Behandlung von Integer-Arithmetik bei der Verifikation von Java-Programmen

Diplomarbeit

Universität Karlsruhe

Fakultät für Informatik

Institut für Logik, Komplexität und Deduktionssysteme

Verantwortlicher Betreuer: Prof. Dr. P. H. Schmitt

Betreuer: Dr. B. Beckert

Steffen Schlager

31.5.2002

Inhaltsverzeichnis

Vorwort

Thema der Arbeit

Das Thema dieser Diplomarbeit ist die Behandlung von Integer-Arithmetik bei der Verifikation von JAVA-Programmen.

Dabei soll nicht nur ein theoretisches Konzept erarbeitet werden, sondern dieses soll auch implementiert und in das KeY-System, das den Rahmen dieser Diplomarbeit bildet, integriert werden.

Aufbau der Arbeit

Im theoretischen Teil dieser Diplomarbeit soll der schon teilweise existierende Kalkül, basierend auf einer Dynamische Logik für JAVA CARD (siehe [Beckert, 2001]), um die Behandlung von JAVA Integer-Arithmetik erweitert werden. Ein wichtiger Aspekt ist dabei die Behandlung der Probleme, die der sogenannte *overflow* mit sich bringt.

Nach einer kurzen Einführung in Kapitel 1, werden in Kapitel 2 wichtige Begriffe und die der Arbeit zugrunde liegende Dynamische Logik definiert. Nachdem in Kapitel 3 der Zusammenhang von Spezifikation, Implementierung und Verifikation erläutert wurde, wird in Kapitel 4 unsere Semantik für JAVA-Integer Arithmetik definiert, die auf einer Erweiterung der JAVA Syntax basiert. Die Motivation für unsere Semantik und für die Erweiterung der Syntax wird in Kapitel 5 dargelegt. In Kapitel 6 präsentieren wir einen Sequenzenkalkül für eine Dynamische Logik für JAVA Integer-Arithmetik, der auf der in Kapitel 4 definierten Syntax und Semantik beruht.

Der praktischen Teil der Arbeit besteht aus der Implementierung der in Kapitel 6 vorgestellten Regeln und deren Integration in das KeY-System. Die Implementierung ist in Kapitel 7 beschrieben.

Bemerkung

Diese Diplomarbeit ist eine Kurzfassung der ausführlichen, in englischer Sprache verfaßten, Arbeit [Schlager, 2002].

Danksagung

An dieser Stelle möchte ich mich bei meinen Betreuern Prof. Peter H. Schmitt und Dr. Bernhard Beckert für ihre Unterstützung bei der Erstellung dieser Diplomarbeit bedanken. Ganz besonders bedanke ich mich bei Bernhard Beckert dafür, daß er immer Zeit für mich hatte und meine vielen Fragen geduldig beantwortete. Außerdem möchte ich mich bei meinen HiWi-Kollegen Richard Bubel und Andreas Roth für ihre unverzichtbare Unterstützung bei der Implementierung bedanken. Zu guter letzt bedanke ich mich bei allen Mitarbeitern der Forschungsgruppe von Prof. Dr. Wolfram Menzel und Prof. Dr. Peter H. Schmitt für die tolle Arbeitsatmosphäre.

Erklärung

Ich erkläre hiermit, diese Arbeit selbständig verfaßt und keine anderen als die angegebenen Quellen und Hilfsmittel verwendet zu haben.

Karlsruhe, den 31. Mai 2002

Steffen Schlager

1 Einleitung

Diese Diplomarbeit entstand im Rahmen des KeY-Projekts [Ahrendt et al., 2000], welches die Integration formaler Methoden in die industrielle Software-Entwicklung zum Ziel hat. Bisher werden formale Methoden in der Software-Entwicklung nur sehr selten eingesetzt. Der Grund dafür liegt aber nicht etwa in der mangelnden Leistungsfähigkeit der Werkzeuge. Vielmehr wird der breite Einsatz formaler Methoden durch folgende Punkte erschwert:

- Die Werkzeuge für formale Methoden sind nicht in die Standardwerkzeuge (z.B. CASE-Tools) für die Software-Entwicklung integriert.
- Oft werden als Zielsprache für die Verifikation „künstliche“, stark vereinfachte, Programmiersprachen verwendet und nicht die in der Software-Entwicklung üblichen modernen Sprachen.
- Die meisten Werkzeuge setzten tiefergehende Kenntnisse in den Bereichen Logik und Theorem-Beweisen voraus, die man von den meisten Software-Entwicklern nicht erwarten kann.

Die Beseitigung der angesprochenen Probleme bildet die Motivation für das KeY-Projekt. Um das Ziel, nämlich die Beseitigung obiger Probleme zu erreichen, wurden folgende grundlegende Entscheidungen für das Projekt gefällt:

- Als Spezifikations- und Modellierungssprache wird UML/OCL verwendet.
- Als Zielsprache für die Verifikation dient eine Untermenge von Java, genannt Java Card.
- Das Verifikationswerkzeug wird in das kommerzielle CASE-Tool Together Control Center der Firma TogetherSoft integriert.
- Die Benutzerschnittstelle muß graphisch orientiert und ergonomisch sein.

Die Aufgabe, die es in dieser Arbeit zu lösen gilt, bezieht sich auf die Deduktionskomponente des KeY-Systems. Der in dieser Arbeit entwickelte Kalkül ermöglicht die Behandlung der gesamten Java Integer-Arithmetik, wobei insbesondere die Probleme im Zusammenhang mit *overflow* berücksichtigt werden.

2 Präliminarien

In diesem Kapitel definieren wir unter anderem die Syntax und Semantik einer Dynamischen Logik für JAVA. Wir folgen dabei weitgehend den Definition in [Beckert, 2001]

2.1 Syntax der Dynamische Logik für Java

Der nicht-dynamische Teil unserer Dynamischen Logik (DL) entspricht einer sortierten Prädikatenlogik erster Stufe. Für die Definition der Syntax müssen wir zuerst die Menge der Variable, ihre Typen und Terme festlegen. Danach definieren wir die zulässigen Programme der DL und die Syntax der Formeln der DL.

Kontext

Programme, die in DL-Formeln vorkommen, dürfen keine Klassen-Definitionen enthalten. Stattdessen wird die Syntax und Semantik von DL-Formeln bezüglich eines gegebenen JAVA-Programms (dem *Kontext*) definiert.

Variablen

In der klassischen DL verwendet man üblicherweise nur eine Art von Variablen. Um Mißverständnissen vorzubeugen, verwenden wir in dieser Arbeit zwei Arten von Variablen:

- *Programmvariablen* werden mit $\mathtt{x}$, $\mathtt{y}$, $\mathtt{z}$, ... bezeichnet. Ihr Wert kann in jedem Zustand ein anderer sein und durch Programme verändert werden. Programmvariablen dürfen nicht durch Quantoren gebunden oder mit Termen instantiiert werden.

- *Logische Variablen* werden mit x, y, z, ... bezeichnet. Sie haben in jedem Zustand denselben Wert und können nicht durch Programme verändert werden. Logische Variablen müssen immer durch Quantoren gebunden sein, dürfen also nicht frei vorkommen. Ferner können logische Variablen mit Termen instantiiert werden.

Typen

Die Menge der Typen unserer DL besteht aus

- den primitiven Typen `boolean`, `byte`, `short`, `int` und `long`,
- den zusätzlichen arithmetischen Typen `arithByte`, `arithShort`, `arithInt` und `arithLong`, die in Kapitel 4 definiert werden,
- den Klassen, die im Kontext definiert sind,
- dem Typ *Null*,
- den eingebauten Klassen wie z. B. `Object` und `String`,
- einem Array-Typ *T[]* für jeden primitiven Typ, für jeden arithmetischen Typ, für jeden Array-Typ und für jede Klasse, und
- benutzerdefinierten Typen (typischerweise als ADTs).

Terme

Logische Terme werden wir üblich aus Programmvariablen, logischen Variablen und den Funktionssymbolen aller Typen gebildet. Die Menge der Terme beinhaltet insbesondere Java-Literale für die primitiven Typen, String-Literale und die `null`-Referenz.

Zusätzlich gilt, falls *o* ein Term des Klassentyps *C* und *a* ein Feld (Attribut) von *C* ist, dann ist auch *o.a* ein Term ist. Ebenso, falls *a* ein Term eines Array-Typs und *i* vom Typ `byte`, `short`, `int`, `arithByte`, `arithShort` oder `arithInt` ist, dann ist auch *a[i]* ein Term.[1]

Programme

Die Programme, die in DL-Formeln enthalten sein dürfen, bestehen nur aus ausführbarem Code, d. h. sie dürfen keine Klassen- oder Interface-Definitionen enthalten.

Allerdings dürfen Programme in DL-Formeln (nicht aber im Kontext) logische Terme enthalten. Anstelle eines Java-Ausdrucks kann immer auch ein Term mit demselben Type verwendet werden, d. h. Java-Ausdrücke können Terme enthalten, aber nicht umgekehrt.

[1] Variablen des Typs `long` sind nach der Java-Sprachspezifikation [Bracha et al., 2000] als Indizes für Array-Zugriffe nicht zugelassen.

Formeln

Formeln werden wie üblich aus logischen Termen, den Prädikatssymbolen aller Typen, dem Gleichheitsprädikat $\doteq$, den logischen Verknüpfungszeichen $\neg$, $\wedge$, $\vee$, $\rightarrow$, den Quantoren $\forall$ und $\exists$ und den Modaloperatoren $\langle\cdot\rangle$ und $[\cdot]$ gebildet, d. h. wenn p ein Programm ist und ϕ eine Formel, dann ist $\langle p\rangle\phi$ ebenfalls eine Formel.

Update

Ein sogenanntes *Update* hat die Form $\{\mathit{ls} \leftarrow t\}$, wobei ls eine Programmvariable, ein Feldzugriff (d. h. von der Form `o.a`), ein Array-Zugriff (d. h. von der Form `a[j]`) und t ein logischer Term ist.

Updates werden eingeführt um die Schreibweise zu vereinfachen. Die Semantik der Formel $\phi^{\{\mathit{ls}\leftarrow t\}}$ entspricht der Semantik der Formel $\langle\, \mathit{ls}\ \texttt{=}\ t;\rangle\phi$.

2.2 Semantik Dynamischen Logik für Java

Um die Semantik für unsere DL zur definieren, müssen wir zuerst die Semantik der verwendeten Programmiersprache, eine erweiterte Version von JAVA, definieren. Für die Definition der Semantik von Anweisungen und Ausdrücken, die keine zusätzlichen arithmetischen Typen enthalten, verweisen wir auf die JAVA-Sprachspezifikation [Bracha et al., 2000]. Die Semantik der erweiterten JAVA Integer-Arithmetik ist in Kapitel 4 definiert.

Die Modelle für DL sind Kripke-Strukturen, die aus den möglichen Welten (in DL werden diese als *Zustände* bezeichnet) und einer Zugänglichkeitsrelation auf der Menge der Zustände bestehen. Die Funktions- und Prädikatssymbole werden in jedem Zustand gleich interpretiert unter Verwendung einer einzigen festen prädikatenlogischen Struktur, wobei auch das Universum für jeden Zustand dasselbe ist und aus einer unendlichen Anzahl aus Elementen jedes Typs besteht.

Logische Variablen werden durch eine globale Variablenbelegung interpretiert, d. h. sie haben in jedem Zustand des Modells denselben Wert.

Zustände

Zustände sind durch die Belegung der Programmvariablen (einschließlich `this`), der Attribute (Felder) aller Objekte (einschließlich Arrays) und der Klassenattribute (statischen Felder) aller Klassentypen definiert.

Man beachte, daß Zustände keine Information über den Kontrollfluß eines Programms, wie z. B. einen Programmzähler oder die Information, daß eine Exception geworfen wurde, enthalten.

Programme und Formeln

Die Semantik eines Programms ist durch eine Menge von Zustandsübergängen definiert. Man schreibt $s\,[\![p]\!]\,t$, falls der Zustand t durch das Programm p, gestartet im Zustand s, erreichbar ist. Da JAVA eine deterministische Programmiersprache ist, ist die Menge der erreichbaren Zustände entweder leer (p terminiert nicht) oder besteht aus genau einem Element (p terminiert). Wir legen fest, daß die Menge der Zustände unter der Zugänglichkeitsrelation $[\![]\!]$ abgeschlossen sein muß, d. h. jeder Zustand, der durch ein Programm erreichbar ist, muß auch im Modell existieren.

Die Semantik eines logischen Terms t innerhalb eines Programms ist dieselbe wie die eines seiteneffektfreien JAVA-Ausdrucks, dessen Auswertung denselben Wert ergibt wie t.

Für Formeln ϕ, die keine Programme enthalten, ist die Gültigkeit von ϕ in einem Zustand s definiert wie in der Prädikatenlogik üblich. Eine Formel $\langle p \rangle \phi$ ist in einem Zustand s gültig, falls p in einem Zustand t terminiert, d. h. $s\,[\![p]\!]\,t$, in dem ϕ gültig ist. Eine Formel ϕ ist erfüllbar in einem Modell $\mathcal{M}$, falls ϕ in einem Zustand von $\mathcal{M}$ gültig ist. Die Formel ϕ ist gültig in $\mathcal{M}$, falls ϕ in jedem Zustand von $\mathcal{M}$ gültig ist. Allgemeingültig ist eine Formel, falls sie in jedem Modell gültig ist.

Kalkül

Um auf den in Kapitel 6 anhand einiger Regeln exemplarisch vorgestellen Kalkül Bezug nehmen zu können, führen wir das Symbol $\mathcal{C}_{DLJ}$ ein.

Definition 2.2.1 (Kalkül $\mathcal{C}_{DLJ}$)
Der Kalkül $\mathcal{C}_{DLJ}$ *besteht aus den in [Schlager, 2002] in den Abschnitten 6.3 bis 6.9 definierten Regeln und aus den in [Beckert, 2001] definierten Regeln.*

2.3 Binärdarstellung der ganzzahligen Java-Typen

In JAVA gibt es die fünf primitiven, ganzzahligen Typen `byte`, `short`, `int`, `long` und `char`. Die Wertebereiche dieser Typen sind alle endlich und sind in Tabelle 2.1 abgebildet, wobei auch die Anzahl der Bits angegeben ist, die zur Speicherung eines Wertes des jeweiligen Typs erforderlich sind. Im folgenden wird der minimale bzw. der maximale Wert eines Typs *T* als `MIN_`*T* bzw. `MAX_`*T* bezeichnet.

Der Datentyp `char` wird intern in der üblichen Binärdarstellung ohne Vorzeichen, gespeichert. Im folgenden werden wir den Typ `char` nicht weiter betrachten, was die Integer-Arithmetik angeht, da die Werte dieses Typs als Unicode-Zeichen interpretiert werden und deshalb für Arithmetik nicht verwendet werden sollten.

Datentyp	Wertebereich	Anzahl der Bits
`byte`	-128 bis 127	8
`short`	-32768 bis 32767	16
`int`	-2147483648 bis 2147483647	32
`long`	-9223372036854775808 bis 9223372036854775807	64
`char`	0 bis 65535	16

Tabelle 2.1: Wertebereiche der primitiven ganzzahligen JAVA-Typen.

Die anderen vier Datentypen werden intern im sogenannten *Zweierkomplement* gespeichert. Diese Darstellung wird in vielen Programmiersprachen verwendet, da damit insbesondere die Addition und die Subtraktion sehr effizient ausgeführt werden können.

Für die Definition des Zweierkomplements und der Konvertierung in Dezimaldarstellung und umgekehrt verweisen wir auf Abschnitt 2.4 in [Schlager, 2002].

3 Spezifikation, Implementierung und Verifikation

Wie schon erwähnt, wird die Unified Modeling Language (UML) im KeY-Projekt als Modellierungssprache eingesetzt. Der Grund für diese Entscheidung liegt darin, daß die UML mittlerweile einen Industriestandard darstellt, was für eine möglichste weite Verbreitung des KeY-Werkzeugs sehr wichtig ist.

Die UML an sich ist graphisch orientiert und für eine formale Spezifikation nicht ausreichend präzise. Allerdings enthält sie seit der Version 1.3 als eine Untermenge die Object Constraint Language (OCL). Die OCL ist textuell orientiert und erlaubt es, auch Eigenschaften, die nicht graphisch ausgedrückt werden können, formal zu spezifizieren. Ein Problem ist allerdings, daß es keine präzise formale Definition der Semantik der OCL gibt. Eine Möglichkeit, um dennoch zu einer formalen Semantik zu gelangen, ist es, die OCL in eine andere Sprache oder Logik zu übersetzen, die eine formale Semantik besitzt. Dieser Ansatz wird im KeY-Projekt verfolgt, wobei die OCL in eine Dynamische Logik übersetzt wird. Wir werden im Rahmen dieser Arbeit nicht näher auf die Übersetzung eingehen, sondern verweisen auf [Keller, 2002].

Die Zielsprache für die Verifikation ist, wie bereits erwähnt, Java Card. Diese Sprache ist eine Untermenge von Java und wird für die Programmierung von Smartcards eingesetzt. Da Anwendungen auf Smartcards oft sicherheitskritisch sind (man denke etwa an Authentisierung per Smartcard), bieten sich diese für die formale Verifikation geradezu an. Ferner besitzen Java Card-Anwendungen noch weitere Vorteile, die sie für die formale Verifikation prädestinieren:

- Java Card-Anwendungen sind sehr klein, da der Speicher auf Smartcards beschränkt ist (aktuelle Smartcards bieten gerade einmal 16 kilo-byte an Speicher für Programme an).
- Im Gegensatz zu Java gibt es bei Java Card keine Threads, und keinen Garbage-Collector, was die Verifikation deutlich erleichtert. Ferner gibt es in Java Card weniger Datentypen und Bibliotheken.

In Java Card gibt es als primitive ganzzahlige Typen nur `byte` und `short`. Im folgenden werden wir aber in unsere Betrachtungen auch die Java-Typen `int` und `long` miteinbeziehen, da es, was unsere Arbeit angeht, zwischen den erstgenannten und den zuletztgenannten keinen fundamentalen Unterschied gibt. Aus diesem Grunde sprechen wir ab jetzt von Java Integer-Arithmetik. Allerdings gelten unsere Überlegungen auch für die Sprache Java Card.

Nun haben wir sowohl die Spezifikationssprache als auch die Implementierungssprache, die im KeY-Projekt verwendet werden, vorgestellt und gehen im letzten Teil dieses Abschnitts noch etwas näher auf die Verifikation ein.

Programmverifikation bedeutet ganz allgemein zu beweisen, daß ein Programm seine Spezifikation erfüllt. Die logische Grundlage des KeY-Projekts bildet die Dynamische Logik [Harel, 1979], und die Korrektheit eines Programms wird bewiesen, indem man die Gültigkeit gewisser DL-Formeln (sogenannten *Beweisverpflichtungen*) nachweist, die aus der Spezifikation und Implementierung erzeugt werden.

Die Dynamische Logik (DL) kann als eine multi-modale Prädikatenlogik betrachtet werden, die für jedes erlaubte Programm p der Zielsprache[1] die Modalitäten $\langle p \rangle$ und $[p]$ enthält. Diese Modalitäten beziehen sich auf die durch das Programm p erreichbaren Welten (in DL spricht man üblicherweise von „Zuständen"). Die DL-Formel $\langle p \rangle \psi$ drückt aus, daß das Programm p in *zumindest einem* Zustand terminiert, der die Formel ψ erfüllt und $[p]\psi$ drückt aus, daß ψ in *allen* erreichbaren Zuständen gilt. Im allgemeinen gibt es mehrere mit einem Programm erreichbare Zustände, da die zugrundeliegende Programmiersprache nicht deterministisch sein muß. Da wir aber die deterministische Sprache JAVA verwenden, gibt es in unserem Falle nur einen erreichbaren Zustand (falls das Programm terminiert) oder keinen (falls das Programm nicht terminiert).

DL kann auch als eine Art Hoare-Logik betrachtet werden, denn die Formel $\phi \rightarrow [p]\psi$ ist dem Hoare-Tripel $\{\phi\}p\{\psi\}$ ähnlich. Allerdings ist die DL unter den logischen Operatoren abgeschlossen. Deshalb können ϕ und ψ selbst wieder Programme enthalten, wohingegen in der Hoare-Logik ϕ und ψ rein prädikatenlogische Formeln sein müssen.

[1] Je nach Zielsprache gelangt man zu verschiedenen Varianten der Dynamischen Logik und daher sollte man eigentlich nicht von *der* Dynamischen Logik reden. Wir werden dies dennoch tun, da bei uns klar aus dem Kontext hervorgeht, daß es sich um eine Dynamische Logik für JAVA handelt.

4 Erweiterung der Syntax und Semantik von Java

In diesem Kapitel definieren wir die Erweiterung der JAVA-Syntax um die sogenannten arithmetischen Typen. Anschließend wird dann für diese erweiterte Syntax die Semantik $\mathcal{S}_{KeY}$ definiert.

4.1 Erweiterung der Syntax von Java

4.1.1 Zusätzliche primitive Typen

Zusätzlich zu den eingebauten primitiven JAVA-Typen `byte`, `short`, `int` und `long` führen wir die sogenannten *arithmetischen Typen* `arithByte`, `arithShort`, `arithInt` und `arithLong` ein.

4.1.2 Zusätzliche Methoden

Wir erweitern die JAVA-Klasse `java.lang.Object` um die zusätzliche Methode `overflow`. Die Deklaration dieser Methode ist in Abbildung 4.1 gezeigt.[1]

Da die Methode `overflow` in der Klasse `Object` definiert ist, wird sie von jeder anderen Klasse geerbt. Ferner kann `overflow` nicht überschrieben werden, da sie als `final` deklariert wurde. Wir legen fest, daß `overflow` in einem Programm nicht explizit aufgerufen werden darf.

Im Gegensatz zu den anderen Methoden der Klasse `Object` ist das Verhalten der Methode `overflow` nicht spezifiziert. Die Gründe dafür sind in Abschnitt 5.3 erklärt.

[1] Eigentlich sind es mehrere Methoden, da `overflow` „überladen“ ist.

```
final arithInt overflow(arithInt arg, String op)
final arithInt overflow(arithInt arg1, arithInt arg2, String op)
final arithLong overflow(arithLong arg, String op)
final arithLong overflow(arithLong arg1, arithInt arg2, String op)
```

Abbildung 4.1: Diese Abbildung zeigt die Signatur der zusätzlichen („überladenen“) Methode `overflow` in der Klasse `Object`.

4.2 Erweiterung der Semantik von Java

In diesem Abschnitt definieren wir die Semantik $\mathcal{S}_{KeY}$ der zusätzlichen arithmetischen Typen und darauf anwendbaren arithmetischen Operatoren.

4.2.1 Wertebereich der arithmetischen Typen

Die arithmetischen Typen haben alle einen unendlichen Wertebereich und sind damit isomorph zu dem OCL-Typ Integer.

4.2.2 Semantik der arithmetischen Operatoren

In diesem Abschnitt definieren wir die Semantik der Operatoren, die auf die eingebauten und auf die arithmetischen Typen angewendet werden können. Wir setzen dabei den Begriff *promoted type* als bekannt voraus.[2] Ferner definieren wir hier nur dann explizit die Semantik der Operatoren, wenn sie sich von der „üblichen" Java-Semantik unterscheidet, d. h. also, wenn für einen Operator keine Semantik definiert ist, dann entspricht sie der Definition in [Bracha et al., 2000].

- Vergleichsoperatoren (Ergebnistyp ist `boolean`):
  - Numerische Vergleichsoperatoren `<`,`<=`,`>` und `>=`
  - Numerische Gleichheitsoperatoren `==` und `!=`
- Numerische Operatoren (Ergebnistyp ist `int`, `long`, `arithInt` oder `arithLong`):
  - Monadische Operatoren `+` und `-`

    Falls der *promoted type* T des Arguments ein arithmetischer Typ ist, wird beim monadischen Minus-Operator `overflow` aufgerufen, falls der Wert des Arguments `MIN_`T ist. Ansonsten entspricht die Semantik der üblichen mathematischen Bedeutung der Operatoren.
  - Multiplikative Operatoren `*`,`/` und `%`

    Falls der *promoted type* T des Arguments ein arithmetischer Typ ist und die Werte beider Argumente im zulässigen Bereich liegen, das mathematische Ergebnis der jeweiligen Operation aber nicht, dann wird `overflow` aufgerufen. Ansonsten entspricht die Semantik der üblichen mathematischen Bedeutung der Operatoren.
  - Additive Operatoren `+` and `-`

    Die Semantik ist analog zu den multiplikativen Operatoren definiert.

[2] Ansonsten verweisen wir für die eingebauten Typen auf die Java-Sprachspezifikation [Bracha et al., 2000] und für die arithmetischen Typen auf Kapitel 5 in [Schlager, 2002].

– Präfix (Postfix) *increment* und *decrement* Operatoren `++` and `--`

  Die Semantik dieser Operatoren ist wie in der JAVA-Sprachdefinition unter Verwendung der Semantik der additiven Operatoren definiert.

– Bit-Operatoren `<<`, `>>`, `>>>`, `~`, `&`,`|`, und `^`

  Bit-Operatoren dürfen nicht auf arithmetische Typen angewendet werden, da ihre Semantik von der verwendeten Binärdarstellung der Zahlen abhängt, und damit auch von der Endlichkeit des Wertebereichs der eingebauten Typen.

– *Conditional*-Operator `? :`

– *Cast*-Operatoren

  Falls der Zieltyp *T* der *cast*-Operation ein arithmetischer Typ ist, dann wird `overflow` genau dann aufgerufen, wenn der Wert des Arguments nicht in dem *T* entsprechenden eingebauten Typ repäsentierbar ist. Ansonsten entspricht die Semantik der üblichen JAVA-Semantik.

- Übrige Operatoren:

  – Logische Operatoren `!`, `&&` und `||`

  – Boolsche logische Operatoren `&`, `|` und `^`.

  – Einfache und komplexe Zuweisungsoperatoren `=`, `*=`, `/=`, `%=`, `+=`, `-=`, `&=`, `^=`, `|=`, `<<=`, `>>=` und `>>>=`

    Die komplexen Zuweisungsoperatoren `&=`, `^=`, `|=`, `<<=`, `>>=` und `>>>=` dürfen nicht auf arithmetische Typen angwendet werden, da sie Bit-Operatoren enthalten.

5 Verschiedene Semantiken für Java Integer-Arithmetik

In diesem Kapitel motivieren wir die im vorherigen Kapitel definierte Semantik für JAVA Integer-Arithmetik, die wir im folgenden mit $\mathcal{S}_{KeY}$ bezeichnen. Bevor wir näher auf $\mathcal{S}_{KeY}$ eingehen, stellen wir zuerst zwei andere Semantiken $\mathcal{S}_{OCL}$ und $\mathcal{S}_{Java}$ für JAVA Integer-Arithmetik vor, da es sich im Nachhinein herausgestellt hat, daß $\mathcal{S}_{KeY}$ eine Kombination dieser beiden Semantiken ist.

Wichtig für das Verständnis dieses Kapitels ist, sich immer vor Augen zu halten, daß die hier vorgestellten Semantiken $\mathcal{S}_{OCL}$, $\mathcal{S}_{Java}$ und $\mathcal{S}_{KeY}$ Programmsemantiken für die Dynamische Logik sind und sich durchaus von der Semantik $\mathcal{S}_{JVM}$, die die virtuelle Machine implementiert, unterscheiden können.

5.1 Semantik $\mathcal{S}_{OCL}$

Die erste Semantik $\mathcal{S}_{OCL}$ für JAVA Integer-Arithmetik, die wir vorstellen, ist von der Semantik der OCL motiviert. In $\mathcal{S}_{OCL}$ werden die primitive JAVA-Typen `byte`, `short`, `int` und `long` alle wie der OCL-Typ INTEGER interpretiert, d. h. ihr Wertebereich ist unendlich. Ferner werden die arithmetisch JAVA-Operatoren in $\mathcal{S}_{OCL}$ ebenfalls wie in der OCL interpretiert und also wie in der Mathematik. Insbesondere bedeutet das, daß *overflow* nicht berücksichtigt wird. Man erkennt also sofort, daß sich $\mathcal{S}_{OCL}$ und $\mathcal{S}_{JVM}$ unterscheiden.

Dieser Unterschied kommt zum Tragen, wenn man über die Korrektheit von Programmen redet. Ganz allgemein bedeutet die Korrektheit eines Programms, daß es bei der tatsächlichen Ausführung[1] seine Spezifikation erfüllt. Im KeY-Projekt wird die Korrektheit von Programmen formal bewiesen, indem man die Gültigkeit von DL-Beweisverpflichtungen, die aus der Spezifikation und der Implementierung generiert werden, nachweist. Da für die Interpretation der Programme in DL die DL-Programmsemantik verwendet wird und diese sich, wie bei $\mathcal{S}_{OCL}$, von der Ausführungssemantik (in JAVA ist das $\mathcal{S}_{JVM}$) unterscheiden kann, folgt aus der Gültigkeit der DL-Beweisverpflichtungen im allgemeinen nicht, daß das Programm sich bei der Ausführung auch tatsächlich verhält, wie in der Spezifikation gefordert. Also ist der Begriff „Korrektheit“ immer in Bezug zu der verwendeten DL-Programmsemantik zu sehen. Für die Semantik $\mathcal{S}_{OCL}$ bedeutet das, daß im

[1] Im Gegensatz zur tatsächlichen Ausführung auf einem Rechner steht die symbolische Ausführung in DL.

allgemeinen aus der Gültigkeit der Beweisverpflichtungen nicht folgt, daß das Programm der Spezifikation entspricht. Deshalb ist diese Semantik für die Verifikation von Programmen nicht geeignet. Trotzdem wird sie in der Literatur immer wieder verwendet (z. B. [Huisman, 2001]), wenn die Autoren nicht auf Details, wie z. B. die Endlichkeit der Datentypen, eingehen wollen.

5.2 Semantik $\mathcal{S}_{Java}$

Die zweite Semantik, genannt $\mathcal{S}_{Java}$, entspricht genau der Semantik, die in der JAVA-Sprachspezifikation [Bracha et al., 2000] definiert ist. Daher ist klar, daß $\mathcal{S}_{Java}$ und $\mathcal{S}_{JVM}$ äquivalent sind und deshalb folgt aus der Gültigkeit der DL-Beweisverpflichtungen auch, daß das Programm bei der tatsächlichen Ausführung der Spezifikation entspricht. Eigentlich sollte man denken, daß $\mathcal{S}_{Java}$ daher die „richtige“ Semantik für die Verifikation von Programmen ist. Allerdings hat $\mathcal{S}_{Java}$ auch einige Nachteile, auf die wir im folgenden eingehen werden. Diese Nachteile liegen zwar im Prinzip nicht an der Semantik selbst, sondern sie erwachsen eher aus der Kombination der Spezifikationssprache UML/OCL und der Implementierungssprache JAVA. Das Problem dabei ist nämlich, daß in der Spezifikation der OCL-Typ INTEGER, der einen unendlichen Wertebereich besitzt, verwendet wird. Für die Implementierung wird dann allerdings ein endlicher JAVA-Typ, wie z. B. `int`, eingesetzt. Wenn sich der Modellierer und/oder der Implementierer dieser Tatsache nicht bewußt sind, kann das zu Implementierungen führen, die nur „zufällig“ korrekt sind. Wie wir im folgenden anhand eines Beispiels zeigen werden, treten diese Probleme vor allem dann auf, wenn die Spezifikation unvollständig ist, d. h. zum Beispiel nur aus Zusicherungen besteht, die die Implementierung in jedem Fall erfüllen muß.

Angenommen es sei folgende unvollständige Spezifikation einer Operation gegeben, die den doppelten Wert des Aufrufparameters zurückgeben soll.

```
context Math::double(arg:Integer):Integer
post: (even(arg@pre) implies even(result)) and
      arg@pre <> result
```

Diese Spezifikation fordert nur, daß der Rückgabewert gerade sein muß, falls der Wert des Arguments gerade war und sich in jedem Fall davon unterscheiden muß.

Unserer Meinung nach ist folgende Implementierung der Methode, die die Zusicherungen der obigen Spezifikation erfüllt, nur „zufällig“ korrekt, falls der Programmierer sich der Probleme, die der *overflow* mit sich bringt, nicht bewußt war.

```
public int double(int arg) {
  int res=arg*2;
  return res;
}
```

In vielen Fällen verhält sich das Programm tatsächlich wie vermutlich vom Programmierer erwartet und liefert als Rückgabewert das Doppelte des Ausrufparameters zurück. Wird die Methode allerdings mit einem Wert, der größer als `MAX_int`/2 ist, aufgerufen, tritt bei der Multiplikation *overflow* auf und das Ergebnis entspricht nicht mehr dem doppelten Wert des Arguments. In diesem Fall ist das Ergebnis, wie in der OCL-Spezifikation gefordert, aber dennoch gerade und unterscheidet sich vom Wert des Arguments—allerdings nur aufgrund des glücklichen Umstands, daß das Produkt zweier gerader Zahlen, auch wenn *overflow* auftritt, wieder gerade ist.

Man könnte nun argumentieren, die Hauptsache ist, daß ein Programm korrekt ist und es keine Rolle spielt, ob „zufällig" oder nicht. Wir sind aber der Meinung, daß Programme, die nur „zufällig" korrekt sind, eine andauernde Fehlerquelle im Software-Entwicklungsprozeß darstellen, die es zu vermeiden gilt.

Angenommen obige OCL-Spezifikation wird um die zusätzliche Bedingung erweitert, daß der Rückgabewert positiv sein muß, wenn der Wert des Aufrufparameters positiv ist. Der Programmierer der obigen Implementierung ist sehr wahrscheinlich der Meinung, daß seine Implementierung die erweiterte Spezifikation ohne jede Änderung erfüllt. Mit dieser Meinung liegt er allerdings falsch, denn tritt *overflow* ein, ist des Ergebnis negativ obwohl das Argument positiv war. In diesem Falle ist es für den Programmierer nicht leicht den Fehler zu finden, denn das Programm verhält sich anders als er es erwartet. Wir glauben, daß unser Ansatz hilft, solche Fehler zu vermeiden.

Ein Nachteil, der wirklich an der Semantik $\mathcal{S}_{Java}$ selbst liegt, ist die Tatsache, daß Formeln wie $\forall x \exists y (y > x)$, die über den ganzen Zahlen intuitiv gültig sind, mit $\mathcal{S}_{Java}$ nicht mehr gelten, falls x, y einen „eingebauten" Typ, wie z. B. `int`, haben. Ferner sind auch DL-Formeln wie $\forall x \in \mathbb{Z}\{\mathtt{i} \leftarrow x\}\langle\mathtt{i=i;}\rangle\mathtt{i} \doteq \mathtt{i}$, deren Gültigkeit man eigentlich erwartet, plötzlich nicht mehr gültig, was irritierend ist.

5.3 Semantik $\mathcal{S}_{KeY}$

Nachdem wir nun also dargelegt haben, warum unserer Meinung nach sowohl $\mathcal{S}_{OCL}$ als auch $\mathcal{S}_{Java}$ für die Verifikation von Software nicht geeignet sind, stellen wir im folgenden unseren Ansatz vor. Natürlich will man die Eigenschaft von $\mathcal{S}_{Java}$, daß verifizierte Programme (d. h. wenn die Gültigkeit der DL-Beweisverpflichtungen nachgewiesen wurde) während der tatsächlichen Ausführung die spezifierten Eigenschaften erfüllen, beibehalten. Allerdings ist ein Ziel unserer Ansatzes zu verhindern, daß Programme, die nur „zufällig" korrekt sind, verifizierbar sind. Ferner sollten Formeln, die über den ganzen Zahlen intuitiv gültig sind, auch in unserer Semantik gültig sein.

Um diese Ziele zu erreichen, erweitern wir die Syntax von Java, wie in Kapitel 4 beschrieben, um die zusätzlichen primitive Typen `arithByte`, `arithShort`, `arithInt` und `arithLong`, die als *arithmetische* Typen bezeichnet werden, im Gegensatz zu den *eingebauten* Typen `byte`, `short`, `int` und `long`. Die exakte Semantik $\mathcal{S}_{KeY}$ der zusätzlichen arithmetischen Typen und der Operatoren, die darauf anwendbar sind, ist in Kapitel 4 definiert. Im Rest dieses Kapitels werden wir die Semantik $\mathcal{S}_{KeY}$ motivieren und auf die Unterschiede zu $\mathcal{S}_{OCL}$ und $\mathcal{S}_{Java}$ eingehen.

Der offensichtlichste Unterschied zwischen $\mathcal{S}_{KeY}$ auf der einen und $\mathcal{S}_{OCL}$ und $\mathcal{S}_{Java}$ auf der anderen Seite ist, daß sich die Signaturen der zugrunde liegenden Programmiersprachen aufgrund der erweiterten Syntax unterscheiden.

Grundsätzlich haben die arithmetischen Typen einen unendlichen Wertebereich und die Operatoren, die darauf anwendbar sind, haben, mit folgender Einschränkung, dieselbe Semantik wie in der Mathematik (und damit wie in $\mathcal{S}_{OCL}$): falls die Werte der Argumente im zulässigen Bereich sind (das bedeutet, daß die Werte in den entsprechenden eingebauten JAVA-Typen repräsentierbar sind), das Ergebnis aber nicht (d. h. ersetzte man die arithmetischen Typen durch die eingebauten Typen, dann verursachte die Operation *overflow*), dann wird implizit die unspezifizierte Methode `overflow` aufgerufen und das Ergebnis der Operation entspricht dem Rückgabewert dieses Methodenaufrufs. Dabei werden der Methode `overflow` die Werte der Argumente und der Operator als Parameter übergeben. Da `overflow` nicht spezifiziert ist, bedeutet das insbesondere, daß ein Aufruf von `overflow` nicht unbedingt terminiert.

Im folgenden werden wir erklären, warum wir keine konkrete Implementierung für die Methode `overflow` angeben, sondern sie unspezifiert lassen.

Der erste Grund dafür ist, wie wir gleich erklären werden, daß es keine konkrete Implementierung gibt, mit der sich die an unsere Semantik gestellten Anforderungen erreichen lassen. Prinzipiell gibt es drei mögliche Implementierungen für `overflow`:

- Die Methode `overflow` ist so implementiert, daß sie immer terminiert und ein Ergebnis zurückliefert, das dem mathematischen Ergebnis der Operation, die den Aufruf von `overflow` verursacht hat, entspricht. In diesem Fall hat man dieselben Probleme wie bei $\mathcal{S}_{OCL}$. Deshalb ist diese Implementierung nicht geeignet.

- Die Implementierung „simuliert“ *overflow*, d. h. daß der Rückgabewert von `overflow` entspricht dem Wert, den die Operation, die den Aufruf von `overflow` verursacht hat, in $\mathcal{S}_{Java}$ (und damit auch bei der Ausführung auf der virtuellen Maschine) hätte. Die Semantik, die durch diese Implementierung entsteht, bezeichnen wir im folgenden als $\mathcal{S}_{JVM'}$. Diese Implementierung von `overflow` hat den Nachteil, daß, wie mit $\mathcal{S}_{Java}$, Programme verifiziert werden können, die nur „zufällig“ korrekt sind. Diesen Nachteil wollten wir mit unserer Semantik ja gerade beheben und deshalb ist auch diese Implementierung nicht geeignet.

- Die Methode `overflow` ist so implementiert, daß ihr Aufruf in keinem Fall terminiert. Diese Implementierung hat den Nachteil, daß Formeln wie

$$[\texttt{arithInt i=MAX\_int+1;}]\texttt{i} \doteq 5$$

trivialerweise gültig sind, da *overflow* eintritt und damit das Programm nicht terminiert. Allerdings terminiert solch ein Programm bei der tatsächlichen Ausführung und damit hat man wieder dasselbe Problem wie in $\mathcal{S}_{OCL}$, daß ein verifiziertes Programm bei der tatsächlichen Ausführung die spezifizierten Eigenschaften im allgemeinen nicht erfüllt.

Wie man sieht führen alle drei angesprochenen Implementierungen zu einer Semantik, mit der sich unserer Ziele nicht erreichen lassen. Ein weiterer Grund, warum wir `overflow` unspezifiziert lassen, betrifft den Zusammenhang zwischen einer Formel, die im Kalkül, der in Kapitel 6 definiert ist, ableitbar ist und dem Verhalten eines Programms, das in dieser Formel enthalten ist, bei seiner tatsächlichen Ausführung. Bevor wir näher darauf eingehen können, bedarf es einiger Definitionen.

Da die arithmetischen Typen in $\mathcal{S}_{KeY}$ und also auch in $\mathcal{S}_{JVM'}$ (was ja eine „Instanz" von $\mathcal{S}_{KeY}$ ist) einen unendlichen Wertebereich haben, gibt es in der DL Zustände, die es auf der virtuellen Maschine nicht gibt. Dies trifft genau dann auf einen Zustand zu, wenn in diesem eine Variable eines arithmetischen Typs einen Wert besitzt, der nicht im entsprechenden eingebauten Java-Typ repräsentierbar ist. Solche Zustände werden als *irreale Zustände* bezeichnet, im Gegensatz zu *realen* Zuständen, bei denen alle Werte von Variablen eines arithmetischen Typs im entsprechenden eingebauten Java-Typ repräsentierbar sind.

Definition 5.3.1 (Realer Zustand)
Sei s ein beliebiger DL-Zustand.

Der Zustand s heißt realer Zustand*, falls die Werte aller Variablen, von denen s abhängt und die einen arithmetischen Typ haben, im entsprechenden eingebauten* Java-*Typ repräsentierbar sind.*

Definition 5.3.2 (Irrealer Zustand)
Ein Zustand, der nicht real ist, heißt irrealer Zustand.

Man beachte, daß unsere Semantik $\mathcal{S}_{KeY}$ so definiert ist, daß in irrealen Zuständen die Methode `overflow` nicht aufgerufen wird, falls *overflow* eintritt, da solche Zustände von der virtuellen Maschine nicht erreicht werden können und es damit egal ist, was das Ergebnis der Operation ist.

Bevor ein Programm, das arithmetische Typen enthält, tatsächlich übersetzt und ausgeführt werden kann, müssen die arithmetischen Typen durch die entsprechenden eingebauten Typen ersetzt werden.

Definition 5.1 (Typ-Transformation)
Sei p ein Programm, das arithmetische Typen enthalten kann.

Das Programm $ptransf(p)$ entsteht aus p, indem alle arithmetischen Typen durch die entsprechenden eingebauten Typen ersetzt werden.

Die Folge dieser Typ-Transformation ist natürlich, das daß Programm, das verifiziert wurde, nicht mehr mit dem Programm, das tatsächlich ausgeführt wird, übereinstimmt. Welche Beziehungen dennoch zwischen beiden Programmen bestehen, drücken folgende Sätze aus, deren Beweise in Kapitel 5 in [Schlager, 2002] zu finden sind.

Satz 5.2
Gegeben sei ein JAVA-*Programm p, das arithmetische Typen enthalten kann.*

Falls p richtig getypt ist, dann ist auch das transformierte Programm $ptransf(p)$ richtig getypt.

Um das Verhalten eines transformierten Programms $ptransf(p)$ während der Ausführung auf der virtuellen Maschine zu beschreiben, brauchen wir den Begriff des *isomorphen Zustands.*

Definition 5.3 (Isomorpher Zustand)
Sei s ein realer Zustand und sei Var die Menge aller Variablen eines arithmetischen Typs, von denen s abhängt.

Der zu s isomorphe Zustand *$iso(s)$ ist der Zustand, in dem jede Variable aus Var den entsprechenden eingebauten Typ hat und denselben Wert besitzt wie in s.*

Für den folgenden Satz ist die Tatsache, daß die Methode `overflow` unspezifiziert ist, von zentraler Bedeutung.

Satz 5.4
Sei p ein JAVA-*Programm das arithmetische Typen enthalten kann, sei $ptransf(p)$ das transformierte Programm und seien Γ und ψ reine prädikatenlogische Formeln.*

Falls $\vdash_{\mathcal{C}_{DLJ}} \phi$, dann gelten folgende Aussagen:

1. $\models_{\mathcal{S}_{OCL}} \phi$
2. $\models_{\mathcal{S}_{JVM'}} \phi$.

Den Zusammenhang zwischen der Ausführung des Programms p in DL, basierend auf Semantik $\mathcal{S}_{JVM'}$, und der Ausführung des transformierten Programms $ptransf(p)$ auf der virtuellen Maschine, drückt der folgende Satz aus.

Satz 5.5
Sei p ein JAVA-*Programm, das arithmetische Typen enthalten kann.*

Dann gilt für alle realen Zustände s und alle beliebigen Zustände s':

$$\textit{Wenn}\ \ s\,[\![p]\!]_{\mathcal{S}_{JVM'}}\,s'\ \ \textit{dann}\ \ iso(s)\,[\![ptransf(p)]\!]_{\mathcal{S}_{JVM}}\,iso(s').$$

Das folgende Korollar drückt aus, daß, falls $\Gamma \rightarrow \langle p \rangle \psi$ in unserem Kalkül ableitbar ist und das Programm p in einem realen Zustand s mit $s \models_{\mathcal{S}_{KeY}} \Gamma$ gestartet wird, das transformierten Programms $ptransf(p)$, gestartet in $iso(s)$, während der Ausführung auf der virtuellen Maschine kein *overflow* verursacht und im Zustand $iso(s')$ terminiert.

Korollar 5.6
Seien Γ, ψ rein prädikatenlogische Formeln, sei p ein Programm, das arithmetische Typen enthalten kann und sei s ein beliebiger Zustand.

Falls gilt,

- $\vdash_{\mathcal{C}_{DLJ}} \Gamma \rightarrow \langle p \rangle \psi$,
- $s \models_{\mathcal{S}_{KeY}} \Gamma$,
- *s ist ein realer Zustand und*
- *p wird in s gestartet,*

dann gilt für alle Zustände s' mit $s\, [\![p]\!]_{\mathcal{S}_{KeY}}\, s'$

$$iso(s') \models \psi,$$

wobei $iso(s')$ der zu s isomorphe Zustand ist.[2] *Insbesondere gilt, daß während der Ausführung des transformierten Programms $ptransf(p)$ auf der virtuellen Maschine kein overflow eintritt.*

Man beachte, daß die Aussage des Korollars 5.6 nicht für beliebige Formeln gilt, da z. B. eine Formel wie $[p]true$ immer ableitbar ist, egal ob *overflow* eintritt oder nicht. Allerdings sind die Beweisverpflichtungen, die durch das Verfahren aus [Keller, 2002] erzeugt werden, normalerweise von der Form, daß Korollar 5.6 anwendbar ist.

Wie schon erwähnt ist eine Forderung an unsere Semantik, die intuitive Gültigkeit von Formeln über den ganzen Zahlen beizubehalten. Semantik $\mathcal{S}_{KeY}$ erfüllt diese Forderung für Variablen der arithmetischen Typen, d. h. eine Formel wie $\forall x \exists y (y > x)$ ist gültig, wenn x, y einen arithmetischen Typ haben.

Die eingebauten JAVA-Typen haben in Semantik $\mathcal{S}_{KeY}$ dieselbe Semantik wie in $\mathcal{S}_{Java}$ und damit ist es auch in $\mathcal{S}_{KeY}$ möglich, Effekte, die durch *overflow* ausgelöst werden, auszunutzen. Allerdings sollte das in der Spezifikation explizit vermerkt werden und es obliegt dann der Verantwortung des Programmierers sicherzustellen, daß keine unerwünschten Effekte aufgrund von *overflow* eintreten.

Einen Vergleich der drei diskutierten Semantiken findet man in Tabelle 5.1 in [Schlager, 2002].

[2] $s\, [\![p]\!]_{\mathcal{S}_{KeY}}\, s'$ bedeutet, daß es eine Implementierung von `overflow` gibt mit $s\, [\![p]\!]_{\texttt{overflow}}\, s'$

5.3.1 Software-Entwicklung mit unserem Ansatz

Nachdem wir im letzten Abschnitt unseren Ansatz vorgestellt haben, wollen wir zum Abschluß dieses Kapitels kurz auf die einzelnen Schritte der Software-Entwicklung eingehen, wie wir sie uns vorstellen.

1. *Spezifikation:* In der UML/OCL-Spezifikation werden Eigenschaften über ganze Zahlen unter Verwendung des OCL-Typs INTEGER spezifiziert.

2. *Implementierung:* Eine Operation, die mit INTEGER spezifiziert ist, wird unter Verwendung der arithmetischen Typen `arithByte`, `arithShort`, `arithInt` oder `arithLong` implementiert.

3. *Verifikation:* Die Gültigkeit der Beweisverpflichtungen, die mit dem Verfahren aus [Keller, 2002] erzeugt werden, muß mit Hilfe unseres Kalküls, der auf $\mathcal{S}_{KeY}$ basiert, nachgewiesen werden. Wenn die Gültigkeit aller Beweisverpflichtungen nachgewiesen wurde, dann folgt daraus, daß das Programm während der Ausführung auf der virtuellen Maschine der Spezifikation entspricht und insbesondere bei JAVA-Ausdrücken, die unter Verwendung der arithmetischen Typen implementiert wurden, kein *overflow* auftritt.

6 Ein Sequenzenkalkül für Java Integer-Arithmetik

In diesem Abschnitt stellen wir einen Sequenzenkalkül für eine Dynamische Logik für JAVA Integer-Arithmetik basierend auf $\mathcal{S}_{KeY}$ vor. Aus Platzgründen können wir den Kalkül nur anhand einiger Regeln exemplarisch vorstellen und verweisen für die übrigen Regeln auf Kapitel 6 in [Schlager, 2002]. Der gesamte Kalkül in [Schlager, 2002] umfaßt insgesamt knapp 40 Regeln. Im einzelnen beinhaltet der Kalkül Regeln für die Vereinfachung komplexer arithmetischer Ausdrücke, Regeln für die symbolische Ausführung einfacher, d. h. nicht-komplexer arithmetischer Ausdrücke, Regeln für die Behandlung von Type-Casts, Regeln für die Behandlung von Vergleichsoperatoren und Regeln für logische Operatoren.

Grundsätzlich besteht der Kalkül sowohl aus Regeln, die eine Programmtransformation durchführen, als auch aus Regeln, die eine Programmanweisung symbolisch ausführen.

6.1 Definitionen

Bevor wir die Regeln des Kalküls präsentieren, bedarf es zunächst der Definition einiger Begriffe und Symbole, die in den Regeln des Kalküls verwendet werden.

Definition 6.1.1 (Metavariablen)
Die Metavariablen `slhs`, `srhs` und `expr` *sind wie folgt definiert.*

Eine `slhs` *(simple left-hand side) steht für eine lokale Variable, ein Attribut, eine Klassenvariable oder einen Array-Zugriff, wobei die Array-Referenz kein Methodenaufruf sein darf und die Indizes* `srhs` *sein müssen.*

Eine `srhs` *(simple right-hand side) ist dasselbe wie eine* `slhs`*, kann aber zusätzlich auch noch für ein* JAVA*-Literal stehen.*

Eine `expr` *(expression) steht für einen beliebigen* JAVA*-Ausdruck.*

Definition 6.1.2 („Neue“ Variable)
Wenn bei einer Kalkülregel die Einführung einer „neuen“ Variable *verlangt wird, dann bedeutet das, daß die Variable neu bezüglich der Konklusion der Regel sein muß, d. h. noch nicht in der Konklusion vorkommen darf.*

Definition 6.1.3 (Abkürzungen in Programmen)
Die griechischen Buchstaben π, ω und μ sind Abkürzungen mit folgender Bedeutung.

π und ω stehen für Programmteile, so daß π nur Klammern, Marken (Labels), `try`*,* `catch`*,* `return` *und Methodenaufrufen besteht und $\pi\ \omega$ ein korrektes Programm ist.*

μ ist eine Menge von Updates, die einer Formel zugeordnet sind (insbesondere kann μ auch leer sein).

Definition 6.1.4 (Prädikate und Funktionen)
Sei simpleExpr eine `slhs` *oder eine* `srhs`*, sei* `T` *eine primitiver ganzzahliger Typ, sei t ein logischer Term von einem dieser Typen und seien* `MIN_T` *und* `MAX_T` *der kleinste bzw. größte mögliche Wert des Typs* `T`*. Ferner sei* `T'` *der dem Typ* `T` *entsprechende eingebaute Typ ist, falls* `T` *ein arithmetischer Typ ist. Anderenfalls ist* `T'` *gleich* `T`*.*

Das einstellige Prädikat isdef(simpleExpr) ist erfüllt, gdw. die Auswertung von simpleExpr keine Exception verursacht.

Das einstellige Prädikat $in_T(t)$ ist eine Abkürzung für die Formel

$$\mathtt{MIN\_}T' \leq t \wedge t \leq \mathtt{MAX\_}T'.$$

6.2 Regeln des Kalküls

Als Beispiel für Regeln, die eine Programmtransformation durchführen, zeigen wir folgende Regel, die komplexe Ausdrücke in einfachere Ausdrücke zerlegt damit diese dann später symbolisch abgearbeitet werden können. Dies ist erforderlich, da komplexe Ausdrücke zahlreiche Seiteneffekte beinhalten können.

Regel 6.2.1 *Vereinfachung komplexer Additions-Ausdrücke*

Zweck Diese Regel transformiert einen komplexen Ausdruck, dessen Operator mit der höchsten Priorität die Addition ist, in einfachere Ausrücke.

Voraussetzungen
- Mindestens einer der Ausdrücke $expr_1$ und $expr_2$ ist ein komplexer Ausdruck, d. h. keine `srhs`.

Bemerkung Die Regel für den Box-Operator $[\cdot]$ ist analog.

$$\frac{\begin{array}{ll} 1. & \Gamma \vdash isdef(\mathtt{slhs}),\ \Delta \\ 2. & \Gamma \vdash \mu\langle\pi\ y\ \mathtt{=}\ expr_1;\ z\ \mathtt{=}\ expr_2;\ \mathtt{slhs}\ \mathtt{=}\ y\ \mathtt{+}\ z;\ \omega\rangle\phi,\ \Delta \end{array}}{\Gamma \vdash \mu\langle\pi\ \mathtt{slhs}\ \mathtt{=}\ expr_1 \mathtt{+}\ expr_2;\ \ \omega\rangle\phi,\ \Delta}$$

Beschreibung
- y ist eine neue Variable, die denselben Typ wie der Ausdruck $expr_1$ hat.

- $\mathtt{z}$ ist eine neue Variable, die denselben Typ wie der Ausdruck $\mathtt{expr}_2$ hat.

Die Voraussetzung, daß mindestens einer der beiden Argumente ein komplexer Ausdruck sein muß, garantiert, daß diese Regel nur dann angewendet werden kann, wenn tatsächlich noch eine Vereinfachung des Gesamtausdrucks möglich ist. Damit wird verhindert, daß diese Regel beliebig oft hintereinander angewendet werden kann.

Als Beispiel für Regeln, die eine Anweisung symbolisch abarbeiten, zeigen wir zwei Regeln für die Zuweisung, da diese das wichtigste Programmkonstrukt darstellt.

Regel 6.2.2 *Einfache Zuweisung*

Zweck Symbolische Ausführung einer Zuweisung mit einfacher rechter Seite.

Bemerkung Die Regel für den Box-Operator $[\cdot]$ ist analog.

$$\begin{array}{l}
1.\ \Gamma \vdash \mathit{isdef}(\mathtt{slhs}),\ \Delta \\
2.\ \Gamma \vdash \mathit{isdef}(\mathtt{srhs}),\ \Delta \\
3.\ \Gamma \vdash (\{\mathtt{slhs} \leftarrow \mathtt{srhs}\} \circ \mu)\langle \pi\ \omega\rangle\phi,\ \Delta \\
\hline
\quad\ \Gamma \vdash \mu\langle \pi\ \mathtt{slhs = srhs;}\ \ \omega\rangle\phi,\ \Delta
\end{array}$$

Beschreibung Diese Regel führt symbolisch eine Zuweisung mit einfacher rechter Seite aus, was eine Zustandsänderung bewirkt, welche durch das Update $\{\mathtt{slhs} \leftarrow \mathtt{srhs}\}$ beschrieben wird.

Eine komplexere Regel ist die folgende, die eine Addition auf arithmetischen Typen symbolisch abarbeitet.

Regel 6.2.3 *Arithmetische Addition rechts*

Zweck Symbolische Abarbeitung einer Addition eines arithmetischen Typs für den *diamond* Operator.

Voraussetzungen

- Der Ausdruck $\mathtt{srhs}_1\ \mathtt{op}\ \mathtt{srhs}_2$ muß einen arithmetischen Typ haben.

$$\begin{array}{l}
1.\ \Gamma \ \vdash\ \mathit{isdef}(\mathtt{slhs}),\ \Delta \\
2.\ \Gamma \ \vdash\ \mathit{isdef}(\mathtt{srhs}_1),\ \Delta \\
3.\ \Gamma \ \vdash\ \mathit{isdef}(\mathtt{srhs}_2),\ \Delta \\
4.\ \Gamma,\ \mathit{in}_{T_1}(\mathtt{srhs}_1),\ \mathit{in}_{T_2}(\mathtt{srhs}_2),\ \mathit{in}_{T}(\mathtt{srhs}_1 + \mathtt{srhs}_2)\ \vdash \\
\qquad (\{\mathtt{slhs} \leftarrow \mathtt{srhs}_1 + \mathtt{srhs}_2\} \circ \mu)\langle \pi\ \omega\rangle\phi,\ \Delta \\
5.\ \Gamma,\ \mathit{in}_{T_1}(\mathtt{srhs}_1),\ \mathit{in}_{T_2}(\mathtt{srhs}_2),\ \neg \mathit{in}_{T}(\mathtt{srhs}_1 + \mathtt{srhs}_2)\ \vdash \\
\qquad \mu\langle \pi\ \mathtt{slhs}\ \texttt{= overflow(}\mathtt{srhs}_1\texttt{,}\mathtt{srhs}_2\texttt{,"+");}\ \ \omega\rangle\phi,\ \Delta \\
6.\ \Gamma,\ \neg(\mathit{in}_{T_1}(\mathtt{srhs}_1)\ \wedge \mathit{in}_{T_2}(\mathtt{srhs}_2))\ \vdash \\
\qquad (\{\mathtt{slhs} \leftarrow \mathtt{srhs}_1\ +\ \mathtt{srhs}_2\} \circ \mu)\langle \pi\ \omega\rangle\phi,\ \Delta \\
\hline
\quad \Gamma \ \vdash\ \mu\langle \pi\ \mathtt{slhs}\ \texttt{=}\ \mathtt{srhs}_1\ \texttt{+}\ \mathtt{srhs}_2\texttt{;}\ \ \omega\rangle\phi,\ \Delta
\end{array}$$

Beschreibung

- T_1 ist der Typ von $\mathtt{srhs}_1$.
- T_2 ist der Typ von $\mathtt{srhs}_2$.
- T ist der Typ von $\mathtt{srhs}_1$ + $\mathtt{srhs}_2$.

Für die Erklärung der Prämissen nehmen wir an, daß Γ gilt und Δ nicht, da die Regel sonst trivialerweise anwendbar ist.

Die ersten drei Prämissen stellen sicher, daß die Zugriffe auf die beteiligten Java-Ausdrücke definiert sind.

Die vierte Prämisse drückt aus, daß die Addition und die Zuweisung symbolisch abgearbeitet werden, wenn die Argumente alle im zulässigen Bereich sind und das Ergebnis ebenso (d. h. es tritt kein *overflow* auf). Daraus resultiert eine Zustandsänderung, die durch das *update* $\{\mathtt{slhs} \leftarrow \mathtt{srhs}_1 + \mathtt{srhs}_2\}$ beschrieben wird.

Die fünfte Prämisse drückt aus, daß die unspezifizierte Methode `overflow` aufgerufen wird, wenn die Argumente im zulässigen Bereich sind aber das Ergebnis nicht (d. h. es tritt *overflow* auf). Da man nicht weiß, ob `overflow` terminiert, ist die Formel $\langle \pi\ \mathtt{slhs}\ \texttt{= overflow(}\mathtt{srhs}_1\texttt{,}\mathtt{srhs}_2\texttt{,"}\mathtt{op}\texttt{");}\ \ \omega\rangle\phi$ nicht gültig und könnte deshalb auch weggelassen werden. Allerdings ist es durch diese Formel möglich, die Semantik $\mathcal{S}_{KeY}$ zu verändern, indem man `overflow` durch gewisse Axiome spezifiziert, ohne daß man daraufhin die Regeln des Kalküls ändern muß.

Die sechste Prämisse deckt den Fall ab, daß bereits mindestens eines der Argumente nicht im zulässigen Bereich ist. Da man sich in diesem Falle bereits in einem irrealen Zustand befindet, wird die Methode `overflow` nicht aufgerufen, weil solch ein Zustand von der virtuellen Maschine nicht erreichbar ist und daher für das Verhalten eines Programms bei dessen tatsächlicher Ausführung irrelevant ist.

Eine äquivalente Variante der Regel erhält man, wenn man die Prämissen vier und sechs durch die einzelne Prämisse

$$\Gamma,\ in_T(\texttt{srhs}_1\ \texttt{logOp}\ \texttt{srhs}_2) \vdash \\ (\{\texttt{slhs} \leftarrow \texttt{srhs}_1\ \texttt{logOp}\ \texttt{srhs}_2\} \circ \mu)\langle \pi\ \omega\rangle\phi$$

ersetzt, da aus obiger Sequenz die beiden Prämissen durch Anwendung der Schnitt-Regel ableitbar sind.

7 Implementierung

In diesem Abschnitt gehen wir auf den praktischen Teil dieser Diplomarbeit, die Implementierung des Kalküls, ein.

Für die Implementierung der Kalkülregeln wird die in [Habermalz, 2000] präsentierte Sprache der *schematischen theoriespezifischen Regeln (STSR)* verwendet. In dieser Sprache ist es möglich, Wissen über bestimmte Theorien in der Form von Beweisregeln auszudrücken. Genauer gesagt enthält eine STSR neben rein logischem Wissen auch Informationen darüber, wann und wo dieses Wissen beim automatischen oder interaktiven Beweisen angewendet werden kann. Eine prototypische Implementierung eines Beweisers basierend auf STSRs ist IBIJa (Interaktiver Beweiser in Java) [Supp, 1998].

Eine STSR besteht im wesentlichen aus drei Teilen. Im ersten Teil wird beschrieben, in welcher Situation die Regel anwendbar ist, der zweite Teil beinhaltet die logische Information der Regel und der dritte Teil gibt an, zu welchen Heuristiken die Regel gehört und damit automatisch angewendet werden kann, wenn die betreffenden Heuristiken aktiviert sind. Abbildung 7.1 in [Schlager, 2002] zeigt die allgemeine Struktur einer STSR. Aus Platzgründen gehen wir nicht näher auf die einzelnen Teile ein, sondern erklären deren Bedeutung anhand eines Beispiels.

Als Beispiel dient Regel 6.2.3 aus dem vorherigen Kapitel. Da, wie in der Beschreibung dieser Regel erwähnt, die Formel

$$\mu\langle\pi\ \texttt{slhs = overflow(}\mathit{srhs}_1\texttt{,}\mathit{srhs}_2\texttt{,"}\mathit{op}\texttt{");}\ \ \omega\rangle\phi$$

in $\mathcal{S}_{KeY}$ nicht gültig ist, lassen wir sie der Einfachheit halber weg. Damit läßt sich folgende kürzere Variante der Regel ableiten.

$$\begin{array}{l}
1.\ \Gamma\ \vdash\ \mathit{isdef}(\texttt{slhs}),\ \Delta \\
2.\ \Gamma\ \vdash\ \mathit{isdef}(\texttt{srhs}_1),\ \Delta \\
3.\ \Gamma\ \vdash\ \mathit{isdef}(\texttt{srhs}_2),\ \Delta \\
4.\ \Gamma,\ \mathit{in}_{T_1}(\texttt{srhs}_1),\ \mathit{in}_{T_2}(\texttt{srhs}_2)\ \vdash\ \mathit{in}_T(\texttt{srhs}_1+\texttt{srhs}_2),\ \Delta \\
5.\ \Gamma\ \vdash\ (\{\texttt{slhs} \leftarrow \texttt{srhs}_1\ +\texttt{srhs}_2\}\circ\mu)\langle\pi\ \omega\rangle\phi,\ \Delta \\
\hline
\quad\ \Gamma\ \vdash\ \mu\langle\pi\ \texttt{slhs = srhs}_1\texttt{ + srhs}_2\texttt{;}\ \ \omega\rangle\phi,\ \Delta
\end{array}$$

Als STSR sieht diese Regel folgendermaßen aus, wobei die ersten drei Prämissen in der STSR nicht berücksichtigt werden, da die Prädikate *isdef*, das die Definiertheit der beteiligten Java-Ausdrücke überprüft, zum momentanen Entwicklungsstand des KeY-Beweisers noch nicht behandelt werden können.

```
assignment_arithmetical_addition {
  find (==> <{.. #slhs = #se0 + #se1; ...}>post)
  replacewith (==> {#slhs:=+(#se0,#se1)}<{.. ... }>post);
  replacewith (#inType(#se0) & #inType(#se1) ==> #inType(#se0 + #se1))
  heuristics(simplify_expression)
};
```

Der `find`-Teil in der STSR entspricht der Konklusion der Sequenzenregel und bestimmt die Anwendbarkeit der Regel, d. h. diese STSR ist anwendbar auf eine Sequenz, die der im `find`-Teil beschriebenen Form entspricht.[1] Wird diese STSR auf eine Sequenz angewendet, dann werden dadurch zwei neue Sequenzen erzeugt, und zwar indem der durch den `find`-Teil beschriebene Teil der Sequenz durch die in den `replacewith`-Teilen beschriebenen ersetzt wird.

Im `heuristics`-Teil kann festgelegt werden, welchen Heuristiken eine Regel angehören sollen. Aktiviert der Benutzer beim Beweisen die entsprechenden Heuristiken, kann diese Regel auch automatisch angewendet werden kann.

[1] Dies macht noch einmal sehr deutlich, daß die Beweiskonstruktion in einem Sequenzenkalkül von „unten“ nach „oben“ verläuft, d. h. man beginnt mit der Sequenz, die man eigentlich ableiten will.

8 Zusammenfassung und Ausblick

8.1 Zusammenfassung

In dieser Diplomarbeit haben wir einen Ansatz zur Behandlung der Integer-Arithmetik bei der Verifikation von Java-Programmen vorgestellt.

Unser Ansatz basiert auf einer Erweiterung der Java-Syntax um sogenannte arithmetische Typen und der Semantik $\mathcal{S}_{KeY}$ für diese erweiterte Sprache, die dazu beitragen zu verhindern, daß Programme aufgrund von unbeabsichtigtem *overflow* nur „zufällig“ korrekt sind. Wie in Kapitel 5 dargelegt bilden solche „zufällig“ korrekten Programme im Software-Entwicklungsprozeß eine permanente Fehlerquelle, die vermieden werden sollte.

In Kapitel 6 haben wir einen Sequenzenkalkül für eine Dynamische Logik für Java Integer-Arithmetik, basierend auf der erweiterten Syntax und Semantik, angegeben. Die Regeln dieses Kalküls sind in der Sprache der *schematischen theoriespezifischen Regeln (STSR)* implementiert und in das KeY-System integriert worden.

8.2 Ausblick

Da sich das KeY-System noch in der Entwicklung befindet, war es nicht möglich, alle Regeln des Kalküls vollständig in das System zu integrieren. Zum Beispiel ist die Behandlung des Prädikats *isdef* zum gegenwärtigen Zeitpunkt noch nicht möglich. Allerdings wurde mit der Beseitigung dieses Problems bereits begonnen.

Ein weiteres interessantes Thema ist die Erweiterung der OCL um zusätzliche Datentypen. Der einzige, derzeit in OCL enthaltene ganzzahlige Datentyp ist Integer, welcher einen unendlichen Wertebereich hat. Es wäre vorteilhaft, in der OCL Datentypen zur Verfügung zu haben, die den in Programmiersprachen enthalten endlichen Datentypen entsprechen. Eine solche Erweiterung erlaubt eine präzisere Spezifikation, da schon bei der Modellierung festgelegt werden kann, welche Typen bei der Implementierung verwendet werden sollen. Ein weiterer Vorteil ist, daß damit auf der Modellierungsebene Typen vorhanden sind, die sowohl den eingebauten Java-Typen wie z. B. `int` als auch den zusätzlichen arithmetischen Typen wie z. B. `arithInt` entsprechen.

Literaturverzeichnis

[Ahrendt et al., 2000] Ahrendt, W., Baar, T., Beckert, B., Giese, M., Habermalz, E., Hähnle, R., Menzel, W., and Schmitt, P. H. (2000). The KeY approach: Integrating object oriented design and formal verification. In Ojeda-Aciego, M., Guzman, I. P. d., Brewka, G., and Pereira, L. M., editors, *Proceedings, Logics in Artificial Intelligence (JELIA), Malaga, Spain*, LNCS 1919. Springer.

[Beckert, 2001] Beckert, B. (2001). A dynamic logic for the formal verification of Java Card programs. In Attali, I. and Jensen, T., editors, *Java on Smart Cards: Programming and Security. Revised Papers, Java Card 2000, International Workshop, Cannes, France*, LNCS 2041, pages 6–24. Springer.

[Bracha et al., 2000] Bracha, G., Gosling, J., Joy, B., and Steele, G. (2000). *The Java Language Specification, Second Edition.* Addison Wesley.

[Habermalz, 2000] Habermalz, E. (2000). Interactive theorem proving with schematic theory specific rules. Technischer Bericht 19/00, Fakultät für Informatik, Universität Karlsruhe.

[Harel, 1979] Harel, D. (1979). *First-order Dynamic Logic.* LNCS 68. Springer.

[Huisman, 2001] Huisman, M. (2001). *Java Program Verification in Higher-Order Logic with PVS and Isabelle.* Dissertation, Universität Nijmegen, Niederlande.

[Keller, 2002] Keller, U. (2002). Übersetzung von OCL Constraints in Formeln einer Dynamischen Logik für Java Card. Diplomarbeit, Universität Karlsruhe.

[Schlager, 2002] Schlager, S. (2002). Handling of Integer Arithmetic in the Verification of Java Programs. Diplomarbeit, Universität Karlsruhe.

[Supp, 1998] Supp, M. (1998). Implementierung eines integriert interaktiven und reflexiven Beweisers für Prädikatenlogik. Diplomarbeit, Universität Karlsruhe.