Ein Bytecode–Verifier zur Verifikation von Betriebssystemkomponenten

Diplomarbeit im Fach Informatik

vorgelegt von

Martin Alt

geboren am 12. Mai 1978 in Nürnberg

Angefertigt am Institut für Informatik (IV)
Friedrich-Alexander-Universität Erlangen-Nürnberg

Betreuer:	*Prof. Dr. rer. nat. Fridolin Hofmann*
	Dipl. Inf. Michael Golm
Beginn der Arbeit:	2. Januar 2001
Abgabe der Arbeit:	2. Juli 2001

Ich versichere, daß ich meine Arbeit ohne fremde Hilfe und ohne Benutzung anderer als der angegebenen Quellen angefertigt habe und dass die Arbeit in gleicher oder ähnlicher Form noch keiner anderen Prüfungsbehörde vorgelegen hat und von dieser als Teil einer Prüfungsleistung angenommen wurde. Alle Ausführungen, die wörtlich oder sinngemäß übernommen wurden, sind als solche gekennzeichnet.

Erlangen, den 2. Juli 2001 ____________________

Kurzfassung

Ziel dieser Arbeit war die Entwicklung eines Bytecode-Verifiers für das Java-Betriebssystem JX. Dieser Verifier wird vor der Übersetzung einer Komponente nach Maschinensprache aufgerufen, um den Bytecode zu analysieren und verschiedene Anforderungen zu verifizieren. Damit werden verschiedene Ziele verfolgt:

Zum einen überprüft der Verifier die in der Java Virtual Machine Spezifikation aufgeführten Anforderungen. Diese betreffen im wesentlichen die Typsicherheit, sowie die Einhaltung von Zugriffsbeschränkungen bei Zugriffen auf Felder und Methoden.

Neben diesen „Standard-Anforderungen" werden weitere, JX-spezifische Anforderungen geprüft. Zum Beispiel darf eine Interrupt Behandlungsroutine nur dann installiert werden, wenn sichergestellt ist, dass sie innerhalb einer gewissen Zeit terminiert. Zu diesem Zweck wurde eine Analyse entwickelt, die es für viele Methoden erlaubt, eine Abschätzung der maximalen Ausführungsdauer statisch zu bestimmen. Da dieses Problem jedoch im Allgemeinen unentscheidbar ist, können nicht alle Methoden analysiert werden. Die nicht analysierbaren Methoden werden durch neue ersetzt, die Laufzeitüberprüfungen enthalten, welche eine Terminierung innerhalb der Zeitvorgabe sicherstellen.

Außerdem hat der Verifier die Aufgabe, Analysen durchzuführen, die der Optimierung dienen. Beispielsweise wird eine „Null-Pointer Analyse" durchgeführt, um statisch zu überprüfen ob eine Referenz den Wert null hat oder nicht. Kann zur Verifikationszeit festgestellt werden, dass die Referenz ungleich null ist, so kann eine entsprechende Laufzeitüberprüfung entfallen.

Um die verschiedenen Verifikationsschritte zu vereinheitlichen wurde eine allgemeine Vorgehensweise für die Bytecodeanalyse implementiert.

Abstract

The aim of this thesis was to develop a Java Bytecode verifier for the JX Java operating system. Before compiling a component to native code and executing it, this component verifier is invoked to check several constraints on the bytecode. The verifier has several tasks:

First of all, standard Java Bytecode verification was implemented. It guarantees that the constraints described by the JVM Specification are met. Most of these are concerned with type-safety and the restrictions on field and method access imposed by access-flags.

Besides those Java rules, there are additional constraints the verifier ensures in JX. For example, an interrupt handler can only be installed after verifying that the handler terminates within a certain time bound. To this end, a worst case execution time analysis was developed, which allows to statically estimate the execution time for a wide range of methods. However, as this problem is in general undecidable, not all methods can be analyzed. Those methods that cannot be analyzed are replaced with new ones, which contain runtime-checks to ensure termination within a time limit.

Another aim of the verifier is to analyse the code to find optimisation possibilities. For example, a 'null-pointer analysis' can be performed, which checks whether a reference is valid or not. If the reference is valid, the compiler can omit runtime checking this reference.

To even the different verification processes, a framework for bytecode analysis was implemented.

Inhaltsverzeichnis

Abbildungsverzeichnis

1 Einleitung

Im Rahmen dieser Arbeit wurde ein Bytecode-Verifier für das Java-Betriebssystem JX (siehe [MG01]) implementiert, der die Aufgabe hat, Java Bytecode zu analysieren und verschiedene Annahmen zu verifizieren.

Das Speicherschutzkonzept von JX basiert nicht auf Hardwaremechansimen, sondern auf den durch Java zur Verfügung gestellten Schutzmechanismen. Damit diese Schutzmechanismen nicht umgangen werden können, muss der Bytecode eine Reihe von Anforderungen erfüllen, damit er ausgeführt werden kann. Die Einhaltung der Bedingungen muss vor Ausführung jeder Methode durch den Verifier überprüft werden.

Neben dieser Standard Bytecode-Verifikation, die von jeder Java Virtual Machine Implementierung durchgeführt werden muss, soll der im Rahmen dieser Arbeit implementierte Verifier noch weitere, JX spezifische Bedingungen prüfen. Insbesondere sollen Methoden, die ununterbrechbar ausgeführt werden müssen (z. B. Interrupthandler) analysiert werden, um sicherzustellen, dass sie bei der Ausführung spätestens nach einer vorgegebenen Zeit terminieren. Dadurch wird verhindert, dass eine nicht unterbrechbare Methode beliebig viel Rechenzeit verbrauchen und dass System beliebig lange blockieren kann.

Eine weitere Aufgabe des Verifiers besteht darin, dem Bytecode-nach-Maschinensprache Übersetzter Informationen zur Verfügung zu stellen, die eine Optimierung des Codes erlauben. Im Rahmen dieser Arbeit wurde beispielsweise eine Null-Pointer Analyse implementiert. Deren Aufgabe ist es festzustellen, ob eine Referenz an einer bestimmten Stelle im Bytecode einen gültigen Wert hat oder nicht. Kann verifiziert werden, dass die Referenz an dieser Stelle stets gültig ist, so kann die anderenfalls nötige Laufzeitüberprüfungen entfallen.

Außerdem soll eine möglichst einheitliche Vorgehensweise für die unterschiedlichen Analysen entwickelt werden, die es zudem erlaubt, weitere Analysen möglichst einfach zu implementieren.

Kapitel 2 gibt einen Überblick über die Java Virtual Machine (JVM). Das darauffolgende Kapitel beschäftigt sich mit den Grundlagen der automatischen Verifikation und Programmanalyse. Dabei wird auch auf die speziellen Anforderungen der JVM eingegangen.

In den restlichen Kapiteln werden die implementierten Analysen und Verifikationsschritte vorgestellt: Kapitel 4 beschreibt die Standard Java Bytecode Verifikation. Im darauffolgenden Kapitel wird die Null-Pointer Analyse erläutert und in Kapitel 6 wird auf die Bestimmung der Laufzeit einer Methode eingegangen. In Kapitel 7 werden schließlich verschiedene einfache Verifikationsschritte beschrieben, die weitere JX-spezifische Anforderungen verifizieren.

2 Überblick über die Java Virtual Machine

Zum besseren Verständnis der in den folgenden Abschnitten dargestellten Analysen soll zunächst ein Überblick über die Java Virtual Machine bzw. den Java Bytecode gegeben werden. Die Angaben in diesem Abschnitt sind im wesentlichen der JVM-Spezifikation ([LY97]) entnommen.

2.1 Aufbau der Klassendateien

Der vom Java-Compiler erzeugte Bytecode wird in sogenannten Klassendateien abgelegt, wobei für jede Klasse bzw. für jedes Interface genau eine Datei erzeugt wird. In dieser werden alle nötigen Informationen über die Klasse (z. B. Name, Name der Oberklasse und Zugriffsmodifikatoren wie `public` oder `private`) gespeichert, sowie alle Konstanten und der Bytecode der implementierten Methoden.

Die benötigten Konstanten (z. B. Zeichenfolgen) werden in einem sogenannten Konstantenpool („constant pool"), getrennt vom Bytecode der Methoden, gespeichert. Der Zugriff auf die Konstanten erfolgt mit Hilfe spezieller Befehle (siehe 2.3). Neben den im Programmtext vorkommenden Konstanten enthält der Konstantenpool auch einen Eintrag für jede Methode.

Dieser Eintrag beinhaltet Namen und Typdeskriptor[1] der Methode, Zugriffsmodifikatoren (z. B. `protected`, `private` usw.) sowie einen Zeiger auf den eigentlichen Bytecode der Methode, es sei denn es handelt sich um eine abstrakte Methode. Zusätzlich enthält jeder Eintrag eine Tabelle mit Ausnahmebehandlungsroutinen. Für jede Routine enthält diese Tabelle einen Eintrag, in dem der Typ der Ausnahme, die Startadresse der Ausnahmebehandlung so wie Start- und Endadresse des durch sie geschützten Bytecodes (entspricht dem „`try` Block" in Java) angegeben sind.

2.2 Struktureller Aufbau und Speichermodell

An den internen Aufbau und die Arbeitsweise der JVM werden in der JVM Spezifikation ([LY97]) keine direkten Anforderungen gestellt. Vielmehr muss eine Implementierung der JVM lediglich in der Lage sein, Klassendateien zu lesen und korrekt auszuführen (vorausgesetzt die Klassendatei selbst enthält keine Fehler). Die Details der Implementierung bleiben ausdrücklich dem Entwickler vorbehalten.

[1] Ein Typdeskriptor ist eine Zeichenkette, die die Datentypen der Argumente und Rückgaberwerte einer Methode bzw. den Datentyp von Feldern angibt. Der Aufbau kann in [LY97] nachgelesen werden.

Trotzdem skizziert die Spezifikation eine mögliche Implementierung, die als Referenz dienen kann, aber vor allem auch als Denkmodell sehr nützlich erscheint. Deshalb soll im Folgenden der Aufbau dieser Maschine kurz dargestellt werden.

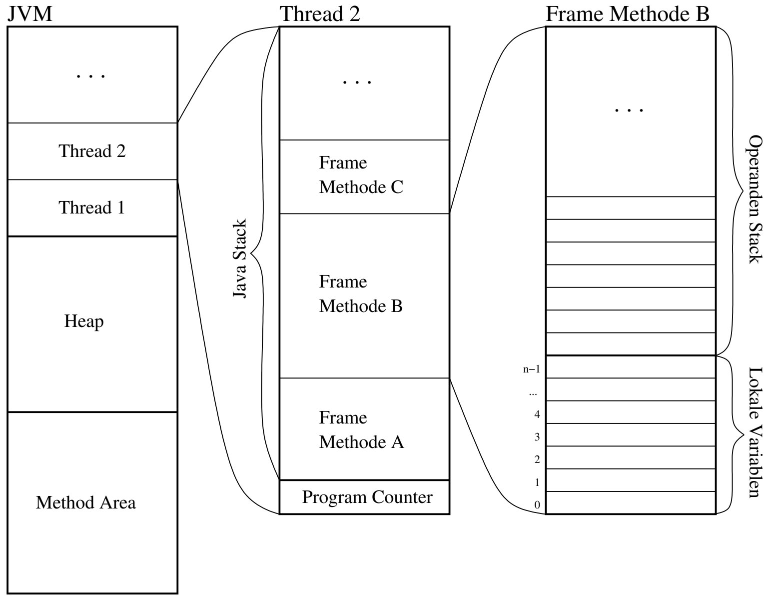


Abbildung 2.1: Speichermodell der JVM.

Das Speichermodell der Maschine ist in Abbildung 2.1 dargestellt. Der Arbeitsspeicher ist unterteilt in einen globalen Teil, der jedem Thread zugänglich ist, und einen Thread-lokalen Teil, der nur dem jeweiligen Thread vorbehalten ist. Der globale Teil enthält den Heap, auf dem der Speicher für alle Objektinstanzen und Arrays alloziert wird. Außerdem existiert eine globale „Method Area", die den Bytecode und die Konstantenpools aller geladenen Klassen enthält.

Der Thread-lokale Teil besteht jeweils aus einem Befehlszähler und einem „Java Stack". Dieser Java Stack wird nicht direkt durch den Bytecode verändert. Stattdessen wird für jeden Methodenaufruf ein Rahmen („Frame") auf dem Stack reserviert, der groß genug ist, die lokalen Variablen und den Operanden Stack (siehe unten) der Methode aufzunehmen. Die Größe des Java Stacks ist nicht standardisiert.

Zusätzlich kann noch ein „Native Method Stack" zur Verfügung gestellt werden, der zur Ausführung von „native" Methoden dient, also solchen Methoden, die nicht in Bytecodeform, sondern als Maschinencode vorliegen (in der Abbildung nicht dargestellt).

Alle Bytecode Befehle können nur auf Daten zugreifen, die entweder auf dem Heap liegen (also Teil von Objekten und Arrays sind) oder im Rahmen der gerade ausgeführten Methode liegen. Auf die Rahmen anderer Methoden kann nicht zugegriffen werden.

Jeder Methodenrahmen ist wiederum in zwei Teile untergliedert: den Operanden Stack und die lokalen Variablen. Letztere werden ähnlich angesprochen wie ein Array, jede Variable wird über einen Index adressiert. Die Werte in lokalen Variablen können nicht direkt als Operanden für Berechnungen verwendet werden, sondern müssen vorher auf den Operanden Stack geladen werden. Ebenso werden Ergebnisse eines Befehls stets auf dem Operanden Stack abgelegt. Für den Datentransfer zwischen lokalen Variablen und Operanden Stack existieren spezielle Lade- bzw. Speicheroperationen.

Die Zahl der lokalen Variablen kann für verschiedene Methoden unterschiedlich sein und wird für jede Methode vom Java-Compiler bestimmt. Gleiches gilt für die maximale Stackgröße. Demnach steht die Größe des Methodenrahmens bereits zu Compile-Zeit fest und ändert sich während der Ausführung der Methode nicht.

Alle Zugriffe auf Daten außerhalb des Methodenrahmens erfolgen über spezielle Befehle, die entweder Daten aus dem Konstantenpool laden oder auf Felder bzw. Methoden eines Objekts auf dem Heap zugreifen.

Beim Anlegen neuer Objekte wird zunächst geprüft, ob die benötigte Klasse bereits geladen wurde (der Name der benötigten Klasse ist als Eintrag im Konstantenpool der Klasse der aktuellen Methode gespeichert). Ist dies nicht der Fall wird sie geladen und die zugehörige Klassen-Initialisierungsfunktion („static initializer" in Java – siehe [GJS96]) aufgerufen. Anschließend kann der entsprechende Platz auf dem Heap reserviert werden und eine Referenz auf das neu erzeugte Objekt wird auf dem Operanden Stack der erzeugenden Methode hinterlegt. Der Aufruf des Konstruktors erfolgt nicht automatisch durch die JVM, vielmehr muss vor einem ersten Zugriff der Konstruktor für die entsprechende Klasse explizit, durch den Bytecode der erzeugenden Methode, aufgerufen werden.

Der Zugriff auf Felder und Methoden anderer Klassen erfolgt über den Namen der Klasse sowie Name und Typ der Methode, die beide als Einträge im Konstantenpool der aktuellen Klasse hinterlegt sind. Nachdem geprüft wurde, ob der Zugriff erlaubt ist, wird – bei Feldern – der entsprechende Wert zurückgeliefert.

Beim Aufruf einer Methode wird ein neuer Rahmen auf dem Java Stack reserviert. Die lokalen Variablen der neuen Methode werden mit den Parametern initialisiert, wobei in Variable 0 der Zeiger `this` auf die Instanz an der die Methode aufgerufen wurde,[2] und in den Variablen 1 bis n die n Parameter der Methode gespeichert werden. Schließlich wird der Bytecode der Methode ausgeführt. Nach Beendigung der aufgerufenen Methode wird deren Rahmen wieder vom Java Stack entfernt und das Resultat auf dem Operanden Stack des Aufrufers abgelegt – es sei denn die Methode wird durch eine Ausnahme beendet (dieser Fall wird in Abschnitt 2.4 beschrieben).

2.3 Überblick über den Befehlssatz

Mit wenigen Ausnahmen arbeiten alle Befehle der JVM Stack orientiert. Die Operanden müssen vor der Ausführung in der richtigen Reihenfolge auf den Stack gebracht werden und das Resultat wird ebenfalls auf dem Stack abgelegt. Auf die lokalen Variablen kann nur durch spezielle Befehle zugegriffen werden, die Daten vom Stack nehmen und in den Variablen speichern und umgekehrt. Auch Konstanten müssen zunächst auf dem Stack abgelegt werden, bevor sie verwendet werden können.

Jedes Datum auf dem Stapel oder in den lokalen Variablen hat einen Datentyp. Entsprechend haben auch die meisten Operationen einen Datentyp, auf dem sie arbeiten. Wird eine Operation auf einen falschen Datentyp angewandt, so wird der Fehler von der JVM entdeckt und

[2] vorausgesetzt es handelt sich um keine statische Methode

eine Ausnahme wird ausgelöst. Wird bereits bei der Verifikation festgestellt, dass ein falscher Datentyp verwendet wird, so wird ein entsprechender Fehler zurückgeliefert – siehe Kapitel 4.

Im Einzelnen existieren folgende ganzzahlige Datentypen: `byte` (8 Bit), `short` (16 Bit), `int` (32 Bit), `long` (64 Bit) und `char` (16 Bit). Mit Ausnahme des Typs `char` sind alle ganzzahligen Typen vorzeichenbehaftet. Zusätzlich existieren die beiden vorzeichenbehafteten Fließkommatypen `float` (32 Bit) und `double` (64 Bit), sowie die Typen `returnAddress` und `reference`, die für die Rücksprungadresse in einer Subroutine bzw. Zeiger auf Objekte und Arrays vorgesehen sind. Zu Daten vom Typ `reference` wird zusätzlich vermerkt, welcher Klasse das referenzierte Objekt angehört (in der JVM Spezifikation als „Laufzeitdatentyp" bezeichnet).

Auf dem Operanden Stack werden die Daten in Wörtern einheitlicher Größe gespeichert. Die Wortlänge ist nicht vorgegeben, es wird jedoch gefordert, dass alle Typen mit Ausnahme von `long` und `double` genau ein Wort, letztere zwei Wörter benötigen (wobei auch bei einer Wortgröße von 64 Bit zwei Wörter verwendet werden müssen).

Die Befehle bestehen jeweils aus einem Opcode von einem Byte Länge und einer – je nach Befehl variierenden – Anzahl Bytes zusätzlicher Parameter. Viele Befehle, wie beispielsweise Addition und Multiplikation, aber auch Load- und Store-Befehle, existieren in mehreren Versionen für unterschiedliche Typen. Der erste Buchstabe des Befehlskürzels gibt dann den Typ an (z. B. `i` für `int`, `l` für `long` und `a` für `reference`) .

Dabei handelt es sich bei den Typen `byte`, `short` und `char` lediglich um „Speichertypen". Daten dieser Typen dürfen lediglich in Objekten oder Arrays vorkommen. Beim Laden dieser Daten auf den Operanden Stack oder in die lokalen Variablen werden sie in den Typ `int` konvertiert. Umgekehrt werden beim Speichern in Felder oder Arrays dieser Typen wieder `int` Werte in den entsprechenden Datentyp konvertiert.[3] Innerhalb eines Methodenrahmens kommen diese Datentypen also nicht vor.

Für einige Befehle existieren mehrere Opcodes, die bereits einen Parameter des Befehls festlegen. So ist beispielsweise der `iload` Befehl, zum Laden eines `int` Wertes aus den lokalen Variablen auf den Stapel, fünf mal vorhanden: `iload`, `iload_0`, `iload_1`, `iload_2` und `iload_3`. Die erste Version bekommt den Index der lokalen Variable als Argument im Bytecode, die anderen dienen dazu, die Variable 0, 1, 2 bzw. 3 direkt zu laden und haben keine Parameter.

Die folgende Liste gibt einen Überblick über den Befehlssatz, wobei `T` für einen Typkürzel steht. In Tabelle 2.1 ist für jeden `T`-Befehl vermerkt, welche Datentypen im Einzelnen vorgesehen sind.

- Laden von Konstanten: `aconst_null` zur Erzeugung der konstanten `null` Referenz, `iconst_m1` und `iconst_[0-5]` zur Erzeugung ganzzahliger Konstanten im Bereich von -1 bis 5, sowie die Befehle `Tconst_[0-2]`.
  Mit `Tipush` werden `byte` und `short` Konstanten erzeugt, deren Wert als Parameter im Bytecode steht (wobei die abgelegten Werte wiederum vom Typ `int` sind).
  Mit `ldc` werden Konstanten aus dem Konstantepool geladen. Dabei kann es sich um Werte der Typen `int`, `long`, `float` oder `double`, sowie Zeichenketten handeln. Für letztere wird automatisch ein Objekt vom Typ `java.lang.String` erzeugt und eine Referenz auf dem Operanden Stack abgelegt.

[3] Der Versuch `long`, `float` oder `double` Werte in einem `byte`, `short` oder `char` Speicherplatz zu speichern wird als Typfehler behandelt. Werte vom Typ `int` werden hingegen automatisch konvertiert.

opcode	b	s	i	l	f	d	c	a	opcode	b	s	i	l	f	d	c	a
`Tipush`	x	x							`Tshr`			x	x				
`Tconst`			x	x	x	x		x	`Tushr`			x	x				
`Tload`			x	x	x	x		x	`Tand`			x	x				
`Tstore`			x	x	x	x		x	`Tor`			x	x				
`Tinc`			x						`Txor`			x	x				
`Taload`	x	x	x	x	x	x	x	x	`i2T`	x	x		x	x	x		
`Tastore`	x	x	x	x	x	x	x	x	`l2T`			x		x	x		
`Tadd`			x	x	x	x			`f2T`			x	x		x		
`Tsub`			x	x	x	x			`d2T`			x	x	x			
`Tmul`			x	x	x	x			`Tcmp`				x				
`Tdiv`			x	x	x	x			`Tcmpl`					x	x		
`Trem`			x	x	x	x			`Tcmpg`					x	x		
`Tneg`			x	x	x	x			`if_TcmpOP`				x				x
`Tshl`			x	x					`Tretun`			x	x	x	x		x

Tabelle 2.1: Überblick über JVM Befehlssatz.

- Zugriff auf lokale Variablen: Die Befehle `Tload` lesen einen Wert aus den lokalen Variablen und legen ihn auf dem Stack ab.
  Die Befehle `Tstore` speichern entsprechend das oberste Stack-Element in den lokalen Variablen.
  Der Index wird durch das nachfolgende Byte im Bytecode bestimmt. Außerdem existieren Opcodes um einige Variablen direkt zu adressieren.

- Arithmetische und logische Operationen: `Tadd`, `Tsub`, `Tmul`, `Tdiv`, `Trem`, `Tneg`, `Tshl`, `Tshr`, `Tushr` („unsigned shift right"), `Tor`, `Tand`, `Txor` und `iinc`.
  Die Operanden der Befehle liegen auf dem Operanden Stack und werden bei der Ausführung entfernt. Anschließend wird das Ergebnis abgelegt.
  Einzige Ausnahme dieser Regel bildet `iinc`: Er dient dazu, eine lokale Variable um einen bestimmten Betrag zu erhöhen. Inkrement und Index der Variable bestehen aus den beiden dem Opcode folgenden Bytes.

- Typumwandlungen: Die Befehle `i2T`, `l2T`, `f2T` und `d2T` dienen dazu, Werte in andere Datentypen zu konvertieren. Die Einzelheiten können in [LY97] nachgelesen werden. An dieser Stelle soll lediglich darauf hingewiesen werden, dass die Befehle `i2b` und `i2s` zwar einen `int` Wert in einen `byte` oder `short` Wert umwandeln, diesen dann jedoch wieder als `int` Wert (dessen höherwertigen 16 oder 24 Bit 0 sind) auf dem Stapel ablegen.

- Objekterzeugung und Verwaltung: `new` dient der Erzeugung neuer Objekte, `newarray`, `anewarray` und `multianewarray` erzeugen neue Arrays.
  Der Zugriff auf Felder und statische Klassenvariablen wird mit `getfield` und `getstatic` (lesen) sowie `putfield` und `pustatic` (schreiben) durchgeführt.
  Um Elemente eines Arrays zu speichern oder zu laden stehen die Befehle `Tastore` und `Taload` zur Verfügung. Außerdem kann mit dem Befehl `arraylength` die Länge eines Arrays bestimmt werden.
  Der Befehl `instanceof` entspricht dem gleichnamigen Befehl auf Java Ebene und

checkcast dient dazu Typ-Konvertierungen auf Klassenebene zu überprüfen.

- Operanden Stack Verwaltung: Zur Verwaltung des Operanden Stacks dienen die Befehle pop, dup (oberstes Wort duplizieren) und swap (Reihenfolge zweier Wörter vertauschen).

- Kontrollflusstransfer: Die ifOP und if_TcmpOp Befehle dienen als bedingte Verzweigungen. Dabei steht OP für eine der Bedingungen eq (gleich), lt (kleiner als), le (kleiner gleich), ne (ungleich), gt (größer als) oder ge (größer gleich). Vor Anwendung der ifOP Befehle muss zunächst ein Vergleich der obersten beiden Stackwörter mittels Tcmp Befehl durchgeführt werden. Das Ergebnis der Tcmp Befehle ist ein int Wert, der von den ifOp Befehlen ausgewertet wird.
  Im Unterschied dazu arbeiten die if_TcmpOP Befehle direkt auf den zu vergleichenden Daten, die Tcmp Operation muss nicht explizit ausgeführt werden.
  Zusätzlich existieren die Befehle ifnull und ifnonnull, die Verzweigen wenn eine Referenz null bzw. nicht null ist.
  Außerdem stehen die Befehle tableswitch und lookupswitch zur Verfügung, die über einen int Wert auf dem Stack eines von mehreren Sprungzielen auswählen.
  Des Weiteren führen goto, jsr und ret unbedingten Sprünge aus. Der jsr („jump to subroutine”) Befehl dient, dazu die Ausführung mit einer Subroutine innerhalb des Bytecodes der aktuellen Methode fortzusetzten. Die Rücksprungadresse wird auf dem Stapel abgelegt. Nach Ende der Subroutine wird die Ausführung durch eine ret Anweisung wieder and dieser Adresse fortgesetzt.

- Methodenaufrufe: Es existieren vier Befehle zum Aufruf von Methoden. Bei allen Befehlen steht nach dem Opcode ein Zeiger in den Konstantenpool, der auf den Methodeneintrag der auszuführenden Methode verweist.
  Um Methoden über ein Interface aufzurufen dient der Befehl invokeinterface, für statische Methoden der Befehl invokestatic. Mit invokespecial werden spezielle Methoden aufgerufen. Dabei handelt es sich entweder um einen Konstruktor oder um private Methoden einer Klasse. Auch der explizite Aufruf einer Methode der Superklasse (durch super.method() in Java) wird durch einen invokespecial Aufruf durchgeführt.
  Alle anderen Methoden werden mit invokevirtual aufgerufen.

- Auslösen von Ausnahmen: Der athrow Befehl dient dazu, eine Ausnahme auszulösen. Dabei muss ein initialisiertes Objekt vom Typ java.lang.Throwable bzw. einer Unterklasse auf dem Operanden Stack liegen.

2.4 Ausnahmebehandlung und Subroutinen

Alle Ausnahmebehandlungsroutinen einer Methode stehen im Bytecode der Methode und sind zunächst nicht weiter gekennzeichnet. Allerdings darf der zur Ausnahmebehandlung gehörende Code nicht durch Verzweigungs- oder Sprungbefehle adressiert werden. Stattdessen existiert in der Klassendatei eine Tabelle, die angibt, an welcher Adresse im Bytecode die Ausnahmebehandlungsroutine zu welchem Ausnahmetyp steht.

Wird während der Ausführung einer Methode eine Ausnahme ausgelöst (z. B. durch Aufruf des athrow Befehls), so wird nach einer passenden Behandlungsroutine in der aktuellen Methode gesucht. Eine Behandlungsroutine passt zur Ausnahme, wenn sie entweder Ausnahmen des ausgelösten Typs oder einer seiner Oberklassen behandelt. Wird eine entsprechende Routine gefunden, so wird der Operanden Stack geleert, das Ausnahme Objekt wird auf dem Stack abgelegt und die Ausführung wird an der Anfangsadresse der Behandlungsroutine fortgesetzt. Wird keine passende Behandlung gefunden, so wird die Methode beendet, und die Ausnahme wird in der aufrufenden Methode erneut ausgelöst.

Zusätzlich zur Ausnahmebehandlung existiert in Java die Möglichkeit, mit Hilfe eines „finally" Blocks, Code anzugeben, der immer ausgeführt wird, egal ob eine Ausnahme ausgelöst wurde oder nicht (siehe [GJS96]). Diese Funktionalität wird im Bytecode mit Hilfe der Befehle `jsr` und `ret` umgesetzt, die dazu dienen ein Unterprogramm (Subroutine) auszuführen, bzw. aus einem solchen zurückzukehren.

Wird nun ein „finally" Block angegeben, so wird dieser in ein Unterprogramm übersetzt. Jede zuständige Ausnahmebehandlung enthält einen Aufruf des Unterprogramms und ein weiterer Aufruf wird am Ende des geschützten Codes („try" Block) eingefügt. Wird vom Programmierer keine Ausnahmebehandlung vorgesehen, so wird eine generische Ausnahmebehandlungsroutine installiert, die Ausnahmen vom Typ `java.lang.Throwable` behandelt. Diese ruft das Unterprogramm auf und löst danach die Ausnahme erneut aus.

Eine Subroutine ist jedoch nicht mit einem Methodenaufruf zu verwechseln: Sie unterscheided sich von einem Methodenaufruf dadurch, dass die Ausführung innerhalb des aktuellen Methodenrahmens stattfindet. Insbesondere bleiben Operanden Stack und lokale Variablen bei der Ausführung einer `jsr` und `ret` Anweisung erhalten.

3 Allgemeines Vorgehen bei der Programmanalyse

Allen durch den Verifier durchgeführten Analysen liegt – trotz zum Teil sehr unterschiedlicher Zielsetzung – eine gemeinsame Vorgehensweise zu Grunde. Diese soll in den nächsten Abschnitten eingehend vorgestellt werden.

Ziel der Programmanalyse ist es, Aussagen über Eigenschaften eines Programmes oder dessen Daten an einer bestimmten Stelle zu machen bzw. vorgegebene Aussagen zu verifizieren, ohne das Programm auszuführen. Zu diesem Zweck muss zumeist bestimmt werden, welche Werte eine Variable (bzw. ein Stapelwort oder eine lokale Variable) an bestimmten Stellen im Programm annehmen kann, bzw. ob diese Werte gewissen Bedingungen genügen. Beispielsweise muss, um zu überprüfen ob der Bytecode einer Methode Datentypen korrekt behandelt, für jeden Befehl geprüft werden, ob die Operanden Werte des entsprechenden Typs enthalten.

Ein Hauptproblem dabei ist, dass die Ausführung eines Programmes bzw. einer Methode von vielen Faktoren abhängt, die zum Zeitpunkt der Analyse noch nicht bekannt sind. Zum einen wird das Verhalten einer Methode in der Regel durch Parameter bestimmt, die beim Aufrufen übergeben werden und nur selten vorherbestimmt werden können. Zum anderen hängen die meisten größeren Programme aber auch von Einflüssen ab, die außerhalb des durch die Programmanalyse betrachteten Bereiches liegen, wie z. B. Zustände bestimmter Systemresourcen oder gar Eingaben durch den Benutzer.

Dieses Problem kann auf zwei Wegen umgangen werden: Zum einen kann man bei der Analyse Werte für die verschiedenen Parameter festlegen bzw. die Menge der möglichen Werte eingrenzen. Dieser Ansatz hat den Nachteil, dass die Wertemengen für die Parameter nur sehr wenige Elemente enthalten dürfen. Schließlich muss die Methode für jede mögliche Kombination von Parametern analysiert werden. Zudem ist es kaum möglich *maschinell* eine sinnvolle Menge von Werten für die Parameter zu bestimmen. Deshalb ist man häufig auf zusätzliche Informationen angewiesen, die vom Programmierer bereitgestellt werden müssen. Dennoch ist ein solches Vorgehen manchmal unumgänglich, wie beispielsweise die Überlegungen in Abschnitt 6 zeigen. Da die Umsetzung einer solchen Vorgehensweise jedoch in hohem Maße von den Zielen der Analyse abhängt, soll in diesem Kapitel nicht weiter darauf eingegangen werden.

Die zweite Möglichkeit besteht darin, nur solche Aussagen über ein Programm zu treffen, die für jeden möglichen Ablauf des Programmes gelten. Dieser Ansatz bringt jedoch das Problem mit sich, dass die Zahl der möglichen Programmausführungen sehr groß werden kann, bei Schleifenkonstrukten sogar unendlich. Dementsprechend muss ein Weg gefunden werden, Aussagen über alle Programmläufe machen zu können, ohne tatsächlich jeden einzelnen betrachten zu müssen.

Eine in optimierenden Compilern häufig eingesetzte Technik um Programme ohne Annahme von Parameterwerten zu analysieren ist die Datenflussanalyse (siehe [ASU88, NNH99]), die auch in dem in dieser Arbeit erstellten Verifier zum Einsatz kommt.

In Abschnitt 3.1 wird diese ausführlich beschrieben, wobei die Ausführungen dieses Abschnittes weitgehend auf dem zweiten Kapitel von [NNH99] basieren.

In Abschnitt 3.2 wird die allgemeine Datenflussanalyse an die speziellen Bedürfnisse der Bytecode-Analyse angepasst und entsprechend erweitert.

3.1 Datenflussanalyse

Wie bereits erwähnt, soll in diesem Abschnitt eine wichtige Technik der Programmanalyse beschrieben werden: Die Datenflussanalyse (DFA).

Dabei geht es darum, Aussagen über eine Methode bzw. deren Datenstrukturen zu machen, ohne sie auszuführen. Die Argumente der Methode sind jedoch zum Zeitpunkt der Analyse unbekannt, so dass nur solche Eigenschaften analysiert werden können, die für alle möglichen Ausführungen der Methode gelten. Alternativ kann man auch nach Eigenschaften suchen, die in mindestens einem möglichen Programmablauf zutreffen. Beide Alternativen sind insofern sehr ähnlich, als eine bestimmte Aussage genau dann für *alle* Programmläufe zutrifft, wenn es nicht *mindestens einen* Lauf gibt, für den die Aussage nicht zutrifft.

Um die weiteren Erläuterungen zu vereinfachen zunächst zwei Definitionen:

- Ein *Basic Block* ist ein Codestück, dass keinerlei Sprungbefehle enthält und dessen Befehle nicht Ziel anderer Sprungbefehle sind. Das heißt, dass die in einem Basic Block enthaltenen Befehle stets sequentiell ausgeführt werden, beginnend bei dem ersten Befehl und endend mit dem letzten Befehl des Basic Blocks.

- Der *Kontrollflussgraph (KFG)* eines Programmes P ist ein gerichteter Graph $KFG = (E, K)$ mit Knotenmenge E und Kantenmenge K. Jedem Knoten aus E ist ein Basic Block von P zugeordnet und umgekehrt. Zwei Knoten b_1 und b_2 sind durch eine Kante verbunden, wenn die Programmausführung nach Abarbeitung des letzten Befehls aus b_1 mit dem ersten Befehl aus b_2 fortgesetzt werden kann.
  Zusätzlich wird ein eindeutiger Startknoten S, sowie ein eindeutiger Endknoten R eingeführt. Dabei wird in K eine Kante von S zum ersten Basic Block des Programmes eingefügt und jeweils eine Kante von Basic Blocks, deren letzter Befehl die Methode beendet, zum Knoten R.

In Abbildung 3.1 ist ein kurzes Java Beispielprogramm mit dessen KFG aufgeführt. Da in diesem Beispielprogramm jeder Basic Block nur aus einer Zeile besteht, wurde im KFG jeder Knoten mit der entsprechenden Zeilennummer beschriftet.

Ein möglicher Programmablauf ist nun ein Pfad von S nach R. Allerdings ist nicht jeder Pfad auch ein möglicher Programmablauf. In dem Beispiel aus Abb. 3.1 etwa kann man den Pfad $(S, 2, 3, 5, 6, 5, 8, R)$ angeben, der keinen gültigen Programmablauf darstellt: Wird nämlich die Anweisung in Zeile 3 ausgeführt, so bedeutet das, dass die Variable a kleiner null ist. Somit

```java
1 public int foo(int a) {
2   if (a < 0) {
3     a=-1;
4   }
5   while(a>=0) {
6     a--;
7   }
8   return a;
9 }
```

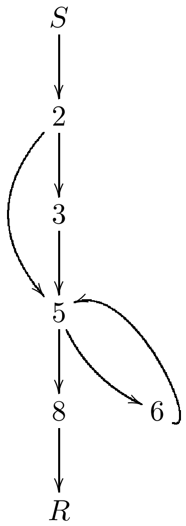


Abbildung 3.1: Java Beispielprogramm und dessen Kontrollflussgraph.

wird jedoch der Rumpf der `while`-Schleife, also Knoten 6, nicht ausgeführt. Tatsächlich sieht man leicht, dass ein Pfad der eine mögliche Programmausführung darstellt niemals die Knoten 3 und 6 enthalten kann.

In diesem Beispiel ist leicht erkennbar, welche Pfade im KFG eine mögliche Programmausführung darstellen. Im Allgemeinen ist dies jedoch – vor allem automatisch – nur schwer feststellbar und in einigen Fällen sogar unentscheidbar (vgl. [NNH99]). Deshalb betrachtet man bei der DFA in der Regel nicht nur die Pfade, die möglichen Programmausführungen entsprechenden, sondern alle Pfade von S nach R.

Dadurch wird natürlich eine gewisse „Unschärfe" herbeigeführt, insofern, als manche Eigenschaften wohl für alle möglichen Ausführungen, nicht aber für alle Pfade gelten und somit nicht erkannt werden können. Da sich dieses Problem – wenn überhaupt – nur mit großem Aufwand umgehen ließe, muss diese Ungenauigkeit in der Analyse jedoch in Kauf genommen werden.

Dass es also zumeist nicht möglich ist, eine gesuchte Eigenschaft immer, für jedes Datum und an jeder Stelle der Methode zu erkennen, hat auch Auswirkungen auf Korrektheitsbedingungen einer Datenflussanalyse: Man kann zumeist nicht fordern, dass eine Analyse die gesuchte Eigenschaft erkennt, wenn sie vorliegt. Stattdessen muss man sich auf die Forderung beschränken, dass eine Eigenschaft tatsächlich zutrifft, wenn sie erkannt wird. Eine nach diesem Prinzip zu einem Analyseproblem gefundene Lösung wird auch als „sichere Lösung" bezeichnet.

Ein weiteres Problem bei der Analyse einer Methode, die Schleifen enthält, ist, dass die Zahl der möglichen Pfade von S nach R nicht endlich ist, da jede Schleife einem Zyklus im KFG entspricht, der beliebig oft durchlaufen werden kann. Soll trotzdem ein Ergebnis erreicht werden, so muss die Analyse so gestaltet sein, dass das Ergebnis weitgehend unabhängig von der Zahl der Schleifendurchläufe ist. Weitere Einzelheiten finden sich im nächsten Abschnitt.

3.1.1 Vorgehensweise

Bevor auf die Details der Vorgehensweise eingegangen wird, sollen zunächst die Grundzüge der Datenflussanalyse erörtert werden.

Das Prinzip besteht darin, für jeden Befehl im Programm eine „Transferfunktion" anzugeben. Diese gibt an, wie der Zustand der ausführenden Maschine bei Ausführung des Befehls in einen neuen Zustand transformiert wird. Von diesen Zuständen wiederum lassen sich dann die gesuchten Ergebnisse ableiten.

Dadurch entsteht eine Art Gleichungssystem, bei dem das Ergebnis einer Transferfunktion Argument der Funktion des nachfolgenden Befehls ist. Hat ein Befehl mehrere Vorgänger, so müssen deren Ergebniszustände zunächst vereinigt werden. Wie genau eine solche Vereinigung aussieht ist stark problemabhängig. Einige an sie gestellte Anforderungen werden jedoch im nächsten Abschnitt vorgestellt.

Im Folgenden ist ein kurzes Beispielprogramm zur Berechnung der Fakultätsfunktion angegeben. Daneben steht das zugehörige Gleichungssystem. Bei `i` und `j` handelt es sich um ganzzahlige Programmvariablen, f_n ist die Transferfunktion des Befehls in Zeile n und die z_n sind Zustandsvariablen. Dabei steht z_n für den Zustand direkt vor Ausführung der Anweisung in Zeile n und z_R für den Endzustand. Die Vereinigung zweier Zustände wird mit $\sqcup$ bezeichnet und z_0 bezeichnet einen vorgegebenen Anfangszustand.

```
1 if(i<=0){
2   i=-i;
3 }
4 j=1;
5 while (i>0) {
6   j=j*i;
7 }
```

$$\begin{aligned} z_1 &= z_0 \\ z_2 &= f_1(z_1) \\ z_4 &= f_2(z_2) \sqcup f_1(z_1) \\ z_5 &= f_4(z_4) \sqcup f_6(z_6) \\ z_6 &= f_5(z_5) \\ z_R &= f_4(z_4) \sqcup f_6(z_6) \end{aligned}$$

Wie man sieht wird jeder Zustand z_n des Befehls aus der Vereinigung der Ergebniszustände seiner möglichen Vorgänger berechnet. Diese Ergebniszustände ergeben sich wiederum aus der Anwendung der Transferfunktion des Vorgängers auf dessen Zustand. In diesem Beispiel werden auch zwei unterschiedliche Anwendungsfälle der Vereinigungsoperation $\sqcup$ deutlich: Für den Zustand z_4 werden die Ergebniszustände der Befehle aus Zeile 2 und 1 vereinigt, da beides mögliche Vorgänger sind, abhängig davon ob `i` negativ ist oder nicht. Allerdings ist bei Ausführung des Programmes immer nur eine der beiden Anweisung der direkte Vorgänger der Anweisung in Zeile 4. Anders bei z_5: Hier kann die `while` Anweisung innerhalb einer Programmausführung mehrfach, mit unterschiedlichen Vorgängern, durchlaufen werden. In beiden Fällen wird durch die Vereinigungsoperation ein einheitlicher Zustand erzeugt.

Die Lösung dieses Gleichungssystems erfolgt durch ein iteratives Lösungsverfahren. Dabei wird die Transferfunktion jedesmal erneut ausgehend vom vorgegebenen Ausgangszustand z_0. Daraus wird eine Lösung für die erste Gleichung berechnet, das Ergebnis wird wiederum zur Berechnung der zweiten Gleichung verwendet usw. Dabei muss der Ergebniszustand einer Anweisung immer dann neu berechnet werden, wenn sich einer der Vorgängerzustande geändert hat. Das ganze muss so lange fortgesetzt werden, bis sich keiner der Zustände mehr ändert, d. h. bis eine Art „Fixpunkt" erreicht wird.

Während das bei linearen Programmen schon nach einem „Durchlauf" der Fall ist, bestehen bei Verzweigungen und Schleifenkonstrukten jedoch rekursive Abhängigkeiten zwischen den Zuständen. Zum Beispiel hängt in obigem System z_5 von z_6 ab und umgekehrt. Dementsprechend besteht die Gefahr, dass ein solcher Fixpunkt nicht erreicht wird, d. h. dass das Lösungsverfahren nicht terminiert. Deshalb muss beim Festlegen der Zustände darauf geachtet werden, dass gewisse Bedingungen erfüllt sind, die garantieren, dass ein Fixpunkt erreicht wird. Diese werden in Abschnitt 3.1.4 vorgestellt.

Im Folgenden soll nun die skizzierte Vorgehensweise etwas genauer betrachtet werden. Die Betrachtungen beschränken sich hierbei auf die Analyse von Java Bytecode, die Analyse von Hochsprachen wird jedoch ganz ähnlich gehandhabt.

Zunächst werden die verschiedenen Komponenten der Datenflussanalyse näher erläutert, bevor das Lösungsverfahren eingehend vorgestellt wird. Abschließend werden die Bedingungen genannt, die Zustände und Transferfunktionen erfüllen müssen, damit ein Fixpunkt erreicht wird.

3.1.2 Komponenten der Datenflussanalyse

Zunächst einmal muss bestimmt werden, wie die Zustände und Transferfunktionen aussehen. Das heißt die beiden folgenden Komponenten müssen festgelegt werden:

- *Zustandsraum* L: Der Zustandsraum ist die Menge aller möglichen Zustände, welche die ausführende Maschine (im Rahmen dieser Arbeit also die JVM) annehmen kann. Zusätzlich muss ein Zustand $\perp \in L$ existieren, der angibt, dass der erzeugende Befehl noch nicht analysiert wurde. Ferner muss die Vereinigung eines Zustandes z_a mit Zustand $\perp$ wieder den Zustand z_a ergeben, also $z_a \sqcup \perp = z_a$.

  Wie genau diese Zustände aussehen ist problemabhängig. Im Allgemeinen geht es jedoch nicht darum, ein möglichst genaues Abbild des Zustands der Maschine an einer bestimmten Stelle zu bestimmen. Vielmehr geht es zumeist lediglich um bestimmte Eigenschaften die analysiert werden sollen. In der Regel genügt es deshalb, die gesuchten Eigenschaften zu betrachten, z. B. könnte der „Zustand" nur aus einer Menge von Variablen bestehen, die gewisse Attribute erfüllen (wie „nichtnull" oder „null"). Deshalb wird L auch häufig als *Eigenschaftsraum* bezeichnet.

  Auf Grund der Nähe des Java Bytecodes zur ausführenden (virtuellen) Maschine muss bei dessen Analyse jedoch meist tatsächlich ein Abbild der ausführenden Maschine simuliert werden (vor allem der Operanden Stack und die lokalen Variablen).

  Außerdem muss es möglich sein, einen Anfangszustand anzugeben, der von den Parametern der Methode unabhängig ist (da diese ja zur Verifikationszeit nicht bekannt sind).

- *Transferfunktionen* $\mathcal{F}$: Eine Menge von Funktionen $f : L \to L$ auf dem Zustandsraum. Diese modellieren, wie oben beschrieben, den Effekt bestimmter Befehle auf einen Zustand.

Der Zustandsraum und die Menge der Transferfunktionen hängen zum einen vom Zweck der Analyse, zum anderen von der zu analysierenden Sprache ab. Bei der Analyse einer konkreten

Methode müssen jedoch noch die folgenden Komponenten bestimmt werden (dabei wird folgende Notation benutzt: A steht für die Menge der gültigen Adressen, d. h. der Adressen bei denen ein neuer Befehl beginnt, a steht für eine Adresse aus A):

- *Zuordnungsfunktion* $f_.$: Die Zuordnungsfunktion $f_. : A \rightarrow \mathcal{F}$ wird benötigt, um der Anweisung an einer gegebenen Adresse $a \in A$ eine Transferfunktion $f_a \in \mathcal{F}$ zuzuordnen. In der Regel wird in $\mathcal{F}$ eine Funktion für jede Bytecode Operation vorhanden sein. Die Zuordnung einer Adresse zu einer Funktion hängt dann von dem an dieser Adresse stehenden Befehl ab. Allerdings kann prinzipiell eine beliebige Zuordnung gewählt werden.

- *Flussrelation* F: Die Relation $F \subseteq A \times A$ weist jeder Anweisung eine Menge von Nachfolger- bzw. Vorgängeranweisungen zu. Soll die Transformationsfunktion für den Befehl an Adresse a ausgeführt werden, so wird der Vorgängerzustand aus den Ergebniszuständen der Befehle aus der Menge $\{n|(n, a) \in F\}$ berechnet. Analog dazu müssen bei verändertem Ergebniszustand von a alle Befehle aus $\{m|(a, m) \in F\}$ neu berechnet werden.

  Die Flussrelation ähnelt in gewisser Weise der Kantenmenge des KFG, wobei die Knoten nicht mehr Blöcken sondern einzelnen Befehlen zugeordnet sind. Allerdings ist es durchaus denkbar, dass nicht immer alle Kanten aus dem KFG verfolgt werden müssen, beispielsweise auf Grund schon berechneter Analyseergebnisse. Außerdem besteht die Möglichkeit, statt wie oben beschrieben beim Anfang der Methode zu beginnen, eine Methode von ihrem Ende aus zu analysieren, indem die Richtung aller Kanten in F umgekehrt wird. Man spricht in diesem Fall von einer Rückwärtsanalyse (im Gegensatz zur Vorwärtsanalyse).

- *Anfangsadressen* E: Eine Menge von Adressen, bei denen die Analyse beginnen soll. Bei einer Vorwärtsanalyse ist in der Regel $E = \{0\}$, bei einer Rückwärtsanalyse hingegen kann es mehrere Befehle geben, welche die Methode beenden, dementsprechend auch mehrere Endadressen.

- *Anfangszustand* ι: Der Zustand mit dem die Analyse an den durch E bezeichneten Adressen beginnt.

3.1.3 DFA-Algorithmus

Der Analysealgorithmus ist in Abbildung 3.2 schematisch dargestellt, wobei W die Menge der noch zu bearbeitenden Befehle darstellt. Die Funktion $takeone(W)$ entnimmt der Menge W ein Element und liefert es zurück. Der Ausdruck aF bezeichnet die Menge der Nachfolger von a, also $aF = \{n \in A|(a, n) \in F\}$, analog bezeichnet $Fa = \{m \in A|(m, a) \in F\}$ die Menge der Vorgänger von a.

Der Zustand bzw. die geltenden Eigenschaften direkt vor Ausführung des Befehls an Adresse a werden in $Analysis[a]$ vermerkt, die Adressen aller Befehle, die noch betrachtet werden müssen werden in der „Worklist" W gespeichert. Zunächst wird diese initialisiert, indem alle Befehle des Programms, bzw. alle Befehle die laut F und E betrachtet werden müssen, eingetragen werden.

Eingabe: $(L, \mathcal{F}, F, E, \iota, f_\cdot)$

1: {Initialisierung}
2: $W = \phi$ {Worklist}
3: **for all** $(a, a') \in F$ **do**
4: $\quad W = W \cup \{a, a'\}$
5: **end for**
6: **for all** $a \in W \cup E$ **do**
7: $\quad$ **if** $a \in E$ **then**
8: $\quad\quad Analysis[a] = \iota$
9: $\quad$ **else**
10: $\quad\quad Analysis[a] = \bot$
11: $\quad$ **end if**
12: **end for**
13: {Iteration; eigentliche Analyse}
14: **while** $W \neq \phi$ **do**
15: $\quad a = takeone(W)$
16: $\quad$ **for all** $b \in aF$ **do**
17: $\quad\quad$ **if** $(Analysis[b] \sqcup f_a(Analysis[a])) \neq Analysis[b]$ **then**
18: $\quad\quad\quad Analysis[b] = Analysis[b] \sqcup f_a(Analysis[a])$
19: $\quad\quad\quad W = W \cup \{b\}$
20: $\quad\quad$ **end if**
21: $\quad$ **end for**
22: **end while**

Abbildung 3.2: Datenflussanalyse (nach [NNH99])

Die eigentliche Analyse beginnt in Zeile 14: So lange W nicht leer ist wird ein Element entnommen und aus dem Vorgängerzustand wird der Ergebniszustand berechnet. Für jeden Nachfolgebefehl wird der Ergebniszustand mit dessen Vorgängerzustand vereinigt. Im Falle einer Änderung wird der Nachfolger wiederum in die Worklist eingetragen.

Wie bereits erwähnt muss dieser Algorithmus, bei ungünstiger Wahl des Zustandsraumes, nicht terminieren. Deshalb werden im nächsten Abschnitt Bedingungen vorgestellt, die hinreichend für eine Terminierung sind.

3.1.4 Terminierung und Anwendbarkeit

Das Problem des oben vorgestellten Algorithmus ist, dass er nicht immer terminiert, falls der KFG (oder vielmehr der Graph (A, F)) Zyklen enthält. Dieses Problem tritt dann auf, wenn innerhalb einer Schleife jede Zustandsänderung die Änderung des Zustands eines nachfolgenden Befehls zur Folge hat. Dadurch ändert sich auch dieser Zustand erneut, so dass der Kreislauf von vorne beginnt. Zudem ist nicht garantiert, dass der Algorithmus, falls er terminiert, tatsächlich eine Lösung des zugehörigen Gleichungssystems berechnet.

Bei geschickter Wahl des Zustands- bzw. Eigenschaftsraums und vor allem der Vereinigungsoperation kann sichergestellt werden, dass der Algorithmus terminiert. In [NNH99] wird gezeigt, dass das dann der Fall ist, wenn der Eigenschaftsraum aus einem vollständigen Gitter

(„complete Lattice") besteht. Außerdem wird gezeigt, dass das Verfahren auch eine Lösung zu dem zugehörigen Gleichungssystem liefert, wenn die Menge der Transferfunktionen $\mathcal{F}$ bestimmte Eigenschaften erfüllen. Die Bedingungen werden nun im Einzelnen vorgestellt.

3.1.4.1 Terminierungsbedingungen

Zunächst soll das Konzept des vollständigen Gitters näher erläutert werden. Dazu einige Definitionen:

- Eine *Halbordnung* über L ist eine Relation $\sqsubseteq: L \times L$, die
  - reflexiv ($\forall l \in L : l \sqsubseteq l$),
  - transitiv ($\forall l_1, l_2, l_3 \in L : l_1 \sqsubseteq l_2 \wedge l_2 \sqsubseteq l_3 \Rightarrow l_1 \sqsubseteq l_3$) und
  - antisymmetrisch ($\forall l_1, l_2 \in L : l_1 \sqsubseteq l_2 \wedge l_2 \sqsubseteq l_1 \Rightarrow l_1 = l_2$) ist.

  Eine *halbgeordnete Menge* $(L, \sqsubseteq)$ ist eine Menge L mit einer über L definierten Halbordnung $\sqsubseteq$.

- Eine Untermenge $Y \subseteq L$ hat eine *obere Schranke* $l \in L$, falls für alle $l' \in Y$ gilt $l' \sqsubseteq l$. Falls zusätzlich für alle oberen Schranken $m \in L$ von Y gilt $l \sqsubseteq m$, so ist l die *kleinste obere Schranke* von Y.
  Die kleinste obere Schranke wird auch mit $l = \sqcup Y$ bezeichnet. Des Weiteren wird eine binäre Operation $a \sqcap b$ eingeführt, die für $\sqcap\{a, b\}$ steht.
  Analog kann auch die *größte untere Schranke* $\sqcap Y$ einer Menge Y definiert werden.

- Ein *vollständiges Gitter* $(L, \sqsubseteq, \sqcup, \sqcap, \perp, \top)$ besteht aus einer halbgeordneten Menge $(L, \sqsubseteq)$, wobei jede Teilmenge von L eine kleinste obere und größte untere Schranke besitzt. Außerdem bezeichnen $\top = \sqcup L = \sqcap\phi$ das *größte Element* und $\perp = \sqcup\phi = \sqcap L$ das *kleinste Element* von L.

  Man beachte, dass einem vollständigen Gitter keine vollständige Ordnung zu Grunde liegen muss. Es kann also durchaus sein, dass zwei Elemente a und b aus L nicht vergleichbar sind (d. h. es gilt weder $a \sqsubseteq b$ noch $b \sqsubseteq a$). Es muss lediglich eine gemeinsame kleinste obere und untere Schranke existieren.
  Da man für die Zwecke der Datenflussanalyse lediglich die $\sqcup$ Operation benötigt, wird im Folgenden ein vollständiges Gitter häufig nur als $(L, \sqsubseteq, \sqcup, \perp, \top)$ angegeben (wobei die $\sqcap$ Operation nicht explizit genannt wird).

Ein Beispiel für ein vollständiges Gitter ist durch die Potenzmenge einer beliebigen endlichen Menge und die Relation $\subseteq$ gegeben. In Abbildung 3.3 ist das Gitter $(\mathcal{P}(\{1, 2, 3\}), \subseteq, \cup, \cap, \phi, \{1, 2, 3\})$ graphisch dargestellt, wobei $a \subseteq b$ durch einen Pfeil von a nach b dargestellt wird. In der Abbildung sind nur die Beziehungen zwischen benachbarten Knoten dargestellt, es fehlen alle durch Reflexivität und Transitivität entstehenden Verknüpfungen.

- Eine *Kette* eines vollständigen Gitters L ist eine Untermenge $K \subseteq L$, die vollständig geordnet ist. Eine *aufsteigende* Kette ist eine Folge von Elementen l_j, die so angeordnet sind, dass gilt: $m < n \Rightarrow l_m \sqsubseteq l_n$. Im Diagramm entspräche dies einem Pfad ohne jegliche Verzweigung, z. B. $\{\phi, \{1\}, \{1, 2\}, \{1, 2, 3\}\}$.

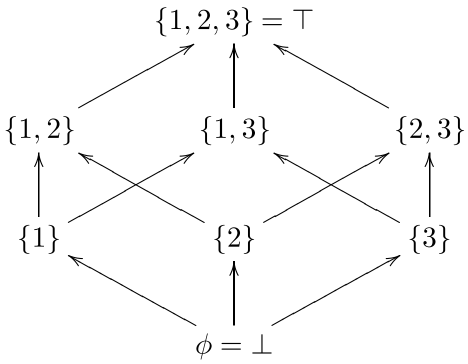


Abbildung 3.3: Beispiel eines vollständigen Gitters.

- Ein vollständiges Gitter genügt der *aufsteigenden Kettenbedingung* („ascending chain condition"), wenn für jede aufsteigende Kette $l_1 \sqsubseteq l_2 \sqsubseteq l_2 \sqsubseteq \ldots$ ein n existiert, so dass gilt $l_n = l_{n+1} = l_{n+2}$ usw. Das heißt jede aufsteigende Kette stabilisiert sich irgendwann.

Ist der Eigenschaftsraum nun ein vollständig geordnetes Gitter, das zudem der aufsteigenden Kettenbedingung genügt, so lässt sich zeigen, dass der in Abbildung 3.2 vorgestellte DFA-Algorithmus in jedem Fall terminiert.

Der Beweis ist in [NNH99] angeführt, wobei in der if-Anweisung in Zeile 17 des DFA-Algorithmus (Abbildung 3.2, Seite 15) statt $f_a(\mathit{Analysis}[a]) \sqcup \mathit{Analysis}[b] \neq \mathit{Analysis}[b]$ die äquivalente Bedingung $f_a(\mathit{Analysis}[a]) \not\sqsubseteq \mathit{Analysis}[b]$ verwendet wird.[1]

In Rahmen der vorliegenden Arbeit wurde diese Bedingung deshalb verändert, weil dadurch eine explizite Angabe von $\sqsubseteq$ nicht mehr nötig ist. Dadurch verringert sich der Aufwand bei der Implementierung einer Analyse.

3.1.4.2 Anwendbarkeitsbedingung

Während die Terminierung durch die angegebenen Bedingungen sichergestellt werden kann, bleibt noch die Frage nach der Korrektheit. Dabei bedeutet Korrektheit hier lediglich, dass die Einträge in $\mathit{Analysis}[y]$ die Lösung des Gleichungssystems

$$z_y = \bigsqcup_{x \in Fy} f_x(z_x)$$

bilden. Dass der obige Algorithmus eine solche Lösung tatsächlich berechnet, lässt sich zeigen, wenn zusätzlich zu den Anforderungen an den Eigenschaftsraum gilt, dass die Funktionen aus $\mathcal{F}$ monoton bzgl. $\sqsubseteq$ sind. Eine Funktion f heißt monoton, falls für alle $a, b \in L$ gilt $a \sqsubseteq b \Rightarrow f(a) \sqsubseteq f(b)$.

In diesem Fall lässt sich zeigen, dass die errechnete Lösung $Z = \{z_0, z_1, \ldots, z_n\}$ minimal ist, in dem Sinne, dass für alle anderen Lösungen $L' = \{z'_0, z'_1, \ldots, z'_n\}$ und wenigstens einen Zustand z_m gilt $z_m \sqsubseteq z'_m \wedge z_m \neq z'_m$. Der Beweis ist wiederum in [NNH99] zu finden.

[1] Die Äquivalenz folgt bei Betrachtung der Negation der Bedingungen: Aus $f_a(\mathit{Analysis}[a]) \sqcup \mathit{Analysis}[b] = \mathit{Analysis}[b]$ folgt $f_a(\mathit{Analysis}[a]) \sqsubseteq \mathit{Analysis}[b]$ und aus $f_a(\mathit{Analysis}[a]) \sqsubseteq \mathit{Analysis}[b]$ und $\mathit{Analysis}[b] \sqsubseteq \mathit{Analysis}[b]$ (Reflexivität) folgt $f_a(\mathit{Analysis}[a]) \sqcup \mathit{Analysis}[b] = \mathit{Analysis}[b]$.

Es bleibt jedoch anzumerken, dass damit lediglich garantiert wird, dass der DFA-Algorithmus eine Lösung des Gleichungssystems auf Seite 12 liefert. Ob die gefundene Lösung jedoch korrekt ist, im Sinne einer sinnvollen Analyse und zuverlässiger Ergebnisse, hängt hauptsächlich von der Wahl der Transferfunktionen ab. Dementsprechend wird die obige Bedingung im Folgenden als „Anwendbarkeitsbedingung" bezeichnet.

3.2 Anpassung an die Anforderungen der JVM

Im letzten Abschnitt wurde ein allgemeines Verfahren für die DFA angegeben, dem ein nicht näher spezifizierter Eigenschafts- oder Zustandsraum zu Grunde lag.

In diesem Abschnitt soll nun auf die Besonderheiten bei der Analyse von Java Bytecode eingegangen werde. Zunächst werden Probleme betrachtet, die daraus resultieren, dass der Java Bytecode eine sehr maschinennahe Sprache ist.

In Abschnitt 3.2.2 wird dann die im Rahmen dieser Arbeit entwickelte Grobstruktur eines Zustandsraumes angegeben, der für die Bytecode Analyse gut geeignet ist. Anschließend wird in Abschnitt 3.2.3 die Form der zugehörigen Transferfunktionen dargestellt. In Abschnitt 3.2.4 wird ein weiterer Zustandsraum für die Bytecodeanalyse vorgestellt.

Abschließend werden in Abschnitt 3.2.5 die zur Behandlung von Subroutinen und Ausnahmen nötigen Anpassungen und Erweiterungen vorgestellt.

3.2.1 Spezielle Probleme bei der Analyse von Java Bytecode

Bei der Programmanalyse geht es zumeist um Programme, die als Quellcode in einer Hochsprache, wie beispielsweise Java, vorliegen. Bei der Analyse des wesentlich maschinennäheren Java Bytecodes tritt eine ganze Reihe zusätzlicher Schwierigkeiten auf.

Ein großes Problem bereitet das Fehlen von Variablennamen: Während eine Eigenschaft bei der Analyse einer Hochsprache in der Regel an eine benannte Variable geknüpft wird, kann diese bei der Bytecode Analyse lediglich mit lokalen Variablen oder Stapelwörtern verknüpft werden. Eine Variable existiert jedoch während der gesamten Methode (oder vielmehr innerhalb ihres Gültigkeitsbereichs) und ist durch ihren Namen eindeutig bestimmt. Die Position eines Datenwortes auf dem Stapel ist hingegen keineswegs eindeutig, vielmehr werden laufend Wörter vom Stapel genommen und neue abgelegt.

Insbesondere werden die Operanden vieler Befehle, wie z. B. `iadd` oder `imul` vom Stapel entfernt und durch das Ergebnis ersetzt. Alle Eigenschaften, die nun bereits mit dem Operanden verbunden waren gehen damit wieder verloren.

Soll der Wert des Operanden eines Befehls an anderer Stelle erneut verwendet werden, so wird dieser vorher kopiert und beispielsweise in einer lokalen Variablen gesichert. Ein Zusammenhang zwischen der Kopie und dem Original ist später jedoch nicht mehr erkennbar, so dass Eigenschaften die der Kopie zugewiesen werden beim Original nicht vermerkt werden, und umgekehrt.

Diesem Problem kann abgeholfen werden, wenn man eine Kennung einführt, die es erlaubt, Original und Kopie miteinander zu identifizieren. Dieser Ansatz wird im nächsten Abschnitt genauer beschrieben.

Ein weiteres Problem, das durch die Stapelorientierung der JVM auftritt, ist, dass jeder Zustand von seinem Vorgängerzustand abhängt, insofern, als die Anzahl der Stapelwörter nur über den Vorgänger bestimmt werden kann. Das bedeutet zum einen, dass die Analyse meist alle Befehle betrachten muss und nicht auf einige ausgesuchte beschränkt werden kann, zumindest dann nicht, wenn der Zustand des Stapels berücksichtigt werden muss.

Zum anderen kann auch eine Rückwärtsanalyse nicht ohne weiteres durchgeführt werden. Schließlich ist die Größe des Stapels zum Ende der Methode zunächst unbekannt. Somit muss also vorher eine Vorwärtsanalyse durchgeführt werden, die zumindest die Größe des Stapels für jeden Zustand bestimmt.

Ein zusätzliches Problem, das die Größe des Stapels betrifft, ist die Tatsache, dass ein Befehl mehr als nur einen Vorgänger haben kann, und dass die verschiedenen Vorgängerzustände eine unterschiedliche Stackgröße aufweisen könnten. Diesem Problem wird jedoch bereits in der JVM Spezifikation durch die Festlegung begegnet, dass der Stapel an einer beliebigen Stelle des Programms immer die gleiche Größe aufweisen muss, egal auf welchem Wege dieser Programmpunkt erreicht wurde.

Ein weiteres Problem, dass allerdings nicht nur beim Java Bytecode auftritt sondern auch bei Java, bzw. bei fast allen objektorientierten Sprachen ist, dass Methoden überschrieben werden können. Dadurch ist statisch nur relativ schwer feststellbar, welche Methode aufgerufen wird: Benötigt die zu analysierende Methode beispielsweise einen Parameter vom Typ `java.lang.Object`, so kann ein beliebiges Objekt übergeben werden. Wird nun an diesem Objekt die Methode `toString()` aufgerufen, so muss nicht unbedingt die `toString()` Methode von `java.lang.Object` aufgerufen werden, da diese in der Klasse des tatsächlichen Parameters überschrieben worden sein kann. Dieses Problem tritt bei allen Interfaces und Klassen mit Unterklassen auf.

Das Problem kann auf zwei Arten umgangen werden. Zum einen kann man – wenn möglich – auf eine interprozedurale Analyse völlig verzichten und stattdessen bei jedem Methodenaufruf einen Standardwert zurückliefern. Dieser Ansatz wurde bei der Datentypanalyse in Abschnitt 4 und bei der Null-Pointer Analyse in Abschnitt 5 gewählt.

Ist diese Vorgehensweise nicht oder nur bei großem Verlust an Genauigkeit möglich, so kann man stattdessen *alle* in Betracht kommenden Methoden analysieren. Das Ergebnis der Methodenausführung wird dann – analog zur Vorgehensweise bei mehreren Vorgängern eines Befehls – aus der Vereinigung der verschiedenen Ergebniszustände berechnet.

Natürlich müssen in diesem Fall häufig sehr viele Methoden analysiert werden, auch wenn man eigentlich nur am Ergebnis für eine einzige Methode interessiert ist. Dennoch ist eine solche Vorgehensweise in manchen Fällen unumgänglich, wie beispielsweise bei der Analyse der Ausführungsdauer in Abschnitt 6.

Eine weitere Besonderheit, die es bei der Analyse von Java Bytecode zu berücksichtigen gilt, ist die Behandlung von Ausnahmen sowie die Analyse der Subroutinen. Dieser Problemkreis wird in Abschnitt 3.2.5 eingehend besprochen.

3.2.2 Ein allgemeiner Zustandsraum für die Bytecode Analyse

Bei der Analyse von Hochsprachen bestehen die Eigenschaften des Eigenschaftsraumes meist aus einer Menge von Variablen und Ausdrücken, die eine bestimmte Eigenschaft erfüllen.

Sucht man beispielsweise nach Konstanten, so ist eine Eigenschaft in obigem Sinne die Menge der Variablen deren Wert sich seit der ersten Zuweisung nicht geändert hat. Auf Grund der Einfachheit und Maschinennähe des Java Bytecodes, insbesondere aber auch deshalb, weil es keine Variablen im hochsprachlichen Sinne gibt, muss der Zustandsraum bei dessen Analyse meist etwas komplexer gebaut sein. Statt einer Liste von „interessanten" Variablen muss ein wesentlich größerer Teil des Zustandes der JVM beschrieben werden.

Bleibt die Frage, auf welche Teile des JVM-Zustandes sich die DFA beschränken kann. Zunächst einmal ist die Analyse im Rahmen dieser Arbeit auf einzelne Methoden begrenzt und Informationen über aufrufende und aufgerufene Methoden haben keinen direkten Einfluss auf das Ergebnis (diese Information geht nur über die Transferfunktionen entsprechender Befehle, wie z. B. `invokevirtual` in die Analyse ein). Dementsprechend muss auch nur der Teil des JVM-Zustandes simuliert werden, der diese Methode unmittelbar betrifft, also die lokalen Variablen und der Operandenstapel sowie die auf dem Heap angelegten Datenstrukturen.

Da Java allerdings von vornherein „multithreaded" ist, kann man die Zuweisung von Werten an Objektfelder und Arrays nicht ohne weiteres simulieren. Denn bei der Ausführung könnte eine Zuweisung bereits vor dem nächsten Lesezugriff wieder von einem anderen Thread überschrieben werden, so dass sich auch wichtige Eigenschaften ändern könnten.

Deshalb können Werte die in Datenstrukturen auf dem Heap liegen nur in Ausnahmefällen berücksichtigt werden, wenn sichergestellt ist, dass nur die analysierte Methode darauf zugreift. Im Sinne einer einheitlichen Vorgehensweise ist der Zustand deshalb im Weiteren auf lokale Variablen und Operandenstapel beschränkt.

Die Zustände des Zustandsraumes L bestehen also in der Regel aus zwei Listen mit „Datenwörtern" für Stapel und lokalen Variablen. Den Datenwörtern sind dann die eigentlich interessierenden Eigenschaften zugeordnet, wobei diese wiederum einem untergeordneten Eigenschaftsraum L' entstammen.

Während bislang die Begriffe Eigenschaftsraum und Zustandsraum weitgehend synonym verwendet wurden, bezeichnet im Folgenden der Eigenschaftsraum die Menge der möglichen Eigenschaften die einzelnen Datenwörtern zugeordnet werden können. Zustände sind dagegen aus einem Stapel und lokalen Variablen aufgebaut, die wiederum aus Elementen des Eigenschaftsraumes bestehen.

Terminierungsbedingunen

Im Rahmen der vorliegenden Arbeit wurden die allgemeinen Terminierungsbedingungen des DFA-Algorithmus auf den oben entwickelten speziellen Zustandsraum übertragen. Es zeigt sich, dass ein solcher Zustandsraum den Bedingungen genügt, wenn sie vom zu Grunde liegenden Eigenschaftsraum erfüllt werden. Somit kann man sich beim Nachweis, dass eine Datenflussanalyse terminiert, auf die Betrachtung des Eigenschaftsraumes beschränken.

In Abschnitt 3.1.4 wurden zwei Terminierungsbedingungen genannt:

1. Auf dem Zustandsraum L muss ein vollständiges Gitter $(L, \sqsubseteq, \sqcup, \perp, \top)$ definiert sein.

2. Die aufsteigende Kettenbedingung muss erfüllt sein.

Um daraus Bedingungen für den Eigenschaftsraum abzuleiten, muss zunächst eine Halbordnung auf dem Zustandsraum definiert werden. Besteht der Eigenschaftsraum aus dem vollständigen Gitter $(L', \sqsubseteq', \sqcup', \sqcap', \bot', \top')$, so kann mit dessen Hilfe nun ein vollständiges Gitter auf L mit folgenden Komponenten definiert werden:

- L: Die Zustände des Zustandsraumes sollen den jeweiligen Zustand der JVM widerspiegeln, wie oben erläutert. Dementsprechend enthält jeder Zustand l zwei Listen mit Elementen aus L', $l.stack$ und $l.lVars$, für Stapel und lokale Variablen. Auf das i-te Element wird jeweils mit $l.stack[i]$ bzw. $l.lVars[i]$ zugegriffen.

  Dabei ist $l.lVars$ in allen Zuständen von gleicher Größe, die Größe von $l.stack$ variiert hingegen zwischen 0 und der maximal möglichen Anzahl von Stapelwörtern $maxStack$. Die Anzahl der lokalen Variablen wird mit $\#l.lVars$ bezeichnet, die (aktuelle) Größe des Stapels mit $\#l.stack$. Für den Index i bei Zugriffen auf lokale Variablen und Stapel gilt entsprechend $0 \leq i < \#l.lVars$ und $0 \leq i < \#l.stack$.

- $\sqsubseteq$: Mit Hilfe von $\sqsubseteq'$ wird nun eine Halbordnung $\sqsubseteq$ auf L definiert, indem die Relation an die Stapelwörter und lokalen Variablen „weitergereicht" wird. Für zwei beliebige Zustände $l_1, l_2 \in L$ gilt $l_1 \sqsubseteq l_2$ wenn folgende Bedingungen erfüllt sind:

  - $H1$: Die Größe des Stacks von l_1 ist nicht größer als die von l_2,
    also $\#l_1.stack \leq \#l_2.stack$.
  - $H2$: Alle Stapelwörter von l_1 sind nicht „größer" als die von l_2,
    d. h. für alle $0 \leq i < \#l_1.stack$ gilt $l_1.stack[i] \sqsubseteq' l_2.stack[i]$.
  - $H3$: Alle lokalen Variablen von l_1 sind nicht „größer" als die von l_2,
    d. h. für alle $0 \leq i < \#l_1.lVars$ gilt $l_1.lVars[i] \sqsubseteq' l_2.lVars[i]$.

  Die Bedingung H1 wird nur der Vollständigkeit halber gefordert. Tatsächlich wird, wie bereits erwähnt, von korrektem Java Bytecode gefordert, dass die Größe des Stacks an einer bestimmten Stelle im Programm immer gleich ist, egal auf welchem Pfad diese Stelle erreicht wurde. Dementsprechend werden immer nur solche Zustände mit gleicher Stapelgröße verglichen.

- $\sqcup$ und $\sqcap$: Die Vereinigungs- (kleinste obere Schranke) bzw. Schnittoperation (größte untere Schranke) wird auf die komponentenweise Vereinigungs bzw. Schnittoperation $\sqcup'$ und $\sqcap'$ zurückgeführt. Hat ein Zustand mehr Stapelwörter als der andere, so werden diese im Ergebniszustand einfach übernommen (Vereinigung) bzw. ignoriert (Schnitt). Das bedeutet für $l = l_1 \sqcup l_2$ gelten folgende Bedingungen:

  - *O1*: $\#l.stack = \max(\#l_1.stack, \#l_2.stack)$.
  - *O2*: Für alle $0 \leq i < \min(\#l_1.stack, \#l_2.stack)$ gilt
    $l.stack[i] = l_1.stack \sqcup' l_2.stack$ und
    für alle $\min(\#l_1.stack, \#l_2.stack) \leq j < \#l.stack$ gilt
    $l.stack[j] = l_1.stack[j]$ wenn $\#l_1.stack > \#l_2.stack$ und $l.stack[j] = l_2.stack[j]$ sonst.

  Für $l = l_1 \sqcap l_2$ gilt analog:

  - *U1*: $\#l.stack = \min(\#l_1.stack, \#l_2.stack)$
  - *U2*: Es gilt $l.stack[i] = l_1.stack[i] \sqcap' l_2.stack[i]$ für alle $0 \leq i < \#l.stack$.

  Es gilt außerdem:

  - *O3/U3*: Für alle $0 \leq j < \#l.lVars$ gilt $lVars[j] = l_1.lVars[j] \diamond l_2.lVars[j]$, wobei $\diamond \in \{\sqcup, \sqcap\}$.

- $\bot$: Als kleinstes Element wird der Zustand $\bot$ gewählt, dessen Stapel leer ist und dessen lokale Variablen alle den Wert $\bot'$ enthalten. Das heißt $\#\bot.stack = 0$ und für alle $0 \leq i < \#\bot.lVars$ gilt $\bot.lVars[i] = \bot'$.

- $\top$: Das größte Element ist der Zustand $\top$ dessen Stapel maximale Größe hat und dessen Datenwörter alle den Werte $\top'$ haben: $\#\top.stack = maxStack$ und für alle $0 \leq i < maxStack$ und $0 \leq j < \#\top.lVars$ gilt $\top.stack[i] = \top'$ bzw. $\top.lVars[i] = \top'$.

Man kann leicht zeigen, dass es sich bei der so definierten Struktur $(L, \sqsubseteq, \sqcup, \sqcap, \bot, \top)$ um ein vollständiges Gitter handelt, wenn dies auch auf L' zutrifft. Gleiches gilt für die aufsteigende Kettenbedingung. Der Beweis wird in Anhang A.1 skizziert.

3.2.3 Transferfunktionen

Obwohl die Transferfunktionen natürlich vor allem vom Eigenschaftsraum sowie den mit der Analyse verfolgten Zielen abhängen, soll in diesem Abschnitt eine Klasse von Transferfunktionen beschrieben werden, die häufig verwendet werden können.

Die Ausführung eines Befehls durch die JVM erfolgt in der Regel nach folgendem Schema: Zunächst werden die Operanden des Befehls vom Stapel genommen oder aus den lokalen Variablen gelesen. Anschließend erfolgt die Berechnung der Ergebnisse und schließlich werden diese auf dem Stapel abgelegt oder in lokale Variablen geschrieben. Dabei hängt die Anzahl der Operanden vom Befehl (und dessen Argumenten im Bytecode) und nicht vom momentanen Zustand der Maschine ab. Ebenso ist die Zahl der Ergebniswörter zustandsunabhängig.

Somit scheint eine ähnliche Vorgehensweise auch bei der Formulierung der Transferfunktionen angebracht, wobei natürlich die Berechnung der Ergebnisse entfällt. An ihre Stelle tritt die Berechnung der *Eigenschaften* der Ergebnisse.

Im Allgemeinen kann eine so aufgebaute Transferfunktion durch fünf Komponenten bestimmt werden: Die Anzahl der Operanden auf dem Stapel, die Anzahl und Indizes der Operanden in den lokalen Variablen, die Zahl der Ergebnisse, die auf dem Stapel abgelegt werden müssen, die Zahl und die Position der zu beschreibenden lokalen Variablen sowie eine Funktion, um die Eigenschaften der Ergebnisse aus denen der Operanden zu bestimmen.

Dabei werden die ersten vier Komponenten ausschließlich durch den Befehl und seine Argumente im Bytecode bestimmt. Es bleibt jedoch anzumerken, dass die Positionen von Operanden und Ergebnissen der *Analyse* nicht unbedingt die selben sein müssen, wie die bei der *Berechnung* des Befehls. Beispielsweise sind diese bei einer Rückwärtsanalyse in der Regel vertauscht. Wichtig ist im Folgenden lediglich, dass die Anzahl und die Positionen aller betroffenen Daten vom Zustand, auf den die Transferfunktion angewendet wird, unabhängig ist.

Um die Transferfunktion für einen Befehl zu bestimmen wird also nur die Angabe einer Funktion für jedes Ergebniswort benötigt. Anders ausgedrückt handelt es sich um eine Klasse von Transferfunktionen in der jede Funktion f, aus m verschiedenen Funktionen $f_i : E^n \rightarrow E$ besteht. Diese berechnen aus den Eigenschaften von n Operanden die Eigenschaft eines von m Ergebnissen. Dabei stammen die Eigenschaften wiederum aus dem Eigenschaftsraum E.

Das Problem ist nun, diese f_i so zu wählen, dass eine dergestalt aufgebaute Transferfunktion der Anwendbarkeitsbedingung genügt, also monoton ist. Dabei ist $f : Z \rightarrow Z$ auf einem Zustandsraum Z definiert, der nach dem oben genannten Schema aus E konstruiert wurde.

Man kann zeigen, dass eine Transferfunktion, die aus mehreren Teilfunktionen zur Berechnung der Ergebnisse aufgebaut ist, die Anwendbarkeitsbedingung erfüllt, wenn jede dieser Ergebnisfunktionen monoton in ihren Parametern ist. Allerdings gilt dies nur, so lange die Position und Anzahl von Parametern und Ergebnissen vom Zustand unabhängig ist. Der Nachweis ist in Anhang A.2 aufgeführt.

3.2.4 Eigenschaftsraum mit eindeutiger Kennzeichnung gleicher Datenwörter

In diesem Abschnitt soll eine weitere Spezialisierung des Zustandsraumes vorgestellt werden, die im Rahmen dieser Arbeit entwickelt wurde, um folgendem Problem zu begegnen: Wie eingangs bereits erwähnt, werden die Operanden eines Befehls bei dessen Ausführung in der Regel vom Stapel entfernt, so dass alle mit diesem Datum verknüpften Eigenschaften für die Analyse nachfolgender Zustände verloren gehen. Andererseits werden Werte, die im späteren Verlauf der Methode erneut benötigt werden kopiert (indem sie beispielsweise in eine lokale Variable geschrieben werden), so dass dasselbe Datum an zwei unterschiedlichen Stellen im Zustand steht.

Es wäre nun wünschenswert, feststellen zu können, welche Elemente von Stapel und lokalen Variablen tatsächlich die gleichen Werte enthalten, um so Eigenschaften die für eines der Datenwörter festgestellt werden auch an dessen Kopien „vererben" zu können.

Zu diesem Zweck kann jedem Datum eine eindeutige Identifikationsnummer verliehen werden. Die Transferfunktionen müssen dann so erweitert werden, dass diese Nummer erhalten bleibt falls das Datum kopiert wird. Wird ein neues Datenwort erzeugt, so muss auch eine neue Kennungsnummer vergeben werden.

Dabei stellt sich die Frage, wie die Kennungen mit den eigentlichen Eigenschaften verbunden werden soll, so dass das Resultat wiederum einen Eigenschaftsraum ergibt, der die Terminierungsbedingungen erfüllt.

Dazu wird die Menge möglicher Kennungsnummern K' durch zwei spezielle Kennungen $\bot_K$ und $\top_K$ zu einem vollständigen Gitter $(K, \sqsubseteq_K, \sqcup_K, \bot_K, \top_K)$ mit $K = K' \cup \{\bot_K, \top_K)$ erweitert.

Wie üblich bezeichnet $\bot_K$ ein Element, dass noch nicht analysiert wurde. Das Element $\top_K$ wird allen Datenwörtern zugeordnet, die keine Kennung benötigen, da sie noch nicht kopiert wurden. Auch bei der Vereinigung zweier Zustände wird diese spezielle Kennung vergeben, wenn zwei Wörter mit unterschiedlichen Kennungen vereinigt werden sollen.

Dementsprechend wird die Halbordnung $\sqsubseteq_K$ für beliebige $k \in K$ durch $\bot_K \sqsubseteq_K k \sqsubseteq_K \top_K$ bestimmt. Die Elemente von K' sind durch $\sqsubseteq_K$ nicht vergleichbar.

Damit gilt für alle $k, k' \in K$:

$$\begin{aligned} k \sqcup_K \bot_K &= k \\ k \sqcup_K \top_K &= \top_K \\ k \sqcup_K k' &= \top_K \end{aligned}$$

Das so definierte vollständige Gitter wird nun mit dem Eigenschaftsraum $(E, \sqsubseteq_E, \sqcup_E, \bot_E, \top_E)$ zu einem neuen Eigenschaftsraum $(E \times K, \sqsubseteq, \sqcup, \bot\, \top)$ vereinigt, wobei für alle $e, e' \in E$ und $k, k' \in K$ gilt:

$$\begin{aligned} (e,k) \sqsubseteq (e',k') &\iff e \sqsubseteq_E e' \wedge k \sqsubseteq_K k' \\ (e,k) \sqcup (e',k') &= (e \sqcup_E e', k \sqcup_K k') \\ \bot &= (\bot_E, \bot_K) \\ \top &= (\top_E, \top_K) \end{aligned}$$

Man kann zeigen, dass ein solches „kartesisches Produkt” zweier Gitter wiederum ein vollständiges Gitter ergibt, gleiches gilt für die aufsteigende Kettenbedingung (vgl. [NNH99]). Dementsprechend erfüllt der so definierte Eigenschaftsraum die in Abschnitt 3.2.2 aufgestellte Terminierungsbedingungen (vorausgesetzt, diese werden von E erfüllt).

3.2.5 Ausnahmebehandlung und Subroutinen

Bei dem unter 3.1 vorgestellten Algorithmus für die Datenflussanalyse wurden zwei Besonderheiten des Java Bytecodes nicht berücksichtigt: Die Behandlung von Ausnahmen (Exceptions) und Subroutinen.

In den folgenden Abschnitten werden die im Rahmen dieser Arbeit entwickelten Erweiterungen zur Analyse von Ausnahmen und Subroutinen vorgestellt.

3.2.5.1 Ausnahmen

Wird eine Ausnahme ausgelöst, so wird diese entweder durch eine entsprechende Behandlungsroutine abgefangen oder die Methode, in der die Aunahme ausgelöst wurde, wird beendet. Letzterer Fall kann bei der Datenflussanalyse zumeist außer Acht gelassen werden. Schließlich wird die Methode dann sofort beendet, ohne dass irgendwelche anderen Befehle ausgeführt werden. Deshalb kann diese Möglichkeit zwar in der Transferfunktion des zugehörigen Befehls berücksichtigt werden, wie z. B. bei der in Abschnitt 5 vorgestellten Null-Pointer Analyse, hat aber auf die allgemeine Vorgehensweise keinerlei Einfluss.

Anders verhält es sich, wenn die Ausnahme innerhalb eines geschützten Bereichs ausgelöst wird und durch eine entsprechende Behandlungsroutine abgefangen wird. In diesem Fall wird die Methode nicht einfach beendet, sondern die Ausführung wird mit der Behandlungsroutine fortgeführt. Dementsprechend muss auch die Analyse des Bytecodes an beiden Stellen fortgeführt werden. Allerdings genügt es nicht, einfach den Fluss zu erweitern und von jedem Befehl der eine entsprechende Ausnahme auslösen kann eine, Kante zum Beginn der zugehörigen Behandlungsroutine einzufügen. Denn während die Zustandsänderung bei der normalen

Ausführung des Befehls an Adresse l durch f_l beschrieben wird, ändert sich der Zustand beim Auslösen einer Ausnahme auf andere Weise: Der Stapel wird verworfen und eine Referenz auf ein Objekt vom Typ der Ausnahme wird abgelegt. Das Löschen des Stapels und Ablegen einer Referenz vom Typ E wird im Folgenden durch die Funktion f_E durchgeführt.

Das bedeutet, dass nun auf einen Vorgängerzustand zwei unterschiedliche Funktionen angewandt werden müssen, abhängig davon, für welchen der Nachfolger der Ergebniszustand bestimmt ist. Für normale Nachfolger muss f_l angewandt werden, für Behandlungsroutinen vom Typ E dagegen f_E. Da der in Abbildung 3.2 vorgestellte Algorithmus eine solche Vorgehensweise nicht unterstützt, muss er entsprechend erweitert werden.

An Stelle der Abbildung $f_.$, die jeder Adresse l eine Transferfunktion f_l zuweist, wird eine neue Abbildung $f_{(.,.)} : F \to \mathcal{F}$ eingeführt, die jedem Paar (l, l') aus dem Fluss F eine Transferfunktion $f_{(l,l')}$ zuweist. Es werden nun also sowohl der aktuelle Befehl als auch dessen Nachfolger berücksichtigt.

Für einen Befehl an Adresse l bei dessen Ausführung eine Ausnahme vom Typ E ausgelöst werden kann und der durch eine entsprechende Behandlungsroutine geschützt ist, sieht die Abbildung folgendermaßen aus:

$$f_{l,l'} = \begin{cases} f_l, & \text{wenn } l' \neq a_E \\ f_E, & \text{sonst} \end{cases}$$

Dabei bezeichnet a_E die Startadresse der Behandlungsroutine.

Da eine Behandlungsroutine nach JVM-Spezifikation nur durch das Auslösen einer Ausnahme erreicht werden kann, und nicht etwa durch Sprungbefehle, besteht bei der Festlegung $f_{l,a_E} = f_E$ auch nicht die Gefahr, dass es Befehle gibt, für die $f_{l,a_E} = f_l$ gilt.

Man kann sich leicht davon überzeugen, dass eine solche Vorgehensweise an den Anwendbarkeits- und Terminierungsbedingungen des Algorithmus nichts ändert. Der in [NNH99] geführten Korrektheitsbeweis kann problemlos für die oben vorgestellte Zuordnung $f_{l,l'}$ erweitert werden, ohne seine Gültigkeit zu verlieren.

3.2.5.2 Subroutinen

Prinzipiell ist die Analyse von Subroutinen durchaus mit der oben beschrieben DFA durchführbar. Wie die folgenden Ausführungen zeigen, brächte dies jedoch verschiedene Nachteile, so dass ein erweitertes Konzept zur Behandlung von Subroutinen vorgestellt wird.

Um eine Subroutine zu analysieren könnte eine `jsr` Anweisung so wie ein unbedingter Sprung behandelt werden. Die `ret` Anweisung könnte entsprechend als bedingter Sprung aufgefasst werden, wobei als mögliche Sprungziele die Nachfolger aller `jsr` Anweisungen gelten, die zur entsprechenden Subroutine springen.

Diese Vorgehensweise birgt jedoch ein Problem, das am Beispiel der in Kapitel 5 beschriebenen Null-Pointer Analyse veranschaulicht werden soll: Angenommen, im Vorgängerzustand einer `jsr` Anweisung sei der lokalen Variable 1 das Attribut `nonnull` zugeordnet, d. h. die Variable enthält an dieser Stelle in jedem Fall eine gültige Referenz. Im Vorgängerzustand einer anderen `jsr` Anweisung mit dem selben Ziel sei der Variablen das Attribut `unknown` zugeordnet, es ist also nicht bekannt, ob die Variable eine gültige Referenz enthält oder nicht.

Die Transferfunktionen beider `jsr` Befehle lassen die Variable unverändert, jedoch müssen beide Ergebniszustände als Vorgänger der ersten Anweisung der Subroutine vereinigt werden. Im Ergebnis muss der Variablen 1 nun das Attribut `unknown` zugewiesen werden, da nicht sichergestellt werden kann, dass diese eine gültige Referenz enthält.

Die Subroutine wird nun normal weiter analysiert, wobei angenommen wird, dass der analysierte Code die Variable 1 nicht verändert. Dementsprechend werden auch, nach der `ret` Anweisung, den Nachfolgern der beiden `jsr` Befehle Zustände zugeordnet, deren erste lokale Variable das Attribut `unknown` erhält.

Diese Zuordnung ist jedoch für den Nachfolger des ersten `jsr` zwar nicht falsch, jedoch sehr ungenau. Schließlich wurde die betrachtete Variable zwischen dem `jsr` und seinem Nachfolger nicht verändert, trotzdem ging die `nonnull`-Eigenschaft verloren.

Dieses Problem kann umgegangen werden, indem die Subroutine für jeden `jsr` Aufruf separat analysiert wird: Wird der entsprechende Aufruf erreicht, so wird eine neue DFA-Instanz gestartet. Dabei ist der Anfangszustand, der dem ersten Befehl der Subroutine zugeordnet wird, einfach der Vorgängerzustand des `jsr`. Die `ret` Anweisung entspricht dabei dem `return` in einer normalen Methode.

Das Ergebnis der Subroutinen Analyse geht dann in die übergeordnete Analyse als Transferfunktion des `jsr` Befehls ein, indem dessen Vorgängerzustand in den Endzustand der untergeordneten Analyse transformiert wird.

Diese Transferfunktion erfüllt das Anwendbarkeitskriterium, die Monotonie, wenn die Transferfunktionen der Befehle in der Subroutine diese erfüllen. Denn während der Analyse werden die Zustände nur durch die Vereinigungsoperation und diese Transferfunktionen transformiert. Da jedoch die Vereinigung ebenfalls monoton in beiden Argumenten ist, besteht die Funktion der gesamten Subroutine letztlich aus einer Verknüpfung von monotonen Funktionen. Eine solche ist jedoch ebenfalls monoton.

Bei der Auswertung der Analyseergebnisse gilt es jedoch eines zu beachten: Zwar sind die Ergebnisse für die Zustände der Befehle außerhalb von Subroutinen bei diesem Verfahren wesentlich genauer, für jeden Befehle innerhalb einer Subroutine existieren jedoch mehrere, meist unterschiedliche Zustände – für jeden Aufruf der Routine einer.

Um auch für diese Befehle zu einem sinnvollen Ergebnis zu kommen, müssen die verschiedenen Anfangszustände der Subroutine zunächst vereinigt werden, und schließlich ein erneuter Analyselauf für diese Routine durchgeführt werden.

4 Der Java Bytecode Verifier

4.1 Einführung

In der Spezifikation ([LY97]) der virtuellen Java Maschine wird ein sogenannter „Bytecode-Verifier" beschrieben, der die Aufgabe hat, verschiedene Anforderungen an den Bytecode bereits vor dessen Ausführung zu überprüfen.

Bei der Analyse des Bytecodes geht es in erster Linie darum, zu verifizieren, dass alle auf Java-Sprachebene vorhandenen Mechanismen, wie etwa Zugriffsbeschränkungen durch `public` und `protected`, auch auf Bytecodeebene korrekt beachtet werden (siehe [LY97, Yel97, EHM96]). Insbesondere muss auch sichergestellt werden, dass jeder Befehl im Bytecode die richtige Anzahl an Operanden auf dem Stack und in den lokalen Variablen vorfindet und dass diese vom erwarteten Typ sind.

Im Folgenden Abschnitt werden die zu verifizierenden Anforderungen aufgeführt. In Abschnitt 4.3 wird die zur Verifikation verwendete Datenflussanalyse dargestellt. Abschließend werden in Abschnitt 4.4 zusätzliche Verifikationsschritte beschrieben, die nicht mittels DFA durchgeführt werden können.

4.2 Anforderungen an den Java Bytecode

Die Anforderungen die korrekter Java Bytecode erfüllen muss, werden in der JVM-Spezifikation in zwei Gruppen unterteilt: statische und strukturelle Anforderungen.

4.2.1 Statische Anforderungen

Die statischen Anforderungen sollen sicherstellen, dass die Klassendatei (siehe Abschnitt 2.1) das korrekte Format hat und in sich konsistent ist. Sie können als erfüllt gelten, wenn die Klassendatei fehlerlos eingelesen werden kann. Des Weiteren nennt die JVM-Spezifikation folgende Anforderungen an den Bytecode der in der Datei gespeicherten Methoden :

- Der Bytecode darf nicht leer sein, und der erste Befehl muss bei Adresse 0 beginnen. Die Länge des Codeteils muss mit der in der Klassendatei angegebenen übereinstimmen.
- Alle Befehle müssen mit einem gültigen Opcode beginnen. Die Ziele aller Sprungbefehle müssen gültige Adressen innerhalb der Methode sein, an denen ein neuer Befehl beginnt.

- Alle Zeiger auf Einträge im Konstantenpool müssen auf einen gültigen Eintrag verweisen. Zum Beispiel muss der Operand der `ldc` Operation zum Laden von Konstanten auf Einträge vom Typ `int`, `float`, oder `String` verweisen.
- Der Befehl `invokespecial` darf nur zum Aufruf der Methode `<init>`[1], privater Methoden oder Methoden der Superklasse verwendet werden. Außerdem darf die Methode `<init>` wiederum nur durch `invokespecial` aufgerufen werden, nicht durch einen der anderen invoke-Befehle.
- Der `anewarray` Befehl darf nicht verwendet werden um einen Array mit mehr als 255 Dimensionen anzulegen.
- Die `new` Operation darf nicht dazu verwendet werden, einen Array anzulegen.
- Alle Operationen, die auf lokale Variablen zugreifen, dürfen keine Variable verwenden, deren Index außerhalb der in der Klassendatei angegeben maximalen Anzahl an Variablen liegt. Operationen, die Datentypen von zwei Wörtern Länge verwenden, dürfen maximal auf die vorletzte gültige Variable zugreifen.

Die Verifikation der genannten Anforderungen erfolgt bereits beim Einlesen der Klassendatei bzw. beim Aufbau der internen Verwaltungsstrukturen für jeden Befehl. Da sich diese statischen Anforderungen leicht nachprüfen lassen soll hier auf Details der Überprüfung nicht näher eingegangen werden.

4.2.2 Strukturelle Anforderungen

Die strukturellen Anforderungen dienen dazu, sicherzustellen, dass alle Java Sicherheitsmechanismen auch auf Bytecode Ebene eingehalten werden. Außerdem soll sichergestellt werden, dass die Ausführung der Befehle die Integrität der JVM selbst nicht beeinflussen kann. Des Weiteren sollen Fehler im Bytecode erkannt werden.

Im Einzelnen werden die folgenden Bedingungen geprüft:

- Jede Operation findet bei ihrer Ausführung die korrekte Anzahl an Operanden auf dem Operanden Stack vor, und diese sind vom richtigen Datentyp.
- Während der Ausführung einer Methode hat der Operanden Stapel hat an einer bestimmten Stelle immer dieselbe Größe, egal welcher Pfad im Kontrollflussgraph durchlaufen wurde, um diese Stelle zu erreichen.
- Die Reihenfolge der Wörter, die zu einem Zwei-Wort-Datentyp gehören (`long` oder `double`), darf nicht verändert werden und die beiden Wörter dürfen nicht getrennt werden.
- Ein Lesezugriff auf eine lokale Variable darf nur erfolgen, falls die Variable bereits initialisiert wurde (d. h. falls sie einen gültigen Wert enthält).

[1] Bei der `<init>`-Methode handelt es sich um den Konstruktor des Objekts an dem die Methode aufgerufen wird.

- Es darf nicht versucht werden, ein Wort vom Stapel zu holen, wenn dieser leer ist, bzw. abzulegen, wenn er bereits die maximale Größe (die im Methodeneintrag der Klassendatei vermerkt ist) erreicht hat.

- Jedes neu erzeugte Objekt muss genau einmal durch Aufruf der Methode `<init>` initialisiert werden. Das heißt, die Konstruktor Methode `<init>` darf nur an einem nicht initialisierten Objekt aufgerufen werden. Ebenso muss ein Objekt vor dem ersten Zugriff (egal ob auf Felder oder Methoden) initialisiert werden.

- Uninitialisierte Objekte dürfen nicht in lokalen Variablen gespeichert sein oder auf dem Stapel liegen, wenn ein Rückwärts-Sprung ausgeführt wird. Außerdem darf innerhalb eines durch eine Ausnahmebehandlung geschützten Bereichs kein uninitialisiertes Objekt in den lokalen Variablen gespeichert sein.

- Eine Konstruktormethode `<init>` muss entweder eine weitere Konstruktormethode des selben Objekts aufrufen oder einen Konstruktor der Vaterklasse, bevor irgendein anderer Zugriff auf das Objekt erfolgt. Von dieser Regel ausgenommen ist einzig der Konstruktor von `java.lang.Object`.

- Die Datentypen der auf dem Stapel liegenden Argumente zu jedem Methodenaufruf müssen mit den im Methoden Deskriptor angegebenen übereinstimmen.

- Eine abstrakte („`abstract`") Methode darf nicht aufgerufen werden.

- Der Datentyp jeder `return` Instruktion muss mit dem Typ des Rückgabewertes der aktuellen Methode übereinstimmen. Wird eine Referenz zurückgeliefert, so muss diese ein Objekt aus der Klasse des Rückgabetyps oder einer Unterklasse dieses Typs referenzieren.

- Beim Zugriff auf „`protected`" Felder oder Methoden muss geprüft werden, ob diese in der aktuellen Klasse oder einer ihrer Oberklassen deklariert wurden. Außerdem muss das tatsächliche Objekt, an dem der Zugriff ausgeführt wird, aus der selben Klasse wie die aktuelle Methode oder einer Unterklasse stammen.

- Die Klasse jedes durch `putfield` oder `getfield` gespeicherten oder geladenen Objekts muss aus der Klasse oder einer Unterklasse des jeweiligen Feldes sein. Ebenso müssen alle Referenzen die in Arrays gespeichert werden auf Objekte verweisen, die entweder aus der als Basistyp angegebenen Klasse oder einer ihrer Unterklassen stammen.

- Das Argument jeder `athrow` Instruktion muss eine Instanz der Klasse `java.lang.Throwable` oder einer ihrer Unterklassen sein.

- Der letzte Befehl im Bytecode der Methode ist entweder ein `return` oder ein unbedingter Sprung, so dass die Ausführung der Methode nicht über den letzten Befehl hinaus führen kann.

- Subroutinen dürfen sich nicht rekursiv selbst aufrufen, d. h. eine Subroutine darf immer nur dann erneut aufgerufen werden, wenn sie vorher bereits durch die zugehörige `ret` Anweisung beendet wurde.

- Jede Instanz vom Typ `returnAdress` kann nur einmal als Rücksprungadresse einer `ret` Anweisung verwendet werden.

Die Frage, inwieweit diese Regeln tatsächlich Typsicherheit garantieren und die Einhaltung der Java Sicherheitmechanismen gewährleisten, ist nicht Gegenstand dieser Arbeit und soll hier nicht weiter diskutiert werden. Eine eingehende Untersuchung der Frage findet man beispielsweise in [Qui99, Nip, vON99].

Obwohl die genannten Einschränkungen recht unterschiedlicher Natur sind, so haben sie dennoch gemeinsam, dass sie nur dann überprüft werden können wenn der Zustand der JVM direkt vor der Ausführung bekannt ist. Deshalb muss zur Verifikation der Bedingungen eine DFA durchgeführt werden. Diese wird im nächsten Abschnitt eingehend beschrieben.

4.3 Datenflussanalyse für die Überprüfung der Typsicherheit

Da die allgemeine Vorgehensweise bei der Datenflussanalyse bereits ausführlich erläutert wurde (vgl. Abschnitt 3.2), beschränken sich die weiteren Ausführungen auf den Aufbau der Zustände des Zustandsraumes (oder vielmehr des Eigenschaftsraumes auf den der Zustandsraum aufbaut) und die Wahl der Transferfunktionen.

4.3.1 Datentypen und Eigenschaftsraum

Die Zustände bei der DFA müssen so gewählt werden, dass sich die oben genannten strukturellen Anforderungen an den Java Bytecode mit dem Resultat der Analyse möglichst leicht verifizieren lassen.

Als wichtigste Elemente müssen offensichtlich die Datentypen der Stapelwörter und der lokalen Variablen simuliert werden. Dabei fällt – sozusagen als Nebenprodukt – auch die Anzahl der Stapelwörter mit an, die ebenfalls benötigt wird. Welche Datentypen unterschieden werden müssen, soll in den nächsten Abschnitten untersucht werden. Außerdem wird angegeben, wie die Datentypen vereinigt werden können (im Sinne der Operation $\sqcup$ aus Abschnitt 3.2.2) bzw. ob die Typen bzgl. $\sqsubseteq$ vergleichbar sind.[2]

Entsprechend den Datentypen der JVM werden auch für die Zustände folgende Typen definiert: `int`, `float`, `returnAddress` und `ref` (für `reference`). An Stelle von `long` und `double` werden die Datentypen `longL` und `longH` sowie `doubleL` und `doubleH` definiert, da diese Typen aus zwei Datenwörtern bestehen. Dabei soll *typ*`H` für das Wort stehen, dass im Stapel weiter oben liegt bzw. in den lokalen Variablen den höheren Index der beiden Wörter hat. Entsprechend steht *typ*`L` für das niedrigere Wort, bzw. das Wort in der lokalen Variable mit niedrigerem Index.

Die Datentypen `byte`, `short` und `char` werden, wie unter 2.3 beschrieben, auf dem Operanden Stapel und in den lokalen Variablen durch Werte vom Typ `int` dargestellt. Dementsprechend müssen diese Typen bei der Analyse nicht berücksichtigt werden.

[2] In Abschnitt 3.2.2 wurden $\sqcup$ und $\sqsubseteq$ auf komplexen Zuständen definiert, auf Ebene der Eigenschaften (Datenwörter) wurde stattdessen $\sqcup'$ und $\sqsubseteq'$ verwendet. Im Sinne einer einfacheren Lesbarkeit werden in diesem Abschnitt $\sqcup$ und $\sqsubseteq$ für die Vereinigung und den Vergleich von Eigenschaften verwendet, die entsprechenden Operationen der Zustandsebene werden speziell gekennzeichnet.

Die oben genannten Datentypen sind untereinander unvergleichbar im Sinne der $\sqsubseteq$ Relation, insbesondere auch `longH` und `longL` bzw. `doubleH` und `doubleL`. Werden zwei Wörter des gleichen Typs vereint, so hat das Ergebnis ebenfalls diesen Typ.

Des Weiteren wird ein zusätzlicher Datentyp $\top$ für „nicht initialisiert/unbekannt" eingeführt, der lokalen Variablen zugewiesen wird, die möglicherweise noch nicht beschrieben wurden. Außerdem wird diese Eigenschaft allen Wörtern zugeordnet, deren Datentyp nicht bekannt ist. Dabei gilt für alle oben festgelegten Datentypen $Typ \sqsubseteq \top$, d. h. die Vereinigung ergibt $Typ \sqcup \top = \top$.

Diese Festlegung ist sinnvoll, da die Vereinigungsoperation zum Zusammenführen zweier Zustände verwendet wird. Wenn nun eine lokale Variable in einem Zustand nicht initialisiert wird, so besteht offensichtlich auch für den Ergebniszustand die Möglichkeit, dass die Variable noch nicht initialisiert wurde. Erst wenn eine Variable in allen Vorgängerzuständen initialisiert wurde, darf sie auch im zusammengeführten Zustand als initialisiert gelten.

Einem Datenwort, dem in den zu vereinigenden Zuständen unterschiedliche Datentypen zugeordnet werden, kann im Ergebnis keiner dieser Typen zugeordnet werden. Entsprechend gilt auch $Typ_1 \sqcup Typ_2 = \top$, für unterschiedliche Datentypen Typ_1 und Typ_2.

Schließlich wird noch ein weiterer Datentyp eingeführt: $\perp$ wird wie üblich verwendet um noch nicht analysierte Daten zu bezeichnen.

Somit ist $\top$ der „größte" und $\perp$ der „kleinste" mögliche Datentyp (bzgl. $\sqsubseteq$), d. h. für jeden Datentyp t gilt $\perp \sqsubseteq t \sqsubseteq \top$.

Zusätzlich wird der Typ `ref` um einen Eintrag für die Klasse des Objekts, auf welche die Referenz verweist, erweitert.Es gilt $\texttt{ref}(C) \sqsubseteq \texttt{ref}(C')$, wenn C eine Unterklasse von C' ist, oder $C = C'$. Somit ist die Vereinigung zweier Referenzen $\texttt{ref}(A) \sqcup \texttt{ref}(B) = \texttt{ref}(C)$ wenn C die „kleinste" Oberklasse von A und B ist, d. h. es gibt keine Unterklasse von C die ebenfalls Oberklasse von A und B wäre. Damit gilt für beliebige Referenzen $\texttt{ref}(C) \sqsubseteq \texttt{ref}(java.lang.Object)$.

Für Arrays A und A' gilt $\texttt{ref}(A) \sqsubseteq \texttt{ref}(A')$, falls beide Arrays die gleiche Anzahl von Dimensionen haben und der Basistyp BA von A „kleiner" ist als der Basistyp BA' von A' , also $\texttt{ref}(BA) \sqsubseteq \texttt{ref}(BA')$ gilt. Arrays unterschiedlicher Dimension sowie Arrays und „normale" Klassen sind nicht vergleichbar – mit einer Ausnahme: Für jeden Array gilt $\texttt{ref}(A) \sqsubseteq \texttt{ref}(java.lang.Object)$.

Die Behandlung von Referenzen, von denen lediglich bekannt ist, dass sie ein bestimmtes Interface implementieren (d. h. die Klasse, der das Objekt angehört, ist unbekannt), ist etwas komplizierter. Es tritt das Problem auf, dass ein Interface von mehreren anderen abgeleitet sein kann. Dadurch wird es wesentlich aufwändiger gemeinsame Oberinterfaces zu finden. Vor allem ist es durchaus möglich, dass nicht ein einziges gemeinsames Oberinterface existiert, sondern mehrere.

Sollen zwei Interfaces vereinigt werden, so muss zunächst jedes Oberinterface des einen, mit den Oberinterfaces des anderen vereinigt werden. Aus der Menge der resultierenden Interfaces können dann alle Elemente gestrichen werden, für die ein abgeleitetes Interface ebenfalls in der Menge vorhanden ist.

Sei beispielsweise Interface I_1 von J_1 und K abgeleitet sowie I_2 von J_2 und K. Außerdem seien J_1 und J_2 von Interface J abgeleitet. Wenn nun zwei Werte vom Typ I_1 und I_2 vereinigt werden sollen, so ergibt sich folgendes (wobei *Object* für *java.lang.Object* steht, das auch als oberstes Interface fungiert): $J_1 \sqcup J_2 = J$; $J_1 \sqcup K = Object$; $K \sqcup J_2 = Object$; $K \sqcup K = K$.

Der vereinigte Wert implementiert somit die Interfaces $J, Object$ und K. Da J und K von *Object* abgeleitet sind, genügt die Angabe von J und K. Eine Referenz die Interfaces J und K implementiert wird im Folgenden als $\texttt{ref}(J, K)$ bezeichnet.

Die Vereinigung einer Referenz einer Klasse C mit einem Interface I wird durchgeführt, indem I mit den Interfaces, die C implementiert, vereinigt wird. Das heißt, wenn C die Interfaces I_C, J_C implementiert so gilt $\texttt{ref}(C) \sqcup \texttt{ref}(I) = \texttt{ref}(I_C, J_C) \sqcup \texttt{ref}(I)$.

Zusätzlich wird noch die Null-Referenz, $\texttt{reference}(null)$, eingeführt. Da diese Referenz jeden beliebigen Typ annehmen kann, muss bei der Vereinigung mit anderen Referenzen einer beliebigen Klasse C gelten: $\texttt{ref}(null) \sqcup \texttt{ref}(C) = \texttt{ref}(C)$, d. h. es gilt $\texttt{ref}(null) \sqsubseteq \texttt{ref}(C)$.

Diese Festlegung ist insofern sinnvoll, als hier ausschließlich *Datentypen* betrachtet werden. Eine Referenz, die aus der Vereinigung von $\texttt{ref}(C)$ und $\texttt{ref}(\text{null})$ entsteht, muss zwar keine gültige Referenz enthalten, ein Zugriff durch einen Bytecode, der ein Wort vom Typ $\texttt{ref}(C)$ erwartet, ist jedoch erlaubt. Will man verhindern, dass ein solcher Zugriff eine Null-Pointer Exception auslöst, so muss eine spezielle Null-Pointer Analyse durchgeführt werden, wie in Abschnitt 5 beschrieben.

Die Menge aller definierten Datentypen wird im Weiteren mit T bezeichnet. Zusammen mit $\sqsubseteq$ ergibt sich so ein vollständiges Gitter $(T, \sqsubseteq, \sqcup, \sqcap, \perp, \top)$. Der Nachweis, dass dieser Eigenschaftsraum alle Terminierungsbedingungen erfüllt, wird in Anhang A.3 geführt.

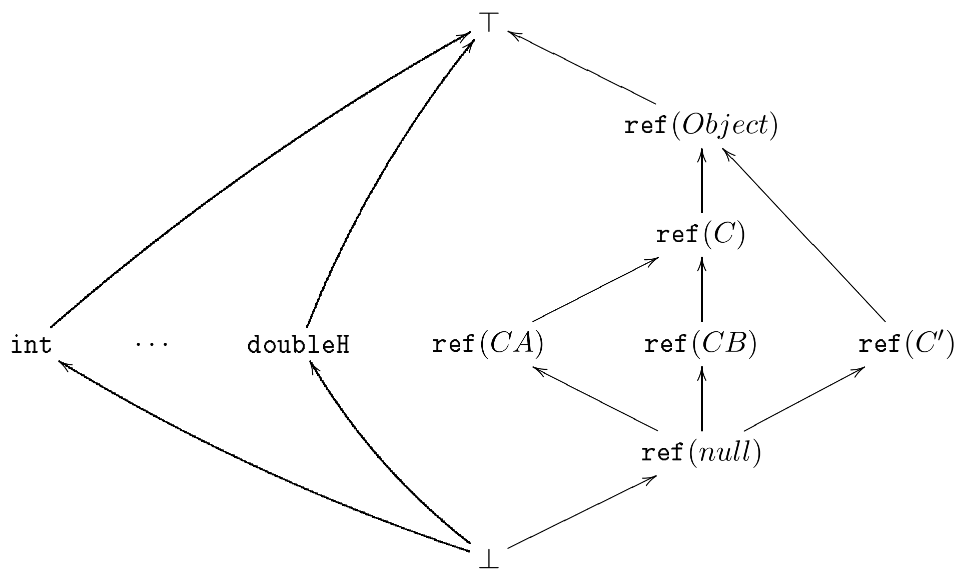


Abbildung 4.1: Halbordnung $\sqsubseteq$ auf der Menge der Datentypen T.

In Abb. 4.1 ist die beschriebene Halbordnung graphisch dargestellt. Dabei stehen $\texttt{ref}(C)$ und $\texttt{ref}(C')$ beispielhaft für Referenzen beliebiger Klasse C und C', die in keiner Ableitungsbeziehung stehen. Die Klassen CA und CB stellen Klassen dar, die von C abgeleitet wurden und $\texttt{ref}(Object)$ steht für Referenzen der Klasse `java.lang.Object`. Arrays und Interfaces sind nicht dargestellt.

Zusätzlich wird zu jeder Referenz vermerkt, ob sie bereits initialisiert wurde. Dieses Initialisierungsattribut hat jedoch keinen Einfluss auf $\sqcup$ oder $\sqsubseteq$ (auf die Initialisierung von Objekten wird im Rahmen von Abschnitt 4.3.2 näher eingegangen).

Anfangszustand

Der Anfangszustand dieser Analyse wird mit Hilfe des Typdeskriptors der zu analysierenden Methode aufgebaut. Dieser gibt an, wie viele Parameter die Methode erwartet und von welchem Typ Parameter und Rückgabewerte sind. Des Weiteren ist im zugehörigen Methodeneintrag in der Klassendatei die maximale Größe des Stapels und die Anzahl der lokalen Variablen vermerkt.

Der Anfangszustand wird nun so konstruiert, dass der Stapel leer ist, und die Zahl der lokalen Variablen der im Methodeneintrag angegebenen entspricht. Alle Variablen enthalten zunächst den Wert $\top$. Anschließend wird der Typdeskriptor ausgewertet und die lokalen Variablen in denen Parameterwerte enthalten sind, werden mit entsprechenden Datentypen versehen.

Bei nicht statischen Methoden enthält dabei die Variable 0 den Typ des Objekts, an dem die Methode aufgerufen wird, bei statischen Methoden enthält sie hingegen den Datentyp des ersten Parameters.

Dabei darf für alle Parameter des Typs `reference` angenommen werden, dass sie bereits initialisiert wurden. Wäre dies nicht der Fall, so würde die Verifikation der aufrufenden Methode scheitern, da ein Methodenaufruf mit nicht initialisierten Parametern nicht erlaubt ist.

Die Zustände aller anderen Befehle werden wie üblich auf den Zustand $\perp_Z$ gesetzt, dessen Stapel leer ist und dessen lokale Variablen alle den Wert $\perp$ enthalten.

4.3.2 Transferfunktionen

Um ein möglichst genaues Abbild der tatsächlichen Zustände der JVM bei der Ausführung der Methode zu bekommen, muss jede Änderung der lokalen Variablen und des Stapels simuliert werden.

Dabei haben die Transferfunktionen, die in diesem Abschnitt beschrieben werden, zwei Aufgaben zu erfüllen: Zum einen müssen Zustandsänderungen simuliert werden, wie in Abschnitt 3.2.3 allgemein beschrieben. Zum anderen müssen bei jeder Ausführung die Bedingungen, die an den jeweiligen Befehl geknüpft sind, geprüft werden. So müssen beispielsweise alle Operanden vom richtigen Datentyp sein, bei Feldzugriffen muss geprüft werden, ob der Zugriff erlaubt ist usw.

Die Zuordnung von Transferfunktionen zu Adressen erfolgt, wie üblich, einfach über den Bytecode, der an der entsprechenden Adresse steht. Dementsprechend konzentrieren sich die Ausführungen im Folgenden auf den Aufbau der Transferfunktionen, die Zuordnung wird nicht weiter behandelt.

Der Teil der Transferfunktionen, der den Zustand aktualisiert, ist wie in Abschnitt 3.2.3 beschrieben aufgebaut: Zunächst wird eine bestimmte Anzahl von Wörtern vom Stapel des Ausgangszustandes entfernt und lokale Variablen gelesen. Anschließend werden neue Datenwörter auf dem Stapel abgelegt und lokale Variablen beschrieben.

Für die meisten Befehle hängt der Datentyp der Ergebnisse – und nur dieser wird benötigt – nicht von den Operanden und tatsächlichen Ergebnissen der Berechnung ab, sondern wird bereits durch die Art des Befehls festgelegt. Dementsprechend kann auch die Transferfunktion der meisten Befehle durch die folgenden vier Komponenten festgelegt werden: Eine Liste POP von Datenwörtern, die vom Stapel genommen werden (oberstes Element zuerst), eine entsprechende Liste $PUSH$ von Wörtern, die abgelegt werden sollen (in der umgekehrten Reihenfolge in der sie abgelegt werden sollen), sowie zwei Mengen $READ$ und $WRITE$ mit Paaren von Datenwörtern und Indizes in die lokalen Variablen.

Die allgemeine Transformationsfunktion eines Befehls B ist in Abbildung 4.2 angegeben. Dabei wird mit $z.stack.pop()$ das oberste Element vom Stapel von z entfernt und zurückgeliefert, mit $z.stack.push(t)$ wird ein Wort vom Typ t abgelegt. Des Weiteren bezeichnet POP_B die Liste der durch Befehl B vom Stapel entfernten Operanden ($PUSH_B, READ_B, WRITE_B$ analog) und mit $POP_B[i]$ wird das i-te Element der Liste zurückgeliefert. Außerdem steht t für einen beliebigen Datentyp.

1: **for** $0 \leq i < \#POP_B$ **do**
2:     **if** $z.stack.pop() \not\sqsubseteq POP_B[i]$ **then**
3:         Fehler!
4:     **end if**
5: **end for**
6: **for all** $(i,t) \in READ_B$ **do**
7:     **if** $z.lVars[i] \not\sqsubseteq t$ **then**
8:         Fehler!
9:     **end if**
10: **end for**
11: **for** $0 \leq i < \#PUSH_B$ **do**
12:     $z.stack.push(PUSH_B[i])$
13: **end for**
14: **for all** $(i,t) \in WRITE_B$ **do**
15:     $z.lVars[i] := t$
16: **end for**

Abbildung 4.2: Grobstruktur der Transformationsfunktion eines Befehls B.

Wie man sieht, wird in den Zeile 2 und 7 geprüft, ob die gelesenen Operanden mit den erwarteten übereinstimmen. Dazu wird die $\sqsubseteq$ Relation genutzt, da ein Datentyp an Stelle eines anderen stehen kann, falls er „kleiner" als dieser ist (für Klassen bedeutet dies, dass es sich um eine Unterklasse des erwarteten Typs handelt). Dabei ist zu beachten, dass die Eigenschaft $\perp$ zwar von dieser Regel ausgenommen ist, jedoch ohnehin nur benutzt wird, um noch nicht analysierte Daten zu kennzeichnen, also im Vorgängerzustand eines Befehls nicht auftreten kann.

Es bleibt noch anzumerken, dass die Überprüfung eventuell auf einem „provisorischen" Zustand ausgeführt wird, da noch nicht alle Vorgänger des aktuellen Befehls analysiert sein müssen. Das spielt jedoch insofern keine Rolle, als ein Operand, der nicht den richtigen Typ hat diesen auch später nicht erlangt, wenn alle Vorgängerzustände analysiert wurden. Schließlich wird der Zustand aus der Vereinigung der Ergebniszustände aller Vorgänger berechnet, so dass sich der Operand nur ändert, wenn einer seiner Vorgänger „größer" wird, d. h. auch der Operand selbst wird „größer".

Natürlich beschränkt sich die Überprüfung nicht unbedingt auf den Datentyp der Operanden. Für verschiedene Befehle sind unten weitere Überprüfungen angegeben. Im Allgemeinen muss jedoch bei allen Befehlen, die den Stapel betreffen, noch überprüft werden, ob die maximale Stapelgröße überschritten wird, bzw. ob der Stapel genügend Elemente enthält.

Die Parameter der Transferfunktion ($PUSH_B, READ_B, WRITE_B$ und POP_B) sind für viele Befehle fest und können unabhängig vom zu analysierenden Code angegeben werden. So lässt sich die Funktion des `iadd` Befehls zum addieren zweier ganzer Zahlen beschreiben durch $POP_{iadd} = (\texttt{int}, \texttt{int})$ und $PUSH_{iadd} = (\texttt{int})$, sowie $READ_{iadd} = WRITE_{iadd} = \phi$ (da die lokalen Variablen nicht betroffen sind). Entsprechend kann auch der `iload_0` Befehl zum Laden eines ganzzahligen Wertes aus der lokalen Variable 0 mit $READ_{iload\_0} = \{(0, \texttt{int})\}$ und $PUSH_{iload\_0} = (\texttt{int})$ simuliert werden.

Die Datentypen von Operanden und Ergebnissen anderer Befehle hingegen lassen sich nicht fest angeben, sondern hängen von ihren Argumenten im Bytecode bzw. im Konstantenpool ab. Zu diesen Befehlen zählen Beispielsweise alle Methodenaufrufe: Jedem Aufruf ist ein Typdeskriptor im Konstantenpool zugeordnet, der Anzahl und Art der Parameter sowie den Rückgabetyp der aufgerufenen Methode bestimmt. Dementsprechend muss zunächst dieser Typdeskriptor ausgewertet werden, um $POP, READ, WRITE$ und $PUSH$ zu bestimmen.

Allerdings existiert auch noch eine dritte Gruppe von Befehlen, die nicht typisiert sind, bzw. bei denen die Ergebnisse den selben Typ haben wie die Operanden. Diese Gruppe besteht aus allen Befehlen, die der Verwaltung des Operanden Stapels dienen, wie `dup` oder `swap`.

Im Folgenden wird ein Überblick über die verschiedenen Befehle und ihre Transferfunktionen gegeben.

- Stapel Verwaltung (`pop`, `dup`, `swap`): Bei `pop` wird einfach das oberste Wort des Stapels entfernt. Bei `swap` wird das Wort zunächst entfernt und anschließend zweimal abgelegt, entsprechend werden bei `dup` mehrere Worte entfernt und in umgekehrter Reihenfolge abgelegt. Überprüft werden muss lediglich, dass die Stackgrenzen nicht verletzt werden (d. h. dass der Stack nicht zu groß wird bzw. genügend Wörter enthält).

- Feldzugriffe (`{get|put}static`, `{get|put}field`): Die Befehle dieser Gruppe haben als Argument im Bytecode einen Verweis auf einen Feldeintrag im Konstantenpool. Dieser enthält die Klasse, dem das Feld angehört sowie dessen Namen und Typ.
  Bei nicht statischen Feldzugriffen muss zunächst geprüft werden, ob der Operand auf dem Stapel vom Typ der Klasse ist, die das Feld enthält. Dabei kann das Objekt auch einer Unterklasse entstammen. Des Weiteren muss geprüft werden, ob der Zugriff auf das Feld erlaubt ist oder durch `private` bzw. `protected` verhindert wird.
  Schließlich wird überprüft, ob bei den `put` Befehlen der zweite Operand dem Typ des Feldes entspricht. Bei den `get` Befehlen wird ein Wert vom entsprechenden Typ auf dem Stapel abgelegt.
  Außerdem muss sichergestellt werden, dass das Objekt, welches das Feld enthält, initialisiert wurde. Ist an der Referenz vermerkt, dass sie noch nicht initialisiert wurde, so muss die Verifikation beendet werden.

- Methodenaufrufe (`invoke{virtual|static|special|interface}`): Diese werden ähnlich wie Feldzugriffe ausgeführt. Es wird wiederum der Typ des Objekts, der Operanden und der Rückgabewerte aus dem Konstantenpool gelesen. Anschließend wird überprüft, ob diese Typen mit den vom Stapel gelesenen übereinstimmen und die abgelegten Ergebnisse werden mit dem entsprechenden Typ versehen.
  Neben den Überprüfungen, ob die Methode aufgerufen werden darf, muss außerdem sichergestellt werden, dass die Methode nicht den Namen „`<clinit>`" (Klasseninitialisierungsmethode) trägt, da diese nur implizit von der JVM beim Laden neuer Klassen aufgerufen werden darf.
  Falls der die Methode den Namen „`<init>`" (Konstruktor) trägt, darf sie nur durch den Befehl `invokespecial` aufgerufen werden. Dabei muss zusätzlich überprüft werden, ob der Operand eine nicht initialisierte Referenz ist. Nach dem `invokespecial` Aufruf kann die Methode als initialisiert markiert werden.
  Bei allen anderen Methoden muss hingegen geprüft werden ob das Objekt, an dem sie aufgerufen werden, initialisiert wurde und anderenfalls wird die Verifikation abgebrochen.

- Objekterzeugung (`[a]new[array]`, `multianewarray`): Es wird ein neues Datenwort vom entsprechenden Typ auf dem Stapel abgelegt. Der Typ wird dabei durch einen Eintrag im Konstantenpool bestimmt. Die erzeugten Referenzen werden als nicht initialisiert gekennzeichnet.

- Konstanten aus Konstantenpool laden (`ldc[_w]`): Zunächst wird der Typ der zu ladenden Konstante bestimmt, anschließend wird ein neues Datenwort des entsprechenden Typs auf den Stack gelegt. Handelt es sich dabei um eine String Konstante, so wird ein Wort vom Typ `ref`(*java.lang.String*) abgelegt. Da ein solches Objekt von der JVM automatisch erzeugt und initialisiert wird, muss auch die Referenz als initialisiert markiert werden.

- Zugriff auf lokale Variablen (`Tload[_`*n*`]`, `Tstore[_`*n*`]`, `iinc`): Bei den Ladebefehlen wird zunächst überprüft, ob die angesprochene lokale Variable den richtigen Datentyp enthält, anschließend wird ein Datum dieses Typs auf dem Stapel abgelegt. Beim Speichern wird entsprechend das oberste Stapelwort entfernt und, falls es den richtigen Typ hatte, in die lokale Variable geschrieben. Dabei sind die Datentypen bei allen Befehlen außer `aload` und `astore`, die auf Referenzen arbeiten, bereits durch den Befehl selbst bestimmt. Bei Referenzen muss hingegen der tatsächlich gelesene Wert geschrieben werden, da die zugeordnete Klasse nicht allein durch den Befehl bestimmt werden kann.
  Der Befehl `iinc` dient dazu, einen `int` Wert in einer lokalen Variable um einen bestimmten Betrag zu erhöhen. Dementsprechend muss lediglich überprüft werden, ob diese lokale Variable tatsächlich vom Typ `int` ist.

- Sonstige Befehle: Alle anderen Befehle werden mit einer allgemeinen Transferfunktion verifiziert, wie oben beschrieben. Die erwarteten und geschriebenen Datentypen sind durch den jeweiligen Befehl festgelegt. Dabei führen alle Zugriffe auf nicht initialisierte Referenzen zu einem Fehler.
  Eine spezielle Behandlung verlangen jedoch die `return` Befehle: Nachdem der Operand vom Stapel genommen wurde, muss geprüft werden, ob dessen Typ sowohl mit dem des Befehls (z. B. `int` bei `ireturn`) als auch mit dem Rückgabetyp der aktuellen Methode

übereinstimmt.
Bei Sprungbefehlen werden außerdem der Stapel und die lokalen Variablen untersucht und die Verifikation scheitert, falls eine nicht initialisierte Referenz gefunden wird.

Alle Transferfunktionen sind – was die Aktualisierung des Zustandes betrifft – so aufgebaut, wie die in Abschnitt 3.2.3 untersuchten Transferfunktionen: Die Anzahl und Position der Operanden und Ergebnisse ist vom Zustand unabhängig und wird allein durch den Bytecode bestimmt. Außerdem wird der Typ des Ergebnisses durch eine monotone Funktion bestimmt, denn bei den meisten Befehlen sind die Ergebnistypen operandenunabhängig. Alle anderen Funktionen beschränken sich darauf, Operanden auf dem Stack zu kopieren oder ihre Position zu verändern. Somit hängt jedes Ergebniswort dieser Transferfunktionen von nur einem Operandenwort ab und bekommt den selben Typ wie dieses. Also sind auch diese Funktionen monoton. Damit ist die Anwendbarkeit des DFA-Algorithmus für diese Analyse sichergestellt.

An dieser Stelle soll nochmals auf die Initialisierung von Objekten eingegangen werden: Um ein Objekt zu initialisieren, muss an diesem die Methode `<init>` aufgerufen werden, erst dann kann die Referenz als initialisiert markiert werden. Durch den Methodenaufruf wird diese Referenz jedoch vom Stapel entfernt. Wenn das Objekt im weiteren Verlauf der Methode noch verwendet werden soll, so muss vorher eine Kopie angelegt worden sein. Diese gilt jedoch noch immer als nicht initialisiert.

Deshalb müssen bei der Initialisierung eines Objekts alle Referenzen, die auf das Objekt zeigen, entsprechend gekennzeichnet werden, nicht nur diejenige an der die `<init>` Methode aufgerufen wird. Dazu muss jedoch zunächst bekannt sein, welche Referenzen auf dieses Objekt zeigen – diese Information ist normalerweise nicht verfügbar, da nur bekannt ist, welcher Klasse das Objekt angehört, auf das eine Referenz zeigt.

Dieses Problem könnte mit einer eindeutigen Kennung für jedes Datenwort, wie unter 3.2.4 beschrieben, umgangen werden. Da diese Vorgehensweise jedoch mit einem erheblichen Aufwand verbunden ist, wurde für die Typanalyse eine einfachere Vorgehensweise implementiert: Es wird jeder neuen Referenz eine eindeutige Kennung zugeordnet, die beim Kopieren der Referenz erhalten bleiben muss. Bei der Vereinigungsoperation können allerdings nur uninitialisierte Referenzen mit der gleichen Kennung vereinigt werden. Ist die Kennung in verschiedenen Vorgängerzuständen unterschiedlich, so wird die Verifikation abgebrochen.

So bald die Methode `<init>` an einer nicht initialisierten Referenz aufgerufen wird, werden alle Referenzen mit dieser Kennung ebenfalls als initialisiert markiert und die Kennung wird entfernt.

Die Behandlung von Subroutinen und Ausnahmen erfolgt wie unter 3.2.5 beschrieben: Jeder Befehl der innerhalb eines Bereichs liegt, der durch eine Ausnahme geschützt wird, hat als möglichen Nachfolger nicht nur die direkten Nachfolger im Bytecode oder Ziele eventueller Sprünge, sondern auch den ersten Befehl der Ausnahmebehandlungsrountine. Dessen Zustand wird durch eine spezielle Transferfunktion berechnet, die den Stapel löscht und ein Objekt vom Typ der behandelten Ausnahme ablegt.

Für jeden Subroutinen-Aufruf (`jsr`) wird der Code der Subroutine bis zu deren Ende (`ret`) gesondert analysiert und das Ergebnis als Ergebniszustand des `jsr`-Befehls verwendet.

4.4 Weitere Verifikationsschritte

Die meisten Anforderungen an korrekten Java Bytecode werden entweder bereits beim Einlesen des Codes verifiziert oder durch die beschriebene Datentypanalyse abgedeckt. Allerdings bleiben noch zwei Anforderungen, die gesondert verifiziert werden müssen. Dabei handelt es sich um Einschränkungen bei der Ausnahmebehandlung:

- Der Beginn einer Ausnahmebehandlung darf nicht in dem durch sie geschützten Bereich liegen, außerdem muss die Anfangsadresse kleiner als die Endadresse des geschützten Bereichs sein.
- Die Bereiche zweier unterschiedlicher Ausnahmebehandlungsroutinen müssen entweder disjunkt sein, oder einer muss vollständig im anderen enthalten sein.

Beide Anforderungen werden vor Durchführung der DFA geprüft, indem die Adressen der geschützten Bereiche und der Behandlungsroutinen verglichen werden.

5 Null-Pointer Analyse

Sowohl in Java, als auch auf Bytecode Ebene können Referenzen nur durch das Anlegen neuer Objekte erzeugt und später nicht mehr verändert werden. Zeigerarithmetik, wie etwa in C oder C++, ist nicht möglich. Dadurch wird sichergestellt, dass außer der Null-Referenz keine ungültigen Referenzen existieren.

Somit kann bei der Dereferenzierung nur dann ein Fehler auftreten, wenn die Referenz den Wert Null enthält. In diesem Fall muss der Zugriff – laut JVM-Spezifikation [LY97] – abgefangen und eine Ausnahme vom Typ `NullPointerException` ausgelöst werden. Deshalb muss vor jedem Zugriff auf ein Objekt geprüft werden, ob die Referenz gültig oder Null ist.

Mit Hilfe der in diesem Abschnitt vorgestellten Null-Pointer Analyse (NPA) soll die Zahl der Laufzeitüberprüfungen reduziert werden. Es soll – soweit möglich – schon vor Ausführung eines Programms festgestellt werden, ob eine Referenz Null ist oder nicht. Die Ergebnisse werden dann dem Native-Code Compiler (siehe [Waw01]) zur Verfügung gestellt (in Anhang B.2 wird näher beschrieben, wie die Übergabe der Ergebnisse erfolgt).

Kann verifiziert werden, dass eine Referenz an einer Stelle, an der sie dereferenziert wird, gültig ist, so kann die Laufzeitüberprüfung entfallen. Wird andererseits festgestellt, dass die Referenz immer Null ist, so kann beim Kompilieren eine Warnung ausgegeben werden. Häufig hängt der Wert einer Referenz jedoch von den vorher durchlaufenen Programmteilen ab, und kann erst zur Laufzeit auf ihre Gültigkeit überprüft werden, so dass sicherlich nicht alle Überprüfungen eliminiert werden können.

Deshalb muss man sich bei der NPA auf die Forderung beschränken, dass eine Referenz an allen Punkten, wo das Attribut `null` oder `nonnull` verliehen wird, tatsächlich den Wert Null bzw. einen gültigen Wert hat. Dass umgekehrt, überall wo eine Referenz Null oder nicht Null ist, dies auch vom Verifier erkannt wird, kann nicht garantiert werden.

Die Null-Pointer Analyse verwendet die in Abschnitt 3.2 vorgestellte Datenflussanalyse, mit den an die Anforderungen der JVM angepassten Zuständen. In den beiden folgenden Abschnitten werden der Aufbau des Eigenschaftsraumes der den Zuständen zu Grunde liegt, sowie die Transferfunktionen näher betrachtet.

5.1 Aufbau des Eigenschaftsraumes

Bei der Null-Pointer Analyse handelt es sich um eine Vorwärts-Analyse. Es werden wie üblich Stack und lokale Variablen simuliert, wobei die Information, die zu den einzelnen Datenwörtern gespeichert wird, wesentlich geringer ist, als etwa bei der Analyse der Datentypen.

Statt jedem Datum den wirklichen Datentyp zuzuordnen wird lediglich einer der drei folgenden „Referenztypen" zugeordnet: `null` für Referenzen die den Wert Null enthalten, `nonnull` für Referenzen die einen Wert ungleich Null enthalten und `unknown` für alle Referenzen über die keine Information vorliegt. Auf eine Unterscheidung zwischen Referenzen und einfachen Datentypen wird verzichtet, stattdessen werden letztere mit dem Referenztyp `unknown` versehen. Zusätzlich wird ein Typ $\perp$ eingeführt, der allen noch nicht analysierten Datenwörtern zugeordnet wird.

A	B	$A \sqcup B$
`null`	`null`	`null`
`null`	`nonnull`	`unknown`
`null`	`unknown`	`unknown`
`nonnull`	`nonnull`	`nonnull`
`nonnull`	`unknown`	`unknown`
`unknown`	`unknown`	`unknown`

Tabelle 5.1: Bestimmung des Referenztyps aus den Vorgängerzuständen.

Bei der Zusammenführung der Vorgängerzustände eines Bytecode-Befehls werden die einzelnen Datenwörter auf dem Stack und in den lokalen Variablen wie üblich mit Hilfe einer Operation $\sqcup$ miteinander vereinigt. Diese Operation wird in Tabelle 5.1 beschrieben. Für die zugehörige Relation $\sqsubseteq$ gilt demnach `null`, `nonnull` $\sqsubseteq$ `unknown`. Außerdem ist $\perp$ das „kleinste" Element, d. h. es gilt $\perp \sqsubseteq r$ für jedes Attribut r.

Informationen über den Referenztyp eines Datenwortes können auf zwei Wegen gewonnen werden: Zum einen sind alle Referenzen, die innerhalb der ausgeführten Methode erzeugt werden, vom Typ `nonnull`, bzw. `null` falls die Referenz durch den `aconst_null` Befehl erzeugt wird. Andererseits kann jeder Referenz, auf die in der Methode bereits zugegriffen wurde, ebenfalls der Typ `nonnull` zugewiesen werden. Denn da während der Ausführung der Methode beim ersten Zugriff auf die Referenz eine Laufzeitüberprüfung durchgeführt werden muss, werden alle nachfolgenden Zugriffe nur dann ausgeführt, wenn die Referenz beim ersten Zugriff gültig war. Anderenfalls hätte bereits der erste Zugriff eine Ausnahme ausgelöst und die Methode beendet (es sei denn der Zugriff wird durch eine entsprechende Ausnahmebehandlung geschützt). In jedem Fall wird die Methode an dieser Stelle nicht fortgesetzt.

Die zuletzt beschriebene Vorgehensweise wird jedoch dadurch erschwert, dass alle Operanden auf dem Stack der JVM liegen und die Parameter eines Befehls vor dessen Ausführung vom Stack geholt werden. Wird auf eine Referenz zugegriffen, so wird das entsprechende Datenwort vom Stack entfernt, so dass jede Typzuweisung an dieses Datenwort ebenfalls verloren geht.

Um die durch Zugriffe gewonnene Typinformation dennoch nutzen zu können wird jeder Referenz eine eindeutige Kennung $k \in K$ zugewiesen, wie bereits unter 3.2.4 beschrieben. Wird ein Datenwort kopiert, so bleibt diese Kennung erhalten. Wird nun auf eine Referenz zugegriffen, so werden alle Datenwörter mit der entsprechenden Kennung ebenfalls als `nonnull` gekennzeichnet.

Die oben definierte Operation $\sqcup$ und die zugehörige Relation $\sqsubseteq$ werden entsprechend zu $\sqcup'$ und $\sqsubseteq'$ erweitert, um auch die eingeführten Kennungen in sinnvoller Weise zu berücksichtigen (die Vorgehensweise bei der Erweiterung wird wiederum in Abschnitt 3.2.4 beschrieben).

Dabei wird im Folgenden (A, a) für Referenztyp A und Kennung a geschrieben. Statt $(\perp, \perp_K)$, also der Eigenschaft mit minimalem Referenztyp und minimaler Kennung wird auch schlicht $\perp$ geschrieben. Für beliebige Referenzen (A, a) gilt dann $\perp \sqsubseteq' (A, a)$.

Die Relation $\sqsubseteq'$ ist in Abbildung graphisch dargestellt, wobei $a, b \in K$ wiederum für beliebige Kennungen stehen.

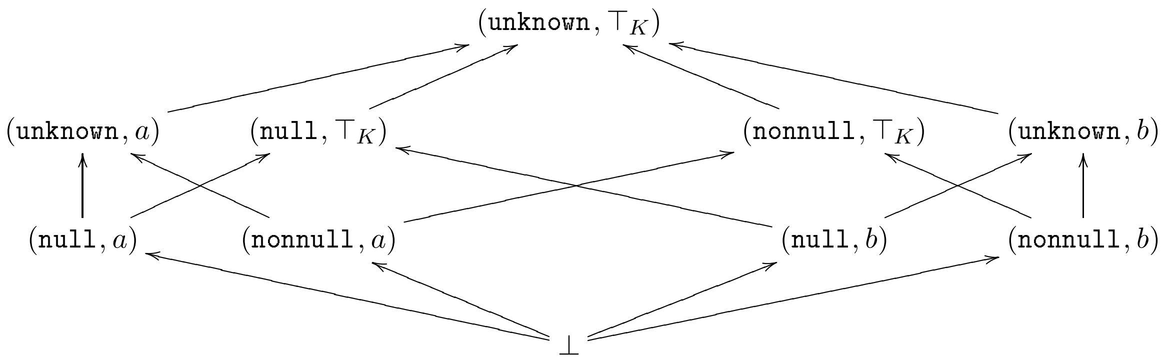


Abbildung 5.1: Halbordnung $\sqsubseteq'$ auf Referenztypen.

Für die $\sqcup'$ Operation auf zwei Referenzen gilt $(A, a) \sqcup (B, b) = (C, c)$ wobei C aus A und B bestimmt wird, wie in Tabelle 5.1 angegeben. Für c gilt $c = \top_K$, falls $a \neq b$ oder $c = a$ falls $a = b$.

Die Menge $R = \{\perp, \texttt{null}, \texttt{nonnull}, \texttt{unknown}\}$ der möglichen Referenzentypen ergibt zusammen mit $\sqsubseteq$ das vollständige Gitter $(R, \sqsubseteq_R, \sqcup_R, \texttt{unknown}, \perp)$. Da R endlich ist, wird auch die aufsteigende Kettenbedingung erfüllt. Da die Erweiterung um eine eindeutige Kennung an diesen Eigenschaften nichts ändert (vgl. Abschnitt 3.2.4) werden diese auch von $(R \times K, \sqsubseteq', \sqcup', (\texttt{unknown}, \top_K), (\perp, \perp_K))$ erfüllt. Somit ist die Terminierung des DFA Algorithmus sichergestellt.

Anfangszustand

Der Anfangszustand der Analyse besteht aus einem Zustand mit leerem Stapel und lokalen Variablen, denen die Eigenschaft $(\texttt{unknown}, \top_K)$ zugeordnet ist. Schließlich kann einer Methode, die eine Referenz als Parameter erwartet, durchaus der Wert Null übergeben werden. Außerdem werden gültige Kennungen erst vergeben, wenn ein Wert kopiert wird, so dass zunächst alle Kennungen den Wert $\top_K$ haben.

Wenn die Methode nicht statisch ist, wird der lokalen Variable 0 die Eigenschaft $(\texttt{nonnull}, \top_K)$ verliehen, da diese den `this` Zeiger enthält, der garantiert gültig ist.

5.2 Transferfunktionen

Wie schon bei der Standard Java Bytecode Verifikation erfolgt auch hier die Beschreibung der Menge der Transferfunktionen durch Angabe einer Funktion für jeden Bytecode Befehl. Die in Abschnitt 3.1.1 geforderte Zuordnungsfunktion $f_{.,.}$ zwischen Adressen a, deren Nachfolgeradresse a' und einer Transferfunktion $f_{a,a'}$ erfolgt wiederum durch Zuordnung der Transferfunktion des Befehls, der an der Adresse a steht.

Für die Null-Pointer Analyse lassen sich die Befehle in zwei Gruppen einteilen: Solche, die Referenzen erzeugen, kopieren oder dereferenzieren, und solche die nur mit einfachen Datentypen arbeiten.

Die Gruppe der Befehle, die nur mit einfachen Datentypen arbeiten, wird simuliert, indem die entsprechende Anzahl von Wörtern vom Stack genommen bzw. auf dem Stack abgelegt wird. Dabei wird der Referenztyp der Stackwörter nicht weiter beachtet, bzw. werden Werte vom Typ `unknown` abgelegt.

Die Zugriffe dieser Befehle auf lokale Variablen werden nicht ausgeführt, da die Zugriffe für die NPA irrelevant sind: Die gelesenen Werte werden ohnehin ignoriert, also können auch die Schreibzugriffe entfallen.

Zwar kann dadurch eine Referenz, die eigentlich durch einen einfachen Datentyp überschrieben werden sollte, in einer lokalen Variable stehenbleiben, so dass nachfolgende Lesezugriffe ein falsches Ergebnis liefern könnten. Aber auch dieser Fall beeinflusst das Ergebnis der NPA nicht: Bei einem solchen Lesezugriff kann es sich – auf Grund der Tatsache, dass korrekter Java Bytecode typisiert ist – nur um einen Befehl handeln, der einen einfachen Datentyp erwartet und die Variable ohnehin nicht ließt. Das Lesen einer Referenz kann erst wieder vorkommen, nachdem auch eine Referenz geschrieben wurde, wobei dieser Schreibzugriff die ungültige Referenz überschreibt.

Die Gruppe von Befehlen, die Referenzen betreffen, lässt sich wiederum unterteilen, in solche, die Referenzen kopieren (z. B. `astore` oder `dup`) und solche, die neue Referenzen erzeugen oder dereferenzieren (z. B. `new` und `getfield`). Nur durch die zuletzt genannten Befehle kann Information über den Referenztyp gewonnen werden.

Die Instruktionen, die Referenzen lediglich kopieren (`aload`, `astore`, `dup`) werden simuliert, indem den entsprechenden Zielspeicherplätzen der gleiche Referenztyp und insbesondere auch die gleiche Kennung zugewiesen wird, wie der Quelle. Der Befehl `aload_1` legt beispielsweise ein neues Stackwort an, mit der selben Kennung und dem gleichen Referenztyp wie die lokale Variable eins.

Dabei gilt es zu beachten, dass beim Kopieren von Datenwörtern mit der Kennung $\top_K$ sowohl der Quelle als auch der Kopie eine neue, bislang noch nicht verwendete Kennung zugeordnet werden muss. Dadurch können Quelle und Kopie bei späteren Zugriffen identifiziert werden.

Für die Gruppe von Befehlen, die Referenzen erzeugen oder verändern, gelten folgende Regeln:

- Alle durch [[`multi`]a]`newarray` und `new` erzeugten Referenzen sind vom Referenztyp `nonnull`, da die Referenz auf das neu erzeugte Objekt zeigt und somit gültig ist. Dasselbe gilt für Objekte die durch `ldc` und `ldc_w` erzeugt werden, wenn es sich bei den zu ladenden Konstanten um Zeichenketten handelt, da dabei automatisch ein neues Objekt vom Typ `java.lang.String` erzeugt wird.
- Die durch `aconst_null` erzeugten Referenzen sind vom Typ `null`.
- Sobald eine Referenz durch `getfield`, `putfield`, `invoke{virtual|special|interface}`, `aaload`, `aastore`, `monitorenter` oder `monitorexit` dereferenziert wird, kann allen Werten mit der gleichen Kennung im Folgezustand der Referenztyp `nonnull` zugewiesen werden. Denn wird eine Referenz während der Ausführung der

Methode dereferenziert, so wird – falls nötig – eine Laufzeitüberprüfung durchgeführt. Wird dabei eine Ausnahme ausgelöst, so kommen auch nachfolgende Befehle nicht mehr zur Ausführung. Anderenfalls war die Referenz gültig und ist damit auch bei nachfolgenden Zugriffen gültig. Dies gilt so lange, bis der Referenz bei der Vereinigung zweier Zustände ein anderer Referenztyp zugewiesen wird.

- Beim Durchlaufen einer `ifnull` Abfrage werden zwei Ergebniszustände erzeugt: Einer für den direkten Nachfolger der Abfrage und einer für das Ziel des bedingten Sprunges. Im ersten Zustand werden alle Referenzen mit entsprechender Kennung auf `nonnull` gesetzt, im zweiten auf `null`, da der Sprung ja genau dann ausgeführt wird, wenn die Referenz `null` ist.
  Analog wird auch die Instruktion `ifnonnull` behandelt.

Wie schon bei der Analyse der Datentypen, ist auch hier die Transferfunktion nicht immer allein durch den Befehl bestimmt. Es müssen vielmehr die entsprechenden Argumente im Bytecode und im Konstantenpool der Methode ausgewertet werden. So hängt die Zahl der vom Stapel zu entfernenden Datenwörter beim `invokevirtual` Befehl beispielsweise von der Zahl der Parameter der aufgerufenen Methode ab. Erst wenn diese Zahl bekannt ist, können die entsprechenden Wörter vom Stapel genommen werden. Die Anzahl der Parameter und Ergebnisse wird wiederum aus den Typdeskriptoren bestimmt.

Die so definierten Transferfunktionen sind monoton: Die kopierenden Funktionen verändern die gelesenen Eigenschaften nicht, wobei die Position von Quell- und Zieldatenwort durch den Befehl vorgegeben ist. Die Ergebnisse aller anderen Befehle sind unabhängig vom Ausgangszustand jeweils `null` oder `nonnull`, demnach sind auch diese Funktionen monoton.

6 Bestimmung der maximalen Ausführungsdauer

6.1 Motivation und Zielsetzung

Ziel dieser Analyse ist es, für eine gegebene Methode die maximale Ausführungsdauer (auch als Worst Case Execution Time (WCET) bezeichnet) zu bestimmen. Eine solche Analyse ist nötig, damit auch Methoden nicht vertrauenswürdiger Komponenten ununterbrechbar, beispielsweise als Interrupthandler, ausgeführt werden können. Wird die maximale Ausführungsdauer nicht vorher bestimmt, so besteht die Gefahr, dass eine solche Methode das System übermäßig lange blockiert oder gar überhaupt nicht terminiert. Da diese ununterbrechbar abläuft, hätte das einen Stillstand des gesamten Systems zur Folge.

Neben Interrupthändlern sieht auch das in [Fel01] erweiterte Schedulingkonzept von JX den Aufruf von unuterbrechbaren Methoden eines – nicht unbedingt vertrauenswürdigen – „application schedulers" vor, so dass auch hier eine WCET Analyse nötig ist.

Natürlich ist es nicht immer möglich, die maximale Ausführungsdauer einer Methode zu bestimmen. Diese Tatsache geht aus der Unentscheidbarkeit des allgemeinen Halteproblems (siehe [Sch95]) hervor. Dieses besteht darin, zu einer vorgegebenen Turingmaschine M und einem Eingabewort x festzustellen, ob M angesetzt auf x anhält oder nicht. Anders ausgedrückt besteht das allgemeine Halteproblem aus der Aufgabe, festzustellen, ob eine Methode terminiert, wenn sie mit einem vorgegebenen Satz von Parametern aufgerufen wird.

Da dies jedoch im Allgemeinen bereits unentscheidbar ist, kann natürlich auch die maximale Laufzeit nicht bestimmt werden. Es wird somit nur für gewisse Methoden und unter bestimmten Voraussetzungen möglich sein, eine Abschätzung der Ausführungsdauer anzugeben.

Dabei wird im Folgenden davon ausgegangen, dass die Dauer jedes einzelnen Befehls bekannt ist, so dass es allein darum geht, zu bestimmen, welcher Befehl wie oft ausgeführt wird. Natürlich sind die so möglichen Ergebnisse insofern sehr ungenau, als viele Faktoren, welche die Ausführungszeit einzelner Maschinenbefehle betreffen, außer Acht gelassen werden. Beispielsweise spielt der Füllzustand der Pipelines des Prozessors eine große Rolle, so wie auch die Zahl der Cache Treffer bzw. Speicherzugriffe. Da jedoch nicht die genaue Ausführungszeit der Methode gesucht wird, sondern lediglich verifiziert werden muss, dass die Methode einen gewissen zetilichen Rahmen einhält, genügt der obige Ansatz für die Ziele dieser Arbeit völlig.

Im Folgenden Abschnitt sollen zunächst einige unterschiedliche Ansätze zur WCET Analyse einer Methode erläutert werden.

In den Abschnitten 6.3 und 6.4 wird dann die im Rahmen dieser Arbeit entwickelte Vorgehensweise eingehend dargestellt. Dabei beschäftigt sich Abschnitt 6.3 mit der Analyse einer

Methode, in Abschnitt 6.4 werden hingegen solche Methoden behandelt, die nicht analysiert werden können.

6.2 Überblick über verschiedene Ansätze

6.2.1 Analyse von Programmen ohne Schleifen

Nachdem die Laufzeit der einzelnen Befehle als bekannt vorausgesetzt wird, lässt sich die Analyse eines Programmes ohne Schleifen relativ einfach bewerkstelligen: Die Ausführungsdauer einer Sequenz von Befehlen ergibt sich aus der Summe der Einzelzeiten. Anders ausgedrückt ergibt sich die Zeit die benötigt wird, um zum Ende eines bestimmten Befehls zu kommen aus der Ausführungsdauer dieses Befehls, addiert zur Zeit die benötigt wird, um zum Ende des direkten Vorgängers zu kommen.

Hat ein Befehl mehrere mögliche Vorgänger, so ergibt sich die Zeit aus dem Maximum der Zeiten der Vorgänger (da es sich ja um eine worst case Abschätzung handelt), addiert zur eigenen Zeit.

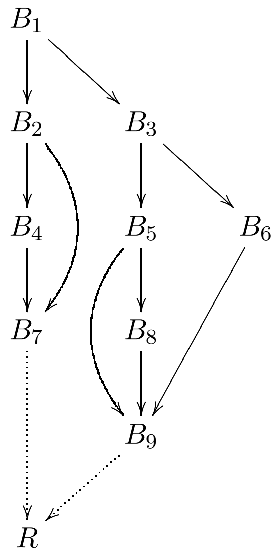


Abbildung 6.1: Kontrollflussgraph eines Beispielprogrammes.

In Abbildung 6.1 ist der KFG eines Beispielprogrammes angegeben, wobei B_1 bis B_9 die Basic Blöcke (siehe Abschnitt 3) des Programmes sind und dieses mit B_7 oder B_9 endet. Bezeichnet man mit $T'(B)$ die Zeit die benötigt wird, um einen Block B zu durchlaufen und mit $T(B)$ die Zeit die benötigt wird um das Programm bis zum Ende von B zu durchlaufen, so gilt:

$$
\begin{array}{rcl}
T(B_1) & = & T'(B_1) \\
T(B_2) & = & T(B_1) + T'(B_2) \\
 & \cdots & \\
T(B_7) & = & \max(T(B_2), T(B_4)) + T'(B_7) \\
 & \cdots & \\
T(B_9) & = & \max(T(B_5), T(B_6), T(B_8)) + T'(B_9)
\end{array}
$$

Die maximale Laufzeit der gesamten Methode ergibt sich dann aus dem Maximum der beiden Endpunkte B_7 und B_9. Im Sinne eines einheitlichen Vorgehens kann man stattdessen auch einen einzigen Endknoten R einführen, der mit allen früheren Endknoten durch Kanten (in der Abbildung gepunktet eingezeichnet) verbunden wird. Die Gesamtlaufzeit ergibt sich dann aus $T(R)$, wobei $T'(R) = 0$ ist. Im Beispiel ergibt sich also $T(R) = \max(T(B_7), T(B_9))$.

Dieses Verfahren erlaubt es also, beliebige Methoden zu analysieren, die keine Schleifen enthalten. Sobald der Kontrollflussgraph jedoch einen Zyklus enthält, erbringt diese „Maximum Regel" keine Lösung mehr, denn dann bestehen rekursive Abhängigkeiten zwischen den einzelnen Termen. Das Hauptproblem der WCET Analyse besteht also darin, für jede Schleife im Programm die maximale Anzahl der Durchläufe zu bestimmen.

6.2.2 Analyse von Programmen mit Schleifen

Verfahren zur Bestimmung der Ausführungsdauer einer Methode werden auch im Bereich der Echtzeitsysteme benötigt, da Echtzeit-Scheduling-Strategien häufig a priori Wissen über die Laufzeit einer Methode bzw. eines Prozesses voraussetzen. Dementsprechend wurden in diesem Zusammenhang verschiedene Verfahren zur WCET Analyse entwickelt, wobei eine Reihe unterschiedlicher Ansätze zur Behandlung von Schleifen verfolgt werden (einen Überblick gibt beispielsweise [Bli95]).

Fast allen Ansätzen gemein ist, dass die Unentscheidbarkeit des Halteproblems umgangen wird, indem die Analyse auf speziell gebaute Methoden eingeschränkt wird, deren Terminierung nachweisbar ist.

Der einfachste Ansatz besteht darin, Schleifen generell zu verbieten und für zeitkritische Anwendungen nur solche Methoden zuzulassen, die lediglich Verzweigungen enthalten. Viele andere Ansätze basieren darauf, nur spezielle Schleifenkonstrukte zuzulassen, wobei es wieder Unterschiede hinsichtlich der Form dieser Schleifen gibt. Eine Möglichkeit ist, die Zahl der Schleifendurchläufe von vorneherein festzulegen. Zu diesem Zweck können beispielsweise spezielle Schleifenkonstrukte eingeführt werden (vgl. [Bli95]). Ein ganz ähnlicher Ansatz lässt zwar beliebige Schleifenkonstrukte zu, verlangt jedoch vom Benutzer eine obere Schranke für die Zahl der Durchläufe anzugeben (siehe [PK89]).

Im letzen Ansatz kommt ein Konzept zum Tragen, das auch in den meisten anderen Verfahren benutzt wird: Die Zahl der Schleifendurchläufe wird ganz oder teilweise durch den Benutzer, bzw. allgemein durch ein sogenanntes „Orakel" (vgl. [Bli00]) bestimmt. Nur in Ausnahmefällen und für sehr „gutmütige" Schleifenfunktionen kann die Zahl der Durchläufe automatisch bestimmt werden.

```java
public int sum(int s) {
  int ret = 0;
  for (int i = 1; i <=s ; i++) {
    ret += s;
  }
  return ret;
}
```

Abbildung 6.2: Java Beispielprogramm.

Einer der Gründe dafür, dass man ohne die Hilfe eines Orakels meist nicht auskommt, ist, dass Schleifen häufig von den – zum Zeitpunkt der Analyse unbekannten – Parametern der Methode abhängen. So kann beispielsweise für das in Abbildung 6.2 angegebene Java Programm keine sinnvolle Abschätzung für die Ausführungsdauer angegeben werden, so lange Parameter `s` unbekannt ist. Wird der Wertebereich für den Parameter hingegen eingegrenzt, so kann eine Abschätzung gegeben werden, indem die Zahl der Schleifendurchläufe mit dem größtmöglichen Wert von `s` gleichgesetzt wird.

Aber auch, wenn konkrete Werte für die Parameter der Schleife angegeben werden, ist die Abschätzung häufig nicht möglich. Schließlich muss zunächst festgestellt werden, wie die Zahl der Schleifendurchläufe von den entsprechenden Parametern abhängt. Ein allgemeines Verfahren zur Lösung dieses Problems scheint jedoch bislang noch nicht zu existieren, so dass man sich wiederum auf die Analyse sehr einfacher Schleifenbedingungen beschränken muss.

Werden für die Parameter statt konkreter Werte nur Wertebereiche angegeben, so ist das Problem noch schwieriger, zumindes so lange der Wertebereich zu groß ist um die Zahl der Durchläufe für jeden Wert einzeln zu berechnen. In diesem Fall kann die Zahl der Schleifendurchläufe nur bestimmt werden, wenn sie einer Funktion entspricht, die monoton in den betreffenden Parametern ist. Die Festellung der Monotonie ist jedoch im Allgemeinen nicht möglich, so dass man auch hier in den meisten Fällen wieder auf ein Orakel oder spezielle Schleifenkonstrukte zurückgreifen muss (vgl. [Bli95]).

Die genannten Ansätze beschränken sich also im wesentlichen auf spezielle Arten von Schleifen, wobei häufig zusätzliche Information benötigt wird, die entweder explizit (durch ein „Orakel”) oder implizit in Form von zusätzlichen Sprachkonstrukten zur Verfügung gestellt wird.

6.2.3 Analyse durch partielle Ausführung

Auf Grund der angeführten Probleme und der Tatsache, dass in dieser Arbeit das Ziel verfolgt wird, normalen, uneingeschränkten Java Bytecode zu verifizieren, wurde eine andere Vorgehensweise gewählt.

Statt die Unentscheidbarkeit des Halteproblems durch Einschränkung der Menge der analysierbaren Funktionen zu umgehen, wird bei diesem Ansatz die Tatsache ausgenutzt, dass das Halteproblem semientscheidbar ist: Wendet man die oben erwähnte Turingmaschine M auf x an, so kann die Terminierung bestätigt werden, so bald M terminiert. So lange M jedoch nicht hält, kann keine Aussage gemacht werden, da nicht sicher ist, ob die Berechnung nur noch nicht beendet wurde oder ob M überhaupt nicht terminiert.

Dementsprechend werden – statt die Analyse auf spezielle Schleifenkonstrukte einzuschränken – Schleifen in diesem Ansatz ausgeführt, um so die Zahl der Durchläufe zu ermitteln. Terminiert diese Ausführung innerhalb der festgelegten Zeit, so hält die Methode ihr Zeitlimit ein. Anderenfalls wurde zwar nicht ermittelt, wie lange die Methode tatsächlich braucht, die vorgegebene Zeit wird jedoch mit Sicherheit nicht eingehalten.

Natürlich muss hierzu die Methode nicht vollständig ausgeführt werden. Stattdessen kann sich die Ausführung auf alle Schleifenkonstrukte sowie diejenigen Befehle, die die Zahl der Schleifendurchläufe beeinflussen, beschränken.

Ein ähnlicher Ansatz wird auch in [NP93] beschrieben. Allerdings wird dort eine speziell für diese Zwecke entwickelte Sprache verwendet, in welcher der Programmierer angibt, welche Anweisungen bereits zur Verifikationszeit ausgewertet werden können. Im Rahmen dieser Arbeit geht es hingegen darum, Programme zu analysieren, die keinerlei spezielle Sprachkonstrukte oder vom Programmierer angegebene Informationen zur partiellen Ausführung enthalten.

Daher muss zunächst festgestellt werden, von welchen Parametern und Befehlen eine Schleifenbedingung abhängt. Sind die benötigten Parameter vorhanden, so kann die Ausführung der Schleife simuliert werden, indem alle Befehle, welche die Schleifenbedingung betreffen, ausgeführt werden. So kann zuverlässig bestimmt werden, wie oft eine Schleife durchlaufen wird und wie viel Zeit dazu benötigt wird. Dabei wird für Verzweigungen weiterhin die oben beschriebene Methode zur Abschätzung der Ausführungsdauer verwendet.

Da die Schleife jedoch möglicherweise nicht terminiert, muss während der Simulation regelmäßig geprüft werden, ob die vorgegebene Zeitschranke bereits überschritten wurde. Ist dies der Fall, muss die Simulation abgebrochen werden. Die Methode darf dann auch nicht ununterbrechbar ausgeführt werden.

Ob eine Methode partiell ausgeführt werden kann, hängt allerdings auch davon ab, ob alle Parameter bekannt sind, die die Zahl der Schleifendurchläufe bestimmen. Ist dies nicht der Fall so kann wiederum auf ein Orakel zurückgegriffen werden, dass entsprechende Parameterwerte angibt.

Ist auch dies nicht möglich – können also die Parameterwerte nicht vorherbestimmt werden – so ist eine Analyse der Laufzeit nicht durchführbar. Dieses Problem ist jedoch nicht auf diesen speziellen Analyseansatz beschränkt; Falls die Parameterwerte, von denen eine Schleife abhängt, zur Verifikationszeit nicht bekannt sind, kommen auch alle anderen Ansätze zu keinem Ergebnis. In diesem Fall kann die Laufzeit generell nicht vorherbestimmt werden.

Ein Nachteil der oben beschriebenen Vorgehensweise besteht allerdings darin, dass es nicht möglich ist, die maximale Laufzeit für einen eingegrenzten Wertebereich von Parametern zu berechnen. Da Schleifen nicht analytisch betrachtet, sondern lediglich partiell ausgeführt werden, kann die Ausführungszeit natürlich immer nur für einen fest vorgegebenen Satz von Parametern berechnet werden.

Dem ist nun allerdings entgegenzuhalten, dass die Berechnung der maximalen Laufzeit für einen Paramteterbereich, wie sie etwa in [Bli95] durchgeführt wird, nur dann möglich ist, wenn die Schleifenbedingung sehr einfach aufgebaut ist. Tatsächlich gelingt eine solche Analyse bislang nur, wenn die Schleifenvariable bei jedem Durchlauf um einen konstanten Betrag erhöht oder verringert, bzw. mit einer Konstante multipliziert wird, wobei der Anfangswert und Endwert der Variablen parametrisiert ist.

6.2.4 Behandlung nicht analysierbarer Methoden

Damit eine nicht analysierbare Methode dennoch ausgeführt werden kann, werden Laufzeitüberprüfungen innerhalb jeder Schleife eingefügt. Diese überprüfen, ob die Zeitschranke der Methode bereits überschritten wurde und lösen gegebenenfalls eine Ausnahme aus. Dadurch wird die – von außen nicht unterbrechbare – Methode beendet und das System wird deblockiert.

Der Nachteil dieses Ansatzes ist natürlich, dass sich in diesem Falle erst zur Laufzeit zeigt, ob das vorgegebene Zeitlimit tatsächlich eingehalten wird oder nicht. Für Echtzeitsysteme ist eine solche Vorgehensweise somit nur bedingt geeignet, da man häufig eher an einer Abschätzung der Laufzeit interessiert ist, als an der Einhaltung einer vorgegebenen Zeitbeschränkung. Im Rahmen der in dieser Arbeit verfolgen Ziele erscheint dies allerdings weitaus weniger problematisch, da es beispielsweise durchaus möglich ist, eine sinnvolle obere Zeitschranke für die Interruptbehandlung anzugeben.

Der Vorteil des oben vorgestellten Ansatzes ist, dass beliebige Methoden „verifiziert" werden können, ohne dass spezielle Einschränkungen erfüllt sein müssen. Außerdem ist ein Orakel nicht unbedingt notwendig, da notfalls die Methode mit Laufzeitüberprüfung ausgeführt wird. Stehen alle Parameterwerte zur Verfügung oder sind alle Schleifen der Methode von Parametern unabhängig, so kann die Laufzeit für ansonsten beliebig komplizierte Schleifenkonstrukte berechnet werden. Vor allem ist dies ohne spezielle programmiersprachliche Konstrukte oder eine weitere Hilfe des Orakels möglich.

Es tritt noch ein weiteres Problem bei Methoden auf, deren Laufzeit nicht berechnet werden kann. Wie oben erläutert, werden in diesem Fall Laufzeitüberprüfungen eingefügt, die eine Ausnahme auslösen, falls die vorgegebene Zeitschranke überschritten wurde. Diese führt zu einem Abbruch der Methode.

Wird die Methode jedoch frühzeitig abgebrochen, so hinterläßt sie mit großer Wahrscheinlichkeit inkonsistente Daten, so dass der zugehörige Treiber, bzw. allgemein die zugehörige Anwendung kaum fehlerfrei weiterarbeiten kann. Dieses Problem kann auf zwei Arten umgangen werden: Zum einen besteht die Möglichkeit, dass die Anwendung eine spezielle „Aufräummethode" zur Verfügung stellt, die nach dem Abbruch der Methode wieder einen konsistenten Zustand herstellt (wobei diese unterbrechbar ausgeführt werden kann). Allerdings ist diese Vorgehensweise insofern nur bedingt sinnvoll, als die Methode abgebrochen wurde, weil ein gegebenes Zeitlimit überschritten wurde, d. h. die Methode in dieser Hinsicht fehlerhaft ist. Wird der Grund für die Überschreitung des Zeitlimits beim Aufräumen nicht beseitigt, so wird die Methode bei ihrem nächsten Aufruf erneut beendet und es muss wiederum ein konsistenter Zustand hergestellt werden.

Die Alternative besteht darin, den Treiber oder die Anwendung komplett zu beenden. Obwohl dieses Vorgehen zunächst etwas radikal anmuten mag, ist es doch insofern gerechtfertigt, als das Abbrechen einer zeitkritischen Methode nicht der Normalfall sein sollte, sondern vielmehr eine Art „Notbremse". Schließlich ist der Mechanismus dazu bestimmt, gewisse Mindestanforderungen an die Antwortzeit des Gesamtsystems sicherzustellen und ein Blockieren des Systems durch defekte Treiber zu verhindern. Er ist nicht dazu gedacht, eine Methode in mehrere Zeitscheiben aufzuteilen oder die Einhaltung des Zeitlimits für beliebige Methoden zu ermöglichen; es bleibt weiterhin die Aufgabe des Programmierers die Methoden so zu gestalten, dass die vorgegebene Zeit eingehalten wird.

6.3 Vorgehensweise

In diesem Abschnitt soll das oben bereits skizzierte Verfahren zur Laufzeitabschätzung genauer erläutert werden. Dabei gilt es zunächst ein Problem zu bewältigen, das durch die maschinennahe Struktur des Java Bytecodes entsteht: Wie bereits beschrieben, werden Verzweigungen und Schleifen auf Bytecodeebene nicht unterschieden, sondern es existieren lediglich eine Reihe von bedingten Sprungbefehlen, so dass zunächst festgestellt werden muss, welcher der bedingten Sprünge in eine Schleife führt.

Anschließend muss analysiert werden, von welchen Daten die in Schleifen führenden Sprungbefehle abhängen und welche Befehle ausgeführt werden müssen, um diese zu berechnen. Dabei muss außerdem festgestellt werden, von welchen Parametern die Schleifenbedingungen abhängen und ob das „Orakel" entsprechende Parameterwerte zur Vefügung stellt. Ist dies nicht der Fall, so muss die Verifikation an dieser Stelle abgebrochen werden und eine entsprechende Methode mit Laufzeitüberprüfungen muss erstellt werden.

Sind alle für die Simulation nötigen Befehle und Anfangswerte ermittelt, so kann die Methode partiell ausgeführt und die Laufzeit der Methode bestimmt werden. Die einzelnen Schritte sind in den folgenden Abschnitten ausführlich beschrieben.

6.3.1 Auffinden von Schleifen

6.3.1.1 Zielsetzung

Das Ziel des im Folgenden beschriebenen Analyseschrittes ist es, festzustellen, welche Sprungbefehle ausgeführt werden müssen, da sie in eine Schleife führen und welche mit Hilfe der bereits beschriebenen Maximum Regel analysiert werden können. Als Ergebnis der Analyse sollte jeder auszuführende Sprungbefehl mit einer entsprechenden Markierung versehen sein.

Obwohl dieses Problem zunächst relativ einfach erscheinen mag, ist es doch äußerst komplex, vor allem, wenn man den zu analysierenden Bytecode keinen Einschränkungen unterwerfen will (mit Ausnahme der in der JVM-Spezifikation genannten). In diesem Fall ist nicht gewährleistet, dass der Code einer Methode wohlstrukturiert ist, wie das bei einem aus Java Quellcode erzeugten Programm der Fall ist. In Abbildung 6.3 (e) ist beispielhaft der Kontrollflussgraph einer Methode dargestellt, die durchaus zulässigen Java Bytecode darstellt, jedoch von einem Java Compiler so nicht erzeugt wird, da verschachtelte Schleifen in Java nicht formuliert werden können.

Im Laufe dieser Arbeit wurde eine Vielzahl unterschiedlicher Ansätze entwickelt und wieder verworfen, da die Ergebnisse unvollständig (d. h. Sprünge in eine Schleife sind nicht markiert) oder ungenau (d. h. auch bedingte Sprünge, die nicht ausgeführt werden müssen sind markiert) waren. Statt auf die unterschiedlichen Ansätze im Einzelnen einzugehen, soll lediglich eine Reihe von „Problemfällen" dargestellt werden, um einen Eindruck der Schwierigkeiten zu vermitteln, bevor schließlich die gewählte Vorgehensweise vorgestellt wird.

In den Abbildungen 6.3 (a) - (d) sind die Kontrollflussgraphen verschiedener Programmfragmente dargestellt, die bei der Analyse gewisse Probleme bereiten. Die Blöcke sind mit Großbuchstaben bezeichnet, wobei die Ausführung der dargestellten Programmstücke stets mit

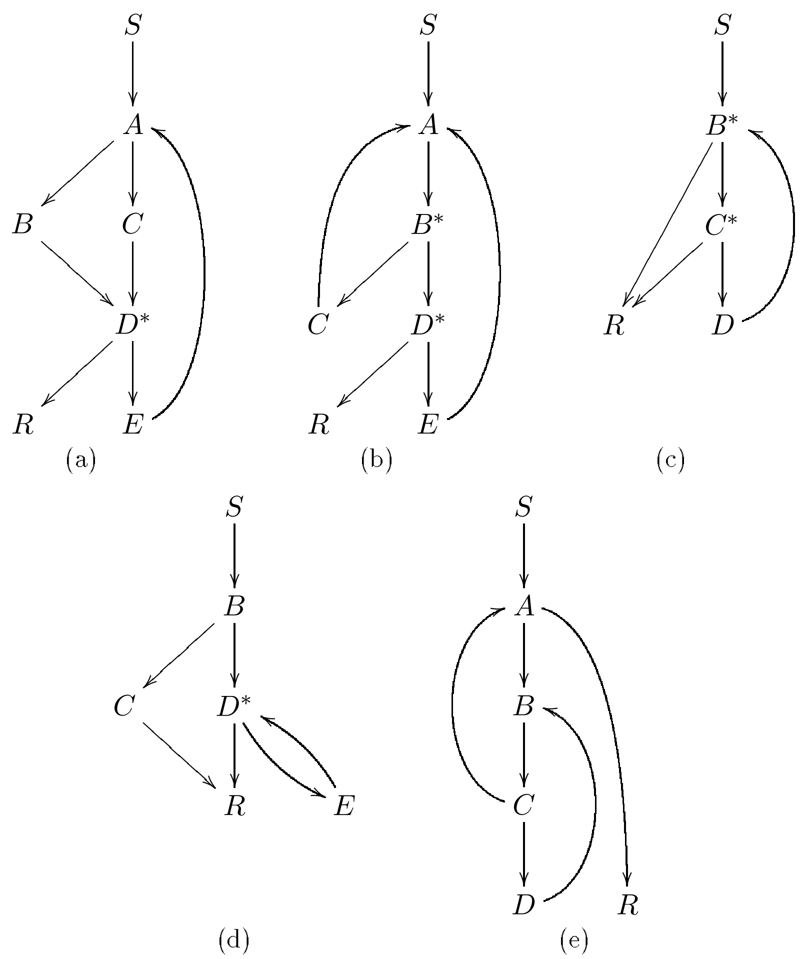


Abbildung 6.3: Kontrollflussgraphen verschiedener „Problemfälle".

Block S beginnt und mit R beendet wird. Des Weiteren sind alle Knoten, die durch die Analyse markiert werden sollten durch * gekennzeichnet.

Unter (a) wird eine Schleife gezeigt, die eine Verzweigung enthält. Das Problem bei dieser Konstruktion ist, dass der KFG zwei Zyklen enthält: A, C, D, E, sowie A, B, D, E, die beide die bedingten Sprünge A und E enthalten. Somit kann eine korrekte Markierung nicht dadurch erreicht werden, dass einfach alle auf einem Zyklus liegenden Sprünge markiert werden, da dann auch der Sprung am Ende von Block A markiert werden würde.

Auch wenn man keine Verzweigungen markiert, bei denen *beide* Ziele in Zyklen führen, wird zwar Fall (a) korrekt behandelt, in Beispiel (b) hingegen würde auch dieses Verfahren scheitern, da Knoten B nicht markiert wird.

In Abbildung 6.3 (c) ist eine Schleife mit zwei Sprüngen, die aus der Schleife führen, gezeigt. Entsprechender Code kann erzeugt werden, wenn die Schleifenbedingung in Java aus zwei durch logisches und (`&&`) verknüpften Einzelbedingungen besteht: Zunächst wird die erste Teilbedingung geprüft. Ist diese falsch, so wird die Schleife beendet. Anderenfalls wird die zweite Bedingung überprüft. Bei der Suche nach Schleifen muss also auch berücksichtigt werden, dass eine Schleife aus mehreren bedingten Sprüngen aufgebaut sein kann.

Abschließend ist unter (d) eine Schleife dargestellt, die Teil einer Verzweigung ist. Das Problem hierbei ist, zu erkennen, dass lediglich der bedingte Sprung am Ende von Block D ausgeführt werden muss. Der Sprung aus Block B kann mit der eingangs eingeführten Maximum-Regel behandelt werden, indem die Laufzeit des rechten Zweiges (berrechnet mit Hilfe der Ausführung der Schleife) mit der des linken verglichen wird.

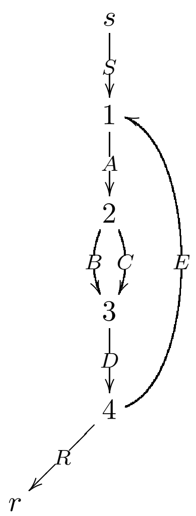
Nachdem nun einige besonders schwierige Konstrukte exemplarisch dargestellt wurden, soll nun ein Verfahren vorgestellt werden, das es erlaubt, Schleifen aufzufinden und zu markieren. Zunächst wird jedoch ein modifizierter Kontrollflussgraph vorgestellt, der dem Algorithmus zu Grunde liegt.

6.3.1.2 Modifizierter Kontrollflussgraph

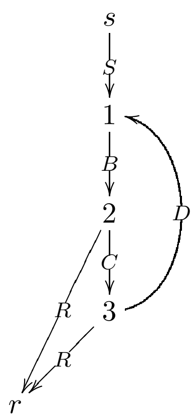
Ein modifizierter Kontrollflussgraph (mKFG) ist in gewisser Weise das Gegenstück zum normalen Kontrollflussgraph, da nicht jedem Knoten, sondern jeder Kante ein Basic Block des betreffenden Programmes zugeordnet wird. Dabei sind bedingte Sprünge nicht mehr in den Blöcken enthalten. Diese werden stattdessen Knoten zugeordnet.

Neben den Knoten, denen ein bedingter Sprung zugeordnet wird, existieren auch Knoten denen kein Befehl zugeordnet wird, sondern die lediglich eine Stelle im Bytecode bezeichnen, die durch Ausführung zweier unterschiedlicher Blöcke erreicht werden kann. Knoten, denen ein Sprungbefehl zugeordnet ist werden im Folgenden als „if-Knoten", alle anderen als „normale Knoten" bezeichnet.

Es werden wiederum dedizierte Startknoten s und Endknoten r eingeführt, wobei der Knoten s der Adresse 0 entspricht. Der Knoten r ist Ziel aller Kanten die mit einem Basic Block bezeichnet sind, der das Programm beendet.



(a)



(c)

Abbildung 6.4: modifizierter Kontrollflussgraph der Beispiele (a) und (c) aus Abb. 6.3.

In Abbildung 6.4 ist der mKFG zu den in Abbildung 6.3 (a) und (c) dargestellten Kontrollflussgraphen zu sehen. Die Kanten sind mit dem jeweiligen Basic Block beschriftet, die Knoten wurden der besseren Übersicht halber durchnummeriert.

Bei Knoten 2 und 4 in (a) bzw. 2 und 3 in (b) handelt es sich um if-Knoten. Wie man sieht, ist die Zuordnung zwischen Kanten und Blöcken nicht umkehrbar, da verschieden Kanten den selben Block enthalten können, wie etwa die beiden mit R beschrifteten Kanten in Beispiel (c).

6.3.1.3 Vereinfachung des Graphen durch Eliminationsregeln

Der im Rahmen dieser Arbeit entwickelte Algorithmus zur Markierung von Schleifensprüngen basiert nun darauf, den mKFG durch Anwendung verschiedener Regeln schrittweise zu vereinfachen. Dabei werden bestimmte Teilgraphen, für die leicht erkennbar ist, ob es sich um Schleifen oder Verzweigungen handelt, durch einfache Kanten oder Knoten ersetzt, so lange bis der Graph auf eine einzige Kante vom Anfangsknoten s zum Endknoten r reduziert ist. Dabei wird bei der Regelanwendung für jede neue Kante und jeden neuen Knoten vermerkt, welche Kanten ersetzt wurden. Auf diese Information wird dann in den weiteren Schritten der Analyse zurückgegriffen.

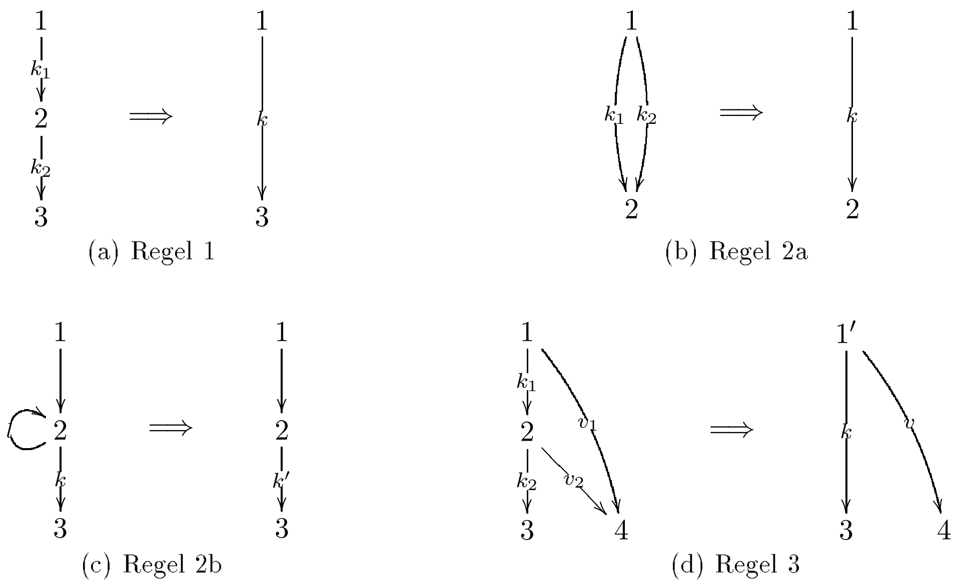


Abbildung 6.5: Graphische Darstellung der Eliminationsregeln.

Bei der Anwendung der Regeln wird vorausgesetzt, dass alle Knoten des mKFG höchstens zwei ausgehende Kanten haben, d. h. dass die Befehle `tableswitch` und `lookupswitch` sowie Ausnahmebehandlungen nicht vorkommen. Wie diese behandelt werden, wird weiter unten noch ausführlich erläutert. Die einzelnen Regeln sind in Abbildung 6.5 dargestellt:

Regel 1 (Sequenz): Diese Regel dient dazu, Knoten zu eliminieren, die nur einen Ausgang haben. Falls also ein Knoten nur einen Ausgang hat (in Bild (a) Knoten 2), so kann er eliminiert werden. Jede Kante k_1 von einem Vorgängerknoten 1 nach 2 wird mit der von 2 ausgehenden Kante k_2 zu einer neuen Kante k verknüpft. Diese enthält Verweise auf die ersetzten Kanten k_1 und k_2.
Die Regel kann auch angewendet werden, wenn ein Knoten mehrere eingehende Kanten besitzt – wichtig ist nur, dass nur eine ausgehende Kante existiert.

Regel 2a (Verzweigung): Einfache Verzweigungen, die sofort wieder zusammenführen, können zu einer Kante verschmolzen werden. Das heißt, dass zwei Kanten k_1 und k_2 zu einer neuen Kante k verschmolzen werden können, falls beide die gleichen Anfangs- und Endknoten (1 und 2) haben. Die neue Kante k enthält Verweise auf k_1 und k_2.

Regel 2b (Schleife): Einfache Schleifen können elimiert werden, wobei der zugehörige bedingte Sprung markiert wird. Existiert eine Kante l die im Knoten 2 sowohl beginnt als auch endet, so kann diese aus dem Graphen entfernt werden. Außerdem wird Knoten 2 für die spätere Ausführung markiert. Zusätzlich wird jede Kante k, die von 2 ausgeht durch eine neue Kante k' ersetzt. Diese neue Kante enhält einen Verweis auf die Kanten l und k.

Regel 3 (Vereinigung von Verzweigungen): Diese Regel dient dazu, zwei bedingte Sprünge, die das gleiche Sprungziel haben, zusammenzufassen. Sind also zwei if-Knoten 1 und 2 durch eine Kante k_1 verbunden, und führt außerdem von beiden Knoten je eine Kante v_1 bzw. v_2 zu einem dritten Knoten 4, so können 1 und 2 durch einen neuen Knoten $1'$ ersetzt werden. Dieser Knoten enthält Verweise auf die Kanten k_1 und k_2 sowie v_1 und v_2. Zusätzlich werden neue Kanten k von $1'$ nach 3 und v von $1'$ nach 4 eingeführt. Beide Kanten dienen lediglich dazu, eine Verbindung zwischen den Knoten herzustellen und enthalten keinerlei Verweise. Außerdem werden die Knoten 1 und 2 markiert.

Die Überlegung, die den Regeln zu Grunde liegt, ist, dass jedes Programm aus Schleifen und Verzweigungen aufgebaut ist, deren Schleifenrumpf bzw. Zweige selbst wiederum aus solchen bestehen können. Irgendwo muss diese Rekursion natürlich damit enden, dass ein Schleifenrumpf oder ein Zweig lediglich aus einer Befehlssequenz, d. h. aus einem Basic Block oder einer Folge von Blöcken aufgebaut ist. Durch Anwendung der Regeln werden solche „Elementarbausteine" eliminiert, so dass die übergeordneten Konstrukte ebenfalls zu solch einfachen Schleifen oder Verzweigungen werden, so lange bis der gesamte mKFG zu einer einzigen Kante vereinfacht wurde.

Während Regel 1 dazu dient „unnötige" Knoten zu eliminieren, die evtl. die Anwendung der anderen Regeln blockieren, aber keinerlei Wahlmöglichkeiten bzgl. des Kontrollflusses bieten, werden in Regel 2a und b Schleifen bzw. Verzweigungen vereinfacht. Wie man sieht, können diese Regeln nur auf solche Teilgraphen angewandt werden, die die einfachste mögliche Verzweigung bzw. Schleife darstellen, in der Erwartung, dass der Schleifenrumpf bzw. die Zweige komplexerer Verzweigungen durch wiederholte Regelanwendungen irgendwann zu einfachen Kanten reduziert werden.

Die dritte Regel hingegen dient der Behandlung von Schleifen, die mehrere „Ausgänge" haben, d. h. die zwei Knoten enthalten, an denen entschieden wird, ob die Schleife weiterdurchlaufen oder das Programm fortgesetzt wird. In der Abbildung entsprechen die Knoten 1 und 2 den „Ausgängen" und 3 ist der erste Knoten des Schleifenrumpfes.

Wie man sieht, wird angenommen, dass beide Ausgänge zum selben Knoten 4 führen. Dies ist zwar zum einen der Fall wenn, wie oben beschrieben, eine Bedingung in zwei getrennte Sprünge übersetzt wird, andererseits können die Ausgänge zunächst durchaus auf unterschiedliche Teilgraphen verweisen. Werden diese jedoch durch die Anwendung der verschiedenen Regeln schließlich auf simple Kanten reduziert, so zeigen beide Kanten auf den selben Knoten, nämlich den einheitlichen Endknoten r.

Der Algorithmus selbst geht so vor, dass zunächst nach Teilgraphen gesucht wird, die der linken Seite einer Regel entsprechen. Wird ein solcher gefunden, so wird die Regel angewandt und der Teilgraph durch eine neue Kante ersetzt. Dabei wird zunächst versucht, die erste Regel anzuwenden. Ist das nicht mehr möglich, so kommen Regeln 2a und 2b zur Anwendung. Erst wenn auch diese nicht mehr benutzt werden können, wird die dritte Regel eingesetzt.

Dabei muss nach jeder Anwendung der Regeln 2a oder b erneut Regel 1 angewandt werden, so lange, bis sie nicht mehr anwendbar ist. Ebenso darf Regel 3 nach einer Anwendung erst dann erneut angewandt werden, wenn weder Regel 1 noch eine der Regeln 2a oder b anwendbar ist. Dadurch wird sichergestellt, dass Regel 3 nicht auf Graphen angewendet wird, die auch mit Hilfe der anderen Regeln vereinfacht werden könnten.

Der Algorithmus wird so lange fortgesetzt, bis der Graph auf eine einzige Kante zwischen Start- und Endknoten reduziert wurde.

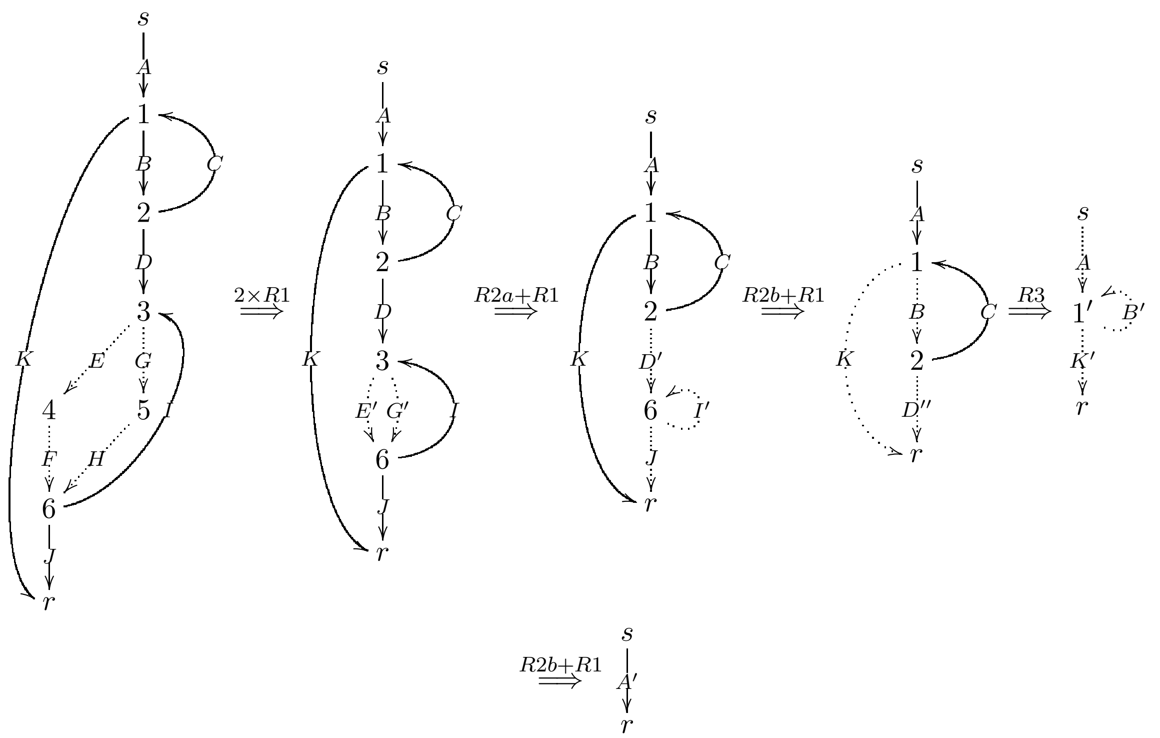


Abbildung 6.6: Vereinfachung eines Beispielprogrammes.

In Abbildung 6.6 ist die Reduzierung eines mKFG mittels Regelanwendung beispielhaft dargestellt. Dabei sind die Kanten, die im nächsten Schritt eliminiert werden gepunktet dargestellt.

In Schritt eins werden die Knoten 4 und 5 mit Hilfe von Regel 1 eliminiert. Dadurch wird Knoten 3 zu einer einfachen Verzweigung, die im nächsten Schritt mit Regel 2a ebenfalls eliminiert werden kann. Auf die resultierende Kante wird wiederum Regel 1 angewendet, so dass diese mit den Kanten D und I vereinigt wird zu D' und I'. Dadurch stellt I' eine einfache Schleife dar, die mit Regel 2b eliminiert wird, wobei auf die resultierende Kante wiederum Regel 1 angewandt wird. Das Ergebnis bildet Kante D''. In diesem Schritt wurde außerdem die zu Knoten 6 gehörende Verzweigung markiert.

Da nun die Kanten K und D'' den selben Zielknoten haben kann Regel 3 zur Elimination von B, D'' und K sowie Knoten 1 und 2 angewandt werden. Dadurch werden Knoten 1 und 2 markiert. Der resultierende Graph enthält, neben Start- und Endknoten, nur noch einen weiteren Knoten mit einer einfachen Schleife. Durch Anwendung der Regel 1 auf das Resultat der Schleifenelimination durch Regel 2b wird der Graph schließlich auf eine einzige Kante reduziert.

Somit wurden drei bedingte Sprünge für die Ausführung markiert: 1, 2 und 6; Knoten 3 hingegen wurde korrekt als einfache Verzweigung erkannt.

Mit der in diesem Schritt erstellten Markierung der auszuführenden Sprungbefehle wird nun in den nächsten Abschnitten weitergearbeitet. Dabei wird für alle Kanten und jeden Knoten auch die Information benötigt, ob sie bereits im ursprünglichen mKFG vorhanden waren, oder ob sie durch Anwendung einer Regel entstanden sind. Im letzteren Fall muss auch vermerkt werden, aus welchen Kanten und Knoten sie entstanden sind.

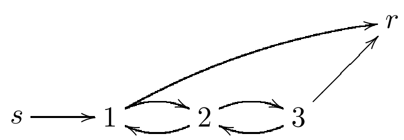


Abbildung 6.7: Nicht reduzierbarer modifizierter Kontrollflussgraph

Es bleibt jedoch die Frage, welche Graphen mit den genannten Regeln vollständig reduziert werden können. Es existieren auf jeden Fall Graphen, die nicht reduziert werden können. Ein Beispiel ist in Abbildung 6.7 dargestellt. Obwohl ein entsprechender Bytecode durchaus möglich und zulässig ist, wird er jedoch kaum von einem Java-Compiler erzeugt werden. Schließlich sind zwei Schleifen in Java stets disjunkt oder eine ist in der anderen vollständig enthalten. Beides ist im Beispiel nicht der Fall.

Insgesamt wurden die Regeln so konzipiert, dass man annehmen kann, dass wohlstrukturierter Code, wie er von einem Java-Compiler erzeugt werden sollte, verarbeitet werden kann. Diese Annahme wurde auch durch Tests bestätigt.

Kann der mKFG einer Methode jedoch nicht aufgelöst werden, so bleibt immer noch die Möglichkeit eine modifizierte Methode zu erstellen, welche die Einhaltung der Zeitbeschränkung zur Laufzeit überprüft (siehe Abschnitt 6.4).

Behandlung von Befehlen mit mehreren Zielen

Wie bereits erwähnt, wird bei der Vereinfachung des mKFGs davon ausgegangen, dass alle Knoten höchstens zwei ausgehende Kanten besitzen. Dadurch können die Befehle `lookupswitch` und `tableswitch` nicht ohne weiteres dargestellt werden. Diese Befehle wählen an Hand eines Wertes auf dem Stack eines von beliebig vielen Sprungzielen aus, ein entsprechender Knoten hätte demnach mehrere Ausgangskanten.

Deshalb werden diese Befehle nicht einem einzigen Knoten zugeordnet, sondern durch einen Knoten pro möglichem Sprungziel dargestellt. Von jedem dieser Knoten führt eine Kante zu einem der Sprungziele des Befehls und die zweite zu seinem nächsten Knoten. Vom letzten Knoten führt wiederum eine Kante zum Ziel, die andere enthält den nachfolgenden Basic Block.

Den Kanten, welche die verschiedenen Knoten des Befehls verbinden, werden keine Befehle zugeordnet; sie dienen lediglich dazu, die Beziehung zwischen den Knoten herzustellen.

Behandlung von Ausnahmen und Subroutinen

Wird ein Bereich des Bytecode durch eine Ausnahmebehandlung geschützt, kann die Kontrolle von denjenigen Befehlen, die eine solche Ausnahme aulösen können, zur Behandlungsroutine übergehen. Dementsprechend müsste auch jeder dieser Befehle ähnlich wie eine Verzweigung behandelt werden, und einem eigenen Knoten zugeordnet werden. Von diesem geht eine Kante aus, die den nachfolgenden Basic Block enthält sowie eine weitere, die zur Ausnahmebehandlungsroutine führt.

Dadurch würden jedoch – je nach Ausnahmetyp – sehr viele neue Knoten eingefügt. Da man jedoch ohnehin an einer *worst-case* Abschätzung der Laufzeit interessiert ist, genügt es, einen solchen Knoten nur nach dem letzten betroffenen Befehl in jedem Basic Block einzuführen. Denn die Ausführungsdauer wird insgesamt am längsten, wenn die Ausnahme erst beim letzten Befehl ausgelöst wird.

In der im Rahmen dieser Arbeit erstellten Implementierung wird jedoch darauf verzichtet, zu jedem Befehl festzustellen ob er eine bestimmte Ausnahme auslösen kann oder nicht. Stattdessen wird einfach nach jedem Basic Block eine Kante zu jeder Ausnahmebehandlung eingefügt, die den Block betrifft.

In jedem Fall können jedoch Knoten entstehen, die mehr als nur eine Ausgangskante besitzen. Diese Knoten werden wie die oben beschriebenen Sprungbefehle mit mehreren Zielen behandelt.

Bei der Behandlung von Ausnahmen muss noch folgendes beachtet werden: Wird bei der Vereinfachung des Graphen ein Knoten markiert, von dem eine Kante zu einer Ausnahmebehandlung ausgeht, so muss die Analyse der gesamten Methode abgebrochen werden. Schließlich kann das Auftreten einer Ausnahme im Allgemeinen nicht simuliert werden, so dass die partielle Ausführung der Methode nicht möglich ist.

Die Behandlung von Subroutinen im Rahmen der WCET Analyse fällt relativ einfach aus: Jeder `jsr` Befehl wird als unbedingter Sprung zur Subroutine aufgefasst. Analog wird die dazugehörige `ret` Instruktion wie ein unbedingter Sprung zu dem Befehl der auf den Subroutinen Aufruf folgt behandelt.

Dadurch ist zwar der Code der Subroutine Teil unterschiedlicher Kanten im Graph, dies hat jedoch keinerlei ungewollte Auswirkungen auf das Ergebnis der Analyse.

6.3.2 Markierung der auszuführenden Befehle

Bevor die Ausführung der Schleifen einer Methode simuliert werden kann, muss zunächst bestimmt werden, von welchen Befehlen die Schleifenbedingungen abhängen. Ausgehend von den im ersten Schritt markierten Schleifen wird mit Hilfe einer Rückwärts-DFA bestimmt, welche Befehle ausgeführt werden müssen, damit eine Schleifenbedingung ausgewertet werden kann.

Dazu werden zunächst die Operanden der bereits markierten Sprungbefehle markiert. Daraufhin werden die Befehle, deren Ergebnisse markiert sind (d. h. diese Ergebnisse sind Operanden der markierten Befehle) ebenfalls markiert. Anschließend werden wiederum die Operanden der neu markierten Befehle markiert. Dieser Vorgang wird so lange fortgesetzt, bis alle benötigten Befehle markiert wurden und jedes markierte Datum entweder das Ergebnis eines markiertn Befehls oder ein Parameter der Methode ist. Im Folgenden wird die eingesetzte DFA ausführlich beschrieben.

Zustandsraum

Der Zustandsraum dieser „Daten Bedarfsanalyse" ist relativ einfach aufgebaut und besteht aus nur zwei möglichen Eigenschaften für die einzelnen Datenwörter: $\perp$ und $\top$. Mit $\perp$ werden Daten bezeichnet, die noch nicht näher analysiert wurden oder die für die Ausführung nicht benötigt werden und $\top$ kennzeichnet entsprechend alle Daten die berechnet werden müssen.

Wie bereits unter 3.2 beschrieben, kann auf Grund der Struktur der JVM eine Rückwärtsanalyse nur dann durchgeführt werden, wenn vorher mit Hilfe einer Vorwärtsanalyse die Größe des Stapels für jeden „Anfangszustand" der Rückwärtsanalyse bestimmt wurde. Allen Datenwörtern der im Rahmen dieser vorbereitenden Vorwärtsanalyse erstellten Zustände wird dabei die Eigenschaft $\perp$ zugewiesen.

Wie üblich, wird auf dem Eigenschaftsraum eine Halbordnung $\sqsubseteq$ und ein vollständiges Gitter definiert, wobei $\perp \sqsubseteq \top$ gilt. Für die Vereinigungsoperation $\sqcup$ gilt entsprechend $\perp \sqcup \top = \top$. Das vollständige Gitter $(\{\perp, \top\}, \sqsubseteq, \sqcup, \perp, \top)$ erfüllt alle Terminierungsbedingungen.

In den Anfangszuständen für die Rückwärtsanalyse (die Zustände der markierten Sprünge) werden die Operandenwörter auf $\top$ gesetzt, da diese auf jeden Fall zur Ausführung benötigt werden.

Transferfunktionen

Da sowohl eine Vorwärts- als auch eine Rückwärtsanalyse durchgeführt werden muss, werden zu jedem Befehl zwei Transferfunktionen benötigt.

Diejenige für die Vorwärtsanalyse entfernt lediglich die Operandenwörter des Befehls vom Stapel und legt für jedes Ergebnis ein Wort auf dem Stapel ab. Dabei sind alle Wörter vom Typ $\perp$. Dass diese Funktion monoton bzgl. $\sqsubseteq$ ist, folgt sofort daraus, dass allen Datenwörtern die gleiche Eigenschaft zugeordnet wird.

Die Transferfunktionen für die Rückwärtsanalyse sind ähnlich aufgebaut, allerdings muss jetzt – wegen der umgekehrten Analyserichtung – eine der Anzahl der Ergebniswörtern entsprechende Zahl von Datenwörtern vom Stapel *entfernt* werden, und stattdessen werden die Operanden *abgelegt*. Des Weiteren wird allen Operanden die Eigenschaft $\top$ zugeordnet, falls einem der Ergebniswörter im „Vorgängerzustand" diese Eigenschaft bereits zugeordnet war.

Allerdings wird diese Zuordnung von $\top$ zu den Operanden nicht bei allen Befehlen vorgenommen, da die Ausführung gewisser Befehle nicht möglich ist, so dass auch die Operanden nicht benötigt werden. Dies ist bei allen Befehlen der Fall, die eine Referenz als Operanden haben, also `getfield` und `putfield` sowie alle `Taload` Befehle (wobei `T` wiederum für einen

Typbezeichner steht). Auch `getstatic` und `putstatic` können nicht ausgeführt werden, da der gelesene Wert nicht bestimmt werden kann und vorher geschriebene Werte durch evtl. vorhandene andere Threads geändert werden können. Auch die Ausführung von Methoden wird nicht unterstützt (obwohl das bei einigen Methoden durchaus möglich wäre, siehe Abschnitt 6.3.3.3).

Zusätzlich muss noch folgendes beachtet werden: Es kann vorkommen, dass zur Berechnung einer notwendigen Operation ein bestimmtes Stapelwort oder eine lokale Variable benötigt wird, denen vorher in zwei unterschiedlichen Zweigen einer if-Anweisung unterschiedliche Werte zugewiesen wurden. In diesem Fall muss auch die betreffende if-Anweisung ausgeführt werden, um schließlich den korrekten Wert zu erhalten.

Allgemein muss eine if-Anweisung dann markiert werden, wenn in einem der Zweige ein Wert erzeugt, aber nicht verwendet wird. Um festzustellen, ob eine if-Anweisung ausgeführt, d. h. markiert werden muss, wird der oben beschriebene vereinfachte mKFG verwendet.

Dazu wird zunächst zu jedem Wert vermerkt, an welcher Stelle er benötigt wird. Wenn der Wert in demselben Basic Block verwendet wird, in dem er erzeugt wird, so muss keine weitere Markierung erfolgen. Ansonsten werden die Kanten, denen der Basic Block im ursprünglichen mKFG zugeordnet waren, untersucht. Dabei kommt es darauf an, ob und durch welche Regel eine solche Kante eliminiert wurde:

Regel 1: Die Kante wurde zusammen mit einer weiteren Kante zu einer neuen zusammengefasst. Wird der Wert in dieser übergeordneten Kante verwendet, so muss kein weiterer Knoten markiert werden.

Regel 2a: Die Kante ist ein Teil einer Verzweigung. In diesem Fall muss der zugehörige Knoten markiert werden, da ein Wert in einem der Zweige erzeugt, aber nicht verwendet wird.

Regel 2b: Ist die Kante, in welcher der Wert erzeugt wird Teil einer Kante, die durch Anwendung von Regel 2b entstanden ist, so muss geprüft werden, ob der Wert innerhalb dieser „Regel 2b Kante" verwendet wird. Wenn ja, kann die Suche beendet werden (der zugehörige if-Knoten ist ohnehin markiert, da er in eine Schleife führt).

Regel 3: Die Kante ist Teil eines durch Regel 3 erzeugten Knotens. Wird der Wert in einer der Kanten oder Knoten die in diesem übergeordneten Knoten enthalten sind verwendet, so wird die Suche beendet.

Dabei muss die Suche so lange mit der übergeordneten Instanz fortgesetzt werden, bis die Kante oder der Knoten, in der der Wert verwendet wird, gefunden wurde.

Um festzustellen, ob eine Kante (ein Knoten) einen Wert verwendet, muss geprüft werden, ob die Adresse der Operation, die den Wert verwendet, innerhalb dieser Kante (dieses Knotens) liegt. Dazu wird die Anfrage an alle zugeordneten Kanten und Knoten weitergereicht. Schließlich werden Kanten erreicht, denen ein Basic Block bzw. Knoten denen ein einziger Sprungbefehl zugeordnet ist, so dass leicht bestimmt werden kann, ob die Adresse enthalten ist.

Aus den in diesem Analyseschritt bestimmten Zuständen kann nun bestimmt werden, welche Befehle ausgeführt werden müssen. Dabei handelt es sich um alle Befehle, deren Ergebnis die

Eigenschaft $\top$ hat. Wie oben bereits erwähnt existieren verschieden Befehle, die nicht wirklich ausgeführt werden können. Wird dennoch das Ergebnis eines solchen Befehls benötigt, so kann das im nächsten Abschnitt beschriebene „Orakel" herangezogen werden, um den Wert des Ergebnisses trotzdem zu bestimmen.

6.3.3 Partielle Ausführung

Nachdem in den vorhergehenden Schritten festgestellt wurde, welche Befehle ausgeführt werden müssen, um die Zahl der Schleifendurchläufe zu bestimmen, kann schließlich die Ausführung selbst durchgeführt werden. Dabei werden einerseits die im vorherigen Schritt markierten Befehle ausgeführt, andererseits wird die Zeit bestimmt, die zur Ausführung aller Befehle benötigt wird.

Bevor jedoch auf die Einzelheiten dieser partiellen Ausführung näher eingegangen wird, soll zunächst das Konzept des Orakels erneut betrachtet werden. Wie bereits erwähnt, ist es die Aufgabe eines solchen Orakels, alle benötigten Informationen zur Verfügung zu stellen, die zur Verifikationszeit nicht automatisch bestimmt werden können.

Anschließend wird der zur partiellen Ausführung verwendete JVM Zustand näher beschrieben, sowie die Ausführung der einzelnen Bytecode Befehle. Schließlich wird die Bestimmung der Ausführungsdauer der gesamten Methode mit Hilfe des im ersten Schritt erzeugten Graphen erläutert.

6.3.3.1 Orakel

Bei den zur Verifikationszeit fehlenden Daten kann es sich zum einen um fehlende Werte von Parametern oder Ergebnissen nicht ausführbarer Befehle handeln. Zum anderen kann aber auch die Laufzeit ganzer Methoden, bei denen die WCET Analyse versagt, vom Orakel zur Verfügung gestellt werden. Woher diese Werte kommen ist dabei irrelevant – es kann sich einerseits um Werte handeln, die einmal (evtl. von Hand) bestimmt wurden und in einer Art Bibliothek verwaltet werden. Andererseits können die Werte auch während der Verifikation vom Benutzer interaktiv angegeben werden.

Allerdings gilt es dabei zu unterscheiden, ob diese Werte aus einer vertrauenswürdigen Quelle stammen oder nicht. Im ersten Fall kann der Wert einfach verwendet werden, in letzterem Fall sollte hingegen zur Laufzeit überprüft werden, ob der tatsächliche Wert mit dem vom Orakel vorgegebenen übereinstimmt.

Allerdings ist es fraglich, ob die Verwendung eines nicht vertrauenswürdigen Orakels überhaupt sinnvoll ist, schließlich wird die Einhaltung einer vorgegebenen Zeitschranke in diesem Fall erst zur Laufzeit, wenn die tatsächlichen Werte bekannt sind, sichergestellt. Werden hingegen keine Werte angegeben, so kann die Methode zwar nicht verifiziert werden, es wird jedoch eine modifizierte Methode erzeugt, die entsprechende Laufzeitüberprüfungen enthält und sich beendet, wenn die zugewiesene Zeitschranke überschritten wird (siehe Abschnitt 6.4).

In beiden Fällen besteht somit weiterhin die Gefahr, dass die Methode abgebrochen wird, wenn die Zeit überschritten wird oder ein Parameterwert nicht mit dem vom Orakel vorgegebenen übereinstimmt. Das nicht vertrauenswürdige Orakel hat zwar den Vorteil, dass weniger Laufzeitüberprüfungen eingefügt werden müssen, die Methode mithin weniger verlangsamt wird, als wenn die Ausführungsdauer nach jedem Schleifendurchlauf überprüft werden muss. Allerdings hat diese Vorgehensweise den Nachteil, dass eine Methode bereits abgebrochen wird, wenn lediglich der Parameterwert nicht mit dem angegebenen übereinstimmt. Tatsächlich ist dabei noch nicht sichergestellt, dass deshalb auch das Zeitlimit wirklich überschritten wird.

Bei der Implementierung des Verifiers wurde darum auf die Möglichkeit eines nicht vertrauenswürdigen Orakels verzichtet und alle angegebenen Werte werden als korrekt angesehen.

6.3.3.2 Simulierter JVM-Zustand

Bevor auf die partielle Ausführung einer Methode näher eingegangen wird, soll zunächst der zur Ausführung verwendete Zustand beschrieben werden. Dieser ähnelt dem bei der DFA verwendeten insofern, als wiederum ein Stapel und lokale Variablen simuliert werden. Auch hier wird auf die Simulation des Heaps verzichtet, da alle auf dem Heap angelegten Datenstrukturen (Objekte und Arrays) möglicherweise auch von anderen Threads verändert werden könnten. Zusätzlich zu Stapel und lokalen Variablen wird die bisher verbrauchte Ausführungsdauer im Zustand vermerkt.

Im Gegensatz zur Vorgehensweise bei der DFA, wird jedoch nicht für jeden Befehl ein Vorgänger- und Nachfolgerzustand konstruiert, sondern die Befehle werden während der partiellen Ausführung nacheinander auf den selben Zustand angewandt. Dementsprechend wird auch eine Art Befehlszähler verwaltet, der angibt, welcher Befehl als nächstes ausgeführt oder simuliert werden soll (tatsächlich enthält dieser Eintrag nicht die Adresse eines Befehls, sondern einen Verweis auf eine Kante – siehe Abschnitt 6.3.3.4).

Die im Zustand verwendeten Datenwörter enthalten nun, für diejenigen Wörter die zur Ausführung benötigt werden, die tatsächlichen Werte, bzw. sind, für alle nicht benötigten Wörter, als ungültig gekennzeichnet.

Der Anfangszustand besteht aus dem leeren Stapel und lokalen Variablen die als ungültig gekennzeichnet sind. Allerdings stehen in denjenigen Variablen, die benötigte Argumente der Methode enthalten, die vom Orakel zur Verfügung gestellten Werte.

Eine einfache Verzweigung wird nicht ausgeführt, sondern ihre Laufzeit wird nur abgeschätzt, indem die Zeit der beiden Zweige bestimmt und daraus das Maximum genommen wird. Um die Zweige zu analysieren muss der Zustand an dieser Stelle kopiert werden, sodass jeder Zweig auf einem eigenen Zustand ausgeführt werden kann. Anschließend müssen die Zustände wieder vereinigt werden, so dass eine Vereinigungsoperation benötigt wird. Diese wird folgendermaßen durchgeführt: Zunächst wird jedes Datum auf dem Stack und in den lokalen Variablen in den beiden Zuständen überprüft. Stimmen die Datenwörter in beiden Zuständen überein, so wird dem Datenwort im Ergebniszustand der selbe Wert zugewiesen. Stimmen sie nicht überein, so wird das Wort im Ergebniszustand als ungültig markiert.

Als Ausführungsdauer wird im Ergebniszustand das Maximum der Zeiten der zu vereinigenden Zustände vermerkt.

6.3.3.3 Ausführung und Simulation einzelner Befehle und Blöcke

Bevor nun die Ausführung der gesamten Methode näher beschrieben wird, soll zunächst die Abarbeitung der einzelnen Befehle betrachtet werden. Dabei muss zwischen den in Schritt zwei markierten bzw. nicht markierten Befehlen unterschieden werden. Letztere müssen lediglich *simuliert* werden, d. h. es muss nur die entsprechende Anzahl von Operanden vom Stapel entfernt bzw. Ergebnisse abgelegt oder in lokale Variablen geschrieben werden, wobei alle erzeugten Daten „ungültig" sind.

Ist der Befehl hingegen markiert, so muss er tatsächlich *ausgeführt* werden. Die erzeugten Ergebnisse müssen also tatsächlich aus den Operanden berechnet werden, und enthalten konkrete Werte. Beim Befehl `iadd` beispielsweise muss als Ergebnis die Summe der Operanden abgelegt werden. Natürlich kann ein Befehl nur dann ausgeführt werden, wenn die Operanden nicht ungültig sind. Dafür sorgt jedoch der zweite Analyseschritt.

Auf eine detaillierte Darstellung aller ausführbaren Befehle wird aus Platzgründen verzichtet, zumal die Ausführungsweise in der JVM Spezifikation ([LY97]) nachgelesen werden kann. Wie bereits erwähnt, gibt es allerdings Befehle (z. B. `getfield` oder `Taload`) die nicht ausgeführt werden können, da das Ergebnis zur Verifikationszeit nicht zuverlässig ermittelt werden kann. In diesem Fall muss das Ergebnis der Befehls vom Orakel zur Verfügung gestellt werden. Ist kein Wert verfügbar scheitert die Verifikation und eine modifizierte Methode muss erstellt werden (siehe Abschnitt 6.4).

Nachdem ein Befehl simuliert oder ausgeführt wurde, wird schließlich noch die Ausführungszeit im JVM-Zustand um den Beitrag des Befehls erhöht. Dieser wird einer Tabelle entnommen, welche die entsprechenden Zeiten enthält.

Wird dabei das vorgegebene Zeitlimit überschritten, so wird die Analyse beendet. Dabei muss die Bearbeitung von Methodenaufrufen speziell behandelt werden, wie im folgenden Abschnitt erläutert.

Die Simulation[1] eines Basic Blocks erfolgt nun, indem die Befehle der Reihe nach abgearbeitet werden.

Methodenaufrufe

Die Simulation von Methodenaufrufen, also `invoke` Befehlen, erfordert ein etwas komplexeres Vorgehen als die Simulation anderer Befehle.[2] Denn in diesem Fall muss die Ausführungsdauer nicht nur um die Zeit erhöht werden, die zur Durchführung des eigentlichen Methodenaufrufs benötigt wird, sondern auch um die Zeit, die die aufgerufene Methode zu ihrer Ausführung benötigt.

Dementsprechend muss für jeden Methodenaufruf zunächst die aufgerufene Methode simuliert und ihre Laufzeit ermittelt werden. Diese wird dann zur Ausführungsdauer im Zustand addiert.

Im Einzelnen sind dabei drei Fälle zu unterscheiden:

- Das Orakel stellt einen Wert für die Laufzeit der Methode zur Verfügung.
- Die Methode wurde noch nicht analysiert, so dass ihre Ausführungsdauer noch nicht bekannt ist. In diesem Fall wird diese Methode ebenfalls einer WCET-Analyse unterzogen. Dabei wird die in der aufrufenden Methode noch verbleibende Zeit als Zeitlimit für

[1] Während die Ausdrücke „Ausführung" und „Simulation" für einzelne Befehle eine unterschiedliche Bedeutung haben, werden sie auf Basic Block Ebene weitgehend synonym verwendet.

[2] Die *Ausführung* von `invoke` Befehlen wird in der momentanen Implementierung noch nicht unterstützt.

die aufgerufene Methode verwendet. Wird die Analyse abgebrochen, so muss auch die Analyse der aufrufenden Methode abgebrochen werden.

- Die aufgerufene Methode wurde bereits analysiert, wobei bei dieser Analyse ein Zeitlimit z verwendet wurde. Hier sind erneut vier Fälle zu unterscheiden:
  - Die Methode konnte erfolgreich analysiert werden und die Laufzeit wurde ermittelt.
  - Die Analyse der Methode wurde auf Grund einer Zeitüberschreitung abgebrochen und das Limit z war kleiner als die in der aufrufenden Methode noch verbleibende Zeit. In diesem Fall muss die Methode erneut analysiert werden, mit der verbleibenden Zeit als Zeitlimit.
  - Die Analyse der Methode wurde abgebrochen und das verwendete Limit z war größer als die noch verbleibende Zeit. In diesem Fall kann die Analyse der aufrufenden Methode ebenfalls beendet werden, da die verbleibende Zeit zur Ausführung der aufgerufenen Methode nicht ausreichen würde.
  - Die Methode konnte auf Grund mangelnder Daten nicht analysiert werden. Dies führt wiederum zum Abbruch der Analyse der aufrufenden Methode.

Dabei tritt jedoch noch ein Problem auf: Es kann zu Verifikationszeit nicht immer bestimmt werden, welche Methode tatsächlich aufgeführt wird. Erwartet eine Methode beispielsweise als Parameter ein Objekt vom Typ `java.lang.Object`, so kann statisch nicht festgestellt werden, welcher Unterklasse das Objekt, das tatsächlich übergeben wird, angehört. Dementsprechend ist auch nicht feststellbar, welche Methode ausgeführt wird, wenn etwa `toString()` aufgerufen wird.

Der einzige Ausweg besteht darin, alle Methoden, die an dieser Stelle in Frage kommen, zu analysieren und die längste Laufzeit zu verwenden. Dieser Ansatz führt jedoch nur dann zum Erfolg, wenn diese Methoden zur Verifikationszeit ermittelt werden können. Das bedeuted insbesondere, dass das dynamische Nachladen von Code erschwert wird.

Nachdem die Simulation von Methodenaufrufen nun beschrieben wurde, bleibt zu bemerken, dass ein Methodenaufruf auch *ausgeführt* werden könnte, wenn alle Parameterwerte berechnet werden können. Der Unterschied zur Simulation besteht lediglich darin, dass im Anfangszustand der aufgerufenen Methode die entsprechenden Parameter eingetragen werden. Allerdings wurde diese Vorgehensweise in der vorliegenden Arbeit aus Zeitgründen nicht implementiert.

6.3.3.4 Laufzeitbestimmung mit Hilfe des modifizierten Kontrollflussgraphen

Die eigentliche Ausführung erfolgt nun mit Hilfe des im ersten Schritt erstellten und vereinfachten modifizierten Kontrollflussgraph. Dabei wird eine Kante simuliert, indem der durch sie ersetzte Teilgraph simuliert wird. Dieser Prozess beginnt mit der nach der Reduktion in Schritt 1 übriggebliebenen Kante und wird rekursiv fortgesetzt, bis schließlich Kanten erreicht werden, die nur einen einzigen Basic Block darstellen bzw. Knoten die aus einem einzigen `if` Befehl bestehen.

Dieser Ansatz mag zunächst unnötig kompliziert erscheinen. Ein einfacherer Ansatz besteht darin, die Methode auszuführen, indem man die Befehle einfach der Reihe nach abarbeitet.

Bei `if`-Befehlen, die nicht ausgeführt werden müssen, wird dann der Zustand kopiert und die Ausführung in den unterschiedlichen Zweigen mit jeweils einem eigenen Zustand fortgesetzt.

Dabei besteht jedoch das Problem, festzustellen, an welchem Punkt die Zustände wieder zusammengeführt werden müssen. Erreicht die Ausführung eines Zweiges beispielsweise einen Befehl mit mehreren Vorgängern, so ist nicht klar, ob die Ausführung fortgesetzt werden soll oder nicht. Denn einerseits ist es möglich, dass dieser Befehl auch bei der parallel verlaufenden Ausführung des anderen Zweiges erreicht wird. In diesem Fall müssten die Zustände an dieser Stelle vereint werden. Andererseits haben die weiteren Vorgänger des Befehls jedoch möglicherweise nichts mit besagtem zweiten Zweig zu tun, sodass die Simulation fortgesetzt werden muss.

Wird die Methode hingegen mit Hilfe des Graphen simuliert, so tritt dieses Problem nicht auf. Deshalb wurde dieser Ansatz gewählt.

Bei der Simulation der Kanten und Knoten muss zwischen drei verschiedenen Knoten und fünf verschiedenen Kanten unterschieden werden. Die einzelnen Typen werden im Folgenden genauer vorgestellt:

einfache Knoten: Solche Knoten, die lediglich als Ziel einer Kante dienen und nur einer Adresse zugeordnet sind, verändern den Simulationszustand nicht.

if-Knoten: Knoten, denen ein `if` Befehl zugeordnet ist, werden unterschiedlich behandelt, je nachdem ob sie markiert sind oder nicht. Ist der Knoten nicht markiert, so wird lediglich die entsprechende Anzahl von Operanden vom Stapel entfernt, d. h. der Befehl wird simuliert.

Ist der Knoten hingegen markiert, so wird der Befehl ausgeführt, d. h. die Operanden werden miteinander verglichen und je nach Ergebnis wird im Zustand vermerkt, welche der beiden Ausgangskanten als nächstes simuliert werden soll.

Außerdem muss natürlich auch die Ausführungszeit entsprechend erhöht werden, egal ob der Knoten markiert ist oder nicht.

Die Ausführung von Knoten, denen kein einfacher `if`-Befehl, sondern ein `lookupswitch` oder `tableswitch` Befehl zugeordnet ist, muss gesondert behandelt werden. Diesen Befehlen sind mehrere Knoten zugeordnet, einer für jedes mögliche Sprungziel; jeder der Knoten enhält zwei Kanten, eine zum Sprungziel und eine zum nächsten Knoten des Befehls.

Zunächst wird der Befehl ausgeführt und ermittelt, welches der Sprungziele ausgewählt wurde. Wenn dieses Sprungziel einer der ausgehenden Kanten zugeordnet ist, so wird diese im Zustand als nächste auszuführende Kante vermerkt. Anderenfalls wird die Kante vermerkt, die zum nächsten Knoten führt, der zum Befehl gehört.

Da der Befehl also mehrfach ausgeführt wird, darf die Gesamtlaufzeit nicht bei jeder Ausführung erhöht werden. Stattdessen wird sie nur dann erhöht, wenn ein Sprungziel ausgewählt wurde, dem eine ausgehende Kante des Knotens zugeordnet ist.

Regel 3 Knoten: Knoten die durch Anwendung der Regel 3 entstanden sind werden durch Simulation des ersetzten Teilgraphen abgearbeitet. Dieser besteht aus zwei Verzweigungsknoten 1 und 2, die durch eine Kante k_1 verbunden sind und jeweils eine Kante v_1

und v_2 zu einem dritten Knoten 4 haben (vgl. Abbildung 6.5 (d), Seite 53). Die Knoten 1 und 2 sind beide markiert und müssen demnach nicht simuliert, sondern ausgeführt werden.

Zunächst wird Knoten 1 ausgeführt und anschließend die von der Verzweigung bestimmte Kante. Handelt es sich um Kante v_1, so ist die Ausführung des Gesamtknotens damit beendet, wobei Kante v im Zustand als die nächste auszuführende Kante vermerkt wird.

Anderenfalls wird Kante k_1, Knoten 2 und – basierend auf der in Knoten 2 getroffenen Entscheidung – Kante k_2 oder v_2 ausgeführt. Wurde v_2 ausgeführt, so wird im aktuellen Zustand wiederum v als nächste auszuführende Kante vermerkt. Bei Ausführung von k_2 wird stattdessen k als nächste Kante eingetragen.

einfache Kanten: Eine einfache Kante, der ein Basic Block der Methode zugeordnet ist, wird simuliert indem besagter Block ausgeführt wird.

Regel 1 Kanten: Eine Kante die durch Anwendung von Regel 1 entstanden ist, d. h. aus einer Sequenz zweier Kanten k_1 und k_2 besteht, wird simuliert indem zunächst die Kante k_1 simuliert wird, und auf dem Ergebniszustand Kante k_2. Der Zielknoten von k_1 und Anfangsknoten von k_2 wird nicht simuliert, da nur eine Kante von ihm ausgeht. Sollte es sich bei diesem Knoten um eine Verzweigung handeln, deren Zweige durch Regel 2a oder b vereinigt wurden, so muss der Knoten an dieser Stelle dennoch nicht ausgeführt werden. Die Ausführung erfolgt stattdessen während der Simulation der entsprechenden Regel 2a oder b Kante.

Regel 2a Kanten: Zur Simulation einer Kante, die durch Anwendung von Regel 2a entstanden ist, d. h. die aus einer Verzweigung besteht, wird zunächst der zugehörige if-Knoten simuliert. Anschließend muss geprüft werden, ob dieser Knoten markiert ist.

Ist er nicht markiert, so wird der Zustand zunächst kopiert. Daraufhin werden die beiden Zweige, Kanten k_1 und k_2, jeweils auf einem eigenen Zustand ausgeführt. Die Ergebnisse werden dann mit Hilfe der oben beschriebenen Vereinigungsoperation wieder zu einem Zustand zusammengeführt.

Ist der Knoten hingegen markiert, so wurde bei obiger Simulation des Knotens im Zustand die nächste auszuführende Kante vermerkt. Diese wird simuliert und das Ergebnis auch als Gesamtergebnis verwendet.

Regel 2b Kanten: Solche Kanten, die eine Schleife darstellen, d. h. durch Anwendung von Regel 2b entstanden sind, werden folgendermaßen simuliert: Zunächst wird der zugehörige if-Knoten ausgeführt. Je nachdem, welche Kante anschließend im Zustand vermerkt ist, wird entweder die Schleifenkante oder die vom if-Knoten wegführende Kante (Kante k in Abbildung 6.5 (c)) ausgeführt. Nach jeder Ausführung der Schleifenkante wird der Knoten erneut simuliert, so lange bis die Schleife schließlich verlassen wird.

Regel 3 Kanten: Kanten die bei der Anwendung von Regel 3 neu eingefügt wurden (Kanten k und v aus Abbildung 6.5 (d)) enthalten keinerlei Code und verändern demnach den Zustand nicht.

Sobald die Analyse der obersten Kante beendet ist, wurden alle nötigen untergeordneten Kanten analysiert und die im Zustand vermerkte Zeit kann als Abschätzung für die worst-case Laufzeit der Methode verwendet werden.

6.4 Erzeugung modifizierter Methoden

Wie bereits erwähnt, soll eine modifizierte Methode erstellt werden, falls die Laufzeit einer Methode nicht mit Hilfe der vorgestelten WCET Analyse bestimmt werden kann. In diesem Fall sollen an entsprechenden Stellen Befehle eingefügt werden, die einerseits festhalten, wie lange die Methode bereits rechnet, andererseits überprüfen, ob diese Ausführungszeit noch unterhalb der vorgegebenen Grenze liegt.

Dabei sind zwei unterschiedliche Ansätze denkbar. Zum einen besteht die Möglichkeit, entsprechende Code Sequenzen in den Bytecode der Methode einzufügen. Zum anderen könnten solche Überprüfungen aber auch direkt durch Einfügen entsprechender Maschinenbefehle durch den Native-Code-Compiler (siehe [Waw01]) erfolgen. In diesem Fall müsste der Verifier lediglich feststellen, an welchen Stellen eine solche Überprüfung zu erfolgen hätte.

Letztere Vorgehensweise hat den Vorteil, dass auch auf die, von der zu Grunde liegenden Hardware zur Verfügung gestellten Mittel zur Zeitmessung (wie beispielsweise die Echtzeituhr) zurückgegriffen werden kann. Bei der Modifikation der Methode hingegen kann die verbrauchte Zeit nur dadurch bestimmt werden, dass für jeden ausgeführten Befehl eine feste Zeit zur Gesamtausführungszeit addiert wird. Dieses Verfahren ist dementsprechend wesentlich zeitaufwändiger und ungenauer als eine Zeitmessung in Hardware. Allerdings hat diese „Softwarelösung" den Vorteil, portabel zu sein.

Im Rahmen dieser Arbeit wurde die Softwarelösung implementiert, allerdings werden die ermittelten Positionen für Laufzeitüberprüfungen ebenfalls gespeichert, so dass die „Hardwarelösung" bei entsprechender Unterstützung durch den Compiler ebenfalls realisierbar ist.

Wie bereits erwähnt, müssen zwei Aufgaben erfüllt werden: Zum einen muss die bisherige Ausführungsdauer der Methode bestimmt werden („Zeitprotokollierung"), zum anderen muss überprüft werden, ob die vorgegebene Laufzeitbeschränkung noch eingehalten wird („Überprüfung").

Erfolgt die Zeitprotokollierung mit Hilfe einer Hardwareuhr, so muss lediglich zum Beginn der Methode die Uhr ausgelesen werden, und dieser Wert bei jeder Überprüfung der Zeitbeschränkung mit der aktuellen Zeit verglichen werden. Im Folgenden wird jedoch davon ausgegangen, dass eine solche Hardwareunterstützung nicht zur Verfügung steht.

Bei der Frage an welchen Stellen in der Methode Code zur Protokollierung und Überwachung eingefügt werden soll, gilt es zu beachten, dass natürlich auch die eingefügten Befehle Zeit verbrauchen, so dass die Methode „langsamer" ausgeführt wird als vorher. Dementsprechend sollte sich die Zahl der eingefügten Anweisungen auf ein Mindestmaß beschränken.

Andererseits muss sichergestellt werden, dass die Zeit möglichst genau mitprotokolliert wird und außerdem eine eventuelle Überschreitung der Zeitschranke möglichst schnell erkannt wird.

Zunächst zur Protokollierung: Aus den genannten Gründen kann der Zeitzähler natürlich nicht nach jedem Befehl erhöht werden, sondern die Ausführungszeit einer Befehlssequenz muss schon bei der Verifikation ermittelt werden. Allerdings muss zumindest vor jeder Überprüfung sowie vor jedem Ende eines Basic Block die Ausführungsdauer erhöht werden.

Die Frage, wann eine Überprüfung eingefügt werden soll, hängt davon ab, ob bzw. um wieviel die vorgegebene Zeit überschritten werden darf. Im Allgemeinen gilt, dass mehr Überprüfungen eingeführt werden müssen, je geringer die erlaubte Zeitüberschreitung ist. Wird beispielsweise nur einmal innerhalb jeder Schleife eine Überprüfung eingefügt, so wird die Methode zwar sicherlich nach einer gewissen Zeit beendet, allerdings kann – wenn der Programmteil vor der Schleife oder der Schleifenrumpf selbst sehr zeitaufwändig ist – viel Zeit vergehen, bevor eine Überschreitung der Schranke erkannt wird.

Aber auch wenn, nebst der Zeitschranke, eine maximale Abweichung angegeben wird, kann noch immer nicht genau festgelegt werden, wo eine Überprüfung nötig ist, um diese Abweichung zu garantieren. Schließlich kann im Allgemeinen nicht bestimmt werden, wie viel Zeit zwischen zwei Stellen im Programm verbracht wird, wenn diese beiden Stellen nicht im selben Basic Block liegen.

In dieser Arbeit wurde dieses Problem folgendermaßen umgangen: Zunächst wird eine obere Schranke der Ausführungszeit der Methode unter der Annahme bestimmt, dass jeder Schleifenrumpf nur einmal ausgeführt wird. Diese Ausführungszeit kann leicht bestimmt werden, indem die Zeiten aller Befehle der Methode zusammengezählt werden. Liegt diese obere Schranke unter der vorgegebenen Abweichung, so genügt es, Überprüfungen vor jedem Rückwärtssprung (also von einer höheren zu einer niedrigeren Adresse) im Bytecode einzuführen. Schließlich muss jede Schleife einen Rückwärtssprung enthalten und der Schleiferumpf kann nicht mehr Zeit benötigen als die oben bestimmte obere Schranke. Diese Vorgehensweise kann bei Methoden angewendet werden, die nur wenig Code enthalten und wenn die erlaubte Abweichung relativ groß ist.

Sollte diese Vorgehensweise nicht anwendbar sein, so wird entweder nach jedem Basic Block oder – sollte die Ausführungsdauer eines Basic Blocks die maximale Abweichung überschreiten – nach jeder Befehlsfolge, die der Dauer der maximalen Abweichung entspricht, eine Überprüfung eingeführt.

7 Weitere JX spezifische Verifikationsschritte

Zusätzlich zu den bislang beschriebenen komplexen Analysen werden vom Verifier noch verschiedene, weniger aufwändige Verifikationsschritte durchgeführt. Diese werden im Folgenden erläutert.

System-Final Analyse: Bei der System-Final Analyse geht es darum, Methoden zu markieren die zwar durch den Programmierer nicht als `final` gekennzeichnet wurden, jedoch tatsächlich nicht mehr überschrieben werden. Diese Information wird vom Compiler benötigt, um Inlining zu ermöglichen.

Zu diesem Zweck werden alle abgeleiteten Klassen der aktuellen Klasse überprüft. Wird in einer dieser Klassen die Methode überschrieben, so ist die Methode nicht System-Final.

Leaf Analyse: Die Leaf Analyse dient dazu, Methoden aufzufinden die selbst keine anderen Methoden mehr verwenden. Auch diese Information wird vom Compiler für das Inlining benötigt.

Um entsprechende Methoden zu markieren wird der Bytecode jeder Methode nach `invoke` Befehlen durchsucht. Wird kein solcher Aufruf gefunden, so handelt es sich um eine Leaf-Methode.

Überprüfung von `long`, `float` oder `double` Befehlen: Da JX bzw. der zugehörige Native-Code Compiler bislang keine Operationen auf `long`, `float` oder `double` Werten unterstützt, muss für jede Methode geprüft werden, ob solche Werte verwendet werden. Dazu wird der Bytecode jeder Methode durchlaufen und nach Befehlen dieser Typen durchsucht.

Überprüfung nicht implementierbarer Interfaces: In JX existieren verschiedene Interfaces, wie z. B. `MappedBigEndian`, die nicht implementiert werden dürfen (vgl. [MG01]). Deshalb muss für jeden `new` Aufruf überprüft werden, ob die angegebene Klasse eines dieser Interfaces implementiert. Diese Überprüfung kann statisch erfolgen, indem jede Methode nach `new` Befehlen durchsucht wird. Anschließend wird geprüft, ob die im Bytecode als Argument des `new` Befehls angegebene Klasse das Interface implementiert. Falls das der Fall ist, wird die Verifikation abgebrochen.

8 Zusammenfassung und Ausblick

8.1 Zusammenfassung

Ziel dieser Arbeit war es, einen Verifier für das Java Betriebssystem JX zu entwickeln und zu implementieren. Dieser sollte zusätzlich zur Standard Java Bytecodeverifikation verschiedene JX spezifische Verifikationsschritte durchführen. Außerdem sollte auf eine leichte Erweiterbarkeit geachtet werden.

Zunächst wurde ein im Bereich der Programmanalyse und des Compilerbaus weit verbreitetes Vefahren, die Datenflussanalyse, an die speziellen Bedürfnisse der Verifikation von Java Bytecode angepasst. Insbesondere wurden Erweiterungen für die Analyse von Ausnahmen und Subroutinen eingeführt.

Des Weiteren wurde ein Zustandsraum für die Analyse vorgestellt, dessen Zustände denen der Java Virtual Machine entsprechen und aus einem Stapel und lokalen Variablen aufgebaut sind. Dabei werden den einzelnen Datenwörtern die untersuchten Eigenschaften zugeordnet. Außerdem wurden, ausgehend von Terminierungs- und Anwendbarkeitsbedingungen des DFA-Algorithmus für Zustandsräume, ähnliche Bedingungen für Eigenschaftsräume formuliert.

Schließlich wurde ein spezieller Eigenschaftsraum vorgestellt, der es erlaubt zu erkennen, ob zwei Datenwörter an unterschiedlichen Positionen im Zustand identisch sind.

Der in der JVM-Spezifikation beschriebene Bytecode Verifier wurde mit Hilfe der so entwickelten Konzepte implementiert. Des Weiteren wurde eine Null-Pointer Analyse entwickelt und implementiert, die es erlaubt unnötige Laufzeitüberprüfungen bei Zugriffen auf Referenzen zu eliminieren.

Einer der Kernpunkte dieser Arbeit war außerdem die Entwicklung und Umsetzung einer Analyse, die sicherstellt, dass eine bestimmte Methode ein vorgegebenes Zeitlimit einhält. Hierzu wurde ein Verfahren entwickelt, um mit Hilfe einer partiellen Ausführung der Methode die Zahl der Durchläufe eventuell vorhandener Schleifen zu ermitteln. Durch die partielle Ausführung wird erreicht, dass ein – im Vergleich zu anderen WCET Analyseverfahren – relativ großes Spektrum an Methoden analysierbar ist. Die einzigen Bedingung ist, dass alle Parameter, von denen Schleifen abhängen, bereits zur Verifikationszeit bekannt sind. Insbesondere werden an die Schleifenbedingungen selbst keinerlei Anforderungen gestellt.

Sind nicht alle zur partiellen Ausführung benötigten Informationen vorhanden, so besteht die Möglichkeit die fehlenden Daten durch anderweitig ermittelte Werte ergänzen zu lassen. Dadurch kann beispielsweise Information durch den Programmierer in die Analyse eingebracht werden.

Ist eine Methode trotzdem nicht analysierbar, so kann als letzter Ausweg eine modifizierte

Methode erstellt werden. Diese enthält Laufzeitüberprüfungen, die sicherstellen, dass eine vorgegebene Zeitschranke nicht, oder nur um einen bestimmten Betrag überschritten wird.

Insgesamt kann also stets garantiert werden, dass eine Methode ihre Laufzeitbeschränkung einhält. Somit kann jede beliebige Methode ununterbrechbar ausgeführt werden, ohne dass die Gefahr besteht, dass die Antwortzeit des Systems dadurch wesentlich beeinträchtigt wird.

8.2 Ausblick

Die Analyse der Laufzeit beschränkt sich momentan auf die partielle Ausführung der Methode. Wie bereits erwähnt, müssen dazu die benötigten Parameter bekannt sein. Die Zahl der Durchläufe von Schleifenkonstrukten, bei denen eine einzelne Variable in jedem Durchlauf um einen konstanten Betrag erhöht wird, ließe sich jedoch auch analytisch bestimmen. Dabei müssen die Anfangswerte nicht fest vorgegeben sein, sonern es genügt einen Bereich von Anfangswerten vorzugeben. Deshalb wäre es unter Umständen sinnvoll die Analyse der Ausführungsdauer so zu erweitern, dass ein solches Verfahren zusätzlich implementiert wird.

Außerdem können, mit Hilfe des in dieser Arbeit vorgestellten Grobkonzepts, leicht weitere Analysen implementiert werden, die dem Compiler Ergebnisse für weitere Optimierungen zur Verfügung stellen. Beispielsweise kann eine Escape-Analyse (siehe [WR, Bla99]) eingeführt werden, um Festzustellen, ob ein in einer Methode erzeugtes Objekt den Kontext der Methode verlässt oder ob es nur lokal in der Methode oder von aufgerufenen Methoden verwendet wird. In letzterem Fall kann es auf dem Stapel der Methode angelegt werden, so dass es automatisch, beim Verlassen der Methode, wieder gelöscht wird, wodurch der Garbage Collector entlastet wird.

Aufbauend auf eine solche Escape Analyse kann auch festgestellt werden, ob ein Objekt von anderen Threads verwendet werden kann oder nicht. Ist ein Objekt Thread-lokal, so können andere Analysen (z. B. die NPA), die momentan auf lokale Daten der Methode beschränkt sind, entsprechend erweitert werden um auch solche Objekte zu berücksichtigen.

A Eigenschaften der definierten Zustandsräume

A.1 Allgemeiner Zustandsraum für die Bytecodeverifikation

In diesem Abschnitt soll gezeigt werden, dass der in Abschnitt 3.2.2 definierte Zustandsraum $(L, \sqsubseteq, \sqcup, \sqcap, \perp, \top)$ tatsächlich ein vollständiges Gitter darstellt, wenn der zu Grunde liegende Eigenschaftsraum $(L', \sqsubseteq', \sqcup', \sqcap', \perp', \top')$ diese Bedingung erfüllt. Außerdem muss nachgewiesen werden, dass sich auch die Erfüllung der aufsteigenden Kettenbedingung vom Eigenschaftsraum auf den Zustandsraum überträgt.

Zunächst gilt es nachzuweisen, dass die in Abschnitt 3.2.2 definierte Relation $\sqsubseteq$ tatsächlich eine Halbordnung, d. h. reflexiv, transitiv und antisymmetrisch ist.

- Dass $\sqsubseteq$ reflexiv ist, folgt sofort aus der Reflexivität von $\sqsubseteq'$.
- Für den Nachweis der Transitivität gelte $l_1 \sqsubseteq l_2$ und $l_2 \sqsubseteq l_3$. Für die drei Bedingungen H1-3 (siehe Seite 21) gilt:
  - H1: Da gelten muss $\#l_1.stack \leq \#l_2.stack$ und $\#l_2.stack \leq \#l_3.stack$ gilt auf Grund der Transitivität von $\leq$ auch $\#l_1.stack \leq \#l_3.stack$.
  - H2 und H3: Für alle $0 \leq i < \#l_1.lVars$ gilt $l_1.lVars[i] \sqsubseteq' l_2.lVars[i]$ und $l_2.lVars[i] \sqsubseteq' l_3.lVars[i]$. Wegen der Transitivität von $\sqsubseteq'$ folgt daraus sofort $l_1.lVars[i] \sqsubseteq' l_3.lVars[i]$. Gleiches gilt auch für die Stapelworte 0 bis $\#l_1.stack - 1$.

  Somit folgt aus der Annahme, dass gilt $l_1 \sqsubseteq l_3$ und die Transitivität von $\sqsubseteq$.
- Die Antisymmetrie von $\sqsubseteq$ wird auf ähnliche Weise auf die Antisymmetrie von $\sqsubseteq'$ und $\leq$ zurückgeführt. Es gelte $l_1 \sqsubseteq l_2$ und $l_2 \sqsubseteq l_1$. Aus den Bedingungen H1-3 folgt dann:
  - H1: $\#l_1.stack \leq \#l_2.stack$ und $\#l_2.stack \leq \#l_1.stack$, so dass gelten muss $\#l_1.stack = \#l_2.stack$.
  - H2 und H3: Für alle $0 \leq i < \#l_1.lVars$ gilt $l_1.lVars[i] \sqsubseteq' l_2.lVars[i]$ und $l_2.lVars[i] \sqsubseteq' l_1.lVars[i]$. Wegen der Antisymmetrie von $\sqsubseteq'$ gilt dann $l_1.lVars[i] = l_2.lVars[i]$. Gleiches gilt auch für die Stapelworte 0 bis $\#l_1.stack - 1$.

  Da die Zustände aber nach Definition nur aus den lokalen Variablen und den Stapelwörtern bestehen, ist somit $l_1 = l_2$.

Demnach ist $\sqsubseteq$ reflexiv, transitiv und antisymetrisch, also eine Halbordnung.

Weiter muss gezeigt werden, dass die Operationen $\sqcup$ und $\sqcap$ tatsächlich mit den in Abschnitt 3.1.4 angegebenen übereinstimmen, d. h. dass es sich um die kleinste obere bzw. größte untere Schranke handelt. Im Folgenden wird lediglich $\sqcup$ betrachtet, der Nachweis für $\sqcap$ verläuft analog.

Dass $l = l_1 \sqcup l_2$ tatsächlich eine obere Schranke von l_1 und l_2 ist, folgt sofort aus der Definition und soll nicht weiter betrachtet werden. Es bleibt zu zeigen, dass es sich auch um die *kleinste* obere Schranke handelt. Dazu werden zwei Fälle unterschieden: Entweder sind die beiden Zustände vergleichbar, also $l_1 \sqsubseteq l_2$ oder $l_2 \sqsubseteq l_1$, oder sie sind nicht vergleichbar.

Es sei also $l_1 \sqsubseteq l_2$ (der Fall $l_2 \sqsubseteq l_1$ verhält sich analog). Dann gilt:

- wegen O1 und H1: $\#l.stack = \max(\#l_1.stack, \#l_2.stack) = \#l_2.stack$
- wegen O2, H2 und H1: Für $0 \leq i < \#l_1.stack$ gilt $l.stack[i] = l_1.stack[i] \sqcup' l_2.stack[i] = l_2.stack[i]$, da $l_1.stack[i] \sqsubseteq' l_2.stack[i]$.
  Für $\#l_1.stack \leq i < \#l_2.stack$ gilt per Definition ebenfalls $l.stack[i] = l_2.stack[i]$, da $\#l_1.stack \leq \#l_2.stack$.
- wegen O3 und H3: Für $0 \leq j < \#l_2.lVars$ gilt $l.lVars[j] = l_1.lVars[j] \sqcup' l_2.lVars[j] = l_2.lVars[j]$, da $l_1.lVars[j] \sqsubseteq' l_2.lVars[j]$.

Deshalb ist $l = l_1 \sqcup l_2 = l_2$ und somit minimal (da jedes kleinere Element keine obere Schranke zu l_2 mehr bildet).

Es seien nun l_1 und l_2 unvergleichbar bzgl. $\sqsubseteq$, und $l_1 \sqcup l_2 = l$. Des Weiteren sei l' eine obere Schranke von l_1 und l_2, die kleiner als l ist, also $l_1 \sqsubseteq l'$, $l_2 \sqsubseteq l'$ und $l' \sqsubseteq l$.

- wegen H1 und O1 gilt:
  (i) $\#l_1.stack \leq \#l'.stack \leq \#l.stack = \max(\#l_1.stack, \#l_2.stack)$ und
  (ii) $\#l_2.stack \leq \#l'.stack \leq \#l.stack = \max(\#l_1.stack, \#l_2.stack)$.
  Wenn gilt $\#l_1.stack < \#l_2.stack$, so ist wird (ii) zu $\#l_2.stack \leq \#l'.stack \leq \#l_2.stack$.
  Also ist $\#l'.stack = \#l_2.stack = \#l.stack$.
  Gilt hingegen $\#l_1.stack \geq \#l_2.stack$, so wird (i) zu $\#l_1.stack \leq \#l'.stack \leq \#l_1.stack$.
  Also ist auch in diesem Fall $\#l'.stack = \#l.stack$.
  Insgesamt gilt also $\#l'.stack = \#l.stack$.
- wegen H3 und O3 gilt für lokale Variablen:
  (i) $l_1.lVars[i] \sqsubseteq' l'.lVars[i] \sqsubseteq' l.lVars[i] = l_1.lVars[i] \sqcup' l_2.lVars[i]$ und
  (ii) $l_2.lVars[i] \sqsubseteq' l'.lVars[i] \sqsubseteq' l.lVars[i] = l_1.lVars[i] \sqcup' l_2.lVars[i]$
  Da es sich bei $l_1.lVars[i] \sqcup' l_2.lVars[i]$ jedoch um die *kleinste* obere Schranke handelt muss auch gelten $l_1.lVars[i] \sqcup' l_2.lVars[i] \sqsubseteq' l'.lVars[i]$, also $l.lVars[i] \sqsubseteq l'.lVars[i]$.
  Demnach ist $l.lVars[i] = l'.lVars[i]$.
- Für die Elemente des Stapels kommt wegen H2 und O2 auf ähnliche Weise zu dem Ergebnis
  $l.stack[i] = l'.stack[i]$.

Somit sind l' und l gleich, d. h. $l = l_1 \sqcup l_2$ ist tatsächlich die kleinste obere Schranke.

Es bleibt der Nachweis, dass $\top$ und $\bot$ tatsächlich die kleinste obere bzw. größte untere Schranke von L bilden.

Angenommen, es gäbe ein Element l mit $\top \sqsubseteq l$. Dann gilt:

- wegen H1: Es muss gelten $\#\top.stack \leq \#l.stack$. Da aber die Größe des Stapels von $\top$ bereits maximal gewählt wurde muss demnach auch $\#l.stack = maxStack = \#\top.stack$ sein.
- wegen H2 und H3: Für alle $0 \leq i < \#l.lVars$ gilt $\top.lVars[i] \sqsubseteq' l.lVars[i]$. Andererseits haben die lokalen Variablen von $\top$ bereits den maximalen Wert $\top'$, so dass auch die Variablen von l den gleichen Wert haben müssen. Gleiches gilt auch für die Stapelworte 0 bis $\#l_1.stack - 1$.

Somit gilt $l = \top$ für alle $\top \sqsubseteq l$, d. h. $\top$ ist maximal. Analog ist $\bot$ das kleinste Element von L, da die Stapelgröße minimal gewählt wurde (0), genauso wie die einzelnen Datenwörter ($\bot'$).

Dass es sich auch um die *kleinste* obere bzw. *größte* untere Schranke handelt folgt sofort daraus, dass $\top$ und $\bot$ in L enthalten sind.

Somit wird die erste Terminierungsbedingung (vollständiges Gitter) von dem so definierten Zustandsraum L erfüllt.

Die zweite Bedingung, die aufsteigende Kettenbedingung, wird von L ebenfalls erfüllt, wenn dies für den Eigenschaftsraum L' der Fall ist. Betrachtet man eine aufsteigende Kette $l_1 \sqsubseteq l_2 \sqsubseteq l_3 \sqsubseteq \ldots$, so muss sich ein n finden lassen, ab dem die Stapelgröße nicht mehr wächst. Schließlich ist diese Größe für jede Methode begrenzt.

Für jede lokale Variable $l_z.lVars[i]$ und für jedes Stapelwort $l_z.stack[j]$ lassen sich ebenfalls Zahlen n_{zi} bzw. n_{zj} finden ab denen gilt $l_{n_{zi}}.lVar[i] = l_{n_{zi}+1}.lVar[i] = \ldots$ bzw. $l_{n_{zj}}.stack[j] = l_{n_{zj}+1}.stack[j] = \ldots$, da der Eigenschaftsraum die aufsteigende Kettenbedingung erfüllt.

Dadurch stabilisiert sich aber auch die Kette $l_1 \sqsubseteq l_2 \sqsubseteq l_r \sqsubseteq \ldots$ ab dem Glied $\max(\max_i n_{zi}, \max_j n_{zj})$.

A.2 Aus Teilfunktionen zusammengesetzte Transferfunktionen

Im Folgenden soll dargestellt werden, welche Anforderungen die in Abschnitt 3.2.3 vorgestellten Transferfunktionen erfüllen müssen, damit die in Abschnitt 3.1.4 genannte Anwendbarkeitsbedingung erfüllt wird.

Es werden nun zwei Elemente $z, z' \in Z$ betrachtet, die zum einen die selbe Stapelgröße aufweisen und für die des Weiteren gilt $z \sqsubseteq z'$. Da die Position von Parametern und Ergebnissen in den lokalen Variablen und auf dem Stapel allein durch den Bytecode des Befehls bestimmt wird, muss auch die Zuordnung von Datenwörtern im Ausgangszustand zu Operanden, sowie Wörtern des Endzustands zu Ergebnissen an dieser Stelle nicht weiter betrachtet werden. Schließlich ist die $\sqsubseteq$ Relation so definiert, dass jeweils Elemente, die an der selben Position im Zustand stehen, verglichen werden. Somit ist es irrelevant, an welcher Stelle das Ergebnis

$f_i(e_1, \ldots, e_n)$ oder seine Operanden $e_1, \ldots, e_n$ stehen, so lange die Position in beiden Zuständen übereinstimmt.

Sind nun also $e_1, \ldots, e_n$ die Operanden des Befehls im Zustand z und $e'_1, \ldots, e'_n$ diejenigen im Zustand z', so muss gelten: $e_1 \sqsubseteq_E e'_1, e_2 \sqsubseteq_E e'_2, \ldots, e_n \sqsubseteq_E e'_n$, da $z \sqsubseteq z'$ gilt,

Außerdem wird gefordert, dass $f(z) \sqsubseteq f(z')$ gilt, so dass auch für alle Ergebnisse gelten muss $f_i(e_1, \ldots, e_n) \sqsubseteq_E f_i(e'_1, \ldots e'_n)$.

Dies ist offensichtlich dann der Fall, wenn alle Ergebnisfunktionen monoton sind in dem Sinne, dass gilt

$$e_1 \sqsubseteq_E e'_1, \ldots, e_n \sqsubseteq_E e'_n \Longrightarrow f_i(e_1, \ldots, e_n) \sqsubseteq_E f_i(e'_1, \ldots, e'_n)$$

Somit erfüllt also eine Transferfunktion, die aus mehreren Teilfunktionen zur Berechnung der Ergebnisse aufgebaut ist, die Anwendbarkeitsbedingung, wenn jede dieser Ergebnisfunktionen monoton in ihren Parametern ist. Allerdings gilt dies nur, so lange die Position und Anzahl von Parametern und Ergebnissen vom Zustand unabhängig ist.

A.3 Eigenschaftsraum für die Analyse der Typsicherheit

In diesem Abschnitt soll grob skizziert werden, dass es sich bei dem unter 4.3.1 vorgestellten Eigenschaftsraum um ein vollständiges Gitter handelt. Dazu muss gezeigt werden, dass zu jeder Teilmenge eine kleinste obere und größte untere Schranke existiert.

Der Eigenschaftsraum kann in zwei Teile zerlegt werden: Einmal die Menge der einfachen Datentypen, wie `int`, `float` usw., zum anderen die Menge der Referenzen unterschiedlicher Klassen $\texttt{ref}(C)$.

Die Teilmenge der einfachen Datentypen ist ein vollständiges Gitter, da die Elemente (mit Ausnahme von $\top$ und $\bot$) untereinander unvergleichbar sind und mit $\top$ und $\bot$ eine kleinste obere bzw. größte untere Schranke gegeben ist.

Die Menge der Referenzen kann wiederum in drei Gruppen unterteilt werden: Die Menge der Referenzen denen eine Klasse (die kein Array ist) zugeordnet werden kann, die Referenzen, denen ein oder mehrere Interfaces zugeordnet werden und diejenigen, die auf Arrays verweisen.

Auf der Menge der Referenzen mit Klassenzuordnung $\texttt{ref}(C)$ ist die Relation $\sqsubseteq$ definiert, die die Ableitungsverhältnisse der Klassen wiedergibt: Es gilt $\texttt{ref}(C) \sqsubseteq \texttt{ref}(C')$, wenn es sich bei C entweder um die Klasse C' oder um eine ihrer Unterklassen handelt Diese Relation ist per Definition reflexiv, transitiv und antisymmetrisch (da in Java jede Klasse nur von genau einer anderen Klasse abgeleitet sein darf), also eine Halbordnung. Außerdem ist jede Klasse zumindest von $\texttt{ref}(\textit{java.lang.Object})$ abgeleitet, d. h. es exisitiert auf jeden Fall eine kleinste obere Schranke zu jeder Teilmenge von Referenzen.

Für die Menge der Interfacereferenzen $\texttt{ref}(I_1, \ldots, I_n)$ gibt die $\sqsubseteq$ Relation ebenfalls die Ableitungsverhältnisse wieder: $\texttt{ref}(I_1, \ldots, I_n) \sqsubseteq \texttt{ref}(J_1, \ldots, J_m)$, wenn für jedes I_k ein J_l existiert, von dem I_k abgeleitet wurde. Die kleinste obere Schranke lässt sich nun finden, wie in Abschnitt 4.3.1 beschrieben: Um zwei Referenzen $\texttt{ref}(I_1, \ldots, I_n)$ und $\texttt{ref}(I'_1, \ldots, I'_n)$ zu vereinigen müssen zunächst alle I_k und I'_l paarweise vereinigt werden. Aus der Menge der Ergebnisse werden dann alle Interfaces gestrichen, die durch ein anderes Interface der Menge erweitert werden. Das Resultat ist offensichtlich die kleinste obere Schranke bzgl. $\sqsubseteq$.

Mit der Null-Referenz `ref`$(null)$ ist auch eine größte untere Schranke für die Menge der Klassen- und Interfacereferenzen gegeben. Damit sind beide Mengen jeweils vollständige Gitter.

Für die Menge der Arrays gilt, dass nur Arrays gleicher Dimension vereinigt werden können, indem der Basistyp der Ergebnisreferenz auf die Vereinigung der Basistypen beider Referenzen gesetzt wird. Anderenfalls ergibt die Vereinigung eine Referenz der Klasse `ref`$(java.lang.Object)$. Da es sich bei den Basistypen um ein vollständiges Gitter handelt, gilt das auch für die Menge der Arrays.

Damit ist auch die Menge aller Referenzen ein vollständiges Gitter, da es sich um die Vereinigung dreier Gitter handelt, für die mit $\sqcup$ zu jeder Teilmenge eine kleinste obere Schranke und mit `ref`$(null)$ eine kleinste untere Schranke gegeben ist.

Außerdem sind alle anderen einfachen Datentypen mit Referenzen unvergleichbar. Für eine Menge, die Referenzen und einfache Datentypen enthält ist die kleinste obere Schranke somit $\top$, die kleinste untere $\bot$. Insgesamt ergibt sich somit wieder ein vollständiges Gitter, da jede Teilmenge eine kleinste obere und größte untere Schranke besitzt.

Das Gitter erfüllt auch die aufsteigende Kettenbedingung, da nur endlich viele Elemente enthalten sind.

B Implementierungsbeschreibung

B.1 Beschreibung der Packages

Generische Datenflussanalyse (`jx.verifier` und`jx.verifier.bytecode`)

Im Package `jx.verifier` befinden sich alle Klassen, die zur Durchführung einer Datenflussanalyse wie in Abschnitt 3.2 beschrieben, benötigt werden.

Um eine neue DFA zu implementieren, genügt es die folgenden abstrakten Klassen zu implementieren:

- `JMVLocalVarsElement`: Die Klasse dient zur Verwaltung einzelner lokaler Variablen die im Zustand gespeichert werden können. Es muss die Vereinigungsoperation `merge` implementiert werden, die der $\sqcup$ Operation entspricht.
- `JVMOpStackElement`: Die Klasse verwaltet einzelne Stapelwörter. Es muss wiederum die `merge` Methode implementiert werden.
- `JVMState`: Diese Klasse repräsentiert die Zustände des Zustandsraumes. Es müssen zwei Methoden implementiert werden:
  - `doExecuteNextBC` hat die Aufgabe die Transferfunktion für den nächsten auszuführenden Befehl zu bestimmen und sie auszuführen.
  - `exceptionThrown` bekommt als Parameter die Klasse einer Ausnahme und die Adresse der zugehörigen Behandlungsroutine. Die Methode muss den Stapel des Zustandes leeren und ein Datenwort mit den Eigenschaften für ein Objekt der Ausnahmeklasse ablegen.

Die Datenflussanalyse wird in der Klasse `MethodVerifier` durchgeführt, die als Argument die zu verifizierende Methode erwartet.

Zusätzlich zu den oben genannten Klassen für die DFA existiert die Klasse `ClassTree`, die den Ableitungsbaum der geladenen Klassen enthält.

Um eine neue Instanz des Verifiers zu erzeugen muss ein Objekt vom Typ `jx.verifier.Verifier` erzeugt werden. Der Konstruktor erwartet zwei Argumente: Einen `jx.classstore.ClassFinder`, der alle geladenen Klassen enthält, sowie eine `Enumeration` aller Klassen, die verifiziert werden sollen.

Alle weiteren Packages (mit Ausnahme von `jx.verifier.fla`) bestehen im wesentlichen aus den abgeleiteten DFA-Klassen. Die entsprechenden Klassen werden im Folgenden nicht weiter aufgezählt.

Standard Bytecode Verifikation (`jx.verifier.typecheck`)

Das Paket enthält die Implementierung des Standard Java Bytecodeverifiers. Die Klasse `Typecheck` dient zum Aufruf dieses Verifikationsschrittes und instantiiert einen entsprechenden `MethodVerifier`. In `BCStackEffect` sind die Transferfunktionen der Befehle untergebracht.

Null-Pointer Analyse (`jx.verifier.npa`)

Das Paket implementiert die Null-Pointer Analyse aus Kapitel 5. Die Hauptklasse, die zum Aufruf der NPA-Analyse dient, ist `NullPointerAnalysis`. Die Transferfunktionen liegen in der Klasse `NPABCEffect`. Die Ergebnisse der Analyse werden in Objekten der Klasse `NPAResult` gespeichert.

WCET Analyse (`jx.verifier.wcet`)

Das Paket implementiert die WCET-Analyse. Mit der Klasse `WCETAnalysis` wird ein entsprechender Analyselauf gestartet. Des Weiteren existieren Klassen für den modifizierten Kontrollflussgraph (Präfix `CFG`), sowie eine Klasse für den Simulationszustand, `SimState`.

Zur Implementierung des Orakels steht das Interface `ValueProvider` zur Verfügung.

Die Bestimmung der Laufzeit einzelner Befehle erfolgt über das Interface `ExecutionTime`. Eine einfache Implementierung ist in der Klasse `SimpleExecutionTime` zu finden.

Die Ergebnisse werden in Objekten der Klasse `WCETResult` untergebracht.

Final und Leaf Analyse (`jx.verifier.fla`)

Die beiden einfachen Analysen sind zusammen in der Klasse `FinalAndLeafAnalysis` untergebracht. Die Ergebnisse werden in Objekten vom Typ `FLAResult` gespeichert.

B.2 Mechanismus zur Ergebnisübergabe

Zur Übergabe von Ergebnissen des Verifiers an den Compiler bzw. um Ergebnisse zu sichern wurde die Klasse `jx.classfile.MethodData` um die Methode `getVerifyResult(int type)` erweitert.

Diese Methode erwartet als Parameter einen Schlüsselwert, der angibt welches Ergebnis tatsächlich benötigt wird. Zurückgeliefert wird ein Objekt aus einer Unterklasse von `jx.classfile.VerifierResult`. Die Schlüssel sind ebenfalls in `jx.classfile.VerifierResult` definiert. Es existieren folgende Schlüsselwerte:

- `NPA_RESULT`: Ein Objekt vom Typ `jx.verifier.npa.NPAResult` wird zurückgeliefert. Es enthält Informationen darüber, an welcher Stelle der Methode eine Laufzeitüberprüfung entfallen kann.

- `FLA_RESULT`: Ein Objekt vom Typ `jx.verifier.fla.FLAResult` wird zurückgeliefert. In dem Objekt ist gespeichert, ob es sich um eine Leaf bzw. System-Final Methode handelt.

- `WCET_RESULT`: Ein Objekt vom Typ `jx.verifier.wcet.WCETResult` wird zurückgeliefert. Dieses enthält die obere Laufzeitschranke, falls sie ermittelt werden konnte. Anderenfalls enthält es einen Verweis auf die modifizierte Methode (vlg. Abschnitt 6.4).

Literaturverzeichnis

[ASU88] Aho, Alfred, Ravis Sethi und Jeffrey Ullman: *Compilers: Principles, Techniques, and Tools.* Addison-Wesley, Readings, Massachusetts, 1988.

[Bla99] Blanchet, Bruno: *Escape Analysis for Object Oriented Languages. Application to Java.* OOPSLA, Denver, 1999.

[Bli95] Blieberger, Johann: *Discrete Loops and worst case Performance.* Computer Languages, 20(3), 1995.

[Bli00] Blieberger, Johann: *Data-Flow Framework for Worst-Case Execution Time Analysis.* Technischer Bericht, Department of Computer-Aided Automation, Technical University of Vienna, 2000.

[EHM96] Erdos, Marlena, Bret Hartmann und Marianne Mueller: *Security Reference Model for the Java Developer's Kit 1.0.2.* http://www.javasoft.com/security/SRM.html, November 1996.

[Fel01] Felser, Meik: *Integration einer Mehrprozessor Unterstützung in das Java-Betriebsystem JX.* Diplomarbeit, Universität Erlangen, Institut für Informatik IV, Juli 2001.

[GJS96] Gosling, James, Bill Joy und Guy Steele: *The Java Language Specification.* The Java Series. Addison Wesley Longman, Reading, Massachusetts, 1996.

[LY97] Lindholm, Tim und Frank Yellin: *The Java Virtual Machine Specification.* The Java Series. Addison Wesley Longman, Reading, Massachusetts, 1997.

[MG01] Michael Golm, Jürgen Kleinöder, Frank Bellosa: *Beyond Address Spaces – Flexibility, Performance, Protection and Resource Management in the Type-Safe JX Operating System.* Technischer Bericht TR-I4-01-04, Universität Erlangen, Institut für Informatik IV, 2001.

[Nip] Nipkow, Tobias: *Verified Bytecode Verifiers.* Technischer Bericht, Fakultät für Informatik, Technische Universität München.

[NNH99] Nielson, Flemming, Hanne Riis Nielson und Chris Hankin: *Principles of Program Analysis.* Springer-Verlag, Berlin, Heidelberg, New York, 1999.

[NP93] Nirke, Vivek und William Pugh: *A partial evaluator for the Maruti hard real-time system.* The Journal of Real-Time Systems, 5:13–33, 1993.

[PK89] Puschner, P. und C. Koza: *Calculating the maximum execution time of real-time programs.* The Journal of Real-Time Systems, 1:159–176, 1989.

[Qui99] Quian, Zhenyu: *A Formal Specification of Java Virtual Machine Instructions for Objects, Methods and Subroutines.* In: Alves-Foss, Jim (Herausgeber): *Formal Syntax and Semantics of Java*, Band 1523 der Reihe *LNCS*. Springer, 1999.

[Sch95] Schöning, Uwe: *Theoretische Informatik – kurzgefaßt.* Spektrum Akademischer Verlag, Heidelberg, Berlin, Oxford, 2. Auflage, 1995.

[vO] Oheimb, David von: *Hoare Logic for Mutual Recursion and Local Variables.*

[vON99] Oheimb, David von und Tobias Nipkow: *Machine-checking the java Specification: Proving Type-Safety.* In: Alves-Foss, Jim (Herausgeber): *Formal Syntax and Semantics of Java*, Band 1523 der Reihe *LNCS*. Springer, 1999.

[Waw01] Wawersich, Christian: *Design und Implementierung eines Profilers und optimierenden Compilers für das Betriebssystem JX.* Diplomarbeit, Institut für Informatik IV, Universität Erlangen, April 2001.

[WR] Whaley, John und Martin Rinard: *Compositional Pointer and Escape Analysis for Java Programs.*

[Yel97] Yellin, Frank: *Low Level Security in Java.* http://www.javasoft.com/sfaq/verifier.html, September 1997.