Java – formal fundiert*

David von Oheimb und Cornelia Pusch

Fakultät für Informatik, Technische Universität München
`http://www.in.tum.de/~oheimb/`
`http://www.in.tum.de/~pusch/`

Zusammenfassung Dieser Artikel gibt eine Übersicht über das Projekt Bali zur formalen Behandlung möglichst vieler Aspekte von Java. Die Arbeiten umfassen bisher eine formale Semantik großer Teile der Java-Quellsprache und des Bytecodes, jeweils zusammen mit einem Beweis der Typsicherheit. Als Spezifikations- und Verifikationswerkzeug dient Isabelle/HOL. Wir beschreiben die Ziele dieses Projekts und die grobe Vorgehensweise, geben einen knappen Einblick in die Formalisierung und die bewiesenen Aussagen, und stellen unsere bisherigen Ergebnisse und Erfahrungen dar.

1 Einführung

Java ist eine inzwischen weit verbreitete Programmiersprache, bei der die Betrachtung verschiedener Sicherheitsaspekte große Aufmerksamkeit erfährt. Neben den üblichen Problemen der Programmverifikation und Compilerkorrektheit spielen die Typsicherheit und die Integrität von übersetzten Programmen eine große Rolle. Im Rahmen unseres DFG-Forschungsprojekts *Bali* [NOP98] behandeln wir solche Fragestellungen formal, und zwar mit Unterstützung des maschinellen Beweissystems *Isabelle/HOL* [Pau94].

Ziel unseres Projekts ist es, möglichst große Teile der Sprache Java selbst, eines Compilers, des Bytecodes und der Ablaufumgebung *Java Virtual Machine (JVM)* zu formalisieren, ihre Eigenschaften zu untersuchen und verläßlich zu beweisen. Die Resultate dieser Arbeit finden im wesentlichen in folgenden Bereichen Anwendung:

- als Referenzspezifikation für Programmierer und Compilerbauer, die präziser und übersichtlicher ist als die offizielle Definition der Sprache und der JVM
- zur Überprüfung des Designs der Sprache und der Ablaufumgebung (auch von geplanten Erweiterungen), um Fehler und Verallgemeinerungsmöglichkeiten aufzuzeigen
- als (teilweiser) Korrektheitsbeweis für die Implementierung von Compilern und Ablaufumgebungen
- als Basis für die Verifikation von Java-Programmen, d.h. den Nachweis, daß sie ihre Spezifikation erfüllen

* gefördert durch das DFG-Projekt *Bali*

Herausragendes Merkmal unseres Ansatzes ist Präzision und Verläßlichkeit, die wir durch die maschinengestütze Verifikation erreichen. Andere uns bekannte Arbeiten, die ähnlich große Teile von Java mit dem Anspruch mathematischer Präzision behandeln[DE98,BS98], erweisen sich leider als im Detail lückenhaft. Es liegt wohl in der Natur der Sache, daß einem bei umfangreichen Arbeiten auf Papier trotz sorgfältiger Arbeitsweise mit hoher Wahrscheinlichkeit zumindest Flüchtigkeitsfehler unterlaufen. Besonders problematisch sind in diesem Zusammenhang nachträgliche Erweiterungen und Korrekturen der Formalisierung und der zugehörigen Beweise. Einem maschinellen Beweissystem dagegen entgehen solche Fehler nicht [Sym98]. Mit Isabelle/HOL verwenden wir ein leistungsfähiges und flexibles Werkzeug, das sowohl eine ausdrucksstarke Spezifikationssprache als auch mächtige und sichere Beweisverfahren bietet.

Diese Arbeiten sind nicht nur aus theoretischer Sicht interessant: Es zeigt sich der Bedarf an einer formalen Fundierung von Java in der Praxis immer deutlicher. Dies betrifft vor allem sicherheitskritische Anwendungen wie z.B. Smart Cards, deren Hersteller an einer formalen Überprüfung ihrer Implementierungen interessiert sind. Wie die Anforderungen an solch eine Zertifizierung auszusehen haben, wird derzeit noch erarbeitet, und auch hier kann unser Projekt einen wertvollen Beitrag leisten.

1.1 Behandelter Sprachumfang

Unsere Formalisierung von Java und der JVM beinhaltet die unserer Ansicht nach zentralen Aspekte der objektorientierten Programmierung:

- Klassen- und Interface-Deklarationen mit Feldern und Methoden,
- statische Klassen-Initialisierung,
- Subklassen-, Subinterface- und Implementierungsrelation mit Vererbung, Überschreibung und Verschattung
- Methodenaufrufe mit statischer Überladung und dynamischer Bindung
- einige primitive Typen, Objekte (incl. Arrays)
- Exception-Erzeugung und -Behandlung

Aus Vereinfachungsgründen behandeln wir keine Packages und andere Aspekte der getrennten Übersetzung. Vorläufig behandeln wir auch noch keine Sichtbarkeitsbeschränkungen von Namen und keine parallele Programmausführung. Bei vielen Konstrukten konnten wir die Komplexität verringern, ohne ihre Ausdrucksstärke zu beeinträchtigen.

1.2 Aktueller Fokus: Typsicherheit

Zunächst haben wir uns der *Typsicherheit* angenommen, einer Spracheigenschaft, die auf Quell- und ebenso auf Bytecode-Ebene wesentlich ist. Sie besagt, daß in wohlgetypten Programmen keine Laufzeitfehler auftreten können derart, daß Operationen Argumente eines falschen Typs erhalten. Wo die Wohlgetyptheit statisch nicht entscheidbar ist (z.B. bei Typkonvertierungen und Feld-Zuweisungen in Java), muß zumindest eine geeignete Laufzeit-Überprüfung erzeugt werden, die bei Verletzung der Typkonsistenz eine Ausnahmebehandlung erzwingt. Die Typsicherheit erfordert also das korrekte Zusammenspiel von Typsystem, Wohlgeformtheitsbedingungen und Programmablauf. Eine hinreichende Bedingung hierfür ist die *Korrektheit des Typsystems (type soundness)*, die verlangt, daß für wohlgetypte Programme der dynamische Typ eines jeden Wertes, der beim Programmablauf erzeugt wird, zum statischen Typ des entsprechenden Programmteils paßt. Sie sollte auf beiden Sprachebenen gelten:

- Auf Quellsprachen-Ebene bedeutet sie, daß statisch erkennbare Typfehler durch ein geeignetes Typsystem tatsächlich statisch ausgeschlossen werden. Wegen möglicher Übersetzungsfehler ergibt dies zwar noch keine Ablaufsicherheit, ist aber wesentlich für eine disziplinierte Programmentwicklung und stellt daher ein wichtiges Prüfkriterium für das Sprachdesign dar.
- Bytecode-Programme, besonders solche, die von einem nicht vertrauenswürdigen Server geladen wurden, stammen nicht notwendigerweise von einem korrekten Java-Compiler. Die Frage der Wohlgetyptheit und Typsicherheit stellt sich deshalb auf dieser Ebene neu. Sie hat hier sogar besondere Brisanz, weil sie einen entscheidenden Teil des Sicherheitskonzepts von Java darstellt. Deshalb enthält die JVM einen *Bytecode Verifier*, der den Bytecode vor seiner Ausführung auf Wohlgetyptheit überprüft, und dessen Korrektheit in diesem Zusammenhang zu untersuchen ist.

1.3 Designziele der Formalisierung

Bei der Entwicklung unserer Formalisierung verfolgen wir als Designziele:

- gute Lesbarkeit durch Einfachheit und Übersichtlichkeit
- leichte Validierung durch Nähe zur offiziellen Spezifikation von Java [GJS96] und der JVM [LY96]
- Änderungsfreundlichkeit und Erweiterbarkeit
- Angemessenheit für maschinengestütztes Beweisen

Im folgenden wollen wir einen Eindruck davon vermitteln, auf welche Weise wir diese Ziele bisher erreicht haben. Dazu gehen wir in Abschnitt 2 auf unsere Formalisierung der Java-Quellsprache und den Beweis ihrer Typsicherheit ein. Abschnitt 3 behandelt die JVM, insbesondere den Bytecode Verifier und seine Korrektheit. Schließlich stellen wir in Abschnitt 4 unsere bisher gewonnenen Ergebnisse und Erfahrungen dar.

2 Formale Behandlung von Java

In diesem Abschnitt geben wir einen Überblick über BALI, die Formalisierung der von uns behandelten Teilsprache von Java. Dabei werden hier nur wesentliche Punkte dargestellt und exemplarisch erklärt. Details sind in [ON98] und unserer Online-Dokumentation[NOP98] zu finden.

2.1 Abstrakte Syntax

Programme stellen wir als Listen von Klassen- und Interface-Deklarationen dar, die in Isabelle/HOL mittels der Typdefinition

$prog = (cdecl)\,list \times (idecl)\,list$

eingeführt werden. Entsprechendes gilt für Klassen, Interfaces, Felder und Methoden. Wie haben die Listenrepräsentation einer Mengendarstellung vorgezogen, weil sie a priori endlich und operational orientiert ist.

Anweisungen, Ausdrücke und Variablen werden durch rekursive Datentypen dargestellt, die ihre abstrakte Syntax direkt beschreiben:

$$\begin{array}{lll} stmt = \texttt{throw}\ (expr) & expr = \texttt{new}\ tname & var\ = ename \\ \quad\mid stmt;\ stmt & \quad\mid var\texttt{:=}expr & \quad\mid \{ty\}expr.\,ename \\ \quad\mid \dots & \quad\mid \dots & \quad\mid \dots \end{array}$$

Eine Besonderheit sind hierbei sogenannte *Typannotationen* $\{\dots\}$ bei Feldzugriffen und Methodenaufrufen, die eigentlich nicht zur Quellsprache gehören. Sie stellen vom Compiler während der Typprüfung eingesetzte Informationen dar, die für die statische Bindung von Feldern und die Auflösung von überladenen Methoden nötig sind.

2.2 Typsystem

Wir definieren explizit die primitiven Typen `boolean` und `int` sowie alle Arten von Referenztypen $ref\_ty = \mathsf{NT} \mid \mathsf{Iface}\ tname \mid \mathsf{Class}\ tname \mid ty[]$ von Java und die Relationen zwischen ihnen. Die wichtigste davon ist die *Widening*-Ordnung, in Zeichen $\preceq$, eine Art syntaktische Subtyp-Relation. Die Formel $\Gamma \vdash S \preceq T$ bedeutet, daß im Kontext Γ jeder Wert vom Typ S auch als Wert vom Typ T aufgefaßt werden darf, also (bei Referenztypen) über eine syntaktisch kompatible Menge von Feldern und Methoden verfügt. Die Typrelationen werden induktiv definiert, beispielsweise wie folgt, wobei $\Gamma \vdash C \rightsquigarrow I$ dafür steht, daß die Klasse C das Interface I implementiert:

$$\frac{\mathsf{is\_type}\ \Gamma\ T}{\Gamma \vdash T \preceq T} \qquad \frac{\Gamma \vdash C \rightsquigarrow I}{\Gamma \vdash \mathsf{Class}\ C \preceq \mathsf{Iface}\ I} \qquad \frac{\Gamma \vdash \mathsf{RefT}\ S \preceq \mathsf{RefT}\ T}{\Gamma \vdash (\mathsf{RefT}\ S)[] \preceq (\mathsf{RefT}\ T)[]} \qquad \dots$$

Darauf aufbauend können auch die Typregeln für Terme in sehr natürlicher Weise als induktive Relationen definiert werden. Diese Regeln enthalten nebenbei auch alle sonstigen lokalen Wohlgeformtheitsbedingungen. Hier als Beispiel die Typisierung von Array-Erzeugung und Feldvariablen-Zugriff:

$$\frac{\mathsf{is\_type}\ \Gamma\ T \quad \Gamma,\Lambda \vdash i{::}\texttt{int}}{\Gamma,\Lambda \vdash (\texttt{new}\ T[i]){::}T[]} \qquad \frac{\Gamma,\Lambda \vdash e{::}\mathsf{Class}\ C \quad \mathsf{cfield}\ \Gamma\ C\ fn = \mathsf{Some}\ (T,fT)}{\Gamma,\Lambda \vdash (\{T\}e.fn){::}fT}$$

Dabei bedeutet $\Gamma,\Lambda \vdash e{::}T$, daß im Kontext Γ,Λ der Ausdruck e wohlgeformt ist und den Typ T hat. Entsprechendes gilt für die Wohlgetyptheit von Anweisungen $\Gamma,\Lambda \vdash s{::}\diamond$. Übrigens läßt sich leicht zeigen, daß die Typisierung eindeutig ist.

2.3 Wohlgeformtheit

Für alle Arten von Deklarationen gibt es entsprechend den Regeln von Java eine Menge von globalen Wohlgeformtheitsbedingungen, die wir direkt als Prädikate über die Deklarationen formulieren. Zum Beispiel ist eine Methoden-Deklaration mit Signatur *sig* wohlgeformt, wenn ihr Methoden-Kopf *mh* wohlgeformt ist, die lokalen Variablen *lvars* eindeutige Namen und korrekte Typen T haben, usw.:

$$\begin{aligned}\mathsf{wf\_mdecl}\ \Gamma\ C\ (\mathit{sig},\mathit{mh},\mathit{lvars},\mathit{blk},\mathit{res}) \stackrel{\mathsf{def}}{=}\ & \mathsf{wf\_mhead}\ \Gamma\ (\mathit{sig},\mathit{mh}) \wedge \\ & \mathsf{unique}\ \mathit{lvars} \wedge (\forall (\mathit{vn},T) \in \mathsf{set}\ \mathit{lvars}.\ \mathsf{is\_type}\ \Gamma\ T) \wedge \ldots\end{aligned}$$

Für die dynamische Semantik dürfen wir die Wohlgeformtheit voraussetzen.

2.4 Semantik

Wir beschreiben die Wirkung der Ausführung von Programmtermen auf den Zustand auf operationelle Weise. Diese ablauforientierte Sicht ist (im Vergleich zu einer denotationellen oder axiomatischen Semantik) für den Anwender die natürlichste. Speziell haben wir eine *Auswertungs*semantik gewählt, die eine direkte Umsetzung der offiziellen Sprachspezifikation ermöglicht, weil sie für jeden Term die Beziehung zwischen den Zuständen vor und nach seiner Ausführung (bzw. Auswertung) darstellt und sie auf die Zustandsübergänge der jeweiligen Teilterme zurückführt. Daher können wir mit ihr abstrakter argumentieren als mit einer *Transitions*semantik, die die Ausführung in atomaren Einzelschritten beschreibt, was andererseits die Darstellung der verzahnt parallelen Ausführung von Threads erlauben würde.

Den Programmzustand repräsentieren wir durch ein Paar aus der eventuell aktiven Exception und dem Zustand im engeren Sinne, nämlich die aktuellen lokalen Variablen sowie globale Werte, speziell der Heap, der Referenzen auf Objekte abbildet. Durch die Auswertungssemantik ist kein expliziter Methodenaufruf-Stack nötig. Wir behandeln die dynamische Erzeugung von Objekten (inklusive möglichem Speicherüberlauf), aber keine Garbage Collection.

Unsere operationelle Semantik gibt für jede Art von Programmterm genau eine Regel an, die Zustandsübergänge bei der Ausführung spezifiziert. Dabei bedeutet z.B. $\Gamma \vdash \sigma\ -e{\triangleright}v{\rightarrow}\ \sigma'$, daß im Programm Γ der Ausdruck e zu einem Wert v ausgewertet wird, wobei sich der Zustand σ zu σ' verändert. Die Regeln sind dadurch relativ übersichtlich, daß auftretende Exceptions weitgehend implizit propagiert werden. Beispielsweise beschreibt die folgende Regel sehr kompakt die Bedeutung des lesenden Zugriffs auf eine Feldvariable *fn* im normalen (Exception-freien) Anfangszustand $\mathsf{Norm}\ \sigma_0$: Nach der Initialisierung – falls nötig – der Klasse C, in der das Feld deklariert wurde, und der Auswertung des Ziel-Ausdrucks e zur Adresse a wird im Zwischenzustand (x_2,σ_2) der Wert v aus dem Objekt extrahiert, das durch $\mathsf{the}\ (\mathsf{heap}\ \sigma_2\ a)$ bezeichnet ist. Falls a' eine Null-Referenz war, wird mittels $\mathsf{np}\ a'\ x_2$ im Endzustand eine `NullPointer`-Exception ausgelöst und der gelesene Wert ignoriert.

$$\frac{\begin{array}{c}\Gamma \vdash (\mathsf{Norm}\ \sigma_0)\ -\mathsf{init}\ C{\rightarrow}\ \sigma_1 \qquad \Gamma \vdash \sigma_1\ -e{\triangleright}a'{\rightarrow}\ (x_2,\sigma_2) \qquad a = \mathsf{the\_Addr}\ a' \\ v = \mathsf{the}\ (\mathsf{snd}\ (\mathsf{the}\ (\mathsf{heap}\ \sigma_2\ a))\ (\mathsf{Inl}\ (\mathit{fn},C)))\end{array}}{\Gamma \vdash (\mathsf{Norm}\ \sigma_0)\ -(\{C\}e.\mathit{fn}){\triangleright}v{\rightarrow}\ (\mathsf{np}\ a'\ x_2,\sigma_2)}$$

2.5 Beweis der Typsicherheit

Um die Korrektheit des Typsystems formulieren zu können, verwenden wir mehrere Hilfskonstrukte, die im Begriff der Konformität eines Zustandes zum Kontext münden: $\sigma ::\preceq \Gamma,\Lambda$ bedeutet, daß die Werte aller Variablen im Zustand σ zum jeweiligen (statischen) Typ im Kontext Γ,Λ passen. Damit lautet die Korrektheitsaussage für Anweisungen s wie folgt: Für ein wohlgeformtes Programm Γ, wenn s wohlgetypt ist und seine Ausführung den Zustand σ in σ' abbildet und σ konform zum Kontext ist, dann ist es auch σ'. Formal:

$$\mathsf{wf\_prog}\ \Gamma \ \wedge\ \Gamma,\Lambda \vdash s{::}\diamond \ \wedge\ \Gamma \vdash \sigma\ -s\rightarrow\ \sigma' \ \wedge\ \sigma ::\preceq \Gamma,\Lambda \ \longrightarrow\ \sigma' ::\preceq \Gamma,\Lambda$$

Der Beweis wird per Regelinduktion über die Ausführung der Programmterme geführt und basiert auf etwa 300 (nur teilweise aufwendig zu beweisenden) Hilfsaussagen. Als einfache Konsequenz für die Typsicherheit ergibt sich zum Beispiel, daß 'method not understood'-Laufzeitfehler nicht auftreten können.

3 Formale Behandlung der JVM

Im folgenden geben wir einen Überblick über unsere Formalisierung der JVM, für eine komplette Beschreibung siehe [Pus98]. Die Isabelle-Dateien sind unter [NOP98] erhältlich.

Entsprechend der offiziellen JVM-Spezifikation [LY96] beschreiben wir das operationelle Verhalten der JVM-Befehle als Transitionssemantik, wobei wir von einigen Details wie z.B. Auflösung von symbolischen Referenzen und Einbettung des gesamten Codes in einen einheitlichen Adreßraum abstrahieren. Auch das dynamische Laden von Klassen haben wir noch nicht betrachtet. Von den umfassenden Möglichkeiten der Fehlerbehandlung (Exceptions) haben wir auf der Bytecode-Ebene bis jetzt nur einige der vordefinierten Fehlertypen formalisiert, z.B. die Dereferenzierung des Null-Zeigers.

3.1 Operationelle Semantik

In [LY96] wird die operationelle Semantik der JVM nur für Zustände beschrieben, in denen gewisse strukturelle Voraussetzungen erfüllt sind (z.B. Stackgröße bei Stackzugriffen und richtiger Typ der Operanden). Andernfalls ist das Verhalten undefiniert. Diese Partialität wird oft durch bedingte Regeln modelliert. Dieses Vorgehen hat aber den Nachteil, daß aus der Definition nicht direkt ersichtlich ist, ob einerseits alle korrekten Programme bis zum Ende abgearbeitet werden und andererseits die Regeln eindeutig sind.

In unserem Ansatz formalisieren wir das Verhalten der Maschine durch totale Funktionen, wodurch die vollständige Abarbeitung schon per definitionem sichergestellt ist. Undefiniertheit wird durch den nicht näher definierten Wert arbitrary modelliert. Es handelt sich hierbei nicht um einen besonderen Fehler-Wert, der die Undefiniertheit anzeigt, sondern lediglich um einen beliebigen Wert, über den keine Aussage gemacht werden kann.

Ähnlich wie auf der Quellsprachen-Ebene werden Wohlgeformtheitsbedingungen für die Classfiles formuliert, die vor der Ausführung des Bytecodes überprüft werden.

Die operationelle Semantik für den Befehl Getfield, der eine Feldvariable ausliest, sieht in unserer Formalisierung z.B. folgendermaßen aus:

```
exec_mo (Getfield idx) CFS cls hp stk pc =
    (let oref      = hd stk;
         (cl,fs)   = get_Obj (hp !! (get_Addr oref));
         cpool     = get_cpool (CFS !! cls);
         (fc,fn,fd) = extract_Fieldref cpool idx;
         xp'       = if oref=Null then Some NullPointer else None
     in
     (xp', hp, (fs !! (fc,fn))#(tl stk), pc+1))
```

CFS enthält ein JVM-Programm bestehend aus einer Menge von Classfiles. Auf dem Stack *stk* wird eine Referenz zu einem auf dem Heap *hp* gespeicherten Objekt erwartet. Ist der Stack leer, oder ist der oberste Wert keine Adresse, oder befindet sich an der Adresse keine Klasseninstanz, geben die Zugriffsfunktionen hd bzw. get_Addr bzw. get_Obj den Wert arbitrary zurück. Der Index *idx* muß im Constant Pool der aktuellen Klasse *cls* auf einen Fieldref-Eintrag zeigen, der eine Klasse *fc*, einen Feldbezeichner *fn* und einen Feld-Deskriptor *fd* enthält. Wir verwenden das Paar (*fc,fn*) bestehend aus Klasse und Feldbezeichner, um auf das entsprechende Feld zuzugreifen. Die Objekt-Referenz auf dem Stack wird durch den Wert des Feldes ersetzt. Im Falle einer Null-Referenz wird eine Exception ausgelöst. Schließlich wird der Programmzähler *pc* inkrementiert.

3.2 Bytecode Verifier

Die JVM muß sicherstellen, daß der auszuführende Code einer Reihe von Anforderungen genügt, die neben der Initialisierung und dem Zugriff auf Variablen vor allem die Wohlgetyptheit betreffen. Die Überprüfung des Bytecodes wird in [LY96] nur lose spezifiziert; es sind unterschiedliche Implementierungen möglich. Eine Möglichkeit ist, die gesamte Überprüfung erst zur Laufzeit vorzunehmen; dieser Ansatz wird z.B. von Cohen [Coh97] verfolgt. Dieses Vorgehen ist allerdings nicht sehr effizient. Die Beschreibung des Bytecode Verifiers, bei dem der Großteil der Überprüfung vor der Ausführung vorgenommen wird, bezieht sich stark auf die Implementierung der JVM von Sun selbst, und die Grenzen zwischen abstrakterAnforderungsspezifikation und konkreter Implementierung werden nicht deutlich.

Die vollständige statische Überprüfung des Codes hat Qian in seinem Ansatz [Qia98] auf Papier beschrieben. Unsere Formalisierung basiert auf dieser Arbeit, weicht allerdings in einigen Punkten ab, z.B. was die konkrete Darstellung der Typregeln betrifft.

Wie auf der Quellsprachen-Ebene wird ein Typsystem eingeführt. Auf der Bytecode-Ebene besteht ein Programm aus einer Liste von Befehlen. Entsprechend formalisieren wir die Typisierung eines Programms durch eine Funktion Φ :: *p_count* $\Rightarrow$ *instr_type*, die jedem Programmpunkt im Bytecode eine Befehls-Typisierung zuordnet. Diese Befehls-Typisierung beschreibt in einem Paar (*ST*,*LT*) die Typen aller Stackelemente und der lokalen Variablen. Prädikate prüfen ab, ob ein Befehl im Kontext eines Programms und eines Programmtyps wohlgetypt ist. Für den Befehl Getfield sieht dies z.B. folgendermaßen aus:

$$
\begin{array}{l}
\mathsf{wt\_mo}\ (\mathsf{Getfield}\ idx)\ CFS\ cls\ \Phi\ max\_pc\ pc =\\
\quad (\mathsf{let}\ (ST,LT) \;\; = \Phi\ pc;\\
\qquad cpool \qquad = \mathsf{get\_cpool}\ (CFS\ !!\ cls);\\
\qquad (fc,fn,fd) = \mathsf{extract\_Fieldref}\ cpool\ idx\\
\quad\ \mathsf{in}\\
\quad\ \exists rs\ ST'.\ pc{+}1 < max\_pc \wedge\\
\qquad\qquad \mathsf{is\_class}\ CFS\ fc \wedge\\
\qquad\qquad \mathsf{get\_fields}\ (CFS\ !!\ fc)\ (fc,fn) = \mathsf{Some}\ fd \wedge\\
\qquad\qquad ST = \mathsf{Refs}\ rs\ \#\ ST' \wedge\\
\qquad\qquad \mathsf{wideRefsConvertible}\ CFS\ rs\ [\mathsf{CT}\ fc] \wedge\\
\qquad\qquad \Phi\ (pc{+}1) \sqsupseteq ((\mathsf{fd2tys}\ CFS\ fd)\ \#\ ST',\ LT))
\end{array}
$$

Hier wird überprüft, ob der inkrementierte Programmzähler die Code-Länge nicht überschreitet und das Classfile der indizierten Klasse ein dem Zugriff entsprechendes Feld enthält. Weiterhin wird sichergestellt, daß das oberste Stackelement vom richtigen Referenztyp ist und daß der Folgebefehl auf dem Stack ein Element vom Typ des Feldwerts erwartet.

3.3 Korrektheitsbeweis für den Bytecode-Verifier

Die Korrektheitsaussage haben wir folgendermaßen formalisiert und bewiesen:

$$
\begin{array}{l}
\mathsf{wf\_clsfiles}\ CFS \wedge \mathsf{wt\_clsfiles}\ CFS\ \Phi \wedge \mathsf{state\_ok}\ CFS\ \Phi\ \sigma \wedge CFS \vdash \sigma \longrightarrow^* \sigma'\\
\longrightarrow \mathsf{state\_ok}\ CFS\ \Phi\ \sigma'
\end{array}
$$

Das heißt, daß für eine Menge wohlgeformter und vom Bytecode Verifier als statisch wohlgetypt anerkannter Classfiles, ausgehend von einem Zustand σ, der konform zu seinem statischen Typ ist, in allen weiteren Zuständen der Ausführung die Typkonformität gilt.

Aus der Korrektheit des Bytecode Verifiers lassen sich wieder Aussagen über die Typsicherheit folgern.

4 Konklusion

4.1 Ergebnisse

Die bisherigen Ergebnisse unserer Arbeit lassen sich wie folgt zusammenfassen:

- Die Sprache Java ist – jedenfalls in dem von uns behandelten Umfang – sowohl auf Quell- und als auch auf Bytecode-Ebene tatsächlich typsicher. Dies bestätigt entsprechende Behauptungen der Java-Entwickler mit größter praktisch erreichbarer Gewißheit.

– In den Spezifikationen wurden Lücken aufgedeckt und gefüllt, z.B. daß die `throw`-Anweisung eine `NullPointer`-Exception auslösen kann, daß bei jedem Auslösen einer System-Exception diese durch ein jeweils eigenes Exception-Objekt dargestellt wird, und daß die Programmausführung anhält, wenn nicht einmal genug Speicher für eine `OutOfMemory`-Exception vorhanden ist.
– Das Einfließen von Implementierungsdetails in die JVM-Spezifikation wurde aufgedeckt und eliminiert, z.B. die Bezugnahme auf Methodentabellen beim Methodenaufruf.
– In der JVM-Spezifikation auftretende Redundanzen und kleinere Fehler wurden vermieden. So werden z.B. wiederholte Definitionen für den Begriff der *Assignment Conversion* gegeben, die in einem Fall sogar unvollständig ist.
– Verallgemeinerungsmöglichkeiten des Typsystems von Java wurden aufgezeigt: Das Resultat einer Methode, die eine andere Methode überschreibt, darf einen spezielleren Typ haben, die Resultatstypen verschiedener zusammengeführter Methoden mit gleicher Signatur müssen nicht notwendigerweise gleich sein, und als Typ einer Zuweisung darf auch der (i.A. speziellere) Typ des Ausdrucks auf der rechten Seite angesehen werden.

4.2 Statistik

Für die Java-Formalisierung benötigten wir (einschließlich Modifikationen) rund zwei Monate und etwa 1200 Zeilen wohldokumentierter Spezifikationen, und für den Beweis der Typsicherheit mit allen nötigen Lemmata rund drei Monate und etwa 2300 Zeilen Beweisskript.

Auf der Bytecode-Ebene umfassen die Spezifikation für operationelle Semantik und Bytecode Verifier etwa 1700 Zeilen, die Beweise haben eine Länge von knapp 2400 Zeilen. Für Spezifikationen und Verifikation zusammen benötigten wir etwa 6 Monate.

4.3 Erfahrungen

Abschließend unsere bisherigen Erfahrungen mit dem Bali-Projekt:

– Wir sind mit der Ausdrucksstärke und Beweismächtigkeit eines maschinellen Beweissystems wie Isabelle/HOL sehr zufrieden: Alle nötigen Aspekte lassen sich adäquat formalisieren und beweisen.
– Zur Validierung der Formalisierung wäre Werkzeugunterstützung hilfreich, die aus den Spezifikationen ausführbare Prototyp-Programme erzeugt. Die meisten der (kleinen) Formalisierungsfehler wurden allerdings bei der Erstellung der Typsicherheits-Beweise aufgedeckt und behoben.
– Eine möglichst einfache und abstrakte Formalisierung erleichtert die Beweisarbeit sehr. Der Aufwand für Modifikationen läßt sich, wie im Software-Engineering, durch Beachtung des Kapselungsprinzips begrenzen.
– Bei der schrittweisen Erweiterung der Formalisierung verhindert das maschinelle Beweissystem, daß man indirekte, aber notwendige Anpassungen der Spezifikation und der Beweise vergißt.

Somit ergibt sich als Schlußfolgerung: Die voll formale Analyse einer realistischen Programmiersprache, wie sie Java darstellt, ist kein Kinderspiel, sie ist jedoch für Spezialisten mit der heutzutage zur Verfügung stehenden Beweiser-Technologie mit vertretbarem Aufwand machbar.

4.4 Ausblick

Die Aspekte von Java, die wir als nächstes formal behandeln wollen, sind die Compiler- und Programmverifikation, sowie die Konsistenz der geplanten Erweiterung von Java um parametrisierte Typen [AFM97,BOSW98].

Literatur

[AFM97] Ole Agesen, Stephen N. Freund, and John C. Mitchell. Adding type parameterization to the Java language. In *ACM Symp. Object-Oriented Programming: Systems, Languages and Applications*, 1997.

[BOSW98] Gilad Bracha, Martin Odersky, David Stoutamire, and Philip Wadler. Generic Java specification. Draft version, 1998.

[BS98] Egon Börger and Wolfram Schulte. A mathematical definition of the dynamic semantics of Java. In Jim Alves-Foss, editor, *Formal Syntax and Semantics of Java*, volume 1523 of *LNCS*. Springer-Verlag, 1998.

[Coh97] Richard M. Cohen. The defensive Java Virtual Machine specification. Technical report, Computational Logic Inc., 1997. Draft version.

[DE98] Sophia Drossopoulou and Susan Eisenbach. Towards an operational semantics and proof of type soundness for Java. In Jim Alves-Foss, editor, *Formal Syntax and Semantics of Java*, volume 1523 of *LNCS*. Springer-Verlag, 1998.

[GJS96] James Gosling, Bill Joy, and Guy Steele. *The Java Language Specification*. Addison-Wesley, 1996.

[LY96] Tim Lindholm and Frank Yellin. *The Java Virtual Machine Specification*. Addison-Wesley, 1996.

[NOP98] Tobias Nipkow, David von Oheimb, and Cornelia Pusch. Project Bali. 1998. `http://www.in.tum.de/~isabelle/bali/`.

[ON98] David von Oheimb and Tobias Nipkow. Machine-checking the Java specification: Proving type-safety. In Jim Alves-Foss, editor, *Formal Syntax and Semantics of Java*, volume 1523 of *LNCS*. Springer-Verlag, 1998.

[Pau94] Lawrence C. Paulson. *Isabelle: A Generic Theorem Prover*, volume 828 of *LNCS*. Springer-Verlag, 1994.

[Pus98] Cornelia Pusch. Formalizing the Java Virtual Machine in Isabelle. Technical Report TUM-I9816, Institut für Informatik, Technische Universiät München, 1998.

[Qia98] Zhenyu Qian. A formal specification of Java Virtual Machine instructions. In Jim Alves-Foss, editor, *Formal Syntax and Semantics of Java*, volume 1523 of *LNCS*. Springer-Verlag, 1998.

[Sym98] Donald Syme. Proving Java type soundness. In Jim Alves-Foss, editor, *Formal Syntax and Semantics of Java*, volume 1523 of *LNCS*. Springer-Verlag, 1998.