

Rheinisch-Westfälische Technische Hochschule Aachen

Lehr- und Forschungsgebiet Informatik II
Prof. Dr. J. Giesl

Seminar "Verifikation von Programmen"

Java-Verifikation mit *Isabelle*

Achim Lücking

Wintersemester 2001/2002

Inhaltsverzeichnis

| | | |
|----------|--|-----------|
| 1 | Einführung | 3 |
| 2 | <i>Isabelle</i> - eine kurze Einführung | 3 |
| 2.1 | Kartesisches Produkt | 3 |
| 2.2 | Listen | 4 |
| 2.3 | Optionale Werte | 4 |
| 2.4 | Mappings | 4 |
| 3 | Die Formalisierung von Java | 5 |
| 3.1 | Java-Programm | 5 |
| 3.2 | Das Typsystem von Java | 8 |
| 3.2.1 | Typrelationen | 9 |
| 3.2.2 | Regeln für korrekte Typung | 9 |
| 3.3 | Wohldefiniiertheit von Java-Programmen | 11 |
| 3.4 | Operationelle Semantik | 12 |
| 4 | Nachweis der Typkorrektheit von Java | 14 |
| 5 | Ausblick: Möglichkeiten der Bytecode-Verifikation | 17 |
| | Literaturverzeichnis | 19 |

1 Einführung

Diese Arbeit beschäftigt sich mit der Möglichkeit, Programme in der Programmiersprache Java mit dem Verifikationswerkzeug *Isabelle* zu verifizieren. Das heißt in diesem Fall, daß ein Weg aufgezeigt wird, wie sich Java-Programme in der Logik des Verifikationswerkzeugs *Isabelle* modellieren lassen, mit dem Ziel, einen Typkorrektheitsbeweis für Java mit *Isabelle* durchführen zu können. Es gilt also unter anderem, die abstrakte Syntax, die Semantik und das Typsystem von Java in einer Higher-Order-Logic (HOL), wie sie *Isabelle* verwendet, zu formalisieren. Ferner sollen die Abhängigkeiten gezeigt werden, die für die Typkorrektheit von Java erforderlich sind, so daß ein Beweis der Typkorrektheit direkt möglich ist. An dieser Stelle kann nicht die komplette Sprache Java behandelt werden, da diese sehr mächtig und umfangreich ist, so daß eine Beschränkung des Sprachumfangs von Java bei der Modellierung vorgenommen wird. Bei den Notationen wird kein vollständiges und lauffähiges Verifikationsskript für *Isabelle* angegeben, vielmehr werden die Abhängigkeiten und Theoreme formal innerhalb der HOL notiert, so daß man sie jederzeit in die Syntax eines Verifikationsskriptes übertragen kann.

2 *Isabelle* - eine kurze Einführung

Im Rahmen dieses Seminars wurde das Verifikationswerkzeug *Isabelle* bereits vorgestellt. Daher sei im folgenden Abschnitt lediglich ein kurzer Überblick über die Modellierung mit der HOL in *Isabelle* gegeben. Dabei werden vor allem diejenigen Bereiche betrachtet, die im weiteren Verlauf dieser Arbeit von Relevanz sind.

Isabelle arbeitet auf einer higher-order-logic (HOL). Bei näherer Betrachtung kann man die Skriptsprache von *Isabelle* als eine Kombination aus funktionaler und logikbasierter Programmierung beschreiben. Ihr Typkonzept ist ähnlich dem der funktionalen Programmiersprache Haskell. Dabei sind auch rekursive Datentypen und Funktionen, sowie induktiv definierte Mengen möglich.

Für die Formalisierung von Java sind einige vordefinierte Konstrukte notwendig. Die wichtigsten sind hier noch einmal in Kürze zusammengetragen.

2.1 Kartesisches Produkt

Wichtig für die Formalisierung ist das kartesische Produkt. Der Typ des kartesischen Produktes von τ_1 und τ_2 wird mit $\tau_1 \times \tau_2$ bezeichnet. Ferner sind die folgenden beiden Funktionen fst und snd zur Projektion definiert:

$$\begin{aligned}\text{fst} &:: \alpha \times \beta \rightarrow \alpha \\ \text{snd} &:: \alpha \times \beta \rightarrow \beta\end{aligned}$$

Sie liefern das erste bzw. das zweite Element des kartesischen Produktes.

2.2 Listen

Die HOL von *Isabelle* bietet auch eine Datenstruktur zur Darstellung von Listen an. Eine Liste ist dabei wie folgt als rekursiver Datentyp definiert:

$$\mathbf{datatype} \quad \alpha \textit{ list} = \mathbf{Nil} \mid \mathbf{Cons} \ \alpha \ (\alpha \textit{ list})$$

Hierbei ist α ein beliebiger Typ. Für die Darstellung von \mathbf{Nil} kann auch der Ausdruck \square verwendet werden. Außerdem kann man statt des Konstruktors \mathbf{Cons} alternativ auch den Infixoperator $\#$ nutzen. Ferner sind für die Java-Formalisierung auf Listen die folgenden Listenfunktionen interessant:

$$\begin{aligned} \mathbf{app} &:: \alpha \textit{ list} \rightarrow \alpha \textit{ list} \rightarrow \alpha \textit{ list} \\ \mathbf{length} &:: \alpha \textit{ list} \rightarrow \mathit{nat} \\ \mathbf{set} &:: \alpha \textit{ list} \rightarrow \alpha \textit{ set} \\ \mathbf{map} &:: (\alpha \rightarrow \beta) \rightarrow \alpha \textit{ list} \rightarrow \beta \textit{ list} \\ \mathbf{zip} &:: \alpha \textit{ list} \rightarrow \beta \textit{ list} \rightarrow (\alpha \times \beta) \textit{ list} \\ \mathbf{nodups} &:: \alpha \textit{ list} \rightarrow \mathit{bool} \\ \mathbf{nth} &:: \alpha \textit{ list} \rightarrow \mathit{nat} \rightarrow \alpha \end{aligned}$$

Die Funktion \mathbf{app} hängt an eine bestehende Liste eine andere Liste an. Dabei kann alternativ der Infixoperator $\mathbf{@}$ gewählt werden. \mathbf{length} liefert die Länge einer Liste, \mathbf{set} wandelt eine Liste in eine Menge gleichen Typs. Die Funktion \mathbf{map} wendet eine Funktion auf jedes Element einer Liste an, \mathbf{zip} fügt zwei Listen paarweise zusammen, \mathbf{nodups} liefert den Wert true , falls keine doppelten Elemente in einer Liste enthalten sind und \mathbf{nth} liefert das n -te Element einer Liste.

2.3 Optionale Werte

Für die Modellierung von optionalen Werten ist der folgende Datentyp definiert:

$$\mathbf{datatype} \quad \alpha \textit{ option} = \mathbf{None} \mid \mathbf{Some} \ \alpha$$

Zusätzlich ist die Funktion \mathbf{the} definiert, die als Selektor für den Datentyp $\alpha \textit{ option}$ fungiert.

$$\begin{aligned} \mathbf{consts} \quad \mathbf{the} &:: \alpha \textit{ option} \rightarrow \alpha \\ \mathbf{primrec} \quad \mathbf{the} \ (\mathbf{Some} \ x) &= x \\ &\mathbf{the} \ (\mathbf{None}) = \square_{\alpha} \end{aligned}$$

\square_{α} bezeichnet dabei einen beliebigen Wert, der abhängig vom Typ α ist.

2.4 Mappings

Mappings sind partielle Funktionen, mit denen man entscheiden kann, ob ein Eintrag definiert ist oder nicht. Dazu ist zunächst der folgende Typ definiert:

$$\mathbf{types} \quad \alpha \rightsquigarrow \beta = \alpha \rightarrow \beta \textit{ option}$$

Mögliche Anwendungen sind z.B. Symboltabellen, in denen Namen mit anderen Informationen verknüpft sind. Für den Umgang mit den Mappings stehen die folgenden Funktionen zur Verfügung:

- $\text{empty} :: \alpha \rightsquigarrow \beta$
 $\text{empty} \equiv \lambda k. \text{None}$
 So wird das leere Mapping dargestellt.
- $_(- \mapsto _) :: (\alpha \rightsquigarrow \beta) \rightarrow \alpha \rightarrow \beta \rightarrow (\alpha \rightsquigarrow \beta)$
 $m(x \mapsto y) \equiv \lambda k. \text{if } k = x \text{ then } (\text{Some } y) \text{ else } (m\ k)$
 Für ein Mapping m wird eine Ersetzung des Eintrags für einen Schlüssel x vorgenommen.
- $_ \oplus _ :: (\alpha \rightsquigarrow \beta) \rightarrow (\alpha \rightsquigarrow \beta) \rightarrow (\alpha \rightsquigarrow \beta)$
 $s \oplus t \equiv \lambda k. \text{case } t\ k \text{ of } \text{None} \rightarrow s\ k \mid \text{Some } y \rightarrow \text{Some } y$
 Mit diesem Operator wird ein Mapping s durch ein Mapping t überschrieben. D.h. erst wenn sich kein Eintrag für einen Schlüssel k im Mapping t findet, wird nach einem entsprechenden Eintrag im Mapping s gesucht.
- $\text{map_of} :: (\alpha \rightsquigarrow \beta) \text{list} \rightarrow (\alpha \rightsquigarrow \beta)$
 $\text{map_of } [] = \text{empty}$
 $\text{map_of } ((x, y) \# l) = (\text{map_of } l)(x \mapsto y)$
 Hier wird eine Assoziationsliste in ein Mapping überführt.
- $_(-[\mapsto]_) :: (\alpha \rightsquigarrow \beta) \rightarrow \alpha \text{ list} \rightarrow \beta \text{ list} \rightarrow (\alpha \rightsquigarrow \beta)$
 $m([], [\mapsto] []) = m$
 $m(a \# as [\mapsto] b \# bs) = m(a \mapsto b)(as[\mapsto]bs)$
 Für ein Mapping m wird eine Liste von Ersetzungen vorgenommen.

3 Die Formalisierung von Java

Die Programmiersprache Java ist eine mächtige Programmiersprache mit einem großen Sprachumfang. Eine vollständige Formalisierung der gesamten Sprache würde über den Rahmen dieser Arbeit hinausgehen. Daher ist es sinnvoll, sich auf eine Teilmenge von Java zu beschränken, um die Grundkonzepte zu verdeutlichen. Es werden daher nicht alle Feinheiten der Sprache formal ausgearbeitet. Bei der folgenden Formalisierung wird eine formale Darstellung als Datentypen der von *Isabelle* verwendeten HOL gewählt.

3.1 Java-Programm

Ein Java-Programm besteht aus einer Reihe von Klassen. Man kann auch sagen, daß ein Java-Programm durch eine Liste von Klassendeklarationen repräsentiert wird. Eine solche Liste kann man nun modellieren. Formal schreibt man:

$$\text{types} \quad \text{prog} = \text{cdecl list}$$

Interfaces werden bei dieser Formalisierung nicht betrachtet. Ein möglicher Ansatz ist, ein Java-Programm als kartesisches Produkt aus einer Liste von Klassen und einer Liste von Interfaces zu beschreiben. In diesem Fall muß bei den weiteren Formalisierungen immer darauf geachtet werden, daß aus jedem Interface auch jede deklarierte Methode implementiert wird. Außerdem sind dann weitere Abhängigkeiten zu prüfen.

Eine Klasse selbst besteht aus einem 4-Tupel. Zur Klasse gehört ein Klassenname, eine übergeordnete Klasse (Superklasse) und die Feld- und Methodendeklarationen. Da Feld- und Methodendeklarationen nicht auf jeweils eine Deklaration beschränkt sind, wird auch hier als Repräsentation eine Liste gewählt. Formal ergibt sich für eine Klassendeklaration folgende Signatur:

types $cdecl = cname \times cname\ option \times fdecl\ list \times mdecl\ list$

Die Angabe der übergeordneten Klasse ist optional, da es auch eine absolut übergeordnete Klasse `Object` gibt, die keine Superklasse hat und auch modellierbar sein soll.

Eine Felddeklaration ist wiederum zusammengesetzt aus einem Namen und einem Typ. Eine Methodendeklaration ist etwas komplexer und setzt sich aus dem Namen der Methode, einer Liste von Parametern, einem Rückgabewert und dem Methodenrumpf zusammen. Formal ergibt sich:

types $fdecl = vname \times ty$

types $mdecl = (mname \times ty\ list) \times ty \times mbody$

Auf die Modellierung der Klassen-, Feld- und Methodennamen wird hier nicht weiter eingegangen.

Der Methodenrumpf besteht aus einem 3-Tupel. Es wird optional eine Liste von lokalen Variablen angegeben, die Statements und ein Ausdruck für die Rückgabe sind auf jeden Fall anzugeben. Beim Rückgabewert ist zu beachten, daß unsere Java-Vereinfachung keine `return`-Anweisung enthält. Der Rückgabewert wird also über einen Ausdruck im dritten Element des Tupels geliefert und würde im Quelltext einfach dieser Rückgabevariable zugeschrieben. Formal geschrieben sieht das dann wie folgt aus:

types $mbody = (vname \times ty)list \times stmt \times expr$

In diesem Modell ist der Sprachumfang von Java beschränkt auf Ausdrücke, eine Folge von Statements, `if`-Anweisungen und `while`-Schleifen. In der Logik von *Isabelle* kann man für ein Statement einen rekursiven Datentyp deklarieren, der ähnlich einer Grammatik notiert wird:

datatype $stmt =$ `Skip`
 | `expr`
 | `stmt; stmt`
 | `if (expr) stmt else stmt`
 | `while (expr) stmt`

Skip hat dabei dieselbe Bedeutung, wie das ε einer Grammatik.

Ein Ausdruck besteht in der hier eingeschränkten Version von Java aus Literalen, lokalen Variablen (schreibend und lesend) und expliziten Typzuweisungen (type casts). Außerdem müssen objekt-orientierte Eigenschaften, wie Klasseninstanziierung, Feldzugriffe (lesend und schreibend), sowie Methodenaufrufe berücksichtigt werden.

```

datatype expr = This
                | new cname
                | (ty) expr
                | val
                | vname
                | vname := expr
                | {cname}expr.vname
                | {cname}expr.vname := expr
                | expr.mname({ty list})expr list

```

This ist der Ausdruck für die Adresse der eigenen Klasse. Es ist zu beachten, daß an dieser Stelle Teile mitmodelliert wurden, die erst vom Compiler bei der Typüberprüfung erzeugt werden. Es handelt sich um die Ausdrücke in geschweiften Klammern (type annotations). Die Angaben werden in Java für die Handhabung der Methodenüberladung und das Binden von Feldern an eine Klasse benötigt. Wenn mit diesem Modell ein konkreter Java-Quelltext dargestellt werden soll, liegen die vom Compiler berechneten Informationen nicht vor und müssen manuell bei der Modellierung mit angegeben werden.

Es bleiben noch Werte und Typen zu modellieren. Das hier betrachtete Modell soll mit fünf Typen auskommen. Arrays werden nicht betrachtet. Es ist aber möglich, das Modell um Referenztypen zu erweitern, die dann auch Array-Konstruktionen zulassen. Bei den Basistypen beschränkt sich das Modell auf die Typen Integer und Boolean.

```

datatype ty = void
                | boolean
                | int
                | NT
                | Class cname

```

void ist dabei zur Modellierung des nicht vorhandenen Rückgabewertes von Methoden, die mit dem Modifier void deklariert sind, gedacht; NT ist ein Nulltyp, d.h. jede Nullreferenz ist vom Typ NT. Variablen der oben angeführten Typen können folgende Werte annehmen:

```

datatype val = Unit
                | Bool bool
                | Intg int
                | Null
                | Addr loc

```

Hier ist *loc* die Angabe einer Speicheradresse. Jeder Wert hat einen Standardwert, der je nach Typ vorgegeben ist. Es ist die Methode `defaultval` deklariert, die diese Standardwerte liefert. Ferner ist eine Methode `typeof` deklariert, die den entsprechenden Typ eines Ausdrucks liefert.

```

primrec defaultval void      = Unit
          defaultval boolean = Bool False
          defaultval int      = Intg 0
          defaultval NT       = Null
          defaultval (Class C) = Null

```

```

primrec typeof dt Unit      = Some void
          typeof dt (Bool b) = Some boolean
          typeof dt (Intg i)  = Some int
          typeof dt Null      = Some NT
          typeof dt (Addr a)  = dt a

```

Hierbei ist *dt* ein Argument vom Typ $(loc \rightarrow ty\ option)$. Es wird später noch benötigt, um die Existenz und den Typ bzw. den Klassentyp von Objekten auf einem Heap (vgl. Kap. 3.4) zu bestimmen.

Ferner seien noch zwei Look-Up-Funktionen angegeben, die im weiteren Verlauf benötigt werden:

- `method :: prog \times cname \rightarrow ((mname \times ty list) \rightsquigarrow cname \times ty \times mbody)`
Diese Funktion liefert ein optionales 3-Tupel für eine Methode mit den gesuchten Eigenschaften $(mname \times ty\ list)$ bestehend aus dem Klassennamen der Klasse, in dem die Methode deklariert wird, dem Typ des Rückgabewertes der Methode und dem Methodenrumpf. Dabei wird in dem Programm von der gegebenen Klasse aus in der Vererbungshierarchie aufwärts gesucht. Ein Ergebnis ist also entweder `Some(D, T, b)`, d.h. die Methode mit der gesuchten Signatur ist in einer Klasse *D* deklariert, hat den Rückgabety *T* und den Rumpf *b*, oder das Ergebnis ist `None`, d.h. es wurde keine Methode gefunden. Die Funktion arbeitet rekursiv, d.h. sie steigt solange in der Hierarchie weiter in die Superklassen aufwärts, bis ein Ergebnis gefunden wird oder ein weiterer Aufstieg nicht möglich ist, die Klasse `Object` demnach erreicht ist.
- `fields :: prog \times cname \rightarrow ((vname \times cname) \times ty)list`
Diese Funktion arbeitet ähnlich wie die Funktion `method`. Hier wird eine Liste mit 2-Tupel zurückgegeben, die von einer Klasse aus in der Hierarchie aufsteigend alle Felder mit ihrem Typ und ihrer deklarierenden Klasse auflistet.

3.2 Das Typsystem von Java

Ein weiterer Schwerpunkt der Modellierung ist das Typsystem von Java. Dabei werden insbesondere Unterklassenrelationen und Regeln für die korrekte Typung betrachtet.

3.2.1 Typrelationen

Ein Java-Programm (im folgenden auch kurz mit Γ bezeichnet) besteht aus Klassen und deren Unterklassen. Zwischen diesen Klassen bestehen eindeutige Beziehungen. Um die Beziehungen zwischen den Klassen zu formalisieren, sei folgende Definition eingeführt:

$$\text{subclass } \Gamma \equiv \{(C, D) \mid \exists r (C, \text{Some } D, r) \in \text{set } \Gamma\}$$

$\text{subclass } \Gamma$ ist die Menge aller Klassenpaare, bei denen das erste Element eine direkte Unterklasse des zweiten Elementes ist. Darauf aufbauend kann man nun folgende Unterklassenrelation definieren:

$$\Gamma \vdash C \preceq_C D \equiv (C, D) \in (\text{subclass } \Gamma)^*$$

Hiermit wird der Sachverhalt beschrieben, daß C eine Unterklasse von D innerhalb des Programms Γ ist.

Basierend auf dieser Unterklassenrelation, kann man nun eine erweiterte Relation $\Gamma \vdash S \preceq T$ definieren, die allgemein auf Typen gilt und aussagt, daß überall dort, wo ein Ausdruck vom Typ T erwartet wird, auch ein Ausdruck vom Typ S stehen kann. Die Definition der Relation ist induktiv:

- dort, wo ein Ausdruck vom Typ T erwartet wird, kann trivialerweise auch ein Ausdruck vom Typ T stehen:

$$\overline{\Gamma \vdash T \preceq T}$$

- dort, wo ein Typ der Form $\text{Class } C$ erwartet wird, darf auch der Nullpointer stehen:

$$\overline{\Gamma \vdash \text{NT} \preceq \text{Class } C}$$

- dort, wo ein Typ der Form $\text{Class } D$ erwartet wird, darf aufgrund der Vererbungsbeziehung dann ein Typ der Form $\text{Class } C$ stehen, wenn C eine Unterklasse von D ist:

$$\frac{\Gamma \vdash C \preceq_C D}{\Gamma \vdash \text{Class } C \preceq \text{Class } D}$$

Dies ist erlaubt, weil die Klasse C die Klasse D bei der Vererbung allenfalls verfeinert, zumindest bleiben die Eigenschaften der Klasse D bei der Vererbung erhalten.

3.2.2 Regeln für korrekte Typung

Für eine korrekte Typung muß man sich überlegen, wann ein Statement bzw. ein Ausdruck wohlgeformt, also auch korrekt ist. Dazu sei zunächst der Begriff einer Umgebung eingeführt. Eine Umgebung ist ein 2-Tupel bestehend aus dem Programm und die Typbindung der lokalen Variablen der Umgebung.

Eine korrekte Typung zeigt das folgende Beispiel:

$$\frac{E \vdash e :: \text{boolean} \quad E \vdash s \checkmark}{E \vdash \text{while}(e) s \checkmark}$$

Diese Regel beschreibt die Typabhängigkeit für eine while-Schleife. Wenn in der Umgebung E ein Ausdruck e vom Typ `boolean` ist und zugleich in derselben Umgebung ein Statement s korrekt ist, dann ist auch die while-Schleife in der gegebenen Form korrekt, andernfalls liegt ein Fehler vor. Nach diesem Schema läßt sich auch die if-Anweisung beschreiben. Eine Erweiterung auf den case-Ausdruck oder die do-while-Schleife ist möglich, soll aber aufgrund der getroffenen Einschränkungen nicht weiter betrachtet werden.

Etwas komplexer ist die Betrachtung der Typung beim Zugriff auf ein Feld und beim Aufruf von Methoden. Zunächst sei hier der Feldzugriff formalisiert:

$$\frac{E \vdash a :: \text{Class } D \quad \text{field}(\text{fst } E, D) \text{ } fn = \text{Some}(C, fT)}{E \vdash \{C\}a.fn :: fT}$$

Hierzu wird die Hilfsfunktion `field` verwendet. Diese ist wie folgt deklariert:

$$\text{field} \equiv \text{map_of} \circ (\text{map}(\lambda((fn, fd), ft).(fn, (fd, ft)))) \circ \text{fields}$$

Mit dieser Hilfsfunktion erkennt man, daß der Ausdruck `field (fst E , D) fn` für einen gesuchten Feldnamen fn einen optionalen Wert liefert, der Auskunft gibt, ob und in welcher Klasse ein Feld mit Namen fn deklariert ist und welchen Typ das Feld hat. Ein Feldzugriff ist somit typkorrekt, wenn innerhalb einer Umgebung ein Ausdruck a vom Typ `Class D` ist und gleichzeitig ein Feld fn existiert, welches in einer Klasse C deklariert ist und den Typ fT hat. Dann hat nämlich auch der Zugriff aus dem Ausdruck a auf das Feld fn der Klasse C den Typ fT . Gleichzeitig ist auch klar, daß D eine Unterklasse von C sein muß.

Für einen Methodenaufruf ergibt sich folgende Regel:

$$\frac{E \vdash e :: \text{Class } C \quad E \vdash ps[::]pTs}{\text{max_spec}(\text{fst } E) C (mn, pTs) = \{((-, rT), fpTs)\}} \quad E \vdash e.mn(\{fpTs\}ps) :: rT$$

In der angegebenen Formel ist ps die Liste der Ausdrücke, die einer Methode als Parameter übergeben werden sollen. Ferner ist pTs die Liste der Typen dieser Parameter. Außerdem ist e ein Ausdruck vom Typ `Class C` . Die Funktion `max_spec` erzeugt nun eine Menge. Diese Menge enthält ausgehend von der Referenzklasse C im Programm zunächst alle Methoden, die dem Methodennamen mn entsprechen, einen Rückgabotyp rT haben und deren Parameterliste der geforderten Parameterliste entspricht, d.h. alle in der aktuellen Umgebung zu den gegebenen Konditionen prinzipiell ausführbaren Methoden. Zusätzliche Bedingung ist jedoch, daß bei mehreren gleichen Methoden nur die Methoden der untersten Hierachiestufe genommen werden, da diese die speziellsten Methoden sind. Die Funktion `max_spec` liefert also die Menge der "genausten" ausführbaren Methoden für

einen Referenztyp C . $fpTs$ ist dann die Parameterliste der speziellsten Methode. Diese Parameterliste kann von der Liste der Typen, die übergeben werden sollen, abweichen, da in der Parameterliste auch Obertypen der Typen, die der Methode übergeben werden sollen, erlaubt sind. Formal gilt zwischen pTs und $fpTs$ die Beziehung:

$$\text{length } pTs = \text{length } fpTs \wedge (\forall(T, T') \in \text{zip } pTs \text{ } fpTs : \Gamma \vdash T \preceq T')$$

Wenn die Menge der speziellsten Methoden nun genau einelementig ist, dann ist der Methodenaufruf typkorrekt, da die Typzuweisung eindeutig ist, nämlich mit dem Rückgabety rT dieser Methode.

3.3 Wohldefiniertheit von Java-Programmen

Beim Kompilieren eines Programms führt der Compiler eine Reihe von Prüfungen durch, unter anderem wird das Programm auf Wohldefiniertheit überprüft. Mit den bis jetzt vorgenommenen Modellierungen ist es nun möglich, auch die Definition der Wohldefiniertheit von Programmen, Klassen und Methoden formal zu beschreiben.

Ein Java-Programm ist wohldefiniert, wenn es eine absolute Superklasse `Object` besitzt, alle enthaltenen Klassendeklarationen wohldefiniert sind und keine Klasse doppelt vorkommt. Formal:

$$\begin{aligned} \text{wf_prog } \Gamma &\equiv (\text{Object}, (\text{None}, -, -)) \in \text{set } \Gamma \wedge \\ &\forall c \in \text{set } \Gamma : \text{wf_cdecl } \Gamma \ c \wedge \\ &\text{nodups}(\text{map fst } \Gamma) \end{aligned}$$

Eine Klassendeklaration wiederum ist wohldefiniert, wenn alle Typen der Felddeklaration erlaubt sind, kein Feld doppelt deklariert ist, alle Methodendeklarationen wohldefiniert sind, keine Methode doppelt deklariert ist und eine Superklasse vorhanden ist. Eine Ausnahme bildet die Klasse `Object`, bei der keine Superklasse vorhanden sein darf. Außerdem muß für die Vererbung noch gelten, daß die Superklasse auch Bestandteil des Programms ist und daß sie ferner nicht auch eine Unterklasse ist. Für den Fall, daß eine Methode eine Methode der Superklasse überschreibt, muß der Rückgabety ein Untertyp des Rückgabety der überschriebenen Superklassenmethode sein. Formal:

$$\begin{aligned} \text{wf_cdecl } \Gamma (C, sc, fs, ms) &\equiv (\forall(-, T) \in \text{set } fs : \text{is_type } \Gamma \ T) \wedge \\ &\text{nodups}(\text{map fst } fs) \wedge \\ &(\forall m \in \text{set } ms : \text{wf_mdecl } \Gamma \ C \ m) \wedge \\ &\text{nodups}(\text{map fst } ms) \wedge \\ &\text{case } sc \text{ of} \\ &\quad \text{None} \quad \rightarrow C = \text{Object} \\ &\quad | \text{Some } D \rightarrow D \in \text{set}(\text{map fst } \Gamma) \wedge \\ &\quad \quad \neg(\Gamma \vdash D \preceq_C C) \wedge \\ &\quad \quad \forall(sg, T, -) \in \text{set } ms, \forall T' : \\ &\quad \quad \text{method}(\Gamma, D)sg = \text{Some}(-, T', -) \\ &\quad \quad \rightarrow \Gamma \vdash T \preceq T' \end{aligned}$$

Dabei bezeichnet sg die Signatur einer Methode.

Die Wohldefiniertheit einer Methodendeklaration ist genau dann gegeben, wenn sowohl die Methode selbst, als auch ihr Kopf wohldefiniert sind. Als Hilfsfunktion ist die Funktion `is_type` gegeben, die überprüft, ob ein Typ überhaupt existiert. Das ist genau dann relevant, wenn der Typ eine Klasse sein soll.

$$\begin{aligned}
\text{wf_mdecl } \Gamma \ C \ ((mn, pTs), rT, b) &\equiv \text{wf_mhead } \Gamma \ (mn, pTs) \ rT \\
&\quad \wedge \text{wf_method } \Gamma \ C \ ((mn, pTs), rT, b) \\
\text{wf_mhead } \Gamma \ (mn, pTs) \ rT &\equiv (\forall T \in \text{set } pTs : \text{is_type } \Gamma \ T) \wedge \text{is_type } \Gamma \ rT \\
\text{is_type } \Gamma \ T &\equiv \text{case } T \text{ of Class } C \rightarrow C \in \text{set}(\text{map fst } \Gamma) \mid _ \rightarrow \text{True}
\end{aligned}$$

Methoden werden nur innerhalb des Kontextes einer Klasse betrachtet. Für die Wohldefiniertheit einer Methode müssen in der Signatur mindestens genauso viele Parameternamen (pns) angegeben werden, wie Typen für Parameter (pTs) angegeben sind. Die Namen der Parameter und lokalen Variablen sind jeweils einmalig vergeben und kollidieren nicht miteinander. Alle lokalen Variablen ($lvars$) sind korrekt getypt. Außerdem muß der Methodenrumpf (blk) im Kontext des Programms, der lokalen Variablen und der aktuellen Klasse korrekt getypt sein. Der Ausdruck mit dem Rückgabewert (res) muß vom Typ eines Untertyps des formal für die Methode definierten Rückgabetyps (rT) sein. Formal:

$$\begin{aligned}
\text{wf_method } \Gamma \ C \ ((mn, pTs), rT, (pns, lvars, blk, res)) &\equiv \\
\text{length } pns = \text{length } pTs \wedge & \\
\text{nodups } pns \wedge \text{nodups}(\text{map fst } lvars) \wedge & \\
(\forall pn \in \text{set } pns : \text{map_of } lvars \ pn = \text{None}) \wedge & \\
(\forall (vn, T) \in \text{set } lvars : \text{is_type } \Gamma \ T) \wedge & \\
\text{let } E = (\Gamma, \text{map_of } lvars(pns[\mapsto]pTs)(\text{this} \mapsto \text{Class } C)) & \\
\text{in } E \vdash blk \ \checkmark \wedge (\exists T : E \vdash res :: T \wedge \Gamma \vdash T \preceq rT) &
\end{aligned}$$

Im Gegensatz zu Java ist es hier nicht erforderlich, lokale Variablen explizit zu initialisieren. Die operationale Semantik macht dieses beim Methodenaufruf.

3.4 Operationelle Semantik

Die Semantik sei hier als operationelle Beschreibung über Programmzustände angegeben. Demnach besteht ein Programmzustand aus dem aktuellen Heap und den lokalen Variablen. Der Heap wird aus einem Mapping von Speicheradressen auf Objekte gebildet. Die Objekte setzen sich aus dem Klassennamen und den Instanzen der Variablen der Klasse zusammen. Ein erweiterter Programmzustand berücksichtigt dann auch zusätzlich noch Ausnahmebehandlungen (in unserem Fall auf drei Basisausnahmen reduziert). Formal

kann man dieses wie folgt angeben:

types $obj = cname \times (vname \rightsquigarrow val)$
 $heap = loc \rightsquigarrow obj$
 $locals = vname \rightsquigarrow val$
 $state = heap \times locals$
 $xstate = xcpt\ option \times state$
datatype $xcpt = \text{NullPointer} \mid \text{ClassCast} \mid \text{OutOfMemory}$

Als Übergangsregeln zwischen zwei Programmzuständen unterscheidet man zwischen drei Formen: ein Zustand geht in einen nächsten Zustand über mit einem einfachen Statement, einem Ausdruck, der zu einem Wert ausgewertet wird, oder mit einer Liste von Ausdrücken, die zu einer Liste von Werten ausgewertet werden. Formal:

$$\begin{array}{l} prog \vdash xstate \quad -stmt \rightarrow \quad \quad \quad xstate \\ prog \vdash xstate \quad -expr \succ val \rightarrow \quad \quad xstate \\ prog \vdash xstate \quad -expr\ list[\succ]val\ list \rightarrow \quad xstate \end{array}$$

Da eine einmal auftretende Ausnahme erhalten bleibt, gilt die generelle Regel für Ausnahmen:

$$\Gamma \vdash (\text{Some } xc, \sigma) - c \rightarrow (\text{Some } xc, \sigma)$$

Dabei ist σ ein Programmzustand. Für alle anderen Regeln setzt man voraus, daß der Startzustand frei von Ausnahmen ist. Jeder weitere Zustand s_1 oder s_2 kann wiederum eine Ausnahme beinhalten, die dann nach obiger Regel erhalten bleibt.

$$\frac{\Gamma \vdash (\text{None}, \sigma) - c_1 \rightarrow s_1 \quad \Gamma \vdash (\text{None}, \sigma) - c_2 \rightarrow s_2}{\Gamma \vdash (\text{None}, \sigma) - c_1; c_2 \rightarrow s_2}$$

Für einen Programmzustand muß ein Folgezustand berechenbar sein. Dazu gibt es eine Reihe von Berechnungsregeln, die an dieser Stelle nicht alle aufgeführt werden können. Beispielhaft sei hier die Berechnungsregel für einen Ausdruck **New** C angegeben. Dieser Ausdruck wird zu einer Adresse ausgewertet. Diese Adresse darf auf dem Heap bislang noch keinen Eintrag haben und der Heap wird um einen Eintrag für diese Adresse erweitert. Dann folgt aus dem alten Programmzustand mit dem Ausdruck **New** C der neue Programmzustand mit dem nun erweiterten Heap.

$$\frac{h(\text{the_Addr } a) = \text{None} \quad h' = h[a \mapsto \text{Some}(C, \text{init_vars}(\text{fields}(\Gamma, C)))]}{\Gamma \vdash (x, h, l) - \text{New } C \succ a \rightarrow (x, h', l)}$$

Dabei sind `the_Addr` und `init_vars` wie folgt definiert:

$$\begin{array}{l} \text{the_Addr}(\text{Addr } l) = l \\ \text{init_vars} \quad \quad \equiv \text{map_of} \circ \text{map}(\lambda(n, T).((\text{fst } n), \text{default_val } T)) \end{array}$$

Durch die Verwendung der Methode `init_var` werden die Felder der neu erzeugten Klasse sofort mit den Standardwerten initialisiert. Eine Initialisierung durch den Benutzer entfällt. Weitere Berechnungsregeln finden sich in der Literatur bei [NvO98].

Mit diesem vorgestellten Modell lassen sich nun in der HOL von *Isabelle* Java-Programme modellieren. Dies ist zwar derzeit nur auf recht einfache Programme beschränkt, das Modell ist aber, wie bereits angedeutet, durchaus auch erweiterbar, so daß auch komplexere Programme abgebildet werden können. Die Verwendbarkeit der so modellierten Programme wird im folgenden Kapitel an einem Beispiel erläutert.

4 Nachweis der Typkorrektheit von Java

Als Beispiel der Einsetzbarkeit des beschriebenen Modells zur Verifikation soll nun der Nachweis der Typkorrektheit von Java betrachtet werden. Das oben beschriebene Modell läßt sich allgemein schon mit *Isabelle* verifizieren, ohne daß ein konkretes Programm anzugeben ist. Dadurch ist es nun möglich, auch eine allgemeingültige Aussage über die Typkorrektheit der Sprache Java, bzw. hier der eingeschränkten Sprache, zu machen. Es müssen dabei die aus der Java Sprachreferenz hergeleiteten Regeln für korrekte Typung von Java-Programmen in ihrer allgemeinen Form verifiziert werden.

Dazu muß betrachtet werden, was für die Typkorrektheit einer Programmiersprache gilt. Eine Programmiersprache kann nur typkorrekt sein, wenn sie ein definiertes Typkonzept hat. Programmiersprachen ohne Typkonzept können nicht typkorrekt sein. Ferner ist die Typkorrektheit einer Programmiersprache genau dann erreicht, wenn ihr Typkonzept verhindert, daß es zu einer inkorrekten Typzuweisung (*type mismatch*) kommt, d.h. der Compiler sollte beim Versuch einer fehlerhaften Zuweisung bereits einen Fehler melden. Bei der Typkorrektheit werden Fälle, wie die Division durch 0 oder die Nichtterminierung, nicht berücksichtigt, es geht vielmehr um die syntaktisch korrekte Typung. Bei einer objekt-orientierten Sprache wie Java muß für die Typkorrektheit zusätzlich gelten, daß ein Methodenaufruf auch immer eine ausführbare Methode findet. Andernfalls ist eine eindeutige Aussage nicht möglich.

Um die Typkorrektheit zu zeigen, muß eine Invariante betrachtet werden. Diese Invariante besagt, daß alle Werte zur Laufzeit mit ihrem deklarierten Typ übereinstimmen. Das beinhaltet auch, daß jeder Wert auch vom Untertyp seines deklarierten Typs sein kann. Um diese Vorstellung zu prüfen, geht man schrittweise vor. Zunächst werden einzelne Werte geprüft, dann weitet man die Betrachtung auf die Mappings der Werte aus, um schließlich einen ganzen Zustand auf Übereinstimmung mit einer Typumgebung zu prüfen. Die Übereinstimmungsprüfung hat die folgende Form:

$$\begin{aligned} prog, heap &\vdash val :: \prec ty \\ prog, heap &\vdash (\beta \rightsquigarrow val) [:: \prec] (\beta \rightsquigarrow ty) \\ &state :: \prec env \end{aligned}$$

Außerdem sind hierfür die folgenden Definitionen gegeben:

$$\begin{aligned}
\text{obj_ty } o &\equiv \text{ case } o \text{ of None} \rightarrow \text{None} \mid \text{Some}(C, fm) \rightarrow \text{Some } C \\
\Gamma, h \vdash v :: \preceq T &\equiv \exists S : \text{typeof } (\text{obj_ty} \circ h) v = \text{Some } S \wedge \Gamma \vdash S \preceq T \\
\Gamma, h \vdash vm[:: \preceq] Tm &\equiv \forall n T : Tm \ n = \text{Some } T \rightarrow (\exists v : vm \ n = \text{Some } v \\
&\quad \wedge \Gamma, h \vdash v :: \preceq T)
\end{aligned}$$

Die Invariante muß zu jedem Zeitpunkt des Programms gelten.

Ein weiterer wichtiger Aspekt ist in einem Lemma zur Typkorrektheit gegeben. Das Lemma beschreibt eine weitere Invariante, die ausdrückt, daß ein Objekt während der Ausführung des Programms nie verloren geht und seinen Typ beibehält. Das gilt für alle Objekte, die einmal instanziiert worden sind. Vorher ist keine Aussage über die Typkorrektheit eines Programmes möglich. Das Lemma berücksichtigt nicht die Garbage Collection. Um dieses Lemma formal auszudrücken, sei die folgende Relation für eine Heap-Erweiterung gegeben:

$$h \trianglelefteq h' \equiv \forall a C fm : a = \text{Some}(C, fm) \rightarrow (\exists fm' : h' a = \text{Some}(C, fm'))$$

Auf dem erweiterten Heap h' muß wieder ein Eintrag für die Adresse a , die auf ein Objekt mit dem Klassennamen C verweist, enthalten sein. Mit dieser Relation läßt sich nun das Ziel des Beweises der Invariante angeben:

$$\begin{aligned}
(\Gamma \vdash (x, h, l) - c \rightarrow (x', h', l')) &\quad \rightarrow h \trianglelefteq h' \quad \wedge \\
(\Gamma \vdash (x, h, l) - e \succ v \rightarrow (x', h', l')) &\quad \rightarrow h \trianglelefteq h' \quad \wedge \\
(\Gamma \vdash (x, h, l) - es[\succ]vs \rightarrow (x', h', l')) &\quad \rightarrow h \trianglelefteq h'
\end{aligned}$$

Die Invariante wird von *Isabelle* nach dem Prinzip der simultanen Regelinduktion bewiesen. Unabhängig von der Art des Zustandsübergangs muß die Relation zwischen dem Heap des alten und des neuen Zustands erfüllt sein. D.h. keine der drei Arten eines Zustandsübergangs darf dazu führen, daß ein Objekt vom Heap entfernt wird oder seinen Typ ändert.

Zur Typkorrektheit muß nun ein Hauptsatz verifiziert werden. Dieser Satz sagt aus, daß es für wohldefinierte Programme bei der Ausführung von typkorrekten Statements und der Auswertung von Ausdrücken eine Übereinstimmung der dabei verwendeten Typen mit den jeweils deklarierten Typen gibt.

$$\begin{aligned}
\text{wf_prog } \Gamma \longrightarrow & (\Gamma \vdash (x, h, l) - c \rightarrow (x', h', l')) \longrightarrow \\
& \forall lT : (h, l) :: \preceq (\Gamma, lT) \longrightarrow \\
& (\Gamma, lT) \vdash c \checkmark \longrightarrow \\
& (h', l') :: \preceq (\Gamma, lT)) \\
\wedge & (\Gamma \vdash (x, h, l) - e \succ v \rightarrow (x', h', l')) \longrightarrow \\
& \forall lT : (h, l) :: \preceq (\Gamma, lT) \longrightarrow \\
& (\Gamma, lT) \vdash e :: T \longrightarrow \\
& (h', l') :: \preceq (\Gamma, lT) \wedge h \trianglelefteq h' \wedge (x' = \text{None} \longrightarrow \Gamma, h' \vdash v :: \preceq T)) \\
\wedge & (\Gamma \vdash (x, h, l) - e[\succ]v \rightarrow (x', h', l')) \longrightarrow \\
& \forall lT : (h, l) :: \preceq (\Gamma, lT) \longrightarrow \\
& (\Gamma, lT) \vdash e[::]T \longrightarrow \\
& (h', l') :: \preceq (\Gamma, lT) \wedge h \trianglelefteq h' \wedge \\
& (x' = \text{None} \longrightarrow \text{list_all}(\lambda v T. \Gamma, h' \vdash v :: \preceq T) \ vs \ Ts))
\end{aligned}$$

Dabei ist die Funktion `list_all` wie folgt definiert:

$$\text{list_all } P \text{ } xs \text{ } ys \equiv (\text{length } xs = \text{length } ys) \wedge (\forall (x, y) \in \text{zip } xs \text{ } ys : P \text{ } x \text{ } y)$$

Um diesen Hauptsatz etwas lesbarer zu gestalten, können die drei Teile des Satzes umformuliert werden. Das Beweisziel des ersten Teils, in dem Zustandsübergänge durch Ausführen eines Statements betrachtet werden, kann dann wie folgt formuliert werden:

$$\begin{aligned} & \text{wf_prog } \Gamma \\ \wedge & \quad E = (\Gamma, lT) \\ \wedge & \quad E \vdash c \checkmark \\ \wedge & \quad \Gamma \vdash (-, h, l) - c \rightarrow (-, h', l') \\ \wedge & \quad (h, l) :: \preceq E \\ \longrightarrow & \quad (h', l') :: \preceq E \wedge h \preceq h' \end{aligned}$$

Im Kontext eines wohldefinierten Programms geht ein Zustand unter Ausführung eines wohldefinierten Statements in einen Folgezustand über, der wiederum typkonform zur betrachteten Umgebung ist.

Auch der zweite Teil, in dem Zustandsübergänge durch Auswertung von Ausdrücken zu Werten betrachtet werden, läßt sich ebenfalls lesbarer formulieren:

$$\begin{aligned} & \text{wf_prog } \Gamma \\ \wedge & \quad E = (\Gamma, lT) \\ \wedge & \quad E \vdash e :: T \\ \wedge & \quad \Gamma \vdash (-, h, l) - e \succ x \rightarrow (-, h', l') \\ \wedge & \quad (h, l) :: \preceq E \\ \longrightarrow & \quad (h', l') :: \preceq E \wedge h \preceq h' \wedge (x' = \text{None} \longrightarrow \Gamma, h' \vdash v :: \preceq T) \end{aligned}$$

Auch hier gilt, daß im Kontext eines wohldefinierten Programms ein Zustand unter Auswertung eines wohldefinierten Ausdrucks zu einem Wert in einen Folgezustand übergeht, der wiederum typkonform zur betrachteten Umgebung ist. Zusätzlich folgt daraus unter der Voraussetzung, daß keine Ausnahme aufgetreten ist, daß der resultierende Wert typkonform zum ausgewerteten Ausdruck ist. Analog hierzu läßt sich auch der verbleibende dritte Teil des Satzes umformulieren. Dort wird statt einzelner Ausdrücke eine Folge von Ausdrücken betrachtet.

Ein Korollar des Beweises ist, daß beim Methodenaufruf immer eine anwendbare Methode gefunden wird, weshalb zum Beispiel das Auftreten eines "method not understood" Fehlers zur Laufzeit nicht möglich ist.

Um den Hauptsatz zu prüfen, erfordert es weitere komplexe Hilfslemmata, die hier nicht weiter ausgeführt werden. In [NvO98] werden einige der Hilfslemmata kurz angesprochen. *Isabelle* weist diesen Satz ebenfalls per simultaner Regelinduktion nach. Der Induktionsbeweis besteht auf der höchsten Ebene aus 19 verschiedenen Fällen, die zu prüfen sind. Davon sind 7 Fälle direkt nach Induktionsvoraussetzung zu beweisen, 4 Fälle basieren auf einfachen Lemmata und der Zustandsstruktur und 8 Fälle erfordern eine umfangreiche Argumentation über die charakteristischen Eigenschaften der entsprechenden Konstrukte. Um diesen Beweis in *Isabelle* tatsächlich durchführen zu können, sind etwa 2400 Zeilen

Skriptcode notwendig. Aus diesen Gründen wird für tiefere Betrachtungen des Beweises auf die angegebene Literatur verwiesen.

Der Typkorrektheitsbeweis dient hier nur als Beispiel der Einsetzbarkeit der vorgestellten Modellierung der Sprache Java. Auch andere Beweise sind denkbar, sofern die Ziele des Beweises in der HOL von *Isabelle* formuliert werden können.

5 Ausblick: Möglichkeiten der Bytecode-Verifikation

Bislang hat sich diese Arbeit auf die Möglichkeit der Verifikation von Java-Programmen, die im Quelltext vorliegen, bzw. der Sprache Java beschränkt. In der Praxis ist es aber ebenso interessant, auch ein Programm zu untersuchen, welches als Bytecode vorliegt. Zum Abschluß soll noch ein Ausblick gegeben werden, wie man auch Bytecode verifizieren kann.

Die Java Virtual Machine (JVM) ist die abstrakte Maschine, die den vom Compiler aus Java Quellcode erzeugten Bytecode ausführt. Da dieser Bytecode plattformunabhängig ist (bzw. sein soll) und vom Interpreter noch plattformabhängig übersetzt werden muß, besteht der Bytecode aus assembler-ähnlichen Instruktionen, die über den Stack operieren. Ähnlich wie die JVM die Instruktionen aus dem Bytecode zur Laufzeit ausliest und ausführt, kann man diese Instruktionsfolge aus dem Bytecode auch auslesen und verifizieren. Diese nun entstandene Instruktionsfolge kann in ähnlicher Weise, wie es in der Arbeit bereits beschrieben ist, modelliert werden und mit Verifikationsskripten für *Isabelle* verifiziert werden.

Diese Folge von Instruktionen kann man auch formal als eine Liste von Instruktionen ansehen. Diese Instruktionen können dann als Datentyp der HOL angegeben werden. Beispielsweise ist folgende Deklaration denkbar:

```
datatype instr = LS load_and_store
               | CO create_object
               | MO manipulate_object
               | CH check_object
               | MI meth_inv
               | MR meth_ret
               | OS op_stack
               | BR branch
```

Auch hier werden Arrays und Interfaces nicht berücksichtigt, das Modell ist aber in dieser Richtung erweiterbar. Auch wurde von Typkonvertierungen Abstand genommen, da nur ein numerischer Typ *int* verwendet werden soll. Bei der Benennung wurde weitestgehend die Nomenklatur der JVM verwendet. Die genaueren Spezifikationen sind in [Nip01] und [NOP00] beschrieben.

Hat man den Bytecode modelliert, muß man noch Aussagen über die operationelle Semantik dieser Instruktionsfolgen treffen. Diese werden, ähnlich wie bereits in Kapitel 3

gezeigt, in analoger Weise modelliert. Auch hier werden Zustände benötigt, die den aktuellen Heap, den Stack, die Liste der lokalen Variablen, usw. beschreiben. Auf Basis dessen kann man auch für den Bytecode Typrelationen aufstellen und Regeln für eine korrekte Typung bestimmter Instruktionssequenzen angeben.

Nach dieser Modellierung kann zum Beispiel ein Typkorrektheitsbeweis auch auf dem Bytecode durchgeführt werden. Für den Bytecode gelten andere Bedingungen und Abhängigkeiten für eine Typkorrektheit, da der Bytecode nicht so viele Informationen enthält, wie sie das allgemeine Modell der Sprache Java für den Beweis hergibt. Eine Übersicht über die für den Typkorrektheitsbeweis notwendigen Formalisierungen und Bedingungen finden sich in [Nip01] und [NOP00].

Literatur

- [Nip01] Tobias Nipkow. Verified bytecode verifiers. In M. Miculan F. Honsell, editor, *Foundations of Software Science and Computation Structures (FOSSACS 2001)*, volume 2030, pages 347–363, 2001.
- [NOP00] Tobias Nipkow, David von Oheimb, and Cornelia Pusch. μ Java: Embedding a programming language in a theorem prover. In F.L. Bauer and R. Steinbrüggen, editors, *Foundations of Secure Computation. Proc. Int. Summer School Marktoberdorf 1999*, pages 117–144. IOS Press, 2000.
- [NvO98] Tobias Nipkow and David von Oheimb. *Java_{light}* is type-safe — definitely. In *Proc. 25th ACM Symp. Principles of Programming Languages*, pages 161–170. ACM Press, 1998.