Java – formal fundiert

Tobias Nipkow, David von Oheimb, Cornelia Pusch

Institut für Informatik, TU München

DFG-Projekt BALI   `www.in.tum.de/~isabelle/bali/`

Motivation zur Formalisierung

Probleme:

- Korrektheit von Programmen und Umgebung
- Sicherheitsrisiken (Fehler, Manipulationen)

unsere Ziele:

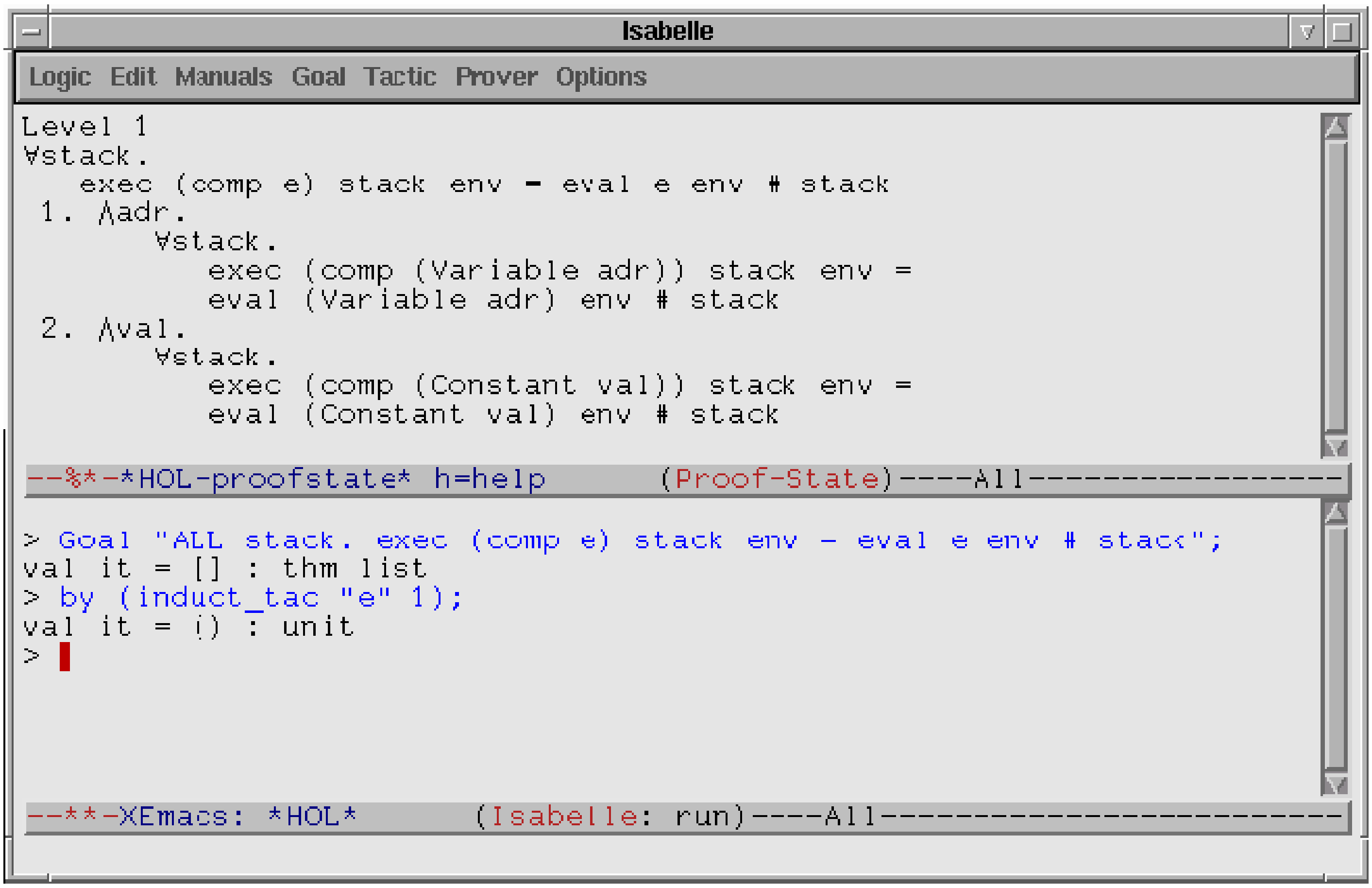
- präzise, übersichtliche Referenzspezifikation
- Design-Check für Sprache und Umgebung
- Überprüfung von Spracherweiterungen
- Korrektheitsbeweis für Implementierungen
- Verifikation von Anwendungsprogrammen

unsere Lösung:

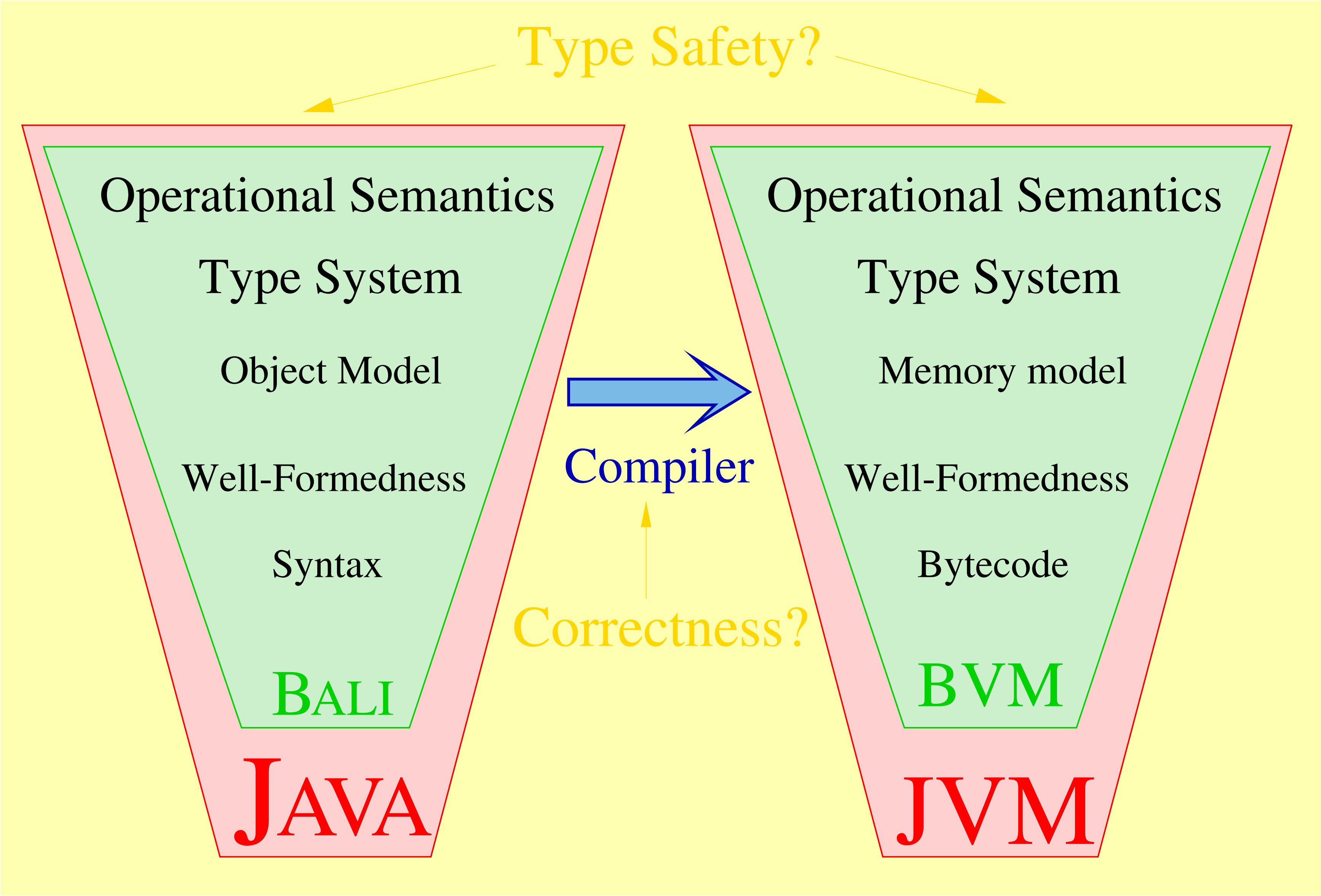
Formalisierung und Verifikation
mit dem Beweissystem *Isabelle/HOL*

Isabelle [*Paulson, Nipkow*]

- leistungsfähiger, flexibler Theorembeweiser
- ausdrucksstarke Spezifikationssprache HOL
- mächtige und sichere Beweisverfahren
- interaktive und semi-automatische Beweise



Umfang der Formalisierung: $\approx$ *JavaCard*



Typsicherheit

- Schlüsseleigenschaft von Programmiersprachen
- Relation zwischen Typsystem und Semantik
- verhindert Typfehler zur Laufzeit

Java-Quellsprachen-Ebene:

- schließt statisch erkennbare Typfehler aus
- noch keine Ablaufsicherheit
- für disziplinierte Programmentwicklung

Bytecode-Ebene:

- unsichere Quellen von Bytecode-Programmen
- erneute Typprüfung durch *Bytecode Verifier*
- entscheidender Teil des Sicherheitskonzepts

Bali: Formalisierung von Java

inspiriert durch [*Drossopoulou/Eisenbach*]
(als DECLARE-Fallstudie [*Syme*])

möglichst abstrakt und adäquat!

- Datenstrukturen zur Repräsentation von Java-Programmen (Deklarationen, Terme und Typen), z.B.
  $prog = (cdecl)list \times (idecl)list$

- Prädikate zur Darstellung von Typrelationen, Wohlgetyptheit (z.B. $\Gamma,\Lambda \vdash s{::}\Diamond$) und Wohlgeformtheit (z.B. wf_prog Γ)

- Datenstrukturen und Definitionen zur Modellierung von Werten, Zuständen
  $state = (xcpt)option \times (heap \times stat \times locals)$,
  Programmausführung und Konformität

Beweise für Bali

Auswertungssemantik: $\Gamma \vdash \ \sigma \ -e \rhd v \rightarrow \ \sigma'$
Zustandsübergang bei der Auswertung
eines Ausdrucks zu einem Wert

Notation: Konformität

$\sigma :: \preceq \Gamma, \Lambda \ \stackrel{\text{def}}{=}$ "die Werte aller Variablen im
Zustand σ passen zu ihrem deklarierten Typ"

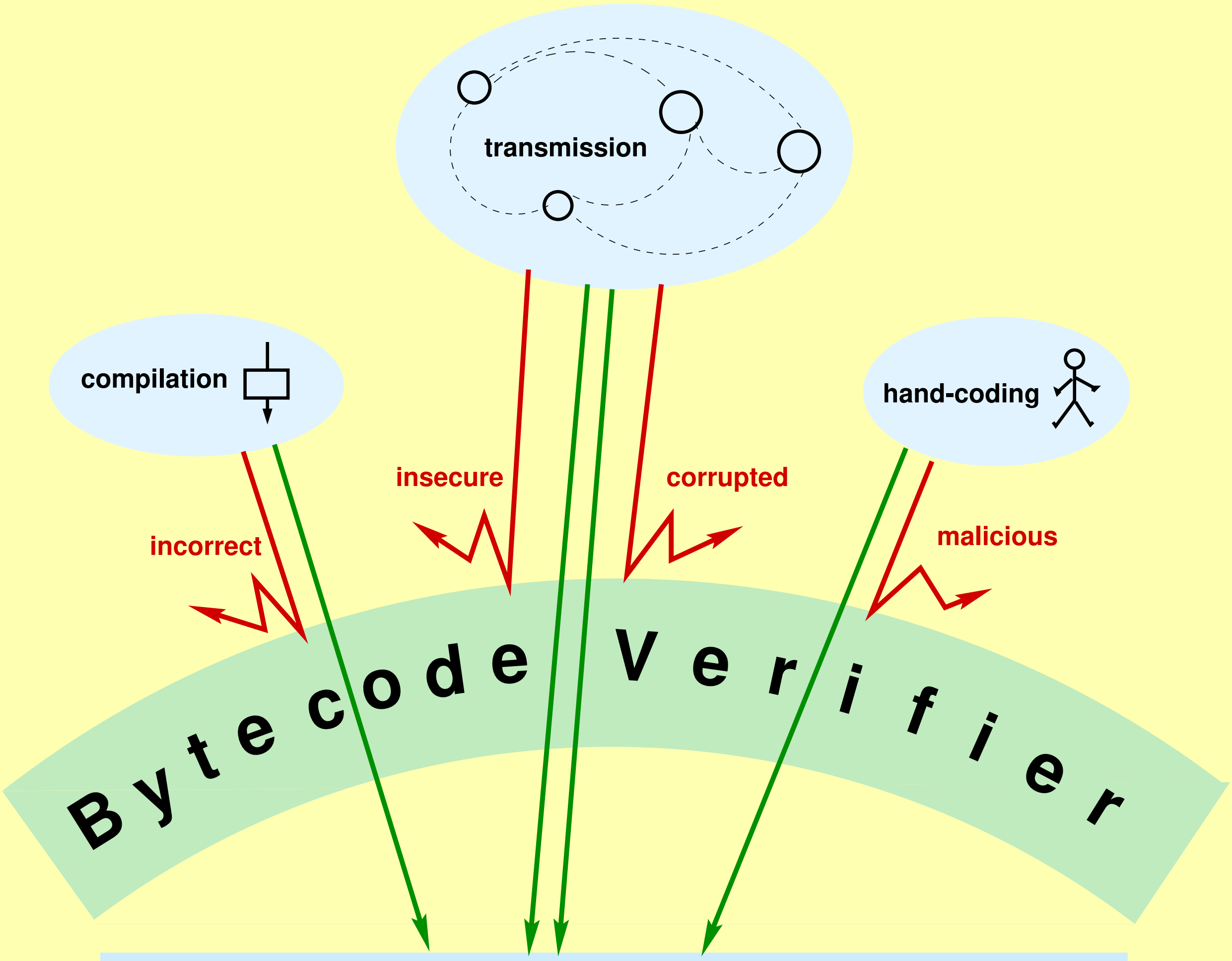
Typsicherheits-Theorem: Bei der Ausführung
bleibt die Konformität erhalten.

wf_prog $\Gamma \ \wedge \ \Gamma, \Lambda \vdash s :: \diamond \ \wedge \ \Gamma \vdash \sigma \ -s \rightarrow \ \sigma' \ \wedge$
$\sigma :: \preceq \Gamma, \Lambda \ \Longrightarrow \ \sigma' :: \preceq \Gamma, \Lambda$

Korollar:

"method not understood"-Laufzeitfehler
können nicht auftreten.

Aufgaben des Bytecode Verifiers



BV checks → JVM may assume:

- no stack under/overflow
- valid register accesses/stores
- valid heap access
- instructions have correct parameters
- no illegal data conversion

Überprüfung des Bytecodes

Design-Alternativen:

- Typüberprüfung zur Laufzeit
  (*defensive* JVM in ACL2 [*Cohen*])
- Aufteilung in statische und dynamische Anteile
  (Bytecode Verifier von Sun)
- Vollständig statische Typüberprüfung
  (erweiterte BV Spezifikation [*Qian*]:
  erfordert Erweiterung des Typsystems)

Wir folgen Qians Ansatz.

Spezifikation des BV:

Typcheck-Prädikate überprüfen gegebenen Typ

Anforderung an BV-Implementierung:

BV berechnet einen Typ, der vom Typchecker akzeptiert wird, oder meldet Fehler

BVM: Semantik und Typsicherheit

Classfile: Java-Kompilat, weiterhin mit objektorientierter Struktur, enthält $Bytecode$ (Instruktionen einer Stack-Maschine)

Einzelschritt-Semantik: exec $instr\ \sigma\ = \sigma'$
Jede Instruktion bewirkt eine atomare Änderung des Maschinenzustands.

Korrektheit des Bytecode Verifiers:
In vom BV akzeptierten Programm sind alle erzeugten Werte passend zum statischen Typ.

Korollar:
Im akzeptierten Code sind zur Laufzeit die Argumente aller Maschineninstruktionen vom richtigen Typ, z.B.

$$(\text{get\_code}\ \Gamma\ \ C\ m)\ !\ pc\ \ =\ \ \text{Aload}\ idx \Longrightarrow$$
$$loc\ !\ idx = \text{Null}\ \ \vee\ \ \exists a.\ loc\ !\ idx = \text{Addr}\ a$$

Ergebnisse

- Java – soweit formalisiert – ist typsicher.
- Spezifikationen: nur kleine Lücken und Fehler.
  z.T. redundant und zu implementierungsnah
- Sprachdesign: kleine Verallgemeinerungsmöglichkeiten

Statistik

Teilprojekt	Formalisierung	Typsicherheits-Beweis
Java	1200 Zeilen / 2,5 Monate	2300 Zeilen / 3 Monate
JVM, BV	1700 Zeilen / 3 Monate	2400 Zeilen / 3,5 Monate

Weitere Arbeiten

- Compilerverifikation, Axiomatische Semantik, Typparameter
- Programmverifikation, . . .

Erfahrungen

- Ausdrucksstärke und Beweismächtigkeit von Isabelle/HOL
- Werkzeugunterstützung zur Validierung wäre hilfreich
- Formalisierung möglichst einfach und abstrakt
- Maschine sichert Korrektheit auch von Erweiterungen

Resümee

Viele interessante Aspekte
der Programmiersprache Java
können voll formal behandelt werden,
und zwar mit vertretbarem Aufwand.