UNIVERSITÄT AUGSBURG
Universitätsbibliothek

OPEN ACCESS
OPUS AUGSBURG

Verification of Java Card Programs

Kurt Stenzel

Angaben zur Veröffentlichung / Publication details:

Stenzel, Kurt. 2005. “Verification of Java Card Programs.” Augsburg: Universität Augsburg.

Nutzungsbedingungen / Terms of use:

licgercopyright

Dieses Dokument wird unter folgenden Bedingungen zur Verfügung gestellt: / This document is made available under these conditions:
Deutsches Urheberrecht
Weitere Informationen finden Sie unter: / For more information see:
https://www.uni-augsburg.de/de/organisation/bibliothek/publizieren-zitieren-archivieren/publiz/

Verification of Java Card Programs

Dissertation
zur Erlangung des Doktorgrades Dr. rer. nat.
der Fakultät für Angewandte Informatik
der Universität Augsburg
im Jahr 2005 von

Kurt Stenzel

Amtierender Dekan:	Prof. Dr. Wolfgang Reif
Gutachter:	Prof. Dr. Wolfgang Reif Prof. Dr. Bernhard Bauer
Tag der Prüfung:	30. Mai 2005
Prüfer:	Prof. Dr. Bernhard Bauer Prof. Dr. Bernhard Möller Prof. Dr. Wolfgang Reif Prof. Dr. Theo Ungerer

Summary

Smart cards are used in security critical applications where money or private data is involved. Examples are the German Geldkarte or new passports with biometrical data. Design or programming errors can have severe consequences. Formal methods are the best means to avoid errors. Java Card is a restricted version of Java to program smart cards. This work presents a logical calculus to formally prove the correctness and security of Java Card programs. The calculus is implemented in the KIV system, and ready for use. First, an operational big-step semantics for sequential Java is presented based on algebraic specifications. All Java language constructs are modeled. Then, a sequent calculus for dynamic logic for Java Card is developed, and the correctness of the calculus is formally proved. The calculus is designed to support libraries, the reuse of proofs, and program modifications. This entails two different notions of type soundness, the standard one, and a weaker version. Furthermore, the calculus is not restricted to Java Card, but can be used for arbitrary sequential Java program. The work ends with some intricate examples. All properties and theorems are formally proved with the KIV system. The resulting verification system is able to cope with real-life e-commerce applications.

Acknowledgments

I wish to thank my supervisor Wolfgang Reif for his support, patience, trust, and guidance –
without him this work would not exist.

I wish to thank the members of the KIV group, especially Gerhard Schellhorn, Michael Balser, and Christoph Duelli for the years they have spent for improving the KIV system –
without them this work would not have been possible.

I wish to thank all my colleagues, particularly Dominik Haneberg and Holger Grandy, for many discussions and demands (yes!) –
without them this work would be less than it is.

Contents

1 Introduction **7**

2 Java Programs **13**
2.1 Expressions . . . 13
2.2 Statements . . . 15
2.3 Java Type Declarations . . . 17
2.4 Assumptions about the Compiler . . . 18
2.5 Discussion . . . 19

3 Semantics **23**
3.1 The Evaluation Environment . . . 23
3.2 Semantics of Expressions . . . 30
3.3 Semantics of Statements . . . 46
3.4 Proof Technique . . . 55

4 The Calculus **63**
4.1 Overview . . . 63
4.2 Semantics of Formulas and Sequents . . . 69
4.3 Expressions . . . 71
4.4 Statements . . . 87
4.5 Generic Rules . . . 94
4.6 Additional Rules . . . 99
4.7 Predicate Logic Proof Rules . . . 103
4.8 Soundness of the Calculus . . . 104

5 Type Safety **109**
5.1 Primitive Type Correctness . . . 110
5.2 Definition of Primitive Type Correctness . . . 113
5.3 Properties of Primitive Type Correct Programs . . . 117
5.4 Full Type Correctness . . . 118

6 Libraries and Modifications **127**
6.1 Java Libraries in KIV . . . 127
6.2 Extending Type Declarations . . . 129
6.3 Context dependency of Proof Rules . . . 130
6.4 Axioms for the Class Hierarchy . . . 131
6.5 Detailed Analysis of Dependencies . . . 133
6.6 Theorems in Libraries . . . 135
6.7 Libraries and Compatibility . . . 135
6.8 Context Modifications . . . 137

7 Four Examples **139**
7.1 A CopyCard 139
7.2 The Java Card API 144
7.3 Decimal Numbers 150
7.4 Linked Lists 156
7.5 An example proof 161

8 Conclusion **165**

Bibliography **167**

Index **175**

Chapter 1

Introduction

Overview

When Java appeared in the mid-nineties it rapidly became an object of big interest in the research community. The main reason is that Java joins a number of interesting concepts. It is object oriented, contains threads, uses a virtual machine to run byte code, has a byte code verifier, has a sandbox concept to run untrusted code, etc. (Other reasons are: it is backed by a well-known company; large libraries, compiler, and virtual machine are freely available on different platforms.) Furthermore, the language has been designed from scratch, is rather small, and is intended to be programmer-friendly; i.e tries to prevent the programmer from making errors. This makes Java a worthwhile target for a formal treatment, but also poses a challenge to the maturity of formal methods (and their supporting tools) since it is no toy language. For program verification, however, another target is needed: programs that are worth proving correct, and that are not too big. As it turns out, smart cards programmed in Java Card are an almost perfect target. This thesis is concerned with the verification of sequential Java programs with a focus on Java Card. All work has been done with KIV (described below).

Smart Cards and Java Card

Smart Cards are credit card sized plastic cards with an embedded chip that contains a microprocessor together with memory, i.e. a full – though small – computer. Without the enclosing card the chip can be found, for example, in mobile phones as the Subscriber Identity Module (SIM). Smart card applications often involve money or sensitive data; their importance is increasing. Some examples:

- The German Geldkarte is based on smart cards.
- Several credit card companies have begun to issue smart cards to reduce fraud.
- Next year, passports will contain embedded chips with biometric data to identify terrorists.
- Also next year, the German health care card will be issued, a smart card.

These applications are highly security critical; millions of cards have been and will be issued. Breaking the security of these cards – possibly due to a programming error – in a systematic manner would be devastating. On a smaller scale, smart cards are used in universities (as student cards and to pay for lunch), in larger companies for employees, in public transport, or in fitness studios for access and payment. Embedded chips are used in mobile phones, for pay-TV applications, in car keys to avoid theft. Since security is the most important aspect in these applications, they are are a worthwhile target for program verification (among others; security of the protocols, the cryptographic methods, and physical security is equally important).

Java Card [Jav00] [Che00] is a reduced version of Java designed for smart cards. Smart cards are full computers, but have very limited resources, e.g. 64 KByte EEPROM for code and data, 4 KByte RAM, and a 8-bit processor running with 5 MHz. This means that a fully-fledged Java virtual machine does not fit on a smart card. Therefore, in Java Card all resource consuming features have been discarded: threads, garbage collection, streams, floating point arithmetic, characters and strings, and integers. Essentially, Java Card is sequential Java with fewer primitive data types (only boolean, byte, and short remain) and a much smaller (and different) API. A normal Java compiler is used to convert the source code into byte code, but then a converter must be used that converts the byte code to a more condensed form that can be loaded onto a smart card. The converter also checks that no unsupported features (like integers, strings, etc.) are used in the byte code. (This is sometimes called off-card or off-line byte code verification.) This means that a formal Java semantics is also a Java Card semantics.

The Thesis in a Nutshell

Chapter 2 describes the abstract syntax for Java expressions, statements, and type declarations.

Chapter 3 then describes a run time semantics for sequential Java. It is a natural (big-step) operational semantics based on algebraic specifications and specified in the KIV system. It includes the full language with two trivial exceptions (described in Chapter 2). Inner classes are not modeled. Special emphasis is put on the correct specification of operations on primitive types. Instead of being as short as possible, the aim was to be a readable as possible. Therefore, 126 rules are specified. The semantics uses an explicit store that is specified using algebraic specifications. No assumptions about type correctness of the program is made. The semantics is formally proven to be well-defined and deterministic.

Chapter 4 describes the calculus. It is a sequent calculus using a dynamic logic (which is more expressive than Hoare logic) for the Java programs. The calculus is formally proven sound with respect to the semantics, and implemented in the KIV system. The dynamic logic is based on algebraic specifications. This gives the user the freedom to specify auxiliary predicates and functions. Because of the explicit store, reasoning about arbitrary pointer structures is possible. Furthermore, it is possible to express that a program terminates with an exception. The calculus has been designed to be efficiently usable and extendable. For example, the Java type declarations are not part the formulas. The calculus is targeted at Java Card application, not specifically to support a general object-oriented design methodology.

Chapter 5 contains the formal type soundness proof. Two notions of type correctness are introduced: *primitive* and *full* type correctness. Primitive type correctness makes no assumptions about the store, and ensures that every expression has a value that is compatible with its type when the class hierarchy is ignored. Full type correctness guarantees that at run time the value of every expression is a subtype (i.e. sub class for objects) of its static type. This requires a compatible store. The type soundness proof handles binary operations and conversions correctly; it allows nested blocks and local variable declarations anywhere in a block; `return` statements may occur anywhere in method bodies; they can be freely mixed with `break` statements.

Chapter 6 is concerned with libraries and modifications. The idea of a library is to have a set of useful classes and theorems that is used in different applications. When the library is imported the class declarations are extended, and the question is whether the library theorems are still valid. The answer is “yes”, but this requires a carefully designed calculus. This also requires that theorems (and the calculus) do not rely on a compatible store, so that full type correctness is nice, but not useful for libraries. The rest of the chapter then describes a correctness management for Java theorems that analyses what theorems are still valid when methods, classes, fields etc. are modified.

Chapter 7 contains four examples that illustrate different possibilities of the calculus. The first is a Java Card e-commerce application, the second shows how the Java Card API can be treated as a library, the third is an example of difficult byte and short arithmetic, and the fourth is a non-Java Card example that shows how cyclical pointer structures can be verified.

KIV

KIV is a proof system with quite a long tradition [HHRS86] [HRS91] [Rei95] [BRS+00], the author of this thesis is one of its developers. It has been used in large case studies [SB94] [FRSS95] [SA98] [ORS+02] The core of KIV are structured algebraic specifications [Wir90] [CoF04] [Rei91] [Rei92a] [Rei93] with an excellent proof support for them. Algebraic specifications can be used to specify a complete software system; or they allow a user to specify arbitrary data types and operations like bounded integers, floating point arithmetic, graphs, pointer structures, etc. They are heavily used in this work. A library of standard data types contains thousands of theorems, and is constantly enlarged.

Based on the algebraic specification the system has been extended in different directions: imperative Pascal-like programs and modules [Rei91] [Rei92b] [Rei95], abstract state machines [Sch99] [Sch01], temporal logic and parallel programs [BDRS02], state charts [TSOW04] and fault trees [TS03], and now Java. Independent from the logics KIV has

- explicit proof trees that can be saved to disk, inspected, pretty printed, reused, etc.
- a graphical user interface for structured specifications, proof trees, formulas etc. Proof rules can be applied by clicking on the relevant part of a formula [HBB+05].
- a correctness management that analyses which proofs become invalid after a modification, and which are still valid. [RSSB98] (The correctness management must be extended for a new calculus.)
- the capability to manage and use tens of thousands of theorems and proofs [RS97].

The invaluable benefit for the work of this thesis was that these features became available for the new Java calculus with only minor effort (for a KIV developer).

Achievements

The following achievements were accomplished with this work:

- A formal natural semantics for the largest part of sequential Java was defined.
- A type soundness proof with the fewest simplifications up to date has been done.
- A dynamic logic calculus together with a formal correctness proof has been developed. This is up to date unique.
- The proofs revealed several otherwise-almost-impossible-to-detect errors.
- The formal correctness proof is in itself a very large case study.
- The calculus was designed for efficiency and library creation; it has a built-in correctness management.
- An existing prover (KIV) was extended. This is unique for Java calculi.
- The calculus allows to reason about arbitrary pointer structures, and has special support for Java Card.
- Several intricate examples have been done as illustration.

Related Work

The language reference for Java is *The Java Language Specification* (abbreviated JLS in the rest of this work). The first edition [GJS96] is from 1996, the second [JSGB00] from 2000. While the descriptions in this book are informal, they are quite precise, and often contain detailed algorithms in natural language. Often they can be translated directly into a formal specification. Part of the

book describes the checks that must be performed by the compiler, another part describes the run time behavior of the Java statements and expressions. The style used in the latter is operational, and sometimes called *natural* or *big-step* semantics [NN98]. While often precise, the book contains some omissions and ambiguities that must be resolved when formalizing Java.

In 1999, the book *Formal Syntax and Semantics of Java* [AF99] appeared. As the title indicates, it contains a collection of (unrelated) papers with a formal approach to Java's syntax and semantics. These papers are mostly updates of previously published versions. Börger and Schulte [BS99] present a Java semantics based on abstract state machines (ASMs [Gur95]); later expanded into a book [SSB01]. It covers a large part of the language (including threads), but omits 'syntactical sugar' (e.g. `for` and `do` loops, postfix increment, compound assignment etc.). The definition should be called 'mathematical' instead of 'formal', because the authors do not use a fixed syntax for their ASM, nor did they specify the ASM in a formalism or tool. Many assumptions are only mentioned, but not defined. For example, they assume that every execution path of a method body ends with a `return` statement, but never specify this in the ASM context. Furthermore, the ASM formalism requires to reason about occurrences of code fragments in a Java program (discussed in [Gle03]). This leads to a technical overhead that is also not defined precisely. It also implies that there is a rather large distance to the description in the *Java Language Specification*. The semantics is short, but very dense (e.g. they use macros that are redefined in the text), and therefore quite difficult to read. The author of this dissertation specified the ASM in KIV. The formalization of the technical details makes the semantics even more difficult to read. This eventually lead to the natural semantics presented in this work. Several ideas concerning data types were inspired by the ASM formalization, e.g. the very simple Java store with keys that are a pair of a reference and a field specification or an array index.

[AF99] contains different formal semantics, type soundness proofs, but no papers about Java program verification. However, some work already existed. Hartel and Moreau [HM01] give a survey of the literature of formal treatments of Java up to 2001. There is only a handful of groups that are concerned with tool supported Java verification. Most of them focus on Java Card; the EU project VerifiCard [Ver] was devoted to Java Card.

Von Oheimb and Nipkow [NvO98] [vON99] [vO00] [vO01] have formalized Java in Isabelle/HOL [Pau94a] [Isa]. Oheimb's dissertation [vO01] contains a natural (big-step) semantics for a simplified version of sequential Java, a proof of type soundness, and a Hoare calculus together with correctness and completeness proof. All this is done in Isabelle, and the formal completeness proof is up to now unique, at least for Java. The nice thing about this work is that it is all-in-one. Only one formalism and one tool is used; there is no translation, or encoding; and a deep embedding is used. Furthermore, the complete specification is quite short (56 pages). On the other hand, this makes it sometimes difficult to see the correspondence between the formal specification and the description in JLS. For example, the semantics rule for method invocation contains no null pointer test that is obviously visible (there is only one rule for static and instance method invocation). This test is 'buried' in the function that binds the actual arguments to the formal parameters when the method call is replaced by the method body; raising a null pointer exception is done by the two-letter function `np`. Actually, the formalized language is called Java^{light}; "Statements are reduced to their bare essentials.". The dissertation is mainly concerned with the meta aspects (correctness and completeness) of the calculus, not with using the calculus in applications. Accordingly, only one small example is included. The rules of the calculus are theorems in Isabelle/HOL; using the calculus essentially means to apply these theorems on a goal and simplify with Isabelle. His work will be referenced throughout this thesis. Later work by Nipkow and his group deal with other aspects of Java [ON02] [Sch03] or introduce Hoare calculi for rather generic imperative/object oriented languages [Nip02b] [Nip02a] [Sch05].

In the KeY project [KeY] [ABB+04] [ABB+03] a dynamic logic for Java Card [Bec00] was developed. So superficially the calculi are similar. One major difference to this work is that no own formal semantics was specified (hence type soundness was not proved), and that no formal correctness proof for the calculus exists. Furthermore, the emphasis is on fully automatic proofs instead of support for interactive proofs. A new prover was implemented from scratch for the calculus together with proof support for the data types. Current work is more on extensions, e.g.

the calculus was extended to incorporate Java Card's transaction mechanism [BM03] [HM05], and properties can be formulated using OCL [LM04]. The differences in the calculus will be discussed in chapter 4.

All other calculi are either Hoare-style or weakest precondition calculi. Often, the *Java Modeling Language* (JML [BCC+03] [LLP+00] [JML]) is used to annotate Java programs (invariants for loops etc.), or to specify pre- and post conditions for methods, or class invariants. JML is intended to be used by a Java programmer, hence it has a Java-like syntax and semantics with some extensions. The core is formed by the functional Java expressions (boolean and binary operators, etc.) with extensions like quantifiers. Since JML does not include a full store model it is not possible to express complex relations between objects, for example that a pointer structure forms an acyclical list. This automatically limits the properties that can be proved/expressed with a calculus based on JML.

Huisman [Hui01] in her dissertation defined a Java semantics and a Hoare calculus in PVS and Isabelle. A shallow embedding is used; the memory model is based on coalgebras. The calculus has been proven sound in PVS and Isabelle (completeness or type soundness was not considered). There is no strict division between the semantics and the calculus; the rules of the calculus are theorems with respect to the semantics, and both can be mixed. The dissertation includes two non-trivial examples.

The LOOP tool [JP03] [vdBJ01] translates Java source code with JML annotations into PVS, using the theories defined by Huisman (with some modifications). Here, the focus has been shifted to the verification of Java Card applications, e.g. [BJvdB02], [JMR04]. The proof rules of a weakest precondition calculus for Java+JML have been added as theories to PVS. Not surprisingly, the result of applying the wpc as a verification condition generator is large: "PVS can run for hours without completing the proof, or it can crash because the proof state becomes too big." ([JP03]).

Other tools like KRAKATOA [Kra] [MPMU04] and JACK [BRL03] also start with a JML annotated Java program. They have an internal verification condition generator (based on weakest precondition) that generates proof obligations that no longer contain Java statements or expressions. They can be proved with existing tools, e.g. with Coq [Coq] (in case of KRAKATOA), or the B-tool [Abr96] or Isabelle in case JACK. In neither case a formal proof for the correctness of the generator exists. The focus is again on Java Card applications. The JIVE tool [MPH00] is a dedicated prover that uses a Hoare-style calculus for Java programs (support of JML annotations is planned). No machine checked correctness proof exists for the calculus. The predicate logic parts can be proved with Isabelle or PVS. Obviously, these tools require an experienced user of the underlying provers for the non-Java parts.

Chapter 2

Java Programs

In this chapter we will introduce the Java programming language as it is used in the rest of this work. The basis for Java is the *Java Language Specification* [GJS96] [JSGB00] (abbreviated as JLS in the sequel) that exists in two editions. The main differences between the first and second edition are the introduction of nested type declarations (inner classes) and an extended specification (and some modifications) of floating point operations. Some errors have been corrected, but there are no modifications of the semantics of expressions and statements. The only exception is that instead of an `IndexOutOfBoundsException` an `ArrayIndexOutOfBoundsException` is thrown when an array access contains an illegal index. In the sequel we will use the second edition for references.

The Java language specification contains a grammar that can be used to obtain an abstract syntax tree from the source code. This syntax tree contains no type information, and the meaning of names has not been determined. ("Determining the meaning of a name" is JLS speech – chapter 6.5 has this title. It means that a "name" `N` occurring in a Java program in the context `N.O.P` could be the name of a local variable, or a field, or a class, or a package. This must be determined by the compiler.) The language specification then describes for every expression and statement how this is done at compile time and what checks must be performed. The result of this compilation phase can be described as an annotated abstract syntax tree. This annotated abstract syntax tree is the basis for the runtime semantics of Java that is also described in the language specification. We use an abstract syntax for Java that contains the informations of an annotated abstract syntax tree. This means that every expression contains its computed type and that the meaning of every name has been determined. In the following sections we present the abstract syntax for expressions, statements, and type declarations. The rest of the chapter contains the differences between these definitions and the Java language specification and a discussion.

2.1 Expressions

Java makes a distinction between *expressions* (described in chapter 15 JLS) and *statements* (described in chapter 14). Expressions are the basic building blocks and compute values (and may have side effects), while statements control the execution of expressions (and do not compute values).

The JLS does not specify an abstract syntax tree, and the distinction between expressions is not always clear. This means that every formalization is free to choose its own division of expressions. (This is a further item that makes a comparison between different formalizations more difficult.)

While the division between expressions and their order of presentation in JLS is more or less syntax driven the division chosen here depends more on their possible behavior. For example, a primitive cast and a reference cast are considered as two different expressions because a reference cast may throw a `ClassCastException` while a primitive cast never throws an exception. The same applies for an *ExBinExpr*, a binary expression that may throw an exception (i.e. integer

Table 2.1: **Java expressions**

Notation	**Constructor**	**arguments**
l	LiteralExpr	literal × type → javaExpr
$\oplus e$	UnaryExpr	UnOp × javaExpr × type → javaExpr
$e\oplus$	IncDecExpr	IncDecOp × javaExpr × type → javaExpr
$(ty)e$	PrimCast	type × javaExpr × type → javaExpr
$(ty)e$	RefCast	type × javaExpr × type → javaExpr
e instanceof ty	InstanceExpr	javaExpr × type × type → javaExpr
$e_1?e_2 : e_3$	CondExpr	javaExpr × javaExpr × javaExpr × type → javaExpr
$e_1 \oplus e_2$	CondBinExpr	javaExpr × CondOp × javaExpr × type → javaExpr
$e_1 \oplus e_2$	BinaryExpr	javaExpr × BinOp × javaExpr × type → javaExpr
$e_1 \oplus e_2$	ExBinExpr	javaExpr × ExBinOp × javaExpr × type → javaExpr
x	LocVarAccess	variable × type → javaExpr
f	SFieldAccess	fieldspec × type → javaExpr
$e.f$	FieldAccess	javaExpr × fieldspec × type → javaExpr
$e_1[e_2]$	ArrayAccess	javaExpr × javaExpr × type → javaExpr
$x = e$	LocVarAssign	variable × javaExpr × type → javaExpr
$f = e$	SFieldAssign	fieldspec × javaExpr × type → javaExpr
$e_1.f = e_2$	FieldAssign	javaExpr × fieldspec × javaExpr × type → javaExpr
$e_1[e_2] = e_3$	ArrayAssign	javaExpr × javaExpr × javaExpr × type → javaExpr
$e_1 \oplus= e_2$	CompAssign	javaExpr × AsgOp × javaExpr × type → javaExpr
new $c(e_1, \ldots, e_n)$	NewExpr	classname × javaExpr* × type* × type → javaExpr
new $ty[e_1]..[e_n][i]$	NewArray	type × javaExpr* × int × type → javaExpr
$\{e_1, \ldots, e_n\}$	ArrayInit	javaExpr* × type → javaExpr
$e.c(e_1, \ldots, e_n)$	ConstrCall	javaExpr × classname × javaExpr* × type* × type → javaExpr
$e.m(e_1, \ldots, e_n)$	MethodCall	javaExpr × methodname × invMode × javaExpr* × type* × type → javaExpr

division or remainder), and a *BinaryExpr* that does not. On the other hand, there is no need to distinguish between a numerical comparison (e.g. $<$) and an integer bitwise operation because the result of the operation depends only on the binary operator and the arguments; their semantics is identical otherwise.

Every expression has its result type as the last argument. This is usually omitted to improve readability. For example, we will normally use the notation $e_1 \oplus e_2$ instead of BinaryExpr(e_1, $\oplus$, e_2, ty). (The full notation will be used in chapter 5 where type correctness is the main topic.) A detailed description of the different expressions and their semantics is provided in chapter 3. Here we only provide some remarks. The additional types used in the Java expressions are summarized in table 2.4.

- A literal may contain an arbitrary algebraic term.
- The four unary operations (`UnOp`) are `+`, `-`, `~` (bitwise complement), and `!` (boolean negation).
- An *IncDecExpr* subsumes the four expressions postfix increment, prefix increment, postfix decrement, and prefix decrement.
- A *refCast* is intended for reference casts where the cast type is a class or an array type, a *primCast* for a primitive cast that converts a number.
- A *CondBinExpr* (conditional binary expression) is `&&` and `||` (the first argument is evaluated before the second); a *BinaryExpr* contains a *BinOp* that is one of `+`, `-`, `*`, `&`, `|`, `^`, `<<`, `>>`, `>>>`; an *ExBinOp* is either `/` or `%`.

- There are four kinds of accesses: local variable access (*LocVarAccess*), static field access (*SFieldAccess*) that can be a *first active use*, instance field access (*FieldAccess*) that has one invoking expression and can cause a `NullPointerException`, and array access (*ArrayAccess*) with two expressions that can additionally cause an `ArrayIndexOutOfBoundsException`. A field is described by a *field specification* (*fieldspec*) consisting of the type of the field, the class where the field is declared, and the field name (used to deal with hiding of fields).

- For every access there is a similar assignment expression (*LocVarAssign*, *SFieldAssign*, *FieldAssign*, and *ArrayAssign*). A compound assignment (*CompAssign* will have as its left hand side one of the four accesses. The possible assign operators are `+=`, `-=`, `*=`, `/=`, `%=`, `&=`, `|=`, `^=`, `<<=`, `>>=`, and `>>>=`.

- A class instance creation expression *NewExpr* consists of a class name, the list of actual arguments, and a list of types. This list of types are the formal types of the constructor declaration that was chosen by the compiler (JLS 15.9.3). The class name together with the formal types form the *constructor signature* and will be used at run time to determine the correct constructor declaration.

- A new array is created with the *NewArray* expression. It contains the element type of the array, the list of dimensions and the number of additional dimensions, e.g. an expression `new byte[3][7][][]` has the abstract syntax NewArray(byte_type, 3 + 7, 2, array_type(byte_type,4)) (here 3 + 7 denotes a list with two elements, and the result type is a four dimensional array of bytes).

- A constructor call *ConstrCall* is used for a `this` or `super` call at the beginning of a constructor. It contains the class name of the constructor to invoke and the list of formal types (i.e. the constructor signature as in a *NewClass* expression). The constructor call will also be used to describe the semantics of a class instance creation expression.

- There is one expression for a method call *MethodCall*. It contains an invoking expression, a method name, an invocation mode, the list of actual arguments, and the list of formal types used to determine the correct method declaration. The invocation mode is one of *nonVirtual*(c), *super*(c), *virtual*, *static*(c), and *interface*. Here, c is the class to search (or in case of *super*(c) the class where the search begins) for the method declaration.

The next two sections will describe the abstract syntax for Java statements and type declarations. The last section of this chapter contains a discussion of some of the choices and assumptions.

2.2 Statements

Statements are described in chapter 14 JLS. Statements control the execution of expressions (and do not compute values). Again there are choices regarding the distinction between statements, but the situation is simpler than for expressions. For example, we distinguish between a return statement with an expression and one without. On the other hand, there are more differences between JLS and the statements used in this work than for expressions. Table 2.2 lists the abstract syntax for statements.

Some statements are missing. The reasons for these decisions are discussed in detail in section 2.5. Other statements have slightly different arguments. They are discussed below and in more detail when their semantics is defined. A novelty (compared to expressions and most other Java formalizations) is the fact that seven new statements (*static*, *endstatic*, *target*, *targetExpr*, *catches*, *finally*, and *endfinally*) are introduced. They are used by the calculus to deal with static initialization and jumps, respectively. A 'correct' Java type declaration does not contain these statements.

Some initial remarks concerning statements:

Table 2.2: **Java statements**

Notation	**Name**	**arguments**
$\{\alpha_1 \dots \alpha_n\}$	Block	javaStm* → javaStm
$ty\ x = e;$	LocVarDecl	type × variable × javaExpr → javaStm
$e;$	ExprStm	javaExpr → javaStm
if $(e)\ \alpha$ **else** β	If	javaExpr × javaStm × javaStm → javaStm
$l : \alpha$	Label	label × javaStm → javaStm
while $(e)\ \alpha$	While	javaExpr × javaStm → javaStm
do α **while** (e)	Do	javaStm × javaExpr → javaStm
for$(e; e_1 \dots e_n)\ \alpha$	For	javaExpr × javaExpr* × javaStm → javaStm
switch $(e)\ \alpha$	Switch	javaExpr × javaStm → javaStm
case $l_1, \dots, l_n :$	switchLabel	literal* → javaStm
break $l;$	Break	label → javaStm
return;	Return	→ javaStm
return $e;$	ReturnExpr	javaExpr → javaStm
throw $e;$	Throw	javaExpr → javaStm
try$\{\alpha\}$ cts **finally**$\{\beta\}$	Try	javaStm × javaStm* × javaStm → javaStm
catch $c(x)\{\alpha\}$	Catch	classname × variable × javaStm → javaStm
static(c)	Static	classname → javaStm
endstatic(c)	EndStatic	classname → javaStm
target(m)	Target	mode → javaStm
targetExpr(x)	TargetExpr	variable → javaStm
$ct_1 \dots ct_n$	Catches	javaStm* → javaStm
finally$\{\alpha\}$	Finally	javaStm → javaStm
endfinally(m)	EndFinally	mode → javaStm

- In JLS, a block contains *BlockStatements* (JLS 14.2). We simplify this to (normal) Java statements. This means that a *local variable declaration* is a normal Java statement, and may appear anywhere (not only inside a block).
- An *expression statement* may contain any Java expression (JLS 14.8 introduces a *Statement-Expression*).
- An `if` always has an `else` part.
- The three loop constructs (`while`, `do`, and `for`) have no label. See chapter 2.5.
- The `for` statement has no initialization to allow iteration in the calculus. See chapter 2.5.
- The body of a `switch` statement is a normal Java statement, not a *SwitchBlock* as defined in JLS 14.10. This means that switch labels are normal Java statements.
- A `try` statement (JLS 14.19) always has a list of (possibly empty) catch clauses and a finally statement (that can be an empty block). The list of catch clauses and therefore a single catch clause are normal Java statements.
- The two new statements `static` and `endstatic` both have as argument a class name. `static`(c) is used to initialize the super classes of c, and to execute c's static initializer; `endstatic`(c) is used to catch exceptions during the static initialization of c.
- The new statements `target` and `targetExpr` catch jumps that result from a `return` or `break`.
- The new statements `catches`, `finally`, and `endfinally` are used by the calculus to handle the different possibilities when executing a `try` block.

2.3 Java Type Declarations

When a Java program runs its statements and expressions are executed and evaluated. They have a semantics. Method declarations, classes, packages etc. provide a context that determines the semantics of statements and expressions to some degree. They have no semantics of their own. Since we are concerned with the runtime behavior of Java programs and start with an annotated abstract syntax tree the context used in this work is a list of type declarations – a list of class or interface declarations. JLS (chapter 7) describes *packages* consisting of *compilation units* which in turn consist of type declarations. However, packages and compilation units are used solely for compilation purposes (where they govern visibility and access rules) and dynamic loading of classes. For a formal runtime semantics of Java they are not necessary: We assume that in the annotated syntax tree all class names are fully qualified, and that access rights have been checked. Dynamic loading of classes is not modeled because it is not part of the Java semantics at the source code level, but rather of the Java virtual machine. Table 2.3 contains the abstract syntax for type declarations.

Table 2.3: Abstract syntax for Java type declarations

TypeDecl	= ClassDecl | InterfaceDecl
ClassDecl	= modifier* × classname × classname* × classname* × MemberDecl*
InterfaceDecl	= modifier* × classname × classname* × MemberDecl*
MemberDecl	= StatInit | FieldDecl | MethodDecl | ConstrDecl
StatInit	= javaStm
FieldDecl	= modifier* × type × fieldspec
MethodDecl	= modifier* × type × methodname × ParamDecl* × javaStm
ConstrDecl	= modifier* × classname × ParamDecl* × javaStm

Some remarks:

- A *class declaration* consists of the modifiers, the class name, the *extends* classes (which should be either empty or a list with one class), the implemented interfaces, and the class body (a list of member declarations).

- An *interface declaration* has modifiers, the interface name (also of sort *classname* because the names are interchangeable), the *extends* interfaces, and the interface body. The body of a class and an interface may contain the same *member declarations*. This means we allow an interface to contain a constructor declaration (which makes no sense), and a static initializer (which makes sense because it is used to initialize the static fields of the interface).

- The members of a class or interface are either a static initializer (*StatInit*), a field declaration (*FieldDecl*), a method declaration (*MethodDecl*), or a constructor declaration (*ConstrDecl*). We do not allow an instance initializer because we assume that the compiler added the code of the instance initializer to the body of the constructor. Type declarations are also not allowed because inner classes are not supported.

- The static initializer can contain an arbitrary Java statement (JLS specifies a block).

- A field declaration declares only one field. We assume that the compiler divides the declaration of multiple fields (e.g. `int x = 3, y[] = { 1, 2 };`) into different declarations (in the example into `int x = 3;` and `int[] y = { 1, 2 };`). The example is a valid Java field declaration; because of the C syntax for arrays it is actually possible to declare fields of different types in one field declaration. A field is always referenced by a field specification (see the field access and field assignment expressions) to make it unique.

- A method or constructor declaration do not contain a `throws` clause because they are used only for compile time checks and not at run time. The formal parameters *ParamDecl* consist of a type and a local variable (a `final` modifier is not supported).

Table 2.4 lists the additional types used in expressions, statements, and type declarations. They are specified algebraically.

Table 2.4: Additional types used in Java expressions and statements

Type	**description**
literal	a term
type	a Java type
UnOp	`+`, `-`, `~`, `!`
IncDecOp	pre-`++`, pre-`--`, post-`++`, post-`--`
CondOp	`&&`, `||`
BinOp	`+`, `-`, `*`, `&`, `|`, `^`, `<<`, `>>`, `>>>`
ExBinOp	`/`, `%`
variable	a variable
fieldspec	a field specification (classname, fieldname, type)
AsgOp	`+=`, `-=`, `*=`, `/=`, `%=`, `&=`, `|=`, `^=`, `<<=`, `>>=`, `>>>=`
classname	a class name
fieldname	a field name
methodname	a method name
invMode	*nonVirtual*(c), *super*(c), *virtual*, *static*(c), *interface*
label	a label name
mode	*normal*, *throw*(val,ty), *break*(l), *return*, *return*(val, ty)
ParamDecl	type × variable
modifier	static, public, private etc.

2.4 Assumptions about the Compiler

The semantics presented in this work does not include type checks. This means that type incorrect programs have a meaning if they terminate. Or put in another way: Only non-terminating programs have no semantics. The semantics is defined relative to an annotated abstract syntax tree. In this respect no assumptions are made or needed. However, the annotated abstract syntax tree is usually derived from source code by parsing and annotation, even though this is not part of the formal semantics. The KIV system contains a routine to create the annotated abstract syntax tree. This derivation must fulfill some requirements so that the formal semantics is identical to the (informal) meaning of the source code. They are more or less the same requirements that are made in the Java Language Specification about the compiler. This reflects the distinction between the compile-time and run-time aspects of Java.

The following list contains these requirements:

Every class has a constructor. The compiler should add a default constructor if none is defined (JLS 8.8.7).

A Constructor begins with `this` or `super`. The compiler should add a `super` call if a constructor does not begin with a `this` or `super` call (JLS 8.8.5).

Instance fields are initialized in the constructor. Field initializations should be added as assignments to the constructors. More precisely, to every constructor beginning with a super call, after the super call, before other code of the constructor body. The same applies for instance initializers, their textual order is preserved (JLS 8.8.7).

Modifiers for interface fields. All interface fields are implicitly public, final, static. These modifiers should be made explicit (JLS 9.3).

No compile-time constants. We assume that all compile-time constants have been eliminated, i.e. no static final fields with constant initializations appear (JLS 12.4.2 and 13.4.8).

Static fields are initialized in a static initializer. Static fields with initializations should be transformed into static fields without initializations, and a static initializer with an assignment should be added (preserving the the textual order of static fields and initializers) (JLS 12.4.2). This should also be done for interfaces even though interfaces in Java may not contain static initializers.

Breaks have labels. Every break has a label. Breaks without label are transformed into breaks with label by introducing a new label around the old break target. See below.

No continues. Continue statements are transformed into break statements by adding new labels to the body of the iteration construct that is the target for the continue. See below.

Every local variable has an initializer. The compiler should add an initializer to a local variable declaration without an initializer. The initializer should be the default value for the type of the variable. This is necessary because the concept of definite assignments (the complete chapter 16 in JLS) is not specified.

Local variables in `switch` statements. The compiler should move local variable declarations that are declared in the top level switch block outside the switch statement. This is necessary because definite assignments are not supported. See below.

Primitive conversions. Java includes some automatic primitive conversions. For example, a byte value may be assigned to an integer variable without an explicit cast. (The reverse requires an explicit cast because casting an integer to a byte may change its value.) The automatic conversions are usually *primitive widening conversions*, and in one case a *primitive narrowing conversion*.

The compiler should make these conversion explicit by adding a cast for the following expressions: local variable assignment, static field assignment, instance field assignment, array assignment, conditional expression, arguments for method and constructor calls, local variable declaration, return statement

The compiler does not have to add explicit casts for the following expressions: unary, binary, and exception binary expression; indices in array access, assignment, creation; increment, decrement expression; compound assignment; and test in switch statement, labels.

See chapter 5.1 for details.

The first six requirements are made explicitely in JLS. All requirements are useful to reduce the size of the Java run time semantics. It is possible write source code that matches these requirements. If they are met then the run time behavior of the program is what one expects of the source code.

2.5 Discussion

The `continue` statement

A `continue` statement may occur only in a `while`, `do`, or `for` statement (JLS 14.15). The `continue` statement ends the current iteration of the loop and begins a new one. The problem is that a `continue` with a label requires a labeled loop. It is not adequate to view a labeled loop as a labeled statement containing a loop: The semantics of a labeled statement `label :` α containing only `break` statements can be expressed for arbitrary statements α, but the semantics of labeled statement containing a `continue` can be expressed only when the loop is also mentioned, i.e.

`label :` `while` (e) `do` α etc. This means the three loop statements should include a label, i.e. the arguments of a `while` should be label $\times$ javaExpr $\times$ javaStm $\rightarrow$ javaStm instead of javaExpr $\times$ javaStm $\rightarrow$ javaStm.

However, this has some disadvantages. One problem is that two loops (without `continue` statements) with the same test and body are still be syntactically different, even though they obviously have the same semantics. Another problem is that for a loop with `continue` the usual equation `while` (e) α = if (e) { α `while` (e) α} does not hold because a `continue` occurring in α has no longer a target.

It is easy to transform a loop containing a `continue` statement into a semantically equivalent loop by replacing the `continue` with a `break` statement with a new label and adding a labeled statement with this new label around the body of the loop:

While loop:

```
lab :  while e do { ...continue lab; ...}
```

is equivalent to

```
lab :  while e do new_lab :  { ...break new_lab; ...}
```

Do loop:

```
lab :  do { ...continue lab; ...} while (e);
```

is equivalent to

```
lab :  do new_lab :  { ...break new_lab; ...} while (e);
```

For loop:

```
lab :  for(init; test; updates) { ...continue lab; ...}
```

is equivalent to

```
lab :  for(init; test; updates) new_lab :  { ...break new_lab; ...}
```

The old label must be kept because it may be the target of a `break` statement.

Because of this simple transformation it was felt that the benefit of including the `continue` statement does not justify the additional complications (both semantics and calculus would become more complicated). Therefore the assumption is made that the compiler does the transformation.

A similar consideration lead to the omission of the initialization of a `for` loop: It is certainly possible to have a `for` statement with an initializer, however an adequate semantics (and calculus) would do the initialization and describe the iteration part with the help of a `for` statement without initialization (JLS does the same, see 14.13.1 and 14.13.2). So the assumption is that a compiler transforms

```
for(init; test; updates) { ...}
```

into

```
{ init; for(; test; updates) { ...} }
```

Static fields with an invoker

Java allows static fields to have an invoking expression that must have a result type that is a class containing the field. The invoking expression is evaluated, its value discarded, and the value of the field returned. This is usually considered a bad programming style because writing `x.i` suggests that `i` is an instance field, and in fact this construct is seldom used. However, a formal specification of this construct requires an unacceptable overhead for the following reason. Consider the following program.

```
class StatAccess1 {

    public final static int i = 5;
```

```
    static { System.out.println(4); }
}

public class StatAccess {

    static StatAccess1 m() { System.out.println(3); return null; }

    public static void main(String[] args) {
        System.out.println(m().i);
    }
}
```

This program compiles and prints 3 and 5. `i` is the static field that is used with the invoking expression `m()`. The result shows that the following happens: Method `m` is invoked and prints 3, then `i` is printed (equal to 5). The class `StatAccess1` is never initialized (4 is not printed). The reason is that `i` is a compile time constant. This means the expression `m().i` behaves as follows: Evaluate the first expression, discard its value, and return 5. However, it is illegal in Java to write `m().5`, and it is not possible to write `m(); i;` because the expression is nested in another expression. Modeling this behavior requires a new Java expression $e_1\ e_2$ where e_1 is evaluated for its side effect, and the value of e_2 is returned! It is not correct to evaluate `m()` and then access the field `i` because this would initialize the class `StatAcess1`. Therefore, static fields may not have an invoking expression in this work.

Definite Assignments

The concept of a *definite assignment* ensures that each local variable (and every blank `final` field) has been assigned a value before the variable is used (accessed). This is checked by the compiler using a "specific conservative flow analysis" (JLS 16).

Whether this property is fulfilled for a Java program is irrelevant for its run-time semantics. However, it is important for the definition of the free variables of a Java program that will be used in the calculus. The specification of free variables used in this work does not include the flow analysis mentioned above. The result is that some valid Java programs will be rejected, or more precisely: the calculus is incomplete for some valid Java programs. However, it is easy for a compiler to transform these programs.

The abstract syntax for a local variable declaration must contain an initialization ($ty\ x = e$; with arguments type $\times$ variable $\times$ javaExpr). This is not a problem because Java defines for every type a default value that can be added by the compiler. This ensures that a local variable is definitely assigned before it is used in all but two cases:

right hand side of the initialization. The scope of a local variable declaration includes its initialization itself. JLS 14.4.2 presents the following example:

```
int x = (x=2)*2;
```

This program will be rejected by the calculus. This is no real restriction, because it is obviously very bad programming style (and should be avoided), and a compiler can transform it into

```
int x = 0; x = (x=2)*2;
```

The publicly available source code of JDK 1.4 from SUN (more than 500.000 lines) contains no instance of this type of initialization.

local variables in a `switch` statement. The body of a `switch` statement is a block and may contain local variable declarations. It is possible to use this variable in different `cases`, which means that the declaration may be bypassed (JLS 4.5.3 (item 7)). However, the variable must be *definitely assigned* in each `case`. So the following is a valid Java statement:

```
switch (i) {
   case 1:  int j = 2; ...
   case 2:  j = 3; ...
}
```

Again, this statement is rejected by the calculus, because j is considered as a free variable. This style of programming is used several times in the JDK source code. However, it is possible to transform the code by extracting the local variable declaration:

```
{ int j = 2;
 switch (i) {
   case 1:  j = 2; ...
   case 2:  j = 3; ...
} }
```

This transformation must be done on the annotated syntax tree, not on the source code because the code preceding the local variable declaration may contain a field access with the same name.

This finishes the description of the abstract syntax for Java. It contains 24 expressions (plus expression lists), 23 statements (plus statement lists), 4 member declarations (field declaration etc.), 2 type declarations (classes and interfaces), and lists of them. The following language restrictions apply: Only Java Card primitive types are allowed (`boolean`, `byte`, `short`, `int`), and static fields may not be used with an invoking expression. Next, the semantics of Java is described.

Chapter 3

Semantics

This chapter describes a semantics for Java for the expressions and statements from the previous chapter. It is a big step (or natural) operational semantics that is defined with (or embedded in) algebraic specifications.

The chapter is organized as follows: First the environment is described in which expressions and statements are evaluated. This includes a store (or heap), Java values, references, a variable mapping etc. Then the semantics rules for expressions and statements are presented. The fourth section introduces the proof technique for theorems concerning the semantics, and presents a useful theorem, namely that the semantics is deterministic.

3.1 The Evaluation Environment

All data types that form the environment are specified with first-order structured algebraic specifications as they are available in KIV [RSSB98] [Wir90] [CoF04] [Rei91] [Rei92a] [Rei93]. Some of the specifications come from KIV's library (numbers, lists, sets, etc.), others are used both by the semantics and the implemented calculus, and yet others are specified only for the formal semantics. Some are not used in the semantics even though they are subspecifications. Specifications in the library are often more structured than necessary to keep the number of (user defined) theorems in one specification manageable. For example, the specification for unbounded integers is structured into nine different specifications with 20 operations, 39 axioms, and more than 1200 theorems and simplification rules. This shows that a full and precise description of these specifications – or a listing of all axioms – here would be much too large and essentially worthless. Therefore the most important specifications are described informally, and the interested reader is referred to the formal specification for further details.

The evaluation environment is very simple: It consists of a variable mapping, a Java store (basically the heap for objects), and a Java value for the result of expressions. These components will be described now. (KIV's syntax will be described on the fly. Here, we mention only that KIV allows overloading for argument and result sorts for functions and predicates which is used heavily, and can be sometimes confusing.)

3.1.1 Javavalue

The sort *javavalue* is a union type that encapsulates the different (supported) Java values: *int*, *byte*, *short*, *bool*, *reference*. An integer i is converted with the constructor *intval(i)* into a *javavalue*, *.intval* (used as postfix, *intval(i).intval* selects the integer from an *intval*, and the predicate *is_integervalue* is true if a *javavalue* is an *intval*. Similar operations exist for the other sorts. *int*, *short*, *byte* correspond to their Java primitive types, and *bool* corresponds to `boolean`. The sort *reference* is used for references to objects and arrays. However, *javavalue* has some more constructors so that all information that is needed for the evaluation can be stored in a *javavalue*:

- . ++ . constructs a *javavalue* from two Java values. This allows to encode lists of values.
- *noval* is a constant – it can be viewed as an error or bottom element.
- *normal* is used for the evaluation mode (described below).
- *typeval* turns a Java type into a value. This is used to store the type of objects in the store.
- *initval* converts an *initstate* into a value. The initstate is used to record class initialization, and can be either *undone* , *error* , or *done* .
- *break*, *return*, and *throw* describe the reason for an abrupt transfer of control, i.e. when an expression or statement does not terminate normally. The constructor *break* has a label as its argument, *return* a value (the returned value) and its type, and *throw* a reference (to the thrown object) and its type.

The additional Java values allow to define a very simple environment and causes no confusion because the non-standard values are used only in special situations.

In addition to the values themselves the Java operations on these value are specified, i.e. addition, multiplication, conversion between bytes, shorts, and integers, and the bitwise operations on integers. (Since the focus is on Java Card applications there exist no specifications for operations on floats or doubles.) Especially the bitwise operations &, |, >> etc. and the casts are a nice exercise in algebraic specification. Since they are often used in Java Card programs, their correct specification (and proof support with already proved theorems) is very important for applications, but more or less irrelevant for the Java semantics and calculus. Therefore, they are omitted in this work. Chapter 7.3 contains an example that uses them very heavily.

3.1.2 Abstract Store

The most important part of the environment is a *store*. A Java store is an actualization of a generic store. In a generic store (generic) values can be stored under (generic) keys. If st is a store, and a a key, then $st[a]$ looks up the value stored for this key. $a \in st$ is true if a value is stored for the key (i.e. the key exists in the store). $\emptyset$ is the empty store (containing no keys and values), and a value d can be added for key a with the put operation $st[a, d]$. # returns the number of keys in a store, and – deletes a key. This specification is shown in full.

generic specification

parameter elemdata

using nat

target

⟨*structured specifications: the two specifications elemdata and nat are used.*⟩

sorts store; elemdata;

constants $\emptyset$: store;

functions

. × . : elem × data → elemdata;

. [., .] : store × elem × data → store;

. [.] : store × elem → data;

. : store → nat;

. – . : store × elem → store prio 9 left;

⟨*the single dots indicate argument positions; infix, prefix and mixfix operations are used.*⟩

predicates

. ∈ . : elem × store; . ∈ . : elemdata × store; . ⊆ . : store × store;

variables st, st0, st1, st2 : store; elemdatavar : elemdata;

⟨*the signature part. The axioms follow.*⟩

induction store **generated by** ∅, [ ] :: (store × elem × data → store);
elemdata **freely generated by** × ;

axioms

Extension : st1 = st2 ↔ ∀ a. (a ∈ st1 ↔ a ∈ st2) ∧ st1[a] = st2[a];

In-empty : ¬ a ∈ ∅;

In-insert : a ∈ st[b, d] ↔ a = b ∨ a ∈ st;

At-same : st[a, d][a] = d;

At-other : a ≠ b → st[b, d][a] = st[a];

In-store : a × d ∈ st ↔ a ∈ st ∧ st[a] = d;

Subset : st1 ⊆ st2 ↔ ∀ a. a ∈ st1 → a ∈ st2 ∧ st1[a] = st2[a];

Size-empty : #(∅) = 0;

Size-insert : ¬ a ∈ st → # st[a, d] = #(st) + 1;

Del-in : a ∈ st – b ↔ a ≠ b ∧ a ∈ st;

Del-at : a ≠ b → (st – b)[a] = st[a];

end generic specification

This is original KIV syntax. The *Extension* axiom states that two stores are equal if they contain the same key with the same values. Since stores are generated by the empty store ∅ and the put operation, stores always contain only a finite number of keys.

3.1.3 The Java Store

The Java store (of sort *store*) uses *javavalue* as the stored values, not objects as might be expected. Objects do not exist explicitly in the store. The keys in the Java store are more basic: A key (of sort *refkey* is a pair consisting of a reference and either a *field specification* or an *array index*. This means that every field of every object, and every index of every array has its own key where its value is stored.

refkey = . – . (. .ref : reference; . .key : storekey)

⟨*This notation is used to specify free generated data types. The sort* refkey *is generated by the single constructor –.* .ref *selects the first component, and* .key *the second component of a* refkey.⟩

storekey = . ' (. .fs : fieldspec) | . ' (. .index : int)

⟨*The sort* storekey *has two constructors. Both are named ' and used as a postfix operation.* .fs *selects the field specification of the first constructor, and* .index *the integer from the second.*⟩

An array index is just an integer that is encapsulated in a *storekey* (by a postfix quote). If r is a reference to an array and st a Java store, then $st[r-3']$ looks up the value stored at index 3 in the array, and $st[r-_length]$ looks up the length field of the array. Actually, *_length* is an abbreviation (a constant), for a *storekey* containing a field specification, *_length = mkfs(*Array*,int,length)'*. A field specification contains a field name, the static type of the field, and the class where the field is declared. This is necessary to make fields unique (or to handle overloading). In the rest of this work we will omit the quote that converts a *fieldspec* or an integer into a storekey. This means we will write r – 3 instead of r – 3 ', and r – fs instead of r – fs ' (It is possible to define two functions – of sorts reference × fieldspec → refkey and reference × int → refkey because of overloading . . .)

An object in the store is the collection of all keys with the same reference. It is now easy to define more complex operations on the Java store that are useful for the semantics. For example, when a new object is created, its fields – initialized with their default value – can be added to the store with a function *addobj(r, class, fis, st)*. *r* is the new reference for the object, *class* its class, and *fis* a list of store keys (field specifications) plus their values. The definition of *addobj* is simply

addobj-base :

addobj(r,class1,[],st) = st[r – _type, typeval(classtype(class1))];

addobj-rec :

addobj(r,class1,(sk × val) + fis,st) = addobj(r,class1,fis,st)[r – sk, val];

[] is the empty list, a + x a list with first element a and rest list x (+ is heavily overloaded). The first axiom adds another field for this reference to the store, the special field _type (special because it cannot occur in a normal Java program) that records the objects class type. Since the values in the store are Java values (of sort *javavalue*) it must be possible to convert a type into a value. This is the reason why the sort *javavalue* contains additional constructors. If the reference *r* is new we have the intended situation that all keys with reference *r* in the store together form the object. To create arrays three functions exist: to create a one-dimensional array with default values, a one-dimensional array with given values (used for array initializers), and a multi-dimensional array. Arrays have indices in the store, a length field, and a type field.

This part of the store is the heap. However, some more information is needed: the initialization state of classes, their static fields, and the evaluation mode. This information is stored under keys with a special reference *jvmref* (a reference for the Java virtual machine, so to say; it is also used as the `null` reference):

- The evaluation mode is stored under the key mode = jvmref – mkfs(Object, void, mode) '. st[mode] looks up the mode in the store.

  The mode is either *normal* (a special javavalue), or it contains the reason for an abrupt transfer of control: return, break, or throw.

- The initialization state of a class *c* is stored in the special key jvmref – mkfs(c, void, initstate). To test the initialization state the derived predicates *initdone(c, st)*, *initundone(c, st)*, and *initerror(c, st)* will be used.

- A static field *fs* of a class is simply stored under jvmref – fs (fs is a field specification that already contains the class where the field is declared).

The static initialization of classes uses the function *addclass* that sets the last two fields:

addclass-base :

addclass(class,[],st) = st[jvmref – mkfs(class, void, initstate), initval(done)];

addclass-yes :

addclass(class,(mkfs(class,ty,fieldvar) × val) + fis,st)
= addclass(class,fis,st)[jvmref – mkfs(class,ty,fieldvar), val];

addclass-no :

class ≠ class0
→ addclass(class,(mkfs(class0,ty,fieldvar) × val) + fis,st)
= addclass(class,fis,st);

It should be noted that the semantics makes no assumption that these special keys really exist in the store. This is no problem. New references for objects and arrays will always be different from jvmref. Furthermore, a normal (i.e. type correct) Java program cannot modify these fields, because they have an illegal **void** type. However, a type incorrect program could modify these

fields, and this makes some proofs more difficult, because some properties do not hold. The issue of type correctness will be discussed in chapter 5.

This finishes the description of the Java store.

3.1.4 Variable Mapping

The local variables of a Java program and their values are not part of the store. They are contained in a *variable mapping* . A variable mapping is also an instance of an abstract store, but of a store that is not (finitely) generated. This means it can contain entries for an infinite number of keys (variables). The reason is that in the calculus local Java variables will be identified with logical variables, and for logical variables only one global mapping for all (countably infinitely many) variables exists. Therefore the semantics is defined with respect to one global variable mapping. There is no need to have several mappings for different method invocations, as e.g. [vO01]. Method calls are handled by modifying the values of the formal parameters when the method is invoked, and by restoring their old value when the method is finished. This is done by two functions *bind* and *restore* that operate on variable mappings. The variable mapping not only contains Java values (of sort *javavalue*), but all values that can possibly occur, most notably Java stores of sort *store*. Technically, another algebraic specification *statevalue* is used that defines another union type containing *javavalue* and *store* among others.

3.1.5 Predefined Classes

Some predefined classes are used in the semantics: `java.lang.Object` appears only implicitly in the semantic (we will omit the package prefix `java.lang` in the sequel). The following Exceptions and Errors are used in the semantic rules:

Class	**Used in section**
ClassCastException	3.2.5
ArithmeticException	3.2.10
NullPointerException	3.2.13, 3.2.14, 3.2.17, 3.2.18, 3.2.25, 3.3.14
ArrayIndexOutOfBoundsException	3.2.14, 3.2.18
ArrayStoreException	3.2.18
NegativeArraySizeException	3.2.22
ClassDefNotFoundError	3.3.21
ExceptionInInitializerError	3.3.22
Exception	3.3.22

This also implies the usage of the classes `IndexOutOfBoundsException`, `RuntimeException`, `Error` and `Throwable`. In JLS(2) ArrayIndexOutOfBoundsException (a subclass of IndexOutOfBoundsException) is thrown for illegal array indices; this has been changed from JLS(1) where an IndexOutOfBoundsException is thrown.

Other predefined Java classes (e.g. `Class` or `String`) do not appear. They are unnecessary for the semantics, and not supported by Java Card anyway. The predefined classes have no fields, no methods, and only a default constructor. (Java Card does not support a string as an error message. The `Object` class has only an *equals* method – it tests if two references are equal – which is unnessary and omitted here.) The predefined classes are treated as always initialized, and a correct Java program (in this work) may not contain class declarations with these names.

3.1.6 The Semantics Relations

As mentioned in chapter 1, most Java semantics are operational. Börger [BS99] [SSB01] uses abstract state machines (ASMs), Alves-Foss [AFL99] a denotational semantics [Mos91]; see [HM01] for others. Operational semantics (based on Plotkin [Plo81] [Plo83] and Kahn [Kah87]) can be divided into two branches: natural or big-step semantics, and small-step or structural operational semantics (SOS) (see e.g. [Ast91] [NN98] [AC96] [Gle03]). The distinction is not really clear;

sometimes SOS is divided into big-step and small-step semantics; and natural semantics is seen as SOS "revisited", and several mixtures ("mixed-step" semantics) and extensions exist (e.g. modular SOS [Mos02]). Operational semantics uses an inference system of rules to describe the evaluation of a program. The difference between big-step and small-step semantics can best be described with an example. Consider sequential composition: $\alpha;\beta$ and a program state s (this not Java, but an abstract imperative programming language). Then the semantics of $\alpha;\beta$ with initial state s is: evaluate α to obtain an intermediate state s_a, then evaluate β to obtain the final state s_b. In big-step semantics α and β are each evaluated in one big step to obtain the final state (below on the left); in small-step semantics α is evaluated in one small step to a program α' and intermediate state s', and β is not (yet) evaluated (on the right):

$$\frac{\langle s,\alpha\rangle \longrightarrow s_a \qquad \langle s_a,\beta\rangle \longrightarrow s_b}{\langle s,\alpha;\beta\rangle \longrightarrow s_b} \qquad \frac{\langle s,\alpha\rangle \longrightarrow \langle s',\alpha'\rangle}{\langle s,\alpha;\beta\rangle \longrightarrow \langle s',\alpha';\beta\rangle}$$

In big-step semantics, evaluation of a state and a program yields a final state, in small-step semantics the evaluation yields a state and another program. It is possible to prove the equivalence of a big-step and a small-step semantics (e.g. [Nip03]). Small-step semantics allows to reason about non-terminating program runs or concurrent languages (for Java with threads, e.g. [CKRW99] [Abr04]), while big-step semantics is a little bit more abstract and (perhaps) more natural for human presentation. The Java language specification uses a natural semantics style presentation for statements and expressions.

The semantics of a Java program in this work is defined by a natural (big-step) operational semantics. It differs from all other semantics in that it does not tries to be as short as possible, but rather to be as readable as possible. (For example, no superscripts are used, and subscripts only for variable names.) For every Java expression and statement one or more evaluation rules describe their effect on the environment (a pair of a variable mapping and a store). Both may be modified by the evaluation. The list of Java type declarations (classes or interfaces) form the immutable context.

Technically, a Java statement α defines a relation:

$$(v \times st)[\![\alpha]\!]_{tds}(v' \times st')$$

is true iff evaluating α with an initial variable mapping v and store st (in the context of type declarations tds) terminates, and yields a new environment $v' \times st'$. The notation indicates (or stresses) the relational character of the statement semantics: There may exist no resulting environment (for example, if the program does not terminate) or there may be more than one possible resulting environments (if the program is not deterministic). A notation that stresses more the character of a reduction system would be

$$\langle v \times st, \alpha\rangle \;{}_{tds} \Longrightarrow (v' \times st')$$

or

$$tds \vdash (v \times st) \overset{\alpha}{\Longrightarrow} (v' \times st')$$

However, this is just a matter of style, not a difference in meaning. The actual notation used in KIV is

$$\mathrm{sem}(v \times st, \alpha, tds, v_0 \times st_0)$$

where *sem* is a predicate. This notation is more readable in proofs with large goals. However, in the text we will use the more mathematical notation shown above (and usually omit tds because they never change).

Since we have to define the semantics of expressions, lists of expressions, statements, and lists of statements we have four relations that will be defined simultaneously:

expressions: $(v \times st)[\![e]\!]_{tds}(v_0 \times st_0 \times val)$ (e is an expression, val the computed value)

expression lists: $(v \times st)[\![e_1 \ldots e_n]\!]_{tds}(v_0 \times st_0 \times (val_1 \ldots val_n))$

statements: $(v \times st)[\![\alpha]\!]_{tds}(v_0 \times st_0)$

statement lists: $(v \times st)[\![\alpha_1 \ldots \alpha_n]\!]_{tds}(v_0 \times st_0)$

An expression e computes a value val (and possibly modifies the environment as a side effect). A list of expressions computes a list of values (it can be proved that it will be a list of the same length), and a list of statements just modifies the environment. In KIV the four predicates are all called *sem* (thanks to overloading):

expressions: sem(v × st, e, tds, v0 × st0 × val)

expression lists: sem(v × st, es, tds, v0 × st0 × vals)

statements: sem(v × st, α, v0 × st0)

statement lists: sem(v × st, stms, v0 × st0)

Here, es is a list of expressions, vals a list of values, and stms a list of statements. Whenever the dot notation $(e_1 \ldots e_n)$ is used below, in KIV a list is used.

These relations are defined in two steps: First, by reduction rules stating that the relation is true for some inputs if it is true for some other inputs for subexpressions or -statements. These rules define for which inputs the relations are true. Second, by defining the relations as the smallest relations that are closed under the reduction rules. This defines for which inputs the relations are *not* true. We show two examples, first the assignment to a local variable:

$$\frac{(v \times st)[\![e]\!](v_0 \times st_0 \times val_0) \quad st_0[mode] = normal}{(v \times st)[\![x = e]\!](v_0[x, val_0] \times st_0 \times val_0)}$$

The meaning of an assignment $x = e$ is defined by first evaluating the right hand side e: $(v \times st)[\![e]\!](v_0 \times st_0 \times val_0)$. If the evaluation of e terminates with value val_0 and new environment $v_0 \times st_0$, and the evaluation mode is normal ($st_0[mode] = normal$, this means evaluation of e did not raise an exception) then the assignment returns the same value val_0 as e, the same store st_0, and a modified variable mapping $v_0[x, val_0]$ where the value of x has been changed to val_0. The second example is the `if` statement:

$$\frac{(v \times st)[\![e]\!](v_0 \times st_0 \times val_0) \quad st_0[mode] = normal \wedge val_0 = \text{true} \quad (v_0 \times st_0)[\![\alpha_1]\!](v_1 \times st_1)}{(v \times st)[\![\textbf{if } (e) \ \alpha_1 \ \textbf{else } \alpha_2]\!](v_1 \times st_1)}$$

For the `if` statement first the test expression is evaluated $(v \times st)[\![e]\!](v_0 \times st_0 \times val_0)$, and if this terminates normally ($st_0[mode] = normal$) with a result that is true ($val_0 = \text{true}$) the `then` part of the statement is evaluated in the new environment $(v_0 \times st_0)[\![\alpha_1]\!](v_1 \times st_1)$ and the final environment $(v_1 \times st_1)$ is the result of the complete `if` statement. (Two other rules deal with an exception while evaluating e and the case that e evaluates to **false**.)

All of the rules have the form

$$\frac{sem_1 \ldots sem_n \quad \varphi_1 \ldots \varphi_m}{sem}$$

where *sem* is one of the four relations to be defined (for some inputs): If the relations hold for some inputs $(sem_1 \ldots sem_n)$ and some other first-order conditions $\varphi_1 \ldots \varphi_m$ that do not contain the relation the *sem* hold, then *sem* is true. Logically,

$$sem_1 \wedge \ldots \wedge sem_n \wedge \varphi_1 \wedge \ldots \wedge \varphi_m \rightarrow sem$$

Since there are only positive occurrences of the relation in the precondition of the implication (Horn style) the smallest relation closed under the rules is well defined – and not empty, because some rules have no preconditions at all.

After these preliminaries we continue with the rules for expressions and expression lists.

3.2 Semantics of Expressions

3.2.1 Expression lists

The semantics of a list of expressions is defined recursively by evaluating the expressions from left to right:

$$\frac{}{(v \times st)[\![]\!](v \times st \times [])} \qquad \frac{(v \times st)[\![e]\!](v_0 \times st_0 \times val_0) \quad (v_0 \times st_0)[\![es]\!](v_1 \times st_1 \times vals_1)}{(v \times st)[\![e + es]\!](v_1 \times st_1 \times (val_0 + vals_1))}$$

Evaluating the empty list [] does not modify the environment and returns an empty list of values. Evaluating a list $e + es$ (e an expression, es a list of expressions, $+$ adds an element to the head of the list) first evaluates e, then the rest of the list with the new environment $(v_0 \times st_0)$. The result is the final environment and the computed list of values val_0 (for e) plus $vals_1$ (for es). Evaluation of a list of expressions is used for argument lists, and adheres to Java's rules (JLS 15.7.4, argument lists are evaluated left-to-right). If the evaluation of one expression completes abruptly, then the remaining expressions are not evaluated. This is captured by the rule for jumps.

3.2.2 Jumps

The Java language specification introduces the notions of a *normal evaluation* of expressions (JLS 15.6) and *normal execution* of statements (JLS 14.1), and distinguishes between *abrupt* and *normal* completion of (the evaluation of) an expression or statement. Abrupt completion can be due to an exception or error that was thrown, or due to a `return` or `break` statement. We will simply speak of a *jump*. In this case the *mode* in the store will be set to the reason for the jump (see section 3.1.3). Specifically, the mode will not be *normal*. An expression is evaluated only if no jump occurs. This is captured by the following rule:

$$\frac{st[mode] \neq normal}{(v \times st)[\![e]\!](v \times st \times noval)}$$

If the evaluation mode ($st[mode]$) is not normal the expression is not evaluated: The environment remains unchanged, and the dummy value *noval* is returned – which is never used. For a list of expressions this means that a list containing only *noval* is returned, which is also never used. This single rule is sufficient, because no expression can catch a jump. (In contrast to some statements.)

3.2.3 Literal

A literal l in Java is either a concrete value of a primitive type or `null`. In our formalization a literal contains an arbitrary term of the underlying algebraic specification that can contain (logical) variables. Evaluation of the literal simply evaluates the term with the current variable mapping (denoted by $eval(v, t)$):

$$\frac{st[mode] = normal}{(v \times st)[\![literal(t)]\!](v \times st \times eval(v, t))}$$

The literal is not evaluated if the mode is not normal. The function *eval* here is the normal algebraic evaluation of terms. *eval* is heavily overloaded and will be used with different arguments in the sequel.

3.2.4 Unary operation

Unary operations (JLS 15.15) are unary plus (`+`), unary minus (`-`), bitwise complement (`~`) for integers, and logical complement `!` for booleans. The prefix increment and decrement operators (syntactically also unary operations) are *incdec* operations and handled in section 3.2.19 because they perform an assignment as a side condition. Let $\oplus$ be one of the four listed operations. It is evaluated only if the evaluation of the argument expression e completes normally. This implies

that the mode before the evaluation of e was also normal (the jump rule states that the mode remains unchanged if it was not normal). Therefore we have two rules, depending on whether the mode after evaluation of e is normal or not.

$$\frac{(v \times st)[\![e]\!](v_0 \times st_0 \times val_0) \quad st_0[mode] \neq normal}{(v \times st)[\![\oplus\, e]\!](v_0 \times st_0 \times val_0)}$$

The first rule explicitly states that the environment and value are unchanged if the mode is not normal. The second rule is for the normal evaluation:

$$\frac{(v \times st)[\![e]\!](v_0 \times st_0 \times val_0) \quad st_0[mode] = normal}{(v \times st)[\![\oplus\, e]\!](v_0 \times st_0 \times eval(\oplus val_0))}$$

Both rules could be merged in one if the *eval* function that evaluates the unary operation has as an additional input the mode and does nothing if it is not normal. However, this approach was deliberately rejected because it makes the rules much more difficult to read. The first rule cannot be omitted (if the second is unchanged) because this would mean the unary expression has no semantics at all in this case (because of the smallest set definition).

The function *eval* computes the result depending on the operator. Bytes and shorts are converted to integers (numeric promotion, JLS 5.6).

3.2.5 Cast

Cast expressions are described in JLS 15.16 and 5.5. We distinguish between a primitive and a reference cast, i.e. whether the cast type is primitive or not. A primitive cast changes the value of its argument (for example, from integer to byte by discarding the upper 24 bits), and never throws an exception. A reference cast, on the other hand, does not modify its argument, but checks if its runtime type is compatible to the cast type, and throws a `ClassCastException` if not.
Primitive cast:

$$\frac{(v \times st)[\![e]\!](v_0 \times st_0 \times val_0) \quad st_0[mode] = normal}{(v \times st)[\![(ty)e]\!](v_0 \times st_0 \times eval(ty, val_0))}$$

$$\frac{(v \times st)[\![e]\!](v_0 \times st_0 \times val_0) \quad st_0[mode] \neq normal}{(v \times st)[\![(ty)e]\!](v_0 \times st_0 \times val_0)}$$

The function *eval* performs the conversion. Bytes, shorts, and integers can be converted into each other. (Other primitive types like **long**, **float**, **double**, or **char** are not supported.) A **byte** can be converted into a **short** with the (algebraic) function *b2s*. The other conversion functions are *b2i*, *s2i*, *i2s*, *i2b*, and *s2b*.
Reference type cast:

$$\frac{(v \times st)[\![e]\!](v_0 \times st_0 \times val_0) \quad st_0[mode] \neq normal \vee val_0 = null \vee asgcomp(val_0, ty, st_0, tds)}{(v \times st)[\![(ty)e]\!](v_0 \times st_0 \times val_0)}$$

$$\frac{(v \times st)[\![e]\!](v_0 \times st_0 \times val_0) \quad st_0[mode] = normal \quad val_0 \neq null \wedge \neg\, asgcomp(val_0, ty, st_0, tds) \quad (v_0 \times st_0)[\![\texttt{throw new ClassCastException();}]\!](v_1 \times st_1)}{(v \times st)[\![(ty)e]\!](v_1 \times st_1 \times noval)}$$

The predicate *asgcomp* is true if the runtime type of the value val_0 is less or equal to the type ty as specified in JLS 5.5 for reference types. Its name is *asgcomp* because it is also used for the array assignment (section 3.2.18).

A value *val* is *assignment compatible* to a type *ty* iff

1. the value is not a reference value, or

2. the value is the `null` reference, or

3. the type of the reference in the store, st[val.refval – _type].type, is a subtype of ty with respect to type declarations tds, st[val.refval – _type].type $\leq_{tds}$ ty

The first case and second case are specified only to have a complete definition. The rules for the reference cast contain the `null` case explicitly for clarity. Formally,

$$\begin{array}{ll} & \text{asgcomp(val, ty, st, tds)} \\ \leftrightarrow & \quad \neg\ \text{is_referencevalue(val)} \\ & \quad \vee\ \text{val.refval} = \text{null} \\ & \quad \vee\ \text{st[val.refval} - \text{_type].type} \leq_{tds} \text{ty} \end{array}$$

A type ty is subtype of ty_0, ty $\leq_{tds}$ ty_0, iff

1. If ty is primitive, both types must be equal.

   $\neg$ is_classtype(ty) $\wedge$ $\neg$ is_arraytype(ty) $\rightarrow$ (ty $\leq_{tds}$ ty_0 $\leftrightarrow$ ty = ty_0)

2. If ty is an array type, then ty_0 is either the class type `java.lang.Object` and the predicate is true, or ty_0 must also be an array type, and $\leq_{tds}$ ty_0 must hold for the immediate element types (if ty = `int[][]`, the immediate element type is `int[]`).

$$\begin{array}{ll} & \text{arraytype(ty)} \leq_{tds} \text{ty}_0 \\ \leftrightarrow & \quad \text{ty}_0 = \text{classtype(java.lang.Object)} \\ & \quad \vee\ \text{is_arraytype}(\text{ty}_0) \wedge \text{ty} \leq_{tds} \text{ty}_0\text{.type} \end{array}$$

3. If ty is a class type then ty_0 must also be a class type. $\leq_{tds}$ is true iff the class c_0 of ty_0 is member of all super classes of the class c of ty (including c itself). All super classes is the transitive closure of the super classes (via extends), implemented interfaces (via implements), and extended interfaces (via extends).

$$\begin{array}{ll} & \text{classtype(class), tds) ty} \\ \leftrightarrow & \text{is_classtype(ty)} \wedge \text{ty.class} \in \text{allsupers(class, tds)} \end{array}$$

The formal definition of allsupers is omitted. It is specified in such a manner that it is well defined (i.e. consistent) for cyclical or otherwise malformed class hierarchies.

3.2.6 Instanceof

e *instanceof* ty checks if the runtime type of the expression e is less or equal to ty (JLS 15.20.2). ty must be a reference type, e must evaluate to a reference. The result is true iff e is not null and e can be casted to ty.

$$\frac{(v \times st)[\![e]\!](v_0 \times st_0 \times val_0) \quad st_0[mode] = normal \quad val_0 \neq null \wedge asgcomp(val_0, ty, st_0, tds)}{(v \times st)[\![e \texttt{ instanceof } ty]\!](v_0 \times st_0 \times \text{true})}$$

$$\frac{(v \times st)[\![e]\!](v_0 \times st_0 \times val_0) \quad st_0[mode] = normal \quad val_0 = null \vee \neg\ asgcomp(val_0, ty, st_0, tds)}{(v \times st)[\![e \texttt{ instanceof } ty]\!](v_0 \times st_0 \times \text{false})}$$

$$\frac{(v \times st)[\![e]\!](v_0 \times st_0 \times val_0) \quad st_0[mode] \neq normal}{(v \times st)[\![e \texttt{ instanceof } ty]\!](v_0 \times st_0 \times val_0)}$$

3.2.7 Conditional Operator

In $e_0?e_1 : e_2$, first e_0 is evaluated, then either e_1 or e_2 (JLS 15.25).

$$\frac{(v \times st)[\![e_0]\!](v_0 \times st_0 \times val_0) \quad st_0[mode] \neq normal}{(v \times st)[\![e_0?e_1 : e_2]\!](v_0 \times st_0 \times val_0)}$$

$$\frac{(v \times st)[\![e_0]\!](v_0 \times st_0 \times val_0) \quad st_0[mode] = normal \quad val_0.boolval = \text{true} \quad (v_0 \times st_0)[\![e_1]\!](v_1 \times st_1 \times val_1)}{(v \times st)[\![e_0?e_1 : e_2]\!](v_1 \times st_1 \times val_1)}$$

$$\frac{(v \times st)[\![e_0]\!](v_0 \times st_0 \times val_0) \quad st_0[mode] = normal \quad val_0.boolval = \text{false} \quad (v_0 \times st_0)[\![e_2]\!](v_2 \times st_2 \times val_2)}{(v \times st)[\![e_0?e_1 : e_2]\!](v_2 \times st_2 \times val_2)}$$

The result of the evaluation of an expression is always of sort *javavalue* (see chapter 3.1.1). *.boolval* selects the boolean value from a *javavalue*.

3.2.8 Conditional Binary Operator

These are `&&` (conditional-And operator, JLS 15.23) and `||` (conditional-Or operator, JLS 15.24). In contrast to the simple binary operations the right hand side is evaluated only if it is necessary (i.e. if the left hand side is true for `&&` and false for `||`).

$$\frac{(v \times st)[\![e_1]\!](v_0 \times st_0 \times val_0) \quad st_0[mode] \neq normal \vee \oplus = \texttt{\&\&} \wedge val_0.boolval = \text{false} \vee \oplus = \texttt{||} \wedge val_0.boolval = \text{true}}{(v \times st)[\![e_1 \oplus e_2]\!](v_0 \times st_0 \times val_0)}$$

$$\frac{(v \times st)[\![e_1]\!](v_0 \times st_0 \times val_0) \quad st_0[mode] = normal \wedge (\oplus = \texttt{\&\&} \wedge val_0.boolval = \text{true} \vee \oplus = \texttt{||} \wedge val_0.boolval = \text{false}) \quad (v_0 \times st_0)[\![e_2]\!](v_1 \times st_1 \times val_1)}{(v \times st)[\![e_1 \oplus e_2]\!](v_1 \times st_1 \times val_1)}$$

3.2.9 Simple Binary Operator

The *simple* binary operators are those operators that have no special evaluation order, and that cannot raise an exception. These are `==`, `!=`, `*`, `+`, `-`, `<<`, `>>`, `>>>`, `>`, `<`, `<=`, `>=`, `&`, `^`, `|`. The result of these operations is either a boolean or an integer value. Their arguments are either booleans, or numerical values that are converted to integers (or, for `==`, `!=`, arbitrary arguments of the same type).

$$\frac{(v \times st)[\![e_1]\!](v_0 \times st_0 \times val_0) \quad (v_0 \times st_0)[\![e_2]\!](v_1 \times st_1 \times val_1) \quad st_1[mode] = normal}{(v \times st)[\![e_1 \oplus e_2]\!](v_1 \times st_1 \times eval(val_0 \oplus val_1))}$$

$$\frac{(v \times st)[\![e_1]\!](v_0 \times st_0 \times val_0) \quad (v_0 \times st_0)[\![e_2]\!](v_1 \times st_1 \times val_1) \quad st_1[mode] \neq normal}{(v \times st)[\![e_1 \oplus e_2]\!](v_1 \times st_1 \times val_1)}$$

The function *eval* computes the result depending on the operator. Bytes and shorts are converted to integers (numeric promotion, JLS 5.6). The formal specification has one axiom for every operator.

3.2.10 Exception Binary Operator

Division / (JLS 15.17.2) and remainder % (JLS 15.17.3) raise a `java.lang.ArithmeticException` if the divisor is zero.

$$\frac{(v \times st)[\![e_1]\!](v_0 \times st_0 \times val_0) \quad (v_0 \times st_0)[\![e_2]\!](v_1 \times st_1 \times val_1) \quad val2int(val_1) \neq 0 \wedge st_1[mode] = normal}{(v \times st)[\![e_1 \oplus e_2]\!](v_1 \times st_1 \times eval(val_0 \oplus val_1))}$$

$$\frac{(v \times st)[\![e_1]\!](v_0 \times st_0 \times val_0) \quad (v_0 \times st_0)[\![e_2]\!](v_1 \times st_1 \times val_1) \quad val2int(val_1) = 0 \vee st_1[mode] \neq normal \quad (v_1 \times st_1)[\![\texttt{throw new ArithmeticException();}]\!](v_2 \times st_2)}{(v \times st)[\![e_1 \oplus e_2]\!](v_2 \times st_2 \times noval)}$$

If evaluation of one of the arguments raises an exception $st_1[mode] \neq normal$ is true, and the `throw` statement will be skipped.

3.2.11 Local Variable Access

JLS does not really describe the evaluation of a local variable (see JLS 6.5.6.1 and JLS 15.2). Obviously, its value is looked up somewhere and returned.

$$\frac{st[mode] = normal}{(v \times st)[\![x]\!](v \times st \times v[x].javaval)}$$

The value of the local variable x is looked up in the variable mapping, $v[x]$, and the selector .javaval applied that returns a *javavalue*. (The variable mapping contains all values that can possibly occur, e.g. stores.) There is only one global variable mapping v that contains the values of all variables.

3.2.12 Static Field Access

A static field access (JLS 15.11) can be a *first active use* that causes a class initialization. This is captured by the first rule:

$$\frac{st[mode] = normal \quad \neg\ initdoneP(fs.class, st) \quad (v \times st)[\![\texttt{static}(fs.class)]\!](v_0 \times st_0) \quad (v_0 \times st_0)[\![\texttt{SFieldAccess(fs, ty)}]\!](v_1 \times st_1 \times val_1)}{(v \times st)[\![\texttt{SFieldAccess(fs, ty)}]\!](v_1 \times st_1 \times val_1)}$$

The field access expression contains a field specification fs (which in turn consists of a field name, its type, and the class where it is declared). As discussed in chapter 2.5 a static field may not be used with an invoking expression. *fs.class* is the class where the field is declared, and *initdoneP* is true if the class is already initialized (or a predefined class which is always considered initialized, see chapter 3.1.5). If this is not the case, the new Java statement `static` (see chapter 3.3.21) will handle initialization (or raise an error if the class is in an erroneous state). After initialization the field access is performed again. This will yield another result, because after evaluation of the `static` statement the mode is either not normal or the class is initialized. The second rule handles the normal case:

$$\frac{st[mode] = normal \quad initdoneP(fs.class, st)}{(v \times st)[\![\texttt{SFieldAccess(fs, ty)}]\!](v \times st \times coerce(st[jvmref - fs], ty))}$$

If the class is initialized the value of the field is looked up in the store. As described in chapter 3.1.3 static fields are stored under the special reference *jvmref*. The value (of sort *javavalue*) is coerced to the expected primitive result type to allow primitive type safety even for arbitrary (non-compatible) stores. See chapter 5.1 for a full discussion.

3.2.13 Instance Field Access

Instance field access is also discussed in JLS 15.11. The invoking expression e may not be null, otherwise a `NullPointerException` is thrown.

$$\frac{(v \times st)[\![e]\!](v_0 \times st_0 \times val_0) \quad val_0 = null \vee st_0[mode] \neq normal \quad (v_0 \times st_0)[\![\texttt{throw new NullPointerException();}]\!](v_1 \times st_1)}{(v \times st)[\![\texttt{FieldAccess(e, fs, ty)}]\!](v_1 \times st_1 \times noval)}$$

In the normal case the result is looked up in the store (under the given reference and field specification, see chapter 3.1.3) and coerced to the expected primitive result type (see chapter 5.1).

$$\frac{(v \times st)[\![e]\!](v_0 \times st_0 \times val_0) \quad val_0 \neq null \wedge st_0[mode] = normal}{(v \times st)[\![\texttt{FieldAccess(e, fs, ty)}]\!](v_0 \times st_0 \times coerce(st_0[val_0.refval - fs], ty))}$$

3.2.14 Array Access

Array access $e_0[e_1]$ is described in JLS 15.13. Both expressions (the array reference and the index) are evaluated, then e_0 is checked to be not null (otherwise a `NullPointerException` is thrown), then the index is checked (in case of an illegal index an `ArrayIndexOutOfBoundsException` is thrown).

$$\frac{(v \times st)[\![e_0]\!](v_0 \times st_0 \times val_0) \quad (v_0 \times st_0)[\![e_1]\!](v_1 \times st_1 \times val_1) \quad st_1[mode] \neq normal \vee val_0 = null \quad (v_1 \times st_1)[\![\texttt{throw new NullPointerException();}]\!](v_2 \times st_2)}{(v \times st)[\![\ e_0[e_1]\ ]\!](v_2 \times st_2 \times noval)}$$

$$\frac{(v \times st)[\![e_0]\!](v_0 \times st_0 \times val_0) \quad (v_0 \times st_0)[\![e_1]\!](v_1 \times st_1 \times val_1) \quad st_1[mode] = normal \quad val_0 \neq null \quad \neg\ (0 \leq val2int(val_1) \wedge val2int(val_1) < val2int(st_1[val_0.refval - _length])) \quad (v_1 \times st_1)[\![\texttt{throw new ArrayIndexOutOfBoundsException();}]\!](v_2 \times st_2)}{(v \times st)[\![\ e_0[e_1]\ ]\!](v_2 \times st_2 \times noval)}$$

val2int converts bytes, and shorts to an integer.

$$\frac{(v \times st)[\![e_0]\!](v_0 \times st_0 \times val_0) \quad (v_0 \times st_0)[\![e_1]\!](v_1 \times st_1 \times val_1) \quad st_1[mode] = normal \wedge val_0 \neq null \quad 0 \leq val2int(val_1) \wedge val2int(val_1) < val2int(st_1[val_0.refval - length])}{(v \times st)[\![\ e_0[e_1]\ ]\!](v_1 \times st_1 \times coerce(st_1[val_0.refval - val2int(val_1)], ty))}$$

If no exception occurs the value is looked up in the store and coerced to the expected primitive type (see chapter 5.1, *ty* is the static type of the array access).

3.2.15 Local Variable Assignment

Similar to the four types of accesses, four types of simple assignments (JLS 15.26.1) are defined. The local variable assignment modifies the variable mapping if evaluation of the right hand side completes normally.

$$\frac{(v \times st)[\![e]\!](v_0 \times st_0 \times val_0) \quad st_0[mode] \neq normal}{(v \times st)[\![x = e]\!](v_0 \times st_0 \times val_0)}$$

$$\frac{(v \times st)[\![e]\!](v_0 \times st_0 \times val_0) \quad st_0[mode] = normal}{(v \times st)[\![x = e]\!](v_0[x, val_0] \times st_0 \times val_0)}$$

The value for x is updated to the value of e (computed in val_0), and the result of the assignment is also val_0. We assume that all primitive conversions have been made explicit for assignments. This means that x and e should have the same type. Otherwise there is no guarantee that the sort of the variable x matches the computed value. For example, if x has sort *byte*, then val_0 should be a *javavalue* containing a byte, i.e. $is_bytevalue(val_0)$ should be true, but this is not guaranteed. See chapter 5.1.

3.2.16 Static Field Assignment

In Java, a static field assignment is a simple assignment with a static field access on the left hand side (JLS 15.26.1). First the right hand side is evaluated (the static field access may not have an invoking expression, see chapter 2.5). If this completes normally a first active use may occur. If this also completes normally the assignment is carried out. The first rule handles the exception case.

$$\frac{(v \times st)[\![e]\!](v_0 \times st_0 \times val_0) \quad (v_0 \times st_0)[\![c.f]\!](v_1 \times st_1 \times val_1) \quad st_1[mode] \neq normal}{(v \times st)[\![c.f = e]\!](v_1 \times st_1 \times val_1)}$$

The first active is handled by evaluating the static field access from the left hand side of the assignment. If no exception occurs the assignment is carried out:

$$\frac{(v \times st)[\![e]\!](v_0 \times st_0 \times val_0) \quad (v_0 \times st_0)[\![c.f]\!](v_1 \times st_1 \times val_1) \quad st_1[mode] = normal}{(v \times st)[\![c.f = e]\!](v_1 \times st_1[jvmref - c.f, val_0] \times val_1)}$$

The static field (under the special reference *jvmref*) is updated in the store. In general, there is no guarantee that the value matches the type of the field.

The evaluation order is correct: The right hand side is evaluated before a first active use occurs. JLS 15.26.1 states that "First, the left-hand operand is evaluated to produce a variable." This seems to imply that the first active use occurs first, because the left-hand side is a static field access. However, SUN's Java compiler produces different code. The same happens for an instance field assignment (see below). von Oheimb [vO01] first evaluates the left-hand side of an assignment (chapter 3.2.7). This means the first active use comes first. However, he mentions that "We had to fix the exact positions [for static initialization], but we are not sure if this should be considered as an improvement of the specification or if the positions have been left unspecified intentionally." Static fields "are not discussed here" by Huisman [Hui01] (p. 10). Börger et. al. [SSB01] have the first active use after the right-hand side is evaluated.

3.2.17 Instance Field Assignment

Instance fields are also covered in JLS 15.26.1. Again – as with static field assignment – it states that first the field access $e.f$ is evaluated, then the right hand side e_0. This would mean that if $e = null$, the `NullPointerException` is thrown before e_0 is evaluated. However, the JDK compiler does not work this way. Correct is: e and e_0 are evaluated, then e is checked to be not null.

$$\frac{\begin{array}{c}(v \times st)[\![e]\!](v_0 \times st_0 \times val_0) \quad (v_0 \times st_0)[\![e_0]\!](v_1 \times st_1 \times val_1) \\ val_0 \neq null \wedge st_1[mode] = normal\end{array}}{(v \times st)[\![e.fs = e_0]\!](v_1 \times st_1[val_0.refval - fs, val_1] \times val_1)}$$

$$\frac{\begin{array}{c}(v \times st)[\![e]\!](v_0 \times st_0 \times val_0) \quad (v_0 \times st_0)[\![e_0]\!](v_1 \times st_1 \times val_1) \\ val_0 = null \vee st_1[mode] \neq normal \\ (v_1 \times st_1)[\![\texttt{throw new NullPointerException();}]\!](v_2 \times st_2)\end{array}}{(v \times st)[\![e.f = e_0]\!](v_2 \times st_2 \times noval)}$$

3.2.18 Array Assignment

See JLS 15.26.1. All three expressions are evaluated, then the array reference e_0 is checked to be not null (otherwise a `NullPointerException` is thrown), then the index e_1 is checked (otherwise an `ArrayIndexOutOfBoundsException` is thrown), and finally the runtime type is checked to be *assignment compatible* (otherwise an `ArrayStoreException` is thrown). If the evaluation of e_1, e_2, or e_3 raises an exception the throws are skipped. Only in the fourth rule the condition $st_2[mode] = normal$ must be included.

$$\frac{\begin{array}{c}(v \times st)[\![e_0]\!](v_0 \times st_0 \times val_0) \quad (v_0 \times st_0)[\![e_1]\!](v_1 \times st_1 \times val_1) \\ (v_1 \times st_1)[\![e_2]\!](v_2 \times st_2 \times val_2) \quad st_2[mode] \neq normal \vee val_0 = null \\ (v_2 \times st_2)[\![\texttt{throw new NullPointerException();}]\!](v_3 \times st_3)\end{array}}{(v \times st)[\![e_0[e_1] = e_2]\!](v_3 \times st_3 \times noval)}$$

$$\frac{\begin{array}{c}(v \times st)[\![e_0]\!](v_0 \times st_0 \times val_0) \quad (v_0 \times st_0)[\![e_1]\!](v_1 \times st_1 \times val_1) \\ (v_1 \times st_1)[\![e_2]\!](v_2 \times st_2 \times val_2) \quad st_2[mode] = normal \wedge val_0 \neq null \\ \neg\ (0 \leq val2int(val_1) \wedge val2int(val_1) < val2int(st_2[val_0.refval - _length]) \\ (v_2 \times st_2)[\![\texttt{throw new ArrayIndexOutOfBoundsException();}]\!](v_3 \times st_3)\end{array}}{(v \times st)[\![e_0[e_1] = e_2]\!](v_3 \times st_3 \times noval)}$$

$$\frac{\begin{array}{c}(v \times st)[\![e_0]\!](v_0 \times st_0 \times val_0) \quad (v_0 \times st_0)[\![e_1]\!](v_1 \times st_1 \times val_1) \\ (v_1 \times st_1)[\![e_2]\!](v_2 \times st_2 \times val_2) \quad st_2[mode] = normal \wedge val_0 \neq null \\ 0 \leq val2int(val_1) \wedge val2int(val_1) < val2int(st_2[val_0.refval - _length]) \\ \neg\ \ asgcomp(val_2, st_2[val_0.refval - _type].type.type, st_2, tds) \\ (v_2 \times st_2)[\![\texttt{throw new ArrayStoreException();}]\!](v_3 \times st_3)\end{array}}{(v \times st)[\![e_0[e_1] = e_2]\!](v_3 \times st_3 \times noval)}$$

$$\frac{\begin{array}{c}(v \times st)[\![e_0]\!](v_0 \times st_0 \times val_0) \quad (v_0 \times st_0)[\![e_1]\!](v_1 \times st_1 \times val_1) \\ (v_1 \times st_1)[\![e_2]\!](v_2 \times st_2 \times val_2) \quad st_2[mode] = normal \wedge val_0 \neq null \\ 0 \leq val2int(val_1) \wedge val2int(val_1) < val2int(st_2[val_0.refval - _length]) \\ asgcomp(val_2, st_2[val_0.refval - _type].type.type, st_2, tds)\end{array}}{(v \times st)[\![e_0[e_1] = e_2]\!](v_2 \times st_2[val_0.refval - val2int(val_1), val_2] \times val_2)}$$

$st_2[val_0.refval - type].type$ selects the runtime type of the array reference $val_0.refval$. This type must be an array type. *.type* selects the component type of this array type. ($st_2[val_0.refval - type]$ has sort *javavalue*. The first *.type* selects the type from a *typeval*, the second *.type* the element type of an array type; see chapter 3.1.1.) The value val_2 to store must be assignment compatible to this type. If val_2 is of a primitive type the two types must be equal since we assume all primitive conversions to be made explicit. *asgcomp* is always true for primitive types. *asgcomp* is described in detail in chapter 3.2.5.

3.2.19 Prefix/Postfix Increment/Decrement Operators

These are `++` and `--`. They can be used as prefix operators (JLS 15.15.1 and 15.15.2), or as postfix operators (JLS 15.14). The argument must be a variable access, i.e. either a local variable, a

static or instance field, or an array component. For a postfix operation the result is the value of the variable, as a side effect it is modified. Internally, $\oplus$ is one of four operators. We have seven rules; the first is for local variables.

$$\frac{st[mode] = normal \quad (v \times st)[\![x]\!](v_0 \times st_0 \times val_0)}{(v \times st)[\![x \oplus]\!](v_0[x, eval(\oplus, val_0)] \times st_0 \times evalres(\oplus, val_0))}$$

Even though written as postfix $\oplus$ can also be a prefix operator. *eval* either adds 1 oder subtracts 1 depending on $\oplus$. *evalres* computes the result of the operation. It is the identity for the postfix operators, and *eval* for the prefix operators. The next two rules are for static fields.

$$\frac{(v \times st)[\![c.f]\!](v_0 \times st_0 \times val_0) \quad st_0[mode] = normal}{(v \times st)[\![c.f \oplus]\!](v_0 \times st_0[\text{jvmref} - c.f, eval(\oplus, val_0)] \times evalres(\oplus, val_0))}$$

The static field access is evaluated. This handles first active use.

$$\frac{(v \times st)[\![c.f]\!](v_0 \times st_0 \times val_0) \quad st_0[mode] \neq normal}{(v \times st)[\![c.f \oplus]\!](v_0 \times st_0 \times val_0)}$$

The next two rules are for instance fields.

$$\frac{(v \times st)[\![e.f]\!](v_1 \times st_1 \times val_1) \quad st_1[mode] = normal \quad (v \times st)[\![e]\!](v_0 \times st_0 \times val_0)}{(v \times st)[\![e.f \oplus]\!](v_1 \times st_1[val_0.refval - f, eval(\oplus, val_1)] \times evalres(\oplus, val_1))}$$

The instance field access $e.f$ is evaluated with $(v \times st)$, and the invoking expression is evaluated with the same context $(v \times st)$. Evaluating $e.f$ handles the case that the invoking expression is null. e is evaluated to obtain the reference $val_0.refval$.

$$\frac{(v \times st)[\![e.f]\!](v_1 \times st_1 \times val_1) \quad st_1[mode] \neq normal}{(v \times st)[\![e.f \oplus]\!](v_1 \times st_1 \times val_1)}$$

The last two rules deal with arrays.

$$\frac{\begin{array}{c}(v \times st)[\![e_0[e_1]]\!](v_2 \times st_2 \times val_2) \quad st_2[mode] = normal \\ (v \times st)[\![e_0]\!](v_0 \times st_0 \times val_0) \quad (v_0 \times st_0)[\![e_1]\!](v_1 \times st_1 \times val_1)\end{array}}{(v \times st)[\![e_0[e_1] \oplus]\!](v_2 \times st_2[val_0.refval - val_1, eval(\oplus, val_2)] \times evalres(\oplus, val_2))}$$

$$\frac{(v \times st)[\![e_0[e_1]]\!](v_2 \times st_2 \times val_2) \quad st_2[mode] \neq normal}{(v \times st)[\![e_0[e_1] \oplus]\!](v_2 \times st_2 \times val_2)}$$

Again, the array access is evaluated to handle errors. If no errors occur the values of the single components are needed, but without evaluating them twice. The result is cast to byte or short if the original value was a byte or short. This can silently cause an overflow: If `x` is a byte with value 127, `x++` will set `x` to the byte value -128. In the rules the argument for the increment/decrement operation is not an arbitrary expression, but an access. This implies that other expressions have no semantics.

3.2.20 Compound Assignment

Compound assignments are described in JLS 15.26.2. In $e_1 \oplus= e_2$, $\oplus$ is one of the 11 binary operations `*`, `/`, `%`, `+`, `-`, `<<`, `>>`, `>>>`, `&`, `^`, `|`. e_1 must be a variable access, either a local variable, a static or instance field, or an array component. Exceptions are thrown (and static initialization occurs) *before* e_2 is evaluated, because a compound assignment works like $e_1 = (ty)(e_1 \oplus e_2)$ except that e_1 is evaluated only once. If e_1 is a byte or short the result is automatically cast back to the original type, even if this causes over- or underflows. There is one rule for every kind of access.

$$\frac{\begin{array}{c}(v \times st)[\![x]\!](v_0 \times st_0 \times val_0) \quad (v_0 \times st_0)[\![e]\!](v_1 \times st_1 \times val_1) \\ (v_1 \times st_1)[\![x = e']\!](v_2 \times st_2 \times val_2)\end{array}}{(v \times st)[\![x \oplus= e]\!](v_2 \times st_2 \times val_2)}$$

The semantics of the compound assignment is reduced to a simple assignment $x = e'$ where $e' = \text{asg2binjexpr}(\oplus, val_0, val_1)$ is defined as $(ty)(val_0 \oplus val_1)$. The values are converted to literals, and the compound assignment either to a binary or an exception binary operation. This is necessary because we distinguish between normal binary operation (that never raise an exception) and division and remainder that may cause an `ArithmeticException`. Then the result must then be cast back to its original type.

$$\frac{\begin{array}{c}(v \times st)[\![c.f]\!](v_0 \times st_0 \times val_0) \quad (v_0 \times st_0)[\![e]\!](v_1 \times st_1 \times val_1) \\ (v_1 \times st_1)[\![c.f = e']\!](v_2 \times st_2 \times val_2)\end{array}}{(v \times st)[\![c.f \oplus\!= e]\!](v_2 \times st_2 \times val_2)}$$

e' is defined in the same manner as above.

$$\frac{\begin{array}{c}(v \times st)[\![e_1.f]\!](v_1 \times st_1 \times val_1) \quad (v_1 \times st_1)[\![e_2]\!](v_2 \times st_2 \times val_2) \\ (v \times st)[\![e_1]\!](v_0 \times st_0 \times val_0) \\ (v_2 \times st_2)[\![val_0.f = e_2']\!](v_3 \times st_3 \times val_3)\end{array}}{(v \times st)[\![e_1.f \oplus\!= e_2]\!](v_3 \times st_3 \times val_3)}$$

e_2' is defined in the same manner as above. The evaluation order guarantees that $e_1.f$ is completely evaluated before e_2 is evaluated. If e_1 is null the `NullPointerException` is thrown before e_2 is evaluated. Then the field assignment uses the already computed values so that in the result $(v_3 \times st_3 \times val_3)$ no expression is evaluated twice. In the second line e_1 is evaluated so that its value val_0 can be used, but the original state $(v \times st)$ is used, and the resulting variable mapping v_0 and store st_0 is not used in the sequel.

$$\frac{\begin{array}{c}(v \times st)[\![e_1[e_2]]\!](v_2 \times st_2 \times val_2) \quad (v_2 \times st_2)[\![e_3]\!](v_3 \times st_3 \times val_3) \\ (v \times st)[\![e_1]\!](v_0 \times st_0 \times val_0) \quad (v_0 \times st_0)[\![e_2]\!](v_1 \times st_1 \times val_1) \\ (v_3 \times st_3)[\![val_0[val_1] = e_3']\!](v_5 \times st_5 \times val_5)\end{array}}{(v \times st)[\![e_1[e_2] \oplus\!= e_3]\!](v_5 \times st_5 \times val_5)}$$

e_3' is defined in the same manner as above. A similar argumentation concerning the evaluation order applies as for the field assignment.

3.2.21 Class Instance Creation

This means creation of new objects. See JLS 12.5 and 15.9. The following should happen for `new` $c(e_1, \ldots, e_n)$:

1. The object is created and its fields are initialized with their default values. This includes the fields of super classes. Here, a first active use can occur.
2. The arguments $e_1, \ldots, e_n$ are evaluated.
3. The constructor is invoked.

JLS 15.9 mentions that an `OutOfMemoryError` can occur. We ignore this possibility: There is an infinite number of references, and a new object can always be added to the store. While it is no problem to define a predicate *full* for a store (for example, the number of references has reached a limit, or by specifying the memory consumption of the stored values, and overhead for objects etc.) there are no useful applications for this. Modern efficient Java implementations do not throw an `OutOfMemoryError` anyway, but fail silently. Java Card has no `OutOfMemoryError`. Since inner classes are not supported in this work a class instance creation expression has no qualifier, and cannot contain an anonymous class declaration.

The evaluation order (the object is created before the arguments are evaluated) is significant when a first active use occurs. This means the class is initialized before the arguments are evaluated. For a static method invocation the evaluation order is exactly the other way around. The first rule handles the *first active use* .

$$\frac{\begin{array}{c} st[mode] = normal \wedge \neg\ initdoneP(c, st) \\ (v \times st)[\![\texttt{static}(c)]\!](v_0 \times st_0) \\ (v_0 \times st_0)[\![\texttt{new}\ c(e_1, \ldots, e_n)]\!](v_1 \times st_1 \times val_1) \end{array}}{(v \times st)[\![\texttt{new}\ c(e_1, \ldots, e_n)]\!](v_1 \times st_1 \times val_1)}$$

If a first active use occurs the class is initialized by the *static* statement (described in chapter 3.3.21) and the *new* expression is evaluated again. This is well defined since either the initialization state of the class is *done* or the mode is not normal.

$$\frac{\begin{array}{c} st[mode] = normal \wedge initdoneP(c, st) \wedge ref = newref(st) \\ (v \times addobj(ref, c, ifields, st))[\![ref.c(e_1, \ldots, e_n)]\!](v_0 \times st_0 \times val_0) \end{array}}{(v \times st)[\![\texttt{new}\ c(e_1, \ldots, e_n)]\!](v_0 \times st_0 \times val_0)}$$

In the second case we first obtain a new non-null reference ($ref = newref(st)$), and add a new object with instance fields $ifields = instfields(c, tds)$ to the store ($addobj(ref, c, ifields, st)$). The function *instfields* also collect the fields of super classes. Then the constructor is invoked on the new object. $ref.c(e_1, \ldots, e_n)$ is an explicit constructor call that is defined below. This captures quite nicely the description in JLS 15.9.4. Computing a new reference must be done by a function, otherwise the semantics would not be deterministic. For example, using a predicate *newref(ref, st)* that is true if the reference does not occur in the store leads to non-determinism.

3.2.22 Array Creation

We distinguish between the creation of a one-dimensional and a multi-dimensional array, because the latter is much more difficult and requires several complicated auxiliary functions. In JLS (first edition) an array creation may cause a first active use for its component class. This is changed in JLS (second edition). We follow the second edition.

First the one-dimensional case:

$$\frac{\begin{array}{c} (v \times st)[\![e]\!](v_0 \times st_0 \times val_0) \quad val2int(val_0) < 0 \vee st_0[mode] \neq normal \\ (v_0 \times st_0)[\![\texttt{throw new NegativeArraySizeException();}]\!](v_1 \times st_1) \end{array}}{(v \times st)[\![\texttt{new}\ ty[e]]\!](v_1 \times st_1 \times noval)}$$

$$\frac{(v \times st)[\![e]\!](v_0 \times st_0 \times val_0) \quad 0 \leq val2int(val_0) \wedge st_0[mode] = normal \wedge ref = newref(st_0)}{(v \times st)[\![\texttt{new}\ ty[e]]\!](v_0 \times addarray(ref, ty, val2int(val_0), st_0) \times ref)}$$

addarray adds an array of type ty to the store of length $val2int(val_0)$, where all array elements are set to their default values.

In the multi-dimensional case the expression the the form $\texttt{new}\ ty[e_1] \ldots [e_m][n]$ where ty is a type, $e_1, \ldots, e_m$ the list of arguments for the dimensions, and n the number of additional dimensions. It is needed to determine the correct default values for the array components: null, if $n > 0$, the default value for ty otherwise. Either $m \neq 1$ or $n > 0$ must be true, otherwise the array is one-dimensional.

$$\frac{\begin{array}{c} (v \times st)[\![e_1 \ldots e_m]\!](v_0 \times st_0 \times vals_0) \quad \neg\ posints(val2ints(vals_0)) \vee st_0[mode] \neq normal \\ (v_0 \times st_0)[\![\texttt{throw new NegativeArraySizeException();}]\!](v_1 \times st_1) \end{array}}{(v \times st)[\![\texttt{new}\ ty[e_1] \ldots [e_m][n]]\!](v_0 \times st_0 \times noval)}$$

$$\frac{\begin{array}{c} (v \times st)[\![e_1 \ldots e_m]\!](v_0 \times st_0 \times vals_0) \quad posints(val2ints(vals_0)) \wedge st_0[mode] = normal \\ refs = newref_list(countrefs(vals2int(vals_0)), st_0) \end{array}}{(v \times st)[\![\texttt{new}\ ty[e_1] \ldots [e_m][n]]\!]v_0 \times addarrays(refs, ty, vals2ints(vals_0), n, st_0) \times refs.first}$$

The difficulty with multi-dimensional arrays is that several new arrays (and therefore several new references) are created at once. For example, an expression `new int[3][5][2]` creates 19 arrays and requires 19 references: one new reference for the first array, three further new references for

three arrays of length 5, and $3 * 5$ further new references for 15 arrays of length 2. The number of new references is computed by *countrefs*, *newref_list* generates that many new references, and *addarrays* creates the arrays in the store. *addarrays* contains a nested, double recursion, and its specification is omitted here.

3.2.23 Array Initializer

An array initializer (JLS 10.6 and 15.10) has the form $\{e_1, \ldots, e_n\}$, the components of the (newly created) array in curly brackets. In Java it can occur in this syntax on the right hand side of a variable or field declaration or inside another array initializer, for example `int a[][] = {{1, 2}, null};`. In the form `new a[][]{{1, 2}, null}` it can appear anywhere. As described in table 2.1 the array initializer expression used in this work consists of a list of expressions and result type of the expression.

It may be worth to remark that an array initializer cannot cause an `ArrayStoreException` because the compiler can check that the values can indeed be stored in the array. (Its runtime type is equal to its static type.)

$$\frac{(v \times st)[\![e_1 \ldots e_n]\!](v_0 \times st_0 \times vals_0) \quad st_0[mode] \neq normal}{(v \times st)[\![\{e_1, \ldots, e_n\}]\!](v_0 \times st_0 \times noval)}$$

Evaluation of the expressions must complete normally.

$$\frac{(v \times st)[\![e_1 \ldots e_n]\!](v_0 \times st_0 \times vals_0) \quad st_0[mode] = normal \quad ref = newref(st_0)}{(v \times st)[\![\{e_1, \ldots, e_n\}]\!](v_0 \times addarray(ref, ty, vals_0, st_0) \times ref)}$$

Then a new reference ref is allocated. *addarray* (again overloaded with *addarray* in chapter 3.2.22) adds the array with values $vals_0$ for the new reference ref.

3.2.24 Explicit Constructor Invocation

In a correct Java program an *explicit constructor invocation* can occur only at the beginning of a constructor. However, it is also used to describe the behaviour of a class instance creation expression (with `new` *class* ...). At the beginning of a constructor it is either a call of a constructor of the same class (`this`$(e_1, \ldots, e_n)$) or of the super class (`super`$(e_1, \ldots, e_n)$). We assume that the compiler replaces `this` or `super` with an explicit constructor call that contains the correct class name. Furthermore, the constructor requires an object on which it is invoked (a newly created object, or the value of **this**), so that the call is $e.c(e_1, \ldots, e_n)$. This leads to a uniform treatment of constructor calls.

We explicitly include the possibility that no implementation for the constructor is available. This models *native* constructors. Other formal Java semantics assume that the full source code is available. However, this is neither the case for the Java Card API nor the standard JDK by Sun. A native constructor declaration has an empty body, and without special treatment the semantics would define that the constructor does nothing – even if that is not its real behaviour. The second rule deals with this case.

All together there are five rules: 1. Evaluation of the arguments raises an exception, 2. no implementation is available, 3. evaluation of the constructor body raises an exception, 4. the body ends with a return, and 5. the body completes normally.

First the arguments are evaluated. If this does not complete normally the body is not evaluated:

$$\frac{\begin{array}{c}(v \times st)[\![e]\!](v_0 \times st_0 \times val_0) \quad (v_0 \times st_0 \times val_0)[\![e_1 \ldots e_n]\!](v_1 \times st_1 \times vals_1) \\ st_1[mode] \neq normal\end{array}}{(v \times st)[\![e.c(e_1, \ldots, e_n)]\!](v_1 \times st_1 \times noval)}$$

One could expect to see a check that the invoking expression is not null. However, since the invoking expression is not part of Java itself (inner classes are not supported) we are free to specify its behaviour as we want.

The second rule handles the case that the constructor is not implemented:

$$\frac{\begin{array}{c}(v \times st)[\![e]\!](v_0 \times st_0 \times val_0) \quad (v_0 \times st_0 \times val_0)[\![e_1 \ldots e_n]\!](v_1 \times st_1 \times vals_1) \\ st_1[mode] = normal \quad \neg\ implemented(getConstr(c, tys, tds)) \\ is_defined(st_1, val_0, c, vals_1, tys, tds) \quad st_2 = constrsem(st_1, val_0, c, vals_1, tys, tds)\end{array}}{(v \times st)[\![e.c(e_1, \ldots, e_n)]\!](v_1 \times st_2 \times val_0)}$$

getConstr(c, tys, tds) searches the matching constructor declaration in class c with argument types *tys* in the type declarations *tds*. The constructor call $e.c(e_1, \ldots, e_n)$ (in its mathematical abbreviation) is actually `ConstrCall(e, c, jexprs, tys, ty)`. (`e` is the invoking expression, `c` the class of the constructor, `jexprs` $= e_1, \ldots, e_n$ the parameters, and `tys` the types of the formal parameters of the correct constructor declaration as computed by the compiler. `ty` is the result type of the expression.) So all necessary arguments for the function *getConstr* are available.

getConstr works as follows:

1. If the class `c` is a predefined class and `tys` the empty list the constructor declaration `c() { }` is returned. All predefined classes have an empty constructor, and there is no need to include a `super()` call because all predefined classes are considered initialized.
2. If `c` is predefined, but `tys` is not empty, the constructor declaration `native c() { }` is returned. The *native* modifier indicates that no implementation for the constructor is available. It does not matter that the formal parameter types do not match `tys`.
3. If the class `c` does not exist in the type declarations *tds* again `native c() { }` is returned.
4. Otherwise the matching declaration with paramter types `tys` is searched in the class `c`. If it does not exist `native c() { }` is returned; otherwise the declaration (which may or may not contain a *native* modifier).

A constructor declaration is considered to be *implemented* if it does not contain the *native* modifier. If the constructor is not implemented the semantics of the constructor call is defined by the predicate *is_defined* and the function *constrsem*. Both are unspecified (except for some axioms described below), and have all relevant arguments as input. This means a *native* constructor can have an almost arbitrary behaviour (that must be specified by the user). The restrictions are:

1. Its semantics must be a function. This means it must be deterministic. Non-termination can be modelled if the predicate *is_defined* is false. (There is no semantics rule for the case that the constructor is not implemented and *is_defined* is false. Hence, it has no semantics.
2. The constructor can only modify the store, not the variable mapping. The value of a constructor call is always the value of the invoking expression (a constructor body may contain a `return` statement without expression, but may never return a value).
3. It may not delete keys from the store.
4. The mode in the new store may not be a *return* or *break* mode. This is necessary to ensure primitive type correctness (see chapter 5.1).

These restrictions are necessary to ensure some essential properties about the store. An implemented constructor (even if not type correct) and, in fact, all other expresssions and statements ensure these four restrictions.

If the constructor is implemented its body is evaluated. This leads to the last three rules: An exception is thrown, the body ends with a `return` or the body ends normally.

$$\frac{\begin{array}{c}(v \times st)[\![e]\!](v_0 \times st_0 \times val_0) \quad (v_0 \times st_0 \times val_0)[\![e_1 \ldots e_n]\!](v_1 \times st_1 \times vals_1) \\ st_1[mode] = normal \wedge implemented(getConstr(c, tys, tds)) \\ (bind(v_1, this + vars, val_0 + vals_1) \times st_1)[\![\alpha]\!](v_2 \times st_2) \\ st_2[mode] \neq normal \wedge \neg\ is_return_mode(st_2[mode])\end{array}}{(v \times st)[\![e.c(e_1, \ldots, e_n)]\!](v_1 \times st_2 \times noval)}$$

α is the body of the constructor with formal parameter variables *vars*, i.e. *getConstr(c, tys, tds)* returned a constructor declaration *c(params)* α where *params* is the list of parameters with types *tys* and variables *vars*. The local variables are bound to the values of the call arguments and **this** to the invoking reference $val_0.refval$. This binding is done in the 'global' variable mapping – there is no need to introduce 'local' mappings. Then the constructor body is evaluated. If the resulting mode is neither normal nor *return* (for example, a *throw*) it remains unchanged. The new variable mapping is the mapping *before* the binding occurred. This captures the intended meaning of local variables.

If α completes with a `return` the mode is set back to normal and the value val_0 of the invoking reference e is returned as the value of the constructor call.

$$\frac{\begin{array}{c}(v \times st)[\![e]\!](v_0 \times st_0 \times val_0) \quad (v_0 \times st_0 \times val_0)[\![e_1 \ldots e_n]\!](v_1 \times st_1 \times vals_1) \\ st_1[mode] = normal \wedge implemented(getConstr(c, tys, tds)) \\ (bind(v_1, this + vars, val_0 + vals_1) \times st_1)[\![\alpha]\!](v_2 \times st_2) \\ is_return_mode(st_2[mode])\end{array}}{(v \times st)[\![e.c(e_1, \ldots, e_n)]\!](v_1 \times st_2[mode, normal] \times val_0)}$$

If the body completes normally the value val_0 of the invoking reference e is returned as the value of the constructor call.

$$\frac{\begin{array}{c}(v \times st)[\![e]\!](v_0 \times st_0 \times val_0) \quad (v_0 \times st_0 \times val_0)[\![e_1 \ldots e_n]\!](v_1 \times st_1 \times vals_1) \\ st_1[mode] = normal \wedge implemented(getConstr(c, tys, tds)) \\ (bind(v_1, this + vars, val_0 + vals_1) \times st_1)[\![\alpha]\!](v_2 \times st_2) \\ st_2[mode] = noval\end{array}}{(v \times st)[\![e.c(e_1, \ldots, e_n)]\!](v_1 \times st_2 \times val_0)}$$

Another possibility is to consider the result of a constructor call as `void`. In this case the call would never return a value (but always the dummy `noval`), and its result type would be `void` as for a void method.

3.2.25 Method invocation

Eleven rules define the semantics of a method invocation. Three different kinds of method invocation are distinguished: static method invocation that may cause a first active use (5 rules), instance method invocation that requires a null pointer check (4 rules), and a special interface invocation mode (2 rules). Several of the rules are very similar, however, we refrained from merging rules in order to achieve more clarity for the sake of brevity.

Method invocation is described in JLS 15.12, first three compile-time steps, then the run-time evaluation of the method invocation in 15.12.4. Since the semantics describes the run-time behaviour we assume that all compile-time steps have been carried out. This is captured by the arguments of the method invocation: In addition to the method name, the arguments for the method and an invoking expression (denoted $e.m(e_1, \ldots, e_n)$, m below) the method call has an *invocation mode* and a method *signature*. The method signature consists of the formal argument types of the chosen compile-time method declaration (see JLS 15.12.3), and its result type. The invocation mode is also computed by the compiler, and can be either *static* (in this case it also contains the name of the class containing the chosen declaration), *nonvirtual* (for private methods, also containing a class name), *super* (containing the class name of the super class), *virtual* (for dynamic method lookup), or *interface*. This additional information is identical to the description in JLS.

We begin with the *static* method invocation. First, the invoking expression and the arguments are evaluated. In Java, a static method can have an invoking expression. If it does not have one we assume that the compiler added `null` as an invoker. The arguments are evaluated before a

first active use occurs. If this causes an exception the method body is not evaluated.

$$\frac{\begin{array}{c} is_static(invmo) \\ (v \times st)[\![e]\!](v_0 \times st_0 \times val_0) \quad (v_0 \times st_0)[\![e_1 \ldots e_n]\!](v_1 \times st_1 \times vals_1) \\ st_1[mode] \neq normal \end{array}}{(v \times st)[\![e.m(e_1, \ldots, e_n)]\!](v_1 \times st_1 \times noval)}$$

invmo is the invocation mode. If the class is not yet initialized we have a first active use.

$$\frac{\begin{array}{c} is_static(invmo) \\ (v \times st)[\![e]\!](v_0 \times st_0 \times val_0) \quad (v_0 \times st_0)[\![e_1 \ldots e_n]\!](v_1 \times st_1 \times vals_1) \\ st_1[mode] = normal \quad \neg\ initdoneP(invo.class, st_1) \\ (v_1 \times st_1)[\![static(invo.class)]\!](v_2 \times st_2) \\ (v_2 \times st_2)[\![val_0.m(vals_1)]\!](v3 \times st_3 \times val_3) \end{array}}{(v \times st)[\![e.m(e_1, \ldots, e_n)]\!](v_3 \times st_3 \times val_3)}$$

invmo.class selects the class from the invocation mode. If the class is not initialized ($\neg$ *initdoneP* is true) it may be either uninitialized or in an error state. These cases are handled by the additional *static* statement (see 3.3.21). After static initialization the method call is evaluated again, however with the evaluated invoker val_0 and evaluated arguments $vals_1$.

If no first active use occurs the method body is looked up. The next rule deals with the case that either no matching declaration is found, or that no implementation is available.

$$\frac{\begin{array}{c} is_static(invmo) \\ (v \times st)[\![e]\!](v_0 \times st_0 \times val_0) \quad (v_0 \times st_0)[\![e_1 \ldots e_n]\!](v_1 \times st_1 \times vals_1) \\ st_1[mode] = normal \quad initdoneP(invo.class, st_1) \\ \neg\ implemented(getSMethod(invmo.class, m, tys, ty, tds)) \\ is_defined(st_1, val_0, m, invmo, vals_1, tys, ty, tds) \\ methodsem(st_1, val_0, m, invmo, vals_1, tys, ty, tds) = val_2 \times st_2 \end{array}}{(v \times st)[\![e.m(e_1, \ldots, e_n)]\!](v_1 \times st_2 \times coerce(val_2, ty))}$$

The idea is similar to the constructor call (see 3.2.24). *getSMethod* is used to look up a declaration for a static method call. Its arguments are the class to search, the method name, and the method signature. A type correct program must contain a matching declaration, however, since we do not assume type correctness, there may be no matching declaration. In this case *getSMethod* returns a dummy declaration `static native void m() { }` . The modifier `native` indicates that the method is not implemented. If a matching declaration was found it may also have the modifier `native`. This means no implementation is available. This models *native* or API methods where no source code is available.

The behaviour of such a method is then specified by the function *methodsem*. *methodsem* itself is unspecified and allows an (almost) arbitrary behaviour for the method with the following restrictions:

1. *methodsem* is a total function. Non-termination can be modelled if the predicate *is_defined* is false.
2. The behaviour is deterministic.
3. The method cannot modify the variable mapping, only the store.
4. The method may not delete keys from the store, or raise a mode that is *return* or *break*. (A type correct method call either terminates normally or raises an exception.) This is specified by three axioms to ensure some type correctness properties. See chapter 5.1.

The result value val_2 may be any value. There is no guarantee that it matches the expected result type. Therefore, to ensure at least a primitive type correctness, it is coerced to the result type. See chapter 5.1.

The next two rules complete the semantics of a static method invocation. The method body is evaluated, and either returns a value or not.

$$\frac{\begin{array}{c} is_static(invmo) \\ (v \times st)[\![e]\!](v_0 \times st_0 \times val_0) \quad (v_0 \times st_0)[\![e_1 \ldots e_n]\!](v_1 \times st_1 \times vals_1) \\ st_1[mode] = normal \quad initdoneP(invo.class, st_1) \\ implemented(getSMethod(invmo.class, m, tys, ty, tds)) \\ (bind(v_1, vars, vals_1) \times st1)[\![\alpha]\!](v_2 \times st_2) \quad is_return_mode(st_2[mode]) \end{array}}{(v \times st)[\![e.m(e_1, \ldots, e_n)]\!](v_1 \times st_2[mode, normal] \times coerce(st_2[mode].val, ty))}$$

getSMethod(invmo.class, str, tys, ty, tds) returns a method declaration. α is its body, and $vars$ are the formal parameter variables. They are bound to the values $vals_1$ of the arguments (the value of the invoking expression is discarded). Then the body is evaluated. If a value is returned (the mode in the store is *return*, i.e. $is_return_mode(st_2[mode])$ is true, this implies that no exception occurred) it is selected from the mode ($st_2[mode].val$), and coerced to the expected primitive result type. A type correct program will always return a value of the expected type, however, there is no guarantee that the store was only modified by a type correct program. Therefore, to ensure some type correctness properties, the result is coerced. See chapter 5.1 for a full discussion. After the method body the mode is set back to normal, and the variable mapping from before the binding of the parameter variables is used.

If the method body does not return a value (either because an exception occurred, or because the body terminated normally) the store remains unchanged, and the dummy *noval* is used as the result:

$$\frac{\begin{array}{c} is_static(invmo) \\ (v \times st)[\![e]\!](v_0 \times st_0 \times val_0) \quad (v_0 \times st_0)[\![e_1 \ldots e_n]\!](v_1 \times st_1 \times vals_1) \\ st_1[mode] = normal \quad initdoneP(invo.class, st_1) \\ implemented(getSMethod(invmo.class, m, tys, ty, tds)) \\ (bind(v_1, vars, vals_1) \times st1)[\![\alpha]\!](v_2 \times st_2) \quad \neg\ is_return_mode(st_2[mode]) \end{array}}{(v \times st)[\![e.m(e_1, \ldots, e_n)]\!](v_1 \times st_2 \times noval)}$$

This finishes the static method invocation. The next four rules handle the *nonvirtual*, *super*, and *virtual* invocation modes. The only differences to the static method call are: No first active use can occur, the invoking expression may not be null, and a different function to look up the declaration is used. The first rule handles exceptions during argument evaluation and a null invoker, the second deals with an unimplemented method, the third with a method body that returns a value, and the fourth with one that does not.

$$\frac{\begin{array}{c} \neg\ is_static(invmo) \wedge \neg\ is_interface_mode(invmo) \\ (v \times st)[\![e]\!](v_0 \times st_0 \times val_0) \quad (v_0 \times st_0)[\![e_1 \ldots e_n]\!](v_1 \times st_1 \times vals_1) \\ \neg\ (st_1[mode] = normal \wedge val_0 \neq null) \\ (v_1 \times st_1)[\![\texttt{throw new NullPointerException();}]\!](v_2 \times st_2) \end{array}}{(v \times st)[\![e.m(e_1, \ldots, e_n)]\!](v_2 \times st_2 \times noval)}$$

As in other rules the **throw new NullPointerException();** is evaluated only if the mode is normal, i.e. evaluation of the arguments raised no exception.

$$\frac{\begin{array}{c} \neg\ is_static(invmo) \wedge \neg\ is_interface_mode(invmo) \\ (v \times st)[\![e]\!](v_0 \times st_0 \times val_0) \quad (v_0 \times st_0)[\![e_1 \ldots e_n]\!](v_1 \times st_1 \times vals_1) \\ st_1[mode] = normal \wedge val_0 \neq null \\ \neg\ implemented(getMethod(st_1[val_0.refval - _type].type.class, m, invmo, tys, ty, tds)) \\ is_defined(st_1, val_0, m, invmo, vals_1, tys, ty, tds) \\ methodsem(st_1, val_0, m, invmo, vals_1, tys, ty, tds) = st_2 \times val_2 \end{array}}{(v \times st)[\![e.m(e_1, \ldots, e_n)]\!](v_1 \times st_2 \times coerce(val_2, ty))}$$

getMethod is used to look up the method declaration for the call. Its first argument is the runtime class of the invoker ($st_1[val_0.refval - _type].type.class$), the other arguments the method name,

invocation mode, argument and result types, and the type declarations *tds*. *getMethod* performs a dynamic method lookup. It computes an initial class S, and then searches S and its super classes for a matching method declaration. If none is found a dummy declaration `native void m() { }` is returned; the modifier `native` indicates that no implementation is available. The initial class S is the runtime class of the invoker (the first argument to *getMethod*) if the invocation mode is *virtual*, and the class contained in the invocation mode for *nonvirtual* and *super*. (For a type correct program and *nonvirtual* mode the initial class must contain a matching method declaration, for a *super* mode the declaration may be contained in a super class.) If the returned method declaration is not implemented *methodsem* is used in the same manner as for a static method invocation.

The next two rules evaluate the method body. They are similar to the static method invocation except for the preconditions:

$$\frac{\begin{array}{c}\neg\ is_static(invmo) \wedge \neg\ is_interface_mode(invmo) \\ (v \times st)[\![e]\!](v_0 \times st_0 \times val_0) \quad (v_0 \times st_0)[\![e_1 \ldots e_n]\!](v_1 \times st_1 \times vals_1) \\ st_1[mode] = normal \wedge val_0 \neq null \\ implemented(getMethod(st_1[val_0.refval - _type].type.class, m, invmo, tys, ty, tds)) \\ (bind(v_1, this + vars, val_0 + vals_1) \times st_1)[\![\alpha]\!](v_2 \times t_2) \quad is_return_mode(st_2[mode])\end{array}}{(v \times st)[\![e.m(e_1, \ldots, e_n)]\!](v_1 \times st_2[mode, normal] \times coerce(st_2[mode].val, ty))}$$

α is the method body, *vars* the formal parameter variables. Additionally, **this** is bound to the value of the invoking expression.

$$\frac{\begin{array}{c}\neg\ is_static(invmo) \wedge \neg\ is_interface_mode(invmo) \\ (v \times st)[\![e]\!](v_0 \times st_0 \times val_0) \quad (v_0 \times st_0)[\![e_1 \ldots e_n]\!](v_1 \times st_1 \times vals_1) \\ st_1[mode] = normal \wedge val_0 \neq null \\ implemented(getMethod(st_1[val_0.refval - _type].type.class, m, invmo, tys, ty, tds)) \\ (bind(v_1, this + vars, val_0 + vals_1) \times st_1)[\![\alpha]\!](v_2 \times t_2) \quad \neg\ is_return_mode(st_2[mode])\end{array}}{(v \times st)[\![e.m(e_1, \ldots, e_n)]\!](v_1 \times st_2 \times noval)}$$

This completes the rules for dynamic method lookup. The remaining two rules deal with an *interface* invocation mode. This mode is used in a slightly different manner as described in JLS. It is intended for interface methods where no implementing class declaration (i.e. the source code) is available. Therefore, no lookup occurs to find a method declaration, and the invoking expression may be null. The first rule handles the case that evaluation of the arguments raises an exception, the second rule delegates the semantics to *methodsem*.

$$\frac{\begin{array}{c}is_interface_mode(invmo) \\ (v \times st)[\![e]\!](v_0 \times st_0 \times val_0) \quad (v_0 \times st_0)[\![e_1 \ldots e_n]\!](v_1 \times st_1 \times vals_1) \\ st_1[mode] \neq normal\end{array}}{(v \times st)[\![e.m(e_1, \ldots, e_n)]\!](v_1 \times st_1 \times noval)}$$

$$\frac{\begin{array}{c}is_interface_mode(invmo) \\ (v \times st)[\![e]\!](v_0 \times st_0 \times val_0) \quad (v_0 \times st_0)[\![e_1 \ldots e_n]\!](v_1 \times st_1 \times vals_1) \\ st_1[mode] = normal \\ is_defined(st_1, val_0, m, invmo, vals_1, tys, ty, tds) \\ methodsem(st_1, val_0, m, invmo, vals_1, tys, ty, tds) = st_2 \times val_2\end{array}}{(v \times st)[\![e.m(e_1, \ldots, e_n)]\!](v_1 \times st_2 \times coerce(val_2, ty))}$$

The same restrictions apply to *methodsem* as for instance methods. This completes the semantics of expressions. The semantics of statements follows.

3.3 Semantics of Statements

As described in chapter 2 we have 16 'standard' Java statements (described in chapter 14 JLS), and introduce 7 new statements that will be used in the calculus. `static` is used for class initialization,

and has been also used as an auxiliary construct in the semantics of expressions (for a first active use), and the other are used to catch various kinds of jumps:

- `target` – catches return and break
- `targetexpr` – catches return (with an expression)
- `catches` – catches throw
- `finally` – catches all jumps
- `endstatic`(c) – catches exceptions

These statements are not always skipped if the mode in the store is not normal, i.e. they may catch jumps. The standard Java statements are skipped. The other additional statement

- `endfinally`

cannot catch jumps, but is used to re-raise the mode at the end of a `finally` block. The additional statements are described after the standard statements. Since some statements can catch jumps, a generic jump-rule as for expressions (see section 3.2.2) is not possible. This means there are sometimes rules that state explicitly that a statement is skipped in case the mode is not normal.

We begin with the semantics for statement lists. This is identical to expression lists.

3.3.1 Statement lists

A list of statements (not a block!) is executed from left to right:

$$\frac{}{(v \times st)[\![]\!](v \times st)} \qquad \frac{(v \times st)[\![\alpha]\!](v_0 \times st_0) \quad (v_0 \times st_0)[\![stms]\!](v_1 \times st_1)}{(v \times st)[\![\alpha + stms]\!](v_1 \times st_1)}$$

α is one statement, *stms* a list of statements. The rule for the empty expression list, $(v \times st)[\![]\!](v \times st \times [])$ looks identical, but returns an empty list of values. This is not a notational simplification, but a result of overloading of the relation symbol.

3.3.2 Block

A block (JLS 14.2) $\{\alpha_1 \ldots \alpha_n\}$ is evaluated (or executed) from left to right. α_i may be a local variable declaration that is valid (and visible) until the end of the block is reached. This means that all local variables are restored to their old (original) values, similar to the formal parameters of methods. This happens even if a jump occurs inside the block.

$$\frac{st[mode] \neq normal}{(v \times st)[\![\{\alpha_1 \ldots \alpha_n\}]\!](v \times st)}$$

The first rule handles the case that the initial mode is not normal. In this case the block is skipped.

$$\frac{st[mode] = normal \quad (v \times st)[\![\alpha_1 \ldots \alpha_n]\!](v_0 \times st_0)}{(v \times st)[\![\{\alpha_1 \ldots \alpha_n\}]\!](restore(v_0, locvars(\alpha_1 \ldots \alpha_n), v) \times st_0)}$$

If the mode is normal the statements of the block are executed. At the end the local variables of the block are restored to their original value. $locvars(stms)$ returns all variables from local variable declarations (on top level) from *stms*. The function $restore(v_0, vars, v)$ returns v_0, but the values of *vars* are set to the value of *vars* in v, $restore(v_0, vars, v) = v_0[vars, v[vars]]$.

One effect of this definition is that α and `{`α`}` can have different semantics. The semantics differs if α is a local variable declaration or a statement that may catch a jump (see chapter 3.3). However, for a type correct standard Java program the semantics is identical.

In Java, hiding the name of a local variable is not permitted (JLS 14.4.2). Furthermore, every local variable must be declared before it can be used. (Otherwise the name would be interpreted

as a field name.) Since we work with an annotated syntax tree it is possible to write `{ x = 3; int x = 4; }` where `x` is in both cases a local variable. The semantics rule restores the value of `x` to its value before the block was entered, not to its value before the local variable declaration occurred.

3.3.3 Local Variable Declaration

JLS 14.4. Every local variable must have an explicit initialization (the default value for the type). Since the end of the surrounding block restores the old value of the variable we can simply set the variable to its new value.

$$\frac{(v \times st)[\![x = e]\!](v_0 \times st_0 \times val_0)}{(v \times st)[\![ty\ x = e;]\!](v_0 \times st_0)}$$

The semantic of a local variable declaration is reduced to the semantic of a local variable assignment. If the initial mode $st[mode]$ is not normal the assignment is skipped. This means the declaration is evaluated only if no jump occurs. The result of the assignment is discarded, but the assignment modifies the variable mapping.

3.3.4 Expression Statements

The expression e of an expression statement e; (JLS 14.8) is evaluated (if the mode is normal) for its side effect; the computed value is discarded.

$$\frac{(v \times st)[\![e]\!](v_0 \times st_0 \times val_0)}{(v \times st)[\![e;]\!](v_0 \times st_0)}$$

In Java, only certain expressions (those that can have a side effect) are allowed in an expression statement. The reason is to avoid programming errors. We allow all expressions.

3.3.5 The if Statement

JLS 14.9. We have three rules:

$$\frac{(v \times st)[\![e]\!](v_0 \times st_0 \times val_0) \quad st_0[mode] \neq normal}{(v \times st)[\![\textbf{if}\ (e)\ \alpha_1\ \textbf{else}\ \alpha_2]\!](v_0 \times st_0)}$$

If the mode after evaluation of the test is not normal (either because the mode was initially not normal, or because evaluation of e raised an exception) nothing more happens.

$$\frac{\begin{array}{c}(v \times st)[\![e]\!](v_0 \times st_0 \times val_0) \quad st_0[mode] = normal \wedge val_0.boolval = \text{true} \\ (v_0 \times st_0)[\![\alpha_1]\!](v_1 \times st_1)\end{array}}{(v \times st)[\![\textbf{if}\ (e)\ \alpha_1\ \textbf{else}\ \alpha_2]\!](v_1 \times st_1)}$$

$.boolval$ selects a boolean value from a *javavalue* (see chapter 3.1.1). If the result is true the `then` part is evaluated, otherwise the `else` part. An `if` statement must have an `else` part.

$$\frac{\begin{array}{c}(v \times st)[\![e]\!](v_0 \times st_0 \times val_0) \quad st_0[mode] = normal \wedge val_0.boolval = \text{false} \\ (v_0 \times st_0)[\![\alpha_2]\!](v_1 \times st_1)\end{array}}{(v \times st)[\![\textbf{if}\ (e)\ \alpha_1\ \textbf{else}\ \alpha_2]\!](v_1 \times st_1)}$$

3.3.6 Labeled Statements

Labeled statements (JLS 14.7) with the same label may not be nested, but in different blocks the same labels can be used. A labeled statement with label l catches the jump $st[mode] = break(l)$ and ends normally. Otherwise nothing happens.

$$\frac{st[mode] \neq normal}{(v \times st)[\![l : \alpha]\!](v \times st)}$$

If the initial mode is not normal the statement is skipped.

$$\frac{st[mode] = normal \quad (v \times st)[\![\alpha]\!](v_0 \times st_0) \quad st_0[mode] = break(l)}{(v \times st)[\![l : \alpha]\!](v_0 \times st[mode, normal])}$$

If α ends with a break with the label the mode is set back to normal.

$$\frac{st[mode] = normal \quad (v \times st)[\![\alpha]\!](v_0 \times st_0) \quad st_0[mode] \neq break(l)}{(v \times st)[\![l : \alpha]\!](v_0 \times st_0)}$$

Otherwise nothing happens.

3.3.7 The while Statement

The `while` statement (JLS 14.11) in this work has no label. See chapter 2.5 for a discussion. If the test does not complete normally, or evaluates to false (or if the initial mode was not normal) the loop is finished.

$$\frac{(v \times st)[\![e]\!](v_0 \times st_0 \times val_0) \quad st_0[mode] \neq normal \vee val_0.boolval = \text{false}}{(v \times st)[\![\textbf{while } (e) \textbf{ do } \alpha]\!](v_0 \times st_0)}$$

Otherwise the body is evaluated, then again the loop.

$$\frac{\begin{array}{c}(v \times st)[\![e]\!](v_0 \times st_0 \times val_0) \quad st_0[mode] = normal \wedge val_0.boolval = \text{true} \\ (v_0 \times st_0)[\![\alpha]\!](v_1 \times st_1) \\ (v_1 \times st_1)[\![\textbf{while } (e) \textbf{ do } \alpha]\!](v_2 \times st_2)\end{array}}{(v \times st)[\![\textbf{while } (e) \textbf{ do } \alpha]\!](v_2 \times st_2)}$$

If the body does not complete normally the loop is also finished. The inductive definition (more precisely the smallest-relation part of the definition) ensures that a non-terminating loop has no semantics. In this case the relation for the statement is empty, i.e. there are no v_1, st_1 such that $(v \times st)[\![\textbf{while } (e) \textbf{ do } \alpha]\!](v_1 \times st_1)$.

3.3.8 The do Statement

The `do` statement (JLS 14.12) is similar to the `while` statement. Three rules are needed because the case that the initial mode is not normal must be defined explicitly.

$$\frac{st[mode] \neq normal}{(v \times st)[\![\textbf{do } \alpha \textbf{ while } (e);]\!](v \times st)}$$

$$\frac{\begin{array}{c}st[mode] = normal \quad (v \times st)[\![\alpha]\!](v_0 \times st_0) \quad (v_0 \times st_0)[\![e]\!](v_1 \times st_1 \times val_1) \\ st_1[mode] \neq normal \vee val_1.boolval = \text{false}\end{array}}{(v \times st)[\![\textbf{do } \alpha \textbf{ while } (e);]\!](v_1 \times st_1)}$$

$$\frac{\begin{array}{c}st[mode] = normal \quad (v \times st)[\![\alpha]\!](v_0 \times st_0) \quad (v_0 \times st_0)[\![e]\!](v_1 \times st_1 \times val_1) \\ st_1[mode] = normal \wedge val_1.boolval = \text{true} \\ v_1 \times st_1[\![\textbf{do } \alpha \textbf{ while } (e);]\!](v_2 \times st_2)\end{array}}{(v \times st)[\![\textbf{do } \alpha \textbf{ while } (e);]\!](v_2 \times st_2)}$$

It is obvious (and easy to prove) that **do** α **while** (e) is exactly equivalent to the two statements α **while** (e) **do** α. (But not to a block containing α and the `while` loop as discussed in chapter 3.3.2.)

3.3.9 The for Statement

As explained in chapter 2 we assume that the compiler extracted the initialization of the `for` loop (JLS 14.13). This means a `for` loop has only a termination test e and updates $e_1, \ldots, e_n$.

$$\frac{(v \times st)[\![e]\!](v_0 \times st_0 \times val_0) \quad st_0[mode] \neq normal \vee val_0.boolval = \text{false}}{(v \times st)[\![\textbf{for}(e; e_1, \ldots, e_n)\ \alpha]\!](v_0 \times st_0)}$$

If the mode after evaluation of the test is not normal, or the test evaluated to **false** the loop is terminated.

$$\frac{\begin{array}{c}(v \times st)[\![e]\!](v_0 \times st_0 \times val_0) \quad st_0[mode] = normal \wedge val_0.boolval = \text{true} \\ (v_0 \times st_0)[\![\{\alpha\ e_1; \ldots e_n;\}]\!](v_1 \times st_1) \\ (v_1 \times st_1)[\![\textbf{for}(e; e_1, \ldots, e_n)\ \alpha]\!](v_2 \times st_2)\end{array}}{(v \times st)[\![\textbf{for}(e; e_1, \ldots, e_n)\ \alpha]\!](v_2 \times st_2)}$$

If the test is true and ends normally the body, the updates, and again the `for` loop is executed. If one of these throws an exception the loop will be terminated.

A `for` loop with initialization behaves as follows:

$$\frac{(v \times st)[\![\{ForInit\ \textbf{for}(e; e_1, \ldots, e_n)\}]\!](v_0 \times st_0)}{(v \times st)[\![\textbf{for}(ForInit; e; e_1, \ldots, e_n)\ \alpha]\!](v_0 \times st_0)}$$

Compare this to JLS 14.13.1: "If the *ForInit* code is a local variable declaration, it is executed as if it were a local variable declaration statement appearing in a block." This obvious division of a `for` statement in an initialization part and an iteration part further justifies the requirement that the compiler transforms a `for` with initialization into one without an initialization.

3.3.10 The switch Statement

A `switch` statement (JLS 14.10)

```
switch (e) {
    c11, ..., c1m1 : α1
    ⋮
    cn1, ..., cnm1 : αn
}
```

consists of an expression e and some switch statements $\alpha_1 \ldots \alpha_n$ preceded by switch labels c_i. In this work the body of a switch is simply a statement that should be a block. The cases consist of *switchlabels*, a list of terms. The type of the expressions e and the switch labels must be of type `int` or of a type that can be converted `int` by primitive widening conversion (i.e. `byte`, or `short`, `char` is not supported in this work).

In Java, the switch labels must be compile-time constant expressions that have mutually different values. Execution of the statements "falls through labels", i.e. if $e = c_{11}$ first α_1 is executed, then $\alpha_2, \ldots, \alpha_n$. An empty label list denotes the `default` case that is executed when e differs from all labels. The `default` case may appear anywhere in the switch block and also "falls through labels". A missing `default` case is like a `default` with an empty body at the end of the switch block.

$$\frac{(v \times st)[\![e]\!](v_0 \times st_0 \times val_0) \quad st_0[mode] \neq normal}{(v \times st)[\![\texttt{switch}(e)\ \alpha]\!](v_0 \times st_0)}$$

$$\frac{\begin{array}{c}(v \times st)[\![e]\!](v_0 \times st_0 \times val_0) \quad st_0[mode] = normal \\ (v_0 \times st_0)[\![matchingcase(val_0, v_0, \alpha)]\!](v_1 \times st_1)\end{array}}{(v \times st)[\![\texttt{switch}(e)\ \alpha]\!](v_1 \times st_1)}$$

$matchingcase(v_0, val_0, \alpha)$ selects the correct case for e's value val_0, or the default case, or returns an empty block. Every case (consisting of a list of terms) is evaluated with the variable mapping v_0 and compared to val_0 after all values have been converted to integers. If one case matches the remaining statements are returned. They still may contain switch labels which are simply ignored by the following rule:

$$\frac{}{(v \times st)[\![\texttt{case } c_1, \ldots, c_n]\!](v \times st)}$$

For duplicate labels the first matching label is returned, the later labels are ignored.

3.3.11 The break Statement

A break (JLS 14.14) with label l sets the mode to $break(l)$, but only if the current mode is normal.

$$\frac{st[mode] \neq normal}{(v \times st)[\![\texttt{break}(l)]\!](v \times st)}$$

$$\frac{st[mode] = normal}{(v \times st)[\![\texttt{break}(l)]\!](v \times st[mode, break(l)])}$$

A labeled statement (chapter 3.3.6), a `finally` clause and some of the new Java statements can catch breaks.

3.3.12 Empty return Statement

JLS 14.16. We distinguish between a `return` statement with and without an expression.

$$\frac{st[mode] \neq normal}{(v \times st)[\![\texttt{return};]\!](v \times st)}$$

An empty return statement sets the mode to return if the current mode is normal.

$$\frac{st[mode] = normal}{(v \times st)[\![\texttt{return};]\!](v \times st[mode, return])}$$

3.3.13 return Statement with Expression

A return statement with an expression evaluates the expression and sets the mode to return unless the current mode is already a jump. The result value of e is stored in the mode.

$$\frac{(v \times st)[\![e]\!](v_0 \times st_0 \times val_0) \quad st_0[mode] \neq normal}{(v \times st)[\![\texttt{return } e;]\!](v_0 \times st_0)}$$

$$\frac{(v \times st)[\![e]\!](v_0 \times st_0 \times val_0) \quad st_0[mode] = normal}{(v \times st)[\![\texttt{return } e;]\!](v_0 \times st[mode, return(val_0, ty)])}$$

The mode is set to *return* containing a Java value and the static type ty of e. This type is not used in the semantics, but useful for type safety. A *target* statement with expression (chapter 3.3.20) will catch a return and assign the returned value to a variable.

3.3.14 The throw Statement

JLS 14.17. If the evaluation of e yields $val_0 = null$, a `NullPointerException` is thrown (this case was missing in JLS (first edition), but is corrected in the second edition). Otherwise a throw sets the mode (that includes the reference of the thrown expression).

$$\frac{(v \times st)[\![e]\!](v_0 \times st_0 \times val_0) \quad st_0[mode] \neq normal}{(v \times st)[\![\texttt{throw } e;]\!](v_0 \times st_0)}$$

$$\frac{(v \times st)[\![e]\!](v_0 \times st_0 \times val_0) \quad st_0[mode] = normal \wedge val_0 = null \quad (v_0 \times st_0)[\![\texttt{throw new NullPointerException();}]\!](v_1 \times st_1)}{(v \times st)[\![\texttt{throw } e;]\!](v_1 \times st_1)}$$

$$\frac{(v \times st)[\![e]\!](v_0 \times st_0 \times val_0) \quad st_0[mode] = normal \wedge val_0 \neq null}{(v \times st)[\![\texttt{throw } e;]\!](v_0 \times st_0[mode, throw(val_0.refval, ty)])}$$

The mode is set to *throw* containing a reference $val_0.refval$ to the thrown object and the static type ty of the expression e. The type is not used in the semantics, but useful for type safety.

3.3.15 The try Statement

JLS 14.19 distinguishes between a `try` statement with a `finally` block and one without. In this work a `finally` block is always included. A `finally` block with an empty block has exactly the same semantics as no `finally` block (it does nothing). The first rule states that the complete `try` statement is skipped if the mode is not *normal*.

$$\frac{st[mode] \neq normal}{(v \times st)[\![\texttt{try } \alpha \ catches \ \texttt{finally } \alpha_2]\!](v \times st)}$$

catches is the list of catch clauses. For sake of simplicity, a `catch` clause is a normal statement. However, it may occur only in this list.

If the `try` block does not end with a throw or with a throw that has no handler, the `finally` block is executed. We use the additional statement `finally` to capture the behavior of the `finally` block:

$$\frac{\begin{array}{c} st[mode] = normal \quad (v \times st)[\![\alpha]\!](v_0 \times st_0) \\ \neg\ is_throw_mode(st_0[mode]) \vee \neg\ is_caught(refval(st_0[mode].refval), catches, st_0, tds) \\ (v_0 \times st_0)[\![\texttt{finally}(\beta)]\!](v_1 \times st_1) \end{array}}{(v \times st)[\![\texttt{try } \alpha \ catches \ \texttt{finally } \beta]\!](v_1 \times st_1)}$$

The predicate *is_caught* is true if the type of the thrown reference $st_0[mode].refval$ is a subclass of (or equal to) one of the classes of the catch clauses. *asgcomp* is used for this check (*asgcomp* is also used for reference casts and array assignments, see chapter 3.2.5 for the definition). Note that by applying the definition of the `finally` rules the additional construct can be eliminated from these rules. This means that using `finally` is just an abbreviation.

If the `try` block ends with a throw that has a handler, the corresponding catch block is executed. Afterwards the `finally` block is executed. The final mode depends on whether the finally block ended normal. α_c is the body of the correct catch clause for the thrown reference ($\texttt{catch } c(x)\alpha_c = getCatcher(refval(st_0[mode].refval), catches, st_0)$). The variable x serves as a local variable for the thrown reference and must be reset at the end of the statement (as local variables in blocks).

$$\frac{\begin{array}{c} st[mode] = normal \quad (v \times st)[\![\alpha]\!](v_0 \times st_0) \quad is_throw_mode(st_0[mode]) \\ is_caught(refval(st_0[mode].refval), catches, st_0, tds) \\ (v_0[x, refval(st_0[mode].refval)] \times st_0[mode, normal])[\![\alpha_c]\!](v_1 \times st_1) \\ (v_1[x, v_0(x)] \times st_1)[\![\texttt{finally}(\beta)]\!](v_2 \times st_2) \end{array}}{(v \times st)[\![\texttt{try } \alpha \ catches \ \texttt{finally } \beta]\!](v_2 \times st_2)}$$

Before the `catch` clause is executed the thrown reference $st_0[mode].refval$ is bound to the variable x named in the `catch` clause, $v_0[x, refval(st_0[mode].refval)]$, and the mode is set back to normal, $st_0[mode, normal]$. Afterwards, x is restored to its original value, $v_1[x, v_0(x)]$. The correct catch clause is the first clause with an assignment compatible type to the throw type. *getCatcher* also uses *asgcomp*. The first matching catch clause is chosen.

This finishes the semantics of the standard Java statements. It remains to define the additional statements.

3.3.16 The catches Statement

A list of `catch` clauses catches exceptions (or errors). This statement will be used in the proof rule for the `try` statement. Its behavior is exactly that of the `catch` clauses in a `try` statement (see chapter 3.3.15). It is necessary to keep all clauses in one list, because otherwise exceptions occurring in the body of a clause could be caught by one of the following clauses.

$$\frac{is_throw_mode(st[mode]) \wedge is_caught(refval(st[mode].refval), \alpha_1 \ldots \alpha_n, st, tds) \quad (v[x, refval(st[mode].refval)] \times st[mode, normal])[\![\alpha_c]\!](v_0 \times st_0)}{(v \times st)[\![\texttt{catches}(\alpha_1 \ldots \alpha_n)]\!](v_0[x, v(x)] \times st_0)}$$

`catch` $(c\ x)\{\alpha_c\}$ is the correct catch clause from the list *catches* for the thrown reference $st[mode].ref$.

Otherwise the statement is ignored.

$$\frac{\neg\ is_throw_mode(st[mode]) \vee \neg\ is_caught(refval(st[mode].refval), \alpha_1 \ldots \alpha_n, st, tds)}{(v \times st)[\![\texttt{catches}(\alpha_1 \ldots \alpha_n)]\!](v \times st)}$$

These two definitions occurred in the same manner in the semantics of the `try` statement.

3.3.17 The finally Statement

A `finally` block catches all jumps, executes its statement and raises the old mode unless the statement completes abruptly. If the initial mode is normal the statement is executed and nothing more happens.

$$\frac{st[mode] = normal \quad (v \times st)[\![\alpha]\!](v_0 \times st_0)}{(v \times st)[\![\texttt{finally}\ \alpha]\!](v_0 \times st_0)}$$

The second rule handles the case that the finally block completes normally.

$$\frac{st[mode] \neq normal \quad (v \times st[mode, normal])[\![\alpha]\!](v_0 \times st_0) \quad st_0 = normal}{(v \times st)[\![\texttt{finally}\ \alpha]\!](v_0 \times st_0[mode, st[mode]])}$$

The mode is re-raised, $st_0[mode, st[mode]]$. Otherwise, the new mode is kept.

$$\frac{st[mode] \neq normal \quad (v \times st[mode, normal])[\![\alpha]\!](v_0 \times st_0) \quad st_0 \neq normal}{(v \times st)[\![\texttt{finally}\ \alpha]\!](v_0 \times st_0)}$$

3.3.18 The endfinally Statement

`endfinally`(t) raises a jump with mode t if the initial mode is *normal*. This statement is used in the calculus to re-raise the mode at the end of a finally block. t is a term that is evaluated under the current variable mapping.

$$\frac{st[mode] = normal}{(v \times st)[\![\texttt{endfinally}(t)]\!](v \times st[mode, eval(v, t)])}$$

$$\frac{st[mode] \neq normal}{(v \times st)[\![\texttt{endfinally}(t)]\!](v \times st)}$$

t is usually something like $st_0[mode]$ that references the mode of another store.

3.3.19 The empty target Statement

`target(mo)` is intended to catch jumps caused by `return` (without value) or `break`. If the mode is return, any `return` is caught; if the mode is break, a break with the same label is caught.

$$\frac{\begin{array}{c}is_return_mode(st[mode]) \wedge is_return_mode(mo)\\ \vee\ is_break_mode(st[mode]) \wedge st[mode] = mo\end{array}}{(v \times st)[\![\texttt{target}(mo)]\!](v \times st[mode, normal])}$$

In these cases the mode is set back to normal. Otherwise the statement is ignored.

$$\frac{\begin{array}{c}\neg\ (is_return_mode(st[mode]) \wedge is_return_mode(mo)\\ \vee\ is_break_mode(st[mode]) \wedge st[mode] = mo)\end{array}}{(v \times st)[\![\texttt{target}(mo)]\!](v \times st)}$$

3.3.20 The target Statement with Expression

`targetexpr`(x,ty) catches returns and sets x to the returned value $st[mode].val$. It will be used to capture the result of a non-void method call (the method body must return a value with a `return` statement that returns a value).

$$\frac{is_return_mode(st[mode])}{(v \times st)[\![\texttt{targetexpr}(x, ty)]\!](v[x, coerce(st[mode].val, ty)] \times st[mode, normal])}$$

The mode is set back to normal, and the returned value is coerced to the expected result type. This is to ensure primitive type correctness (see chapter 5.1). Otherwise the statement is simply discarded.

$$\frac{\neg\ is_return_mode(st[mode])}{(v \times st)[\![\texttt{targetexpr}(x, ty)]\!](v \times st)}$$

3.3.21 The static Statement

`static`(c) is used to handle the static initialization of the class or interface c. It creates the static fields of the class, sets the class state to *initdone*, handles the initialization of the super classes if necessary, executes the static initializer, and adds a catcher `endstatic` to handle exceptions.

If the mode is not normal or if the initialization state is not undone an error is thrown.

$$\frac{\begin{array}{c}\neg\ (st[mode] = normal \wedge initundoneP(c, st))\\ (v \times st)[\![\texttt{throw new NoClassDefFoundError();}]\!](v_0 \times st_0)\end{array}}{(v \times st)[\![\texttt{static}(c)]\!](v_0 \times st_0)}$$

If the mode is not normal the `throw` statement is skipped. If the initialization state is not *initundoneP* then the class is either already initialized or in an erroneous state. In the latter case a `NoClassDefFoundError` must be thrown (JLS 12.4.2 item 5). The previous case (the class is already initialized) cannot occur, because the `static` statement is introduced only if $\neg\ initdoneP(c, st)$ is true.

Otherwise the super class is initialized if necessary. This is done by `static` again, but for the super class:

$$\frac{\begin{array}{c}st[mode] = normal \wedge initundoneP(c, st)\\ \neg\ (c = java.lang.Object \vee initdoneP(superClass(c, tds), st))\\ (v \times addclass(c, statfields(c, tds), st))[\![\texttt{static}(superClass(c, tds))]\!](v_0 \times st_0)\\ (v_0 \times st_0)[\![statinit(c, tds)]\!](v_1 \times st_1)\\ (v_1 \times st_1)[\![\texttt{endstatic}(c)]\!](v_2 \times st_2)\end{array}}{(v \times st)[\![\texttt{static}(c)]\!](v_2 \times st_2)}$$

$superClass(c, tds)$ is the super class of c (if c is an interface java.lang.Object is returned which is initialized), *statfields* selects the static fields of a class, *statinit* the static initializer(s) in their textual order. *addclass* adds a class with its static fields to the store and sets its initialization state do *done* (see chapter 3.1.3). Since the initialization state is set before the super class is initialized or the static initializer executed, cyclic or recursive initialization cannot happen. An initialization state *in progress* is not needed.

Otherwise only the static initializer is executed.

$$\frac{\begin{array}{c} st[mode] = normal \wedge initundoneP(c, st) \\ c = java.lang.Object \vee initdoneP(superClass(c, tds), st) \\ (v \times addclass(c, statfields(c, tds), st))[\![statinit(c, tds)]\!](v_1 \times st_1) \\ (v_1 \times st_1)[\![\texttt{endstatic}(c)]\!](v_2 \times st_2) \end{array}}{(v \times st)[\![\texttt{static}(c)]\!](v_2 \times st_2)}$$

`endstatic` is described next.

3.3.22 The endstatic Statement

`endstatic`(c) catches exceptions. If during static initialization an exception or error occurs the class object is marked 'erroneous', and an exception is transformed into an *ExceptionInInitializerError*. Otherwise the initialization state is set to *done* (see JLS 12.4.2, items 9–11).

$$\frac{\neg\ is_throw_mode(st[mode])}{(v \times st)[\![endstatic(c)]\!](v \times st[jvmref - mkfs(c, void, initstate), initval(done)])}$$

The initialization state is recorded for every class c in a special field named 'initstate'.

$$\frac{\begin{array}{c} is_throw_mode(st[mode]) \wedge asgcomp(refval(st[mode].refval), Exception, st, tds) \\ (v \times st[mode, normal][jvmref - mkfs(c, void, initstate), initval(error)]) \\ [\![\texttt{throw new ExceptionInInitializerError();}]\!](v_0 \times st_0) \end{array}}{(v \times st)[\![\texttt{endstatic}(c)]\!](v_0 \times st_0)}$$

$asgcomp(st[mode].refval, Exception, st)$ is true if the type of the thrown reference $st[mode].refval$ is a subclass of (or equal to) java.lang.Exception, see chapter 3.2.5.

$$\frac{is_throw_mode(st[mode]) \wedge \neg\ asgcomp(refval(st[mode].ref), Exception, st, tds)}{(v \times st)[\![\texttt{endstatic}(c)]\!](v \times st[jvmref - mkfs(c, void, initstate), initval(error)])}$$

If an error (or more precisely: something else than an exception) was thrown the class is marked 'erroneous' and the mode remains unchanged. Java Card has no `ExceptionInInitializerError`.

This finished the Java semantics.

3.4 Proof Technique

This section describes the proof technique used in KIV to prove properties about the semantics. This is necessary because KIV does not support inductive definitions, as for example Isabelle [Pau94b] [NPW03]. At the end it includes the property that the semantics is indeed deterministic.

3.4.1 The 'smallest' axiom

The semantics of a Java statement is the smallest relation *sem* closed under the reduction rules (likewise for expressions, expression lists, and statement lists). The rules from the previous sections define the positive part of this relation. The 'smallest' part can be expressed by a higher-order axiom. From this higher-order axiom an induction scheme can be derived. The higher-axiom states that

Any four predicates that fulfill the reduction rules are true for all values where the four sem predicates are true.

The four *sem* predicates have the following signature:

$$\begin{array}{l}\text{sem} : \text{statestore} \times \text{jstatement} \times \text{typedecls} \times \text{statestore};\\ \text{sem} : \text{statestore} \times \text{jstmlist} \times \text{typedecls} \times \text{statestore};\\ \text{sem} : \text{statestore} \times \text{jexpr} \times \text{typedecls} \times \text{statestorevalue};\\ \text{sem} : \text{statestore} \times \text{jexprlist} \times \text{typedecls} \times \text{statestorevaluelist};\end{array}$$

Now we define four higher-order variables with the same arguments:

$$\begin{array}{l}\text{Pstm} : \text{statestore} \times \text{jstatement} \times \text{typedecls} \times \text{statestore};\\ \text{Pstms} : \text{statestore} \times \text{jstmlist} \times \text{typedecls} \times \text{statestore};\\ \text{Pexpr} : \text{statestore} \times \text{jexpr} \times \text{typedecls} \times \text{statestorevalue};\\ \text{Pexprs} : \text{statestore} \times \text{jexprlist} \times \text{typedecls} \times \text{statestorevaluelist};\end{array}$$

Then the higher-order axiom has the following structure:

$$\begin{array}{ll} & Cl_\forall(r_1(P)) \wedge \ \ldots \ \wedge Cl_\forall(r_n(P))\\ \rightarrow & \quad (\forall\, v, st, \alpha, tds, v_0, st_0. \quad sem(v \times st, \alpha, tds, v_0 \times st_0)\\ & \qquad\qquad \rightarrow \text{Pstm}(v \times st, \alpha, tds, v_0 \times st_0))\\ & \wedge\ (\forall\, v, st, stms, tds, v_0, st_0. \quad sem(v \times st, stms, tds, v_0 \times st_0)\\ & \qquad\qquad \rightarrow \text{Pstms}(v \times st, stms, tds, v_0 \times st_0))\\ & \wedge\ (\forall\, v, st, e, tds, v_0, st_0, val_0. \quad sem(v \times st, e, tds, v_0 \times st_0 \times val_0)\\ & \qquad\qquad \rightarrow \text{Pexpr}(v \times st, e, tds, v_0 \times st_0 \times val_0))\\ & \wedge\ (\forall\, v, st, es, tds, v_0, st_0, val_0. \quad sem(v \times st, es, tds, v_0 \times st_0 \times vals_0)\\ & \qquad\qquad \rightarrow \text{Pexprs}(v \times st, es, tds, v_0 \times st_0 \times vals_0))\end{array}$$

The preconditions $Cl_\forall(r_1(P)) \wedge \ \ldots \ \wedge Cl_\forall(r_n(P))$ contain the rules: $r_1, \ldots, r_n$ are the reduction rules, P is one of *Pstm*, *Pstms*, *Pexpr*, *Pexprs*, and $r_i(P)$ is a reduction rule where every occurrence of a *sem* predicate is replaced by the corresponding P variable of the same signature. $Cl_\forall(r_i(P))$ is the universal closure ranging over all first-order variables of the rule (this means P is not included). Since this axiom contains the conjunction of all rules it is very large for the Java semantics: about 1000 lines of text. However, it is easy to generate automatically from the rules. From the reduction rules and the 'smallest' axiom induction schemes can be derived. In KIV we choose another approach: The rules are transformed into a program, and proofs are done by induction on the call depth of procedure calls. This requires some explanation.

3.4.2 Dynamic Logic in KIV

The logic of KIV is a higher order logic combined with a dynamic logic (DL, see [Har79] [HKT00] [HRS89]). Dynamic logic extends first-order (or higher-order) logic by two modal operators, $[\ .\ ]\ .$, and $\langle\ .\ \rangle\ .$. The operators contain programs of an imperative abstract programming language (in chapter 4 we will use these operators for Java programs), and as a second argument again a formula of dynamic logic: In $[\alpha]\varphi$, α is a program, and φ a DL-formula. The semantics of $[\alpha]\varphi$ is: If α terminates then afterwards φ holds, and for $\langle\alpha\rangle\ \varphi$: α terminates and afterwards φ holds. (The semantics of these operators will be defined precisely in chapter 4 for the Java calculus. Here, the informal notion is sufficient.) The programs are abstract programs in the sense that they do not have a fixed set of basic data types and operations, but can use arbitrary abstract data types, i.e. algebraic specifications. In an assignment `x := t`, `x` is a logical variable of an arbitrary sort, and `t` an arbitrary term of the same sort. The tests of a conditional and a while loop are arbitrary formulas. The statements of the language include procedure calls, while loops, ifs, assignments, local variable declarations, and a random choice statement. One application of programs and DL formulas is as an alternative formulation for algebraic predicates or functions. For example, a recursive function that computes the sum of the first n natural numbers, can be specified as:

sum : nat → nat

sum(0) = 0

sum(n + 1) = sum(n) + (n + 1)

Alternatively, the sum can be computed by a recursive procedure:

```
procedure SUM(n; var res)
begin
   if n = 0 then res := 0
   else
      begin
         SUM(n − 1; res);
         res := res + n
      end
end
```

The syntax is PASCAL-like: `n` is a value parameter, the result is computed in the reference parameter `res`. (A procedure can have several value and result parameters.) Grouping is done by `begin` and `end`. The syntax does not allow pattern matching: The value parameter must be a variable, and the recursive call uses $n - 1$ (in general, selectors have to be applied). It is possible to prove

sum(n) = m ↔ ⟨SUM(n; m0)⟩ m0 = m

$m0$ holds the result of the SUM computation. This result can then be compared with m in the equation $m0 = m$.

3.4.3 Semantics as a procedure

This technique can be used for the four Java semantics predicates. The idea is to have one procedure for every semantics predicate that is equivalent to this predicate. This means, the following equivalences will hold:

$$\begin{aligned}
&sem(v \times st, \alpha, tds, v_0 \times st_0) \\
\leftrightarrow\ &\langle \text{semstm}(v, st, \alpha, tds; v_1, st_1)\rangle\ (v_1 = v_0 \wedge st_1 = st_0) \\
&sem(v \times st, stms, tds, v_0 \times st_0) \\
\leftrightarrow\ &\langle \text{semstms}(v, st, stms, tds; v_1, st_1)\rangle\ (v_1 = v_0 \wedge st_1 = st_0) \\
&sem(v \times st, e, tds, v_0 \times st_0 \times val_0) \\
\leftrightarrow\ &\langle \text{semexpr}(v, st, e, tds; v_1, st_1, val_1)\rangle\ (v_1 = v_0 \wedge st_1 = st_0 \wedge val_1 = val_0) \\
&sem(v \times st, es, tds, v_0 \times st_0 \times vals_0) \\
\leftrightarrow\ &\langle \text{semexprs}(v, st, es, tds; v_1, st_1, vals_1)\rangle\ (v_1 = v_0 \wedge st_1 = st_0 \wedge vals_1 = vals_0)
\end{aligned}$$

(The predicate *sem* is overloaded four times; procedures in KIV cannot be overloaded.) The body of theses procedures is generated automatically from the reduction rules. This is fairly straightforward, except that selectors have to be used instead of constructors. The result is difficult to read, but that does not matter because there is no need to look at the procedure bodies. It is useful to use sub-procedures for every statement and expression, and the return statement with an expression may serve as an example.

The two rules in their mathematical notation are

$$\frac{(v \times st)[\![e]\!](v_0 \times st_0 \times val_0) \quad st_0[mode] \neq normal}{(v \times st)[\![\texttt{return}\ e;]\!](v_0 \times st_0)}$$

$$\frac{(v \times st)[\![e]\!](v_0 \times st_0 \times val_0) \quad st_0[mode] = normal}{(v \times st)[\![\texttt{return}\ e;]\!](v_0 \times st[mode, return(val_0, ty)])}$$

In KIV they are typed in as

returnexpr-ok :

sem(v × st, jexpr, tds, v0 × st0 × val0) ∧ st0[_mode] = normal
→ sem(v × st, jreturnexpr(jexpr), tds, v0 × st0[_mode, return(val0, jexpr.type)]);

returnexpr-ex :

sem(v × st, jexpr, tds, v0 × st0 × val0) ∧ st0[_mode] ≠ normal
→ sem(v × st, jreturnexpr(jexpr), tds, v0 × st0);

The generated procedure is

```
jreturnexpr(v, st, jstm, tds; var v4, st4)
  begin
     var v0 = v, st0 = st, jr0 = noval
     in begin
           semexpr(v, st, jstm .jexpr, tds; v0, st0, jr0);
           if st0[_mode] = normal
           then v4 := v0 |
                st4 := st0[_mode, return(jr0, jstm .jexpr .type)]
           else begin
                   if not st0[_mode] = normal
                   then skip
                   else abort ;
                   v4 := v0 | st4 := st0
                end
        end
  end;
```

The redundant tests and (unnecessary) local variables are due to the automatic generation routine. The complete implementation is more than 2000 lines long, but there is no reason to look at it.

The question is, of course, are the procedures equivalent to the predicates? Or, more precisely, are the four equivalences semstm-def, semstms-def, semexpr-def, semexprs-def true? If the generation routine is correct and performs all necessary checks (and fails if a check fails) the answer is yes. On the other hand, it is a nice verification exercise to prove the equivalence. (This has been done.)

Using the theorems it is easy to prove equivalences for the *sem* predicates for the Java statements and expressions, for example for the `return` statement:

sem(v × st, `jreturnexpr(jexpr)`, tds, v1 × st1)
↔ ∃ v0, st0, val0.
  sem(v × st, jexpr, tds, v0 × st0 × val0)
  ∧ v1 = v0
  ∧ (st0[_mode] ≠ normal ∧ st1 = st0
    ∨ st0[_mode] = normal ∧ st1 = st0[_mode, return(val0, jexpr .type)])

or in their mathematical notation:

$$
\begin{aligned}
&(v \times st)[\![\texttt{jreturnexpr(e)}]\!]\ (v_1 \times st_1)\\
\leftrightarrow\ &\exists\ v_0, st_0, val_0.\quad (v \times st)[\![\texttt{e}]\!]\ (v_0 \times st_0 \times val_0)\\
&\qquad\qquad \wedge\ v_1 = v_0\\
&\qquad\qquad \wedge\ (\quad st_0[mode] \neq normal \wedge st_1 = st_0\\
&\qquad\qquad\quad \vee\ st_0[mode] = normal \wedge st_1 = st_0[mode, return(val_0, ty)])
\end{aligned}
$$

Now it is possible to do proofs either with the semantics rules (or equivalences), or with the procedures. It depends on the theorems to prove what is more convenient. There are two differences. First, using the equivalences introduces existential quantifiers for the resulting variable mapping and store, and it can be very difficult to find the correct instances. The procedures, on

the other hand, compute the resulting state, so no instances must be guessed. Second, only proofs that require no induction can be done with the equivalences.

A structural induction rule (called *rule induction* in Isabelle) for the semantics has as many premises to prove as there are reduction rules. And there may be proofs where it is necessary (or at least convenient) to use a Noetherian induction scheme. This is automatically available if proofs are done using the procedures: KIV allows (Noetherian) induction on the depth of procedure calls. To do this, the DL programs have an additional special statement, bounded procedure calls. The notation is simply the procedure call followed by the bound (a natural number), $\text{semstm}(v, st, \alpha, tds; v_1, st_1) : m$. If $m = 0$ the statement does not terminate, $\neg \langle \text{semstm}(v, st, \alpha, tds; v_1, st_1) : m \rangle \varphi$. Otherwise the procedure call can be replaced by the procedure body (the usual unfolding of a procedure call), but all procedure calls inside the body (not only for the same procedure, but for other procedure calls as well) must be replaced by bounded procedure calls with the bound $m - 1$. So a procedure call terminates iff there is a bound m such that the bounded procedure call terminates:

$$\langle \text{semstm}(v, st, \alpha, tds; v_1, st_1) \rangle \varphi \leftrightarrow \exists\, m.\ \langle \text{semstm}(v, st, \alpha, tds; v_1, st_1) : m \rangle \varphi$$

(where m is a new variable). This construct is necessary to achieve a complete calculus for DL. Now, the procedure calls of the procedure *semstm* correspond to applications of the reduction rules: A depth of two procedure calls corresponds to the application of one reduction rule, one call to *semstm*, another call for the statement used, for example, `jreturnexpr` for a return statement with an expression. This means that induction on the procedure call depth corresponds to induction on the maximal depth of reduction rule applications. Structural induction is the special case where a property is known for a procedure call bound m and must be established for $m + 2$. So, using the procedures for proofs automatically gives us a very flexible induction scheme, which is indeed used several times.

To summarize, using the procedures has the following disadvantages:

1. Fewer simplification rules. Formulas containing DL programs cannot be used as simplification rules in KIV (just for technical reasons; there is no logical restriction). Therefore the proofs require either more user interaction, or another mechanism of KIV must be used to automatically apply lemmas, which is less efficient than simplification.

2. Larger proofs. Since formulas containing DL programs are not handled by the simplifier, and due to their operational character and restricted syntax, proofs for programs become considerably larger. For example, the case distinction what kind of Java statement is considered, is done by nested if-then-elses (there is no DL switch statement), and every DL statement is one rule application. Of course, these rule applications happen fully automatically.

However, the advantages outweigh the disadvantages:

1. As described above, we can use Noetherian induction.

2. The procedures compute a result. This means they are an interpreter for the semantics. Especially in the proofs of the calculus rules it such simpler to compute the result (the final variable mapping and store) than to 'guess' the result. The latter is necessary when the equivalences are used, because then formulas like $\exists\, v_0, st_0.sem(v \times st, \alpha, v_0 \times st_0) \wedge \ldots$ must be proved by instantiating the quantifier. Experience shows that this can be very difficult.

3. The operational character of the programs increases automation. For example, an `if` statement naturally leads to a case distinction, while it is very difficult to automate predicate logic case distinctions in a manner that does not introduce unwanted case distinctions, or case distinctions too early in the proof.

4. A structural induction rule has as many premises as there are reduction rules. The Noetherian induction allows better control when the different cases are expanded. For example, it is possible to formulate a lemma that is applicable for all expressions, or a lemma dealing with

an expression that has many reduction rules like the compound assignment or the method call.

To conclude: Dynamic Logic can be very useful as a proof technique for inductively defined relations and even for purely first-order proofs. A nice extension for KIV would be automatic support for switching between different representations.

3.4.4 The semantics is deterministic

Finally, we can state two theorems for the semantics definition: It is well defined, and deterministic:

3.4.1 Theorem (Well-definedness of the Semantics)

The semantics axioms are consistent. This means the semantics relations are well-defined.

Proof

All axioms except the 'smallest' axiom are Horn formulas. This implies

1. they are consistent (the relation that is always *true* is a valid interpretation for *sem*).
2. the 'smallest' axiom is consistent, and consistent with the other axioms (the smallest relation exists).

(This cannot be proved in KIV itself.)

3.4.2 Theorem (Alternative formulation of the Semantics)

The following theorems are valid.

semstm-def :

$$sem(v \times st, \alpha, tds, v_0 \times st_0) \leftrightarrow \langle \text{semstm}(v, st, \alpha, tds; v_1, st_1) \rangle\ (v_1 = v_0 \wedge st_1 = st_0)$$

semstms-def :

$$sem(v \times st, stms, tds, v_0 \times st_0) \leftrightarrow \langle \text{semstms}(v, st, stms, tds; v_1, st_1) \rangle\ (v_1 = v_0 \wedge st_1 = st_0)$$

semexpr-def :

$$sem(v \times st, e, tds, v_0 \times st_0 \times val_0) \leftrightarrow \langle \text{semexpr}(v, st, e, tds; v_1, st_1, val_1) \rangle\ (v_1 = v_0 \wedge st_1 = st_0 \wedge val_1 = val_0)$$

semexprs-def :

$$sem(v \times st, es, tds, v_0 \times st_0 \times vals_0) \leftrightarrow \langle \text{semexprs}(v, st, es, tds; v_1, st_1, vals_1) \rangle\ (v_1 = v_0 \wedge st_1 = st_0 \wedge vals_1 = vals_0)$$

Proof

The proofs have been done with KIV. The following two generalizations are used to prove the two implications of the equivalences:

```
(∀ v, st, jstm, tds, v0, st0.
        ⟨semstm(v, st, jstm, tds; v1, st1): m ⟩ (v1 = v0 ∧ st1 = st0)
      → sem(v × st, jstm, tds, v0 × st0))
∧ (∀ v, st, jstms, tds, v0, st0.
        ⟨semstms(v, st, jstms, tds; v1, st1): m ⟩ (v1 = v0 ∧ st1 = st0)
      → sem(v × st, jstms, tds, v0 × st0))
∧ (∀ v, st, jexpr, tds, v0, st0, val0.
        ⟨semexpr(v, st, jexpr, tds; v1, st1, val1): m ⟩
                              (v1 = v0 ∧ st1 = st0 ∧ val1 = val0)
```

$$
\begin{array}{l}
\quad \rightarrow \mathrm{sem}(\mathrm{v} \times \mathrm{st}, \mathrm{jexpr}, \mathrm{tds}, \mathrm{v0} \times (\mathrm{st0} \times \mathrm{val0}))) \\
\wedge\ (\forall\ \mathrm{v}, \mathrm{st}, \mathrm{jexprs}, \mathrm{tds}, \mathrm{v0}, \mathrm{st0}, \mathrm{vals0}. \\
\qquad \langle \mathrm{semexprs}(\mathrm{v}, \mathrm{st}, \mathrm{jexprs}, \mathrm{tds}; \mathrm{v1}, \mathrm{st1}, \mathrm{vals1}){:}\ \mathrm{m}\ \rangle \\
\qquad\qquad\qquad\qquad (\mathrm{v1} = \mathrm{v0} \wedge \mathrm{st1} = \mathrm{st0} \wedge \mathrm{vals1} = \mathrm{vals0}) \\
\qquad \rightarrow \mathrm{sem}(\mathrm{v} \times \mathrm{st}, \mathrm{jexprs}, \mathrm{tds}, \mathrm{v0} \times (\mathrm{st0} \times \mathrm{vals0})))
\end{array}
$$

This is proved by induction on m, applying the induction hypotheses for $m-2$, symbolic execution of the procedures, and application of every reduction rule exactly once. (The induction hypothesis is needed only for the immediate sub expressions or statements, because the procedures correspond exactly to the reduction rules.)
The other direction of the implication is proved with

$$
\begin{array}{l}
(\quad \mathrm{sem}(\mathrm{v} \times \mathrm{st}, \mathrm{jstm}, \mathrm{tds}, \mathrm{v0} \times \mathrm{st0}) \\
\quad \rightarrow \langle \mathrm{semstm}(\mathrm{v}, \mathrm{st}, \mathrm{jstm}, \mathrm{tds}; \mathrm{v1}, \mathrm{st1})\rangle\ (\mathrm{v1} = \mathrm{v0} \wedge \mathrm{st1} = \mathrm{st0})) \\
\wedge\ (\quad \mathrm{sem}(\mathrm{v} \times \mathrm{st}, \mathrm{jstms}, \mathrm{tds}, \mathrm{v0} \times \mathrm{st0}) \\
\quad \rightarrow \langle \mathrm{semstms}(\mathrm{v}, \mathrm{st}, \mathrm{jstms}, \mathrm{tds}; \mathrm{v1}, \mathrm{st1})\rangle\ (\mathrm{v1} = \mathrm{v0} \wedge \mathrm{st1} = \mathrm{st0})) \\
\wedge\ (\quad \mathrm{sem}(\mathrm{v} \times \mathrm{st}, \mathrm{jexpr}, \mathrm{tds}, \mathrm{v0} \times (\mathrm{st0} \times \mathrm{val0})) \\
\quad \rightarrow \langle \mathrm{semexpr}(\mathrm{v}, \mathrm{st}, \mathrm{jexpr}, \mathrm{tds}; \mathrm{v1}, \mathrm{st1}, \mathrm{val1})\rangle\ (\mathrm{v1} = \mathrm{v0} \wedge \mathrm{st1} = \mathrm{st0} \wedge \mathrm{val1} = \mathrm{val0})) \\
\wedge\ (\quad \mathrm{sem}(\mathrm{v} \times \mathrm{st}, \mathrm{jexprs}, \mathrm{tds}, \mathrm{v0} \times (\mathrm{st0} \times \mathrm{vals0})) \\
\quad \rightarrow \langle \mathrm{semexprs}(\mathrm{v}, \mathrm{st}, \mathrm{jexprs}, \mathrm{tds}; \mathrm{v1}, \mathrm{st1}, \mathrm{vals1})\rangle \\
\qquad\qquad\qquad\qquad (\mathrm{v1} = \mathrm{v0} \wedge \mathrm{st1} = \mathrm{st0} \wedge \mathrm{vals1} = \mathrm{vals0}))
\end{array}
$$

This is proved by applying the 'smallest' axiom. The four higher-order predicate variables *Pstm*, *Pstms*, *Pexpr*, *Pexprs* are instantiated with

$$
\begin{array}{l}
\mathrm{Pstm} \leftarrow \lambda\ \mathrm{ss}, \mathrm{jstm}, \mathrm{tds}, \mathrm{ss0}.\ \langle \mathrm{semstm}(\mathrm{ss}\ .1, \mathrm{ss}\ .2, \mathrm{jstm}, \mathrm{tds}; \mathrm{v1}, \mathrm{st1})\rangle \\
\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad (\mathrm{v1} = \mathrm{ss0}\ .1 \wedge \mathrm{st1} = \mathrm{ss0}\ .2) \\
\mathrm{Pstms} \leftarrow \lambda\ \mathrm{ss}, \mathrm{jstms}, \mathrm{tds}, \mathrm{ss0}.\ \langle \mathrm{semstms}(\mathrm{ss}\ .1, \mathrm{ss}\ .2, \mathrm{jstms}, \mathrm{tds}; \mathrm{v1}, \mathrm{st1})\rangle \\
\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad (\mathrm{v1} = \mathrm{ss0}\ .1 \wedge \mathrm{st1} = \mathrm{ss0}\ .2) \\
\mathrm{Pexpr} \leftarrow \lambda\ \mathrm{ss}, \mathrm{jexpr}, \mathrm{tds}, \mathrm{ssv}.\ \langle \mathrm{semexpr}(\mathrm{ss}\ .1, \mathrm{ss}\ .2, \mathrm{jexpr}, \mathrm{tds}; \mathrm{v1}, \mathrm{st1}, \mathrm{val1})\rangle \\
\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad (\mathrm{v1} = \mathrm{ssv}\ .1 \wedge \mathrm{st1} = \mathrm{ssv}\ .2\ .1 \wedge \mathrm{val1} = \mathrm{ssv}\ .2\ .2) \\
\mathrm{Pexprs} \leftarrow \lambda\ \mathrm{ss}, \mathrm{jexprs}, \mathrm{tds}, \mathrm{ssvlist}.\ \langle \mathrm{semexprs}(\mathrm{ss}\ .1, \mathrm{ss}\ .2, \mathrm{jexprs}, \mathrm{tds}; \mathrm{v1}, \mathrm{st1}, \mathrm{vals1})\rangle \\
\qquad\qquad\qquad\qquad\qquad\qquad (\ \mathrm{v1} = \mathrm{ssvlist}\ .1 \wedge \mathrm{st1} = \mathrm{ssvlist}\ .2\ .1 \wedge \mathrm{vals1} = \mathrm{ssvlist}\ .2\ .2)
\end{array}
$$

Essentially this replaces every occurrence of the higher-order variables by a procedure call in the reduction rules. After that, a case distinction yields one case for every reduction rule which is proved by symbolic execution of the program.

A nice consequence of the equivalence is the following corollary.

3.4.3 Theorem (The Semantics is deterministic)

The semantics is deterministic. For expressions this means:

$$
\begin{array}{l}
\quad (v \times st)[\![e]\!](v_0 \times st_0 \times val_0) \\
\wedge\ (v \times st)[\![e]\!](v_1 \times st_1 \times val_1) \\
\rightarrow v_0 = v_1 \wedge st_0 = st_1 \wedge val_0 = val_1
\end{array}
$$

The same holds for the other *sem* relations.

Proof

This follows from the four equivalences. The programs are deterministic. This is checked syntactically by KIV because they do not use indeterministic statements, and the result variables depend only on the input values (this is also checked syntactically).

Chapter 4

The Calculus

In this chapter the calculus for sequential Java is presented. We begin with an overview that sketches the main ideas of the calculus. Then the formal definitions and the detailed proof rules follow. At the end of the chapter the soundness theorem for the calculus is presented. Parts of the overview have been published in [Ste04] by the author in advance of this thesis.

4.1 Overview

4.1.1 Dynamic Logic

We use a dynamic logic (DL, [Har79] [HRS89] [HKT00]) for the verification of Java programs. (First-Order) DL extends first order logic with two modal operators, a box [.]. and a diamond $\langle . \rangle$ . . The box (diamond) contains a program, afterwards follows again a DL formula. In our case, the box (diamond) contains a tuple of a variable for the store, and a Java statement. In $\langle st/\alpha \rangle\ \varphi$, st is the store variable, α a Java statement, and φ a DL formula. The only difference between box and diamond is that the box assumes termination of the program, while the diamond enforces termination.

The store variable is included in the box (diamond), because it is needed as the context in which to evaluate the program. The Java type declarations tds are a global context (they can be seen as part of the underlying specification), and are not included in the formulas.

The formal definitions are

$$tds, \mathcal{A}, v \models [\mathtt{st}/\alpha]\ \varphi \Leftrightarrow \forall\ w.(v \times v[\mathtt{st}])[\![\alpha]\!]_{tds}(w \times st_0) \rightarrow tds, \mathcal{A}, w[\mathtt{st}, st_0] \models \varphi$$

$$tds, \mathcal{A}, v \models \langle \mathtt{st}/\alpha \rangle\ \varphi \Leftrightarrow \exists\ w.(v \times v[\mathtt{st}])[\![\alpha]\!]_{tds}(w \times st_0) \wedge tds, \mathcal{A}, w[\mathtt{st}, st_0] \models \varphi$$

The store variable `st` is lookup in the current variable mapping v, $v[\mathtt{st}]$, and the result is used as the initial store for the semantics of the statement α. If α terminates and computes an new variable mapping w and store st_0 the store is bound to the variable `st` in w, $w[\mathtt{st}, st_0]$. Then the new variable mapping is used to evaluate φ. If α does not terminate, no w exists, and for the box the precondition is trivially true, and $tds, \mathcal{A}, v \models [st/\alpha]\ \varphi$ is true regardless of φ. For a diamond, $tds, \mathcal{A}, v \models \langle st/\alpha \rangle\ \varphi$ is false for every φ, if α does not terminate. $[st/\alpha]\ \varphi \Leftrightarrow \neg\ \langle st/\alpha \rangle\ \neg\ \varphi$ holds. The Java semantics is deterministic; if w and st_0 exist with $(v \times st)[\![\alpha]\!](w \times st_0)$ both are unique.

The verification framework is a sequent calculus [Sza69]. $\varphi_1, ..., \varphi_m \vdash \psi_1, ..., \psi_n$ is a sequent with a left side (left of the turnstyle $\vdash$) that is a list of DL formulas, and a right side, also a list of formulas. The left side is called *antecedent*, the right side *succedent*. We use Γ and Δ for lists of formulas. A sequent holds if the conjunction of the left formulas implies the conjunction of the right formulas:

$$tds, \mathcal{A}, v \models (\varphi_1, \ldots, \varphi_m \vdash \psi_1, ..., \psi_n) \Leftrightarrow tds, \mathcal{A}, v \models \varphi_1 \wedge\ \ldots\ \wedge \varphi_m \rightarrow \psi_1 \vee\ \ldots\ \vee \psi_n$$

Since both sides contain DL formulas, both sides may contain Java programs. Hoare's calculus is a very special case of dynamic logic. A Hoare triple $\varphi\{\alpha\}\psi$ is written $\varphi \vdash [\alpha]\psi$ as a DL sequent.

4.1.2 Main Features of the Calculus

Before going into details we present a short overview of the calculus:

Completeness. The calculus is incomplete for various reasons, but this is irrelevant for normal applications (a proof will never get stuck because of the incompleteness). Reasons for incompleteness are:

1. There is no bounded method call. This means it is not possible to prove that a method that calls itself recursively with the same arguments and does nothing else does not terminate.
2. The calculus is designed in a manner to allow the addition of new type declarations. This implies that no properties can be proved for non-existent classes. See chapter 6.
3. Some proof rules have preconditions that are met only by fully type correct programs. However, the calculus is also applicable for slightly more liberal programs. See for example the rule for the `switch` statement in chapter 4.4.8.

Result of expressions. Boxes and diamonds contain only Java statements, not expressions. The proof rules for an expression e deal with an expression statement that is an assignment of e to some variable x, i.e. `x = e;`, or with the expression statement `e;`. x is an arbitrary variable. This avoids technical overhead to capture the result of an expression (as, for example, in [vO01]).

Evaluation of expressions. Subexpressions must be evaluated before a (top-level) expression is evaluated. This is done by *flattening*: Nested expressions are flattened by introducing auxiliary variables for subexpressions and intermediate values. Of course, this flattening must obey the Java evaluation rules. For example, $\langle st/x = m(f(g(y)), h(z)); \rangle\ \varphi$ is transformed to $\langle st/x_0 = g(y); x_1 = f(x_0); x_2 = h(z); x = m(x_1, x_2); \rangle\ \varphi$. x_0, x_1, x_2 must be new variables, i.e. not occurring free in the sequent. A *basic* expression is a local variable or a literal, all other expressions are flattened. Literals in this calculus are not only constants, but may contain variables and algebraic terms (see chapter 2.1).

Program state. The store and the local variables contain the complete program state. This means it is possible to express that a statement ends with an exception (or even a break or return), or that a class has never been initialized. The store is identical to the store used in the semantics (described in chapter 3.1.3).

Blocks and jumps. Blocks are flattened as well, i.e. $\langle st/\texttt{\{\{}\alpha\texttt{\}\}}\rangle\ \varphi$ becomes $\langle st/\alpha\rangle\ \varphi$. `try` blocks are also eliminated, i.e. $\langle st/\texttt{try \{}\alpha\texttt{\} finally }\beta\rangle\ \varphi$ becomes $\langle st/\alpha\rangle\ \langle st/\texttt{finally}(\beta)\rangle\ \varphi$. finally($\beta$) is a new statement (that does not exist in Java) that serves as a target for throws inside α. $\langle st/\texttt{finally}(\beta)\rangle\ \varphi$ is transformed into $\langle st/\beta\rangle\ \langle st/\texttt{endfinally}(m)\rangle\ \varphi$; endfinally($m$) is again a new statement to mark the end of a finally block. For labeled statements a new Java statement `target` is introduced to catch breaks. Another new statement `targetexpr` catches `return` statements and their value.

A very important feature of the calculus is that the program state is completely and uniquely described by the values of the program variables and the store. This means it is possible to 'talk' about the program state after some method call (provided it terminates) without actually knowing what the method does (see the proof rule in chapter 4.6.1). This property is the key for inductive proofs.

The calculus essentially has one rule for every Java expression and statement, plus some generic rules. It works by symbolic execution of the Java program from its beginning to its end (this roughly corresponds to the computation of strongest postcondition). This means it follows the natural execution of the program, and is much more intuitive than inventing intermediate values (as in a Hoare calculus) or computing weakest preconditions (by evaluating a program from the end to the start). Nested expressions and blocks are flattened to a sequence of simple expressions

and statements that can be executed directly. The additional Java statements mentioned above are used to mark the end of a block. In the sequent calculus, programs may occur in the antecedent (on the left of the turnstile $\vdash$) or the succedent (on the right). Most of the rules are applicable in both situations. Also, most rules are applicable for boxes and diamonds.

An example for a rule is the rule for an instance field assignment. It has three premises:

$$\frac{\begin{array}{ll}1. & \Gamma, st[mode] \neq normal \vdash \varphi, \Delta \\ 2. & \Gamma, st[mode] = normal, e = null \\ & \vdash \langle st/\texttt{throw new NullPointerException();}\rangle\ \varphi, \Delta \\ 3. & \Gamma, st[mode] = normal, e \neq null, st_0 = st[e-f, e_0] \\ & \vdash \langle st_0/x = e_0;\rangle\ \varphi[st/st_0], \Delta\end{array}}{\Gamma \vdash \langle st/x = (e.f = e_0);\rangle\ \varphi, \Delta}$$

$e.f = e_0$ is the instance field assignment. As mentioned previously, the expression may occur on the right hand side of an assignment to capture the result of the expression. Hence, the statement is $x = (e.f = e_0);$. f is the field name, e the invoking expression. The rule is only applicable if e and e_0 are *basic* expressions, either local variables or literals. If the mode is not normal the expression is skipped (first premise). If the invoking expression is null a `NullPointerException` is raised (second premise). Otherwise the store st is modified by setting the field $e--f$ to the value of e_0 ($st[e-f, e_0]$). Often, φ is again a diamond formula containing a program. φ must be evaluated in the context of the new store. This is done by introducing a new variable st_0 and the equation $st_0 = st[e-f, e_0]$, and by replacing st by st_0 in φ, $\varphi[st/st_0]$. Since e and e_0 are local variables or literals they require no further evaluation, but can be taken directly as values. If they are other Java expressions they have to be flattened first. This works as follows:

1. For an expression $x = e$ select the immediate subexpressions $e_1, \ldots, e_n$ of e.
2. Find the first e_i that is not a basic expression (a local variable or a literal), and that does not cause a variable conflict (see case 4).
3. Replace e_i in e by a new variable y yielding e' and add the assignment $y = e_i$ before $x = e'$.
4. A variable conflict occurs if e_i contains an assignment to a variable that occurs in $e_1, \ldots, e_{i-1}$, e.g. in `x * (x = 3)`. In this case a renaming is necessary.

After a finite number of applications the algorithm will return a list of assignments where every subexpression is either a local variable or a literal. Then a rule for this expression is applicable. The test of an `if`, the expression of a `switch`, `return`, or `throw` can also be flattened in this manner (but not the test of a `while`, `do` or `for`). The rule for an `if` statement has three premises, one for a jump, one for the `then` part, and one for the `else` part:

$$\frac{\begin{array}{ll}1. & \Gamma, st[mode] \neq normal \vdash \varphi, \Delta \\ 2. & \Gamma, st[mode] = normal, e \vdash \langle st/\alpha\rangle\ \varphi, \Delta \\ 3. & \Gamma, st[mode] = normal, \neg\, e \vdash \langle st/\beta\rangle\ \varphi, \Delta\end{array}}{\Gamma \vdash \langle st/\texttt{if}\ (e)\ \alpha\ \texttt{else}\ \beta\rangle\ \varphi, \Delta}$$

The test e must be a basic expression. Then its boolean value can be used directly in the sequent. Diamonds and boxes contain exactly one statement. A list of statements (e.g. the statements of a block) is divided into separate diamonds:

$$\frac{\begin{array}{ll}1. & \Gamma, st[mode] \neq normal \vdash \varphi, \Delta \\ 2. & \Gamma, st[mode] = normal \vdash \langle st/\alpha'_1[\underline{x}/\underline{y}]\rangle\ \ldots \langle st/\alpha'_n[\underline{x}/\underline{y}]\rangle\ \varphi, \Delta\end{array}}{\Gamma \vdash \langle st/\{\alpha_1 \ldots \alpha_n\}\rangle\ \varphi, \Delta}$$

α_i are the top level statements of the block, $\underline{x}$ the local variables declared in the block, and $\underline{y}$ new variables. α'_i is α_i except for a local variable declaration. $ty\ x = e$ becomes an assignment $x = e$.

The replacement of the local variables by new variables ensures that the variables really behave as local variables.

One of the most important features of a dynamic logic is that it is well suited for interactive proofs, much better – we feel – than a Hoare or weakest precondition calculus. Because a formula containing a Java statement is only one formula among others, the user can mix predicate logic rules (case distinctions, quantifier instantiations, cut, etc., or advanced rules like rewriting or simplification) freely with Java rules. This helps to reduce the size of non-Java formulas during the proof. Furthermore, a method call (or any other statement) can be either evaluated, or one or several lemmas for the method call can be applied at different points in the proof. The reason is that programs can appear on both sides of the turnstile $\vdash$ so that it is possible to argue "if program α computes x then program β computes y". The following proof rule is valid:

$$\frac{\langle st/\alpha\rangle\ \underline{x} = \underline{y}, \varphi[\underline{x}/\underline{y}], \Gamma \vdash \psi[\underline{x}/\underline{y}], \Delta}{\langle st/\alpha\rangle\ \varphi, \Gamma \vdash \langle st/\alpha\rangle\ \psi, \Delta}$$

$\langle st/\alpha\rangle\ \varphi$ is given as a precondition, i.e. α terminates and afterwards φ holds (this can be a lemma about α). α can only modify a number of variables $\underline{x}$ (the assigned variables of α, and the store st) and these variables describe the state exactly. Therefore it is possible to introduce new variables $\underline{y}$ that hold the result and then we know that $\varphi[\underline{x}/\underline{y}]$ holds (φ with $\underline{x}$ replaced with $\underline{y}$). Now the formula to prove is $\langle st/\alpha\rangle\ \psi$ and we know that α computes $\underline{y}$. Therefore α can be discarded on the right hand side of the turnstile $\vdash$ and it remains to prove that $\psi[\underline{x}, \underline{y}]$ holds. This is the typical situation when a lemma is used. But the program α does not disappear from the sequent. It remains in the antecedent and later the user can apply another lemma that introduces another property. Nothing similar is possible in a Hoare calculus. Another advantage is that induction can be used for loops or recursive methods. This means the user has more freedom to structure proofs.

4.1.3 Comparison

As mentioned in the introduction (chapter 1) there exists only one other Java calculus based on dynamic logic [Bec00] in the KeY project [ABB+04] [KeY]. The calculus itself has three major differences to the one presented in this work:

1. Exceptions are modeled as non-termination.

   This means it is not possible to express the fact that a program throws an exception directly. Rather, it must be surrounded by a `try-catch` block that, for example, sets a variable to **true**:

   $\langle$`try` α `catch(Exception e) { b = true;}` $\rangle$ b = true

2. Blocks are not flattened (expressions are, though).

   This means that jumps (`throw`, `return`, `break`) are not handled by additional Java statements that catch them. Rather, the nesting of statements is preserved if it is relevant for jumps. This requires that a program is divided into a *non-active prefix* φ, a first *active* command p, and the rest of the program ω, for example (taken from [Mos05])

   $$\underbrace{\texttt{l:\{ try \{}}_{\pi}\texttt{i=0;}\ \underbrace{\texttt{j=0; \} finally\{ k=0;\} \}}}_{\omega}$$

   This also requires the introduction of a new statement `method-frame` that groups the body of a method call to handle `return`'s correctly. It is similar to the `targetexpr` statement used in this work.

3. No explicit store exists.

   Objects are represented by (dynamic) functions. This raises the problem of *aliasing*: Two variables can point to the same object. For example, it is not clear whether a formula

$o1.a = 1$ still holds after an assignment $o2.a = 2;$, because $o1$ and $o2$ may or may not point to the same object.

This is handled by introducing *updates* that can precede any formula, $\{loc := val\}\varphi$.

Topics 2. and 3. introduce quite a technical overhead, and have some disadvantages: Some useful properties do not hold (e.g. $\langle\alpha\ \beta\rangle\ \varphi \Leftrightarrow \langle\alpha\rangle\ \langle\beta\rangle\ \varphi$), and it is not possible to reason about arbitrary pointer structures. (The latter is not in the focus of the KeY project anyway.) The advantage is that is easier to express that, for example, an object has the correct fields, or that an update does not modify another object.

A classical Hoare calculus has predicate logic formulas as pre- and postcondition. All existing Hoare calculi for Java use more complicated pre- and postconditions that include, for example, the store, the result value of an expression, or different formulas for the normal termination and abrupt completion ([vO01] uses triples as postcondition, [JP03] septuples). The same is true for the weakest precondition calculi (see chapter 1). So in a sense the calculus presented here requires the least technical overhead for Java.

4.1.4 Some Errors made and found

As can be imagined several errors were found during the soundness proof of the calculus. Most of them were errors only for type incorrect programs. For example, the definition of free variables must handle the case that a local variable declaration occurs outside a block or that a local variable occurs outside the scope of a local variable declaration. Other errors were more serious because they concerned type correct programs:

1. One premise in the instance method invocation was missing (that checks that the run-time type of the invoker is correct, see chapter 4.3.26). This is important even for type correct programs because the store may not be compatible with the program.

2. The flattening rule contained no check for variable conflicts. `y = x * (x = 3);` was flattened to `z = (x = 3); y = x * z;` which is wrong. (A renaming is needed: `x0 = x; z = (x = 3); y = x0 * z;`).

3. A similar problem occurred for the postfix increment: `x = y++` was transformed into `x = y; y = y + 1;` which is wrong for `x = x++;` (the right hand side is fully evaluated before the assignment occurs).

These errors were found because the verification of the proof rules failed. However, some errors were found in the semantics as well.

1. *first active use* was not handled correctly in the semantics rules, i.e. whether static initialization occurs before or after the arguments are evaluated. (compound assignment to static field and `new` class: before evaluation of arguments, simple assignment to static field and static method invocation: after evaluation of arguments.) This was already discussed in chapter 3.2.16.

2. The semantics rules for compound assignment failed to cast the result back to `byte` or `short` if the right hand side was `byte` or `short`. (In `byte b = 127; b += 1;` the result is (byte)−128.)

These errors were found because the verification of the proof rules also failed, and analysis revealed the errors to be in the semantics. However, both semantics and calculus could be wrong. It is possible to validate the semantics by 'running' test programs in KIV (automatically applying the proof rules) and comparing the output with a run of a Java compiler and JVM (currently 150 examples), and this certainly increases confidence in the semantics, but who would think about writing programs like `x = x++;`?

4.1.5 An Example Proof

The aim of the example is to show the 'look and feel' of the calculus for a small example that involves a `for` loop, exceptions and abrupt termination of the loop. It is typical for Java Card programs to use this programming style. We consider a Java Card application for storing tickets. (More information is not necessary at this point. Chapter 7 contains more details on Java Card programs.) Since the available memory for a Java Card program (an *applet*) is severely limited the maximal number of tickets that can be stored must be fixed in advance. Since Java Card has no garbage collection storage cells cannot be reclaimed. Therefore it is usual programming practice to allocate all objects when the applet is loaded onto the smart card, and to reuse these objects. The example has a capacity of 20 tickets that are stored in an array. A field `free` indicates if the entry is free or not. If a ticket is loaded the value is set to false, if it is deleted after usage the entry is set back to true.

```
class Ticket { boolean free = true; ⟨rest of class⟩}

public class Cardlet extends Applet {
   final static byte MAX = 20;
   Ticket[] tickets = new Ticket[MAX];
   Cardlet() {
      for(byte c=0;c<MAX; c++) tickets[c] = new Ticket();}
   ⟨rest of class⟩}
```

The following method can be used somewhere in the applet to find a free position:

```
byte findFree () {
    for(byte c=0; c < MAX; c++)
        if (tickets[c].free) return c;
    ISOException.throwIt(SW_FILE_FULL);
}
```

If no free position is available an exception is thrown (without creating a new exception object!) from the predefined method `ISOException.throwIt`. If the `findFree` method is used several times in the code it is good proving practice to formulate some lemmas about its behavior and re-use them wherever possible. For example,

findFree-install : install(st) $\vdash$ ⟨st/ by = cardlet.findFree();⟩ install(st)

findFree-throw :
  #tickets(st) = b2i(MAX), install(st), st[mode] = normal
$\vdash$ ⟨st/ by = cardlet.findFree();⟩ ISOException(SW_FILE_FULL, st)

findFree-ok :
  #tickets(st) < b2i(MAX), st = st_0, install(st), st[mode] = normal
$\vdash$ ⟨st/ by = cardlet.findFree();⟩ (st = st_0 $\wedge$ free_ticket(b2i(by), st))

The method assumes that the array entries are not null. This is the case when the constructor completed normally. Hence an invariant *install*(st) is needed for the applet. This invariant will contain other properties of the applet that should hold after object creation. *install*(st) is a user defined predicate for the store. The second property, *findFree-throw*, states that the method will throw an exception `SW_FILE_FULL` if the number of tickets stored in the array is already `MAX` (#tickets(st) = b2i(MAX), b2i converts a byte into an integer). Here we can assume that the invariant holds, and we must assume that no abrupt transfer of control happens initially (st[mode] = normal) because then the method call will be skipped. Finally, *findFree-ok* states that the method will return a free position if there is one.

We show the proof for *findFree-ok*. Method call and initialization of the `for` loop results in

$\mathrm{st} = \mathrm{st}_0$, this = cardlet, c = 0, st[mode] = normal,
#tickets(st) < b2i(MAX), install(st)
$\vdash$ ⟨st/`for`(c < MAX; c ++) `if` (this.tickets[c].free) `return` c; `else` { }⟩
⟨st/ISOException.throwIt(SW_FILE_FULL);⟩
⟨st/`targetexpr`(by)⟩ (free_ticket(b2i(by), st) ∧ $\mathrm{st} = \mathrm{st}_0$)

The `for` loop contains no initialization so it can be unwound; *cardlet* is a reference to an object of type `Cardlet` that becomes the value of `this` inside the method. Now we use induction on |MAX − c| and generalize the goal by replacing c = 0 with 0 ≤ c ∧ c ≤ MAX (this is done automatically), and add the formula #tickets(st[cardlet – tickets].refval, b2i(c)−1, st) = b2i(c) stating that the number of tickets from 0 to $c-1$ in the array is c (this means that all tickets below c are not free). st[cardlet – tickets].refval returns the reference that is stored in the `tickets` field of the cardlet object. This property is needed to prove that the loop counter c can never reach MAX. Then we unwind the `for` loop once and obtain

Ind-Hyp, ... ⟨*other preconditions*⟩ ...
$\vdash$ ⟨st/`if` (b2i(c) < b2i(MAX))
if (this.tickets[c].free) `return` c; `else` { } c ++;⟩
⟨st/`for`(c < MAX; c ++) `if` (this.tickets[c].free) `return` c; `else` { }⟩
⟨st/ISOException.throwIt(SW_FILE_FULL);⟩
⟨st/`targetexpr`(by)⟩ (free_ticket(b2i(by), st) ∧ $\mathrm{st} = \mathrm{st}_0$)

If the first `if` test is true and the second one is false we obtain after the postfix increment `c++`

Ind-Hyp, c_0 = i2b(b2i(c) + 1), ¬ $\mathrm{st}[\mathrm{r}_0$ – free].boolval
... ⟨*other preconditions*⟩ ...
$\vdash$ ⟨st/`for`(c_0 < MAX; c_0++) `if` (this.tickets[c_0].free) `return` c_0; `else`{}⟩
⟨st/ISOException.throwIt(SW_FILE_FULL);⟩
⟨st/`targetexpr`(by)⟩ (free_ticket(b2i(by), st) ∧ $\mathrm{st} = \mathrm{st}_0$)

The formula on the right hand side of the turnstyle $\vdash$ (with the `for` loop) is identical to the formula where the induction started, except that c is replaced by c_0. This means we can apply the induction hypothesis (this requires proving that no overflow occurs for the **byte** value c.), and obtain a program formula on the left hand side of $\vdash$. The result is an axiom:

⟨st/`for`(c_0 < MAX; c_0 ++) ...$\vdash$ ⟨st/`for`(c_0 < MAX; c_0 ++) ...

The proof proceeds as a proof done on paper, except that the sequents tend to get longer, because preconditions are accumulated. The same principle (induction and unwinding of the loop) can be used for `while` or `do` loops. The calculus described below also contains special invariant rules. In other proofs only the lemmas for the `findFree` method are used. This allows a nice structuring of the proofs.

4.2 Semantics of Formulas and Sequents

In this section we define precisely dynamic logic (DL) formulas, sequents, and proofs. The definitions (from variable mappings to models to formulas) are standard for algebraic specifications or first-order logic (see e.g. [LEW96] [BM04]), except that a global context *tds* (the Java type declarations, i.e. class and interface declarations) is introduced. The type declarations never change during the evaluation of formulas. A formula φ is evaluated with type declarations *tds*, an algebra $\mathcal{A}$ and a variable mapping v, written as *tds*, $\mathcal{A}$, v $\models \varphi$:

$$
\begin{array}{l}
tds, \mathcal{A}, v \models true \\
tds, \mathcal{A}, v \not\models false \\
tds, \mathcal{A}, v \models t_1 = t_2 \Leftrightarrow (t_1)_{\mathcal{A},v} = (t_2)_{\mathcal{A},v} \\
tds, \mathcal{A}, v \models p(t_1, \ldots, t_n) \Leftrightarrow ((t_1)_{\mathcal{A},v}, \ldots, (t_n)_{\mathcal{A},v}) \in p_{\mathcal{A}} \\
tds, \mathcal{A}, v \models \neg\, \varphi \Leftrightarrow \text{ not } tds, \mathcal{A}, v \models \varphi \\
tds, \mathcal{A}, v \models \varphi \wedge \psi \Leftrightarrow (tds, \mathcal{A}, v \models \varphi \text{ and } tds, \mathcal{A}, v \models \psi) \\
tds, \mathcal{A}, v \models \varphi \vee \psi \Leftrightarrow (tds, \mathcal{A}, v \models \varphi \text{ or } tds, \mathcal{A}, v \models \psi) \\
tds, \mathcal{A}, v \models \varphi \rightarrow \psi \Leftrightarrow (tds, \mathcal{A}, v \models \varphi \Rightarrow tds, \mathcal{A}, v \models \psi) \\
tds, \mathcal{A}, v \models \varphi \leftrightarrow \psi \Leftrightarrow (tds, \mathcal{A}, v \models \varphi \Leftrightarrow tds, \mathcal{A}, v \models \psi) \\
tds, \mathcal{A}, v \models \forall\, x.\varphi \Leftrightarrow \text{ for all } d \text{ with okval}(d, x)\text{: } tds, \mathcal{A}, v[x, d] \models \varphi \\
tds, \mathcal{A}, v \models \exists\, x.\varphi \Leftrightarrow \text{ exists } d \text{ with okval}(d, x)\text{: } tds, \mathcal{A}, v[x, d] \models \varphi \\
tds, \mathcal{A}, v \models [\mathtt{st}/\alpha]\ \varphi \Leftrightarrow \forall\, w.(v \times v[\mathtt{st}])[\![\alpha]\!]_{tds}(w \times st_0) \rightarrow tds, \mathcal{A}, w[\mathtt{st}, st_0] \models \varphi \\
tds, \mathcal{A}, v \models \langle \mathtt{st}/\alpha \rangle\ \varphi \Leftrightarrow \exists\, w.(v \times v[\mathtt{st}])[\![\alpha]\!]_{tds}(w \times st_0) \wedge tds, \mathcal{A}, w[\mathtt{st}, st_0] \models \varphi
\end{array}
$$

okval(d, x) requires that the value d is of the same sort as x since we have simple sorted terms. Formulas (with syntax and semantics) are specified in KIV. For example, the definition of the negation in KIV is written as

$$\text{tds} \times \text{A} \times \text{v} \models _\neg\ \varphi \leftrightarrow \neg\ \text{tds} \times \text{A} \times \text{v} \models \varphi$$

Since the logical operations cannot be overloaded, $_\neg$ is used for the negation. Similarly, all other logical operators are prefixed with an underscore. The underscore will be omitted in the sequel for readability. The algebra $\mathcal{A}$ occurs in the KIV specification, but only as a dummy. No attempt has been made to specify the notion of an algebra. The definition of the semantics of formulas is lifted in the usual manner:

$$
\begin{array}{l}
tds, \mathcal{A} \models \varphi \Leftrightarrow \text{ for all } v\text{: } tds, \mathcal{A}, v \models \varphi \\
tds, \mathcal{A} \models T \Leftrightarrow tds, \mathcal{A} \models \varphi \text{ for all } \varphi \in T \text{ (T a set of formulas)} \\
tds, T \models \varphi \Leftrightarrow \text{ for all } \mathcal{A} \text{ with } tds, \mathcal{A} \models T : tds, \mathcal{A} \models \varphi
\end{array}
$$

For sequents we have the same lifting:

$$
\begin{array}{l}
tds, \mathcal{A}, v \models (\Gamma \vdash \Delta) \Leftrightarrow tds, \mathcal{A}, v \models con(\Gamma) \rightarrow dis(\Delta) \\
tds, \mathcal{A} \models (\Gamma \vdash \Delta) \Leftrightarrow tds, \mathcal{A} \models con(\Gamma) \rightarrow dis(\Delta) \\
tds, \mathcal{A} \models seqs \Leftrightarrow tds, \mathcal{A} \models (\Gamma \vdash \Delta) \text{ for all } (\Gamma \vdash \Delta) \in seqs \\
tds, seqs \models (\Gamma \vdash \Delta) \Leftrightarrow \text{ for all } \mathcal{A} \text{ with } tds, \mathcal{A} \models seqs : tds, \mathcal{A} \models (\Gamma \vdash \Delta)
\end{array}
$$

The calculus consists of a set of proof rules R (defined below), and we define a *proof* relative to type declarations tds and a set of theorems/axioms (sequents) $seqs$:

A proof (derivation) is a tree built up by the rules where the premises are $\subseteq seqs$.

and

$tds, seqs \vdash (\Gamma \vdash \Delta) \Leftrightarrow$ there exists a proof (derivation) with conclusion $\Gamma \vdash \Delta$.

A proof rule $\dfrac{p_1 \ldots p_n}{c}$ consists of n premises $p_1, \ldots, p_n$ and a conclusion c, and possibly some nonlogical conditions (for example, that a variable x is new). $p_1, \ldots, p_n$ and c are sequents. A proof rule is sound if

$$tds, seqs \models p_1 \text{ and } \ldots \text{and } tds, seqs \models p_n \Rightarrow tds, seqs \models c$$

The calculus is defined relative to an algebraic specification of the Java store, the primitive Java types (integers, bytes, shorts, and booleans), and other data types. Given an algebra $\mathcal{A}$ that is a model of these specification the soundness proof for a proof rule of the calculus becomes

$tds, \mathcal{A} \models p_1$ and ... and $tds, \mathcal{A} \models p_n \Rightarrow tds, \mathcal{A} \models c$

As mentioned, $\mathcal{A}$ is only a dummy in the KIV specification. Assumptions about $\mathcal{A}$ are written as axioms. (The KIV specifications models the syntax of terms and formulas, and their semantics. One of these assumptions is, for example, that the term '0' evaluates to the integer value 0.)

As in classical dynamic logic the following duality holds:

tds, $\mathcal{A}$, v $\models$ [st/α] $\varphi \leftrightarrow \neg$ $\langle$st/$\alpha\rangle$ φ

The main theorem at the end of this chapter will be the soundness of the calculus. We proceed with the proof rules for Java expressions.

4.3 Expressions

This section contains the proof rules for expressions. Every rule is needed in different versions: For boxes and diamonds, in the antecedent and the succedent, and with an assignment and without. Since the calculus uses assignments to capture the result value of an expression, the usual form of an expression in a formula is $\langle st/x = e; \rangle\ \varphi$. Technically, the diamond contains an expression statement (indicated by the semicolon) with an assignment. If an expression is evaluated only for its side effect, e.g. `i++`, the formula has the form $\langle st/e; \rangle\ \varphi$. Both forms may appear, except that a method call to a **void** method cannot be assigned to a variable (JLS 15.1). This means that almost every proof rule for expressions has eight versions.

4.3.1 Different Versions of a Proof Rule

From a theoretical point of view, only two versions are needed: For diamonds with assignments in the antecedent and the succedent. Boxes can be eliminated using the duality [st/α] $\varphi \leftrightarrow \neg$ $\langle$st/$\alpha\rangle$ φ, and a generic rule allows the introduction of an assignment to a new variable (except for **void** methods). Most proof rules have the property that the conclusion is equivalent to its premises. This means we can formulate an equivalence formula for a diamond with an assignment and derive all eight versions of the proof rule from this formula.

As an example we show all eight versions of the proof rule and the equivalence formula for the literal rule.

Equivalence: The equivalence formula is

$\langle$st/x = l;$\rangle$ φ
$\leftrightarrow$ (st[mode] $\neq$ normal $\rightarrow$ φ)
$\wedge$ (st[mode] = normal $\rightarrow$ $\forall$ z. z = l.t $\rightarrow$ φ [x/z])

z must be a new variable, i.e. $z \notin vars(l) \cup (vars(\varphi) \setminus \{x\})$. The literal is assigned to a local variable x. The proof rule will discard the assignment and add a new predicate logic equation if the mode is normal. (Otherwise the assignment is skipped.) $l.t$ selects the the algebraic term from the literal. A new variable z is introduced if x occurs in l (literals may contain variables in this work), and x is replaced by z in φ, φ [x/z]. This formula holds. The corresponding proof rules are shown below.

Version 1: diamond succedent with assignment:

$$\frac{\begin{array}{ll}1. & \Gamma, st[mode] \neq normal \vdash \varphi, \Delta \\ 2. & \Gamma, st[mode] = normal, y = l.t \vdash \varphi[x/y], \Delta\end{array}}{\Gamma \vdash \langle st/x = l; \rangle\ \varphi, \Delta}$$

y new, i.e. $y \notin vars(l) \cup (vars(\varphi) \setminus \{x\}) \cup free(\Gamma) \cup free(\Delta)$

Version 2: diamond antecedent with assignment:

$$\frac{\begin{array}{ll}1. & \varphi, st[mode] \neq normal, \Gamma \vdash \Delta \\ 2. & \varphi[x/y], st[mode] = normal, y = l.t, \Gamma \vdash \Delta\end{array}}{\langle st/x = l; \rangle\ \varphi, \Gamma \vdash \Delta}$$

Version 3: box succedent with assignment:

$$\frac{\begin{array}{ll}1. & \Gamma, st[mode] \neq normal \vdash \varphi, \Delta \\ 2. & \Gamma, st[mode] = normal, y = l.t \vdash \varphi[x/y], \Delta\end{array}}{\Gamma \vdash [st/x = l;]\varphi, \Delta}$$

Version 4: box antecedent with assignment:

$$\frac{\begin{array}{ll}1. & \varphi, st[mode] \neq normal, \Gamma \vdash \Delta \\ 2. & \varphi[x/y], st[mode] = normal, y = l.t, \Gamma \vdash \Delta\end{array}}{[st/x = l;]\varphi, \Gamma \vdash \Delta}$$

Version 5: diamond succedent without assignment:

$$\frac{1. \quad \Gamma \vdash \varphi, \Delta}{\Gamma \vdash \langle st/l; \rangle\ \varphi, \Delta}$$

Version 6: diamond antecedent without assignment:

$$\frac{1. \quad \varphi, \Gamma \vdash \Delta}{\langle st/l; \rangle\ \varphi, \Gamma \vdash \Delta}$$

Version 7: box succedent without assignment:

$$\frac{1. \quad \Gamma \vdash \varphi, \Delta}{\Gamma \vdash [st/l;]\varphi, \Delta}$$

Version 8: box antecedent without assignment:

$$\frac{1. \quad \varphi, \Gamma \vdash \Delta}{[st/l;]\varphi, \Gamma \vdash \Delta}$$

The derivation of the eight rules is not difficult, but neither completely trivial. We show it in a schematical manner.

Version 1: The equivalence formula has the form $\psi \leftrightarrow (\psi_1 \rightarrow \varphi_1) \wedge \forall\ z.(\psi_2 \rightarrow \varphi_2)$. From the conclusion $\Gamma \vdash \psi, \Delta$ we can apply the following rules of the sequent calculus:

$$\dfrac{\dfrac{\dfrac{\Gamma, \psi_1 \vdash \varphi_1, \Delta}{\Gamma \vdash \psi_1 \rightarrow \varphi_1, \Delta}\ \text{imp_r} \qquad \dfrac{\dfrac{\Gamma, \psi_2 \vdash \varphi_2, \Delta}{\Gamma \vdash \psi_1 \rightarrow \varphi_1, \Delta}\ \text{imp_r}}{\Gamma \vdash \forall\ z.\psi_1 \rightarrow \varphi_1, \Delta}\ \text{all_r}}{\Gamma \vdash (\psi_1 \rightarrow \varphi_1) \wedge (\forall\ z.\psi_2 \rightarrow \varphi_2), \Delta}\ \text{con_r}}{\Gamma \vdash \psi, \Delta}\ \text{rewrite_equiv_r}$$

The propositional rule rewrite_equiv_r replaces a formula in the succedent (right hand side, hence _r) by an equivalent formula:

$$\frac{\Gamma \vdash \psi, \Delta}{\Gamma \vdash \varphi, \Delta}\ \varphi \leftrightarrow \psi,\ \text{rewrite_equiv_r} \qquad \frac{\psi, \Gamma \vdash \Delta}{\varphi, \Gamma \vdash \Delta}\ \varphi \leftrightarrow \psi,\ \text{rewrite_equiv_l}$$

The rule all_r introduces a new variable with respect to the sequent, for example y. The requirement in the equivalence formula that z is new guarantees that there are no inadvertent dependencies between variables. (If z is new with respect to the equivalence formula, y is new with respect to the sequent, because y also must be new with respect to free(Γ) $\cup$ free(Δ).)

Version 2: The equivalence formula has the form

$$\psi \leftrightarrow (\neg\ \psi_1 \rightarrow \varphi_1) \wedge (\psi_1 \rightarrow \forall\ z.z = l.t \rightarrow \varphi[x/z]).$$

This is more specific than version 1 because it is important that both parts of the conjunction have a mutual exclusive precondition (here $\psi_1 \equiv st[mode] = normal$). Application of the equivalence formula in the antecedent and a cut with $\neg\ psi_1$ the following two goals are obtained:

1. $\neg\ \psi_1, \neg\ \psi_1 \rightarrow \varphi_1, \Gamma \vdash \Delta$ which is simplified to $\varphi_1, \neg\ \psi_1, \Gamma \vdash \Delta$
2. $\psi_1, \psi_1 \rightarrow \forall\ z.z = l.t \rightarrow \varphi[x/z], \Gamma \vdash \Delta$ which is simplified to
   $\forall\ z.z = l.t \rightarrow \varphi[x/z], \psi_1, \Gamma \vdash \Delta$
   There is only one possible instance for the quantifier, $z = l.t$.
   Application of all_l leads to
   $y = l.t, \varphi[x/y], \psi_1, \Gamma \vdash \Delta$

all_l is special: The predicate logic version substitutes the quantified variable by a term. However, substitution is not possible in Java programs. Therefore we use a special version of the rule that introduces a new variable, an equation, and a replacement:

$$\frac{y = t, \varphi[x/y], \Gamma \vdash \Delta}{\forall\ x.\varphi, \Gamma \vdash \Delta}\ \text{all_l}$$

t is term to substitute, and y is a new variable.

Version 3/4: Here we can use duality for boxes/diamonds:

[st/x = l;]$\varphi \leftrightarrow \neg\ \langle$st/x = l;$\rangle\ \neg\ \varphi$

Then the negated formula can be shifted from the succedent into the antecedent (version 3) or from the antecedent into the succedent (version 4). Then the derivation version 2 or version 1 can be applied. This leads to sequents containing $\neg\ \varphi$ (or $\neg\ \varphi$ [x/y]). Shifting the negation to the other side of the turnstyle yields the desired result.

Alternatively, an equivalence formula for boxes can be derived directly:

[st/x = l;]φ
$\Leftrightarrow \neg\ \langle$st/x = l;$\rangle\ \neg\ \varphi$
$\Leftrightarrow \neg\ (\quad$(st[mode] $\neq$ normal $\rightarrow \neg\ \varphi$)
$\quad \wedge$ (st[mode] = normal $\rightarrow \forall$ z. z = l.t $\rightarrow$ ($\neg\ \varphi$)[x/z]))
$\Leftrightarrow (\quad$(st[mode] $\neq$ normal $\rightarrow \neg\ \neg\ \varphi$)
$\wedge$ (st[mode] = normal $\rightarrow \neg\ \forall$ z. z = l.t $\rightarrow$ ($\neg\ \varphi$)[x/z]))
$\Leftrightarrow (\quad$(st[mode] $\neq$ normal $\rightarrow \varphi$)
$\wedge$ (st[mode] = normal $\rightarrow \forall$ z. z = l.t $\rightarrow \varphi$ [x/z]))

The last two transformations are not obvious. It holds:

$\neg\ ((\varphi \rightarrow \psi_1) \wedge (\neg\ \varphi \rightarrow \psi_2)) \Leftrightarrow (\varphi \rightarrow \neg\ \psi_1) \wedge (\neg\ \varphi \rightarrow \neg\ \psi_2)$

$\neg\ \forall\ x.x = t \rightarrow \neg\ \varphi \Leftrightarrow \forall\ x.x = t \rightarrow \varphi$ (provided $x \notin vars(t)$)

Then we can continue as in version 1/2.

Version 5-8: If x is new we have

$$\begin{array}{ll} & \langle \text{st/l;} \rangle\ \varphi \\ \Leftrightarrow & \langle \text{st/x = l;} \rangle\ \varphi \\ \Leftrightarrow & \quad (\text{st[mode]} \neq \text{normal} \rightarrow \varphi) \\ & \wedge\ (\text{st[mode]} = \text{normal} \rightarrow \forall\ \text{z.}\ \text{z} = \text{l.t} \rightarrow \varphi[\text{x/z}]) \end{array}$$

since $x \notin vars(\varphi)$: $\varphi[\text{x/z}] = \varphi$, and we can discard the quantifier:

$$\begin{array}{ll} \Leftrightarrow & \quad (\text{st[mode]} \neq \text{normal} \rightarrow \varphi) \\ & \wedge\ (\text{st[mode]} = \text{normal} \rightarrow \varphi) \end{array}$$

and simplify this to φ.

In the sequel we will only show version 1 of the proof rules with a diamond in the succedent because this is the most common situation: proving a property about a single Java program including termination. Other proof rules have more premises but the same construction can be used. The proof rules for Java expressions that divert slightly from this scheme are:

- reference cast, instanceof, arrayassign, and instance (virtual) method invocation. These rules have an additional precondition. This means an 'equivalence' has a precondition, $\psi \rightarrow (\langle \text{st/x = e;} \rangle\ \varphi \leftrightarrow \dots)$

- method invocation for a void method, because a void method does not return a value and cannot be assigned to a variable.

All proof rules have some syntactical preconditions. This means they are not always applicable. These conditions are met by type correct Java programs. One condition for all proof rules is the same: The type declarations *tds* and the conclusion of the proof rule must be *primitive type correct* (primtc(tds) and primtc(c). Primitive type correctness is a limited form of full type correctness; both are discussed in detail in chapter 5. Since the details are not really relevant for the understanding of the proof rules, but rather technical, this order of presentation was chosen. The reader can safely assume a type correct Java program. Other syntactical preconditions are mentioned for every rule.

Proof rules in the calculus are applied backwards. Looking at the rules it can be seen that the premises are almost uniquely determined by conclusion. The only exceptions are the choice of the names of new variables (as long as they are new), and the order of the premises and the formulas in the antecedent and succedent, respectively. The names of new variables will be determined by the function *newvars(vars, forbiddenvars)* that generates names based on *vars* that are pairwise different and new with respect to $forbiddenvars$. Other new names can also be chosen. Sometimes a new variable for a given Java type is needed. Then a function *newvar(type, forbiddenvars)* is used

Every rule for expressions has a first premise for the jump case. If the current mode $st[mode]$ is not normal ($st[mode] \neq normal$) the expression is not evaluated, but skipped, i.e. discarded. Some expressions include premises for a *first active use*: class instance or array creation, static field access, static field assignment, and static method invocation. A first active use will create a new goal for a not initialized class state. The remaining goals deal with the normal evaluation of the expression. There are no rules for expression or statement lists. They are handled by the proof rules for those expressions and statements where they occur.

Many proof rules are only applicable when all or some expressions are *basic* expressions. A basic expression is either a local variable access or a literal. This is tested by the predicate is_basic_jexpr. A local variable access contains a logical (algebraic) variable, a literal a first-order (algebraic) term. Both can be selected with the postfix operator .t. .t is omitted in the sequel because it does not help to understand the proof rules. If the proof rule is very close to the semantics of the expression the informal description is kept short. In this case the reader is referred to the semantics for explanations and discussions.

4.3.2 Literal

Semantics: See chapter 3.2.3.

$$\frac{\begin{array}{ll}1. & \Gamma, st[mode] \neq normal \vdash \varphi, \Delta \\ 2. & \Gamma, st[mode] = normal, y = l \vdash \varphi[x/y], \Delta\end{array}}{\Gamma \vdash \langle st/x = l; \rangle\ \varphi, \Delta}$$

l is the literal, x a local variable, y a new variable, the postfix operator $.t$ (to select the term of the literal expression) is omitted. $\varphi[x/y]$ is the replacement of x with y in φ.

Preconditions: (none)

New variable: $y = \text{newvar}(x, \text{vars}(l) \cup (\text{vars}(\varphi) \setminus \{x\}) \cup \text{free}(\Gamma) \cup \text{free}(\Delta))$

The variable y must be new with respect to $\text{vars}(l) \cup (\text{vars}(\varphi) \setminus \{x\}) \cup \text{free}(\Gamma) \cup \text{free}(\Delta)$. In the special case that x occurs in φ, but not in $\text{vars}(l)$ and not in $\text{free}(\Gamma) \cup \text{free}(\Delta)$, x itself is chosen as the 'new' variable (and $\varphi[x/x] \equiv \varphi$). This is indeed correct. (Omitting 'forbidden' variables is a very common error source when designing proof rules.)

4.3.3 Unary operations

There is no special rule for unary operations. They are handled by the literalize rule (chapter 4.5.2).

4.3.4 Cast

Semantics: See chapter 3.2.5. We distinguish between a reference type cast and a primitive type cast. There is no special rule for a primitive type cast, it is handled by the literalize rule (chapter 4.5.2). A reference type cast does not modify the class of an object, but performs only a check (see JLS 15.11.4.10 for an example). If the check fails a `ClassCastException` is thrown. The check is true if the expression e is assignment compatible with ty (JLS 5.5 and 5.2).

$$\frac{\begin{array}{ll}1. & \Gamma \vdash e = null \vee type\exists\ (st[e - _type].type), \Delta \\ 2. & \Gamma, st[mode] \neq normal \vdash \varphi, \Delta \\ 3. & \Gamma, st[mode] = normal, \neg\ (e = null \vee asgcomp_fma) \\ & \quad \vdash \langle st/\texttt{throw new ClassCastException();} \rangle\ \varphi, \Delta \\ 4. & \Gamma, st[mode] = normal, e = null \vee asgcomp_fma \vdash \langle st/x = e; \rangle\ \varphi, \Delta\end{array}}{\Gamma \vdash \langle st/x = (ty)e; \rangle\ \varphi, \Delta}$$

Preconditions: is_basic_jexpr(jexpr) $\wedge$ type$\exists$(ty, tds)

The rule is not applicable unless the preconditions are met. is_basic_jexpr(jexpr) means that e is a basic expression (either a literal or a local variable access). type$\exists$(ty, tds) is true if a class occurring in the cast type ty exists in the type declarations tds or is predefined.

asgcomp_fma is the formula that is true if e is assignment compatible with ty:

$$\text{asgcomp_fma} \equiv \text{e} = \text{null} \vee \text{st[e - _type].type} \leq \text{ty}$$

The definition is similar to the predicates *asgcomp* and $\leq$ (subtype relation) that are used in the semantics rule (see 3.2.5). The important difference is that this *subtype* predicate does not have the type declarations *tds* as an argument for efficiency reasons. The question whether a class is subclass of another is determined by axioms that encode the class hierarchy and that depend on the type declarations. The predicate type$\exists$ is true if a class occurring in the type exists in the type declarations or is predefined. This predicate also does not use the type declarations, but is also axiomatized. It is overloaded with the type$\exists$ predicate from the precondition. See chapter 6 for an extensive discussion. Premise 1 is an additional precondition for an 'equivalence' formula. This means the following formula holds:

$$
\begin{array}{l}
\quad (\text{e} = \text{null} \lor \text{type}\exists(\text{st[e - _type].type})) \\
\rightarrow (\quad \langle \text{st/ x = (ty)e;} \rangle\ \varphi \\
\quad \leftrightarrow (\quad (\text{st[mode]} \neq \text{normal} \rightarrow \varphi) \\
\qquad \land (\quad \text{st[mode]} = \text{normal} \land \neg\ (\text{e} = \text{null} \lor \mathit{asgcomp_fma}) \\
\qquad\quad \rightarrow \langle \text{st/ } \texttt{throw new ClassCastException();} \rangle\ \varphi) \\
\qquad \land (\quad \text{st[mode]} = \text{normal} \land (\text{e} = \text{null} \lor \mathit{asgcomp_fma}) \\
\qquad\quad \rightarrow \langle \text{st/ x = e;} \rangle\ \varphi) \\
\qquad))
\end{array}
$$

4.3.5 Instanceof

Semantics: See chapter 3.2.6.

$$
\frac{\begin{array}{ll}
1. & \Gamma \vdash e = \mathit{null} \lor \mathit{type}\exists\ (\mathit{st}[e - \mathit{_type}].\mathit{type}), \Delta \\
2. & \Gamma, \mathit{st}[\mathit{mode}] \neq \mathit{normal} \vdash \varphi, \Delta \\
3. & \Gamma, \mathit{st}[\mathit{mode}] = \mathit{normal}, \neg\ (e \neq \mathit{null} \land \mathit{asgcomp_fma}) \vdash \langle \mathit{st}/\texttt{x = false;} \rangle\ \varphi, \Delta \\
4. & \Gamma, \mathit{st}[\mathit{mode}] = \mathit{normal}, e \neq \mathit{null} \land \mathit{asgcomp_fma} \vdash \langle \mathit{st}/\texttt{x = true;} \rangle\ \varphi, \Delta
\end{array}}{\Gamma \vdash \langle \mathit{st}/x = e\ \texttt{instanceof}\ \mathit{ty}; \rangle\ \varphi, \Delta}
$$

ty must be reference type, *e* a basic expression that is a reference. The result is true if *e* is not null and assignment compatible to *e*.

Preconditions: is_basic_jexpr(jexpr) ∧ type∃(ty, tds)

The rule is very similar to the rule for a reference type cast above. Premise 1 is an additional precondition; see chapter 6 for a discussion about the class hierarchy.

4.3.6 Conditional Operator

Semantics: See chapter 3.2.7.

$$
\frac{\begin{array}{ll}
1. & \Gamma, \mathit{st}[\mathit{mode}] \neq \mathit{normal} \vdash \varphi, \Delta \\
2. & \Gamma, \mathit{st}[\mathit{mode}] = \mathit{normal}, e_1 \vdash \langle \mathit{st}/x = e_2; \rangle\ \varphi, \Delta \\
3. & \Gamma, \mathit{st}[\mathit{mode}] = \mathit{normal}, \neg\ e_1 \vdash \langle \mathit{st}/x = e_3; \rangle\ \varphi, \Delta
\end{array}}{\Gamma \vdash \langle \mathit{st}/x = (e_1 ? e_2 : e_3); \rangle\ \varphi, \Delta}
$$

e_1 must be a basic expression, e_2 and e_3 can be arbitrary Java expressions.

Preconditions: is_basic_jexpr(e_1)

4.3.7 Conditional Binary Operator

Semantics: See 3.2.8. Conditional binary operators are `&&` and `||`. The right hand operand is evaluated only if the left hand operand does not determine the result of the expression (i.e. if the left side is true for `&&` and false for `||`).

Conditional and:

$$
\frac{\begin{array}{ll}
1. & \Gamma, \mathit{st}[\mathit{mode}] \neq \mathit{normal} \vdash \varphi, \Delta \\
2. & \Gamma, \mathit{st}[\mathit{mode}] = \mathit{normal}, \neg\ e_1 \vdash \langle \mathit{st}/x = \mathit{false}; \rangle\ \varphi, \Delta \\
3. & \Gamma, \mathit{st}[\mathit{mode}] = \mathit{normal}, e_1 \vdash \langle \mathit{st}/x = e_2; \rangle\ \varphi, \Delta
\end{array}}{\Gamma \vdash \langle \mathit{st}/x = e_1 \&\& e_2; \rangle\ \varphi, \Delta}
$$

e_1 must be a basic expression, e_2 can be an arbitrary Java expression.

Preconditions: is_basic_jexpr(e_1)

Conditional or:

$$\frac{\begin{array}{ll}1. & \Gamma, st[mode] \neq normal \vdash \varphi, \Delta \\ 2. & \Gamma, st[mode] = normal, e_1 \vdash \langle st/x = true; \rangle\ \varphi, \Delta \\ 3. & \Gamma, st[mode] = normal, \neg\ e_1 \vdash \langle st/x = e_2; \rangle\ \varphi, \Delta\end{array}}{\Gamma \vdash \langle st/x = e_1 \| e_2; \rangle\ \varphi, \Delta}$$

e_1 must be a basic expression, e_2 can be an arbitrary Java expression.

Preconditions: is_basic_jexpr(e_1)

4.3.8 Simple Binary operators

Semantics: See chapter 3.2.9. There is no special rule for simple binary operators. They are handled by the literalize rule (see chapter 4.5.2, p. 95).

4.3.9 Exception Binary Operator

Semantics: See chapter 3.2.10. Division `/` and remainder `%` throw an `ArithmeticException` if the divisor is 0.

$$\frac{\begin{array}{ll}1. & \Gamma, st[mode] \neq normal \vdash \varphi, \Delta \\ 2. & \Gamma, st[mode] = normal, sb2int(e_2) = 0 \vdash \langle st/\texttt{throw new ArithmeticException();} \rangle\ \varphi, \Delta \\ 3. & \Gamma, st[mode] = normal, sb2int(e_2) \neq 0 \vdash \langle st/x = literal(sb2int(e_1) \oplus sb2int(e_2)); \rangle\ \varphi, \Delta\end{array}}{\Gamma \vdash \langle st/x = e_1 \oplus e_2; \rangle\ \varphi, \Delta}$$

e_1, e_2 must be basic Java expressions that are either integers, shorts, or bytes. *Primitive type correctness* guarantees that this is indeed the case (see chapter 5). Shorts and bytes must be converted to integers before the operation is applied. This is done by *sb2int*. Then the corresponding algebraic operation (division or remainder) is applied on the values, and this term becomes the value of a literal expression. This shows why literals can contain arbitrary algebraic terms. Of course, the algebraic integer division and remainder operations must be specified as required by Java.

Preconditions: is_basic_jexpr(e_1) $\wedge$ is_basic_jexpr(e_2)

4.3.10 Local Variable Access

Semantics: See chapter 3.2.11.

$$\frac{\begin{array}{ll}1. & \Gamma, st[mode] \neq normal \vdash \varphi, \Delta \\ 2. & \Gamma, st[mode] = normal, y = z \vdash \varphi[x/y], \Delta\end{array}}{\Gamma \vdash \langle st/x = z; \rangle\ \varphi, \Delta}$$

z is the (local) variable that is accessed, y a new variable.

Preconditions: (none)

New variable: y = newvar(x, $\{z\} \cup (\text{vars}(\varphi) \setminus \{x\}) \cup \text{free}(\Gamma) \cup \text{free}(\Delta)$)

4.3.11 Static Field Access

Semantics: See chapter 3.2.12.

$$\frac{\begin{array}{ll}1. & \Gamma, st[mode] \neq normal \vdash \varphi, \Delta \\ 2. & \Gamma, st[mode] = normal, \neg\ initdoneP(c, st) \\ & \quad \vdash \langle st/\texttt{static}(c) \rangle\ \langle st/x = c.f; \rangle\ \varphi, \Delta \\ 3. & \Gamma, st[mode] = normal, initdoneP(c, st), y = coerce(st[jvmref - c.f], ty) \vdash \varphi[x/y], \Delta\end{array}}{\Gamma \vdash \langle st/x = c.f; \rangle\ \varphi, \Delta}$$

c is the class name of the static field, f the field name, y a new variable.

Preconditions: (none)

New variable: $y = \text{newvar}(x, (\text{vars}(\varphi) \setminus \{x\}) \cup \text{free}(\Gamma) \cup \text{free}(\Delta))$

The second premise is for a *first active use*. The value retrieved from the store st[*jvmref* - c.f] must be coerced to its expected result type *ty* (the static type of c.f) as prescribed by the semantics. See chapter 5.

4.3.12 Instance Field Access

Semantics: See chapter 3.2.13.

$$\frac{\begin{array}{ll}1. & \Gamma, st[mode] \neq normal \vdash \varphi, \Delta \\ 2. & \Gamma, st[mode] = normal, e = null \vdash \langle st/\texttt{throw new NullPointerException();}\rangle\ \varphi, \Delta \\ 3. & \Gamma, st[mode] = normal, e \neq null, y = coerce(st[e - f], ty) \vdash \varphi[x/y], \Delta\end{array}}{\Gamma \vdash \langle st/x = e.f;\rangle\ \varphi, \Delta}$$

e must be a basic expression, y is a new variable. f is the field specification.

Preconditions: is_basic_expr(e)

New variable: $y = \text{newvar}(x, \text{vars}(e) \cup (\text{vars}(\varphi) \setminus \{x\}) \cup \text{free}(\Gamma) \cup \text{free}(\Delta))$

4.3.13 Array Access

Semantics: See chapter 3.2.14. Both arguments are evaluated, then the first argument is checked to be not null (otherwise a `NullPointerException` is thrown), then the index is checked (and possibly an `ArrayIndexOutOfBoundsException` thrown).

$$\frac{\begin{array}{ll}1. & \Gamma, st[mode] \neq normal \vdash \varphi, \Delta \\ 2. & \Gamma, st[mode] = normal, e_1 = null \\ & \vdash \langle st/\texttt{throw new NullPointerException();}\rangle\ \varphi, \Delta \\ 3. & \Gamma, st[mode] = normal, e_1 \neq null, \neg\ (0 \leq sb2int(e_2) \wedge sb2int(e_2) < st[e_1 - _length]) \\ & \vdash \langle st/\texttt{throw new ArrayIndexOutOfBoundsException();}\rangle\ \varphi, \Delta \\ 4. & \Gamma, st[mode] = normal, e_1 \neq null, 0 \leq sb2int(e_2) \wedge sb2int(e_2) < st[e_1 - _length], \\ & y = coerce(st[e_1 - sb2int(e_2)], ty) \\ & \vdash \varphi[x/y], \Delta\end{array}}{\Gamma \vdash \langle st/x = e_1[e_2];\rangle\ \varphi, \Delta}$$

e_1 and e_2 must be basic expressions, y is a new variable. The index may be an integer, a short, or a byte; it must be converted into an integer with *sb2int*.

Preconditions: is_basic_expr(e_1) $\wedge$ is_basic_expr(e_2)

New variable: $y = \text{newvar}(x, \text{vars}(e_1) \cup \text{vars}(e_2) \cup (\text{vars}(\varphi) \setminus \{x\}) \cup \text{free}(\Gamma) \cup \text{free}(\Delta))$

4.3.14 Local Variable Assignment

Semantics: See chapter 3.2.15.

$$\frac{1.\quad \Gamma \vdash \langle st/y = e;\rangle\ \langle st/x = y;\rangle\ \varphi, \Delta}{\Gamma \vdash \langle st/x = (y = e);\rangle\ \varphi, \Delta}$$

e can be an arbitrary expression. The rule has actually two assignments because $x =$ is used to capture the result of the expression as explained in chapter 4.3.1.

Preconditions: (none)

4.3.15 Static Field Assignment

Semantics: See chapter 3.2.16.

$$\frac{\begin{array}{ll}1. & \Gamma, st[mode] \neq normal \vdash \varphi, \Delta \\ 2. & \Gamma, st[mode] = normal, \neg\, initdoneP(c, st) \\ & \quad \vdash \langle st/\texttt{static}(c)\rangle\ \langle st/x = (c.f = e);\rangle\ \varphi, \Delta \\ 3. & \Gamma, st[mode] = normal, initdoneP(c, st), st_0 = st[jvmref - c.f, e] \\ & \quad \vdash \langle st_0/x = e;\rangle\ \varphi[st/st_0], \Delta\end{array}}{\Gamma \vdash \langle st/x = (c.f = e);\rangle\ \varphi, \Delta}$$

e must be a basic expression, st_0 is a new variable. c is the class of the static field, f the field name. $st[jvmref - c.f, e]$ modifies the store and sets the key $jvmref - c.f$ to the new value e. As in the previous rule the conclusion contain two assignments. x is used to capture the result of the expression. This is the first rule that modifies the store and introduces a new variable for the modified store. The second premise is for a *first active use*.

Preconditions: is_basic_expr(e)

Variables: $st_0 = \text{newvar}(st, \{st\} \cup \text{vars}(e) \cup \text{vars}(\varphi) \cup \text{free}(\Gamma) \cup \text{free}(\Delta))$

The set of 'forbidden' variables cannot be made smaller: st_0 must be different from st (otherwise the incorrect formula $st = st[jvmref - c.f, e]$ would be generated), and the expression e could contain a variable of sort *store* in a literal (literals can contain arbitrary algebraic terms, including variables of any sort). All other variables occurring in an expression must be of a valid Java type.

4.3.16 Instance Field Assignment

Semantics: See chapter 3.2.17. JLS 15.26.1 states that the field access is evaluated first $e.f$, then e_0. However, this means that a `NullPointerException` (if $e = null$) is thrown before e_0 is evaluated. This is wrong. Both arguments are evaluated, then e is checked to be not null.

$$\frac{\begin{array}{ll}1. & \Gamma, st[mode] \neq normal \vdash \varphi, \Delta \\ 2. & \Gamma, st[mode] = normal, e = null \vdash \langle st/\texttt{throw new NullPointerException();}\rangle\ \varphi, \Delta \\ 3. & \Gamma, st[mode] = normal, e \neq null, st_0 = st[e - f, e_0] \vdash \langle st_0/x = e_0;\rangle\ \varphi[st/st_0], \Delta\end{array}}{\Gamma \vdash \langle st/x = (e.f = e_0);\rangle\ \varphi, \Delta}$$

e, e_0 must be basic expressions, st_0 is a new variable.

Preconditions: is_basic_expr(e) $\wedge$ is_basic_expr(e_0)

Variables: $st_0 = \text{newvar}(st, \{st\} \cup \text{vars}(x = (e.f = e_0)) \cup \text{vars}(\varphi) \cup \text{free}(\Gamma) \cup \text{free}(\Delta))$

4.3.17 Array Assignment

Semantics: See chapter 3.2.18. Because the run-time type of the assigned value must be checked this rule has six premises. The first premise is an additional precondition for the 'equivalence' formula (similiar to the reference cast in chapter 4.3.4).

$$\frac{\begin{array}{ll}1. & \Gamma \vdash typeex_fma, \Delta \\ 2. & \Gamma, st[mode] \neq normal \vdash \varphi, \Delta \\ 3. & \Gamma, st[mode] = normal, e_0 = null \\ & \quad \vdash \langle st/\texttt{throw new NullPointerException();}\rangle\ \varphi, \Delta \\ 4. & \Gamma, st[mode] = normal, e_0 \neq null, \neg\, (0 \leq sb2int(e_1) \wedge sb2int(e_1) < st[e_0 - _length]) \\ & \quad \vdash \langle st/\texttt{throw new ArrayIndexOutOfBoundsException();}\rangle\ \varphi, \Delta \\ 5. & \Gamma, st[mode] = normal, e_0 \neq null, 0 \leq sb2int(e_1) \wedge sb2int(e_1) < st[e_0 - _length], \\ & \neg\, arrayasgcomp_fma \vdash \langle st/\texttt{throw new ArrayStoreException();}\rangle\ \varphi, \Delta \\ 6. & \Gamma, st[mode] = normal, e_0 \neq null, 0 \leq sb2int(e_1) \wedge sb2int(e_1) < st[e_0 - _length], \\ & arrayasgcomp_fma, st_0 = st[e_0 - sb2int(e_1), e_2] \vdash \langle st_0/x = e_2;\rangle\ \varphi[st/st_0], \Delta\end{array}}{\Gamma \vdash \langle st/x = (e_0[e_1] = e_2);\rangle\ \varphi, \Delta}$$

e_0, e_1, e_2 must be either literals or local variables, st_0 is a new variable. The index can be an integer, a short, or a byte. The rule looks surprisingly complex for a simple array assignment, but compare it to the description in JLS 15.25.1.

The element type must be checked only for reference types. If the assigned value e_2 has a primitive type an `ArrayStoreException` cannot occur. Otherwise *arrayasgcomp_fma* checks if the run-time type of e_2 is assignment compatible to the run-time element type of e_0. *typeex_fma* checks if the type of e_1 exists:

1. If e_2 is not a reference, then *typeex_fma* $\equiv$ *arrayasgcomp_fma* $\equiv$ true. Otherwise:

2. *typeex_fma* $\equiv$ $(e_0 = null \lor \text{type}\exists\ (\text{st}[e_0 - \text{_type}].\text{type}.\text{type})) \land (e_2 = null \lor \text{type}\exists\ (\text{st}[e_2 - \text{_type}].\text{type}))$

3. *arrayasgcomp_fma* $\equiv e_2 = null \lor \text{st}[e_2 - \text{_type}].\text{type} \leq \text{st}[e_0 - \text{_type}].\text{type}.\text{type}$

   $\text{st}[e_2 - \text{_type}].\text{type}$ is the run-time type of e_2, $\text{st}[e_0 - \text{_type}].\text{type}$ the run-time type of the array. The additional selector .type selects the immediate element type of the array (see the array semantics in chapter 3.2.18).

If e_2 is not a reference the first and fifth premise are trivially true.

Preconditions: $\text{is_basic_expr}(e_0) \land \text{is_basic_expr}(e_1) \land \text{is_basic_expr}(e_2)$

Variables: $st_0 = \text{newvar}(st, \{st\} \cup \text{vars}(x = (e_0[e_1] = e_2)) \cup \text{vars}(\varphi) \cup \text{free}(\Gamma) \cup \text{free}(\Delta))$

It is easy to forget the case that the assigned value may be null (JLS 15.26.1 never mentions this, cf. 'Let RC be the class of the object referred to by the value of the right-hand operand at run time.')

4.3.18 Prefix/Postfix Increment/Decrement Operators

Semantics: See chapter 3.2.19. There are four operations: prefix increment (`++ e`), prefix decrement (`-- e`), postfix increment (`e ++`), and postfix decrement (`e --`). The expression e must be a variable access, i.e. either a local variable access, a static field access, an instance field access, or an array access. The type of the expression can be either **int**, or **short**, or **byte**. The result is automatically cast back to the type of e. This is one of two situations in Java where an implicit primitive conversion that may change the value takes place.

For a postfix operation we have

$$\frac{1.\quad \Gamma \vdash \langle st/y = e;\rangle\ \langle st/e = eval(\oplus, y);\rangle\ \langle st/x = y;\rangle\ \varphi, \Delta}{\Gamma \vdash \langle st/x = e\oplus;\rangle\ \varphi, \Delta}$$

e must be an access; its argument expressions must be basic expressions. The evaluation function *eval* takes care of the cast. y must be a new variable. It is needed in case of a variable conflict like `x = x++;` or `x = a[x]++;`. The assignment to `x` must occur after the side effect took place!

Preconditions: basic_exprs(subexprs(e))

Variables: $y = \text{newvar}(x, \text{vars}(e) \cup \text{free}(\varphi) \cup \text{free}(\Gamma) \cup \text{free}(\Delta))$

For a prefix operation we have

$$\frac{1.\quad \Gamma \vdash \langle st/y = e;\rangle\ \langle st/e = eval(\oplus, y);\rangle\ \langle st/x = eval(\oplus, y);\rangle\ \varphi, \Delta}{\Gamma \vdash \langle st/x = \oplus e;\rangle\ \varphi, \Delta}$$

Preconditions: basic_exprs(subexprs(e))

Variables: $y = \text{newvar}(x, \text{vars}(e) \cup \text{free}(\varphi) \cup \text{free}(\Gamma) \cup \text{free}(\Delta))$

4.3.19 Compound Assignment

Semantics: See chapter 3.2.20. The compound assignment $\oplus=$ is quite complicated. The left hand side of the assignment must be an access, either a local variable access, a static field access, an instance field access, or an array access. All types must be primitive (**boolean**, **int**, **short**, or **byte**). For numerical types the result is automatically cast back to the type of the left hand side. This is the second situation where an implicit primitive conversion occurs that may change a value. If the operation is division or remainder an `ArithmeticException` may occur.

$$\frac{\begin{array}{ll}1. & \Gamma, st[mode] \neq normal \vdash \varphi, \Delta \\ 2. & \Gamma, st[mode] = normal \vdash \langle st/x = (e_1 = (ty)(e_1 \oplus e_2)); \rangle\ \varphi, \Delta\end{array}}{\Gamma \vdash \langle st/x = (e_1 \oplus= e_2); \rangle\ \varphi, \Delta}$$

e_1 must be an access expression that contains only basic subexpressions, e_2 can be any expression. The cast type ty is the type of e_1.

Preconditions: basic_exprs(subexprs(e_1))

The presentation is a little simplified since the compound operation must be converted into a binary operation, and either a *binary expression* or an *exception binary expression* must be generated.

4.3.20 Array Creation

Semantics: See chapter 3.2.22. As noted in the semantic chapter we follow JLS (second edition) where array creation never is a *first active use*. We make a difference between one-dimensional and multi-dimensional arrays. First the version for one-dimensional arrays:

$$\frac{\begin{array}{ll}1. & \Gamma, st[mode] \neq normal \vdash \varphi, \Delta \\ 2. & \Gamma, st[mode] = normal, sb2int(e) < 0 \\ & \quad \vdash \langle st/\texttt{throw new NegativeArraySizeException();} \rangle\ \varphi, \Delta \\ 3. & \Gamma, st[mode] = normal, \neg\ sb2int(e) < 0, st_0 = addarray(y, ty, e, st), y = newref(st) \\ & \quad \vdash \langle st_0/x = y; \rangle\ \varphi[st/st_0], \Delta\end{array}}{\Gamma \vdash \langle st/x = \texttt{new}\ ty[e] \rangle\ \varphi, \Delta}$$

e must be a basic expression. It can be an integer, a short, or a byte value. $addarray(y, ty, e, st)$ creates the array of length e in the store and initializes the indices with the correct default values for the type ty (see chapter 3.1.3). y is a new variable, *newref*(st) computes a new reference.

Preconditions: is_basic_expr(e)

Variables: st_0 and y must be new:

$$st_0 + y = \text{newvars}(st + x, \{st, x\} \cup \text{vars}(e) \cup vars(\varphi) \cup \text{free}(\Gamma) \cup \text{free}(\Delta))$$

newvars returns new (and mutually different) variables for its first argument, in this case the list with the two variables st, x.

The multi-dimensional version:

$$\frac{\begin{array}{ll}1. & \Gamma, st[mode] \neq normal \vdash \varphi, \Delta \\ 2. & \Gamma, st[mode] = normal, sb2int(e_1) < 0 \vee\ \ldots\ \vee\ sb2int(e_m) < 0 \\ & \quad \vdash \langle st/\texttt{throw new NegativeArraySizeException();} \rangle\ \varphi, \Delta \\ 3. & \Gamma, st[mode] = normal, \neg\ (sb2int(e_1) < 0 \vee\ \ldots\ \vee\ sb2int(e_m) < 0), \\ & \quad st_0 = addarrays_fma(e_1 \ldots e_m, ty, n, st), y = newrefs(e_1 \ldots e_m, st).first \\ & \quad \vdash \langle st_0/x = y; \rangle\ \varphi[st/st_0], \Delta\end{array}}{\Gamma \vdash \langle st/x = \texttt{new}\ ty[e_1] \ldots [e_m][n] \rangle\ \varphi, \Delta}$$

Multi-dimensional arrays are very complicated. The dimension expressions $e_1, \ldots, e_n$ must be non-negative numbers. $y = newrefs(e_1 \ldots e_m, st).first$ is the top-level reference to the first new array. *addarrays_fma* creates the array in the store. It uses the same function that is used in the semantics definition.

Preconditions: basic_exprs$(e_1, \ldots, e_m) \wedge (\mathrm{m} \neq 1 \vee \mathrm{n} \neq 0)$ (this ensures that the array is not one-dimensional)

Variables: st_0 and y must be new:

$$st_0 + y = \mathrm{newvars}(st + x, \{st, x\} \cup \mathrm{vars}(e_1 \ldots e_m) \cup vars(\varphi) \cup \mathrm{free}(\Gamma) \cup \mathrm{free}(\Delta))$$

It may be noted that the Java compiler accepts `new byte[3][0][2][][]`, i.e. a zero dimension followed by non-zero dimensions. This is identical to `new byte[3][0][][][]`.

4.3.21 Array Initializer

Semantics: See chapter 3.2.23.

$$\frac{\begin{array}{ll} 1. & \Gamma, st[mode] \neq normal \vdash \varphi, \Delta \\ 2. & \Gamma, st[mode] = normal, y = newref(st), st_0 = addarray(y, ty, e_1 + \ldots + e_n, st) \\ & \vdash \langle st_0/x = y; \rangle\ \varphi[st/st_0], \Delta \end{array}}{\Gamma \vdash \langle st/x = \{e_1, \ldots, e_n\}; \rangle\ \varphi, \Delta}$$

The expressions $e_1, \ldots, e_n$ must be basic expressions. y and st_0 are new variables, ty is the element type of the array. $newref(st)$ computes a new reference for the store st. *addarray* is a function on the store that does everything necessary to add an array with initial values to the store. This means that the keys $y - i$ for $0 \leq i < n$ are added with their correct values, and that a *type* and *length* field is added for the reference y.

Preconditions: basic_exprs$(e_1, \ldots, e_n)$

Variables: st_0 and y must be new:

$$st_0 + y = \mathrm{newvars}(st + x, \{st, x\} \cup \mathrm{vars}(e_1 \ldots e_n) \cup vars(\varphi) \cup \mathrm{free}(\Gamma) \cup \mathrm{free}(\Delta))$$

4.3.22 Class Instance Creation

Semantics: See chapter 3.2.21. The rule is very similar to the semantics.

$$\frac{\begin{array}{ll} 1. & \Gamma, st[mode] \neq normal \vdash \varphi, \Delta \\ 2. & \Gamma, st[mode] = normal, \neg\ initdoneP(c, st) \\ & \vdash \langle st/\texttt{static}(c) \rangle\ \langle st/x = \texttt{new}\ c(e_1, \ldots, e_n) \rangle\ \varphi, \Delta \\ 3. & \Gamma, st[mode] = normal, initdoneP(c, st), st_0 = addobj(y, c, ifields, st), y = newref(st) \\ & \vdash \langle st_0/x = y.c(e_1, \ldots, e_n) \rangle\ \varphi[st/st_0], \Delta \end{array}}{\Gamma \vdash \langle st/x = \texttt{new}\ c(e_1, \ldots, e_n); \rangle\ \varphi, \Delta}$$

c is a class name, $e_1, ..., e_n$ may be arbitrary expressions. A class instance creation may be a *first active use*. Premise 2 is for a first active use. Premise 3 deals with the case that the class is already initialized. The `new` expression is simply transformed into an explicit constructor call. y is a new reference for the object, *addobj* adds the instance fields of the object with their default values to the store (see chapter 3.1.3). The object is created before the arguments $e_1, ..., e_n$ are evaluated. This is reflected in the proof rule by the fact that they do not have to be *basic* expressions.

Preconditions: class∃+(c, tds) (the class must be predefined or exist in the type declarations)

Variables: st_0 and y must be new:

$$st_0 + y = \mathrm{newvars}(st + x, \{st, x\} \cup \mathrm{vars}(e_1 \ldots e_n) \cup vars(\varphi) \cup \mathrm{free}(\Gamma) \cup \mathrm{free}(\Delta))$$

4.3.23 Explicit Constructor Invocation

Semantics: See chapter 3.2.24. An explicit constructor invocation can occur after the *new* rule (i.e. the rule for a class instance creation expression) was applied, or at the beginning of a constructor. (A `this`$(e_1, \ldots, e_n)$ call in class c is transformed into this.$c(e_1, \ldots, e_n)$, a `super`$(e_1, \ldots, e_n)$ call is transformed into this.$s(e_1, \ldots, e_n)$ if s is the superclass of c.)

$$\frac{\begin{array}{ll} 1. & \Gamma, st[mode] \neq normal \vdash \varphi, \Delta \\ 2. & \Gamma, st[mode] = normal, vars = es, y = e, z = e \\ & \vdash \langle st/\alpha[args/vars][this/y]\rangle\ \langle st/\texttt{target}(return)\rangle\ \langle st/x = z;\rangle\ \varphi, \Delta \end{array}}{\Gamma \vdash \langle st/x = e.c(es);\rangle\ \varphi, \Delta}$$

Explanations:

- es is the list of arguments $e_1, \ldots, e_n$ of the constructor call c. e and es must be basic expressions. e is a reference to the object on which the constructor is invoked.
- $vars$ are new variables of the same length as es and of the same sorts, y and z are also new variables.
- getConstr(c, tys, tds) selects the correct constructor declaration. This function is described in detail in the semantics of the constructor call (chapter 3.2.24). The result is either the matching declaration with body α or a dummy `native c() { }` . tys are the formal argument types of the correct constructor declaration as determined by the compiler, they are part of the constructor call.
- α is the body of the constructor, α = getConstr(c, tys, tds).body, $args$ are the formal parameter variables of the constructor declaration ($params$ is a list of pairs of type and variable, $params = (ty_1\ x_1, ty_2\ x_2, \ldots, ty_n\ x_n)$, so $args = x_1, \ldots, x_n$.
- $args$ and `this` are replaced by the new variables in the constructor body.
- The constructor body may or may not end with `return`, therefore a `target`($return$) statement is added that will catch a return.
- The result of the constructor call is always the value of the invoking expression. Therefore we add another new variable z – and an assignment – in case the constructor body makes an (illegal) assignment to `this`.

Preconditions: is_basic_jexpr(e) $\wedge$ basic_exprs(es) $\wedge$ implemented(getConstr(c, tys, tds))

The last condition implies that the rule is not applicable for `native` constructors.

Variables: $vars, y, z$ must be new:

$$y + z + vars = \text{newvars}(\texttt{this} + \texttt{this} + args, \{\texttt{this}, s\} \cup \text{vars}(x = e.c(es)) \cup \text{vars}(\text{getConstr(c, tys, tds)}) \cup \text{vars}(\varphi) \cup \text{free}(\Gamma) \cup \text{free}(\Delta))$$

The important additional precondition is implemented(getConstr(c, tys, tds)). This means the constructor is not *native*, and a matching declaration was really found. Together with the generic requirements of *primitive type correctness* (see chapter 5) ensures that

1. the constructor contains a meaningful body (the body of a native constructor is an empty block)
2. the formal parameter types $ty_1, \ldots, ty_n$ are equal to tys, and that the length and types of the actual arguments es fit, so that the equations $vars = es$ are well sorted.

For a fully type correct constructor call and fully type correct type declarations without native constructors it is guaranteed that always implemented(getConstr(c, tys, tds)). See chapter 5.4 for an extensive discussion.

4.3.24 Static Method Invocation

Semantics: See chapter 3.2.25. The semantics distinguishes between five different kinds of method invocation: static, virtual, nonvirtual, super, virtual and interface. An interface method cannot be called, i.e. there is no proof rule for this invocation mode. The invocation modes nonvirtual and super are very similar, while the other two modes differ significantly in their behavior. For clarity we describe three different proof rules for method invocation: first static, then nonvirtual and super, and finally virtual. It is of course possible to design only one proof rule that generates its premises depending on the mode.

First we describe the static method invocation. It may cause a class initialization if it is a *first active use*, otherwise the method body is evaluated.

$$\frac{\begin{array}{ll}1. & \Gamma, st[mode] \neq normal \vdash \varphi, \Delta \\ 2. & \Gamma, st[mode] = normal, \neg\, initdoneP(c, st) \\ & \quad \vdash \langle st/\texttt{static}(c)\rangle\ \langle st/x = e.m(es);\rangle\ \varphi, \Delta \\ 3. & \Gamma, st[mode] = normal, initdoneP(c, st), vars = es \\ & \quad \vdash \langle st/\alpha[args/vars]\rangle\ \langle st/\texttt{targetexpr}(x, ty)\rangle\ \varphi, \Delta\end{array}}{\Gamma \vdash \langle st/x = e.m(es);\rangle\ \varphi, \Delta}$$

α is the method body, $args$ its formal parameters variables, $vars$ new variables. e and es must be basic expressions. `targetexpr`(x,ty) catches a return and assigns x to the returned value. This is the version for a non `void` method that returns a result. The method call is similar to the constructor call, except that `this` is not bound. A static method may have an invoking expression that is evaluated, but its result is discarded. A first active use occurs *after* e (and the arguments es) have been evaluated.

Preconditions: is_basic_jexpr(e) $\wedge$ basic_exprs(es)
$\wedge$ implemented(getMethod(m, invmo, tys, ty, tds))

Variables: $vars$ must be new:

$$vars = \text{newvars}(args, \{s\} \cup \text{vars}(x = e.m(es)) \cup \text{vars}(\text{getMethod(m, invmo, tys, ty, tds)}) \cup \text{vars}(\varphi) \cup \text{free}(\Gamma) \cup \text{free}(\Delta))$$

For a fully type correct method call and fully type correct type declarations it is guaranteed that implemented(getMethod(m, invmo, tys, ty, tds)) always holds (analogous to the constructor call described above). For a method invocation without assignment (i.e. $\langle st/e.m(es);\rangle\ \varphi$) the proof rule is modified to `target`(*return*) instead of `targetexpr`(x). `target`(*return*) catches return, but discards a possibly returned value. (A void method may end normally or with a return (without value), a non-`void` method must always return a value; `target`(*return*) handles all three cases.)

4.3.25 Nonvirtual or Super Method Invocation

Semantics: See chapter 3.2.25. The distinguishing features of these two invocation modes are: they cannot cause a *first active use*; the invoking expression may not be null; a dynamic method lookup is performed, but the initial class to search is statically determined at compile time.

$$\frac{\begin{array}{ll}1. & \Gamma, st[mode] \neq normal \vdash \varphi, \Delta \\ 2. & \Gamma, st[mode] = normal, e = null \\ & \quad \vdash \langle st/\texttt{throw new NullPointerException();}\rangle\ \varphi, \Delta \\ 3. & \Gamma, st[mode] = normal, e \neq null, vars = es, y = e \\ & \quad \vdash \langle st/\alpha[args/vars][this/y]\rangle\ \langle st/\texttt{targetexpr}(x, ty)\rangle\ \varphi, \Delta\end{array}}{\Gamma \vdash \langle st/x = e.m(es);\rangle\ \varphi, \Delta}$$

α is the method body, $args$ its formal parameters, $vars$ new variables. e and es must be basic expressions. `targetexpr`(x, ty) catches a return and assigns x to the returned value. This is again the version for a non-`void` method.

Preconditions: is_basic_jexpr(e) ∧ basic_exprs(es) ∧ implemented(getMethod(m, invmo, tys, ty, tds))

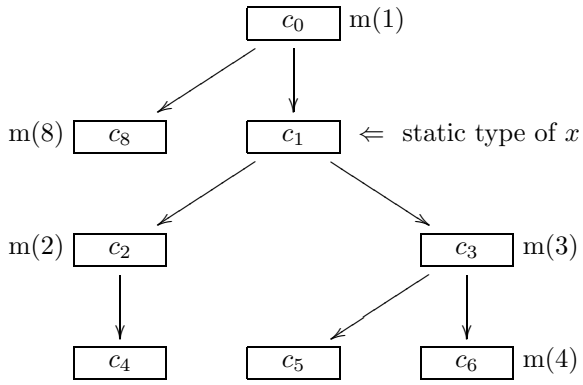
Variables: y and $vars$ must be new:

$$y + vars = \text{newvars}(\texttt{this} + args, \{s\} \cup \text{vars}(x = e.m(es)) \cup \text{vars}(\text{getMethod(m, invmo, tys, ty, tds)}) \cup \text{vars}(\varphi) \cup \text{free}(\Gamma) \cup \text{free}(\Delta))$$

For a fully type correct method call and fully type correct type declarations it is guaranteed that implemented(getMethod(m, invmo, tys, ty, tds)) always holds (similar to the static method invocation described above, though the method lookup works differently). For a method invocation without assignment (i.e. $\langle st/e.m(es);\rangle\ \varphi$) the proof rule is modified to `target`(*return*) instead of `targetexpr`(x, ty). `target`(*return*) catches return, but discards a possibly returned value. (A void method may end normally or with a return (without value), a non-void method must always return a value; `target`(*return*) handles all three cases.)

4.3.26 Instance Method Invocation

Semantics: See chapter 3.2.25. The proof rule for a virtual invocation mode is probably the most complex of all proof rules. The main features of this invocation mode are: It cannot cause a first active use; the invoking expression may not be null; dynamic method lookup is used, and the initial search class depends on the run-time type of the invoking expression/object. This means especially that the search class cannot be determined syntactically. We first illustrate the idea for the proof rule with an example, and afterwards describe the general form of the rule. Consider the following class hierarchy:



Let $e.m(e_1, \ldots, e_n)$ the method invocation, c_1 the static class type of e as determined at compile time. For a correct Java program either c_1 or one of its super classes contains a method m that is *applicable* and *accessible* (in the example, $m(1)$ in class c_0). During execution the run-time class R of e is c_1 or one of its subclasses (because of Java's type safety, but only if the store is compatible to the program). For example, if the class is c_5 a dynamic method lookup occurs that finds the correct method $m(3)$ in class c_3. For all possible run-time classes the following method declarations will be found: R = c_4 or R = c_2 ⇒ m(2), R = c_5 or R = c_3 ⇒ m(3), R = c_6 ⇒ m(4), R = c_1 ⇒ m(1). If the store is not compatible to the program (or the program not fully type correct) then R may not be a subclass of c_1, but, for example, c_8. The semantics defines that in this case the method declaration m(8) is used, because the dynamic method lookup finds m(8). This means that these cases must be considered if we want to have a calculus that is complete even if a compatible store or fully type correct program is not assumed. On the other hand, we are not really interested in those programs. Furthermore, experience shows that usually the run-time type of the invoking expression is known, and a subtype of the static type. Therefore we propose a flexible proof rule that works as follows: The user selects a set of classes. Then one premise will be to prove that the run-time class of the invoker is one of these classes. And

for every class a premise is generated that contains the correct method body as determined by dynamic method lookup for that class. The chosen set of classes must have the property that all classes are subclasses of the static class, and that dynamic method lookup yields an implemented (not native) method declaration (a native method has no method body). This can be checked syntactically. Two other premises are for jumps, and for a NullpointerException if the invoker is null. For example, if the selected classes are $\{c_1, c_2, c_4\}$ the following premises are generated:

$$\frac{\begin{array}{ll}1. & \Gamma, st[mode] \neq normal \vdash \varphi, \Delta \\ 2. & \Gamma, e = null, st[mode] = normal \vdash \langle st/\texttt{throw new NPE();}\rangle\ \varphi, \Delta \\ 3. & \Gamma, e \neq null, st[mode] = normal \vdash \mathit{classOf}(e, st) \in \{c_1, c_2, c_4\}, \Delta \\ 4. & \Gamma, e \neq null, st[mode] = normal, \mathit{classOf}(e, st) = c_1, \\ & y = e, vars = es \vdash \langle st/\alpha_1[args_1/vars][this/y]\rangle\ \langle st/\texttt{targetexpr}(x, ty)\rangle\ \varphi, \Delta \\ 5. & \Gamma, e \neq null, st[mode] = normal, \mathit{classOf}(e, st) = c_2, \\ & y = e, vars = es \vdash \langle st/\alpha_2[args_2/vars][this/y]\rangle\ \langle st/\texttt{targetexpr}(x, ty)\rangle\ \varphi, \Delta \\ 6. & \Gamma, e \neq null, st[mode] = normal, \mathit{classOf}(e, st) = c_4, \\ & y = e, vars = es \vdash \langle st/\alpha_2[args_2/vars][this/y]\rangle\ \langle st/\texttt{targetexpr}(x, ty)\rangle\ \varphi, \Delta\end{array}}{\Gamma \vdash \langle st/x = e.m(es)\rangle\ \varphi, \Delta}$$

classOf looks up the class of the reference e in the store, $\mathit{classOf}(e, st)$ = st[e – _type].type.class. α_i is the corresponding method body for $m(i)$, $args_i$ the formal parameter variables of $m(i)$. The formal parameters and `this` are replaced by new variables $vars, y$. Return statements in α_i are 'caught' by `targetexpr`(x,ty), and assigned to x.

An obvious optimization is to merge premises 5. and 6. into one, because they have the same method body:

$$\begin{array}{ll}5. & \Gamma, e \neq null, st[mode] = normal, \mathit{classOf}(e, st) = c_2 \vee \mathit{classOf}(e, st) = c_4, \\ & y = e, vars = es \vdash \langle st/\alpha_2[args_2/vars][this/y]\rangle\ \langle st/\texttt{targetexpr}(x, ty)\rangle\ \varphi, \Delta\end{array}$$

The rule in KIV works as follows: The system computes all possible classes, i.e. all classes that are subclasses of the static type, and where dynamic method lookup returns a non-native method declaration. Furthermore, the system tries to simplify the expression *classOf*(e,st) in the context of the sequent. If the expression can be simplified to a concrete class then the run-time class of the invoker is uniquely determined. The set of applicable classes and the result of the simplified expression are presented to the user. Then the user selects a subset of the applicable classes. Experience shows that the proof rule is typically used in one of three ways:

1. The system is able to determine the run-time class. Then the user selects only this class.
2. The system cannot determine the run-time class, but the user knows it is unique. Then the user also selects only this class (and will have to prove that the run-time class is indeed the selected class).
3. The system cannot determine the run-time class, and the user does not care and selects all classes.

To summarize, the instance method invocation rule is very flexible, restricted to useful cases, therefore incomplete with respect to an incompatible store or a not fully type correct program, but complete with respect to a compatible store and a fully type correct program (if all applicable classes are chosen). Poetzsch-Heffter and Müller [PHM99] require a user to prove a property for all possible subclasses of the invoker's static type. von Oheimb [vO01] has only one premise for the method body, but quantifies over the run-time class and correct method declaration. This is essentially the third kind of usage of this proof rule, except that the case distinction for the different method bodies can be delayed. Experience shows that for Java Card application the run-time class is always uniquely determined; this happens for most examples. Chapter 7.4 contains a non-Java Card example where two different method bodies are considered; they are so different in their behaviour that the case distinction must be done immediately. It is highly doubtful that

useful examples exist where many different method bodies are available at all; and a property should be proved for all of them at once. Anyway, the calculus presented in this work allows to formulate arbitrary properties with arbitrary assumptions (e.g. about the run-time class) as lemmas, and to use them in other proofs.

The general form of the rule is

$$\frac{\begin{array}{ll}
1. & \Gamma, st[mode] \neq normal \vdash \varphi, \Delta \\
2. & \Gamma, e = null, st[mode] = normal \vdash \langle st/\texttt{throw new NPE();}\rangle\ \varphi, \Delta \\
3. & \Gamma, e \neq null, st[mode] = normal \vdash classOf(e, st) \in \{c_1, \ldots, c_n\}, \Delta \\
4. & \Gamma, e \neq null, st[mode] = normal, classOf(e, st) = c_1, \\
 & y = e, vars = es \vdash \langle st/\alpha_1[args_1/vars][this/y]\rangle\ \langle st/\texttt{targetexpr}(x, ty)\rangle\ \varphi, \Delta \\
\vdots & \\
n+3. & \Gamma, e \neq null, st[mode] = normal, classOf(e, st) = c_n, \\
 & y = e, vars = es \vdash \langle st/\alpha_n[args_n/vars][this/y]\rangle\ \langle st/\texttt{targetexpr}(x, ty)\rangle\ \varphi, \Delta
\end{array}}{\Gamma \vdash \langle st/x = e.m(es);\rangle\ \varphi, \Delta}$$

A method invocation without assignment (i.e. $\langle st/e.m(es);\rangle\ \varphi$) produces a $\texttt{target}(return)$ instead of $\texttt{targetexpr}(x, ty)$ as for the other invocation modes.

In chapter 4.3 it was shown how to derive the different versions of the proof rule from an equivalence formula that roughly states that the conclusion of a proof rule is equivalent to its premises. This equivalence does not hold in this case, because it is not guaranteed that the run-time class of the invoker is one of the selected class (this is premise 3). However, if we make this assumption, the following equivalence holds:

$$\begin{array}{l}
\quad (\text{st[mode]} = \text{normal} \wedge \text{e} \neq \text{null} \rightarrow classOf(e, st) \in \{c_1, \ldots, c_n\}) \\
\rightarrow (\quad \langle \text{st/x} = \text{e.m(es);}\rangle\ \varphi \\
\quad \leftrightarrow \quad (\text{st[mode]} \neq \text{normal} \rightarrow \varphi) \\
\qquad \wedge (\text{st[mode]} = \text{normal} \wedge \text{e} = \text{null} \rightarrow \langle \text{st/}\texttt{throw new NPE();}\rangle\ \varphi) \\
\qquad \wedge (\quad \text{st[mode]} = \text{normal} \wedge \text{e} \neq \text{null} \wedge classOf(e, st) = c_1 \\
\qquad\quad \rightarrow \forall\ \text{y, vars. y} = \text{e} \wedge \text{vars} = \text{es} \rightarrow \langle \text{st/}\alpha_1'\rangle\ \langle \text{st/}\texttt{targetexpr}\text{(x,ty)}\rangle\ \varphi) \\
\quad \vdots \\
\qquad \wedge (\quad \text{st[mode]} = \text{normal} \wedge \text{e} \neq \text{null} \wedge classOf(e, st) = c_n \\
\qquad\quad \rightarrow \forall\ \text{y, vars. y} = \text{e} \wedge \text{vars} = \text{es} \rightarrow \langle \text{st/}\alpha_n'\rangle\ \langle \text{st/}\texttt{targetexpr}\text{(x,ty)}\rangle\ \varphi)
\end{array}$$

Preconditions: is_basic_jexpr(e) ∧ basic_exprs(es)
∧ is_classtype(e.type)
∧ areSubclassesOf(classes, e.type)
∧ areImplementedMethods(classes, str, invmo, tys, ty0, tds)

classes ($= c_1, \ldots, c_n$) is the selected set of classes, *areSubclassesOf* is true iff every class is a subclass of the type of e, *areImplementedMethods* is true iff for every class the dynamic method lookup starting with class returns an implemented (non-native) method declaration.

Variables: y and $vars$ must be new (for every premise):

$$\begin{array}{l}
y + vars = \text{newvars}(\texttt{this} + args, \{s\} \cup \text{vars}(x = e.m(es)) \cup \\
\qquad\qquad \text{vars}(\text{getMethod(m, invmo, tys, ty, tds)}) \cup \text{vars}(\varphi) \cup \text{free}(\Gamma) \cup \text{free}(\Delta))
\end{array}$$

This finishes the description of the proof rules for singe Java expressions. The next section describes the proof rules for Java statements, chapter 4.5 describes generic rules that are applicable to all (or several) statements and expressions. Additional rules are described in chapter 4.6.

4.4 Statements

The proof rules for Java statements are similar to those for Java expressions. There are four version of every rule (diamond in succedent, box in succedent, diamond in antecedent, box in

antecedent). They can be derived from an equivalence formula. The following proof rules deviate from the standard scheme:

- block. No equivalence formula holds for (arbitrary) blocks.
- switch. No equivalence formula holds for (arbitrary) blocks.
- catches, endstatic. These rules have an additional precondition. This means an 'equivalence' has a precondition, $\psi \rightarrow (\langle \mathrm{st}/\alpha \rangle\ \varphi \leftrightarrow \dots)$

4.4.1 Blocks and Local Variable Declarations

Semantics: See chapter 3.3.2. The block is discarded, and local variable declarations are transformed into assignments to new variables. For arbitrary blocks, an equivalence formula cannot be defined. However, the following two proof rules (and identical rules for boxes) hold.

$$\frac{\begin{array}{ll}1. & \Gamma, st[mode] \neq normal \vdash \varphi, \Delta \\ 2. & \Gamma, st[mode] = normal \vdash \langle st/\alpha'_1[locvars, newvars] \rangle \dots \langle st/\alpha'_n[locvars, newvars] \rangle\ \varphi, \Delta\end{array}}{\Gamma \vdash \langle st/\{\alpha_1 \dots \alpha_n\} \rangle\ \varphi, \Delta}$$

$$\frac{\begin{array}{ll}1. & \varphi, st[mode] \neq normal, \Gamma \vdash \Delta \\ 2. & \langle st/\alpha'_1[locvars, newvars] \rangle \dots \langle st/\alpha'_n[locvars, newvars] \rangle\ \varphi, st[mode] = normal, \Gamma \vdash \Delta\end{array}}{\langle st/\{\alpha_1 \dots \alpha_n\} \rangle\ \varphi, \Gamma \vdash \Delta}$$

α_i are the top-level statements of the block. *locvars* are the local variables of the block, i.e. those variables x that occur in a top-level local variable declaration $ty\ x = e$ in the block. *newvars* are new variables. $\alpha'_i \equiv \alpha_i$ except for a local variable declaration. A local variable declaration $ty\ x = e;$ is transformed into the assignment $x = e;$.

Preconditions: (none)

Variables: *newvars* must be new:

$$newvars = \mathrm{newvars}(locvars(\alpha_1 \dots \alpha_n), vars(\alpha_1 \dots \alpha_n) \cup free(\varphi) \cup \mathrm{free}(\Gamma) \cup \mathrm{free}(\Delta))$$

This rule is similar to the first-order sequent calculus rule for a universal quantifier in the succedent (or existential quantifier in the antecedent). An equivalence formula for (illegal) blocks does not hold because the computation may depend on the initial value of the new variable. The two rules do hold, though.

4.4.2 Local variable declaration

Semantics: See chapter 3.3.3. Local variable declarations may occur only inside a block and are handled by the block rule above.

4.4.3 The if Statement

Semantics: See chapter 3.3.5.

$$\frac{\begin{array}{ll}1. & \Gamma, st[mode] \neq normal \vdash \varphi, \Delta \\ 2. & \Gamma, st[mode] = normal, e \vdash \langle st/\alpha \rangle\ \varphi, \Delta \\ 3. & \Gamma, st[mode] = normal, \neg\ e \vdash \langle st/\beta \rangle\ \varphi, \Delta\end{array}}{\Gamma \vdash \langle st/\texttt{if}\ (e)\ \alpha\ \texttt{else}\ \beta \rangle\ \varphi, \Delta}$$

e must be a basic expression.

Preconditions: is_basic_expr(e)

4.4.4 Labeled statements

Semantics: See chapter 3.3.6. A labeled statement is handled by adding a catcher for a break with the given label after the statement.

$$\frac{\begin{array}{ll}1. & \Gamma, st[mode] \neq normal \vdash \varphi, \Delta \\ 2. & \Gamma, st[mode] = normal \vdash \langle st/\alpha \rangle\ \langle st/\texttt{target}(break(l)) \rangle\ \varphi, \Delta\end{array}}{\Gamma \vdash \langle st/l : \alpha \rangle\ \varphi, \Delta}$$

Preconditions: (none)

4.4.5 The while Statement

Semantics: See chapter 3.3.7.

$$\frac{\begin{array}{ll}1. & \Gamma, st[mode] \neq normal \vdash \varphi, \Delta \\ 2. & \Gamma, st[mode] = normal \vdash \langle st/\texttt{if}\ (e)\ \{\alpha\ \texttt{while}\ (e)\ \alpha\}\ \texttt{else}\ \{\} \rangle\ \varphi, \Delta\end{array}}{\Gamma \vdash \langle st/\texttt{while}\ (e)\ \alpha \rangle\ \varphi, \Delta}$$

This rule allows to prove theorems with while loops by Noetherian induction on some data structure. chapter 4.6 contains two other proof rules for `while` loops. e can be an arbitrary Java expression. α may not be a (single) local variable declaration.

Preconditions: $\neg$ is_locvardecl(α)

4.4.6 The do Statement

Semantics: See chapter 3.3.8.

$$\frac{\begin{array}{ll}1. & \Gamma, st[mode] \neq normal \vdash \varphi, \Delta \\ 2. & \Gamma, st[mode] = normal \vdash \langle st/\alpha \rangle\ \langle st/\texttt{if}\ (e)\ \texttt{do}\ \alpha\ \texttt{while}\ e;\ \texttt{else}\ \{\} \rangle\ \varphi, \Delta\end{array}}{\Gamma \vdash \langle st/\texttt{do}\ \alpha\ \texttt{while}\ e; \rangle\ \varphi, \Delta}$$

e can be an arbitrary expression. Other rules for `do` loops could be added.

Preconditions: (none)

4.4.7 The for Statement

Semantics: See chapter 3.3.9. Here, the `for` statement contains no initialization, but only the termination test e, the updates es and the body.

$$\frac{\begin{array}{ll}1. & \Gamma, st[mode] \neq normal \vdash \varphi, \Delta \\ 2. & \Gamma, st[mode] = normal \vdash \langle st/\texttt{if}\ (e)\ \{\alpha\ es;\ \texttt{for}(e; es)\ \alpha\}\ \texttt{else}\ \{\} \rangle\ \varphi, \Delta\end{array}}{\Gamma \vdash \langle st/\texttt{for}(e; es)\ \alpha \rangle\ \varphi, \Delta}$$

If the test e is true the body α and the updates es are evaluated, then again the loop. This rule can be used together with Noetherian induction (for example, on a counter) to prove theorems about `for` loops. Chapter 4.6 contains a specialized invariant rule that is tailored for Java Card applications.

Preconditions: $\neg$ is_locvardecl(α)

4.4.8 The switch Statement

Semantics: See chapter 3.3.10. This rule is deliberately weakening for an illegal switch statement. JLS requires that the switch labels are compile-time constants with different values. Based on this assumption the proof rule simply generates one premise for every group of switch labels (and one for a jump and one for the default case). However, in this work the switch labels may be arbitrary terms; for Java Card applications it is useful to allow symbolic constants. Then it cannot be checked syntactically that the labels have different values. The semantics ignores later occurrences of the same label; the proof rule generates a premise. It would be easy to include a premise that the labels are disjoint, or to add the precondition that previous labels do not match, but all this seems pointless since the labels are always disjoint in normal applications.

$$\frac{\begin{array}{ll} 1. & \Gamma, st[mode] \neq normal \vdash \varphi, \Delta \\ 2. & \Gamma, st[mode] = normal, sb2int(e) \in sb2ints(cases_1) \vdash \langle st/\{\alpha_1 \dots \alpha_n\}\rangle\ \varphi, \Delta \\ & \vdots \\ n. & \Gamma, st[mode] = normal, sb2int(e) \in sb2ints(cases_n) \vdash \langle st/\{\alpha_n\}\rangle\ \varphi, \Delta \\ n+1. & \Gamma, st[mode] = normal, \neg\ sb2int(e) \in sb2ints(cases_1 \dots cases_n) \\ & \vdash \langle st/\{\alpha \dots \alpha_n\}\rangle\ \varphi, \Delta \end{array}}{\Gamma \vdash \langle st/\texttt{switch (e) \{}s_1 \dots s_n\texttt{\}}\rangle\ \varphi, \Delta}$$

- s_i is a switchLabelStatement consisting of some labels (cases) and some other statements without labels, $s_i \equiv cases_i\ \alpha_{i1} \dots \alpha_{ik}$. $cases_i \equiv \texttt{case}\ t_{i1}\texttt{:} \dots \texttt{case}\ t_{ij}$
- $\alpha_i \dots \alpha_n$ are all statements following the labels $cases_i$ with all switchlabels removed since a switch statement 'falls through'.
- Every switchLabelStatement must have one or more labels. The `default` case is a switchLabelStatement with no labels, or the empty block if the `switch` has no `default` case.

The optional `default` case may appear anywhere in the `switch` block. It seems unnecessary to transform the `switch` statement into, for example, a nested `if-then-else`.

Preconditions: is_basic_expr(e)

4.4.9 The break Statement

Semantics: See chapter 3.3.11.

$$\frac{\begin{array}{ll} 1. & \Gamma, st[mode] \neq normal \vdash \varphi, \Delta \\ 2. & \Gamma, st[mode] = normal, st_0 = st[mode, break(l)] \vdash \varphi[st/st_0], \Delta \end{array}}{\Gamma \vdash \langle st/\texttt{break}\ l\texttt{;}\rangle\ \varphi, \Delta}$$

The `break` statement simply raises the appropriate mode as in the semantics. This rule can be optimized by discarding Java statements in φ that cannot catch a `break`. See chapter 4.6.4 for more details.

Preconditions: (none)

Variables: $st_0 = \text{newvar}(st, \{st\} \cup \text{vars}(\varphi) \cup \text{free}(\Gamma) \cup \text{free}(\Delta))$

4.4.10 Empty return Statement

Semantics: See chapter 3.3.12.

$$\frac{\begin{array}{ll} 1. & \Gamma, st[mode] \neq normal \vdash \varphi, \Delta \\ 2. & \Gamma, st[mode] = normal, st_0 = st[mode, return] \vdash \varphi[st/st_0], \Delta \end{array}}{\Gamma \vdash \langle st/\texttt{return;}\rangle\ \varphi, \Delta}$$

This rule can also be optimized by discarding Java statements in φ that cannot catch a `return`. See chapter 4.6.4 for more details.

Preconditions: (none)

Variables: $st_0 = \text{newvar}(st, \{st\} \cup \text{vars}(\varphi) \cup \text{free}(\Gamma) \cup \text{free}(\Delta))$

4.4.11 return Statement with Expression

Semantics: See chapter 3.3.13.

$$\frac{\begin{array}{ll}1. & \Gamma, st[mode] \neq normal \vdash \varphi, \Delta \\ 2. & \Gamma, st[mode] = normal, st_0 = st[mode, return(e, ty)] \vdash \varphi[st/st_0], \Delta\end{array}}{\Gamma \vdash \langle st/\texttt{return}e; \rangle\ \varphi, \Delta}$$

e must be a basic expression. We store the value in the mode for a `targetexpr`. The rule can optimized by discarding Java statements in φ that cannot catch a `return`. See chapter 4.6.4 for more details.

Preconditions: is_basic_expr(e)

Variables: $st_0 = \text{newvar}(st, \{st\} \cup \text{vars}(e) \cup \text{vars}(\varphi) \cup \text{free}(\Gamma) \cup \text{free}(\Delta))$

4.4.12 The throw Statement

Semantics: See chapter 3.3.14.

$$\frac{\begin{array}{ll}1. & \Gamma, st[mode] \neq normal \vdash \varphi, \Delta \\ 2. & \Gamma, st[mode] = normal, e = null \vdash \langle st/\texttt{throw new NullPointerException();} \rangle\ \varphi, \Delta \\ 3. & \Gamma, st[mode] = normal, e \neq null, st_0 = st[mode, throw(e, ty)] \vdash \varphi[st/st_0], \Delta\end{array}}{\Gamma \vdash \langle st/\texttt{throw } e; \rangle\ \varphi, \Delta}$$

The throw is ignored if the current mode is not normal. The throw expression e must be a basic expression. If it is null a `NullPointerException` is thrown. The rule can optimized by discarding Java statements in φ that cannot catch a `throw`. See chapter 4.6.4 for more details.

Preconditions: is_basic_expr(e)

Variables: $st_0 = \text{newvar}(st, \{st\} \cup \text{vars}(e) \cup \text{vars}(\varphi) \cup \text{free}(\Gamma) \cup \text{free}(\Delta))$

4.4.13 The try Statement

Semantics: See chapter 3.3.15. The rule is very simple because two of the new Java statements are used.

$$\frac{\begin{array}{ll}1. & \Gamma, st[mode] \neq normal \vdash \varphi, \Delta \\ 2. & \Gamma, st[mode] = normal \vdash \langle st/\alpha \rangle\ \langle st/\texttt{catches}(catches) \rangle\ \langle st/\texttt{finally}(\beta) \rangle\ \varphi, \Delta\end{array}}{\Gamma \vdash \langle st/\texttt{try } \alpha\ catches\ \texttt{finally } \beta \rangle\ \varphi, \Delta}$$

The list of catch clauses *catches* is transformed into the additional statement `catches`(catches), the `finally` block is transformed into the additional statement `finally`(β).

Preconditions: (none)

4.4.14 The catches Statement

Semantics: See chapter 3.3.16. This statement behaves like the `catch` clauses in a `try-catch` statement. If the mode is not a *throw* mode (this means that no exception occurred) the statement is skipped. Otherwise the correct catcher depends on the run-time class of the thrown reference.

For every `catch` clause one premise is generated. An additional premise guarantees that this class exists.

$$\frac{\begin{array}{ll}1. & \Gamma, is_throw_mode(st[mode]) \vdash r \neq null \wedge type\exists\ (ty), \Delta \\ 2. & \Gamma, is_throw_mode(st[mode]), ty \leq c_1, st_0 = st[mode, normal], y = r \\ & \quad \vdash \langle st_0/\alpha_1[x_1/y]\rangle\ \varphi[st/st_0], \Delta \\ \vdots & \quad \vdots \\ n+1. & \Gamma, is_throw_mode(st[mode]), \neg\ ty \leq c_1 \wedge\ \ldots\ \wedge \neg\ ty \leq c_{n-1}, \\ & ty \leq c_n, st_0 = st[mode, normal], y = r \\ & \quad \vdash \langle st_0/\alpha_n[x_1/y]\rangle\ \varphi[st/st_0], \Delta \\ n+2. & \Gamma, \quad \neg\ is_throw_mode(st[mode]) \vee (\neg\ ty \leq c_1 \wedge\ \ldots\ \wedge \neg\ ty \leq c_n) \\ & \quad \vdash \varphi, \Delta \end{array}}{\Gamma \vdash \langle st/\texttt{catches}(c_1(x_1)\alpha_1 ... c_n(x_n)\alpha_n)\rangle\ \varphi, \Delta}$$

$r \equiv st[mode].refval$ is the thrown reference, $ty \equiv st[st[mode].refval_type].type$ the run-time type of the thrown reference. The class hierarchy is axiomatized in the predicate $\leq$. $ty \leq c$ is true if ty is a class type, and the class is a subclass of c or equal to c. (See chapter 6.)

Consider the following two `catch` clauses:

```
catch(NullPointerException e) {...}

catch(Exception e) {...}
```

`NullPointerException` is a subclass of `Exception`. The first clause is used if the thrown exception was `NullPointerException` or a (user-defined) subclass of `NullPointerException`. The second clause is used if the thrown exception is `Exception` or one of its subclasses, but not `NullPointerException` or one of its subclasses. The clauses behave like a nested `if-then-else`. Therefore, the negated $\leq$ tests of the previous clauses are included in the preconditions of the next clause. The last premise is for the case that the mode was not *throw* or that the exception is not caught. st_0 and y are new variables, y is used for the formal parameter of the catch clause (`e` in the example above). If a catcher was found the mode is set back to *normal*.

Preconditions: areSubclassesOf($c_1 \ldots c_n$, java.lang.Throwable, tds)

Variables: $y + st_0 = \text{newvar}(x_i + st, \{st\} \cup \text{vars}(\alpha_i) \cup \text{vars}(\varphi) \cup \text{free}(\Gamma) \cup \text{free}(\Delta))$

for every catch clause.

4.4.15 The finally Statement

Semantics: See chapter 3.3.17. `finally` catches all non-normal modes.

$$\frac{\begin{array}{ll}1. & \Gamma, st[mode] \neq normal, st_0 = st[mode, normal] \\ & \quad \vdash \langle st_0/\alpha\rangle\ \langle st_0/\texttt{endfinally}(st[mode])\rangle\ \varphi[st/st_0], \Delta \\ 2. & \Gamma, st[mode] = normal \vdash \langle st/\alpha\rangle\ \varphi, \Delta \end{array}}{\Gamma \vdash \langle st/\texttt{finally}(\alpha)\rangle\ \varphi, \Delta}$$

If the mode is normal the second premise applies and the `finally` block is simply executed. Otherwise the mode is set back to normal, the `finally` block is executed, and an `endfinally` statement has been added to re-raise the mode at the end.

Preconditions: (none)

Variables: $st_0 = \text{newvar}(st, \{st\} \cup \text{vars}(\alpha) \cup \text{vars}(\varphi) \cup \text{free}(\Gamma) \cup \text{free}(\Delta))$

4.4.16 The endfinally Statement

Semantics: See chapter 3.3.18.

$$\frac{\begin{array}{ll}1. & \Gamma, st[mode] \neq normal \vdash \varphi, \Delta \\ 2. & \Gamma, st[mode] = normal, st_0 = st[mode, mo] \vdash \varphi[st/st_0], \Delta\end{array}}{\Gamma \vdash \langle st/\texttt{endfinally}(mo)\rangle\ \varphi, \Delta}$$

`endfinally` re-raises the mode if the current mode is normal. Otherwise nothing happens.

Preconditions: (none)

Variables: $st_0 = \text{newvar}(st, \{st\} \cup \text{vars}(mo) \cup \text{vars}(\varphi) \cup \text{free}(\Gamma) \cup \text{free}(\Delta))$

4.4.17 The empty target Statement

Semantics: See chapter 3.3.19. A `target` statement catches `return`'s and `break`'s. It is used in the rules for a labeled statement and method invocation.

$$\frac{\begin{array}{ll}1. & \Gamma, \quad is_return_mode(st[mode]) \wedge is_return_mode(mo) \\ & \quad \vee\ is_break_mode(st[mode]) \wedge st[mode] = mo, \\ & st_0 = st[mode, normal] \vdash \varphi[st/st_0], \Delta \\ 2. & \Gamma, \neg\ (\quad is_return_mode(st[mode]) \wedge is_return_mode(mo) \\ & \qquad \vee\ is_break_mode(st[mode]) \wedge st[mode] = mo) \vdash \varphi, \Delta\end{array}}{\Gamma \vdash \langle st/\texttt{target}(mo)\rangle\ \varphi, \Delta}$$

mo is the mode to catch, either *break* with a given label or *return*. If the mode is caught then the mode is set back to normal. Otherwise nothing happens.

Preconditions: (none)

Variables: $st_0 = \text{newvar}(st, \{st\} \cup \text{vars}(\varphi) \cup \text{free}(\Gamma) \cup \text{free}(\Delta))$

4.4.18 The target Statement with Expression

Semantics: See chapter 3.3.20. This statement is used to catch `return`'s that return a value. The value is then assigned to a variable.

$$\frac{\begin{array}{ll}1. & \Gamma, \neg\ is_return_mode(st[mode]) \vdash \varphi, \Delta \\ 2. & \Gamma, is_return_mode(st[mode]), st_0 = st[mode, normal], y = coerce(st[mode].val, ty) \\ & \quad \vdash \varphi[st/st_0][x/y], \Delta\end{array}}{\Gamma \vdash \langle st/\texttt{targetexpr}(x, ty)\rangle\ \varphi, \Delta}$$

In the rule the assignment has been carried out already. To ensure type soundness the value must be coerced to the type ty of x.

Preconditions: (none)

Variables: $y + st_0 = \text{newvars}(x + st, \{x, st\} \cup \text{vars}(\varphi) \cup \text{free}(\Gamma) \cup \text{free}(\Delta))$

4.4.19 The static Statement

Semantics: See chapter 3.3.21. The `static` statement handles initialization of classes. It has three premises. The first is for the case that the mode is not normal, or that the class is not

uninitialized. The second premise initializes the super class, then executes the static initializer, and the third premise simply executes the static initializer.

$$\frac{\begin{array}{ll}1. & \Gamma, \neg\, initundone(c, st) \lor st[mode] \neq normal \\ & \quad \vdash \langle st/\texttt{throw new NoClassDefFoundError();}\rangle\ \varphi, \Delta \\ 2. & \Gamma, initundone(c, st), st[mode] = normal, c \neq Object, \neg\, initdone(s, st), \\ & st_0 = addclass(c, sfields, st) \vdash \langle st_0/\texttt{static}(s)\rangle\ \langle st_0/\alpha\rangle\ \langle st_0/\texttt{endstatic}(c)\rangle\ \varphi[st/st_0], \Delta \\ 3. & \Gamma, initundone(c, st), st[mode] = normal, c = Object \lor initdone(s, st), \\ & \quad st_0 = addclass(c, sfields, st) \vdash \varphi[st/st_0], \Delta \end{array}}{\Gamma \vdash \langle st/\texttt{static}(c)\rangle\ \varphi, \Delta}$$

In premise 2 and 3 $s \equiv \text{superClass}(c)$ is the superclass of c, $\alpha \equiv \text{statinit}(c)$ the static initializer for class c. *sfields* are the static fields of c with their default values. *addclass* adds the static fields to the store (see chapter 3.1.3). `static` is used to add the static fields of a class to the store, and to initialize the super classes. All goals are mutually exclusive. An `endstatic` statement is added to catch exceptions occurring during initialization. The rule contains some redundancy because premise 2 and 3 both contain the static initializer for the class. However, usually two of the premises are trivially true because their preconditions are not met.

Preconditions: is_tdname+(c, tds) (the class or interface either exists or is predefined)

Variables: $st_0 = \text{newvar}(st, \{st\} \cup \text{vars}(\varphi) \cup \text{free}(\Gamma) \cup \text{free}(\Delta))$

4.4.20 The endstatic Statement

Semantics: See chapter 3.3.22. Endstatic catches exceptions. If an exception or error occurs during class initialization, the class state becomes 'erroneous'. If an exception occurred, an `ExceptionInInitializerError` is thrown. In case of an error the mode remains unchanged. An additional premise ensures that if a throw occurred the thrown reference is not null and has an existing (or predefined) class.

$$\frac{\begin{array}{ll}1. & \Gamma, is_throw_mode(st[mode]) \\ & \quad \vdash r \neq null \land type\exists\ (ty), \Delta \\ 2. & \Gamma, is_throw_mode(st[mode]), \neg\, ty \leq \texttt{Exception}, \\ & st_0 = st[jvmref - mkfs(c, \texttt{void}, initstate), initerror] \vdash \varphi_{[}st/st_0], \Delta \\ 3. & \Gamma, is_throw_mode(st[mode]), ty \leq \texttt{Exception}, \\ & st_0 = st[jvmref - mkfs(c, \texttt{void}, initstate), initerror][mode, normal] \\ & \quad \vdash \langle st_0/\texttt{throw new ExceptionInInitializerError();}\rangle\ \varphi[st/st_0], \Delta \\ 4. & \Gamma, \neg\, is_throw_mode(st[mode]), \\ & st_0 = st[jvmref - mkfs(c, \texttt{void}, initstate), initdone] \vdash \varphi[st/st_0], \Delta \end{array}}{\Gamma \vdash \langle st/endstatic(c)\rangle\ \varphi, \Delta}$$

$r \equiv st[mode].refval$ is the thrown reference, $ty \equiv st[st[mode].refval - _type].type$ its type. $ty \leq$ `Exception` is true if the type of the thrown object $st[mode].refval$ is a subclass of (or equal to) `Exception`. The initialization state of a class c is recorded in the special field mkfs(c, `void`, *initstate*) under the special reference *jvmref*.

Preconditions: (none)

Variables: $st_0 = \text{newvar}(st, \{st\} \cup \text{vars}(\varphi) \cup \text{free}(\Gamma) \cup \text{free}(\Delta))$

4.5 Generic Rules

This section contain four rules that are applicable for more than one expression or statement. They are for simple binary expressions and others, and, most notably the flattening of expressions and statements.

4.5.1 Assignment Introduction

As mentioned in chapter 4.3.1 we need proof rules for expressions with and without an assignment. The version without assignment can be derived from the version with assignment by introducing an assignment. Or, to express it another way, the versions without assignment is not really needed (except for `void` expressions). The assignment introduction rule is the following rule:

$$\frac{\Gamma \vdash \langle st/x = e; \rangle\ \varphi, \Delta}{\Gamma \vdash \langle st/e; \rangle\ \varphi, \Delta}$$

x must be a new variable (which can be selected by the user) with a sort that allows an assignment by e. This implies that the result type of e may not be void unless e is the `null` literal; in this case x must be a reference. This rule comes in the usual four versions (box, diamond, antecedent, and succedent). The following equivalence formula holds:

$\langle$st/e;$\rangle\ \varphi \leftrightarrow \forall$ x. $\langle$st/x = e;$\rangle\ \varphi$

Preconditions: (e = null $\vee$ e.type $\neq$ void_type) $\wedge \neg$ x $\in$ free(φ) $\wedge$ sort(x) = type2sort(e)

x can occur in e, but not free in φ.

4.5.2 Literalize

The rule *literalize* transforms some Java expressions into (pseudo-)literals. For example, `x + y` (with the binary operator `+` on integers) is defined as the addition operation on the algebraic specification of integers. `x + y` (Java binary expression) is transformed into $x + y$ (algebraic addition on integers), and $x + y$ is regarded as a literal, in the sense of a basic Java expression that needs no further evaluation. The following expressions can be literalized:

literal expression is trivially literalizable.

local variable access needs no further evaluation, and can be used in a literal.

primitive cast converts a number to another representation, e.g. from integer to byte. If the argument can be literalized the cast can also be literalized by applying the corresponding algebraic conversion function (i2b etc.).

unary operator `!` (logical negation), `~` (bitwise complement), `+`, `-` can be literalized if the argument can be literalized.

simple binary operator These are `==`, `+`, `-`, `*`, `<`, `>`, `<=`, `>=`, `&`, `^`, `|`, `<<`, `>>`, `>>>`. Both arguments must be literalizable. (e_1 `!=` e_2 is transformed into `!` (e_1 `==` e_2).)

For the other binary operators special proof rules exist: *exception binary operator* (chapter 4.3.9) for `/` and `%`, and *conditional binary operator* (chapter 4.3.7) for `||` and `&&`.

The rule is always applicable, even in case of a jump. Schematically:

$$\frac{\Gamma \vdash \langle x = \mathit{Literal}(\mathit{literalize}(e)); \rangle\ \varphi, \Delta}{\Gamma \vdash \langle x = e; \rangle\ \varphi, \Delta}$$

Preconditions: literalizable(e)

If an expression occurring anywhere in a Java statement or expression is literalizable it can be literalized. This means literalization can be applied recursively.

4.5.3 FlattenOne

The rule *flattenOne* deals with nested expressions by introducing auxiliary variables. For example, $\langle st/x = m(f(g(y)), h(z));\rangle\ \varphi$ becomes $\langle st/x_0 = g(y);\rangle\ \langle st/x = m(f(x_0), h(z));\rangle\ \varphi$. The general idea of this rule was explained at the beginning of this chapter (see 4.1.2). In a sense, flattening modifies the evaluation order in a 'harmless' manner.

The flattening rule works as follows:

1. For an expression $x = e$ select the immediate subexpressions $e_1, \ldots, e_n$ of e that can be flattened (for example, in e_1`&&`e_2 only e_1 can be flattened, because e_2 is not always evaluated).
2. Find the first e_i that is not a local variable or a literal, and that does not cause a variable conflict (see case 4).
3. Replace e_i in e by a new variable y yielding e' and add the assignment $y = e_i$ before $x = e'$.
4. A variable conflict occurs if e_i contains an assignment to a variable that occurs in $e_1, \ldots, e_{i-1}$, e.g. in `x * (x = 3)` or `x * (x++)`. It is wrong to transform these expressions into something like `y = (x = 3), x * y` or `y = (x++), x * y` resp. In this case a renaming is necessary.

Schematically the rule flattenOne is

$$\frac{\Gamma \vdash \langle st/x = e;\rangle\ \langle st/\alpha'\rangle\ \varphi, \Delta}{\Gamma \vdash \langle st/\alpha\rangle\ \varphi, \Delta}$$

e is the first subexpression of α that can be flattened (without variable conflict), α' is α where the occurrence of e is replaced by x, x is a new variable. The rule is called *flattenOne* because exactly one subexpression is extracted.

Preconditions: (none)

Variables: $x = \text{newvar}(e.\text{type}, \{st\} \cup \text{vars}(\alpha) \cup \text{free}(\varphi) \cup \text{free}(\Gamma) \cup \text{free}(\Delta))$

The following equivalence holds:

$$\langle st/\alpha\rangle\ \varphi \leftrightarrow \langle st/x = e;\rangle\ \langle st/\alpha'\rangle\ \varphi$$

The precise definition is as follows:

- Let α be an arbitrary Java statement. If α is an expression statement it is flattened as defined below. Otherwise select the subexpressions of α that can be flattened (see figure 4.1). If this list is not empty it can contain only one expression e.

  (Only the test of an `if` or `switch`, or the expression of an `return` or `throw` can be flattened.)

  If e is not a basic expression (a basic expression is either a local variable access or a literal) continue.

- Generate a new variable x, replace the occurrence of e by x, and add the assignment $x = e$;

An expression statement e; is flattened in the following manner:

- If e is a local variable assignment and the right hand side again an assignment the rule is not applicable (in this case the *local variable assignment* rule can be used).
- Otherwise if e is a local variable assignment $x = e'$ continue with e', i.e. e' will be flattened. Otherwise continue with e.
- Let $e_1, \ldots, e_n = subexprs(e)$ the subexpression of e (or e') that can be flattened as defined in figure 4.2. Usually, only the immediate subexpressions are considered, except for an incdec expression and a compound assignment. Here, a case distinction is made for the leftmost subexpression of the expression (see figure 4.2).

Notation	our name	subexpressions
e;	Exprstatement	$subexprs(e)$
if (e) α **else** β	If	$[e]$
switch (e) $sl_1 \ldots sl_n$	Switch	$[e]$
return e;	ReturnExpr	$[e]$
throw e;	Throw	$[e]$
Otherwise		[]

Figure 4.1: Subexpressions that can be flattened for statements.

- Let k the first index $1 \leq k \leq n$ such that e_k is the first non-basic expression, i.e. e_i (for all $1 \leq i < k$) is a literal or a local variable access, e_k is not.
- Furthermore, the following condition must hold: $\text{vars}(e_1, \ldots, e_{k-1}) \cap \text{asgvars}(e_k) = \emptyset$. Then the rule is applicable.
- Generate a new variable x, and replace the k-th position in $e_1, \ldots, e_n$ by x.
- Then replace the subexpressions of e by the new subexpressions $e_1, \ldots, e_{k-1}, x, e_{k+1}, \ldots, e_n$. This operation is the inverse of $subexprs$. The result is the flattened expression.
- If the original expression was a local variable assignment, add the assignment.

As mentioned above, only five statements can be flattened:

> expression statement, if statement, switch statement, return statement with a value, throw statement.

Only the proof rules for these statements have the precondition that the expression is a basic expression. This means that either a proof rule from chapter 4.4 or the flattening rule is applicable (if other preconditions hold). The same is true for expressions.

The soundness proof for this rule is not really difficult but very large, becaus all possible subexpressions for all possible statements and expressions must be considered.

4.5.4 FlattenConflict

In case of a variable conflict (e.g. `x * x++`) we have the rule *flattenConflict*. It introduces a variable renaming and then performs a flattening. A variable conflict can occur only for an expression statement. Schematically the rule looks like:

$$\frac{\Gamma, newvars = vars \vdash \langle st/x = sube; \rangle \ \langle st/e' \rangle \ \varphi, \Delta}{\Gamma \vdash \langle st/e; \rangle \ \varphi, \Delta}$$

$sube$ is the first subexpression of e that can be flattened (and introduces a variable conflict), e' is e where the conflicting variables $vars$ that occur before $sube$ are replaced by new variables $newvars$, and the occurrence of $sube$ is replaced by another new variable x.

Preconditions: (none)

Variables: $newvars$ and x must be new:

$newvars = \text{newvars}(vars, \{st\} \cup \text{vars}(e) \cup \text{free}(\varphi) \cup \text{free}(\Gamma) \cup \text{free}(\Delta))$

$x = \text{newvar}(e.\text{type}, newvars \cup \{st\} \cup \text{vars}(e) \cup \text{free}(\varphi) \cup \text{free}(\Gamma) \cup \text{free}(\Delta))$

The precise definition is as follows:

Notation	**Abstract syntax**	**Subexpressions**
l	LiteralExpr	[]
$\oplus e$	UnaryExpr	$[e]$
$e\oplus$	IncDecExpr	see below
$(ty)e$	PrimCast	$[e]$
$(ty)e$	RefCast	$[e]$
e instanceof ty	InstanceExpr	$[e]$
$e_1?e_2 : e_3$	CondExpr	$[e_1]$
$e_1 \oplus e_2$	CondBinExpr	$[e_1]$
$e_1 \oplus e_2$	BinaryExpr	$[e_1, e_2]$
$e_1 \oplus e_2$	ExBinExpr	$[e_1, e_2]$
x	LocVarAccess	[]
f	SFieldAccess	[]
$e.f$	FieldAccess	$[e]$
$e_1[e_2]$	ArrayAccess	$[e_1, e_2]$
$x = e$	LocVarAssign	subexprs(e)
$f = e$	SFieldAssign	$[e]$
$e_1.f = e_2$	FieldAssign	$[e_1, e_2]$
$e_1[e_2] = e_3$	ArrayAssign	$[e_1, e_2, e_3]$
$e_1 \oplus= e_2$	CompAssign	see below
new $c(e_1, \ldots, e_n)$	NewExpr	[]
new $ty[e_1]..[e_n][i]$	NewArray	$[e_1, \ldots, e_n]$
$\{e_1, \ldots, e_n\}$	ArrayInit	$[e_1, \ldots, e_n]$
$e.c(e_1, \ldots, e_n)$	ConstrCall	$[e, e_1, \ldots, e_n]$
$e.m(e_1, \ldots, e_n)$	MethodCall	$[e, e_1, \ldots, e_n]$
incdec expression		
Notation	**Abstract Syntax**	**Subexpressions**
$e.f\oplus$	FieldAccess	$[e]$
$e_1[e_2]\oplus$	ArrayAccess	$[e_1, e_2]$
$e\oplus$	Otherwise	[]
compound assignment		
Notation	**Abstract Syntax**	**Subexpressions**
$e_1.f \oplus= e$	FieldAccess	$[e_1]$
$e_1[e_2] \oplus= e$	ArrayAccess	$[e_1, e_2]$
$e_1 \oplus= e$	Otherwise	[]

Figure 4.2: Subexpressions that can be flattened for expressions. For an incdec expression and a compound assignment the leftmost subexpression is considered.

- Let $e;$ be an expression statement, and $e_1, \ldots, e_n = subexprs(e)$ as defined above. (If e is a local variable assignment the right hand side of e is taken. This cannot again be an assignment because this would not introduce a variable conflict.)
- Let k the first index $1 \leq k \leq n$ such that e_k is the first non-basic expression, i.e. e_i (for all $1 \leq i < k$) is a literal or a local variable access, e_k is not.
- Furthermore, the following condition must hold: $\text{vars}(e_1, \ldots, e_{k-1}) \cap \text{asgvars}(e_k) \neq \emptyset$. Let $vars = \text{vars}(e_1, \ldots, e_{k-1}) \cap \text{asgvars}(e_k)$. $vars$ are the variables mentioned above. Then the rule is applicable.
- Generate new variables $newvars$ for $vars$, and another new variable x.
- Replace $vars$ by $newvars$ in $e_1, \ldots, e_{k-1}$, i.e. $e'_1, \ldots, e'_{k-1} = (e_1, \ldots, e_{k-1})[vars/newvars]$, and replace the k-th position in $e_1, \ldots, e_n$ by x.
- Then replace the subexpressions of e by the new subexpressions $e'_1, \ldots, e'_{k-1}, x, e_{k+1}, \ldots, e_n$. The result is e'. If the original e was an assignment add the assignment.

Obviously, a program that has this kind of variable conflict is difficult to understand, and very bad programming style. The author has never seen a 'normal' program that has this variable conflict. In fact, initially the *flattenOne* rule had no provision for a variable conflict, and the error was detected when trying to prove the correctness of the rule.

4.6 Additional Rules

The following rules are not necessary for a complete calculus because they can be proved for every concrete Java program. But they are very important, useful, and an integral part of the proof methodology. A calculus that is usable requires more than just the minimal rule set needed for a (relatively) complete calculus. A typical example for predicate logic is a simplification rule. Therefore additional rules should either

- reduce the number of necessary proof steps (thereby reducing the time needed for the proof), or
- make proofs simpler, or
- spare the user to formulate and prove a theorem.

This section presents some proof rules that fall in these categories. They are listed roughly in a decreasing order of importance/usefulness. They deal with sequents that contain programs in the antecedent and the succedent. These rules usually do not have different versions (for boxes/diamonds and antecedent/succedent), but only the version shown below.

4.6.1 Split left

The first rule allows to introduce auxiliary variables. It deals with programs in the antecedent. Antecedent formulas are preconditions; a formula $\langle st/\alpha \rangle\ \varphi$ means: the program α terminates, and afterwards φ holds. We can access φ with this rule:

$$\frac{1.\quad \langle st/\alpha \rangle\ (st = st_0 \wedge asgvars(\alpha) = vars), \varphi[st/st_0][asgvars(\alpha)/vars], \Gamma \vdash \Delta}{\langle st/\alpha \rangle\ \varphi, \Gamma \vdash \Delta}$$

$\text{asgvars}(\alpha)$ are the assigned variables of the program α, i.e. all variables that may change their value (the definition is given in the next section). st_0 and $vars$ are new variables. This rule shows that the behavior of a Java program is completely determined by the store and the assigned variables.

Preconditions: (none)

Variables: $st_0 + vars = \text{newvars}(st + \text{asgvars}(\alpha), \text{vars}(\langle st/\alpha\rangle\ \varphi) \cup \text{free}(\Gamma) \cup \text{free}(\Delta))$

The following equivalence holds:

$$\begin{aligned} & tds \times A \times v \models \langle st/\alpha\rangle\ \varphi \\ \leftrightarrow\ & \exists\, s_0 + jvars.(\langle st/\alpha\rangle\ (st = st_0 \wedge asgvars(\alpha) = vars)) \wedge \varphi[st/st_0][asgvars(\alpha)/vars] \end{aligned}$$

This shows that the rule could be applied for diamonds in the succedent; however this does not make sense because it introduces an existential quantifier. However, the rule could be used for boxes in the succedent.

4.6.2 Execute Program

If a Java program occurs in the antecedent and succedent, for example $\langle st/\alpha\rangle\ \varphi$ and $\langle st/\alpha\rangle\ \psi$, we know that both programs compute the same values for the assigned variables (Java is deterministic). This can be used to access ψ:

$$\frac{\begin{array}{ll} 1. & \langle st/\alpha\rangle\ (st = st_0 \wedge asgvars(\alpha) = vars), \varphi[st/st_0][asgvars(\alpha)/vars], \Gamma \\ & \vdash \psi[st/st_0][asgvars(\alpha)/vars], \Delta \end{array}}{\langle st/\alpha\rangle\ \varphi, \Gamma \vdash \langle st/\alpha\rangle\ \psi, \Delta}$$

Obviously, α in the antecedent and the succedent compute the same results (the two stores are equal and Java is deterministic). This means we can discard the program in the succedent and continue with ψ if we replace all assigned variables $asgvars(\alpha)$ by their new values $vars$.

Preconditions: (none)

Variables: $st_0 + vars = \text{newvars}(st + \text{asgvars}(\alpha), \text{vars}(\langle st/\alpha\rangle\ \varphi) \cup \text{vars}(\psi) \cup \text{free}(\Gamma) \cup \text{free}(\Delta))$

This situation occurs if a lemma is used, or if an induction hypothesis is applied – this means it is very common, and this is an important rule. α is usually a method call, or a class instance creation, or a loop. The rule can also be applied for two boxes, or a diamond in the antecedent and a box in the succedent. (But not for a box in the antecedent and a diamond in the succedent, because in this case the termination of α must be proved.)

Since the Java programs are deterministic and the complete information needed for evaluating them is contained in the store several distributivity properties hold for diamonds and boxes, e.g.

1. $[st/\alpha]\varphi \leftrightarrow \neg\ \langle st/\alpha\rangle\ \varphi$
2. $tds, \mathcal{A}, v \models \langle st/\alpha\rangle\ (\varphi \wedge \psi) \leftrightarrow \langle st/\alpha\rangle\ \varphi \wedge \langle st/\alpha\rangle\ \psi$
3. $tds, \mathcal{A}, v \models \langle st/\alpha\rangle\ (\varphi \vee \psi) \leftrightarrow \langle st/\alpha\rangle\ \varphi \vee \langle st/\alpha\rangle\ \psi$
4. $tds, \mathcal{A}, v \models [st/\alpha](\varphi \rightarrow \psi) \leftrightarrow (\langle st/\alpha\rangle\ \varphi \rightarrow \langle st/\alpha\rangle\ \psi)$

They are proved trivially by applying the semantics definitions for boxes and diamonds. Then the *execute program* rule can be derived in the following manner:

$\langle st/\alpha\rangle\ (\mathbf{x} = \mathbf{y}), \varphi[\mathbf{x}/\mathbf{y}] \vdash \psi[\mathbf{x}/\mathbf{y}]$	
$\langle st/\alpha\rangle\ (\mathbf{x} = \mathbf{y}), (\varphi \wedge \neg\ \psi)[\mathbf{x}/\mathbf{y}] \vdash$	
$\langle st/\alpha\rangle\ (\varphi \wedge \neg\ \psi) \vdash$	split left
$\vdash \neg\ \langle st/\alpha\rangle\ \neg\ (\varphi \rightarrow \psi)$	
$\vdash [st/\alpha](\varphi \rightarrow \psi)$	Duality
$\vdash \langle st/\alpha\rangle\ \varphi \rightarrow \langle st/\alpha\rangle\ \psi$	item 3
$\langle st/\alpha\rangle\ \varphi \vdash \langle st/\alpha\rangle\ \psi$	

A more flexible version of the rule allows the renaming of variables if they are initially equal.

4.6.3 Contract Program

We can do the same as execute program does if we have the two programs in the antecedent:

$$\frac{\begin{array}{l}1.\quad \langle st/\alpha\rangle\ (st = st_0 \wedge asgvars(\alpha) = vars),\\ \quad\ \varphi[st/st_0][asgvars(\alpha)/vars],\\ \quad\ \psi[st/st_0][asgvars(\alpha)/vars], \Gamma \vdash \Delta\end{array}}{\langle st/\alpha\rangle\ \varphi, \langle st/\alpha\rangle\ \psi, \Gamma \vdash \Delta}$$

This situation typically occurs if more than one lemma is applied for the same program (method call etc.).

Preconditions: (none)

Variables: $st_0 + vars = \text{newvars}(st + \text{asgvars}(\alpha), \text{vars}(\langle st/\alpha\rangle\ \varphi) \cup \text{vars}(\psi) \cup \text{free}(\Gamma) \cup \text{free}(\Delta))$

Again, a more flexible version of the rule allows the renaming of variables.

4.6.4 Optimized Throw Rule

The throw rule sets the mode from 'normal' to 'throw' in the successful case (chapter 4.4.12). If the formula after the `throw` is a box or diamond, and the statement is an `if` or an expression statement, it will be skipped because the mode is not normal (this also requires the same store variables). This means the throw rule can discard formulas with statements after the `throw` statement until a possible catcher for the throw is reached. A possible catcher is either a `finally`, a `catches`, a `catch` or an `endstatic` statement. Some examples:

$\langle$st/`throw x;`$\rangle$ $\langle$st/`i = 3;`$\rangle$ $\varphi \Rightarrow$ skip `i = 3;`, continue with φ

$\langle$st/`throw x;`$\rangle$ $\langle$st0/`i = 3;`$\rangle$ $\varphi \Rightarrow$ do not skip, st $\neq$ st0

$\langle$st/`throw x;`$\rangle$ $\forall$ i. $\varphi \Rightarrow$ do not skip, not a program

More precisely, we can define a function *next_throw_catcher*(st, φ):

nextthrow-fma : $\neg$ is_box(φ) $\wedge$ $\neg$ is_dia(φ) $\rightarrow$ next_throw_catcher(s, φ) = φ

nextthrow-var : (is_box(φ) $\vee$ is_dia(φ)) $\wedge$ φ.var $\neq$ s $\rightarrow$ next_throw_catcher(s, φ) = φ

φ.var selects the store variable from a box or diamond.

nextthrow-catch :

(is_box(φ) $\vee$ is_dia(φ)) $\wedge$ φ.var = s
$\wedge$ (is_finally(φ.stm) $\vee$ is_catches(φ.stm) $\vee$ is_catch(φ.stm) $\vee$ is_endstatic(φ.stm))
$\rightarrow$ next_throw_catcher(s, φ) = φ

φ.stm selects the statement from a box or diamond.

nextthrow-rec :

(is_box(φ) $\vee$ is_dia(φ)) $\wedge$ φ.var = s
$\wedge$ $\neg$ (is_finally(φ.stm) $\vee$ is_catches(φ.stm) $\vee$ is_catch(φ.stm) $\vee$ is_endstatic(φ.stm))
$\rightarrow$ next_throw_catcher(s, φ) = next_throw_catcher(s, φ.fma)

φ.fma selects the formula following a box or diamond.

Then the proof rule is identical to the normal throw rule except that next_throw_catcher(st, φ) is used instead of φ:

$$\frac{\begin{array}{ll}1. & \Gamma, st[mode] \neq normal \vdash \varphi, \Delta\\ 2. & \Gamma, st[mode] = normal, e = null \vdash \langle st/\texttt{throw new NullPointerException();}\rangle\ \varphi, \Delta\\ 3. & \Gamma, st[mode] = normal, e \neq null, st_0 = st[mode, throw(e, ty)]\\ & \quad \vdash next_throw_catcher(st, \varphi)[st/st_0], \Delta\end{array}}{\Gamma \vdash \langle st/\texttt{throw } e;\rangle\ \varphi, \Delta}$$

The same optimization can be done for `return` and `break` statements with corresponding functions *next_return_catcher* and *next_break_catcher*. This is a nice and useful optimization since it saves a number of proof steps.

4.6.5 While invariant

This is a version of the classical Hoare invariant rule for while loops that does not include termination. Since the usual proofs also include termination this rule serves more for educational purposes than anything else. However, it might be useful to prove the termination of a while loop and an invariant property separately.

$$\frac{\begin{array}{ll}1. & \Gamma \vdash \psi, \Delta \\ 2. & \psi \vdash [st/x = e;][st/\texttt{if}(x)\ \alpha](\quad (\quad x \wedge st[mode] = normal \rightarrow \psi) \\ & \qquad\qquad\qquad\qquad\qquad\qquad \wedge\ (\neg\ (x \wedge st[mode] = normal) \rightarrow \varphi)))\end{array}}{\Gamma \vdash [st/\texttt{while}(e)\ \alpha]\varphi, \Delta}$$

ψ is the invariant; it must hold initially (premise 1), and we can assume it at the beginning of every iteration (premise 2). Since the test e of the loop is an arbitrary Java expression it must be evaluated inside a box. Since it may contain side effects it must be evaluated exactly once in every loop iteration. This is done by $[st/x = e;]$. x is a new variable that records the result of the test so that we can access it more than once in the following formula.

The body of the loop is evaluated only if the test evaluates to true (and completes normally), $[st/\texttt{if}(x)\ \alpha]$. A Java `while` loop terminates if either the test evaluates to false or the test or the loop body do not complete normally. This means the invariant must be established after the loop body if x is true and the mode is normal, $x \wedge st[mode] = normal \rightarrow \psi$. Otherwise the loop terminates and the postcondition φ must be proved, $\neg\ (x \wedge st[mode] = normal) \rightarrow \varphi$.

Precondition: (none)

Variables: x = newvar(boolean, $vars(e) \cup free(\varphi) \cup free(\psi)$).

This version of the invariant rule is simpler than, for example, von Oheimb's [vO01] because his rule requires two invariants if the test has side effects (the second must hold after the test, before the body is evaluated.

4.6.6 Bounded While

Classical dynamic logic uses a bounded loop construct to handle `while` loops. This can be mimicked in our calculus by replacing a `while` statement in a diamond by a `for` loop. The following equivalence holds:

$$\begin{array}{l}\quad \langle \text{st}/\texttt{while}\ (\text{e})\ \alpha\rangle\ \varphi \\ \leftrightarrow \exists\ \text{x}.\ 0 < \text{x} \wedge \langle \text{st}/\texttt{y = true;}\rangle\ \langle \text{st}/\texttt{for(; 0 < x \&\& y; x--) \{}\ \text{y} = \text{e; if}\ (\text{y})\ \alpha\ \texttt{\}}\ \rangle \\ \qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad (\neg\ (\text{y} = \text{true} \wedge \text{st[mode]} = \text{normal}) \wedge \varphi)\end{array}$$

The idea is to limit the number of iterations by introducing a counter x. If the `while` loop terminates there exists (a value for) x such that the loop terminates after x iterations, and afterwards φ holds. Since the test e of the `while` loop may contain side effects or throw an exception it is important that it is evaluated in the `for` loop exactly the same number of times and in the same situations as in the `while` loop. The variable y is needed to state that the last test was indeed **false** (unless the loop terminated with a jump).

Precondition: $\neg$ is_locvardecl(α)

Variables: x = newvar(int, free(e) $\cup$ free(α) $\cup$ free(φ)).

y = newvar(boolean, free(e) $\cup$ free(α) $\cup$ free(φ)).

4.6.7 For invariant

With the focus on Java Card, `for` loops that iterate across an array are very common. Properties can be proved by induction, but it may also be convenient to introduce a specialized invariant rule for these cases. One possibility is the following proof rule:

$$\frac{\begin{array}{ll}1. & \Gamma \vdash \psi \wedge 0 \leq sb2int(i) \wedge sb2int(i) \leq sb2int(e) \wedge inrange(i,e), \Delta \\ 2. & \psi \wedge 0 \leq sb2int(i) \wedge sb2int(i) \leq sb2int(t) \\ & \quad \vdash \langle st/x = (i < e); \rangle \langle st/\texttt{if}(x)\texttt{\{}\alpha\ \texttt{i++\}}\rangle (\quad (\quad x \wedge st[mode] = normal \rightarrow \psi) \\ & \qquad\qquad\qquad\qquad\qquad\qquad \wedge (\neg\ (x \wedge st[mode] = normal) \rightarrow \varphi)\end{array}}{\Gamma \vdash [st/\texttt{for(;i < e; i++)}\alpha]\varphi, \Delta}$$

ψ is the invariant. Since the variable i is intended as an array index $0 \leq i \wedge i \leq e$ automatically becomes part of the invariant. Java Card has no integers, so i will be a byte or short value. To ensure proper termination we must ensure that we have no overflow in this case. This is guaranteed by $inrange(i, e)$: The formula depends on the sort of i: If i is a byte then $inrange(i, e) \equiv e < 128$, if i is a short then $inrange(i, e) \equiv t < 32768$, otherwise it is `true` (integers are unlimited). $sb2int(i)$ converts short or byte values to integer. This means that i and e can be of different sorts, for example byte and short. e must be a basic expression (either a local variable access or a literal). The rule is tailored for counting upwards (`i++`), and termination is obvious. The rest of the rule is similar to the invariant rule for the while loop (chapter 4.6.5). Of course, i may not be modified in the body of the loop, and we also have other variable requirements:

Precondition: $\neg$ is_locvardecl(α) $\wedge \neg$ i $\in$ asgvars(α) $\wedge \neg$ i $\in$ vars(e) $\wedge$ vars(t) $\cap$ asgvars(α) $= \emptyset$

Variables: x = newvar(boolean, vars($i < e$) $\cup$ free(α) $\cup$ free(φ) $\cup$ free(ψ)).

Similar invariant rules could be added for `do` loops, however they seem to be used very seldom in practise. And it is always possible to use Noetherian induction and the 'standard' `do` rule (chapter 4.4.6) for practical applications.

4.7 Predicate Logic Proof Rules

The calculus also contains the usual proof rules for first-order logic and induction. However, there is one catch: substitution is (usually) not possible in program formulas. For example, it is not possible to substitute x by 3 in $\langle$st/x = x + 1;$\rangle$ φ because x occurs on the right-hand side of an assignment. This means that the proof rules *all_left* and *exists_right* must be modified:

$$\frac{y = \tau, \varphi[x/y], \forall\ x.\varphi, \Gamma \vdash \Delta}{\forall\ x.\varphi, \Gamma \vdash \Delta}\ \text{(all left)} \qquad \frac{\Gamma \vdash y = \tau, \varphi[x/y], \exists\ x.\varphi, \Delta}{\Gamma \vdash \exists\ x.\varphi, \Delta}\ \text{(exists right)}$$

y is a new variable. For formulas φ without diamonds and boxes the normal substitution can be used.

Precondition: (none)

Variables: y = newvar(x, vars(τ) $\cup$ (vars(φ) $\setminus \{x\}$) $\cup$ free(Γ) $\cup$ free(Δ))

In the special case that the term to substitute is a variable that does not occur in the formula, replacement can be used instead of substituion. For example, the following equivalence holds:

$$\neg\ y \in \text{vars}(\varphi) \wedge tds, \mathcal{A}, v \models x = y \rightarrow (\varphi \leftrightarrow \varphi[x/y])$$

The proof rules for an existential quantifier in the antecedent or a universal quantifier in the succedent can use replacement instead of substitution:

$$\frac{\varphi[x/y], \Gamma \vdash \Delta}{\exists\ x.\varphi, \Gamma \vdash \Delta}\ \text{(exists left)} \qquad \frac{\Gamma \vdash \varphi[x/y], \Delta}{\Gamma \vdash \forall\ x.\varphi, \Delta}\ \text{(all right)}$$

Precondition: (none)

Variables: y = newvar(x, (vars(φ) $\setminus \{x\}$) $\cup$ free(Γ) $\cup$ free(Δ))

4.8 Soundness of the Calculus

4.8.1 Theorem (soundness)

If the type declarations of the context and the conclusion of the proof rule are *primitive type correct* (discussed in detail in chapter 5), and the preconditions of the proof rules are met then the proof rules presented in this chapter are sound. Let $\frac{p_1 \dots p_n}{c}$ be a proof rule, then

$$\text{primtc(tds)} \wedge \text{primtc(c)} \wedge \text{tds}, \mathcal{A} \models p_i \text{ for all } 1 \leq i \leq n \Rightarrow \text{tds}, \mathcal{A} \models c$$

Proof

Formally proved with KIV.

The majority of rules are proved by applying the semantics definitions and using coincidence and replacement lemmas as appropriate. The two invariant rules and the rule for the bounded while loop use induction on the number of semantics rule applications (see chapter 3.4). It is certainly not useful to describe every proof in detail (some remarks have been made above). Rather, we present some important lemmas concerning the treatment of variables.

4.8.1 Unchanged variables

One important property is that only *assigned* variables can change their value: If initially x has value 1 then after an assignment `x = 3;` its value may be 3 (it may also be unchanged if the assignment is skipped due to a jump):

$$v[x] = 1 \wedge \text{st[mode]} = \text{normal} \rightarrow (v \times st)[\![\texttt{x = 3}]\!](v[x, 3] \times st)$$
$$v[x] = 1 \wedge \text{st[mode]} \neq \text{normal} \rightarrow (v \times st)[\![\texttt{x = 3}]\!](v \times st)$$

This leads to the definition of *asgvars* (assigned variables) for expressions, statements, expression lists, and statement lists. The definition for expressions is straightforward. Only the following expressions contain assigned variables:

- local variable assignment, e.g. `x = 3`
- increment/decrement expression with a local variable, e.g. `x++`
- compound assignment to a local variable, e.g. `x+=5`

For statements the definition is a little bit more complicated because local variables are (usually) not assigned variables, for example:

$$\begin{array}{l} v[x] = 1 \wedge \text{st[mode]} = \text{normal} \\ \rightarrow (v \times st)[\![\texttt{\{ int x = 3; if (x > 0) y = 5; \}}]\!](v[y, 5] \times st) \end{array}$$

Only y is modified. The original value of the local variable x is restored at the end of a block (even in case of a jump). We repeat the semantics rules for blocks:

$$\frac{st[mode] \neq normal}{(v \times st)[\![\{\alpha_1 \dots \alpha_n\}]\!](v \times st)}$$

$$\frac{st[mode] = normal \quad (v \times st)[\![\alpha_1 \dots \alpha_n]\!](v_0 \times st_0)}{(v \times st)[\![\{\alpha_1 \dots \alpha_n\}]\!](restore(v_0, locvars(\alpha_1 \dots \alpha_n), v) \times st_0)}$$

However, unless we make assumptions about the Java program, a local variable declaration can occur outside a block (more precisely: not as a top-level statement of a block). In this case the semantics treats it simply as an assignment, and it is considered as an assigned variable. So for statements we have the following cases:

- asgvars(`{`$\alpha_1 \dots \alpha_n$`}`) = asgvars($\alpha_1 \dots \alpha_n$) \ locvars($\alpha_1 \dots \alpha_n$)

  The local variables are not assigned. *locvars* selects all variables of top-level local variable declarations:

  locvars([]) = $\emptyset$

  locvars(`ty x = e;`$\alpha_2 \dots \alpha_n$) = $\{x\}$ ∪ locvars($\alpha_2 \dots \alpha_n$)

  ¬ is_locvardecl(α) → locvars($\alpha\ \alpha_2 \dots \alpha_n$) = $\{x\}$ ∪ locvars($\alpha_2 \dots \alpha_n$)

- asgvars(`ty x = e;`) = $\{x\}$ ∪ asgvars(e)

  A local variable declared outside a block is considered as assigned.

- asgvars(`catch(ty x)` α) = asgvars(α) \$\{x\}$

  The variable declared in a catch clause is never assigned.

- asgvars(`targetexpr(x, ty)`) = $\{x\}$

  A targetexpr is used in the calculus to catch returned values and assigns `x` to the value.

All other statements just compute the union of the assigned variables of their substatements and expressions. This definition can be lifted to method declarations (the formal parameter variables and `this` are not assigned) and type declarations. *Primitive type correct* type declarations have no assigned variables, primtc(tds) → asgvars(tds) = $\emptyset$. Then the theorem holds that not-assigned variables do not change their value:

4.8.2 Theorem (unchanged variables)

1. $(v \times st)[\![\alpha]\!]_{tds}(v_0 \times st_0) \to \forall\, x.x \notin \text{asgvars}(\alpha) \cup \text{asgvars}(tds) \to v[x] = v_0[x]$
2. $(v \times st)[\![e]\!]_{tds}(v_0 \times st_0 \times val_0) \to \forall\, x.x \notin \text{asgvars}(e) \cup \text{asgvars}(tds) \to v[x] = v_0[x]$
3. $(v \times st)[\![\alpha_1 \dots \alpha_n]\!]_{tds}(v_0 \times st_0) \to \forall\, x.x \notin \text{asgvars}(\alpha_1 \dots \alpha_n) \cup \text{asgvars}(tds) \to v[x] = v_0[x]$
4. $(v \times st)[\![e_1 \dots e_n]\!]_{tds}(v_0 \times st_0 \times vals_0)$
   $\to \forall\, x.x \notin \text{asgvars}(e_1 \dots e_n) \cup \text{asgvars}(tds) \to v[x] = v_0[x]$

The theorem comes in four versions, for statements, expressions, statement lists, and expression lists.

Because of the mutually recursive definitions all four theorems must be proved at once. The generalization makes use of the equivalence of the semantics relation with their corresponding procedures (as explained in chapter 3.4):

```
(∀ v, st, jstm, tds, v0, st0, y.
        ⟨semstm(v, st, jstm, tds; v1, st1): m ⟩ (v1 = v0 ∧ st1 = st0)
      ∧ ¬ y ∈ asgvars(jstm) ∧ ¬ y ∈ asgvars(tds) → v[y] = v0[y])
∧ (∀ v, st, jstms, tds, v0, st0, y.
        ⟨semstms(v, st, jstms, tds; v1, st1): m ⟩ (v1 = v0 ∧ st1 = st0)
      ∧ ¬ y ∈ asgvars(jstms) ∧ ¬ y ∈ asgvars(tds) → v[y] = v0[y])
∧ (∀ v, st, jexpr, tds, v0, st0, val0, y.
        ⟨semexpr(v, st, jexpr, tds; v1, st1, val1): m ⟩
                (v1 = v0 ∧ st1 = st0 ∧ val1 = val0)
      ∧ ¬ y ∈ asgvars(jexpr) ∧ ¬ y ∈ asgvars(tds) → v[y] = v0[y])
∧ (∀ v, st, jexprs, tds, v0, st0, vals0, y.
        ⟨semexprs(v, st, jexprs, tds; v1, st1, vals1): m ⟩
                (v1 = v0 ∧ st1 = st0 ∧ vals1 = vals0)
     ∧ ¬ y ∈ asgvars(jexprs) ∧ ¬ y ∈ asgvars(tds) → v[y] = v0[y])
```

The theorem is proved by induction on m and unfolding of the procedures. It shows the property for one variable only. Lifting it to all unassigned variables (as in the theorems above) is trivial.

4.8.2 Coincidence Lemma

In predicate logic we have the property that the value of a formula depends only on the values of the free variables:

$$(\forall\, x.x \in \text{free}(\varphi) \rightarrow v[x] = w[x]) \wedge \text{is_plfma}(\varphi) \rightarrow (tds, \mathcal{A}, v \models \varphi \leftrightarrow tds, \mathcal{A}, w \models \varphi)$$

is_plfma(φ) is true if φ is a predicate logic formula. The same property holds for our Java dynamic logic:

$$(\forall\, x.x \in \text{free}(\varphi) \rightarrow v[x] = w[x]) \wedge \text{free}(tds) = \emptyset \rightarrow (tds, \mathcal{A}, v \models \varphi \leftrightarrow tds, \mathcal{A}, w \models \varphi)$$

The lemma is very important in the correctness proofs when new variables are introduced. Obviously, this means that the usual definition of free variables must be extended to Java statements, expressions, and type declarations.

All variables of a Java expression are defined as free variables, even an assignment to a local variable. The reason is that it is not guaranteed that the assignment really happens. The validity of the formula

$$tds, \mathcal{A}, v \models \text{st[mode]} \neq \text{normal} \rightarrow \langle \text{st/}\texttt{x = 3;}\rangle\ \text{x} = 3$$

depends on the initial value v[x] because the assignment is skipped. It is possible to define the first occurrence of x in the formula as not free, and only the second as free. However, this makes some properties much more complicated to prove, and is pointless. The definition implies that the assigned variables of an expression are a subset of the free variables. For statements we have a similar situation as with assigned variables: local variables are (usually) not free variables. However, some nasty effects can occur. For example, in the statement

```
{ if (x < 0) throw new Exception(); int x = 5; }
```

`x` is used (read) before it is declared as a local variable. The semantics of the statement depends on the initial value of `x`. Therefore, `x` is free. If the order of the statements is changed `x` is not free:

```
{ int x = 5; if (x < 0) throw new Exception(); }
```

Another problem are the newly introduced statements because they can act as catchers for exceptions. For example in

```
{ throw new Exception();
  int x = 5;
  finally { if (x < 0) throw new Error();}
}
```

an isolated `finally` statement occurs. The exception is raised by the `throw`, and the local variable declaration is skipped. Then the exception is caught, and the body of the `finally` statement is evaluated. If $\texttt{x} < 0$ an error is thrown, otherwise the exception is re-thrown when the `finally` block is left. Since the local variable declaration was skipped the result depends on the initial value of `x`. Hence, `x` is free. The same problem can occur in a switch block:

```
switch (i) {
  case 0: int x = 3;
  case 1: if (x < 0) throw new Exception();
}
```

If i is 1 the local variable declaration is skipped and the result depends on the initial value of `x`. All three statements are illegal in Java (and the proof rules never introduce a `finally` statement inside a block). This requires a design decision:

1. To define the free variables in such a manner that the coincidence lemma holds for all formulas and statements (even for 'illegal' statements).

2. To specify 'legal' programs and prove the coincidence lemma only for 'legal' formulas.

The first choice means that the definition of free variables becomes a little bit more complicated, the second requires the definition of 'legal' for these cases. As a matter of fact, 'legal' is defined with the help of 'free', and the first possibility was chosen. This means the coincidence lemma holds for arbitrary programs.

We show the important definitions for statements.

- free(α) = free($\emptyset$, α)

  The specification of free(α) uses an auxiliary function that keeps track of the currently defined local variables. In free(vs, α), vs are the defined local variables (or formal parameter variables inside method declarations).

- free(vs, `if` (e) α `else` β) = (vars(e) $\setminus$ vs) $\cup$ free(vs, α) $\cup$ free(vs, β)

  The variables of the test of an if statement are free, except if they are local (vars(e) $\setminus$ vs) and the free variables of the `if` and `then` part are computed with the same set of locals.

- free(vs, `ty x = e;`) = (vars(e) $\cup$ { x }) $\setminus$ vs

  A local variable declaration not inside a block is a free variable.

- $\neg$ may_catch($\alpha_1 \dots \alpha_n$) $\rightarrow$ free(vs, `{`$\alpha_1 \dots \alpha_n$`}`) = freeblock(vs, $\alpha_1 \dots \alpha_n$)

  In case a block does not contain any statements that may catch a jump (as in the second example above), another auxiliary function freeblock is used. It keeps track of the order in which statements occur (first example above).

  freeblock(vs, []) = $\emptyset$

  freeblock(vs, `ty x = e;` $\alpha_2 \dots \alpha_n$) = (vars(e) $\setminus$ vs) $\cup$ freeblock(vs $\cup$ $\{x\}$, $\alpha_2 \dots \alpha_n$)

  Inside a block a local variable is not free, and added to the set vs for the rest of the block.

  $\neg$ is_locvardecl(α) $\rightarrow$ freeblock(vs, α $\alpha_2 \dots \alpha_n$) = free(vs, α) $\cup$ freeblock(vs, $\alpha_2 \dots \alpha_n$)

- may_catch($\alpha_1 \dots \alpha_n$) $\rightarrow$ free(vs, `{`$\alpha_1 \dots \alpha_n$`}`) = free(vs, $\alpha_1 \dots \alpha_n$)

  If the block contains a catcher local variables are treated as free:

  free(vs, []) = $\emptyset$

  free(vs, α $\alpha_2 \dots \alpha_n$) = free(vs, α) $\cup$ free(vs, $\alpha_2 \dots \alpha_n$)

- free(vs, `switch (e) {`$\alpha_1 \dots \alpha_n$`}`) = (free(e) $\setminus$ vs) $\cup$ free(vs, $\alpha_1 \dots \alpha_n$)

  If the body of a `switch` statement is a block it is treated like a block that may catch jumps (see the third example above).

The definitions for other statements follow this pattern. Free variables can be defined for method declarations (the formal parameter are local, free(m(params) α) = free(vars(params), α) for static methods, for instance methods `this` is also added to the locals) and other declarations, and finally for a list of Java type declarations. Legal type declarations have no free variables, free(tds) = $\emptyset$. This is a precondition for the whole calculus since the type declarations are a global context. The definitions imply that the free variables are a superset of the assigned variables.

The coincidence lemma for formulas is then reduced to a coincidence lemma for Java statements:

4.8.3 Theorem (coincidence)

$$\begin{array}{l} \quad (v \times st)[\![\alpha]\!](v_0 \times st_0) \\ \wedge\ (\forall\, x.x \in \mathrm{free}(\alpha) \cup \mathrm{free}(tds) \rightarrow v[x] = w[x]) \\ \rightarrow (w \times st)[\![\alpha]\!](\mathrm{restore}(w, \mathrm{free}(\alpha) \cup \mathrm{free}(tds), v_0) \times st_0) \end{array}$$

This theorem is a generalization because it states

1. that both computations return the same store st_0,
2. and exactly what the resulting variable mapping of the second computation is:

   All variables that are not free remain unchanged from initial to final variable mapping, and free variables have the same result value as in the first computation.

   This is expressed by $\text{restore}(w, \text{free}(\alpha) \cup \text{free}(tds), v_0)$: It is w except for variables in $\text{free}(\alpha) \cup \text{free}(tds)$. For those variables it is v_0:

   $x \in \text{free}(\alpha) \cup \text{free}(tds) \rightarrow \text{restore}(w, \text{free}(\alpha) \cup \text{free}(tds), v_0)[x] = v_0[x]$

   $x \notin \text{free}(\alpha) \cup \text{free}(tds) \rightarrow \text{restore}(w, \text{free}(\alpha) \cup \text{free}(tds), v_0)[x] = w[x]$

The theorem for an expression additionally states that the computed value of the expression is identical. The theorems are again proved by a generalization that includes statements, expressions, expression lists, and statement lists, including another generalization for statement lists that considers both auxiliary definitions $\text{freeblock}(\alpha_1 \ldots \alpha_n)$ and $\text{free}(\alpha_1 \ldots \alpha_n)$.

It can be noted that the coincidence lemma is not really needed, because there never occurs a situation where two variable mappings differ in infinitely many variables. What is needed is a coincidence lemma for two variable mappings that differ only in finitely many (or only one) variable, for example

$$\begin{aligned} & (v \times st)[\![\alpha]\!](v_0 \times st_0) \wedge \neg\, y \in \text{free}(\alpha) \cup \text{free}(tds) \\ \rightarrow\ & (v[y, val] \times st)[\![\alpha]\!](v_0[y, val] \times st_0) \end{aligned}$$

4.8.3 Replacing variables

As mentioned previously, it is not possible to define a substitution for Java programs that is always applicable. We can, however, replace a variable by another (new) variable in an arbitrary formula (written as $\varphi[x, y]$, every occurrence of x is replaced by y; the definition is trivial and omitted). Then replacing a variable $\varphi[x, y]$ is like modifying the variable mapping v to $v[x, v[y]]$.

4.8.4 Theorem (replacement)

$$\begin{aligned} & \neg\, y \in \text{vars}(\alpha) \wedge \text{free}(tds) = \emptyset \wedge sort(x) = sort(y) \\ \rightarrow\ & (tds \times \mathcal{A} \times v \models \varphi[x, y] \leftrightarrow tds \times \mathcal{A} \times v[x, v[y]] \models \varphi) \end{aligned}$$

This is similar to the substitution lemma for predicate logic. It is based on corresponding theorems for Java statements and expression. For statements we have

$$\begin{aligned} & \neg\, y \in \text{vars}(\alpha) \wedge \text{free}(tds) = \emptyset \wedge sort(x) = sort(y) \\ \rightarrow\ & (\quad (v \times st)[\![\alpha[x, y]]\!](w \times st_0) \\ & \quad \leftrightarrow (v[x, v[y]] \times st)[\![\alpha]\!](w[y, v[y]][x, w[y]] \times st_0) \wedge v[x] = w[x]) \end{aligned}$$

Again the theorem is generalized because a new variable mapping is computed. Replacing one variable can be extended to replacing a list of variables. 26 proof rules use replacements.

This finishes the description of the calculus. The next chapter is concerned with type correctness.

Chapter 5

Type Safety

This chapter deals with types, type safety, and type soundness for Java programs. Type soundness for subsets of Java have been proven by Drossopoulou and Eisenbach [DE99], by Syme [Sym99], and by Oheimb [vON99] [vO01]. The first two works investigate type safety for its own sake (probably with the prominent type unsound language Eiffel [Coo89] in mind). The last seems to be the only work where type soundness and soundness of a calculus have been proven with respect to the same semantics. The calculus makes use of the type soundness result (the run-time class of a method invoker is always a subclass of its static type). There exist calculi that assume type soundness without having a formal specification for it.

A novelty in this work is the introduction of a weaker kind of type safety, namely *primitive type safety* in addition to the usual full type safety. Full type safety for Java guarantees that the run-time class of an object is a subclass (or equal) to the static class for an expression. This implies that every dynamic method lookup finds an applicable method declaration, that every field access accesses an existing field etc. However, these properties are only true if the store is *compatible* to the program. And this is a very strong requirement as will be explained in chapter 6. Full type safety is discussed in the second half of this chapter.

The calculus for Java presented in this work is embedded in an algebraic setting. The store, the primitive types and their operations, and auxiliary operations are specified algebraically. The sort system for algebraic specifications in KIV is a many sorted flat system. Consider the (type-incorrect) Java program `boolean x = 3;`. The proof rules would produce an equation like $x = 3$. This equation must be well-sorted, so the question is: What is the sort of x? In this work, Java local variables are logical variables. This means that `x` in `boolean x = 3;` is a logical variable with a sort. What is the sort of `x`? There are two possibilities:

1. The sort of x is *javavalue* (see chapter 3.1.1), a sort that contains all possible Java values (integer, short, byte, boolean, and reference).

   This means that the proof rule that introduces the equation must convert the value 3 to a *javavalue*: x = intval(3). This guarantees that the equation is well-sorted.

2. The sort of x corresponds to its Java type: sort *integer* for type `int`, *short* for `short`, *byte* for `byte`, *bool* for `boolean`, and *reference* for a reference type, i.e. a class or array type.

   This means that trying to apply a proof rule introducing the equation $x = 3$ would produce an error (essentially this means the proof rule is not applicable, and the proof gets 'stuck'), or to reject the program from the start.

Consider the Java statement `x = y.f;` with a field access `y.f`. Applying a proof rule will yield an equation like $x = st[y - f]$, i.e. the value of the field is looked up in the store. The problem is that the store contains only *javavalue*'s (see chapter 3.1.3). If x is not of sort *javavalue* we have again an ill-sorted equation. Again we have two possibilities:

1. The sort of x is *javavalue*. No problem.

2. The sort of x is a sort corresponding to its Java type. Then we must convert the *javavalue* from the store to the correct sort by applying the correct selector, e.g. $st[y - f].intval$.

   The question is: Is this correct? The problem is that there is no guarantee that the value retrieved from the store is indeed an integer value. And selecting an *.intval* from a boolean value returns an unspecified value.

Consider the Java statement `y.f = z.g;`, the value of one field is assigned to another field. Here, only the *flattening* rule is applicable that introduces a new variable: `x = z.g; y.f = x;`. Again the question is: what is the sort of `x`?

1. The sort of x is *javavalue*. No problem.

2. Otherwise the question is again: Is this correct? If the semantics defines that the value of `z.g` is lookup up in the store and stored in the field `y.f` as $st[y - f, st[z - g]]$ there is a problem: Since the store contains only *javavalue*'s the value is not modified, i.e. the key $y - f$ contains afterwards the same value as $z - g$. If the proof rules convert the value to something else we might end up with something like $st[y - f, intval(st[z - g].intval)]$, i.e. we select from the javavalue its content and convert it again to a javavalue. However, $intval(val.intval) = val$ is only true if the value is indeed an integer value. Otherwise the value is modified in an unspecified way.

The first solution to all these problems (all variables have sort *javavalue*) sounds attractive, but means that we ultimately work more or less unsorted. This is ridiculous in the context of Java ("The Java programming language is strongly typed." – JLS p. 1) Types reduce programming errors in programming languages, and sorts reduce errors in algebraic specifications.

Therefore we introduce the notion of a "primitive type correct" Java program in the next section.

5.1 Primitive Type Correctness

5.1.1 Java Types, Sorts, and Values

The primitive types in Java that are considered in this work are `boolean`, `int`, `short`, and `byte`. If we forget the class hierarchy and the difference between arrays and objects we can view class and array types as another primitive type `reference`. `void` plays a special role. A primitive type correct Java program has the property that

1. the value computed by an expression is compatible with its (primitive) type, and that

2. every local variable has a value that is compatible with its sort.

To summarize, we have the following correspondence between Java types, sorts, and values:

Java type	**sort**	**value**
boolean	bool	bool
int	int	integer
short	short	short
byte	byte	byte
array/class type	reference	reference
void	reference	reference

The predicate okval(sort, value) is true if the sort matches the value in accordance to the table. okvals(v) is true for a variable mapping v if the value of every variable matches its sort.

In the sequel we will define a predicate *primtc* that is true if a Java expression, statement, and list of type declarations is *primitive type correct*. As mentioned in chapter 2.4 we assume that the compiler makes some primitive (widening/narrowing) conversions explicit. If this assumption is

met, then a Java program that passes a standard Java compiler without an error is primitive type correct.

We repeat the compiler assumption for primitive conversion from chapter 2.4:

1. The compiler should make these conversion explicit by adding a cast for the following expressions:

   local variable assignment, static field assignment, instance field assignment, array assignment, conditional expression, arguments for method and constructor calls, local variable declaration, return statement

2. The compiler does not have to add explicit casts for the following expressions:

   unary/binary/exception binary expression, indices in array access/assignment/creation, increment/decrement, compound assignment, switch statement/labels.

The reason for this distinction is pragmatic: The first conversions require an explicit treatment in the semantics. This means the semantics rules must be modified (extended, made more complicated) to incorporate issues that are only slightly related to Java's run-time behavior (the conversion depends on static types). On the other hand, the conversions in item 2 can be incorporated into the definition of auxiliary operations, for example addition, because the result of an addition is always an integer value (the conversion depends not on static types). Furthermore, the conversions in item 1 occur less often than those in item 2 so that adding the explicit casts does not impinge readability.

5.1.2 Return and Break Statements

The definition of *primtc* is rather simple except for a method declaration that returns a value. Most other formalizations (e.g. [vO01]) expect that a method body consists of statements without `return`'s, and one special variable that returns the value. In this work a method body is a statement that may contain arbitrary `return` statements. (For a (primitive) type correct program it is possible to automatically transform an arbitrary method body such that it contains only one return statement at the end. This can be done by introducing a new variable for the result and by – basically – replacing a `return` statement by an assignment to this variable followed by a break to a new label. This is definitely not a trivial transformation. Furthermore, we are defining what a type correct program is. So it cannot be argued that the compiler should do this transformation.)

A method body is guaranteed to return a value that is compatible to the declared return type if

- every execution path through the method body ends with a `return` statement of the correct type, or ends with a `throw`, or does not end at all, and
- the method body does not contain a `return` statement with an incompatible type.

The first property is checked by the predicate *primtcReturn*(α, type), the second by *primtcReturnsOk*(α, type). JLS discusses these properties only partially and rather implicitly in chapter 14.20 (named 'unreachable statements'). We discuss some issues.

- There are obviously two execution paths through an `if` statement: The `then` and the `else` part. Hence, *primtcReturn*(`if (e)` α `else` β, ty) is true iff α and β return correctly:

  primtcReturn(`if (e)` α `else` β, ty) $\leftrightarrow$ primtcReturn(α) $\wedge$ primtcReturn(β)

- A loop (e.g. a `while` statement) returns correctly iff the test is `true`:

  primtcReturn(`while(e)` α, ty) $\leftrightarrow$ e = `true`

  A `while` loop with an (arbitrary) test e may not be entered at all. Therefore it is not hold that every execution path returns correctly. The exception is if the test is `true`: Then the loop either does not terminate at all or completes abruptly with a `return`, `break`, or `throw`. JLS treats loops with test `true` specially.

If the loop body contains a `return` statement it must be of the correct type. This is checked by the predicate *primtcReturnsOk*:

primtcReturnsOk(`while(e)` α, ty) $\leftrightarrow$ primtcReturnsOk(α)

The example shows that both definitions are needed.

- A `try` statement `try` α *catches* `finally` β returns correctly if either α and every catch clause returns correctly (in this case the finally block can never complete normally) or if the finally block β returns correctly (in this case it does not matter what the other statements do). This also requires that there are no incorrect return statements.

- A `switch` statement returns correctly if it has a `default` case (otherwise the switch block may be skipped) and if the complete switch block returns correctly (since cases 'fall through').

- A labeled statement returns correctly if the contained statement returns correctly and if it does not contain a `break` statement for the label. Otherwise we assume that we have an execution path that reaches the `break` statement. This means we ignore the possibility that it is unreachable, for example in `l : { return; break l; }` If the `break` statement can be reached the labeled statement completes normally. A labeled statement without a break for this label is useless, but legal in Java, and indeed occurs in the source code provided with JDK 1.4.

- Every break must be contained within a labeled statement with the same label. This requirement is necessary for the correct definition of *primtcReturn* (the existence of unreachable statements, for example, is not, though also illegal in Java). The reason is due to the semantics definitions: a `break` statement simply raises the *mode* in the store (see chapter 3.1.3) similar to a `return` or `throw` statement. The semantics for a method call deal only with a mode that is a *return*. This implies that a `break` without corresponding label acts like a `throw` and 'passes through' a method call, causing a non-local transfer of control. The effect is that an expression could raise a *break* mode. This, however, invalidates the test in the previous item – a `break` could occur before a `return` is reached and the labeled statement does not return correctly.

  To summarize: Allowing labeled statements without a `break` requires a check that every `break` is contained inside a corresponding labeled statement.

- A list of statements (of a block or switch block) returns correctly if either the remaining statements (without the first) return correctly, or if the first statement returns correctly and the following statements do not contain a switch label, or one of the newly introduced statements. If they contain a switch label then there may be an execution path that bypasses the first statement (in a switch block). The newly introduced statements act as 'catchers' for various kinds of jumps, so that the list of statements may complete normally even though the first statement returns correctly.

These definitions accept all legal Java programs, i.e. all Java programs that are accepted by a normal Java compiler are accepted as primitive type correct (provided the assumptions detailed in chapter 2.4 are met). As a test bench the full source code available with JDK 1.4 is accepted.

The definition of primitive type correctness can be extended to method declarations and Java type declarations (we assume primtc(tds) for the type declarations of the context), and to formulas and sequents. For the predicate logic parts of formulas *primtc* means that terms and predicates are well sorted.

The next section defines primitive type correctness *primtc* in detail and adds some more remarks.

5.2 Definition of Primitive Type Correctness

5.2.1 Primitive Type Correct Expressions

Several auxiliary definitions for *primtc* are used in the following axioms. We begin with a short description of them:

- primtc(ty), ty a Java type, is true if ty is not the **void** type, and does not contain the **void** type as the element type of an array (i.e. `void[]`). The **void** type may occur only as the (dummy) type of the `null` literal.
- sort(ty), ty a Java type, converts the type to the corresponding (algebraic) sort as defined in the previous section. sort(**void**) = reference, because the `null` literal may occur (almost) everywhere where an expression with a reference type may occur. (It may not occur, for example, as the invoking expression of a field access or method call, but as an argument for a method call or on the right-hand side of an assignment.)
- sort(t), t an algebraic term, returns the result sort of the term.
- primtc(t), t an algebraic term, is true if the term is well sorted.
- $ty_1 \leq_{wp} ty_2$, ty_1, ty_2 Java types, is true if ty_1 can be converted with *widening primitive conversion* to ty_2 (JLS 5.1.2): either $ty_1 = ty_2$, or ty_1 = `byte` and ty_2 = `short` or `int`, or ty_1 = `short` and ty_2 = `int`. $\leq_{wp}$ is also applicable to (algebraic) sorts.
- e.type, e a Java expression, selects the result type of an arbitrary expression (every expression has a result type).

In the rest of this section we will use the KIV definitions of expressions, statements, and types, not the mathematical notation because the types (usually hidden) are essential here.

- LiteralExpr :
  primtc(LiteralExpr(t, ty))
  $\leftrightarrow$ primtc(ty) $\wedge$ sort(ty) = sort(t) $\wedge$ primtc(t) $\vee$ ty = void $\wedge$ t = null

  A literal consists of an algebraic term and a Java type. Both must be *primtc* and their sorts must match, except the literal is the `null` literal which must have (dummy) type `void` (JLS 3.10.7: "A *null literal* is always of the null type.")

- primtc-UnaryExpr :
  primtc(UnaryExpr($\oplus$, e, ty))
  $\leftrightarrow$ primtc(ty) $\wedge$ primtc(e) $\wedge$ ty = resType($\oplus$) $\wedge$ e.type $\leq_{wp}$ argType($\oplus$))

  The type ty of the unary expression must be the correct result type of the unary operation (resType($\oplus$), either `int` or `boolean`), and the argument must be convertible to the argument type of operation (argType($\oplus$), again either `int` or `boolean`).

- primtc-PrimCast :
  primtc(PrimCast(ty_1, e, ty))
  $\leftrightarrow$ primtc(ty) $\wedge$ primtc(e) $\wedge$ ty = ty_1 $\wedge$ sort(ty) $\neq$ reference
  $\wedge$ (e.type $\leq_{wp}$ ty $\vee$ ty $\leq_{wp}$ e.type)

  A primitive cast can be used to convert byte, shorts, and integers in all directions. Therefore we have $\leq_{wp}$ in both directions.

- primtc-RefCast :
  primtc(RefCast(ty_1, e, ty))
  $\leftrightarrow$ primtc(ty) $\wedge$ primtc(e) $\wedge$ ty = ty_1 $\wedge$ sort(ty) = reference $\wedge$ sort(e.type) = reference

  The only requirement for a reference cast is that both types are reference types. For *primtc* it is irrelevant if ty_1 is a subtype of the type of e.

- primtc-InstanceExp :
  primtc(InstanceExpr(e, ty_1, ty))
  $\leftrightarrow$ primtc(ty) $\wedge$ primtc(e) $\wedge$ sort(e.type) = reference $\wedge$ sort(ty_1) = reference
  $\wedge$ ty = boolean

  `instanceof` has the same requirements as a reference cast (except for the result type that must be `boolean`).

- primtc-CondExpr :
  primtc(CondExpr(e_1, e_2, e_3, ty))
  $\leftrightarrow$ primtc(ty) $\wedge$ primtc(e_1) $\wedge$ primtc(e_2) $\wedge$ primtc(e_3)
  $\wedge$ e_1.type = boolean $\wedge$ sort(e_2.type) = sort(ty) $\wedge$ sort(e_3.type) = sort(ty)

  The test must be `boolean`, and the `then`- and `else`-part must have the same primitive type.

- primtc-CondBinExp :
  primtc(CondBinExpr(e_1, $\oplus$, e_2, ty))
  $\leftrightarrow$ primtc(ty) $\wedge$ primtc(e_1) $\wedge$ primtc(e_2)
  $\wedge$ ty = boolean $\wedge$ e_1.type = boolean $\wedge$ e_2.type = boolean

  Both arguments and the result must be `boolean`.

- primtc-BinaryExpr :
  primtc(BinaryExpr(e_1, $\oplus$, e_2, ty))
  $\leftrightarrow$ primtc(ty) $\wedge$ primtc(e_1) $\wedge$ primtc(e_2) $\wedge$ ty = binoptype($\oplus$)
  $\wedge$ ($\oplus$ = `==` $\vee$ $\oplus$ = `!=`
  $\rightarrow$ sort(e_1.type) = sort(e_2.type) $\vee$ e_1.type $\leq_{wp}$ e_2.type $\vee$ e_2.type $\leq_{wp}$ e_1.type)
  $\wedge$ ($\neg$ ($\oplus$ = `==` $\vee$ $\oplus$ = `!=`)
  $\rightarrow$ e_1.type $\leq_{wp}$ argType($\oplus$) $\wedge$ e_2.type $\leq_{wp}$ argType($\oplus$))

  A binary expression includes an automatic conversion of bytes and shorts to integers for the arithmetic operations. binoptype returns the result type of the binary operation, argType its argument types.

- primtc-ExBinExpr :
  primtc(ExBinExpr(e_1, $\oplus$, e_2, ty))
  $\leftrightarrow$ primtc(ty) $\wedge$ primtc(e_1) $\wedge$ primtc(e_2) $\wedge$ ty = int $\wedge$ e_1.type $\leq_{wp}$ ty $\wedge$ e_2.type $\leq_{wp}$ ty

  An exception binary operation is either `/` or `%`, the type must be integer.

- primtc-LocVarAccess : primtc(LocVarAccess(jvar, ty)) $\leftrightarrow$ primtc(ty) $\wedge$ sort(ty) = sort(jvar)

  The type of a local variable access must match the sort of the variable.

- primtc-SFieldAccess : primtc(SFieldAccess(fs, ty)) $\leftrightarrow$ primtc(ty) $\wedge$ fs.type = ty

  Fields are defined by a field specification (type of the field, field name, and class containing the field declaration), the type must be result type of the expression.

- primtc-FieldAccess :
  primtc(FieldAccess(e, fs, ty))
  $\leftrightarrow$ primtc(ty) $\wedge$ fs.type = ty $\wedge$ primtc(e) $\wedge$ sort(e.type) = reference

  The invoking expression of an instance field must have a reference type.

- primtc-ArrayAccess :
  primtc(ArrayAccess(e_1, e_2, ty))
  $\leftrightarrow$ primtc(ty) $\wedge$ primtc(e_1) $\wedge$ primtc(e_2) $\wedge$ sort(e_1.type) = reference $\wedge$ e_2.type $\leq_{wp}$ int

  The array expression must have a reference type, and the index must be convertible to `int`.

- primtc-LocVarAssign :
  primtc(LocVarAssign(jvar, e, ty))
  $\leftrightarrow$ primtc(ty) $\wedge$ primtc(e) $\wedge$ sort(ty) = jvar.stype $\wedge$ sort(ty) = sort(e.type)

  In an assignment the primitive types must be equal. The compiler should make primitive conversions explicit by introducing a cast.

- primtc-SFieldAssign :
  primtc(SFieldAssign(fs, e, ty))
  $\leftrightarrow$ primtc(ty) $\wedge$ fs.type = ty $\wedge$ primtc(e) $\wedge$ sort(ty) = sort(e.type)

- primtc-FieldAssign :
  primtc(FieldAssign(e_1, fs, e_2, ty))
  $\leftrightarrow$ primtc(ty) $\wedge$ fs.type = ty $\wedge$ primtc(e_1) $\wedge$ primtc(e_2)
  $\wedge$ sort(ty) = sort(e_2.type) $\wedge$ sort(e_1.type) = reference

- primtc-ArrayAssign :
  primtc(ArrayAssign(e_1, e_2, e_3, ty))
  $\leftrightarrow$ primtc(ty) $\wedge$ primtc(e_1) $\wedge$ primtc(e_2) $\wedge$ primtc(e_3)
  $\wedge$ sort(ty) = sort(e_3.type) $\wedge$ is_arraytype(e_1.type) $\wedge$ e_2.type $\leq_{wp}$ int

- primtc-CompAssign :
  primtc(CompAssign(e_1, $\oplus =$, e_2, ty))
  $\leftrightarrow$ primtc(ty) $\wedge$ primtc(e_1) $\wedge$ primtc(e_2)
  $\wedge$ ty = e_1.type $\wedge$ ty = asgoptype($\oplus =$) $\wedge$ e_2.type $\leq_{wp}$ ty

  asgoptype returns the result type of the assignment operation. We assume that $\oplus =$ is not just the operation, but also includes the desired result type.

- primtc-IncDecExpr :
  primtc(IncDecExpr($\oplus$, e, ty)) $\leftrightarrow$ primtc(ty) $\wedge$ primtc(e) $\wedge$ e.type = ty $\wedge$ ty $\leq_{wp}$ int

  The argument expression must be convertible to `int`.

- primtc-NewArray :
  primtc(NewArray(ty_1, es, i, ty))
  $\leftrightarrow$ primtc(ty) $\wedge$ primtc(es) $\wedge$ sort(ty) = reference $\wedge$ sorts(es.types) $\leq_{wp}$ int

  All dimension expressions of an array creation expression must be convertible to integer.

- primtc-ArrayInit :
  primtc(ArrayInit(es, ty)) $\leftrightarrow$ primtc(ty) $\wedge$ primtc(es) $\wedge$ sort(ty) = reference

  There is no requirement that the expressions of the array initializer are of the same (primitive) type.

- primtc-NewExpr :
  primtc(NewExpr(class, es, tys, ty))
  $\leftrightarrow$ primtc(ty) $\wedge$ primtc(es) $\wedge$ sort(ty) = reference $\wedge$ sorts(es.types) = sorts(tys)

  The primitive types of the actual arguments must be equal to the primitive types of tys that are ultimately used to find the correct constructor declaration.

- primtc-ConstrCall :
  primtc(ConstrCall(e, class, es, tys, ty))
  $\leftrightarrow$ primtc(ty) $\wedge$ primtc(e) $\wedge$ primtc(es)
  $\wedge$ sort(e.type) = reference $\wedge$ sort(ty) = reference $\wedge$ sorts(es.types) = sorts(tys)

  The primitive types of the actual arguments must be equal to the primitive types of tys that are used to find the correct constructor declaration.

- primtc-MethodCall :
  primtc(MethodCall(e, str, invmo, es, tys, ty))
  $\leftrightarrow$ primtc(ty) $\wedge$ primtc(e) $\wedge$ primtc(es)
  $\wedge$ sorts(es.types) = sorts(tys) $\wedge$ sort(e.type) = reference

  The result type of the method call must be primtc; this implies that it is not **void**. This definition is used for subexpressions. A **void** method call may appear only on top level, i.e. in an expression statement. This is captured by the following definition of primtcOrVoid.

- primtcOrVoid-method :
  primtcOrVoid(MethodCall(e, str, invmo, es, tys, ty))
  $\leftrightarrow$ (ty = void $\vee$ primtc(ty)) $\wedge$ primtc(es) $\wedge$ primtc(e)
  $\wedge$ sorts(es.types) = sorts(tys) $\wedge$ sort(e.type) = reference

  It is identical to the previous axiom except that the result type may be **void**. For all other expressions primtcOrVoid(e) $\leftrightarrow$ primtc(e).

The axioms for a primitive type correct statement are straightforward (tests must be of type `boolean`, etc.). The expression e of an expression statement e; and the updates in a `for` statement may be void, hence primtcOrVoid, all other expressions must be primtc.

5.2.2 Primitive Type Correct Java Type Declarations

Here, the definition of primitive type correctness is extended to member declarations (static initializer, field declaration, method declaration, or constructor declaration). The statements must be primtc, but now the requirements explained in chapter 5.1.2 concerning `return` and `break` statements become important. The predicate noReturn(α) is true if the statement does not contain any `return` statements, and breaksOk(labs, α) is true if all uncaught `break` statements have labels occurring the set of labels labs. For example, if α is

```
lab1 :  { if (e) break lab1; else break lab2; }
```

then breaksOk({ lab1, lab2 }, α) and breaksOk({ lab2 }, α), but not breaksOk($\emptyset$, α).

- primtc-statinit : primtc(statinit(α)) $\leftrightarrow$ primtc(α) $\wedge$ noReturn(α) $\wedge$ breaksOk($\emptyset$, α)

  A static initializer may not contain `return` statements (noReturn(α)), and may not contain uncaught `break` statements (breaksOk($\emptyset$, α)). Both requirements are essential because otherwise an expression could end with a mode that is neither normal nor a thrown exception. (An expression that causes a class initialization because of a first active use.) If an expression can raise a *return* mode a method call could return a value of the wrong type. An uncaught break can have a similar effect, as explained in chapter 5.1.2.

- primtc-fielddecl : primtc(fielddecl(mos, ty, fs)) $\leftrightarrow$ primtc(ty) $\wedge$ fs.type = ty

  A field declarations consists of a list of modifiers mos, the type of the field ty, and a field specification. Inititalization of fields happens in the static initializer or the constructor.

- primtc-methoddecl :
  primtc(methoddecl(mos, ty, m, params, α))
  $\leftrightarrow$ primtc(params) $\wedge$ primtc(α) $\wedge$ $\neg$ dups(params.vars) $\wedge$ $\neg$ THIS $\in$ params.vars
  $\wedge$ (ty = void $\vee$ primtc(ty) $\wedge$ primtcReturn(α, ty))
  $\wedge$ primtcReturnsOk(α, ty) $\wedge$ breaksOk($\emptyset$, α)

  A method declaration consists of a list of modifiers mos, its result type ty, the method name m, its formal parameters params, and the body α. The formal parameter variables params.vars may not contain duplicates or `this`. The method body may not contain uncaught `break` statements. If the method is a **void** method it may contain only `return` statements without an expression. This is checked by primtcReturnsOk(α, void). If the method is not **void** every execution path must end in a `return` (or throw, or not end at all) with the correct type.

- primtc-constrdecl :
  primtc(jconstrdecl(mos, class, params, α))
  $\leftrightarrow$ $\neg$ dups(params.vars) $\wedge$ $\neg$ THIS $\in$ params.vars $\wedge$ primtc(params)
  $\wedge$ primtc(α) $\wedge$ primtcReturnsOk(α, void) $\wedge$ breaksOk($\emptyset$, α)

  A constructor body may only contain `return` statements without an expression.

- primtc-td : primtc(td) $\leftrightarrow$ primtc(td.memdecls) $\wedge$ free(td) = $\emptyset$

  A type declaration td (a class or interface declaration) is primtc if every member declaration is. (td.memdecls is the list of member declarations) Furthermore, we include the requirements that it may contain no free variables. This is not really needed for all properties, but most of the proof rules require this (see chapter 4.8), so it is convenient to include here.

  Finally, a list of Java type declarations is *primtc* if every element is.

5.3 Properties of Primitive Type Correct Programs

5.3.1 Theorem (primitive type safety)

A primitive type correct Java expression that completes normally will always return a value that is compatible to its primitive type:

$$\begin{aligned}&(v \times st)[\![e]\!](v_0 \times st_0 \times val_0)\\ &\wedge \text{ okvals}(v) \wedge \text{primtc}(e) \wedge \text{primtc}(tds) \wedge st_0[mode] = normal\\ \rightarrow\ &\text{okval}(val_0, e.type)\end{aligned}$$

okvals and okval are defined in chapter 5.1.1. This property holds only if the expression completes normally; in case of an exception the dummy value *noval* is returned. Furthermore, an expression or statement will always return a correct variable mapping:

$$(v \times st)[\![e]\!](v_0 \times st_0 \times val_0) \wedge \text{okvals}(v) \wedge \text{primtcOrVoid}(e) \wedge \text{primtc}(tds) \rightarrow \text{okvals}(v_0)$$

$$(v \times st)[\![\alpha]\!](v_0 \times st_0) \wedge \text{okvals}(v) \wedge \text{primtc}(\alpha) \wedge \text{primtc}(tds) \rightarrow \text{okvals}(v_0)$$

The same holds for statement and expression lists. All properties must be proved at once. For methods that return a value the properties of breaksOk and primtcReturn become important. This requires

- $(v \times st)[\![\alpha]\!](v_0 \times st_0) \wedge \text{primtcReturn}(\alpha, ty) \wedge \text{primtc}(\alpha) \wedge \text{primtc}(tds)$
  $\rightarrow st_0[mode] \neq normal$

  stating that a statement for which primtcReturn holds never completes normally, which in turn requires

- $(v \times st)[\![e]\!](v_0 \times st_0 \times val_0) \wedge \text{primtc}(e) \wedge \text{primtc}(tds) \wedge \neg\ \text{is_break_mode}(st[mode])$
  $\rightarrow \neg\ \text{is_break_mode}(st_0[mode])$

  (an expression never raises a `break` mode) and

  $$\begin{aligned}&(v \times st)[\![\alpha]\!](v_0 \times st_0) \wedge \text{primtc}(\alpha) \wedge \text{primtc}(tds) \wedge \text{breaksOk}(\text{labs}, \alpha)\\ \rightarrow\ &\neg\ \text{is_break_mode}(st_0[mode])\\ &\vee\ \text{is_break_mode}(st_0[mode]) \wedge st_0[mode].\text{label} \in \text{labs}\\ &\vee\ \text{is_break_mode}(st_0[mode]) \wedge \text{is_break_mode}(st[mode]) \wedge st_0[mode] = st[mode]\end{aligned}$$

  Both properties must be proved together; the generalization for statements is necessary because α may occur inside a labeled statement.

These are properties of the semantics of Java programs. In itself they do not guarantee that the calculus does not produce incorrectly sorted formulas. However, it is an essential prerequisite.

We can extend the notion of a primitive type correct Java program to formulas and sequents. For the predicate logic parts (terms, predicates, equations etc.) we have the usual definitions (both terms in an equation have the same sort etc.). We show only the definition for diamonds and boxes:

primtc-dia :

$$\text{primtc}(\langle \text{st}/\alpha \rangle\ \varphi) \leftrightarrow \text{sort}(\text{st}) = \text{store} \wedge \neg\ \text{st} \in \text{vars}(\alpha) \wedge \text{primtc}(\alpha) \wedge \text{primtc}(\varphi)$$

primtc-box :

$$\text{primtc}([\text{st}/\alpha]\varphi) \leftrightarrow \text{sort}(\text{st}) = \text{store} \wedge \neg\ \text{st} \in \text{vars}(\alpha) \wedge \text{primtc}(\alpha) \wedge \text{primtc}(\varphi)$$

For convenience reasons we also include the requirement that the store variable st never occurs in the statement it governs. Again, this is not necessary to guarantee correctly sorted sequents, but most of the proof rules have this requirement.

Then applying a proof rule of the calculus on a primitive type correct conclusion yields primitive type correct premises:

5.3.2 Theorem (calculus preserves primitive type correctness)

The Java proof rules preserve primitive type correctness, i.e. well-sortedness.

It may be noted that this property is independent of the soundness of the calculus. This finishes the discussion of primitive type correctness. We continue with full type correctness.

5.4 Full Type Correctness

The notion of primitive type correctness is important for the algebraic setting of the calculus, and is independent of the store, i.e. the properties hold for an arbitrary store. This is achieved by converting values that are retrieved from the store to the expected primitive type. Obviously (or maybe not), Java is type safe. This means that a fully type correct Java program together with a compatible store guarantees several properties, for example

1. Every value retrieved from the store is a subtype of the expected type.
2. The computed value of every Java expression has a type that is a subtype of the static type of the expression.
3. A field access always accesses an existing field.
4. A dynamic method lookup always finds a matching method declaration.
5. The store always remains compatible.

The main difference between *primitive* and *full* type correctness is the role of the store: To achieve the mentioned properties a *compatible* store is required that remains compatible throughout the evaluation of statements and expressions. The benefit is that it is possible to design a stronger version of the calculus that automatically preserves compatibility. This in turn may help the user in proving theorems, because usually it is necessary to make some assumptions about the store, and finding the correct assumptions may be simpler for a compatible store. (Actually, it depends on the experience of the user and the theorems to prove. This is discussed in more detail in chapter 7.) The disadvantage in using a compatible store is that it is not possible to build up libraries of useful Java programs and theorems. (This is discussed in chapter 6.7.) Therefore the standard calculus only depends on primitive type correctness, while full type correctness is optional. It is not part of this work to discuss what exactly type soundness means, or which formalism is most appropriate to use. The main theorem will contain items 2 and 5 from the list above (Oheimb [vO01] proves the same two properties); items 3 and 4 are proved implicitly, but cannot be expressed explicitly in this framework. The main type soundness theorem requires a generalization (as usual) that looks for expressions like this:

If

- the expression and type declarations are fully type correct, fulltc(e) $\wedge$ fulltc(tds), and

- the expression is primitive type correct or void, primtcOrVoid(e), and
- the variable mapping is well-sorted, okvals(v), and
- the store is compatible with the type declarations, compat(st, tds), and
- the values of the free variables of the expression are compatible to their types, compat(freeparams(e), v, st, tds), and
- a free variable never occurs with two different types,

then

- the resulting store is compatible with the type declarations, and
- the free variables are still compatible with the rest,

In the rest of the chapter we will define a *compatible* store, a *fully type correct* Java program, and state the main theorem.

5.4.1 Definition of a Compatible Store

Basically, a store is *compatible* with a list of Java type declarations if objects have a class that exists in the type declarations, contain their proper fields, and the values of the fields match their static type. Usually, this is required for *all* objects in the store. However, it is easy to limit this requirement only to *reachable* objects. This allows the store to contain other objects as well.

- store-compat :
  compat(st, tds) $\leftrightarrow$ compatmode(st, tds) $\wedge$ compatstatic(st, tds) $\wedge$ compatrefs(st, tds)

  The Java store was described in chapter 3.1.3. It can be divided into three different parts: The *mode* field, the *jvmref* with static fields and initialization state for classes, and the "normal" references (the heap). This is reflected by the compatible predicate.

- mode-compat : compatmode(st, tds) $\leftrightarrow$ mode $\in$ st $\wedge$ compatmode(st[mode], st, tds)

  The mode field must exist and its value must be *mode* compatible.

- modeval-compat :
  compatmode(val, st, tds)
  $\leftrightarrow$ val = normal $\vee$ is_break_mode(val) $\vee$ val = return(noval, void)
  $\vee$ is_return_mode(val) $\wedge$ compat(val.val, val.type, st, tds)
  $\vee$ is_throw_mode(val) $\wedge$ val.refval $\neq$ null
  $\wedge$ compat(refval(val.refval), val.type, st, tds) $\wedge$ val.type $\leq_{tds}$ java.lang.Throwable

  The value of the mode field is either *normal*, or a *break*, or a *return*, or a *throw*. If it is *return* it stems from a `return e;` statement. The *return* then contains the returned value (the result of e), and the static type of e. Both must be compatible. This is checked by

  compat(val.val, val.type, st, tds)

  which is described below. A similar requirement holds for a *throw*. It contains the thrown reference ($\neq$ **null**), and the static type of the thrown expression. Again, both must be compatible,

  compat(refval(val.refval), val.type, st, tds)

  and this type must be a subtype of `Throwable`, val.type $\leq_{tds}$ java.lang.Throwable

- static-compat :
  compatstatic(st, tds)
  $\leftrightarrow \forall$ class. class $\in$ tds $\wedge$ initdone(class, st)
  $\rightarrow$ compatfields(jvmref, statfields(class, tds), st, tds)

  All initialized classes (initdone(class, st) is true) that exist in the type declarations must have all their static fields which must have compatible values. This is checked by *compatfields* defined below.

- refs-compat :
  compatrefs(st, tds)
  $\leftrightarrow \forall$ r. r $\neq$ jvmref $\wedge$ r – _type $\in$ st $\wedge$ is_typevalue(st[r – _type])
  $\rightarrow$ (is_classtype(st[r – _type].type)
  $\wedge$ class$\exists$+(st[r – _type].type.class, tds)
  $\rightarrow$ compatobj(r, st, tds))
  $\wedge$ (is_arraytype(st[r – _type].type)
  $\wedge$ ($\neg$ has_classtype(st[r – _type].type)
  $\vee$ is_tdname+(st[r – _type].type.jtelemtype.class, tds))
  $\rightarrow$ compatarray(r, st, tds))

  The (normal) references are compatible if every reference in the store with a valid type field (is_typevalue(st[r – _type])) is compatible. This means the store may contain "junk" entries, but they will never be looked at.

  A reference with a valid type field must be either an object or an array. In the first case it must have a class type (is_classtype(st[r – _type].type)), in the second an array type (is_arraytype(st[r – _type].type)).

  An object must be compatible, compatobj(r, st, tds), only if the class exists in the type declarations or is predefined. In this case class$\exists$+ is true. (There are no interface objects.)

  An array either has a primitive innermost type (i.e. no class type, $\neg$ has_classtype), or otherwise its class type must be an existing class or interface, or a predefined class (is_tdname+). Furthermore the array must be compatible (compatarray).

- obj-compat :
  compatobj(r, st, tds)
  $\leftrightarrow$ is_classtype(st[r – _type].type)
  $\wedge$ class$\exists$+(st[r – _type].type.class, tds)
  $\wedge$ compatfields(r, instfields(st[r – _type].type.class, tds), st, tds)

  An object is compatible if the class exists or is predefined, and the objects fields are compatible.

- fields-compat :
  compatfields(r, fis, st, tds) $\forall$ fs. fs $\in$ fis $\rightarrow$ r – fs $\in$ st $\wedge$ compat(st[r – fs], fs.type, st, tds)

  A list of fields fis is compatible if every field exists in the store (r – fs $\in$ st), and its value is compatible with its type.

- compatarray :
  compatarray(r, st, tds)
  $\leftrightarrow$ is_arraytype(st[r – _type].type)
  $\wedge$ r – _length $\in$ st $\wedge$ is_integervalue(st[r – _length]) $\wedge$ 0 $\leq$ st[r – _length].intval
  $\wedge$ is_arrayref(r, st[r – _length].intval, st)
  $\wedge$ ($\forall$ j. 0 $\leq$ j $\wedge$ j < st[r – _length].intval
  $\rightarrow$ compat(st[r – j '], st[r – _type].type.type, st, tds))

  An array reference must point to an array type, it must have a `length` field that contains an integer $\geq$ 0, all index fields must exist (is_arrayref), and their values must be compatible with the array's element type, compat(st[r – j], st[r – _type].type.type, st, tds).

  value-compat :
  compat(val, ty, st, tds)
  $\leftrightarrow$ okval(val.sval, sort(ty))
  $\wedge$ (is_referencevalue(val) $\wedge$ val.refval $\neq$ null
  $\rightarrow$ val.refval – _type $\in$ st $\wedge$ is_typevalue(st[val.refval – _type])
  $\wedge$ is_reftype(st[val.refval – _type].type)
  $\wedge$ (has_classtype(st[val.refval – _type].type)

$\rightarrow$ is_tdname+(st[val.refval – _type].type.jtelemtype.class, tds))
$\wedge$ (is_classtype(st[val.refval – _type].type)
$\rightarrow$ class$\exists$+(st[val.refval – _type].type.class, tds))
$\wedge$ st[val.refval – _type].type $\leq_{tds}$ ty)

This axiom states when a value is compatible with a type:

1. the value must match the type, okval(val.sval, sort(ty)) (see chapter 5.1.1)
2. if the value val is not a reference or `null` no further check is necessary.
3. Otherwise the reference r contained in val, r = val.refval, must fulfill the following:
4. r must have a valid _type field that is either a class or array type, is_reftype(st[val.refval – _type].type).
5. If this type is an array type with an innermost class type, this class or interface must exist, is_tdname+. (The run-time type of an array can contain an interface type.)
6. If it is a class type it must exist, but may not be an interface, class$\exists$+.
7. The type of the reference must be a subtype of the given type ty, le_{tds}.

These definitions are not recursive. However, the definition of compatrefs (for all objects) together with the requirement for objects that they have an existing class type ensures that all reachable objects are guaranteed to be compatible. This is achieved without a special treatment for cyclical pointer structures. Similarly, multi-dimensional arrays do not require a recursive definition. This makes it simple to prove that (valid) modifications to the store preserve compatibility. The only difficult proof is when a multi-dimensional array is added because this involves nested recursion. The definitions only reason about some existing keys in the store. They never require that a key does not exist. This means the store may contain objects of non-existing classes, or that an object could contain 'hidden' fields. This does not matter since they are never accessed.

5.4.2 Definition of a fully type correct program

The definition of full type correctness, *fulltc*, extends primitive type correctness. This means the definitions only have to deal with reference types and the class hierarchy which is a nice modularization of the definitions. We highlight some features:

- All classes that occur in static types must exist.
- The right hand side of an assignment must be a subtype (i.e. subclass for class types) of the left hand side.
- Fields and method declarations etc. must exist for the static type of the invoker, i.e. a suitable method declaration must be found.
- A free variable may not occur with different static types in a statement. If a local variable x occurs with static types `Exception` and `Error` this may initially be ok if x is `null`. However, when an `Exception` object is assigned to x the property that the run-time type is subtype of the static type is violated for x with the `Error` type. A statement like

  ```
  if (e) { Exception x = null; ...} else { Error x = null; ...}
  ```

  is ok since x is not free.
- In method declarations the static type of all returned values must be a subtype of the declared result type. This requires a predicate *fulltcReturn* comparable to *primtcReturn*.
- If a class implements an interface it must provide a method for every method declared in the interface. This is necessary for method calls where the invoker has an interface type.

We continue with the definition of fulltc for expressions. Expressions that contain nothing interesting are omitted.

- fulltc-LiteralExpr : fulltc(LiteralExpr(t, ty), tds) $\leftrightarrow \neg$ is_reftype(ty)

  The result of a literal may not be a class or array type (`null` must have type `void`).

- fulltc-RefCast : fulltc(RefCast(ty_1, e, ty), tds) $\leftrightarrow$ fulltc(e, tds) $\wedge$ type$\exists$(ty_1, tds)

  The static type of a reference type cast must exist (type$\exists$(ty_1, tds)). There is no requirement that the cast is possible (in Java, some casts are rejected by the compiler).

  The same holds for an `instanceof` expression.

- fulltc-CondExpr :
  fulltc(CondExpr(e_1, e_2, e_3, ty), tds)
  $\leftrightarrow$ fulltc(e_1, tds) $\wedge$ fulltc(e_2, tds) $\wedge$ fulltc(e_3, tds)
  $\wedge$ (ty = e_2.type $\wedge$ e_3.type $\leq_{tds}$ ty $\vee$ ty = e_3.type $\wedge$ e_2.type $\leq_{tds}$ ty)

  JLS 15.25: "If the second and third operands are of different reference types, then it must be possible to convert one of the types to the other type." One or both types may be `void`. $\leq_{tds}$ accepts `void`. `void` $\leq_{tds}$ ty $\Leftrightarrow$ ty is a reference type, and ty $\leq_{tds}$ `void` $\Leftrightarrow$ ty = `void`.

- fulltc-LocVarAccess : fulltc(LocVarAccess(jvar, ty), tds) $\leftrightarrow$ type$\exists$(ty, tds)

  The type of a local variable access must exist.

- fulltc-SFieldAccess : fulltc(SFieldAccess(fs, ty), tds) $\leftrightarrow$ fs $\in$ statfields(fs.class, tds)

  The static field must exist.

- fulltc-FieldAccess :
  fulltc(FieldAccess(e, fs, ty), tds)
  $\leftrightarrow$ fulltc(e, tds) $\wedge$ is_reftype(e.type)
  $\wedge$ (is_arraytype(e.type) $\rightarrow$ fs = _length)
  $\wedge$ (is_classtype(e.type) $\rightarrow$ fs $\in$ instfields(e.type.class, tds))

  The special case must be considered that the length field of an array is accessed. Otherwise the instance field must exist (in the class named in the static type or in one of its superclasses). instfields also collects the fields from all superclasses.

- fulltc-ArrayAccess :
  fulltc(ArrayAccess(e_1, e_2, ty), tds)
  $\leftrightarrow$ fulltc(e_1, tds) $\wedge$ fulltc(e_2, tds) $\wedge$ is_arraytype(e_1.type) $\wedge$ e_1.type.type = ty

  The array expression must have an array type.

- fulltc-LocVarAssign :
  fulltc(LocVarAssign(jvar, e, ty), tds) $\leftrightarrow$ fulltc(e, tds) $\wedge$ type$\exists$(ty, tds) $\wedge$ e.type $\leq_{tds}$ ty

  The right hand side of a local variable assignment must be either `null` or a subtype of the variable type ty. This condition holds for all assignments.

- fulltc-SFieldAssign :
  fulltc(SFieldAssign(fs, e, ty), tds)
  $\leftrightarrow$ fulltc(e, tds) $\wedge$ fs $\in$ statfields(fs.class, tds) $\wedge$ e.type $\leq_{tds}$ ty

- fulltc-FieldAssign :
  fulltc(FieldAssign(e_1, fs, e_2, ty), tds)
  $\leftrightarrow$ fulltc(e_1, tds) $\wedge$ fulltc(e_2, tds) $\wedge$ fs $\in$ instfields(fs.class, tds) $\wedge$ e_2.type $\leq_{tds}$ ty

  No assignment to a `length` field of an array is allowed; so no special treatment is required.

- fulltc-ArrayAssign :
  fulltc(ArrayAssign(e_1, e_2, e_3, ty), tds)
  $\leftrightarrow$ fulltc(e_1, tds) $\wedge$ fulltc(e_2, tds) $\wedge$ fulltc(e_3, tds)
  $\wedge$ is_arraytype(e_1.type) $\wedge$ e_1.type.type = ty $\wedge$ e_3.type $\leq_{tds}$ e_1.type.type

- fulltc-CompAssign :
  fulltc(CompAssign(e_1, $\oplus =$, e_2, ty), tds)
  $\leftrightarrow$ fulltc(e_1, tds) $\wedge$ fulltc(e_2, tds) $\wedge$ (is_FieldAccess(e_1) $\rightarrow$ e_1.fs $\neq$ _length)

  No assignment to a `length` field of an array is allowed.

- fulltc-IncDecExpr :
  fulltc(IncDecExpr(id-op, e, ty), tds)
  $\leftrightarrow$ fulltc(e, tds) $\wedge$ (is_FieldAccess(e) $\rightarrow$ e.fs $\neq$ _length)

  No assignment to a `length` field of an array is allowed.

- fulltc-NewArray :
  fulltc(NewArray(ty_1, es, i, ty), tds)
  $\leftrightarrow$ fulltc(es, tds) $\wedge$ is_arraytype(ty) $\wedge$ # es $\neq$ 0 $\wedge$ ty.jtdims = ty_1.jtdims + i + # es
  $\wedge$ 0 $\leq$ i $\wedge$ ty_1.jtelemtype = ty.jtelemtype $\wedge$ type$\exists$(ty_1, tds)

  Basically, the array type must match the number of dimensions, and there must be at least one dimension expression. `new int[][]` is illegal.

- fulltc-ArrayInit :
  fulltc(ArrayInit(es, ty), tds)
  $\leftrightarrow$ fulltc(es, tds) $\wedge$ is_arraytype(ty) $\wedge$ type$\exists$(ty, tds) $\wedge$ es.types $\leq_{tds}$ ty.type

  The arguments to the array initializer must be subtypes of the static element type of the array.

- fulltc-NewExpr :
  fulltc(NewExpr(class, es, tys, ty), tds)
  $\leftrightarrow$ fulltc(es, tds) $\wedge$ ty = classtype(class) $\wedge$ es.types $\leq_{tds}$ tys
  $\wedge$ hasImplementedConstr(class, tys, tds)

  The class must have an implemented constructor with the given argument types tys, and the types of the actual parameters must be subtypes of tys. Again, `null` (with `void` type) is allowed.

- fulltc-ConstrCall :
  fulltc(ConstrCall(e, class, es, tys, ty), tds)
  $\leftrightarrow$ fulltc(e, tds) $\wedge$ fulltc(es, tds) $\wedge$ ty = e.type
  $\wedge$ es.types $\leq_{tds}$ tys $\wedge$ e.type $\neq$ void $\wedge$ e.type $\leq_{tds}$ class
  $\wedge$ hasImplementedConstr(class, tys, tds)

  An explicit constructor invocation has the same requirements as a `new` invocation plus the requirement that the static type of the invoker is a subtype ($\neq$ `void`) of the class. This is needed for `super` calls where the `this` object is the invoker.

- fulltc-MethodCall :
  fulltc(MethodCall(e, m, invmo, es, tys, ty), tds)
  $\leftrightarrow$ fulltc(e, tds) $\wedge$ fulltc(es, tds) $\wedge$ es.types $\leq_{tds}$ tys
  $\wedge$ $\neg$ is_interface_mode(invmo)
  $\wedge$ (is_super(invmo) $\rightarrow$ e.type = classtype(invmo.class))
  $\wedge$ (is_nonvirtual(invmo) $\rightarrow$ e.type = classtype(invmo.class))
  $\wedge$ (is_static(invmo) $\rightarrow$ e = null $\vee$ e.type = classtype(invmo.class))
  $\wedge$ (is_virtual(invmo) $\rightarrow$ is_classtype(e.type))
  $\wedge$ ($\neg$ is_static(invmo)
  $\rightarrow$ hasImplementedMethod(e.type.class, str, invmo, tys, ty, tds))
  $\wedge$ (is_static(invmo) $\rightarrow$ hasImplementedMethod(invmo.class, str, invmo, tys, ty, tds))

  Not surprisingly, a method call has the most complicated requirements, depending on the invocation mode. The invocation mode determines the static type of the invoking expression. For `super` and `nonvirtual` it must the the class of the invoking expression. For `static` the

invoker may be the `null` literal (this is not legal in Java because then the class of the method declaration cannot be determined, however we assume that the compiler adds **null** as invoker for method calls without invoker). For `virtual` must be only a class type. The invocation mode `virtual` is also used if the invoker has an interface type; the special mode `interface` is not allowed in fully type correct programs.

Additionally, a matching method declaration must be found. This is checked by the predicate hasImplementedMethod that again depends on the invocation mode. For a `virtual` or `super` mode super classes and implemented interfaces are searched, otherwise only the given class is searched.

Because of the interfaces a class that implements an interface must implement all methods. Otherwise it is not guaranteed that an implemented method declaration is found at run time.

We continue with the checks for statements

- fulltc-block :
  fulltc(`{`$\alpha_1 \dots \alpha_n$`}`), tds)
  $\leftrightarrow$ fulltc($\alpha_1 \dots \alpha_n$, tds) $\wedge$ disjoint(freeparams($\alpha_1 \dots \alpha_n$)) $\wedge$ $\neg$ may_catch($\alpha_1 \dots \alpha_n$)

  Here the disjointness check (a variable is not used with different types) is necessary. Furthermore, the block statements may not contain any of the newly introduced statements that can catch jumps.

- fulltc-locvardecl : fulltc(`ty x = e;`, tds) $\leftrightarrow$ fulltc(e, tds) $\wedge$ type$\exists$(ty, tds) $\wedge$ e.type $\leq_{tds}$ ty

  The right-hand side of the local variable declaration must be `null` or a subtype of the static type.

- fulltc-for :
  fulltc(`for(; e; es;` α`)`, tds)
  $\leftrightarrow$ fulltc(e, tds) $\wedge$ fulltc(es, tds) $\wedge$ fulltc(α, tds) $\wedge$ $\neg$ may_catch(α)
  $\wedge$ $\neg$ is_locvardecl(α) $\wedge$ disjoint(freeparams(α) $\cup$ freeparams(es))

  The body of the `for` loop may not be a local variable declaration.

- fulltc-throw :
  fulltc(`throw e;`, tds) $\leftrightarrow$ fulltc(e, tds) $\wedge$ e.type $\neq$ void $\wedge$ e.type $\leq_{tds}$ java.lang.Throwable

  Only subclasses of `Throwable` are thrown.

- fulltc-catch :
  fulltc(`catch`(class, x, α), tds)
  $\leftrightarrow$ fulltc(α, tds) $\wedge$ class $\leq_{tds}$ java.lang.Throwable
  $\wedge$ disjoint(freeparams(α) $\cup$ { classtype(class) $\times$ x })

  The body of a `catch` clause acts like a block. It may occur only in *catches* part of a `try` statement.

The additional statements are not fulltc. However, we will allow them on top-level. Therefore we introduce a predicate fulltcOrCatch that also allows those statements. Any classes, e.g. in `static`(class) must exist. We omit the definition of fulltcReturn and continue with the type correctness of type and member declarations:

- fulltc-tds : fulltc(tds) $\leftrightarrow$ primtc(tds) $\wedge$ wfCh(tds) $\wedge$ ($\forall$ td. td $\in$ tds $\rightarrow$ fulltc(td, tds))

  A list of type declarations tds is fully type correct if it is primitive type correct (primtc(tds)), the class hierarchy is well formed (wfCh(tds)), and every type declaration is fully type correct. The definition of a well formed class hierarchy must check the usual things (acyclic, no duplicates, etc.) and is omitted.

- fulltc-td :
  fulltc(td, tds)
  $\leftrightarrow$ (is_classdecl(td)
  $\rightarrow$ ($\forall$ class. class $\in$ td.implements
  $\rightarrow$ class_implements(superints(class, tds), td, tds))
  $\wedge$ fulltcClass(td.class, td.memdecls, tds))
  $\wedge$ (is_interfacedecl(td) $\rightarrow$ fulltcInterface(td.class, td.memdecls, tds))

  We have to distinguish between class (*fulltcClass*) and interface (*fulltcInterface*) declarations. For class declarations, we additionally require that they really implement the interfaces that are listed in the `implements` clause (*td.implements*). This is done by *class_implements*. *superints* computes all super interfaces of a given interface.

- fulltcC-statinit :
  fulltcClass(class, statinit(α), tds) $\leftrightarrow$ fulltc(α, tds) $\wedge$ freeparams(α) = $\emptyset$

  For a class the static initializer is fully type correct if the statement is fully type correct, and if the statement is not one of the special statements ($\neg$ *may_catch*), and if it contains no free variables (freeparams(α) = $\emptyset$). freeparams computes the free variables together with their static type, but omits free variables in literals.

- fulltcC-fielddecl :
  fulltcClass(class, fielddecl(mos, ty, fs), tds) $\leftrightarrow$ fs.class = class $\wedge$ type$\exists$(ty, tds)

  A field declaration is fully type correct if the class mentioned in the field specification is the current class (fs.class = class) and its type exists.

- fulltcC-methoddecl :
  fulltcClass(class, methoddecl(mos, ty, m, params, α), tds)
  $\leftrightarrow$ $\neg$ native $\in$ mos $\wedge$ $\neg$ abstract $\in$ mos $\wedge$ fulltc(α, tds)
  $\wedge$ freeparams(params $\cup$ { classtype(class) $\times$ THIS }, α) = $\emptyset$
  $\wedge$ (static $\in$ mos $\rightarrow$ $\neg$ THIS $\in$ freeparams(α))
  $\wedge$ (ty = void $\vee$ fulltcReturn(α, ty, tds)) $\wedge$ type$\exists$(ty, tds)

  A method declared in a class may neither be *native* nor *abstract* (allowing abstract classes is a rather simple extension and requires the usual check that a non-abstract class implements all abstract members of its super classes). Its body must be fully type correct, it may contain no free variables, and the returned type must match (*fulltcReturn*). Instance methods may contain `this`, but only with a static type that is the current class. A static method may not contain `this`.

- fulltcC-constrdecl :
  fulltcClass(class, constrdecl(mos, class_0, params, α), tds)
  $\leftrightarrow$ $\neg$ native $\in$ mos $\wedge$ $\neg$ abstract $\in$ mos $\wedge$ class = class_0 $\wedge$ fulltc(α, tds)
  $\wedge$ freeparams(params $\cup$ { classtype(class) $\times$ THIS }, α) = $\emptyset$

  Essentially, a constructor declaration has the same requirements as an instance method.

- fulltcI-statinit :
  fulltcInterface(class, statinit(α), tds) $\leftrightarrow$ fulltc(α, tds) $\wedge$ freeparams(α) = $\emptyset$

  A static initializer in an interface has the same requirements as in a class (we use static initializers in interfaces to handle initialization of static fields).

- fulltcI-fielddecl :
  fulltcInterface(class, jfielddecl(mos, ty, fs), tds) $\leftrightarrow$ fs.class = class $\wedge$ type$\exists$(ty, tds)

  Identical to class declarations. It does not matter if the fields are declared as `static` or not.

- fulltcI-methoddecl :
  fulltcInterface(class, jmethoddecl(mos, ty, m, params, α), tds)
  $\leftrightarrow \neg$ native $\in$ mos $\wedge$ abstract $\in$ mos $\wedge \neg$ jstatic $\in$ mos $\wedge$ type$\exists$(ty, tds) $\wedge \alpha =$ `{ }`

  A method declared in an interface must be abstract, not static, not native, and must have an empty body.

- fulltcI-constrdecl :
  $\neg$ fulltcInterface(class, constrdecl(mos, class$_0$, params, α), tds)

  An interface may not contain a constructor declaration.

The definition of *class_implements* is omitted because it contains nothing interesting or unusual.

5.4.3 Java is Type Safe

5.4.1 Theorem (type soundness)

When an expression is evaluated the resulting store is compatible with the type declarations, and the computed value is also compatible if the mode is normal:

$$
\begin{aligned}
&(v \times st)[\![e]\!]_{tds}(v_0 \times st_0 \times val_0) \\
&\wedge \text{ fulltc(e, tds)} \wedge \text{fulltc(tds)} \wedge \text{okvals(v)} \wedge \text{primtc(e)} \\
&\wedge \text{ compat(freeparams(e), v, st, tds)} \wedge \text{disjoint(freeparams(e))} \\
&\wedge \text{ compat(st, tds)} \\
\rightarrow &\text{ compat}(st_0, \text{ tds}) \wedge (st_0[\text{mode}] = \text{normal} \rightarrow \text{compat}(val_0, \text{ e.type}, st_0, \text{ tds}))
\end{aligned}
$$

When a statement is evaluated the resulting store is compatible with the type declarations:

$$
\begin{aligned}
&(v \times st)[\![\alpha]\!]_{tds}(v_0 \times st_0) \\
&\wedge \text{ fulltc}(\alpha, \text{ tds}) \wedge \text{fulltc(tds)} \wedge \text{okvals(v)} \wedge \text{primtc}(\alpha) \\
&\wedge \text{ compat(freeparams}(\alpha), \text{ v, st, tds)} \wedge \text{disjoint(freeparams}(\alpha)) \\
&\wedge \text{ compat(st, tds)} \\
\rightarrow &\text{ compat}(st_0, \text{ tds})
\end{aligned}
$$

Again, a generalization is needed that includes expressions, statements, expression lists, and statement lists. The most complicated proof involves, obviously, the instance method invocation. It requires the proof that the run-time class of the invoker is a subclass of its static class (or interface). Then in turn it can be proved that always a matching method declaration is found. The discussion of a compatible store is continued in chapter 6.7.

Chapter 6

Libraries and Modifications

The calculus presented in the previous chapters allows to prove properties of sequential Java or Java Card programs. However, experience shows that for not completely trivial examples more is needed. Efficient program verification must deal with the fact that programs usually contain errors, and that correct versions are obtained only after a number of iterations. Furthermore, reuse of proofs and theorems – the creation of a useful library – is important.

This chapter deals with the reuse of proofs after modifications and from a library. We begin with the library scenario, and deal with modifications in the last part of the chapter.

6.1 Java Libraries in KIV

The idea for libraries is the following: In KIV, Java type declarations are part of an algebraic specification. Instead of having all type declarations in one specification they should be structured in different specifications. For example, a specification *ISO7816* contains the interface `ISO7816` from the Java Card API. It is enriched with specification *javacarddefs* containing the base classes of the Java Card runtime environment. This specification is enriched with a specification *jcrypto* containing the security and cryptographic related classes of the runtime environment. This specification then can be enriched with different applets. This situation is shown on the left in Fig. 6.1. For full Java there may be different specifications containing useful, but independent, auxiliary classes like the `List` interfaces, `Vector`, or `BigInteger`.

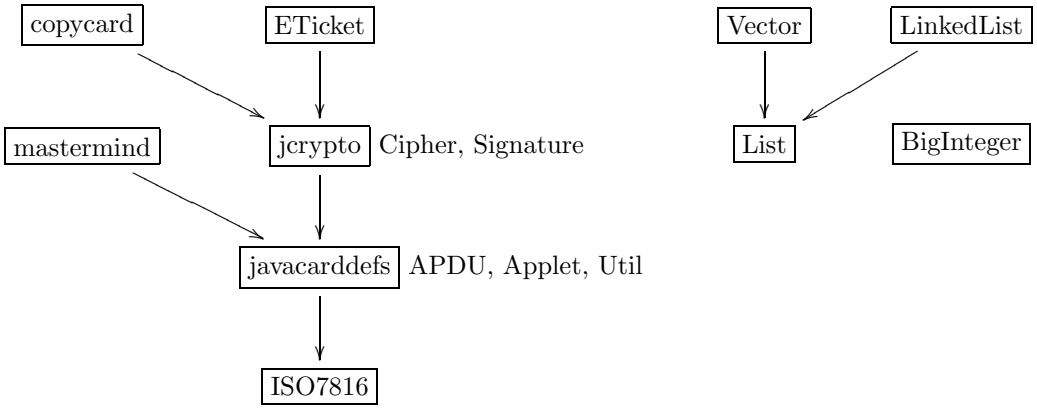


Figure 6.1: Library of Java Card classes

Here, new classes are added, and the question is:

> *Is a theorem that was proved in a subspecification still valid in an enrichment where new classes are added?*

The answer will be yes. However, it is not trivial to ensure this property. The solution is different from other approaches where the emphasis is, for example, on ensuring class invariants or proving frame properties like Müller [Mül01] and Poetzsch-Heffter [MPH99]. Here, programs are annotated with *modifies* clauses, and proof obligations are generated that ensure certain properties; the whole focus is on interface specifications, information hiding etc., not so much on the verification of applications.

We consider the following example library: A specification *Super* containing a class `Super` (extending `Exception`) that is enriched with a specification *Sub* containing a class `Sub extends Super` (see Figure 6.2). *Super* is based on the predefined specifications for the Java store and other data types.

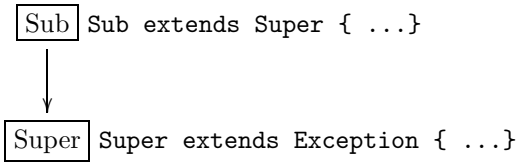


Figure 6.2: Example for the Class Hierarchy

Now we inspect the following sequent:

$$\begin{array}{l} \quad \text{st[_mode]} = \text{normal},\ \text{r} \neq \text{null},\ \text{st[r} - \text{_type]} = \text{typeval(classtype(Sub))} \\ \vdash \langle \text{st/}\texttt{boolvar = r instanceof Super;}\rangle \neg\ \text{boolvar;} \end{array}$$

r is a reference that has a type field in the store stating its class is `Sub`, i.e. it is an object of class `Sub`. The question is, is `r instanceof Super` true or false? Well, this depends on the type declarations. The sequent holds in specification *Super*, but does not hold in specification *Sub*, as will be explained now. The sequent holds in specification *Super*: `r instanceof Super` is false, because the semantics of `instanceof` uses asgcomp to check the subtype relation:

$$\frac{(v \times st)[\![e]\!](v_0 \times st_0 \times val_0) \quad st_0[mode] = normal \quad val_0 \neq null \wedge asgcomp(val_0, ty, st_0, tds)}{(v \times st)[\![e\ \textbf{instanceof}\ ty]\!](v_0 \times st_0 \times \text{true})}$$

asgcomp was described in chapter 3.2.5. Its definition is repeated here:

$$\begin{array}{ll} & \text{asgcomp(val, ty, st, tds)} \\ \leftrightarrow & \quad \neg\ \text{is_referencevalue(val)} \\ & \vee\ \text{val.refval} = \text{null} \\ & \vee\ \text{st[val.refval} - \text{_type].type} \leq_{tds} \text{ty} \end{array}$$

If the value is a reference $\neq$ **null** its runtime type (st[val.refval – _type].type) is looked up in the store, and this must be a subtype of the type *ty* with given type declarations *tds*. The definition of subtype is:

- subtype-other :

  $\neg$ is_classtype(ty) $\wedge$ $\neg$ is_arraytype(ty) $\rightarrow$ ty $\leq_{tds}$ $\text{ty}_0 \leftrightarrow \text{ty} = \text{ty}_0$)

  If the first type is not a reference type both types must be equal (`short` is not a subtype of `int` even though short values can be converted by primitive widening conversion to integer values).

- subtype-array :

  $$\begin{array}{l} \quad \text{arraytype(ty)} \leq_{tds} \text{ty}_0 \\ \leftrightarrow\ \text{ty}_0 = \text{classtype(java.lang.Object)} \vee \text{is_arraytype}(\text{ty}_0) \wedge \text{ty} \leq_{tds} \text{ty}_0\text{.type} \end{array}$$

  If the first type is an array type the second type is either Object or an array type and the subtype relation holds for the element types. (.type selects the element type of an array type.)

- subtype-class :

  classtype(class) $\leq_{tds}$ ty
  $\leftrightarrow$ is_classtype(ty) $\wedge$ ty.class $\in$ allsupers(class, tds)

  This is the important case for the example: a class type containing *class* is subtype of another type *ty* if *ty* is also a class type and the class of *ty* is one of the super classes of *class* (or *class* itself), or an interface implemented by *class*. The function *allsupers* recursively collects all super classes and implemented interfaces. (Recursive means that super interfaces of implemented interfaces and interfaces implemented by a super class and their super interfaces are also collected). This definition captures the definition of JLS 5.2.

For the example this means that

asgcomp(r, classtype(Super), st, tds)
$\Leftrightarrow$ st[r – _type].type $\leq_{tds}$ classtype(Super) (since r is a reference $\neq$ null)
$\Leftrightarrow$ classtype(Sub) $\leq_{tds}$ classtype(Super)
(explicit precondition st[r – _type] = typeval(classtype(Sub)))
$\Leftrightarrow$ Super $\in$ allsupers(Sub, tds)
$\Leftrightarrow$ Super $\in$ { Object }
$\Leftrightarrow$ false

The point is that the class `Sub` does not exist in the type declarations for specification *Super*, and for non-existing classes allsupers returns { `Object` }. This means the sequent holds in specification *Super*. However, in specification *Sub* the sequent does not hold, because the class `Sub` has been added, and allsupers(Sub, tds) = { `Sub`, `Super`, `Object` }.

The example shows that the correctness of a theorem does not necessarily propagate when new classes are added. This is bad, because exactly this happens when theorems from a library are used. But, we can ensure the following property: If a theorem has been proved in a specification (i.e. it is valid) then it is also valid in a specification where classes have been added, essentially because the same proof is possible in the new specification. If both specifications are consistent, and the calculus is correct, then this implies the validity of the theorem. This necessarily means that the theorem presented above cannot be proved in specification *Super* (even though it is valid) – the calculus must be incomplete. However, this is not really a drawback since the example is certainly pathological (we introduce and reason about non-existing classes). To ensure the property the proof rules must be designed in such a manner that it is correct to apply the proof rules in an identical manner in the enriching specification (i.e. every instance of a proof rule in specification *Super* is valid in specification *Sub*). And, we require a precise definition what "adding new classes" means. We continue with this definition.

6.2 Extending Type Declarations

A library should be extensible, i.e. it should be possible to add new classes that extend the existing classes or are completely unrelated. It does not mean that new methods or fields can be added to existing classes.

We define a predicate *extend*: *tds′ extend tds* should be true if the type declarations *tds′* are a valid extension of the existing type declarations *tds*:

tds' extend tds
$\leftrightarrow \forall$ class. (class $\in$ tdnames(tds) $\rightarrow \neg$ class $\in$ tdnames(tds'))
$\wedge$ (class $\in$ tdclasses(tds) $\rightarrow \neg$ extends(getclass(class, tds)) $\in$ tdnames(tds'))

tdsnames simply collects all names of the classes and interfaces in the list of type declarations, and *tdclasses* collects the names of the classes only. *getclass* returns the type declaration for a

class name, and *extends* returns c2 in `class c1 extends c2 ...` The definition states that the new type declarations tds' do not contain class or interface names that are already declared in the old type declarations *tds*, and that they do not contain class names that occur in the `extends` part of a class declaration. This is a fairly weak requirement. For some of the properties below we need a stronger condition:

tds' wfChExtend tds $\leftrightarrow$ wfCh(tds) $\wedge$ wfCh(tds' + tds)

$tds' + tds$ is just the concatenation of the two type declarations. This means that the original *tds* must have a valid class hierarchy and also $tds' + tds$. It is simple to see that

tds' wfChExtend tds $\rightarrow$ tds' extend tds

holds. Additionally, we require that $tds' + tds$ is primitive type correct, $primtc(tds' + tds)$, since this a precondition for most proof rules.

To summarize, creating libraries – adding new classes in an enriched specification – is only possible if the original and new type declarations together have a valid class hierarchy and if they are primitive type correct. Fully type correct type declarations have both properties, fulltc(tds) $\rightarrow$ primtc(tds) $\wedge$ wfCh(tds). If this is the case then a theorem proved in a library is also valid in the enriched specification, as will be seen in the next two sections.

6.3 Context dependency of Proof Rules

The Java type declarations are the context for proofs. The proof style is always backwards: A proof rule is applied on a goal (the conclusion) and yields new subgoals (the premises). Some proof rules, for example the *if* rule for the `if` statement, do not depend on the context at all: the premises depend solely on the conclusion. However, for an instance method invocation the premises (even the number of premises) depend on the conclusion and the context because a dynamic method lookup is performed. This means the class hierarchy and of course the method body is relevant. As a first step we classify the proof rules according to their dependency on the type declarations:

1. Proof rules that do not depend on the type declarations at all.

   conditional operator, conditional binary operator, exception binary operator, local variable access, static field access, instance field access, array access, local variable assignment, static field assignment, instance field assignment, array assignment, incdec, compound assignment, array creation, array initializer, block, label, if, while, do, for, switch, try, break, return, target, returnexpr, targetexpr, throw, finally, endfinally, literalize, flattenOne, flattenConflict, split left, execute program, contract program, while invariant, bounded while, for invariant

2. Proof rules that depend on the class hierarchy.

   cast, instanceof, array assignment, catches, endstatic

   An array assignment may raise an `ArrayStoreException`. The *catches* rule handles catch clauses and thrown exceptions. It depends on the class hierarchy for exceptions and errors which clause is applicable for a (user-defined) exception/error. The *endstatic* rule must distinguish between (user-defined) exceptions and errors that occur during static initialization.

3. Proof rules that depend on specific parts of the type declarations.

   static, class instance creation (*newclass*), constructor invocation (*constr*), static method invocation (*smethod*), instance method invocation (*imethod*), nonvirtual or super method invocation (*nonmethod*)

   static selects the static initializer for a class, *newclass* access the instance fields of an object, and the other four rules depend on the constructor or method body. *static*, *nonmethod*, and *imethod* also depend on the class hierarchy.

We handle these dependencies in the following manner:

1. No treatment necessary.
2. Instead of adding a formula that explicitly encodes the class hierarchy we generate axioms encoding the class hierarchy. The axioms are described in section 6.4.
3. A more detailed analysis is necessary. See section 6.5.

6.4 Axioms for the Class Hierarchy

The example (Figure 6.2) shows that a valid theorem is not necessarily valid when new classes are added, and the problem are non-existing classes. Therefore, all proof rules that depend on the class hierarchy and look up a type in the store have an additional premise stating that these types exist. For other types like the type `Super` in `r instanceof Super` this can be checked statically, and is an additional precondition for the application of the proof rule. We repeat the proof rule for instanceof:

$$\frac{\begin{array}{ll}1. & \Gamma \vdash e = \mathit{null} \lor \mathit{type}\exists\ (\mathit{st}[e - _\mathit{type}].\mathit{type}), \Delta \\ 2. & \Gamma, \mathit{st}[\mathit{mode}] \neq \mathit{normal} \vdash \varphi, \Delta \\ 3. & \Gamma, \mathit{st}[\mathit{mode}] = \mathit{normal}, \neg\ (e \neq \mathit{null} \land \mathit{asgcomp_fma}) \vdash \langle \mathit{st}/\texttt{x = false;} \rangle\ \varphi, \Delta \\ 4. & \Gamma, \mathit{st}[\mathit{mode}] = \mathit{normal}, e \neq \mathit{null} \land \mathit{asgcomp_fma} \vdash \langle \mathit{st}/\texttt{x = true;} \rangle\ \varphi, \Delta\end{array}}{\Gamma \vdash \langle \mathit{st}/x = e\ \texttt{instanceof}\ \mathit{ty}; \rangle\ \varphi, \Delta}$$

Premise 1 contains the *type*$\exists$ test. The proof rules with such an extra premise are:

> reference cast, instanceof, array assignment, instance method invocation, catches, end-static

Another issue is how to express the existence of a type or a subtype relation in the calculus. The sequent should not contain the type declarations: They are too large (making rewriting inefficient), and in case of any modification of the type declarations the proof would become invalid. One solution is to generate a formula that describes the class hierarchy in term of the existing class names. In the example this would be something like:

$$\begin{array}{l}\quad \mathit{class}\ \text{is_subclass_of}\ \texttt{Super} \leftrightarrow \mathit{class} = \texttt{Super} \\ \land\ \texttt{Super}\ \text{is_subclass_of}\ \mathit{class} \leftrightarrow \mathit{class} \in \{\ \texttt{Super}, \texttt{Exception}, \texttt{Throwable}, \texttt{Object}\}\end{array}$$

For larger examples this becomes a rather large formula, and if the class hierarchy is modified a proof becomes invalid even if the modification is irrelevant for the proof.

Therefore we choose another approach: To generate axioms. (The axiom generation process itself is not specified formally.) The formula shown above has the status of an axiom, because it depends solely on the context (the type declarations). So the idea is to generate axioms for the type declarations of a specification that describe the class hierarchy, independent of single theorems or proofs. These axioms can be used by the normal simplification and rewrite rules (because they are predicate logic formulas), and the standard correctness management handles modifications of these axioms because of modified type declarations. If the axioms and predicates are designed adequately they are used efficiently, do not unnecessarily enlarge the sequents, and are rather robust against modifications. There is an additional requirement: The axioms must not lead to inconsistencies if classes are added in enriching specifications as in the library scenario. If the following axioms are generated in the example:

> In specification *Super*: class$\exists$(class) $\leftrightarrow$ class = `Super`
>
> In specification *Sub*: class$\exists$(class) $\leftrightarrow$ class = `Sub` $\lor$ class = `Super`

the specification *Sub* is inconsistent. The solution is to parameterize the predicate with the name of the specification (this name must be unique):

In specification *Super*: class∃(*Super*, class) ↔ class = `Super`

In specification *Sub*: class∃(*Sub*, class) ↔ class = `Sub` ∨ class∃(*Super*, class)

The same technique can be used to describe the class hierarchy. The generation of axioms is designed to work together with some predefined generic funcions and predicates, and their axiomatization. The following generic functions and predicates are used:

- type∃: string × javatype → bool (string is the specification name)

  type∃(str, ty) ↔ ¬ has_classtype(ty) ∨ tdname+∃(str, ty.jtclass)

  type∃ is used to guarantee the existence of a type. has_classtype is true if a type is a class type or an array type that has a class type as its innermost element type; .jtclass selects this class.

  tdname+∃(str, class) ↔ tdname∃(str, class) ∨ is_predefined_class(class)

  tdname+∃ is true if the class name is the name of an existing class or interface, or is predefined. is_predefined_class is true for the predefined classes (Object and the Errors and Exceptions used in the semantics).

  tdname∃(str, class) ↔ class∃(str, class) ∨ interface∃(str, class)

  tdname∃ is true if the class name is the name of an existing class or interface. It makes use of class∃ (class exists) and interface∃ (interface exists). For these two predicates axioms will be generated.

- ≤ : javatype × javatype

  ≤ axiomatizes the subtype relation depending on the class hierarchy:

  classtype($class_1$) ≤ classtype($class_2$) ↔ $class_1$ ≤ $class_2$

  arraytype(ty) ≤ arraytype(ty_0) ↔ ty ≤ ty_0

  arraytype(ty) ≤ classtype($class_1$) ↔ $class_1$ = `java.lang.Object`

  ¬ classtype(class) ≤ arraytype(ty)

  ¬ is_classtype(ty) ∧ ¬ is_arraytype(ty) → (ty ≤ ty_0 ↔ ty = ty_0)

  ¬ is_classtype(ty_0) ∧ ¬ is_arraytype(ty_0) → (ty ≤ ty_0 ↔ ty = ty_0)

  These axioms correspond to the definition of *subtype* $\leq_{tds}$ used in the semantics (see chapter 3.2.5). They are based on ≤ for two class names:

- ≤ : classname × classname

  This describes the actual class hierarchy. Here, axioms are generated.

For the example the following axioms are generated:

- In specification *Super*:

  class∃(*Super*, class) ↔ class = `Super`

  interface∃(*Super*, class) ↔ false

  `Super` ≤ class ↔ class = `Super` ∨ `Exception` ≤ class

  (since `class Super extends Exception {...}`)

- In specification *Sub*:

  class∃(*Sub*, class) ↔ class = `Sub` ∨ class∃(*Super*, class)

  interface∃(*Sub*, class) ↔ interface∃(*Super*, class)

  `Sub` ≤ class ↔ class = `Sub` ∨ `Super` ≤ class

  (since `class Sub extends Super {...}`)

The general form of an axiom for $\leq$ is: For every class `c` with declaration

```
class c extends c1 implements c2, ..., cn { ...}
```

an axiom

$$\texttt{c} \leq \text{class} \leftrightarrow \text{class} = \texttt{c} \vee \text{c1} \leq \text{class} \vee \text{c2} \leq \text{class} \vee \ldots \vee \text{cn} \leq \text{class}$$

Similar axioms are generated for interfaces. Since $\leq$ occurs only in proof rules where it is known that all types exist, these axioms are sufficient to decide all possible cases. (Either $ty_1 \leq ty_2$ or $\neg\ ty_1 \leq ty_2$ can be proved for existing classes.)

If the instanceof rule is applied in specification *Super* the first premise will be

$$\text{e} = \text{null} \vee \text{type}\exists(\textit{Super}, \text{st}[\text{e} - _\text{type}].\text{type})$$

If the same rule is applied in specification *Sub* for the same goal the first premise is

$$\text{e} = \text{null} \vee \text{type}\exists(\textit{Sub}, \text{st}[\text{e} - _\text{type}].\text{type})$$

i.e. different. But

$$\text{type}\exists(\textit{Super}, \text{st}[\text{r} - _\text{type}].\text{type}) \Rightarrow \text{type}\exists(\textit{Sub}, \text{st}[\text{r} - _\text{type}].\text{type})$$

and if the premise could be proved in *Super*, then it can also be proved in *Sub*. The other premises of the instanceof rule are identical if applied in *Super* or *Sub*. This will ultimately lead to the desired property that every theorem proved in a library is still valid when new classes are added.

This finishes the discussion of proof rules that depend on the class hierarchy. We continue with the discussion of proof rules that depend on other parts of the type declarations.

6.5 Detailed Analysis of Dependencies

One of the advantages of a formal specification of the proof rules is that it is very easy to determine what parts of the type declarations are used. It is only necessary to take a look at those functions that use specific parts of the type declarations. The remaining proof rules to analyse are

static, newclass, constr, smethod, nonmethod, imethod

We will show the following property:

> If tds' is a (valid) extension of tds (tds' extend tds), then applying one of the previous listed proof rules yields the same result for the context tds and $tds' + tds$.

We analyse each proof rule in turn.

static

The static rule uses the type declarations in the following functions:

- to compute the super class *superClass(class, tds)*,
- to obtain the static fields of the class to initialize *statfields(class, tds)*,
- and to select its static initializer *statinit(class, tds)*.

The proof rule has the precondition that *class* is an existing class or interface. Then it is simple to prove that all three functions compute the same result if the type declarations are extended. Furthermore, the precondition also holds:

6.5.1 Theorem (extending static)

```
   is_tdname+(class, tds) ∧ tds' extend tds
→    superClass(class, tds' + tds) = superClass(class, tds)
   ∧ statfields(class, tds' + tds) = statfields(class, tds)
   ∧ statinit(class, tds' + tds) = statinit(class, tds)
   ∧ is_tdname+(class, tds' + tds)
```

newclass

The newclass rule uses the type declarations in the following functions:

- to select all instance fields of a class (including fields of super classes), instfields(class, tds)

The proof rule requires that the class is an existing (or predefined) class, class∃+(class, tds). Then we have

6.5.2 Theorem (extending newclass)

class∃+(class, tds) ∧ tds' extend tds
→ instfields(class, tds' + tds) = instfields(class, tds) ∧ class∃+(class, tds' + tds)

The definition of extend ensures that the super classes of *class* are not changed by the new classes.

constr

The constr rule uses the type declarations in the following functions:

- to obtain the correct constructor declaration for the constructor call, getConstr(class, tys, tds)

The proof rule requires that getConstr returns an implemented constructor. This implies that the class exists. Then we have

6.5.3 Theorem (extending constr)

implemented(getConstr(class, tys, tds)) ∧ tds' extend tds
→ getConstr(class, tys, tds' + tds) = getConstr(class, tys, tds)
∧ implemented(getConstr(class, tys, tds' + tds))

smethod

The smethod rule uses the type declarations in the following functions:

- to obtain the correct method declaration, getMethod(invmo.class, str, invmo, tys, ty_0, tds)

The proof rule requires that getMethod returns an implemented declaration. This implies that the class exists, and we have

6.5.4 Theorem (extending smethod)

implemented(getMethod(invmo.class, str, invmo, tys, ty_0, tds)) ∧ tds' extend tds
→ getMethod(invmo.class, str, invmo, tys, ty_0, tds' + tds)
= getMethod(invmo.class, str, invmo, tys, ty_0, tds)
∧ implemented(getMethod(invmo.class, str, invmo, tys, ty_0, tds' + tds))

nonmethod

The nonmethod rule uses the type declarations in the following functions:

- to obtain the correct method declaration, getMethod(invmo.class, str, invmo, tys, ty_0, tds)

The proof rule requires that getMethod returns an implemented declaration. This implies that the class exists. In contrast to the the static method invocation a dynamic method lookup is used. The definition of extend guarantees that the super classes remain unchanged.

6.5.5 Theorem (extending nonmethod)

implemented(getMethod(invmo.class, str, invmo, tys, ty_0, tds)) ∧ tds' extend tds
→ getMethod(invmo.class, str, invmo, tys, ty_0, tds' + tds)
= getMethod(invmo.class, str, invmo, tys, ty_0, tds)
∧ implemented(getMethod(invmo.class, str, invmo, tys, ty_0, tds' + tds))

imethod

The imethod rule is parameterized with a list of possible runtime classes of the invoking expression. The user can select this list. The rule uses the type declarations in the following functions:

- to obtain the correct method declarations for every selected class, getMethod(class, str, invmo, tys, ty_0, tds)

The proof rule requires that the selected classes are subclasses of the static class of the invoking expression, and that getMethod returns an implemented declaration for every class. This implies that the class exists. Again a dynamic method lookup is used, and the definition of extend guarantees that the super classes remain unchanged.

6.5.6 Theorem (extending imethod)

areImplementedMethods(rarg.classes, str, invmo, tys, ty_0, tds) $\wedge$ tds' extend tds
$\rightarrow$ areImplementedMethods(rarg.classes, str, invmo, tys, ty_0, tds' + tds)
$\wedge \forall$ class. class $\in$ classes
$\rightarrow$ getMethod(class, str, invmo, tys, ty_0, tds' + tds)
= getMethod(class, str, invmo, tys, ty_0, tds)

A stronger property is needed for areSubclassesOf because this depends on the class hierarchy:

areSubclassesOf(classes, tds) $\wedge$ tds' wfChExtend tds
$\rightarrow$ areSubclassesOf(classes, tds' + tds)

6.6 Theorems in Libraries

Finally it is possible to state the main theorem:

6.6.1 Theorem (Theorems in libraries)

If tds' is a valid extension of tds, tds' wfChExtends tds $\wedge$ primtc(tds' + tds),
then every theorem proved with context tds is valid with context tds' + tds:

tds' wfChExtends tds $\wedge$ primtc(tds' + tds) $\wedge$ tds, $\mathcal{A} \vdash (\Gamma \vdash \Delta) \Rightarrow$ tds' + tds, $\mathcal{A} \vdash (\Gamma \vdash \Delta)$

Proof

The proof for the theorem with context tds is also a correct proof in context $tds' + tds$. This is true because for every proof rule one of the following is true (as has been shown above):

- the proof rule does not depend on the context at all, or
- it yields the same premises when applied on the same conclusion (and fulfill the preconditions for the rule), as shown above, or
- the proof rule depends on the class hierarchy, and requires a proof that the involved types exist. This property still hold when new classes are added. The other premises are unchanged.

6.7 Libraries and Compatibility

In chapter 5.4.1 the notion of a *compatible* store was introduced. It means that the store contains valid objects that match (in some sense) the type declarations of the context. One requirement is that an object must have the correct fields and they in turn correct values. More important is in the context of libraries the treatment of non-existing classes: The definition allows the store to contain objects of classes that do not exist in the type declarations, provided they are not accessible. The question is: Is a compatible store still compatible when new classes are added?

This is important when theorems should be re-used from the library, and the theorems have as a precondition a compatible store. We consider the example from figure 6.2: The specification *Super* contains a class `Super` that extends `Exception`. Suppose we have a theorem that is proved with *compatible* as a precondition:

$$\text{compat}(\mathit{Super}, \text{st}), \langle \mathit{other\ preconditions} \rangle \vdash \langle \text{st}/\alpha \rangle\ \varphi$$

The specification *Sub* enriches *Super* and adds a new class `Sub extends Super`. Using this theorem in specification *Sub* requires the proof of the preconditions, i.e. compat(Super, st) must be proved. The question is: Does compat(Super, st) hold in specification *Sub*? It certainly does not hold for an arbitrary store (this is in itself a drawback to formulating theorems only for a compatible store). But, we could prove all theorems in specification *Sub* with a precondition compat(Sub, st), assuming a store that is compatible the type declarations with the two classes `Super` and `Sub`. Then the question is reduced to:

$$\text{compat}(\mathit{Sub}, \text{st}) \Rightarrow \text{compat}(\mathit{Super}, \text{st})\ ?$$

Unfortunately, this implication never holds, for several reasons:

1. Assume class `Super` contains a field of class `Object`. Then compat(*Super*, st) requires that the value of the field is either **null** or a reference to an object with an existing class type; i.e. either a predefined class or `Super` itself. However, compat(*Sub*, st) also allows that the field points to an object of class `Sub`. This means that compat(*Sub*, st) $\not\Rightarrow$ compat(*Super*, st).

2. The same is true if `Super` contains a field of class `Super`. compat(*Super*, st) requires that the field value is either **null** or a reference to an object of class `Super`. compat(*Sub*, st) also allows an object of class `Sub` since `Sub` extends `Super`.

3. Even if `Super` contains no fields with a class type the implication does not hold because of arrays. The compatibility definition for arrays containing objects requires that the stored objects have an existing class. This means that an Object array in specification *Sub* may contain objects of class `Sub`, but the same array in specification *Super* may not.

There is only one solution to these problems: To weaken the definition of a compatible store, and to strengthen the analysis of accessible (or reachable) references. This, however, makes the definitions and proofs much more complicated, and it is doubtful if this is worth the effort. To conclude: *compatible* should not be used in library specifications. However, in specifications that are not intended as libraries, *compatible* can be used.

Chapter 5.4.1 contained the definition of a compatible store on the semantical level. Particularly, this means that the *compat* definition uses the type declarations to determine the subclass relation, static and instance fields etc. The calculus, on the other hand, never uses the type declarations in the sequents, but only as a context. In the previous sections it was shown how the class hierarchy and the existence of classes and interfaces is handled in the calculus. The same technique can be used to handle *compat* in the calculus: For the predicate compat(st, tds) a predicate compat(specname, st) is used, the same is done for the other compat predicates. For the compatible predicates a generic specification is used. An analysis then shows that besides the class hierarchy and existence of classes only the instance and static fields of every class must be available. This is done simply by generating two additional axioms for every class and interface of the form:

$$\text{instfields(class)} = \text{field}_1 + \ldots + \text{field}_n$$

$$\text{statfields(class)} = \text{field}_1 + \ldots + \text{field}_m$$

The definition of wfChExtend (see chapter 6.2) guarantees this axiomatization is consistent when new classes are added. This finishes the discussion of libraries and *compatible*.

6.8 Context Modifications

Modifications are ubiquitous. They may be the consequence of errors, or of changed or extended functionality. A useful proof system must deal with changes. KIV has a correctness management that keeps track of the dependencies between proofs and theorems. The problem is that after a change it is not clear which proofs are still valid and which are not. The solution is to record all properties that are used in a proof, and to analyse for every proof if these properties are unchanged after a modification. If this is the case the proof should be still valid. This type of book-keeping and analysis is much more efficient than a re-proof, or the generation of proof obligations that are sufficient for the validity of the theorem.

In this section we define the properties that are recorded for the Java proof rules. The main theorem then states that a theorem is still valid if the recorded properties for its proof are unchanged, because the proof is still correct. (Re-doing the proof yields an identical proof.) Most of the analysis has already been done in the previous sections: For proof rules that do not depend on the type declarations at all no properties must be recorded. Proof rules that depend on the class hierarchy use axioms to determine a subtype relation (see chapter 6.4). The usage of axioms and lemmas is covered by the standard correctness management of KIV; therefore no additional properties must be recorded. This means that only the proof rules analysed in chapter 6.5 must be considered:

static, newclass, constr, smethod, nonmethod, imethod

static

The following information is recorded for a *static* proof step:

1. The class or interface to initialize
2. the result of *superClass(class, tds)*
3. the result of *statfields(class, tds)*
4. the result of *statinit(class, tds)*

If the type declarations are modified in an arbitrary manner, then for the new type declarations mod_tds it must be checked

1. that the class or interface still exists
2. that superClass(class, mod_tds) equals the recorded super class
3. that statfields(class, mod_tds) equals the recorded static fields
4. that statinit(class, mod_tds) equals the recorded static initializer

If this is the case the application of the proof rule is still correct. It is obvious that none of the checks can be omitted.

newclass

The following information is recorded for a *newclass* proof step:

1. the *class*
2. the computed instance fields *instfields(class, tds)*

Then it must be checked that the class still exists in mod_tds, and that the instfields(class, mod_tds) equals the recorded instance fields.

constr

The following information is recorded for a *constr* proof step:

1. the *class* of the constructor and the formal types *tys* (*tys* must be recorded so that all arguments to the *getConstr* function are available)
2. the constructor declaration *getConstr(class, tys, tds)*

Then it must be checked that the class still exists, and that getConstr(class, tys, mod_tds) returns the same declaration.

smethod

The following information is recorded for a *smethod* proof step:

1. *class*, *invmo*, *str*, *tys*, ty_0
2. the method declaration *getMethod(invmo.class, str, invmo, tys,* ty_0*, tds)*

Then it must be checked that the class still exists, and that getMethod(invmo.class, str, invmo, tys, ty_0, mod_tds) returns the same declaration.

nonmethod

The following information is recorded for a *nonmethod* proof step:

1. *class*, *invmo*, *str*, *tys*, ty_0
2. the method declaration *getMethod(invmo.class, str, invmo, tys,* ty_0*, tds)*

Then it must be checked that the class still exists, and that getMethod(invmo.class, str, invmo, tys, ty_0, mod_tds) returns the same declaration.

imethod

The following information is recorded for a *imethod* proof step:

1. the selected classes
2. *str*, *invmo*, *tys*, ty_0, and the static type of the invoking expression
3. for every class the computed method declaration *getMethod(class, str, invmo, tys,* ty_0*, tds)*

Then it must be checked that all classes still exist and are subclasses of the static type of the invoking expression, and that getMethod returns the same declaration.

6.8.1 Theorem (correctness management)

If the checks described above return true for a proof step then the step is still correct.

Experience shows that recording and checking the properties is efficient (and very useful), so that currently there is no need for improvements (for example, by a detailed analysis what the modification was). This finishes the discussion of libraries and modifications. The next chapter shows some examples applications.

Chapter 7

Four Examples

This chapter contains four different examples that show the possibilities of the calculus. The first example is a full Java Card program that implements a copy card, and uses the Java Card API including cryptographic functions. The second example shows how the Java Card API is modeled in KIV. The third example is part of a Java Card program that implements decimal numbers, and relies heavily on byte and short arithmetics. The fourth program is a non-Java Card program that implements linked lists, and shows how pointer structures can be modeled.

The examples are not too long, but certainly not trivial. Everybody who has used a tool to do Java or Java Card verification agrees that the proof support should (and could) be better. Every user feels that he has to deal with technical overhead that is not really difficult, but rather tedious, and that it should be handled (more) automatically. However, it turns out that improving the proof support is not that simple. More experimentation is needed. Unfortunately, the main problem areas differ from tool to tool since they are caused by the specific calculus and underlying prover.

The technical overhead is most visible in the second example because the Java Card API (described below) contains several objects and static fields that require a specification of an adequate store (a *compatible* store, see chapter 5.4.1, cannot be used). Then it must be proved that this predicate remains invariant for every modification of the store. As mentioned previously, this is not difficult, but rather annoying. Some improvements have already been made, but more are desirable.

7.1 A CopyCard

An example for a Java Card (or smart card) application is a copy card that is used for example in a university or a library. This requires three components:

- a smartcard that holds 'value points';
- a 'filling station', i.e. a machine that accepts money and loads value points onto the card;
- a 'pay station', i.e. a card reader connected to a copier that subtracts value points before printing a copy.

Of course it is possible to use this application in other scenarios where 'value points' are an adequate business model, and of course we usually have several filling stations, several pay stations and many smartcards. Usage of the card is simple: the card holder inserts the card into the filling station, sees the number of value points left on the card, selects the number of points to load, and inserts the necessary money into the machine. Then the new points are loaded (added) onto the cardlet. To copy with the card, it is simply inserted into the card reader of the copier. The reader displays the remaining number of points, and the copier asks the cardlet to pay (subtract) one or more points before printing a copy. The problem is that the card holder may try to cheat by

loading points onto the card with his home PC (among others). So cryptography is needed. (The protocol is described in more detail in [HRS02].) The idea is that the Java Card program is loaded onto the card without user interference. This means the card holder cannot inspect the code to extract secret keys or pass phrases during loading. And with the correct protections the card holder cannot extract the code from the smart card. A possible Java Card implementation (also called a 'cardlet') for this scenario is shown below. It assumes that the communication between the cardlet and a valid terminal is secure.

```
package swt;

import javacard.framework.*;
import javacard.security.*;
import javacardx.crypto.*;

public class CopyCardCrypt extends Applet {
```

All Java Card applications extend `Applet`, a class of the Java Card API (very different from the `Applet` class in the standard Java distribution). Next the commands that the card accepts, a secret key (used for triple DES encryption), and a secret pass phrase is defined:

```
final static byte AUTH    = 0x02;
final static byte LOAD    = 0x04;
final static byte PAY     = 0x06;
final static byte BALANCE = 0x08;

byte[] secretkey = {
    (byte)0x67, (byte)0x70, (byte)0xE5, (byte)0xE0,
    (byte)0x20, (byte)0x75, (byte)0x46, (byte)0xCE,
    (byte)0x13, (byte)0x61, (byte)0x83, (byte)0xB5,
    (byte)0xC8, (byte)0x16, (byte)0x02, (byte)0x20,
    (byte)0xB3, (byte)0x16, (byte)0xD5, (byte)0x6D,
    (byte)0x9B, (byte)0x57, (byte)0x46, (byte)0x49
};
byte[] secret    = { 1, 2, 3, 4, 5, 6, 7, 8, };

short value = 0;
Cipher cip;
DESKey deskey;
```

The `value` field holds the value points, `cip` and `deskey` are needed for the Java Card crypto API. The constructor for the class then initializes the crypto fields:

```
public CopyCardCrypt() {
    deskey = (DESKey)KeyBuilder.buildKey(KeyBuilder.TYPE_DES,
                                         KeyBuilder.LENGTH_DES3_3KEY,
                                         false);
    deskey.setKey(secretkey, (short)0);
    cip = Cipher.getInstance(Cipher.ALG_DES_ECB_NOPAD,false);
    register();
}

public static void install( byte[] bArray, short bOffset, byte bLength ) {
    new CopyCardCrypt();
}
```

`register()` (the last line in the constructor) is a predefined method from the `Applet` class. It is used to register the applet with the Java Card virtual machine (JCVM). This is part of the

initialization process for applets: Once the applet code has been loaded onto the smart card, the JCVM calls the `install` method of the applet (see above). This method overwrites a dummy `install` method in the `Applet` class. As shown above the `install` method creates and initializes the object and all fields. When `install` completes normally and the applet has registered, then it is ready for use.

A terminal (card reader) and a smart card work in master/slave mode. The terminal sends a command to the smart card, the smart card receives and processes the data, and returns an answer. The card never initiates a communication or sends data on its own. A Java Card may contain several applets. So the first step is to select an applet for further communication. This selection is handled by the JCVM. As a first step the JCVM calls the `select` method of the applet:

```
public boolean select() { return true; }
```

Since this method is guaranteed to be called first the applet 'knows' that a new session has started. This can be used to reset authentication flags, because a smart card does not 'know' when it is extracted from a card reader.

Then all communication is channeled through the applet's `process` method. This method receives the command sent to the card from the JCVM in an `APDU` object. This object contains a byte array with the data.

```
public void process(APDU apdu) throws ISOException {

    switch (apdu.getBuffer()[ISO7816.OFFSET_INS]) {
    case AUTH:    authenticate(apdu); return;
    case LOAD:    load(apdu)        ; return;
    case PAY:     pay(apdu)         ; return;
    case BALANCE: balance(apdu)     ; return;
    }
}
```

The command has a predefined structure (it is an *application protocol data unit*, hence APDU). One byte is usually interpreted as a command (instruction) what the applet should do. In this example the applet accepts four commands, and calls the appropriate method. The four methods are show below.

```
private void authenticate(APDU apdu) throws ISOException {

    short val = apdu.setIncomingAndReceive();
    if (val != 8) ISOException.throwIt(ISO7816.SW_WRONG_LENGTH);
    byte[] buffer = apdu.getBuffer();
    cip.init(deskey, Cipher.MODE_ENCRYPT);
    cip.doFinal(buffer, ISO7816.OFFSET_CDATA, (short)8,
                buffer, ISO7816.OFFSET_CDATA);
    apdu.setOutgoingAndSend(ISO7816.OFFSET_CDATA, (short)8);
}
```

The `authenticate` method is called before points are loaded onto the card. The applet encrypts a challenge with the secret key and sends it back. This guarantees to the terminal that it deals with a genuine card. The two methods `init` and `doFinal` are part of the Java Card crypto API; the other methods are part of the standard Java Card API (described in the next section). The `load` method is used to load points.

```
private void load(APDU apdu) throws ISOException {

    short len = apdu.setIncomingAndReceive();
```

```
        if (len != 2 + secret.length)
           ISOException.throwIt(ISO7816.SW_SECURITY_STATUS_NOT_SATISFIED);
        byte[] buffer = apdu.getBuffer();
        if (0 != Util.arrayCompare(buffer, (short)(ISO7816.OFFSET_CDATA+2),
                                   secret, (short)0, (short)secret.length))
            ISOException.throwIt(ISO7816.SW_SECURITY_STATUS_NOT_SATISFIED);
        short val = Util.getShort(buffer, ISO7816.OFFSET_CDATA);
        if (val <= 0 || (short)(val + value) < 0)
           ISOException.throwIt(ISO7816.SW_DATA_INVALID);
        value += val;
    }
```

The points are accepted only if the secret pass phrase is also sent. (The terminal checked in the previous step that the card is valid, and we assume that the communication is secure.) `Util.arrayCompare` checks the secret. If something goes wrong (or the points are out of range), an exception is thrown with `ISOException.throwIt`.

The `pay` method subtracts points:

```
    private void pay(APDU apdu) throws ISOException {

        short len = apdu.setIncomingAndReceive();
        if (len != 8) ISOException.throwIt(ISO7816.SW_DATA_INVALID);
        byte[] buffer = apdu.getBuffer();
        short val = Util.getShort(buffer, ISO7816.OFFSET_CDATA);
        if (val <= 0 || (short)(value - val) < 0)
           ISOException.throwIt(ISO7816.SW_DATA_INVALID);
        cip.init(deskey, Cipher.MODE_ENCRYPT);
        cip.doFinal(buffer, ISO7816.OFFSET_CDATA, (short)8,
                    buffer, ISO7816.OFFSET_CDATA);
        value -= val;
        apdu.setOutgoingAndSend(ISO7816.OFFSET_CDATA, (short)8);
    }
```

Again, the terminal must check that the card is valid. This is done in the same step: The terminal sends the points and a challenge, the applet encrypts this with the secret key, and sends it back. The terminal will accept the payment only if the data decrypts correctly.

Finally, the value points stored on the card can be retrieved without security measures.

```
    private void balance(APDU apdu) throws ISOException {

        Util.setShort(apdu.getBuffer(), ISO7816.OFFSET_CDATA, value);
        apdu.setOutgoingAndSend(ISO7816.OFFSET_CDATA, (short)2);
    }
```

This is the end of the cardlet. The code as presented above can be compiled and runs on a real smart card.

What properties can be proved about the applet?

1. The `install` method completes normally, and all fields are correctly initialized.

   This is a more specific form of *compat* (see chapter 5.4.1). It specifies that the `secret` field contains a byte array of length 8; `secretkey` a byte array of length 24; `deskey` a key object that is initialized with `secretkey`; and that `cip` is a Cipher object that was initialized with the key.

   Below, we will use the predicate *okcardlet*(st) for this property.

   The other properties then assume this initialization state, and reason only about the `process` method.

2. The values of the fields mentioned above remain unchanged. (Only the `value` field changes its value.)
3. Only ISOExceptions are thrown, otherwise the `process` method completes normally.
4. The `value` field is always ≥ 0.
5. Given an abstract (algebraic) specification of the communication protocol and an abstract representation of the applet's state it can be proved that the program is a correct implementation of the protocol.

The third property is in dynamic logic:

Process-normal :
st[mode] = normal, okcardlet(st), r = st.applet,
cryptoinit(st), javacardinit(st), is_apduref(apdu, st)
$\vdash$ ⟨st/`r.process(apdu);` ⟩ (
st[mode] = normal
$\vee$ is_throw_mode(st[mode])
$\wedge$ st[mode].type = classtype(javacard.framework.ISOException)

The proof the cardlet is correctly initialized requires 201 proof steps and 99 interactions; the proof that only the value is modified 249 proof steps and 47 interactions; the proof that the value is never negative 243 proof steps and 43 interactions; the proof that only ISO exceptions are thrown 244 proof steps and 42 interactions. (It would be more efficient to prove the last three properties together.)

The last property merits some explanation. Suppose, a protocol is specified algebraically, and the smart card part is specified with a function *send* that has a message and an algebraically specified smart card as inputs, and returns the answer of the smart card and the modified card:

send(message, card) = answer × card'

In a Java Card implementation the *send* function becomes a call of the `process` method, and the (abstract) card is related to the applet object. This means it must be possible to define an algebraic function store2card that creates a *card* from a Java store. Furthermore, it must be possible to convert the message into a byte array, and a byte array back into the answer. This leads to the following theorem:

send(message, card) = answer × card', invariant(card),
getmessage(apdu, st) = message, store2card(st) = card, okcardlet(st),
st[mode] = noval
$\vdash$ ⟨st/`Applet.the_applet.process(apdu);`⟩
(store2card(st) = card' $\wedge$ answer(st) = answer)

Basically, it states that the program behaves as the specification. The problem with this type of theorems is that it usually hold only under certain conditions. Hence, the additional precondition *okcardlet(st)*. Furthermore, the theorem implies that the initial *card* is in a state that can occur on the abstract level.

A theorem that reasons about traces could be formulated in the following manner:

oktrace(tr), st[mode] = normal, uninit(st), messages(st) = cardmessages(tr)
$\vdash$ ⟨st/`javacardx.crypto.Cipher.initJCRE();`⟩
⟨st/`swt.CopyCardCrypt.install(null, (short)0, (byte)0);`⟩
⟨st/`Run();`⟩ (store2card(st) = card(first(tr)) $\wedge$ answer(st) = lastCardAnswer(tr))

The idea is to 'store' the messages that are sent to the card in the trace tr in the store. This is expressed by messages(st) = cardmessages(tr). Then the Java Card runtime environment is initialized, `initJCRE()`, the cardlet is installed, `install(...)`, and a kind of simulation method

`Run()` calls the `process` method of the cardlet with the successive messages, and stores the answers. The proof that the cardlet behaves exactly as an abstract protocol specification is much more complicated than the other proofs due to the transition between the Java store and the abstract data types. It requires 869 proof steps and 316 interactions, and dozens of rewrite rules.

The point of the last two example theorems is not to present a refinement concept, or to describe their proof (this would require a detailed description of the algebraic specification of the card which beyond this work), but rather to show the flexibility of the calculus to express such properties.

7.2 The Java Card API

In the previous section a Java Card program was described, and the Java Card environment was mentioned only implicitly. In this section it is described in more detail. The core of the Java Card runtime environment consists of the classes `Applet` (every Java Card application extends `Applet`), `APDU` (used for communication with a smart card reader, `Util` (utility methods for arrays), and `ISOException` (Java Card's throw mechanism). These classes are used in almost every Java Card program. The methods of these classes can be either specified without providing an implementation, or they can be implemented, and properties can be proved as theorems. The latter has the advantage that no unrealistic (unsatisfiable) specifications are written, that the implementation can be tailored to ones needs (as will be explained below), and that it is a nice example for Java Card verification.

In the sequel a sample implementation of the previously mentioned classes will be presented together with some theorems and experiences from the proofs. The implementation can also be used as part of a Java Card simulation environment.

The first class is the `Applet` class. It contains dummy implementations for those methods that correspond to an applet's life cycle.

```
package javacard.framework;

public class Applet {

        /** store the applet. Needed to call the cardlets process method.
         */
        public static Applet the_applet;

/** initialize javacard runtime environment
*/
        public static void init() {
  short sho = APDU.the_answer;
          sho = ISOException.init;
          // short sho = ISO7816.OFFSET_CLA;
          sho = Util.init;
          //Applet.the_applet = null;
       }

protected final void register() { the_applet = this; }

protected final void register(byte[] bArray, short bOffset, byte bLen) {
           the_applet = this;
        }

public boolean selectingApplet() { return false; }

        public void process(APDU apdu) { }
```

```
        public boolean select() { return false; }
}
```

This class is tailored for a simulation environment. It 'stores' the applet in a static field when `register()` is called. This allows access to the applet's `process` method (useful for simulation or when formulating theorems about an applet). The method `init` (not part of the Java Card API) is used to initialize all classes.

The class `ISOException` contains a method `throwIt` that can be used by the applet to signal an error to the environment (the card reader) without allocating memory. This means that only one exception object exists that is reused. This is the stored in the static field `exception_object` below.

```
public class ISOException extends CardRuntimeException {

    static short init;
    static private ISOException exception_object = new ISOException();

    public ISOException () { this((short)0); }
    public ISOException (short x) { super(x); }

    public static void throwIt (short x) throws ISOException {
exception_object.setReason(x);
throw(exception_object);
    }
}
```

The class APDU handles the communication with the card reader. Basically, sequences of bytes (i.e. byte arrays in Java Card) are exchanged. This is encapsulated in an APDU object.

```
public class APDU {

  public final static short APDU_BUFFER_LENGTH = 262;
  private static byte[] buf = new byte[APDU_BUFFER_LENGTH];

  /** initially 9000, meaning: no setOutgoingAndSend called.
   *  set by setOutgoingAndSend to 0
   *  any other value should be an error code
   */
  public  static short the_answer;
  private static short answer_offset;
  private static short answer_length;

  public APDU() {
     the_answer = ISO7816.SW_OK;
  }

  public APDU(byte[] bArray) {
     setAPDU(bArray);
  }

  public void setAPDU(byte[] bArray) {
     the_answer = ISO7816.SW_OK;
     short len = bArray.length;
     if (len > buf.length) len = (short)buf.length;
```

```
      //for(short i=0;i < len; i++) buf[i] = bArray[i];
      Util.arrayCopy(bArray, (short)0, buf, (short)0, len);
   }

   public boolean isSelectAPDU() { return buf[1] == ISO7816.INS_SELECT; }

   public byte[] getBuffer() { return buf; }

   public short setIncomingAndReceive() { return buf[4]; }

   public void setOutgoingAndSend(short offset, short length) {
      answer_offset = offset;
      answer_length = length;
      the_answer = 0;
   }
```

This implementation omits some low level methods that are not really needed. The APDU buffer `buf` is a byte array that is reused. Data sent by the card reader is stored in this byte array, and the applet's `process` method is called with an APDU object that contains the buffer. The applet stores its answer (the data to send back to the card reader) in this buffer and calls `setOutgoingAndSend`. The static fields `the_answer`, `answer_offset`, and `answer_length` are not part of the Java Card API. They are useful to reason about the answer of an applet. The `Run` method mentioned in the previous section to simulate the life of a cardlet can be implemented as:

```
static int num = 0;
static APDU the_apdu = new APDU();

static void Run () { Run(messages.length); }

static void Run (int num) {
       if (num == 0) return;
       Run(num-1);
       setAPDU(messages[num-1]);
       if (apdu.isSelectAPDU())
           Applet.the_applet.select();
       else try {
              Applet.the_applet.process(apdu);
                }
            catch(ISOException e) { APDU.the_answer = e.getReason(); }
}
```

Here, it is assumed that `messages` is a two dimensional byte array containing the messages to send to the `process` method. The code is not Java Card, but that is no problem since it is not intended to run on a smart card. Rather, it is tailored for easy verification. Hence the recursive method with the recursive call before the main body.

The `Util` class contains some useful methods for arrays. They all operate on byte arrays (not short arrays) because the APDU buffer is a byte array.

```
public class Util {

public static short init;

public static short setShort(byte bArray[], short boff, short svalue) {
    bArray[boff] = (byte)((svalue & 0xFF00) >> 8);
    bArray[boff+1] = (byte)(svalue & 0xFF);
```

```
    return (short)(boff+2);
}

public static short getShort(byte barray[], short boff) {
    return (short)((barray[boff] << 8) | (barray[boff+1] & 0xFF));
}

public static short makeShort(byte b1, byte b2) {
    return (short)((b1 << 8) | (b2 & 0xFF));
}

public static short arrayCopy(byte[] src, short srcOff,
                              byte[] dest, short destOff, short length)
{
 if (srcOff < destOff) {
    for(short i=(short)(length-1); i>=0; i--)
       dest[destOff+i] = src[srcOff+i];
    }
    else {
      for(short i=0; i<length; i++) dest[destOff+i] = src[srcOff+i];
    }
    return (short)(destOff + length);
}

public static short arrayCopyNonAtomic(byte[] src, short srcOff,
                                       byte[] dest, short destOff,
                                       short length)
{
    return arrayCopy(src, srcOff, dest, destOff, length);
}

public static byte arrayCompare(byte[] src, short srcOff,
                                byte[] dest, short destOff,
                                short length)
{
    for(short i=0; i<length; i++) {
        if (src[srcOff+i] < dest[destOff+i]) return -1;
        else if (src[srcOff+i] > dest[destOff+i]) return 1;
    }
    return 0;
}

public static void arrayFillNonAtomic(byte[] bArray, short bOff,
                                      short sLen, byte bValue) {
    for(short i=0; i<sLen; i++) bArray[bOff + i] = bValue;
}

}
```

The methods `arrayCopy` and `arrayCompare` use simple for loops. Here, induction or the *for invariant* proof rule (see chapter 4.6) can be used. The `arrayCopy` method actually contains two loops because it must correctly copy one part of an array into another part of the same array, even if the two parts overlap.

Calling `Applet.init` with a store where all classes are uninitialized will initialize the classes

and the seven static fields (one with a byte array, one with an exception object, the other with primitive types). This can be seen as initialization of the Java Card virtual machine that happens exactly once for a real smart card. After that the APDU buffer may be modified, but the reference to the byte array itself remains unchanged. This leads to a definition of a predicate *javacardinit(st)* for a store that captures this situation: All Java Card classes and their static fields are initialized, and contain correct values, and these values remain correct when an applet runs. The definition of *javacardinit* is similar to the generic *compat* predicate for the store (see chapter 5.4.1 and 6.7). However, neither compat(st) → javacardinit(st) nor javacardinit(st) → compat(st) holds, because *javacardinit* is in some respect more liberal, and in other respects more specific:

- **More liberal:** The static field `the_applet` may contain not yet existing classes. This is essential because the not yet existing applets will be stored there. Furthermore, the definition only considers the static fields and their values, not the rest of the store (i.e. the heap).
- **More specific:** The classes must be initialized; the static field `exception_object` may not be **null**; the static field `buf` must contain a byte array of length `APDU_BUFFER_LENGTH`. *compat* requires neither of these.

This is a typical situation. While *compat* is nice to have, it is usually not enough because more specific properties of the store must be known.

Some example properties are:

Init-inits :

st[mode] = normal, uninit(JAVACARD-CLASSES,st)
⊢ ⟨st/`javacard.framework.Applet.init();` ⟩ javacardinit(st);

After initialization *javacardinit* indeed holds. This theorem does not mention other classes, i.e. the store may contain other (already initialized) classes and objects. However, they cannot interfere the the Java Card classes.

arraycompare-ok :

s2i(sho) + s2i(sho1) ≤ st[r – _length] .intval, 0 ≤ s2i(sho), 0 ≤ s2i(sho1),
s2i(sho0) + s2i(sho1) ≤ st[r0 – _length] .intval, 0 ≤ s2i(sho0),
okbytearray(r, st), okbytearray(r0, st), st[mode] = normal, javacardinit(st), st = st0
⊢ ⟨st/`by = Util.arrayCompare(r, sho, r0, sho0, sho1);`⟩
(st = st0
∧ (by = i2b(0)
↔ getbytearray(r, s2i(sho), s2i(sho1), st)
= getbytearray(r0, s2i(sho0), s2i(sho1), st)))

`arrayCompare` compares two byte portions of two byte arrays. This theorem describes the standard behavior of the method: The store is not modified (this implies that no exception occurs), and the result is zero iff the two arrays r and r_0 are equal in the portions described by offsets s2i(sho) and s2i(sho0) and length s2i(sho1). This is expressed by

getbytearray(r, s2i(sho), s2i(sho1), st) = getbytearray(r0, s2i(sho0), s2i(sho1), st)

getbytearray is an additional function for the store that is very useful for Java Card applications. The standard (or normal) behavior of `arrayCompare` requires a surprisingly large number of preconditions:

1. 0 ≤ s2i(sho), s2i(sho) + s2i(sho1) ≤ st[r – _length] .intval,
   0 ≤ s2i(sho0), s2i(sho0) + s2i(sho1) ≤ st[r0 – _length] .intval,
   0 ≤ s2i(sho1),

   These conditions guarantee that no `ArrayIndexOutOfBoundsException` is raised.

2. okbytearray(r, st), okbytearray(r0, st)

   Both references point to correct byte arrays in the store (this also implies that they are not null). This guarantees that no `NullPointerException` is raised and that the arrays contain indeed byte values.

3. st[mode] = normal, javacardinit(st)

   The initial mode must be normal, and the `Util` class must be initialized (guaranteed by javacardinit(st), otherwise the store would be modified).

4. st = st0

   This allows to compare the store before a method is invoked with the store after the method invocation. st0 'preserves' the initial value of the store, while st contains the modified store in the formula following the diamond operator.

On the other hand, these preconditions are not surprising, and similar to [PvdBJ00b] [PvdBJ00a]. The proof of the theorem *arraycompare-ok* requires 57 proof steps and 11 interactions.

For `arrayCopy`, every property that can be formulated is eventually needed: For correct inputs the array is copied correctly, the rest of the store remains unchanged etc. The proofs are comparable to *arraycompare-ok*, but require more effort because the implementation contains two loops instead of one.

The cryptographic methods are treated differently. Obviously, it is not possible to provide a small 'sample' implementation for encryption etc. Therefore, all methods are declared *native*, and their behavior is specified. We illustrate the general approach with the method `Cipher.doFinal`:

```
cip.doFinal(src, offset1, length1, dest, offset2)
```

is used to encrypt some data. `cip` is a Cipher object that has been initialized with a key (and an algorithm to use, and whether to encrypt or to decrypt). `src` is a byte array with the data to encrypt (starting at `offset1` with length `length1`), `dest` is the destination array were the encrypted data will be stored starting at `offset2`. The Java Card API specification for `doFinal` changes from version to version, so one is fairly free to how to specify this method. The first question is: How to treat the fact that this methods encrypts data? This is easily done in the algebraic setting by using an algebraic function *encrypt*. Then we can define that the result data of `doFinal` is equal to *encrypt* of the initial data and the key. In this manner the problem of cryptography is reduced to an algebraic setting. What axioms are needed for *encrypt* depends on the application. The second question is: How to describe the behavior of `doFinal` on the store? `doFinal` modifies the destination array, and (possibly) the cipher object `cip`. It should not allocate new objects (since the Java Card virtual machine has no garbage collection). It should not modify other objects, and it is reasonable to assume that it does not modify anything else (e.g. private static fields that are used during the computation because they cannot be accessed anyway). be expressed in the following manner: There exists a list of valid fields and values such that the new store is equal to the old store where the given fields for a given reference are updated with the given values:

$$\exists\ \text{fis.}\ \text{st} = \text{update}(\text{dest}, \text{fis}, \text{st0}) \wedge \text{validFields}(\text{dest}, \text{fis}, \text{st0})$$

fis is a list of pairs of fields (for an array: indices) and values. So the formula above states that only the values of some array positions of the destination array for the encryption are modified, while the rest of the store remains unchanged. If the cipher object is also modified we can use

$$\begin{aligned}\exists\ \text{fis, fis'.}\quad & \text{st} = \text{update}(\text{cip}, \text{fis'}, \text{update}(\text{dest}, \text{fis}, \text{st0})) \\ & \wedge\ \text{validFields}(\text{dest}, \text{fis}, \text{st0}) \\ & \wedge\ \text{validFields}(\text{cip}, \text{fis'}, \text{st0})\end{aligned}$$

The existential quantifier is no problem, because we assume that the fields exist. The means when the theorem is used the quantifier disappears, and we have a new precondition

$$
\begin{array}{l}
\quad \text{st} = \text{update(cip, fis', update(dest, fis, st0))} \\
\wedge\ \text{validFields(dest, fis, st0)} \\
\wedge\ \text{validFields(cip, fis', st0)}
\end{array}
$$

for some (arbitrary) fields and values. This approach is much better suited for automation than, for example, using a predicate that relates two stores, or quantified formulas that state that "all fields except ... are equal in both stores". Adding a plethora of preconditions, a possible axiom for the correct behavior of `doFinal` is:

$$
\begin{array}{l}
\quad \text{encryptOk(r, st)}, \text{cryptoinit(st)}, \text{javacardinit(st)}, \text{st[mode]} = \text{normal}, \\
\quad \text{okbytearray(bar, st)}, \text{okbytearray(bar0, st)}, \text{st} = \text{st0}, \\
\quad 0 < \text{s2i(sho0)}, 0 \leq \text{s2i(sho)}, \text{s2i(sho)} + \text{s2i(sho0)} \leq \text{st[bar} - \text{_length]} .\text{intval}, \\
\quad 0 \leq \text{s2i(sho1)}, \text{s2i(sho1)} + \text{s2i(sho0)} \leq \text{st[bar0} - \text{_length]} .\text{intval}, \\
\quad \text{s2i(sho0)} \ \%\ \text{cipherBlockLength(r, st)} = 0 \\
\vdash \langle \text{st/}\texttt{r.doFinal(bar, sho, sho0, bar0, sho1);}\rangle\ (\\
\qquad\qquad \text{getbytearray(bar0, s2i(sho1), s2i(sho0), st)} \\
\qquad\quad = \text{encrypt(getCipherKey(r, st0), getbytearray(bar, s2i(sho), s2i(sho0), st0))} \\
\qquad \wedge\ (\exists\ \text{fis.}\ \text{validFields(bar0, fis, st0)} \wedge \text{st} = \text{update(bar0, fis, st0)}))
\end{array}
$$

Obviously, it is easy to miss some (important) preconditions, so a sample implementation is preferable if it is small.

7.3 Decimal Numbers

This example is inspired by [BCHJ04] which is an improved version of [BJvdB02] which in turn is based on [BMGL00]. The idea is to provide an utility Java Card class that implements decimal numbers. The integer part (before the decimal dot) is represented by a short value, and the decimal part (after the dot) is also represented by a short with a given precision of 3 digits. This means the decimal part ranges from 0.0 to 0.999. The idea is to use the class for monetary values (Euro and Cent, for example), and to use the higher precision for currency conversions.

The first version [BMGL00] (which provided only the implementation, but did not do any verification) had very complicated implementations for addition and multiplication of two decimals. [BJvdB02] verified properties for the class, and improved on the addition, but kept the multiplication (which was wrong). Finally, [BCHJ04] provided (and verified) a correct implementation of the multiplication. However, this implementation uses two loops that may run for as much as 32767 iterations, doing 32767 decimal additions. So the idea is to provide an implementation that contains no loops. The problem with the multiplication is the following: Take for example the two decimals 21000.999 and 1.555:

$$21000.999 * 1.555 = 32656.553445$$

This means the integer part fits into a short (correctness of the implementation will only be proved for these cases), but the decimal part should be truncated back to 3-digit precision, hence the result should be

$$32656.5534$$

The standard approach would be: Multiply both number by 1000, do the multiplication, and divide by $1000 * 1000$ to obtain the integer part, and divide by 1000, and then take the remainder of the division by 1000. So to multiply $a.b$ with $c.d$ do:

1. compute $r = (a * 1000 + b) * (c * 1000 + d)$
2. new integer part: $r/(1000 * 1000)$ (integer division)
3. new decimal part: $(r/1000)\%1000$

The problem is that Java Card has no integers. So the result must be computed using only short values, and $r = 32656553445$ does not fit into a short (not even an integer).

So the question is: How to do the multiplication so that all intermediate values fit into shorts, and the result is correct?

The implementation below does this multiplication. It is not an implementation of the decimal class, so for simplicity only static methods and fields are used. The implementation is rather long and quite difficult to understand, because it relies solely on byte- and short arithmetic and Java's bitwise operations. But the auxiliary methods used in the multiplication can be used on their own, so they can be part of a more or less useful arithmetic class.

The first method interprets two bytes as unsigned bytes, and compares them.

```
// view bytes as unsigned bytes and try <
public static boolean less (byte x, byte y) {
  if (x < 0) {
     if (y < 0) return (x < y);
     else return false;
  }
  else { // 0 <= x
     if (y < 0) return true;
     else return (x < y);
     }
}
```

The unsigned value of a positive byte b is the byte b itself; the unsigned value of a negative byte b is $b + 255$. This means if b is negative then its unsigned value is > 127, the maximal unsigned byte value. This leads to the different cases in the implementation. The next method does the same for shorts.

```
// view shorts as unsigned shorts and try <
public static boolean less (short x, short y) {
  if (x < 0) {
     if (y < 0) return (x < y);
     else return false;
  }
  else { // 0 <= x
     if (y < 0) return true;
     else return (x < y);
     }
}
```

The next method multiplies two unsigned bytes, and returns an unsigned short (i.e. all values are interpreted as unsigned, also they are considered as signed by Java).

```
public static short mult (byte x, byte y) {
     return (short)((x & 0xFF) * (y & 0xFF));
}
```

If $x = y = -1$ this is viewed as $x = y = 255$, and the result should be the short value -511 which is unsigned equal to $65025 (= 255 * 255)$.

The next method adds two unsigned short values. The result is a carry byte (either 1 or 0), and an unsigned short. For the carry a static byte field is used, the short is returned by the method.

```
public static byte carry;

// add 2 unsigned short
public static short add (short x, short y) {
     short res = (short)(x + y);
     if (x < 0) {
```

```
            if (y < 0) { carry = 1; return res; }
            else { carry = (byte)(res < 0 ? 0 : 1); return res; }
         }
         else if (y < 0) { carry = (byte)(res < 0 ? 0 : 1); return res; }
              else { carry = (byte)0; return res; }
   }
```

The first addition `(short)(x + y)` may produce an overflow that is discarded by the cast. In this case the result is negative. As mentioned previously the method is correct even if the input is viewed as unsigned (this is not obvious).

The next methods views two shorts as an unsigned integer (high short, low short), and adds two integers, i.e. it has four shorts as input. The result is a carry (the `carry` field is reused), a high short (stored in a new field `high`), and a low short that is returned by the method.

```
static short high;

// add 2 unsigned integers
public static short add (short xhigh, short xlow, short yhigh, short ylow)
{
    short low = add(xlow, ylow);
    short c = carry;
    high = add(xhigh, yhigh);
    if (c == 1) {
       if (high == -1) { high = 0; carry = 1; }
       else high++;
    }
    return low;
}
```

Here, one of the previous defined methods is reused for the first time. The idea is to add the two low shorts and the two high shorts, and handle the carry correctly.

Next we can multiply two unsigned shorts. The result is a high short (stored in the `high` field), and a low short that is returned. The idea is to use school math for the bytes. (A short consists of a high byte and a short byte):

	$b1$		$b2$	$*$	$b3$		$b4$
$+$					$b2$	$*$	$b4$
$+$			$b1$	$*$	$b4$		
$+$			$b2$	$*$	$b3$		
$+$	$b1$	$*$	$b3$				
$r0$	$r1$		$r2$		$r3$		$r4$

The implementation:

```
// multiply two unsigned shorts
public static short mult (short x, short y) {
    short r1 = mult((byte)(x & 0xFF), (byte)(y & 0xFF));
    short r2 = mult((byte)(x & 0xFF), (byte)((y >> 8) & 0xFF));
    short r3 = mult((byte)((x >> 8) & 0xFF), (byte)(y & 0xFF));
    short r4 = mult((byte)((x >> 8) & 0xFF), (byte)((y >> 8) & 0xFF));
    //   r4H r4L   0   0
    // +   0 r3H r3L   0
    // +   0 r2H r2L   0
    // +   0   0 r1H r1L
    //
    short res1 = add(r2, r3);
    high = (short)((carry << 8) + r4);
```

```
        short low = add((short)((res1 & 0xFF) << 8), r1);
        high += carry;
        high = add(high, (short)((res1 >> 8) & 0xFF));
        return low;
    }
```

Next we can divide a positive integer (represented by two shorts) by an unsigned byte. The divisor and the remainder are computed together. The remainder is stored in the new field `rem`. The divisor consists of two shorts, the high short is stored in the `high` field, the low short is returned. The idea is again school arithmetic, the byte divides in turn all four bytes of the dividend.

```
    public static short rem;

    // divide xhigh|xlow by a byte, xhigh must be positive, the byte is unsigned
    // result is high|div(), remainder is in rem
    public static short div(short xhigh, short xlow, byte d) {
        short ud = (short)(d & 0xFF);
        short d1 = (short)(xhigh / ud);
        byte r1 = (byte)(xhigh % ud);
        short x23 = (short)((r1 << 8) | ((xlow >> 8) & 0xFF));
        byte d2 = (byte)(x23 / ud);
        byte r2 = (byte)(x23 % ud);
        short x12 = (short)((r2 << 8) | (xlow & 0xFF));
        byte d3 = (byte)(x12 / ud);
        rem = (short)((x12 % ud) & 0xFF); // should be this
        rem = (byte)(x12 % ud);
        high = d1;
        return (short)((d2 << 8) | (d3 & 0xFF));
    }
```

It should be noted that it is much more difficult to divide an integer by a short using only short arithmetic, because a normalization may be needed that further increases the numbers ([Knu98]). Fortunately, we do not need this.

This was the last of the methods not specific for the decimal implementation. The next method divides an integer by 1000 by dividing it by 100 and again by 10, and adding what was lost.

```
    // divide xhigh|xlow by 1000, xhigh must be positive
    // result is high|div(), remainder is in rem
    public static short div1000(short xhigh, short xlow) {
        short low1 = div(xhigh, xlow, (byte)100);
        short high1 = high;
        short rem1 = rem; // < 100
        short low2 = div(high1, low1, (byte)10);
        short high2 = high; // should be 0
        //i % (j * j0) = i % j + ((i / j) % j0) * j
        // => i % 1000 = i % 100 + ((i / 100) % 10) * 100
        short rem2 = rem; // (i / 100) % 10 < 10
        rem = (short)(rem1 + (short)(100 * rem2));
        return low2;
    }
```

The decimal addition. Here the two shorts are not interpreted as one integer, but as integer part dot decimal part. The resulting integer part should fit into one short. It is stored in the `high` field, the new decimal part is returned.

```
    public static short decimalAdd(short xhigh, short xlow,
                                   short yhigh, short ylow) {
```

```
        high = (short)(xhigh + yhigh);
        short low1 = (short)(xlow + ylow);
        if (low1 < 1000) return low1;
        high++;
        return (short)(low1 - 1000);
    }
```

And finally the decimal multiplication. $xh.xl$ is one decimal, $yh.yl$ the other. We assume that the resulting integer part fits into a short, i.e. $(xh + 1)(yh + 1) < 32768$. The idea is to compute

$$xh.xl * yh.yl = xh * yh + 0.xl * yh + xh * 0.yl + 0.(xl * yl)/1000$$

where $0.x * y$ is computed in the following manner:

$z = x * y, high = z/1000, low = z\%1000$

```
    public static short decimalMult(short xhigh, short xlow,
                                    short yhigh, short ylow) {
        //xh.xl * yh.yl = xh * yh + 0.xl * yh + xh * 0.yl + 0.(xl * yl) / 1000
        // 0.x * y: z = x * y, high = z / 1000, low = z % 1000
        // works only if result fits in a short,
        // i.e. (xhigh + 1)(yhigh + 1) < 32768
        short high1  = mult(xhigh, yhigh);

        short low2 = mult(xlow, yhigh);
        short high2 = high;
        short high2a = div1000(high2, low2);
        short low2a = rem;
        short reslow2 = decimalAdd(high1, (short)0, high2a, low2a);
        short reshigh2 = high;

        short low3 = mult(xhigh, ylow);
        short high3 = high;
        short high3a = div1000(high3, low3);
        short low3a = rem;
        short reslow3 = decimalAdd(reshigh2, reslow2, high3a, low3a);
        short reshigh3 = high;

        short low4 = mult(xlow, ylow);
        short high4 = high;
        short high4a = div1000(high4, low4);
        return decimalAdd(reshigh3, reslow3, (short)0, high4a);
    }
```

This finished the Java Card part of the implementation. The multiplication of two unsigned shorts can be tested with the following method. It uses integers to compare the results, and tests all possible values (ignoring commutativity).

```
    public static void testMult() {
        // needs 14 minutes on laptop
        for (int x = 0; x < 65536; x++) {
            for (int y = 0; y < 65536; y++) {
                short s1 = (short)x;
                short s2 = (short)y;
                int res1 = mult(s1, s2);
                int res2 = high;
                int out = x * y;
```

```
                boolean ok = (out == ((high << 16) | (res1 & 0xFFFF)));
                if (! ok) {
                    throw new RuntimeException();
                }
            }
        }
    }
```

The method tests 2^{32} multiplications. Running this method needs 14 minutes on a laptop. (Since there are $2^{15}*1000$ positive decimals, testing all decimal multiplications requires $250.000*2^{32}$ tests, and one test takes significantly more time.)

This finishes the implementation. Apart from the test method, the implementation is trivial in terms of statements: No loops, no recursion, no exceptions etc. The arithmetic part, on the other hand, is really nasty. We list some properties that can be proved for the methods:

- decimalMult-def :

  $0 \leq$ s2i(xhigh), $0 \leq$ s2i(yhigh), (s2i(xhigh) + 1) * (s2i(yhigh) + 1) $<$ 32768,
  $0 \leq$ s2i(xlow), $0 \leq$ s2i(ylow), s2i(xlow) $<$ 1000, s2i(ylow) $<$ 1000,
  st[mode] = normal, init(st)
  $\vdash$ ⟨st/`sho = Arithmetic.decimalMult(xhigh, xlow, yhigh, ylow);`⟩
  high(st) * 1000 + s2i(sho)
  = ((s2i(xhigh) * 1000 + s2i(xlow))
  * (s2i(yhigh) * 1000 + s2i(ylow))) / 1000

  This is the main theorem: The result *high.sho* (the decimal represented by an integer part *high* and a decimal part *sho*) equals the truncated *xhigh.xlow * yhigh.ylow*. This is tested by multiplying both sides by 1000, so that only integers have to be compared:

  (high(st) * 1000 + s2i(sho)
  = ((s2i(xhigh) * 1000 + s2i(xlow)) * (s2i(yhigh) * 1000 + s2i(ylow))) / 1000

  high(st) selects the value of the `high` field. The algebraic integers are unbounded.

  This obviously requires theorems about the used methods (evaluating all methods by symbolic execution is a bad idea).

- div1000-def :

  $0 \leq$ s2i(xhigh), st[mode] = noval, init(st)
  $\vdash$ ⟨st/`sho = Arithmetic.div1000(xhigh, xlow);`⟩ (
  high(st) * 2^{16} + us2i(sho) = (s2i(xhigh) * 2^{16} + us2i(xlow)) / 1000
  $\wedge$ remainder(st) = (s2i(xhigh) * 2^{16} + us2i(xlow)) % 1000)

  Here, algebraic function for unsigned values are used explicitely: *s2i* converts a (signed) short into an integer, and *us2i* converts a (signed) short that is interpreted as an unsigned short into an integer. This is very useful for the proofs.

- decimalAdd-def :

  $0 \leq$ s2i(xhigh), $0 \leq$ s2i(yhigh), s2i(xhigh) + s2i(yhigh) + 1 $<$ 32768,
  $0 \leq$ s2i(xlow), $0 \leq$ s2i(ylow), s2i(xlow) $<$ 1000, s2i(ylow) $<$ 1000,
  st[mode] = normal, init(st)
  $\vdash$ ⟨st/`sho = Arithmetic.decimalAdd(xhigh, xlow, yhigh, ylow);` ⟩
  high(st) * 1000 + s2i(sho)
  = s2i(xhigh) * 1000 + s2i(xlow) + s2i(yhigh) * 1000 + s2i(ylow)

- mult-short :

  st[mode] = normal, init(st)
  $\vdash$ ⟨st/`sho = swt.util.Arithmetic.mult(sho0, sho1);` ⟩
  (2^{16} * high(st) + us2i(sho) = us2i(sho0) * us2i(sho1))

- div-pos-def :

  $0 < \text{b2i(by)}$, $0 \leq \text{s2i(xhigh)}$, st[mode] = normal, init(st)
  $\vdash \langle \text{st/}$`sho = swt.util.Arithmetic.div(xhigh, xlow, by);` $\rangle$ (
  $(\text{s2i(xhigh)} * 2^{16} + \text{us2i(xlow)}) \;/\; \text{b2i(by)} = \text{high(st)} * 2^{16} + \text{us2i(sho)}$
  $\wedge\ (\text{s2i(xhigh)} * 2^{16} + \text{us2i(xlow)}) \;\%\; \text{b2i(by)} = \text{remainder(st)})$

Et cetera. The proofs require a plethora of theorems about bytes, their bit representation etc.

The `testMult` method shown above tests all possible multiplications. This means that if `testMult` completes normally, the `mult` method is correct. Now it is possible to prove formally this property. This means, instead of proving the `mult` method correct, it is possible to prove the `testMult` method correct, and to run it. If it completes normally, the `mult` method is correct. This, of course, assumes that the Java compiler and virtual machine work correctly for this program. The theorem to prove is:

testMult-theorem : $\langle \text{st/}$`Arithmetic.testMult();` $\rangle$ st[mode] = normal, st[mode] = normal, init(st)
$\vdash \langle \text{st/}$`sho = Arithmetic.mult(sho0, sho1);` $\rangle$
$(2^{16} * \text{high(st)} + \text{us2i(sho)} = \text{us2i(sho0)} * \text{us2i(sho1)})$

It states that, if the `testMult` method completes normally then the `mult` method computes the correct value. This is another nice example of the possibilities of a dynamic logic, because we have two programs, one in the antecedent, one in the succedent. The proof of this theorem is indeed much simpler than the correctness proof for the `mult` theorem itself, though the reasoning about the nested for loops is not that simple. However, almost no arithmetic is required.

All together, 34 theorems have been proved with 1904 proof steps and 784 interactions. This does not include purely algebraic theorems for the casts and bitwise operations from a library. (Most of them already existed.) *decimalMult-def* (239 proof steps and 160 interactions) and *div-pos-def* (234 proof steps and 132 interactions) are the most complicated theorems.

7.4 Linked Lists

An implementation of linked lists is a non Java Card example. It shows how pointer structures can be handled. It is not really a useful implementation, but a rather nice example. The idea is to implement lists of integers (integers for simplicity) by a simple pointer structure with only one pointer to the next element of the list. This is implemented in class `Intlist`. The empty list is represented by an object of class `EmptyIntlist` that extends `Intlist`. `EmptyIntlist` is used only for the empty list; a non-empty list ends with a **null** pointer. (**null** cannot be used for the empty list, because it should be possible to invoke instance methods on list objects. Another possibility is an extra field – a flag, or a length field –, or to encapsulate the stored value in an object that is null for the empty list.) Since `EmptyIntlist` extends `Intlist`, most methods of `Intlist` are overwritten. For example, the method `isEmpty` simply returns **false** in class `Intlist`, but **true** in `EmptyInlist`.

We begin with the source code for `EmptyIntlist`.

```
class EmptyIntlist extends Intlist {

    public boolean isEmpty() { return true; }
    public Intlist cons (int i) { return new Intlist(i, null); }
    public int length() { return 0; }
}
```

The method `cons` adds an element to the front of the list. The class `Intlist` contains other methods that are not overwritten.

```
public class Intlist {

    private int first;
    private Intlist rest;

    public static Intlist nil() { return new EmptyIntlist(); }

    protected Intlist () { }

    protected Intlist (int i, Intlist r) { first = i; rest = r; }
    protected Intlist cons (int i) { return new Intlist(i,this); }

    public boolean isEmpty() { return false; }

    public int first() { return first; }
    public Intlist rest() { return (rest == null ? nil() : rest); }

    public int length () {
        if (rest == null) return 1;
        else return 1 + rest.length();
    }
```

These methods form the core of the list implementation. The field `first` contains the stored integer value, the `rest` field the pointer to the next list element (or **null**). An empty list can be obtained with the static `nil` method. `first` and `rest` return the first element of the list and the rest of the list, respectively. If the rest is **null**, an `EmptyIntlist` object must be returned. A user of the list should not call the constructors, but create the list using `nil` and `cons` (or one of the methods below).

The class `EmptyIntlist` overwrites three methods: `isEmpty`, `cons`, and `length`. `isEmpty` checks if the list is empty, so it simply returns **true** in `EmptyIntlist` and **false** in `Intlist`. `length` returns the length of the list. It does not terminate if the list is cyclical.

The `append` method uses an auxiliary method `apprec` to append (concatenate) two lists. The first list is modified by replacing the final **null** pointer with a pointer to the beginning of the second list. The `append` method handles the case that one list is empty.

```
    public Intlist append (Intlist y) {
        if (isEmpty()) return y;
        if (y.isEmpty()) return this;
        apprec(y);
        return this;
    }

    private void apprec (Intlist y) {
        if (rest == null) rest = y;
        else rest.apprec(y);
    }
```

`append` and `apprec` assume that the first list is not cyclical; `apprec` does not terminate for a cyclical list. This means it is necessary to express this property on the store when proving properties about `append`.

`copy` copies a list, i.e. it returns a new pointer structure with the same values as the original list. `copy` does not terminate for a cyclical list.

```
    public Intlist copy () {
        if (isEmpty()) return this;
        else if (rest == null) return new Intlist(first, null);
```

```
        else return rest.copy().cons(first);
    }
```

The last two methods check if a list is cyclical. The implementation is rather unusual. The idea is to break a possible cycle by replacing the `rest` pointers, and restoring them when the test has been decided. `cyclic` handles the case that the list is empty. If not, a new empty list is created and `cyclicrec` is called. The new empty list object serves as the target for the replaced `rest` pointers.

```
    public boolean cyclic () {
        if (isEmpty()) return false;
        else return cyclicrec(nil());
    }

    private boolean cyclicrec (Intlist elem) {
        if (rest == null) return false;
        if (rest instanceof EmptyIntlist) return true;
        Intlist cur = rest;
        rest = elem;
        boolean res = cur.cyclicrec(elem);
        rest = cur;
        return res;
    }
```

Correct non-empty lists end with a **null** pointer. Therefore, if a **null** pointer is reached the list was not cyclical, and **false** is returned (line one of `cyclicrec`). Otherwise, if an `EmptyIntlist` object is reached it must be a `rest` pointer that has been replaced previously. This means that the list is cyclical (line two). Otherwise the correct `rest` pointer is saved in a local variable `cur` (line 3), and the `rest` field is directed to the `EmptyIntlist` object (line 4). After the recursive call (line 5) the `rest` field is restored (line 6) and the result returned (line 7). It can be seen that `cyclicrec` always terminates, and – when finished – leaves the store unchanged.

To prove properties about the methods requires the possibility to express, for example, that a reference points to a correct list, and that a list is cyclical or acyclical. This can be done with suitable predicates on the store that reason about references. The approach is similar to [MN03]. The top-level predicate is *oklist*. oklist(r, st) is true if r is a reference to a valid list.

$$\text{oklist-def}: \text{oklist}(r, st) \leftrightarrow \text{is_obj}(r, \text{EmptyIntlist}, st) \vee \text{okref}(r, st)$$

A valid list is either an empty list (an object of class `EmptyIntlist`) or a non-empty list. The latter case is handled by okref.

$$\begin{aligned}\text{okref-def}: \quad & \text{okref}(r, st) \\ \leftrightarrow \quad & r \neq \text{null} \\ & \wedge \exists\, \text{refs}. \quad \neg\, \text{dups}(\text{refs}) \wedge \text{okrefs}(r, \text{refs}, st) \\ & \qquad\quad \vee\, \text{dups}(\text{refs}) \wedge \text{okrefscyc}(r, \text{refs}, st)\end{aligned}$$

okref distinguishes between acyclical and cyclical lists. A list is acyclical if there exists a list (an algebraic list) of references that correspond to the list structure (and that have no duplicates). This is captured by $\exists$ refs. $\neg$ dups(refs) $\wedge$ okrefs(r, refs, st). For example, the references $[r_1, r_2, r_3]$ form an acyclical list:

Otherwise a valid list is cyclical. This means there exists an algebraic list of references that corresponds to the list structure and contains duplicates. (If the same reference occurs twice the list must be cyclical.) This case is handled by $\exists$ refs. dups(refs) $\wedge$ okrefscyc(r, refs, st). The difference between okrefs and okrefscyc is that okrefs requires that the list ends with **null**

which okrefscyc does not. For example, $[r_4, r_5, r_6, r_5]$ corresponds to the cyclical list above, but also $[r_4, r_5, r_6, r_5, r_6, r_5, r_6]$. $[r_4, r_5]$ does not represent a valid list because it may or may not be cyclical.

okrefs-empty : okrefs(r, [], st) $\leftrightarrow$ r = null

okrefs-rec : okrefs(r, r0 + refs, st)
$\leftrightarrow$ r = r0 $\wedge$ is_obj(r, Intlist, st) $\wedge$ okrefs(st[r – rest].refval, refs, st)

If the algebraic list is empty the reference must be **null**, i.e. the pointer structure in the store ends with **null**. Otherwise the reference must be equal to the first list element, it must be an object of class `Intlist`, and the rest must be again a valid list.

okrefscyc-empty : okrefscyc(r, [], st)

okrefscyc-rec : okrefscyc(r, r0 + refs, st)
$\leftrightarrow$ r = r0 $\wedge$ is_obj(r, Intlist, st) $\wedge$ okrefscyc(st[r – rest].refval, refs, st)

The specification of okrefscyc is identical to okrefs except for the base case. For convenience we can define a predicate cyclic:

cyclic-def : cyclic(r, st) $\leftrightarrow \exists$ refs. dups(refs) $\wedge$ okrefscyc(r, refs, st)

There are several possibilities to reason about the elements of a linked list. One possibility is to select the elements with a function getlist specified below:

getlist-no : $\neg$ is_obj(r, EmptyIntlist, st) $\wedge \neg$ is_obj(r, Intlist, st) $\rightarrow$ getlist(r, st) = []

getlist-empty : is_obj(r, EmptyIntlist, st) $\rightarrow$ getlist(r, st) = []

getlist-rec : is_obj(r, Intlist, st)
$\rightarrow$ getlist(r, st) = st[r – first].intval + getlist(st[r – rest].refval, st – r)

In the third axiom the reference is deleted from the store, st – r. This is necessary for cyclical lists, because the list returned by getlist must be finite. For an acyclical list it can be proved that it is not necessary to delete the reference,

is_obj(r, List, st) $\wedge \neg$ cyclic(r, st)
$\rightarrow$ getlist(r, st) = st[r – first] .intval + getlist(st[r – rest] .refval, st)

The proof requires a generalization that a reference not occurring in the list can be deleted which in turn can be proved by induction on the number of references in the store. Then different properties for the methods can be proved, for example

- Length-def :

  oklist(r, st), $\neg$ cyclic(r, st), init(st), st[mode] = normal, st0 = st
  $\vdash \langle$st/`i = r.length();`$\rangle$ (st0 = st $\wedge$ i = n2i(# getlist(r, st)))

  `length` terminates for an acyclical list and returns a result that is equal to the length returned by getlist (# is the length of a list as a natural number, n2i converts the natural into an integer), i.e. returns the correct length of the list. The store remains unchanged.

  The proof is done by expanding oklist – leading to okrefs(r, refs, st) – and induction on the length of refs. The proof has 41 steps and 16 interactions.

- Copy-copies :

  okref(r, st), $\neg$ cyclic(r, st), init(st), st[mode] = normal, st0 = st
  $\vdash \langle$st/`r0 = r.copy();`$\rangle$ (
  st[mode] = normal $\wedge$ init(st)
  $\wedge$ ($\exists$ st1. st = st1 $\cup$ st0)
  $\wedge$ okref(r0, st) $\wedge \neg$ cyclic(r0, st)

$\wedge$ is_newref_list(getrefs(r0, st), st0)
$\wedge$ getlist(r, st) = getlist(r0, st))

This theorem states various properties for the `copy` method: The new store is the old store plus some new references; a new reference r0 is returned, that points to an acyclical list; all references of this list are new; the new list contains the same elements as the old list. The proof requires 118 proof steps and 54 interactions.

- Apprec-works :

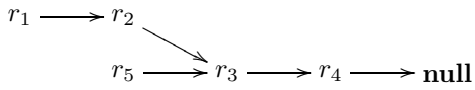
  okrefs(xr, refs, st), okrefs(yr, refs0, st), refs $\neq$ [], refs0 $\neq$ [],
  disj(refs, refs0), st[mode] = normal, init(st), st0 = st
  $\vdash \langle$st/`xr.apprec(yr);`$\rangle$ (
  st[mode] = normal $\wedge$ init(st)
  $\wedge$ okrefs(xr, refs + refs0, st)
  $\wedge \neg$ cyclic(xr, st) $\wedge$ okrefs(yr, refs0, st)
  $\wedge$ getlist(xr, st) = getlist(xr, st0) + getlist(yr, st0)
  $\wedge$ st = st0[refs .last – rest, refval(yr)])

  Again, various properties can be stated for `apprec`. The preconditions guarantee that both lists are acyclic (okrefs requires that the list ends with **null**), and that they are disjoint, i.e. they have no references in common. Then the result list is still acyclical and contains the references of both lists, okrefs(xr, refs + refs0, st), and the elements are indeed the appended elements of both lists. It is also possible to describe precisely how the store is modified: The `rest` field of the last reference of the first list is set to the first reference of the second list, the rest of the store remains unchanged, st = st0[refs .last – rest, refval(yr)]. This proof requires 84 proof steps and 39 interactions.

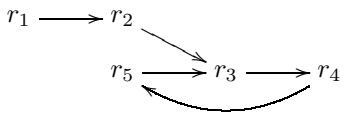
- Apprec-not-disjoint :

  okrefs(xr, refs, st), okrefs(yr, refs0, st), refs $\neq$ [], refs0 $\neq$ [],
  $\neg$ disj(refs, refs0), st[mode] = normal, init(st), st0 = st
  $\vdash \langle$st/`xr.apprec(yr);`$\rangle$ (
  st[mode] = normal $\wedge$ init(st)
  $\wedge$ cyclic(xr, st) $\wedge$ cyclic(yr, st)
  $\wedge$ st = st0[refs .last – _rest, refval(yr)])

  This theorem is rather unusual. The preconditions state the two lists to append are acyclical, but have at least one reference in common. This also implies that all remaining references are identical:

  Then `apprec` creates two cyclical lists:

  This is expressed by the theorem. The proof has 95 steps and 45 interactions.

The `cyclic` method creates a new `EmptyIntlist` object and calls `cyclicrec`. The main property is that the method always terminates, correctly tests for a cyclical list, and does not modify the store except for the new object:

- Cyclic-def :

  st[mode] = normal, oklist(xr, st), init(st), st0 = st
  $\vdash \langle$st/`boolvar = xr.cyclic();`$\rangle$ (

(boolvar ↔ cyclic(xr, st))
∧ (¬ is_obj(xr, EmptyIntlist, st) → (∃ r. st = addobj(r, EmptyIntlist, st0))))

This is proved by unfolding the definition of `cyclic` and oklist, and with the following two theorems.

- Cyclicrec-nocyclic :

  okrefs(xr, refs, st), refs ≠ , init(st), st[mode] = normal,
  is_obj(r0, EmptyIntlist, st), st0 = st
  ⊢ ⟨st/`boolvar = xr.cyclicrec(r0);`⟩ (¬ boolvar ∧ st = st0)

  This theorem states that `cyclicrec` terminates for an acyclical list and does not modify the store. The proof has 83 steps and 34 interactions.

- Cyclicrec-cyclic :

  cyclic(xr, st), init(st), st[mode] = normal, is_obj(r0, EmptyIntlist, st), st0 = st
  ⊢ ⟨st/`boolvar = xr.cyclicrec(r0);`⟩ (boolvar ∧ st = st0)

  If the list is cyclical the result is **true** and the store is also unchanged. This proof requires the auxiliary theorem below and has 106 steps with 38 interactions.

- Cyclicrec-inner :

  st = st0[r – rest, refval(r0)], st[mode] = normal, is_obj(r0, EmptyIntlist, st0), init(st),
  okrefscyc(xr, refs, st0), ¬ dups(refs), r ∈ refs
  ⊢ ⟨st/`boolvar = xr.cyclicrec(r0);`⟩ (boolvar ∧ st = st0[r – rest, refval(r0)])

  This theorem is necessary because redirecting a pointer to an `EmptyIntlist` object creates a list that is not valid. The proof has 117 proof steps and 48 interactions.

Most interactions deal with the predicate logic parts of the theorems. As mentioned above, the class is not a useful implementation of linked lists, but shows how pointer structures can be modeled.

7.5 An example proof

We present a short example proof that demonstrates some features of the calculus, especially the flattening of blocks and the handling of jumps. Actually, the proof is just a symbolic evaluation of the program, and runs automatically.

```
public class returnt {

    static int x = 1;

    public static int me1 () {
        int x = 3;
        try { return x; }
        finally { return x / (x - 3); }
    }

    public static void main (String[] argv) {
        try { x = me1(); }
        catch (ArithmeticException e) { System.out.println(x); }

    }
}
```

The program compiles, runs, and prints 1. The static initialization sets `x` to 1. Then method `me1` is called. `return x;` does not return immediately, but enters the finally clause that raises an ArithmeticException. No assignment to `x` occurs, the exception is caught, and the initial value of `x`, i.e. 1, is printed.

We want to prove the following goal:

st[mode] = normal, initundone(returnt, st), _out(st) = noval
⊢ ⟨st/returnt.main();⟩ _out(st) = noval ++ intval(1)

`initundone(returnt, st)` states explicitly that the class `returnt` is uninitialized (instead of initialized or erroneous). `_out(st)` is used to select the list of outputs by System.out.print(ln). For simplicity, System.out.print is transformed by the annotation program that creates the abstract syntax tree (see chapter 2) into a native method call _out. _out adds its argument to the list of outputs. An initial value `noval` means that we have no outputs yet while after the program the output consists of the integer value 1 (_out(st) = noval ++ intval(1)). In the sequel _out is rewritten to st[_out] = noval and flatten(st[_out]) = noval ++ intval(1). This is a very simple possibility to handle outputs. Applying the rule for static method invocation (4.3.24, p. 84) yields four premises:

1. st[_out] = noval, initundone(returnt, st), st[mode] = normal, st[mode] ≠ normal,
   flatten(st[_out]) ≠ noval ++ intval(1) ⊢
2. st[_out] = noval, initundone(returnt, st), st[mode] = normal, st[mode] = normal,
   ¬ initdone(returnt, st)
   ⊢ ⟨st/static(returnt)⟩ ⟨st/returnt.main();⟩ _out(st) = noval ++ intval(1)
3. st[_out] = noval, initundone(returnt, st), st[mode] = normal, st[mode] = normal,
   initdone(returnt, st)
   ⊢ ⟨st/try { returnt.x = returnt.me1(); }
   ArithmeticException (e) { object._out(returnt.x); }⟩ ⟨st/return;⟩
   ⟨st/target(return(refval(jvmref), void_type))⟩ flatten(st[_out]) = noval ++ intval(1)

The first goal is for the jump case. However, since st[mode] = normal is true, this goal is closed immediately. The third goal is for the case that `returnt` is already initialized, which is false. This means the goal is also closed immediately. We continue with the second goal that deals with the static initialization. This is done by the `static` statement. Applying the `static` rule adds the static initializer `returnt.x = 1;` and an `endstatic` statement. After evaluating the assignment only the `endstatic` remains:

st_0 = addclass(returnt, returnt.x × intval(0), st),
st[_out] = noval, st[mode] = normal, initundone(returnt, st),
st_1 = st_0[jvmref – returnt.x ', intval(1)],
st_0[mode] = normal, initdone(returnt, st_0)
⊢ ⟨st_1/endstatic(returnt)⟩ ⟨st_1/returnt.main();⟩ flatten(st_1[_out]) = noval ++ intval(1)

Static fields are stored under the special reference `jvmref`, and st_0[jvmref – returnt.x ', intval(1)] means that the static field returnt.x is set to the integer value 1 in the store st_0. The `endstatic` is discarded, and `main` is called again. However, this time only the fourth premise of the rule remains since the class is now initialized:

st_1 = addclass(returnt, returnt.x × intval(1), st),
st[_out] = noval, st[mode] = normal, initundone(returnt, st)
⊢ ⟨st_1/try { returnt.x = returnt.me1(); }
catch ArithmeticException (e) { object._out(returnt.x); }⟩
⟨st_1/target(return)⟩ flatten(st_1[_out]) = noval ++ intval(1)

The `main` call is replaced by its body, and a `target` statement is added that will catch returns. (Since the body does not contain any `return`'s it is a simple optimization to omit the target statement.) Application of the try rule together with the rules block and flattening leads to

st_1 = addclass(returnt, returnt.x × intval(1), st),
st[_out] = noval, st[mode] = normal, initundone(returnt, st)
⊢ ⟨st_1/i = returnt.me1();⟩ ⟨st_1/returnt.x = i;⟩
⟨st_1/catch ArithmeticException (e) { object._out(returnt.x); }⟩
⟨st_1/target(return)⟩ flatten(st_1[_out]) = noval ++ intval(1)

The catcher for the ArithmeticException becomes a new *catches* statement, and a new local variable `i` is introduced. The next method call for `me1` together with the block rule gives

st_1 = addclass(returnt, returnt.x × intval(1), st),
st[_out] = noval, st[mode] = normal, initundone(returnt, st)
⊢ ⟨st_1/x_0 = 3;⟩ ⟨st_1/try { return x_0; } finally { return x_0 / x_0 - 3; }⟩ ⟨st_1/targetexpr(i)⟩
⟨st_1/returnt.x = i;⟩ ⟨st_1/catch ArithmeticException (e) { object._out(returnt.x); }⟩
⟨st_1/target(return)⟩ flatten(st_1[_out]) = noval ++ intval(1)

The first line of the succedent contains the body of the `me1` method, the following lines the body of the `main` method. The new method body is just added in front of the program. The rest remains unchanged throughout the proof (except for some variable renamings), and is completely irrelevant until the preceding statements have been executed. The local variable declaration `int x = 3;` is transformed by the block rule into an assignment x_0 = 3; (with x renamed to x_0). Note that `returnt.x` is a static field access and distinguished from a variable `x` or x_0. After the assignment and try, the `return` x_0`;` statement jumps to the `finally` clause that introduces an `endfinally`:

st_1 = addclass(returnt, returnt.x × intval(1), st),
st[_out] = noval, x = 3, st[mode] = normal, initundone(returnt, st)
⊢ ⟨st_1/{ return x / x − 3; }⟩ ⟨st_1/endfinally(return(intval(x)))⟩ ⟨st_1/targetexpr(i)⟩
⟨st_1/returnt.x = i;⟩
⟨st_1/catch ArithmeticException (e) { object._out(returnt.x); }⟩
⟨st_1/target(return)⟩ flatten(st_1[_out]) = noval ++ intval(1)

The `endfinally` statements records in its argument the mode `return(intval(x))` that was active when the `finally` statement was reached. Flattening leads to

st_1 = addclass(returnt, returnt.x × intval(1), st),
st[_out] = noval, x = 3, st[mode] = normal, initundone(returnt, st)
⊢ ⟨st_1/i_1 = x - 3;⟩ ⟨st_1/i_0 = x / i_1;⟩ ⟨st_1/return i_0;⟩
⟨st_1/endfinally(return(intval(x)))⟩
⟨st_1/targetexpr(i)⟩ ⟨st_1/returnt.x = i;⟩
⟨st_1/catch ArithmeticException (e) { object._out(returnt.x); }⟩
⟨st_1/target(return)⟩ flatten(st_1[_out]) = noval ++ intval(1)

Here, two new variables i_0 and i_1 have been introduced, one for the evaluation of the `return` expression, one for the second argument of the binary division. After evaluation of the subtraction and the assignment to i_1, the exbin rule creates an arithmetic exception for the division:

st_1 = addclass(returnt, returnt.x × intval(1), st),
st[_out] = noval, i_2 = 0, st[mode] = normal, initundone(returnt, st),
x_0 = 3, st_1[mode] = normal, i_2 = 0
⊢ ⟨st_1/throw new ArithmeticException();⟩ ⟨st_1/return i_0;⟩
⟨st_1/endfinally(return(intval(x_0)))⟩
⟨st_1/targetexpr(i)⟩ ⟨st_1/returnt.x = i;⟩
⟨st_1/catch ArithmeticException (e) { object._out(returnt.x); }⟩
⟨st_1/target(return)⟩ flatten(st_1[_out]) = noval ++ intval(1)

`new` creates a new object (with addobj) that is finally thrown:

st_0 = addobj(r_0, ArithmeticException, @, addclass(returnt, returnt.x × intval(1), st)),
st[_out] = noval, x = 3, $r_0 \neq$ jvmref, st[mode] = normal, initundone(returnt, st),
newref(r_0, st)
$\vdash$ ⟨st_0/throw r_0;⟩ ⟨st_0/return i_0;⟩
⟨st_0/endfinally(return(intval(x)))⟩
⟨st_0/targetexpr(i)⟩ ⟨st_0/returnt.x = i;⟩
⟨st_0/catch ArithmeticException (e) { object._out(returnt.x); }⟩
⟨st_0/target(return)⟩ flatten(st_0[_out]) = noval ++ intval(1)

r_0 is a new reference (newref(r_0, st)) that points to the newly created object. The throw leaves the body of the `me1` method and jumps directly to the catch clause of the `main` method which is transformed into a block with a local variable declaration:

st_0 = addobj(r_0, ArithmeticException, @, addclass(returnt, returnt.x × intval(1), st)),
st[_out] = noval, $r_0 \neq$ jvmref, st[mode] = normal, initundone(returnt, st),
newref(r_0, st)
$\vdash$ ⟨st_0/{ catch ArithmeticException e = r_0; object._out(returnt.x); } ⟩
⟨st_0/target(return)⟩ flatten(st_0[_out]) = noval ++ intval(1)

Finally, the out method is called that adds its argument to the list of outputs:

st_0 = addobj(r_0, ArithmeticException, @, addclass(returnt, returnt.x × intval(1), st)),
st[_out] = noval, $r_0 \neq$ jvmref, st[mode] = normal, initundone(returnt, st),
newref(r_0, st), st_0[mode] = normal, st_1 = st_0[_out, st_0[_out] ++ intval(1)]
$\vdash$ ⟨st_1/target(return)⟩ flatten(st_1[_out]) = noval ++ intval(1)

A last application of the return rule finishes the proof. As mentioned previously the proof is done automatically. If the heuristics stop the user only has to look at the first statement of the diamonds. The remaining statements can be ignored until they are evaluated.

Chapter 8

Conclusion

This thesis presented a formal semantics for sequential Java; a calculus based on dynamic logic together with a formal correctness proof; extensions like support for building libraries and correctness management; a type soundness proof; some examples. The calculus is implemented in the KIV system together with the correctness management. The core of the Java Card API has been either implemented or (the cryptographic methods) specified.

The specification of the Java store and auxiliary functions that the calculus uses comprises about 40 specifications with 580 axioms and 2500 proved theorems (not all by the author). This does not count standard specifications from a library like lists or sets, but include the bitwise operations for Java's primitive types.

Specifying the semantics and proving type soundness and soundness of the calculus is in itself a large case study in structured algebraic specifications. The model became more refined over time. For example, an early first version did not model sorts for the predicate logic parts, and all variables were assumed to have the sort javavalue. An earlier version also did not have the Java type declarations as a global context, but as part of the boxes/diamonds. When this was changed KIV's correctness management for algebraic specifications was invaluable. These specifications are based on the specification of the Java store. They add about 50 specifications with approximately 2000 axioms. The large number of axioms is not really surprising considering that there are 24 Java expressions and 23 Java statements. Every function (for example, variables, free variables, assigned variables) and predicate (for example for type correctness) that deals with statements and expressions requires about 47 axioms. Almost 4000 theorems have been proven. (It should be noted that 'theorem' here includes simplification rules that are only a slight variant of another theorem and that are often proved automatically or with one interaction.) As it turned out, the most important questions became: "What will happen (to the proofs) if something is changed? How many proofs become invalid? How often is a function or predicate used?" A prover should support this kind of questions. Sometimes, however, the only solution is to do the change, and see what happens. Internally, KIV handles 35000 simplification rules. This large number is due to automatically generated rules for freely generated data types, and multiple instances of lists, sets etc. for different instantiations. This shows that provers are capable of handling large verification projects, and a formalization of a real-world programming language is a large project. The Java developers had many design goals in mind when they created the language; perhaps future language designers also keep an eye on *easy to formalize*. It is very difficult to see if a formalization captures the intricate details of Java correctly. A small test suite of about 100 Java programs can be used to test the calculus, the semantics, and the type checks. The idea is to have a small Java program that can be proved automatically, and to compare the output when the program is run (using a normal Java compiler and virtual machine) with a symbolic execution of the program using the proof rules of the calculus. This helps to reveal errors in the semantics, or to test rules of the calculus before proving them correct. It also helps to find type checks that are too restrictive. For example, at one time the type check for an array assignment required that the static type of the right hand side of the assignment is a subtype of the array's static element

type. This inadvertently prohibited assigning null (i.e. `a[x] = null;`) because null has a `void` type that is not subtype of another type. This was found only because the type check failed for one of the example programs. However (or: obviously) these tests cannot catch all errors. One error occurred only when `finally` statements were nested, but the test suite did not contain an example for this (not to mention the `x = x++;` problem).

Future work includes better proof support for the explicit store; some optimized proof rules; a stronger version of the calculus that uses a compatible store; and support for Java applications that run, for example, on mobile phones or personal (mobile) digital assistants. A very promising approach for the latter was explored in a master's thesis [Gra04]. The idea is to divide the source code into a security related and a non-security related part (e.g. graphical user interface). This requires some support from the calculus because the non-security source code is discarded which in turn violates some assumptions about the existence of types etc. Furthermore, Java applications running on mobile devices use more types (most notably strings) and more classes from the standard Java distribution. It does not seem to be necessary to support threads because security related Java code typically runs in one thread. Other possible extensions are support for the Java modeling language JML that is used by several tools (see chapter 1); or support for inner classes (this requires only small extensions for the run time semantics, because most of the work is done by the compiler); or a formal specification how the compiler creates the annotated abstract syntax tree from the source code (or an abstract syntax tree without annotations). But more important, and possibly more rewarding, is to *use* the calculus in real applications.

Bibliography

[ABB+03] W. Ahrendt, T. Baar, B. Beckert, R. Bubel, M. Giese, R. Hähnle, W. Menzel, W. Mostowski, A. Roth, S. Schlager, and P. Schmitt. The KeY tool. Technical Report 2003-5, University of Karlsruhe, Department of Computer Science, 2003.

[ABB+04] W. Ahrendt, T. Baar, B. Beckert, R. Bubel, M. Giese, R. Hähnle, W. Menzel, W. Mostowski, A. Roth, S. Schlager, and P. Schmitt. The KeY tool. *Software and Systems Modeling*, April 2004. Online First issue, to appear in print.

[Abr96] J.-R. Abrial. *The B Book*. Cambridge University Press, 1996.

[Abr04] Erika Abraham. *An Assertional Proof System for Multithreaded Java – Theory and Tool Support –*. PhD thesis, University of Leiden, Netherlands, 2004.

[AC96] M. Abadi and L. Cardelli. *A theory of objects*. Springer, 1996.

[AF99] J. Alves-Foss, editor. *Formal Syntax and Semantics of Java*. Springer LNCS 1523, 1999.

[AFL99] J. Alves-Foss and F. Lam. Dynamic Denotational Semantics of Java. In J. Alves-Foss, editor, *Formal Syntax and Semantics of Java*. Springer LNCS 1523, 1999.

[Ast91] Egidio Astesiano. Inductive and operational semantics. In *Formal Description of Programming Concepts*, IFIP State-of-the-Art Reports. Springer, 1991.

[BCC+03] L. Burdy, Yoonsik Cheon, David Cok, Michael Ernst, Joe Kiniry, Gary T. Leavens, K. Rustan M. Leino, and Erik Poll. An overview of jml tools and applications. In Thomas Arts and Wan Fokkink, editors, *Eighth International Workshop on Formal Methods for Industrial Critical Systems (FMICS '03)*. Volume 80 of Electronic Notes in Theoretical Computer Science, Elsevier, 2003.

[BCHJ04] C.-B. Breunesse, N. Catano, M. Huisman, and B.P.F. Jacobs. Formal methods for smart cards: an experience report. Technical Report ICIS report NIII-R0316, University of Nijmegen, 2004.

[BDRS02] M. Balser, C. Duelli, W. Reif, and G. Schellhorn. Verifying concurrent systems with symbolic execution. *Journal of Logic and Computation*, 12(4):549–560, 2002.

[Bec00] Bernhard Beckert. A dynamic logic for the formal verification of java card programs. In I. Attali and T. Jensen, editors, *Java on Smart Cards: Programming and Security, Proceedings of the First International Workshop, Java Card 2000, Cannes, France, September 14, 2000*. Springer LNCS 2041, 2000.

[BJvdB02] C. Breunesse, B. Jacobs, and J. van den Berg. Specifying and verifying a decimal representation in Java for smart cards. In *Proceedings AMAST 2002*, Reunion Island, France, 2002. Springer LNCS 2422.

[BM03] B. Beckert and W. Mostowski. A program logic for handling java card's transaction mechanism. In M. Pezze, editor, *Fundamental Approaches to Software Engineering (FASE)*. Springer LNCS 2621, 2003.

[BM04] Michel Bidoit and Peter D. Mosses. CASL *User Manual*. LNCS 2900 (IFIP Series). Springer, 2004. With chapters by T. Mossakowski, D. Sannella, and A. Tarlecki.

[BMGL00] E. Bretagne, A. El Marouani, P. Girard, and J.-L. Lanet. Pacap purse and loyalty specification. Technical Report Technical Report V 0.4, Gemplus, 2000. `http://www.gemplus.com/smart/r_d/publications/case-study/`.

[BRL03] L. Burdy, A Requet, and J.-L. Lanet. Java applet correctness: a developer-oriented approach. In K. Araki, S. Gnesi, and D. Mandrioli, editors, *FME'2003: Formal Methods*. Springer LNCS 2805, 2003.

[BRS+00] M. Balser, W. Reif, G. Schellhorn, K. Stenzel, and A. Thums. Formal system development with KIV. In T. Maibaum, editor, *Fundamental Approaches to Software Engineering*, number 1783 in LNCS, pages 363–366. Springer-Verlag, 2000.

[BS99] E. Börger and W. Schulte. A Programmer Friendly Modular Definition of the Semantics of Java. In J. Alves-Foss, editor, *Formal Syntax and Semantics of Java*. Springer LNCS 1523, 1999.

[Che00] Zhiqun Chen. *Java Card Technology for Smart Cards: Architecture and Programmer's Guide*. Addison-Wesley, 2000.

[CKRW99] P. Cenciarelli, A. Knapp, B. Reus, and M. Wirsing. An Event-Based Structural Operational Semantics of Multi-Threaded Java. In J. Alves-Foss, editor, *Formal Syntax and Semantics of Java*. Springer LNCS 1523, 1999.

[CoF04] CoFI (The Common Framework Initiative). CASL *Reference Manual*. LNCS 2960 (IFIP Series). Springer, 2004.

[Coo89] William Cook. A proposal for making eiffel type-safe. In *Proceedings European Conference on Object-Oriented Programming*. Cambridge University Press, 1989.

[Coq] Coq home page. `http://coq.inria.fr/`.

[DE99] S. Drossopoulou and S. Eisenbach. Describing the Semantics of Java and Proving Type Soundness. In J. Alves-Foss, editor, *Formal Syntax and Semantics of Java*. Springer LNCS 1523, 1999.

[FRSS95] T. Fuchß, W. Reif, G. Schellhorn, and K. Stenzel. Three Selected Case Studies in Verification. In M. Broy and S. Jähnichen, editors, *KORSO: Methods, Languages, and Tools for the Construction of Correct Software – Final Report*, LNCS 1009. Springer, 1995.

[GJS96] J. Gosling, B. Joy, and G. Steele. *The Java Language Specification, Edition 1.0*. Addison-Wesley, 1996.

[Gle03] Sabine Glesner. Asms versus natural semantics: A comparison with new insights. In E. Börger, A. Gargantini, and E. Riccobene, editors, *ASM 2003*. Springer LNCS 2589, 2003.

[Gra04] Holger Grandy. Beweisbare Sicherheit mobiler Java-Anwendungen. Diplomarbeit, Universität Augsburg, 2004. (in German).

[Gur95] M. Gurevich. Evolving algebras 1993: Lipari guide. In E. Börger, editor, *Specification and Validation Methods*, pages 9 – 36. Oxford University Press, 1995.

[Har79] D. Harel. *First Order Dynamic Logic*. LNCS 68. Springer, Berlin, 1979.

[HBB+05] D. Haneberg, S. Bäumler, M. Balser, H. Grandy, F. Ortmeier, W. Reif, G. Schellhorn, J. Schmitt, and K. Stenzel. The User Interface of the KIV Verification System — A System Description. In *Proocedings of UITP 05*, 2005.

[HHRS86] R. Hähnle, M. Heisel, W. Reif, and W. Stephan. An Interactive Verification System Based on Dynamic Logic. In J. Siekmann, editor, *8th International Conference on Automated Deduction. Proceedings*. Springer LNCS 230, 1986.

[HKT00] D Harel, D. Kozen, and J. Tiuryn. *Dynamic Logic*. MIT Press, 2000.

[HM01] P. Hartel and L. Moreau. Formalizing the safety of Java, the Java virtual machine, and Java card. *ACM Computing Surveys (CSUR)*, 33(4), December 2001.

[HM05] R. Hähnle and W. Mostowski. Verification of safety properties in the presence of transactions. In G. Barthe and M. Huisman, editors, *Construction and Analysis of Safe, Secure and Interoperable Smart devices (CASSIS'04)*. Springer LNCS 3362, 2005.

[HRS89] M. Heisel, W. Reif, and W. Stephan. A Dynamic Logic for Program Verification. In A. Meyer and M. Taitslin, editors, *Logical Foundations of Computer Science*, LNCS 363, pages 134–145, Berlin, 1989. Logic at Botik, Pereslavl-Zalessky, Russia, Springer.

[HRS91] M. Heisel, W. Reif, and W. Stephan. Formal Software Development in the KIV System. In R. McCartney and M. Lowry, editors, *Automating Software Design*. AAAI press, 1991.

[HRS02] D. Haneberg, W. Reif, and K. Stenzel. A Method for Secure Smartcard Applications. In H. Kirchner and C. Ringeissen, editors, *Algebraic Methodology and Software Technology, Proceedings AMAST 2002*. Springer LNCS 2422, 2002.

[Hui01] M. Huisman. *Reasoning about JAVA programs in higher order logic with PVS and Isabelle*. PhD thesis, University of Nijmegen, IPA dissertation series, 2001-03, 2001.

[Isa] Isabelle home page. `http://isabelle.in.tum.de/`.

[Jav00] *Java Card 2.1.1 Specification*, 2000. http://java.sun.com/products/javacard/.

[JML] JML home page. `http://www.jmlspecs.org/`.

[JMR04] Bart Jacobs, Claude Marche, and Nicole Rauch. Formal verification of a commercial smart card applet with multiple tools. In C. Rattray, S. Maharaj, and C. Shankland, editors, *Algebraic Methodology and Software Technology (AMAST) 2004, Proceedings*, Stirling Scotland, July 2004. Springer LNCS 3116.

[JP03] B. Jacobs and E. Poll. Java program verification at nijmegen: Developments and perspective. Technical Report NIII-R0318, University of Nijmegen, 2003.

[JSGB00] Bill Joy, Guy Steele, James Gosling, and Gilad Bracha. *The Java (tm) Language Specification, Second Edition*. Addison-Wesley, 2000.

[Kah87] G. Kahn. Natural semantics. In *Proceedings STACS'87*. Springer LNCS 247, 1987.

[KeY] KeY project homepage. `http://www.key-project.org`.

[Knu98] D. E. Knuth. *The Art of Computer Programming (Seminumerical Algorithms)*. Addison-Wesley, third edition edition, 1998.

[Kra] Krakatoa home page. `http://krakatoa.lri.fr/`.

[LEW96] J. Loeckx, H.D. Ehrich, and M. Wolf. *Specification of Abstract Data Types*. Wiley-Teubner, 1996.

[LLP+00] G.T. Leavens, K.R.M. Leino, E. Poll, C. Ruby, and B. Jacobs. Jml: notations and tools supporting detailed design in java. Technical Report TR #00-15, Department of Computer Science, Iowa State University, 2000.

[LM04] D. Larsson and W. Mostowski. Specifying Java Card API in OCL. In P. Schmitt, editor, *OLC 2.0 Workshop at UML 2003*. volume 102C of ENTCS, Elsevier, 2004.

[MN03] Farhad Mehta and Tobias Nipkow. Proving pointer programs in higher-order logic. In F. Baader, editor, *Automated Deduction — CADE-19*, volume 2741 of *LNCS*, pages 121–135. Springer, 2003.

[Mos91] Peter D. Mosses. A practical introduction to denotational semantics. In *Formal Description of Programming Concepts*, IFIP State-of-the-Art Reports. Springer, 1991.

[Mos02] Peter D. Mosses. Pragmatics of modular sos. In H. Kirchner and C. Ringeissen, editors, *Proceedings Algebraic Methodology and Software Technology, AMAST 2002*. Springer LNCS 2422, 2002.

[Mos05] Wojciech Mostowski. *Formal Development of Safe and Secure Java Card Applets*. PhD thesis, Chalmers University, Göteborg, Sweden, 2005.

[MPH99] P. Müller and A. Poetzsch-Heffter. Modular specification and verification techniques for object-oriented software components. In G. T. Leavens and M. Sitaraman, editors, *Foundations of Component-Based Systems*. Cambridge University Press, 1999.

[MPH00] J. Meyer and A. Poetzsch-Heffter. An architecture for interactive program provers. In S. Graf and M. Schwartzbach, editors, *Tools and Algorithms for the Construction and Analysis of Software (TACAS)*. Springer LNCS 1785, 2000.

[MPMU04] C. Marche, C. Paulin-Mohring, and X. Urbain. The Krakatoa tool for certification of Java/Javacard programs annotated in jml. *Journal of Logic and Algebraic Programming*, 58(1-2), 2004.

[Mül01] P. Müller. *Modular Specification and Verification of Object-Oriented Programs*. PhD thesis, FernUniversität Hagen, 2001.

[Nip02a] Tobias Nipkow. Hoare logics for recursive procedures and unbounded nondeterminism. In J. Bradfield, editor, *Computer Science Logic (CSL 2002)*, volume 2471 of *LNCS*, pages 103–119. Springer, 2002.

[Nip02b] Tobias Nipkow. Hoare logics in Isabelle/HOL. In H. Schwichtenberg and R. Steinbrüggen, editors, *Proof and System-Reliability*, pages 341–367. Kluwer, 2002.

[Nip03] Tobias Nipkow. Jinja: Towards a comprehensive formal semantics for a Java-like language. In H. Schwichtenberg and K. Spies, editors, *Proc. Marktobderdorf Summer School 2003*. IOS Press, 2003.

[NN98] Hanne Riis Nielson and Flemming Nielson. *Semantics with Applications: A Formal Introduction*. John Wiley & Sons, 1998. `http://www.daimi.au.dk/~hrn`.

[NPW03] Tobias Nipkow, Lawrence C. Paulson, and Markus Wenzel. *Tutorial on Isabelle/HOL*. Springer LNCS 2283, 2003. available at `http://isabelle.in.tum.de/`.

[NvO98] T. Nipkow and D. von Oheimb. Java light is Type-Safe – Definitely. In *25th ACM Symposium on Principles of Programming Languages*. ACM, 1998.

[ON02] David von Oheimb and Tobias Nipkow. Hoare logic for NanoJava: Auxiliary variables, side effects and virtual methods revisited. In Lars-Henrik Eriksson and Peter Alexander Lindsay, editors, *Formal Methods – Getting IT Right (FME'02)*, volume 2391 of *LNCS*, pages 89–105. Springer, 2002.

[ORS+02] F. Ortmeier, W. Reif, G. Schellhorn, A. Thums, B. Hering, and H. Trappschuh. Safety analysis of the height control system for the Elbtunnel. In *SafeComp 2002*, pages 296 – 308, Catania, Italy, 2002. Springer LNCS 2434.

[Pau94a] L. C. Paulson. *Isabelle: A Generic Theorem Prover*. LNCS 828. Springer, 1994.

[Pau94b] Lawrence C. Paulson. A fixedpoint approach to implementing (co)inductive definitions. In Alan Bundy, editor, *Automated Deduction — CADE-12*, LNAI 814, pages 148–161. Springer, 1994. 12th international conference.

[PHM99] A. Poetzsch-Heffter and P. Müller. A programming logic for sequential Java. In S. D. Swierstra, editor, *European Symposium on Programming (ESOP)*. Springer LNCS 1576, 1999.

[Plo81] Gordon D. Plotkin. A structural approach to operational semantics. Technical Report DAIMI FN-19, Aarhus University, 1981.

[Plo83] Gordon D. Plotkin. An operational semantics for CSP. In Dines Bjørner, editor, *Proceedings IFIP TC2-Working Conference: Formal Description of Programming Concepts II*. North-Holland, 19983.

[PvdBJ00a] E. Poll, J. van den Berg, and B. Jacobs. Specification of the Java Card API in JML. In J. Domingo-Ferrer, D. Chan, and A. Watson, editors, *Fourth Smart Card Research and Advanced Application Conference (CARDIS'2000)*. Kluwer Academic Publishers, 2000.

[PvdBJ00b] E. Poll, J. van den Berg, and B. Jacobs. Specification of the javacard api in jml. Technical Report CSI-R0005, Computing Science Institute, University of Nijmegen, 2000.

[Rei91] W. Reif. *Korrektheit von Spezifikationen und generischen Moduln*. PhD thesis, Universität Karlsruhe, Germany, 1991. (in German).

[Rei92a] W. Reif. Correctness of Full First-Order Specifications. In *4th Conference on Software Engineering and Knowledge Engineering. Proceedings*. Capri, Italy, IEEE press, 1992.

[Rei92b] W. Reif. Correctness of Generic Modules. In Nerode and Taitslin, editors, *Symposium on Logical Foundations of Computer Science*, LNCS 620, Berlin, 1992. Logic at Tver, Tver, Russia, Springer.

[Rei93] W. Reif. An Approach to Parameterized First-Order Specifications: Semantics, Correctness, Parameter Passing. In D. Bjørner, M. Broy, and I. Pottosin, editors, *Conference on Formal Methods in Programming and Their Applications*. Springer LNCS 735, 1993.

[Rei95] W. Reif. The KIV-approach to Software Verification. In M. Broy and S. Jähnichen, editors, *KORSO: Methods, Languages, and Tools for the Construction of Correct Software – Final Report*, LNCS 1009. Springer-Verlag, Berlin, 1995.

[RS97] W. Reif and G. Schellhorn. Theorem Proving in Large Theories. In *CADE 97 Workshop on Automated Theorem Proving in Software Engineering*, 1997.

[RSSB98] W. Reif, G. Schellhorn, K. Stenzel, and M. Balser. Structured specifications and interactive proofs with KIV. In W. Bibel and P. Schmitt, editors, *Automated Deduction—A Basis for Applications*, pages 13 – 39. Kluwer Academic Publishers, Dordrecht, 1998.

[SA98] G. Schellhorn and W. Ahrendt. The WAM Case Study: Verifying Compiler Correctness for Prolog with KIV. In W. Bibel and P. Schmitt, editors, *Automated Deduction—A Basis for Applications*, pages 165 – 194. Kluwer Academic Publishers, Dordrecht, 1998.

[SB94] G. Schellhorn and A. Burandt. Specification and Verification of Distributed Technical Systems with Central Control. In C. Lewerentz and T. Lindner, editors, *Formal Development of Reactive Systems*. Springer LNCS 891, 1994.

[Sch99] Gerhard Schellhorn. *Verification of Abstract State Machines*. PhD thesis, Universität Ulm, Fakultät für Informatik, 1999. (available at `http://www.informatik.uni-augsburg.de/swt/Publications.htm`).

[Sch01] G. Schellhorn. Verification of ASM Refinements Using Generalized Forward Simulation. *Journal of Universal Computer Science (J.UCS)*, 7(11):952–979, 2001. available at `http://hyperg.iicm.tu-graz.ac.at/jucs/`.

[Sch03] Norbert Schirmer. Java Definite Assignment in in Isabelle/HOL. In Susan Eisenbach, Gary T. Leavens, Peter Müller, Arnd Poetzsch-Heffter, and Erik Poll, editors, *Formal Techniques for Java-like Programs 2003 (Proceedings)*. Chair of Software Engineering, ETH Zürich, 2003. Technical Report 108.

[Sch05] Norbert Schirmer. A verification environment for sequential imperative programs in Isabelle/HOL. In F. Baader and A. Voronkov, editors, *Logic for Programming, Artificial Intelligence, and Reasoning*. Springer LNCS, 2005.

[SSB01] R.F. Stärk, J. Schmid, and E. Börger. *Java and the Java Virtual Machine: Definition, Verification, Validation*. Springer, 2001.

[Ste04] K. Stenzel. A formally verified calculus for full Java Card. In C. Rattray, S. Maharaj, and C. Shankland, editors, *Algebraic Methodology and Software Technology (AMAST) 2004, Proceedings*, Stirling Scotland, July 2004. Springer LNCS 3116.

[Sym99] D. Syme. Proving Java Type Soundness. In J. Alves-Foss, editor, *Formal Syntax and Semantics of Java*. Springer LNCS 1523, 1999.

[Sza69] M. E. Szabo, editor. *The Collected Papers of Gerhard Gentzen*. North-Holland, 1969.

[TS03] A. Thums and G. Schellhorn. Model checking FTA. In K. Araki, S. Gnesi, and D. Mandrioli, editors, *FME 2003: Formal Methods*, LNCS 2805, pages 739–757. Springer-Verlag, 2003.

[TSOW04] A. Thums, G. Schellhorn, F. Ortmeier, and W.Reif. Interactive verification of statecharts. In H. Ehrig, editor, *Integration of Software Specification Techniques for Applications in Engineering*, pages 355 – 373. Springer LNCS 3147, 2004.

[vdBJ01] J. van den Berg and B. Jacobs. The loop compiler for java and jml. In T. Margaria and W. Yi, editors, *Tools and Algorithms for the Construction and Analysis of Software (TACAS)*. Springer LNCS 2031, 2001.

[Ver] VerifiCard project homepage. `http://www.verificard.org`.

[vO00] David von Oheimb. Axiomatic semantics for Java$^{\ell ight}$ in Isabelle/HOL. In S. Drossopoulou, S. Eisenbach, B. Jacobs, G. T. Leavens, P. Müller, and A. Poetzsch-Heffter, editors, *Formal Techniques for Java Programs*. Technical Report 269, 5/2000, Fernuniversität Hagen, 2000.

[vO01] D. von Oheimb. *Analyzing Java in Isabelle/HOL: Formalization, Type Safety and Hoare Logic*. PhD thesis, Technische Universität München, 2001.

[vON99] D. von Oheimb and T. Nipkow. Machine-checking the Java Specification: Proving Type-Safety. In J. Alves-Foss, editor, *Formal Syntax and Semantics of Java*. Springer LNCS 1523, 1999.

[Wir90] M. Wirsing. *Algebraic Specification*, volume B of *Handbook of Theoretical Computer Science*, chapter 13, pages 675 – 788. Elsevier, Oxford, 1990.

Index

.intval, 23
&&
    proof rule, 76
    semantics, 33
||
    semantics, 33, 76

abstract syntax tree, 13
    annotated, 13
addarray, 40, 41
addarrays, 41
addclass, 26, 55
addobj, 26
algebraic specifications, 23
allsupers, 32
antecedent, 63
array access
    proof rule, 78
    semantics, 35
array assignment
    proof rule, 79
    semantics, 37
array creation
    proof rule, 81
    semantics, 40
array initializer
    proof rule, 82
    semantics, 41
ArrayAccess, 15
ArrayAssign, 15
asgcomp, 37, 52, 55, 128
    definition, 31
assignment compatible, 37
    definition, 31

basic expression, 64, 65
big-step semantics, 27
binary operator
    proof rule, 77
    semantics, 33
BinaryExpr, 14
block
    proof rule, 88
    semantics, 47
box, 63
break statement
    proof rule, 90
    semantics, 51

cast
    proof rule, 75
    semantics, 31
catches statement, 16
    proof rule, 91
    semantics, 53
class instance creation
    proof rule, 82
    semantics, 39
ClassCastException, 31
CompAssign, 15
compatible, 135
compound assignment
    proof rule, 81
    semantics, 38
conclusion, 70
CondBinExpr, 14
conditional binary operator
    proof rule, 76
    semantics, 33
conditional operator
    proof rule, 76
    semantics, 33
ConstrCall, 15
ConstrDecl, 17
constructor invocation
    proof rule, 83
    semantics, 41

decrement operator
    proof rule, 80
    semantics, 37
derivation, 70
deterministic semantics, 61
diamond, 63
DL, 63
do statement
    proof rule, 89
    semantics, 49
done, 24
dynamic logic, 63

empty target statement
proof rule, 93
semantics, 54
endfinally, 64
endfinally statement, 16
proof rule, 93
semantics, 53
endstatic statement, 16
proof rule, 94
semantics, 55
error, 24
evaluation mode, 26
exception binary operator
proof rule, 77
semantics, 34
explicit constructor invocation
proof rule, 83
semantics, 41
expression statement
semantics, 48
extend
definition, 129

field specification, 25
FieldAccess, 15
FieldAssign, 15
FieldDecl, 17
fieldspec, 15, 25
finally, 64
finally statement, 16
proof rule, 92
semantics, 53
first active use, 15, 39, 78, 79
static field access, 34
flattening, 64
for statement
proof rule, 89
semantics, 50
formulas
proof rule, 69
fulltc, 130

getConstr, 42

if statement
proof rule, 88
semantics, 48
incdec, 30
proof rule, 80
semantics, 37
IncDecExpr, 14
increment operator
proof rule, 80
semantics, 37
inductive defintions, 55
initdone, 26
initerror, 26
initstate, 24
initundone, 26
initval, 24
instance field access
proof rule, 78
semantics, 35
instance field assignment
proof rule, 79
semantics, 36
instance method invocation
proof rule, 84, 85
semantics, 45
instanceof
proof rule, 76
semantics, 32
intval, 23
invariant rule
for `for` loops, 103
for `while` loops, 102
is_basic_jexpr, 74
is_integervalue, 23
Isabelle, 55

Java store, 23
javavalue, 23
JLS, 13
jump
expression semantics, 30
jvmref, 26

labeled statement
proof rule, 89
semantics, 48
literal
may contain variables and terms, 14, 64
proof rule, 75
semantics, 30
local variable access
proof rule, 77
semantics, 34
local variable assignment
proof rule, 78
semantics, 35
local variable declaration
proof rule, 88
semantics, 48
LocVarAccess, 15
LocVarAssign, 15

may_catch
definition, 46

method invocation
    proof rule, 84
    semantics, 43
MethodDecl, 17
mode, 26

native
    constructor, 41
new array
    proof rule, 81
    semantics, 40
new class
    proof rule, 82
    semantics, 39
new statements, 15
noval, 24

okval, 70, 110, 117, 121
okvals, 110, 117
overloading, 23

postfix operator
    proof rule, 80
    semantics, 37
predefined classes, 27
prefix operator
    proof rule, 80
    semantics, 37
premise, 70
primCast, 14
primitive cast
    semantics, 31
primtc, 130

refCast, 14
reference cast
    proof rule, 75
    semantics, 31
refkey, 25
return statement
    proof rule, 90, 91
    semantics, 51

sem, semantical relation, 29
sequent calculus, 63
SFieldAccess, 15
SFieldAssign, 15
small-step semantics, 27
sound, 70
statement lists
    semantics, 47
static
    static field access, 34
static field access
    proof rule, 77
    semantics, 34
static field assignment
    proof rule, 79
    semantics, 36
static method invocation
    proof rule, 84
    semantics, 43
static statement, 16
    proof rule, 93
    semantics, 54
StatInit, 17
store, 23
storekey, 25
strongest postcondition, 64
subtype, 132
    definition, 32
succedent, 63
switch statement
    proof rule, 90
    semantics, 50
symbolic execution, 64

target, 16
target statement
    proof rule, 93
    semantics, 54
targetExpr, 16
throw statement
    proof rule, 91
    semantics, 52
try statement
    proof rule, 91
    semantics, 52
turnstyle, 63
typeval, 24

unary operations, 14, 30
undone, 24

variable mapping, 27

weakest precondition, 64
while statement
    proof rule, 89
    semantics, 49