiert sind als die entsprechende generelle Bag-Operation.

Es werden hier nur Operationen ohne die impliziten Löschaufrufe dargestellt, da diese mit den gleichen grundlegenden Algorithmus implementiert werden wie die gleichen Operationen ohne impliziten Löschen[8)].

$$\begin{array}{lllll}
\textbf{type } union1 & : Bagmap \times Element & & \longrightarrow Bagmap \\
\textbf{type } union1n & : Bagmap \times Element \times & N_0 & \longrightarrow Bagmap \\
\textbf{type } union & : Bagmap \times Bagmap & & \longrightarrow Bagmap \\
\textbf{type } sub1 & : Bagmap \times Element & & \longrightarrow Bagmap \\
\textbf{type } sub1n & : Bagmap \times Element \times & N_0 & \longrightarrow Bagmap \\
\textbf{type } sub1all & : Bagmap \times Element & & \longrightarrow Bagmap \\
\textbf{type } diff & : Bagmap \times Bagmap & & \longrightarrow Bagmap \\
\textbf{type } inters & : Bagmap \times Bagmap & & \longrightarrow Bagmap
\end{array}$$

$$union1(bagmap, elem) \mathrel{\hat{=}} union1n(bagmap, elem, 1)$$

$$\begin{array}{l}
union1n(bagmap, elem, n) \mathrel{\hat{=}} \\
\quad \textbf{if } n = 0 \;\; \textbf{then} \;\; bagmap \\
\qquad\qquad\quad\;\; \textbf{else} \;\; bagmap + [elem \mapsto (occurs(elem, bagmap) + n)]
\end{array}$$

[8)]meist besteht der Unterschied nur in den Namen der Map-Operation, z.B. **ovwrt1d** statt **ovwrt1**.

$union(bm_1, bm_2) \widehat{=}$
  if $bm_2 = [\,]$
    then bm_1
    else **let** $elem$ **sel dom** bm_2 **in**
      $union(union1n(bm_1, elem, bm_2(elem)), bm_2 - \{elem\})$

$sub1(bagmap, elem) \widehat{=} sub1n(bagmap, elem, 1)$

$sub1n(bagmap, elem, n) \widehat{=}$
  if $n = 0$ **then** $bagmap$
    else **let** $anz = occurs(elem, bagmap)$ **in**
      if $anz > n$ **then** $bagmap + [elem \mapsto anz - n]$
        else $bagmap - \{elem\}$

$sub1all(bagmap, elem) \widehat{=} bagmap - \{elem\}$

$diff(bm_1, bm_2) \widehat{=}$
  if $bm_2 = [\,]$
    then bm_1
    else **let** $elem =$ **sel dom** bm_2 **in**
      $diff(sub1n(bm_1, elem, bm_2(elem)), bm_2 - \{elem\})$

$inters(bm_1, bm_2) \widehat{=}$
  $[elem \mapsto minimum(bm_1(elem), occurs(elem, bm_2)) \mid$
    $elem \in$ **dom** $bm_1 : occurs(elem, bm_2) > 0]$

6.3.4 Attribute und Prädikate

Auch hier wird wieder ausgenutzt, daß die Operationen, die den ganzen Bag betreffen — **card** und **is_subbag** — durch eine Teiloperation über alle Paare $[Element \mapsto Anzahl]$ abbgebildet werden anstelle über alle Elemente einzeln.

Bei den Operationen **is_eq** und **is_in** wird die entsprechende Map-Operation aufgerufen. Dies geschieht in der Hoffnung, daß die Map-Operationen effizienter arbeiten als die jeweilige **com**-Operation von Bag.

Die Operation **occurs** ist in dieser Implementation sehr einfach, da einfach der Map-Wert ihres Argumentes, falls dieser in der Map eingetragen ist, anderenfalls Null zurückgegeben wird.

Die Operationen **sel** steht nicht als generelle Operationen zur Verfügung und wird durch eine ähnliche Map-Operationen implementiert.

$$
\begin{array}{lll}
\textbf{type } card & : Bagmap & \longrightarrow N_0 \\
\textbf{type } is\_subbag & : Bagmap \times Bagmap & \longrightarrow Bool \\
\textbf{type } is\_eq & : Bagmap \times Bagmap & \longrightarrow Bool \\
\textbf{type } occurs & : Element \times Bagmap & \longrightarrow N_0 \\
\textbf{type } sel & : Bagmap & \longrightarrow Element \\
\textbf{type } is\_in & : Element \times Bagmap & \longrightarrow Bool
\end{array}
$$

$$
\begin{array}{l}
card(bagmap) \mathrel{\widehat{=}} \\
\quad \textbf{if } bagmap = [\,] \ \textbf{then} \ 0 \\
\qquad\qquad\qquad\quad \textbf{else} \ \ \textbf{let } elem = \ \textbf{sel} \ \ \textbf{dom } bagmap \ \textbf{in} \\
\qquad\qquad\qquad\qquad\quad bagmap(elem) + card(bagmap - \{elem\})
\end{array}
$$

$$
\begin{array}{l}
is\_subbag(bm_1, bm_2) \mathrel{\widehat{=}} \\
\quad \forall\ (elem \in \ \textbf{dom } bm_1)(bm_1(elem) \leq occurs(elem, bm_2))
\end{array}
$$

$$
is\_eq(bm_1, bm_2) \mathrel{\widehat{=}} bm_1 = bm_2
$$

$$
\begin{array}{l}
occurs(elem, bagmap) \mathrel{\widehat{=}} \\
\quad \textbf{if } elem \in \ \textbf{dom } bagmap \ \ \textbf{then} \ \ bagmap(elem) \\
\qquad\qquad\qquad\qquad\qquad\quad \textbf{else} \ \ 0
\end{array}
$$

$$sel(bagmap) \mathrel{\hat=} \textbf{sel} \ \ \textbf{dom}\ bagmap$$
$$\textbf{pre}\ sel(bagmap) \mathrel{\hat=} bagmap \neq [\,]$$

$$is\_in(elem, bagmap) \mathrel{\hat=} elem \in \ \textbf{dom}\ bagmap$$

Wie Sie (vielleicht) bemerkt haben, wurde **union1** mit Hilfe von **union1n** implementiert. Da **union1n** ebenfalls hier implementiert wurde, bleibt dies ohne Folgen. Wenn aber aus der Implementierung **mapdom** diese Operation entfernt würde, hätten wir ein schönes Beispiel für eine zirkuläre Definition[9)], da dann die **com**-Implementierung von **union1n** verwendet wird. Diese Version basiert aber auf der Verwendung von **union1**! (Dies gilt entsprechend auch für **sub1** und **sub1n**!)

Fazit:

> Entferne aus der Implementierung **mapdom** niemals die Operationen **union1** / **sub1** und **union1n** / **sub1n**, auch wenn eine generelle Implementierung von **union1n** / **sub1n** existiert!
>
> (Oder ändere die **mapdom**-Implementierung von **union1** / **sub1**!)

6.4 C++ Schnittstelle

Die **C++**-Schnittstelle wurde vollständig von Set übernommen. (D.h. es wurde der **sign**-Teil, der die entsprechenden **cpp**-Komponenten enthält, von Set nach Bag kopiert.) Die Set-Operation **is_subset** wurde in **is_subbag** umbenannt, die Schnittstelle wurde dabei beibehalten.

Die neuen Bag-Operationen (siehe 4.3) wurden mit Hilfe der in [WM92, 2.1.2 im Abschnitt 5] definierten Makros wie folgt hinzugefügt:

---

9) oder auch rekursive Endlosschleife genannt

- **mk1n**

  C++ Konstruktor: domain::domain(const elem&,const nat0&)

- **occurs**

  Member: nat0 domain::occurs(const elem&)

  Friend: nat0 occurs(const domain&, const elem&)

- **sub1all**

  Member: domain& domain::sub1all(const elem&)

  Friend: domain& sub1all(domain&,const elem&)

- **sub1alld**

  Member: domain& domain::sub1alld(const elem&)

  Friend: domain& sub1all(domain&,const elem&)

- **sub1n**

  Member: domain& domain::sub1n(const elem&,const nat0&)

  Friend: domain& sub1n(domain&,const elem&, const nat0&)

- **sub1nd**

  Member: domain& domain::sub1nd(const elem&,const nat0&)

  Friend: domain& sub1nd(domain&,const elem&, const nat0&)

- **union1n**

  Member: domain& domain::plus1n(const elem&,const nat0&)

  Friend: domain& plus1n(domain&,const elem&, const nat0&)

  Achtung: Umbenennung in **plus1n**[10)]

- **union1nd**

  Member: domain& domain::plus1nd(const elem&,const nat0&)

  Friend: domain& plus1nd(domain&,const elem&, const nat0&)

  Achtung: Umbenennung in **plus1nd**[10)]

[10)] siehe dazu [WM92, 5.1 im Abschnitt 3]

Kapitel 7

Testgenerator

Die Testfallwerte für Bags, die als Element(e) von anderen ADT verwendet werden, wurden von Set übernommen und um die Fälle, bei denen die Bag-Elemente mehrfach im Bag gespeichert werden, erweitert.

Das Testgeneratorfile wurde weitgehend von Set übernommen, sofern es nicht um den Test der Kombinatoren **union**, **inters** und **diff** bzw. **sub1** und die ihnen verwandten Operationen geht. Diese wurden größtenteils völlig neu implementiert, genauso wie für die neuen Operatione **mk1n** und **occurs**.

Die Testfälle wurden dabei als Makros definiert, um für alle Operationen des gleichen Typs mit der Operation als Makroparameter die gleichen Testfälle zu generieren.

7.1 Test von mk1n und occurs

Der Test erfolgt z.T. mit Hilfe der Operation **card**.

Als Testfälle bieten sich an:

Sei *elem* irgendein passendes Element. Dann gilt:

$$
\begin{aligned}
ok = \ \forall \ & (n \in \mathcal{M} \subseteq \ N_0 \ ) \\
& (\ card(mk1n(elem, n)) = n \ \wedge \\
& \quad occurs(elem, mk1n(elem, n)) = n)
\end{aligned}
$$

7.2 Test der Vereinigungsoperation

Hier sind alle **union**-Operationen zusammengefaßt, also auch die **union1**- und **uniond**-Varianten.

Die Testfälle sind:

Seien bag_1 und bag_2 zwei beliebige Bags gleichen Typs. Dann gilt:

$$
\begin{aligned}
ok = \ \forall \ & (elem \in bag_1) \\
& (occurs(elem, (bag_1 \cup bag_2)) = \\
& \qquad\qquad occurs(elem, bag_1) + occurs(elem, bag_2) \\
\wedge \ \forall \ & (elem \in bag_2) \\
& (occurs(elem, (bag_1 \cup bag_2)) = \\
& \qquad\qquad occurs(elem, bag_1) + occurs(elem, bag_2) \\
\wedge \ & card(bag_1 \cup bag_2) = card(bag_1) + card(bag_2)
\end{aligned}
$$

7.3 Test der Durchschnittsoperation

Hier sind die beiden **inters**-Operationen zusammengefaßt.

Die Testfälle sind:

Seien bag_1 und bag_2 zwei beliebige Bags gleichen Typs. Dann gilt:

$$
\begin{aligned}
ok = \ \forall \ & (elem \in (bag_1 \cup bag_2)) \\
& (occurs(elem, (bag_1 \cap bag_2)) = \\
& \qquad minimum(occurs(elem, bag_1), occurs(elem, bag_2)) \\
\wedge \ & (bag_1 \cap bag_1) = bag_1 \\
\wedge \ & (bag_2 \cap bag_2) = bag_2
\end{aligned}
$$

7.4 Test der Differenzoperation

Hier sind alle **diff**- und **sub1**-Operationen zusammengefaßt.

Die Testfälle sind:

Seien bag_1 und bag_2 zwei beliebige Bags gleichen Typs. Dann gilt:

$$
\begin{aligned}
ok = \ \forall \ & (elem \in (bag_1 \cup bag_2)) \\
& (\ occurs(elem, (bag_1 \setminus bag_2)) = \\
& \quad \textbf{if} \ \ occurs(elem, bag_1) > occurs(elem, bag_2)) \\
& \qquad \textbf{then} \ \ occurs(elem, bag_1) - occurs(elem, bag_2) \\
& \qquad \textbf{else} \ \ 0 \\
\wedge \ & (bag_1 \setminus bag_1) = \{ \ \} \\
\wedge \ & (bag_2 \setminus bag_2) = \{ \ \}
\end{aligned}
$$

7.5 Änderung der Löschmakros

Diese Makros waren z.T. mit einer nichtportablen Operation definiert worden:

```
#define E_ E_array[Ei++] =
```

Die kritische Operation ist dabei die Inkrementierung von `Ei`, da dieses Makro in einigen Fällen mehrfach innerhalb eines Ausdrucks auftrat, denn in **C** ist dann nicht definiert, wann `Ei` inkrementiert wird und welchen Wert der Ausdruck `Ei++` jeweils hat.

Daher habe ich eine Funktion definiert, die einen Zeiger zurückgibt (bei jeden Aufruf mit einen anderen Wert!), der dann im Makro dereferenziert wird, um einen *Lvalue*-Ausdruck zu bekommen, und ein = angehängt. (Es wäre besser gewesen, das Makro mit einem Parameter anstelle dieser Zuweisung zu definieren, aber dazu ist es wohl zu spät.)

Da ich weiterhin ein Array zum Speichern verwende, habe ich in diese Funktion noch eine Überlaufprüfung eingebaut. Außerdem habe ich die Zahl der Einträge deutlich erhöht.

Für `EE` habe ich dann konsequenterweise ebenfalls eine Funktion (Prozedur) definiert, um Code-Speicherplatz zu sparen.

Da die Makros `D_` und `DD` äquivalente Operationen ausführen, habe ich die Definition der Funktionen, Arrays und Makros in einer Makrodatei zusammengefaßt, die je nachdem ob ein Schalter definiert oder nicht definiert ist, entwder die Element- oder die Domain-Löschoperationen codiert.

Kapitel 8

Fazit

Die Implementierung eines neuen ADT Bag in die Klassenbibliothek von VDM ist gelungen.

Da ich eine großere Menge von Operationen und Typdefinitionen des ADT Set wiederverwenden konnte, wurde die Arbeit doch erleichert.

Mit der Zeit hat es sich aber gezeigt, daß es doch noch einige Schwachstellen in der VDM-Biblithek gibt, die sich auch hier bemerkbar machten.

Dies ist zum einen die Definition der Makros, die immer noch in **PASCAL**- und **C**-Makros aufgeteilt sind, obwohl sie meistens vollkommen identisch sind.

Zum anderen ist auch die Kombination von Makros und Funktionale bzw. Iteratoren meist nicht möglich, da letztere Zeiger auf Funktionen erwarten. Wird das Makro dann durch eine Dummy-Funktion eingeklammert, kann man zwar die Operation durch einen Iterator ausführen lassen, aber eine schöne Lösung ist dies nicht. Ist die Operation nämlich normal, d.h. als **C**-Funktion, implementiert, hat man einen unnötigen zusätzlichen Funktionsaufruf.

Vielleicht sollte man, wenn eine **C++**-Schnittstelle erstellt wird, die Funktionen nicht durch Makros ersetzen, sondern diese als **inline**-Funktionen implementieren.

Dies hat zwei Vorteile:

1. Die Parameter der Funktion unterliegen weiterhin der Typprüfung durch den Compiler.

2. Zumindest in **TURBO-C++** ist es mir gelungen, einen Zeiger auf eine **inline**-Funktion zu bekommen. Es ist also möglich, diese Funktionen in den Funktionalen und Iteratoren zu verwenden.

Die Implementation des **forall**-Iterators bei der Implementierung **mapdom** hat zu Problemen mit löschenden Operationen geführt, so daß diese in der generellen Implementierung nicht mehr effizient implementiert werden konnten. (Meist bedeutet dies eine Multiplizung der Ordnung mit N, so zum Beispiel bei **del**, bei der die **com**-Operation die Ordnung: $N * O(sub1alld) = N^2$ und die **linklist**-Operation die Ordnung N hat.)

Zuletzt sei nochmal auf die Namenskonvention für die **mk**-Konstruktoren hingewiesen, die sehr uneinheitlich ist. Man Vorschlag ist es daher, **mk_one** in **mk1** analog zu den **Tuple**-Konstruktoren **mk2** und **mk3** umzubenennen. Dafür müßte der **Tuple**-Konstruktor **mkn** dannn in **mk1n** umbenannt werden.

Anhang A

Bag-Operationen

In den folgenden Tabellen wird ein Überblick über die Verteilung der Bag-Operationen in der Datenbank gegeben (Stichwort Suchstrategie, sie 3.2). Ein „•“ in einer Spalte bedeutet, daß er der entsprechende Teil vorhanden ist, ansonsten ist er entweder nicht vorhanden oder befindet sich an einem anderen Ort in der Datenbank. Bei den Spalten com, linklist und mapdom, die jeweils in den Unterspalten ccod, cmcr und pmcr unterteilt sind, genügt **ein** „•“ in diesen Unterspalten, um anzuzeigen, daß die entsprechende Komponente vorhanden ist. Ist keine Spalte (oder nur die sign-Spalte) markiert, so ist die Operation global definiert (Pfad *vdmdb–c/com*).

Die einzelnen Spalten haben dabei folgende Bedeutungen:

sign Die Signatur befindet sich im Pfad *vdmdb–c/bag/sign*, ansonsten befindet sie sich im Pfad *vdmdb–c/com/sign*.

com Eine Operation ist unabhängig von der ADT-Implementierung codiert. (Pfad: *vdmdb–c/bag/com*)

linklist Die Operation ist für die **linklist**-Implementierung des Bag angepaßt. (Pfad: *vdmdb–c/bag/linklist*)

mapdom dito für die Map-Implementierung als Bag, mit dem Pfad: *vdmdb–c/bag/mapdom*.

ccod Die Operation ist als **C**-Funktion codiert.

cmcr Die Operation ist (auch) als **C**-Makro codiert.

pmcr Die Operation ist (auch) als **PASCAL**-Makro codiert.

A.1 Konstruktoren und Destruktoren

Name	sign	com			linklist			mapdom		
		ccod	cmcr	pmcr	ccod	cmcr	pmcr	ccod	cmcr	pmcr
mk_empty	•				•	•		•	•	•
mk_one	•	•	•	•	•			•	•	
mk1n	•	•	•	•	•			•	•	
copy		•			•			•	•	•
copy1		•			•			•	•	•
del		•			•			•	•	•
del1					•			•	•	•

A.2 Attribute und Prädikate

Name	sign	com			linklist			mapdom		
		ccod	cmcr	pmcr	ccod	cmcr	pmcr	ccod	cmcr	pmcr
card	•	•			•			•		
hash		•								
is_empty	•	•	•	•	•	•	•	•	•	•
is_eq		•	•	•				•	•	•
is_in	•	•			•			•	•	•
is_subbag	•	•						•		
occurs	•	•			•			•		
sel	•				•			•	•	•

A.3 Kombinatoren

Name	sign	com			linklist			mapdom		
		ccod	cmcr	pmcr	ccod	cmcr	pmcr	ccod	cmcr	pmcr
diff	•	•						•		
diffd	•	•						•		
inters	•	•						•		
intersd	•	•			•			•		
sub1	•				•			•	•	•
sub1d	•				•			•	•	•
sub1n	•	•						•		
sub1nd	•	•						•		
sub1all	•	•	•	•				•	•	•
sub1alld	•	•	•	•				•	•	•
union	•	•			•			•		
uniond	•	•			•	•	•	•		
union1	•				•			•	•	•
union1d	•				•	•	•	•	•	•
union1n	•	•			•			•	•	
union1nd	•	•			•	•	•	•		

A.4 Funktionale und Iteratoren

Name	sign	com			linklist			mapdom		
		ccod	cmcr	pmcr	ccod	cmcr	pmcr	ccod	cmcr	pmcr
binary	•	•								
fold	•	•			•					
forall	•				•	•		•		
forsome	•	•								

A.5 Konvertierungs- und Druckoperationen

Name	sign	com			linklist			mapdom		
		ccod	cmcr	pmcr	ccod	cmcr	pmcr	ccod	cmcr	pmcr
fpr										
pr										
print		•								
cvlines		•								
cvfromtup1		•								
cvfromtupl										
cvtotup1		•								
cvtotupl										

Anhang B

Bag-Implementation

B.1 DSL-Domaingleichung

Die Syntax für die Domaingleichung lautet:

```
Mybag = Myelement -bag
```

B.2 Implementierungen

1. **linklist**

   Parameter: **ALLOC**: Speicherallokierungstrategie
   Default: vmalloc

   Repräsentation: unsortierte verkettete Liste

2. **mapdom**

   Parameter: **MAPIMPL**: Implementation der zugrundeliegenden Map
   Default: linklist

   Repräsentation: $Element \xrightarrow[\mathrm{m}]{} N_0$

Literaturverzeichnis

[BJ78] Bjørner, Dines / Jones, Cliff B. 1978: *The Vienna Devolopment Method: The Meta-Language*, Lecture Notes in Computer Science Bd. 61, Berlin, Heidelberg, New York: Springer 1978

[Me88] Meyer, Bertrand, 1988: *Objektorientierte Softwareentwicklung*, Wien: Hanser / London: Prentice Hall International, 1990

[SH90] Schmidt, Uwe / Hörcher, Hans-Martin, 1990: *Programming with VDM-DOMAINS*, Norsk Data GmbH, Kiel

[JM91] Mohr, Jan Peter, 1991: *Programmierung von abstrakten Datentypen unter Verwendung von VDM*, FH-Wedel

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

[RS91] Sedgewick, Robert, 1991: *Algorithmen*, Addison-Wesley (Deutschland) 1992

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

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

Danksagung

Ich möchte meinen Dank aussprechen an

Prof. Dr. Uwe Schmidt für die hevorragende und erfolgreiche Betreuung der Diplomarbeit

Dipl. Inf. Birger Wolter für die Betreuung in den Rechenzentren

Rune Baßmann für die Bereitstellung von LaTeX

Elmar Zeman für die gemeinsame Vorbereitung zur letzten Klausur

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