Eberhard Karls
Universität
Tübingen

Spezifikation durch Funktionen und die Behandlung von Speicher

Inhaltsverzeichnis

1 Einleitung

Formale Spezifikation bedeutet, Artefakte der realen Welt durch mathematische Objekte zu beschreiben. In VDM-SL und in vielen anderen modellbasierten Spezifikationsmethoden werden die Berechnungsleistung von Prozeduren durch Funktionen beschrieben, welche die Eingaben auf die Ausgaben abbilden.

In den Implementationssprachen findet sich eine Vielfalt an Mechanismen, mit denen Parameter und Variablenadressen an eine Prozedur übergeben werden können, wie Var-Parametern und Zeigern (Pass by Reference). Dem steht in VDM-SL nur eine Art von Funktionsargument gegenüber (Pass by Value). Die Mathematik kennt nichts anderes. Auf den ersten Blick erscheint dieses Ausdrucksmittel unzureichend.

Ich möchte im folgenden Korrespondenz zwischen VDM-SL-Funktionen und Prozeduren, die als Parameter Zeiger auf Variablen erhalten, bzw. zu Methoden in der objektorientierten Programmierung näher beleuchten.

Danach werde ich in groben Zügen erläutern, wie sich Zeiger (Pass by Reference) in VDM-SL explizit modellieren lassen. Die Erarbeitung eines Heap-Modells ist Gegenstand von Aufgabenblatt 3.

2 Syntaktischer Ringelreihen

Eine VDM-SL Spezifikation einer Operation *op*, die aus einem Element f eines abstrakten Datentyps *foo* ein anderes f' produziert, könnte folgendermaßen aussehen:

1.0 $op\,(f : foo, b : bar)\; f' : foo$
.1 post $some\text{-}condition\,(f, b, f')$

Allerdings verraten die Parameter der Spezifikation der Operation wenig über die Aufrufparameter der späteren Implementation. Der Rumpf einer „funktionalen“ Implementation könnte so aussehen:

```
foo op (foo f, bar b) { ... };
```

Diese würde z. B. mit `f1=op(f0,b)` aufgerufen.

In einer anderen Implementation könnte die Konvention die sein, dass ein Zeiger auf eine Variable, in der sich ein *foo* (nämlich f) befindet, an die Prozedur übergeben wird, und der Inhalt der Variablen mit dem Ergebnis überschrieben wird:

```
void op (foo* f, bar b){ ... };
```

Diese würde mit `op(&fvar,b);` aufgerufen.

Oder f könnte ein Objekt der Klasse *foo* sein, das seinen inneren Zustand beim Aufruf einer Methode *op* mit einem Parameter des Typs *bar* ändert.

```
class foo {
     ...
  op (bar b);
     ...
};

foo::op (bar b){ ... };
```

Hier würde die Operation mit `f.op(b)` aufgerufen. Trotz der unterschiedlichen Syntax entspricht auch diese Implementation der obigen Spezifikation: Das Resultat der Operation ist ein neuer Zustand des Objektes. Dieser hängt ab vom anfänglichen Zustand des Objektes und dem Methodenargument vom Typ *bar*. Der Anfangszustand des Objektes ist also auch hier eine Eingabe in die Operation, wenn auch implizit.

3 Unterschiede in der Behandlung von Speicher

Der Unterschied zwischen der dritten (objektorientierten) und der zweiten Implementation (mit der Referenz auf eine Variable, statt des Wertes selbst als Parameter) ist im vorliegenden Beispiel rein syntaktischer Natur, wenn man vom typabhängigen Aufrufdispatch und den dadurch ermöglichten Formen der Inklusions-Polymorphie absieht, die der Klassenmechanismus zwar potentiell bietet, die hier aber keine besondere Rolle spielen.

Interessanter und aufschlussreich dagegen ist der Vergleich zur ersten Implementation: Dort wird aus einem Element des Typs *foo* ein weiteres produziert, beide Werte können nach der „Berechnung“ referenziert werden. Dagegen ist in der zweiten und dritten Implementation der Anfangszustand der Variablen (bzw. des Objektes) – also der Eingabeparameter von Typ *foo* – nach der Berechnung überschrieben, also verloren. Man spricht hier von *destruktivem Update*.

4 Welche Art der Implementation ist zu wählen?

Welche Art der Implementation man wählt, hängt zum einen von der Zielsprache ab, zum anderen von gewissen pragmatischen Überlegungen. Man muss sich hier vor Augen halten, dass in der ersten („funktionalen“) Implementation Speicher belegt werden muss, in dem das Ergebnis gespeichert werden kann[1], und, wenn die Implementationssprache keine Garbage-Collection besitzt, dann auch gelegentlich wieder explizit freigegeben werden muss. Dies ist zum Beispiel der Fall in C und C++, weswegen man dort wohl die zweite und dritte Implementationsvariante bevorzugen würde.

Funktionale Sprachen dagegen erlauben – dank Garbage-Collection – mit komplexen Datenstrukturen ohne explizites Speichermanagement wie mit beliebigen Werten umzugehen. In solchen Fällen wird man – wenn nicht noch andere Überlegungen ins Spiel kommen – die erste Implementierungsvariante (im funktionalen Stil) bevorzugen.

Was den pragmatischen Gesichtspunkt betrifft, so seien einfach zwei von der *Berechnungsleistung*, d. h. von der funktionalen Spezifikation, identische Fälle vorgeführt, in denen die Pragmatik (wie soll die Struktur später eingesetzt werden?), die Art der Implementation diktiert.

```
typedef listnode* list;

list cons (item it, list l) { ... };
```

Hier wird aus einer Liste eine weitere produziert [2]. Anders in

```
typedef struct { int ptr; item store[MAXITEMS] } stack;

void push(item it, stack * s){ ... };
```

[1] Wenn foo nicht zufällig ein sehr kleiner Typ ist, dessen Darstellung in ein Prozessorregister passt.

[2] Der Aufwand für die explizite Speicherverwaltung bleibt aber. Ich gebe zu, dieser Vergleich von *cons* und *push* ließe sich in einer funktionalen Sprache, in der dieser Aufwand nicht entstünde, zwangloser und überzeugender durchführen.

Hier ändert sich der innere Zustand des Stacks. Er dient als *Container* für Werte, deshalb macht die Möglichkeit, seinen alten Zustand referenzieren zu können, in der Implementation nur begrenzt Sinn.

Zuweilen ist eine Implementation mit destruktivem Update auch einfacher durchzuführen (und manchmal auch effizienter) als eine Implementation im funktionalen Stil.

5 Verhältnis von Spezifikation und Definitionsmodul

Fassen wir also zusammen: Der Unterschied zwischen beiden Implementation liegt im Umgang mit Speicher und beide Arten mit Speicher umzugehen sind im geeigneten Kontext sinnvoll.

Der Unterschied ist, andererseits, in der Spezifikation nicht sichtbar, da dort die Berechnungsleistung – die Überführung eines *foo* und eines *bar* in ein weiteres *foo* – beschrieben wird.

An dieser Stelle wird vielleicht auch das Verhältnis zwischen Schnittstellenbeschreibung á la Modula-2 Definitionsmodul – nämlich durch Prozedurköpfe – und der Spezifikation, wie wir sie bis hierher versucht haben durchzuführen, klar. Das eine ist nicht die Verfeinerung oder Konkretisierung des anderen, vielmehr sind beide komplementär: Die Spezifikation charakterisiert die Berechnungsleistung einer Operation präzise, weiß aber wenig über Aufruf und über den Umgang mit Speicher zu sagen. Dagegen charakterisieren die Modula-2-Schnittstellen Syntax und Speicherstrategie des Aufrufs, sagen aber (bis auf suggestive und möglicherweise falsche Prozedurnamen) wenig über die tatsächlich erbrachte *datenverarbeitende* Leistung.

6 Ansätze zur expliziten Beschreibung von Speicher

Zuweilen – und damit nähern wir uns dem entscheidenden Punkt dieser Motivation – möchte man den Umgang mit Speicher (und Zeigern, das sind Referenzen auf Speicher) dennoch explizit thematisieren.

Im vorliegenden Fall mag das nicht zwingend sein, da die Identifikation der zu Ein- und Ausgabenwerten korrespondierenden Speicherinhalte gewissermaßen „mit bloßen Auge“ durchgeführt werden kann. Ich möchte trotzdem bei dem eingeführten Beispiel bleiben: Was tun, wenn man explizit machen möchte, daß der Parameter, den die Operation *op* nimmt, ein Zeiger ist? Der folgende Ansatz (kopieren der Parameternamen aus der zweiten Implementation) hilft jedenfalls nicht, unabhängig davon, wie *foopointer* definiert ist.

2.0 $op\,(f : foopointer, b : bar)$ **Grundfalsch !**
.1 post $\cdots$

Wo ist hier das Ergebnis? Vom Typ *foopointer* kann es außerdem nicht sein, da die Implementation ja offensichtlich keinen Zeiger zurückgibt (sondern einen neuen Wert vom Typ *foo* produziert, und über den müssten wir irgendwie reden, um die Operation zu charakterisieren).

Hier hilft ein Schritt zurück um Überblick zu gewinnen, und die folgende Überlegung: Die Implementation nimmt die Eingabe aus dem Speicher der Maschine, und schreibt das Resultat in den Speicher der Maschine. Die Operation *verändert* den Speicher. Welchen Teil des Speichers sie liest, und welchen sie verändert, sagt ihr der als Parameter übergebene Zeiger, der ja auf einen bestimmten Abschnitt des Speichers verweist. Damit wird die Signatur der Operation klar:

3.0 $op\,(m : memory, fp : foopointer, b : bar)\ m' : memory$

.1 post $\cdots$

Memory müsste natürlich noch geeignet definiert werden, darüberhinaus würde man früher oder später auch noch eine Funktion haben müssen, die festlegt, wie Speicherinhalt als ein Wert vom Typ foo zu interpretieren ist.

4.0 $interpret\text{-}as\text{-}foo\,(m : memory, fp : foopointer)\ f : foo$

Die Verwandtschaft von *interpret-as-foo* zu den schon mehrfach bemühten Retrieve- und Abstraktions-Funktionen ist unübersehbar.

Deutlich zu erkennen ist hoffentlich auch, warum Zeiger „böse“ sind: Ihre Verwendung führt dazu, daß der Speicher der Maschine bei Spezifikation und Beweisen als globale Zustand ständig mitgeführt werden muss und dies erhöht natürlich die gegenseitige Abhängigkeiten und die Gefahr unvermuteter Seiteneffekte ungemein.

7 Schlussfolgerung

Die explizite Beschreibung von Zustand, Variablen bzw. Speicher ist also – mit entsprechendem Aufwand – in VDM-SL (und anderen Spezifikationssprachen mit einem funktionalen Kern) möglich. Das ist gut so, ermöglicht es doch prinzipiell, ein Problem anzugehen, das, damit es möglich ist, sichere, wohldefiniert verhaltene Programme zu erstellen, dringend einer formalen Beschreibung bedarf: Bis hierher war es uns nicht möglich, bei der Spezifikation zu beschreiben, wie sich eine Funktion, die Speicher vom Heap benötigt, verhalten soll, wenn der Speicher nicht alloziert werden kann. Dies erfordert eine explizite Modellierung des Heap, die wir unter anderem auch deshalb nicht vornehmen konnten, weil ein Heap-Mechanismus Speicher verwaltet, auf den über Zeiger zugegriffen wird.

Nachdem nun jedoch die begrifflichen Grundlagen gelegt worden sind, soll die Beschreibung eines Heap und des Zugriffs über Zeiger (Heapreferenzen) der Gegenstand von Übungsblatt 3 sein.

8 Ausblick

Über den bisher angeführten Nutzen hinaus ist die Modellierung des Heap auch deswegen von Interesse, da sie typisch dafür ist, wie in Spezifikationen Seiteneffekte und Referenzen auf impliziten (also in der Implementation nicht benannten Zustand) behandelt werden müssen: Zu dieser Kategorie zählt beispielsweise das Dateisystem, Ausgabegeräte,

5.0 $print\text{-}to\text{-}terminal\,(ts : terminalstate, t : text)\ ts' : terminalstate$

.1 post $\cdots$

oder – im extremsten Fall – die Außenwelt:

6.0 $move\text{-}robot\,(w : world, r : robot\text{-}id, d : distance)\ w' : world$

.1 post $this\text{-}robot\text{-}didn't\text{-}kill\text{-}anybody\,(w, r, w')$