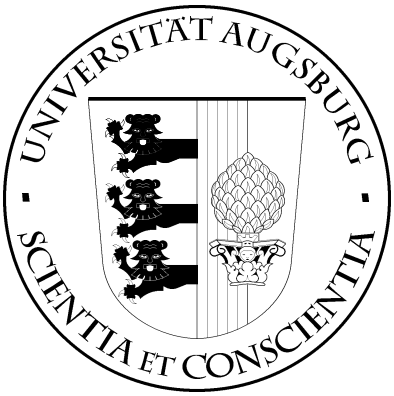
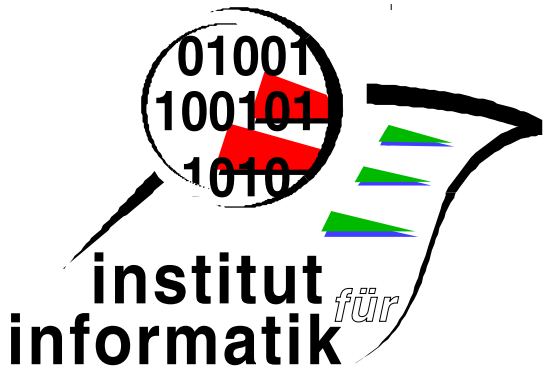
Universität Augsburg



Verification of JavaCard Programs

Kurt Stenzel

Report 2001-5 April 2001



Institut für Informatik
D-86135 Augsburg

Contents

Chapter 1

Introduction

1.1 Overview

This report is concerned with the formal verification of JavaCard [Jav00] programs, or sequentiel Java [GJS96] programs. A calculus in dynamic logic is presented. This calculus is implemented in KIV[1] [BRS+00][RSSB98], and ready for use. It is the first implemented proof system for JavaCard. KIV parses the original JavaCard (or Java) program, resolves names and types in the same manner as a normal Java compiler, and produces an annotated abstract syntax tree that is the input for the verification. All sequential Java statements (i.e. all Java statements except `synchronized`), and all Java expressions are supported. Not supported are threads, strings, floats, and Java programs in different files. Exceptions, breaks, static initialization, objects, dynamic method lookup, and arrays are supported.

This report is intended as a reference manual for the calculus. The abstract syntax of Java programs, the proof rules, and the underlying algebraic specifications for the object store and the primitive data types, and a formal semantic is described in detail. The report does not provide an introduction to Java or a tutorial about how to prove properties of Java programs.

JavaCard is a subset of Java for programming smart cards. Since a smart card has very limited memory resources (and reduced computational capacities), JavaCard omits those language features that make the JVM big and complicated. Most notably this applies to garbage collection and threads. Typical smart card processors are 16 bit processors without floating point support. This means that integers, longs, floats and doubles are not supported. All arithmetic has to be done with bytes and shorts. Strings and characters are also not supported. All other language features are supported, for example exception handling, inheritance, static initialization etc. To summarize, it is reasonable to describe JavaCard as sequential Java without floats and strings. It goes without saying that the predefined classes and packages of JavaCard differ completely from those of Java or toolkits like JDK (in any version). However, predefined classes have (almost) no impact on the proof calculus, and this report is not concerned with them.

The rest of this chapter provides tables of cross references between the Java language specification and our proof rules and semantic rules. Chapter 2 describes the abstract syntax of Java programs. Chapter 3 presents the calculus, chapter 4 an example proof, chapter 5 several test programs that can serve as validations and also as challenges, chapter 6 the semantic, and chapter 7 the algebraic specifications.

Related work: A good starting point for a formal treatment of Java is [AF99] (other collections are [JLMPH99] and [DEJ+00]). It contain e.g. a formal semantic with ASMs for Java with threads by Börger and Schulte [BS99], and an I/O semantics for a subset of Java by Oheimb and Nipkow [vON99] that is formalized in Isabelle/HOL. [vO00] presents a Hoare logic for a 'nearly full subset of sequential Java'. Jacobs et. al. [JvdBH+98] give an executable semantic that is formalized in PVS (again for a Java subset); [HJ00] is an extension and presents again a Hoare logic. Still

[1] http://www.Informatik.Uni-Augsburg.DE/swt/fmg/

another Hoare logic for a Java subset give Poetzsch-Heffter and Müller [PHM99] though it doesn't seem to be implemented or formalized in a proof system. Beckert [Bec00] presents a concept for a dynamic logic for JavaCard. In our opionion this concept cannot work, but the paper is very vague anyway.

1.2 Where to find it

The basis for every formal semantics or calculus for Java is *The Java Language Specification* [GJS96] by Gosling, Joy, and Steele. Chapter 14 deals with statements, chapter 15 with expressions. The following tables cross-reference between the language specification, and the sections in this document where the single expressions and statements are described.

1.2.1 Java Expressions

name	JLS	our name	semantics	calculus
array initializer	10.6	ArrayInit	6.1.6, p. 63	3.2.2, p. 11
literal	15.7.1	LiteralExpr	6.1.3, p. 58	3.2.1, p. 11
this	15.7.2	LocVarAccess	6.1.4, p. 59	3.2.4, p. 12
new class	15.8	NewExpr	6.1.6, p. 61	3.2.5, p. 12
new array	15.9	NewArray	6.1.6, p. 62	3.2.6, p. 12
field access	15.10	FieldAccess	6.1.4, p. 60	3.2.7, p. 13
field access	15.10	SFieldAccess	6.1.4, p. 60	3.2.8, p. 14
method invocation	15.11	MethodCall	6.1.6, p. 64	3.2.9, p. 14
method invocation	15.11	MethodCall	6.1.6, p. 63	3.2.10, p. 15
method invocation	15.11	ConstrCall	6.1.6, p. 63	3.2.11, p. 15
array access	15.12	ArrayAccess	6.1.4, p. 60	3.2.12, p. 15
local variable (name)	15.13.1	LocVarAccess	6.1.4, p. 59	3.2.4, p. 12
postfix increment	15.13.2	IncDecExpr	6.1.5, p. 61	3.2.13, p. 16
postfix decrement	15.13.3	IncDecExpr	6.1.5, p. 61	3.2.13, p. 16
prefix increment	15.14.1	IncDecExpr	6.1.5, p. 61	3.2.13, p. 16
prefix decrement	15.14.2	IncDecExpr	6.1.5, p. 61	3.2.13, p. 16
unary plus	15.14.3	UnaryExpr	6.1.3, p. 58	3.2.14, p. 16
unary minus	15.14.4	UnaryExpr	6.1.3, p. 58	3.2.14, p. 16
bitwise complement	15.14.5	UnaryExpr	6.1.3, p. 58	3.2.14, p. 16
logical complement	15.14.6	UnaryExpr	6.1.3, p. 58	3.2.14, p. 16
cast	15.15	Cast	6.1.3, p. 58	3.2.15, p. 16
*	15.16.1	binary op	6.1.3, p. 59	3.2.17, p. 17
/	15.16.2	ExBin op	6.1.3, p. 59	3.2.18, p. 17
%	15.16.3	ExBin op	6.1.3, p. 59	3.2.18, p. 17
string concatenation	15.17.1	–	–	–
+, -	15.17.2	BinaryExpr	6.1.3, p. 59	3.2.17, p. 17
shift operators	15.17	BinaryExpr	6.1.3, p. 59	3.2.17, p. 17
numerical comparison	15.19.1	BinaryExpr	6.1.3, p. 59	3.2.17, p. 17
instanceof	15.19.2	InstanceExpr	6.1.3, p. 59	3.2.16, p. 17
equality	15.20	BinaryExpr	6.1.3, p. 59	3.2.17, p. 17
integer bitwise	15.21.1	BinaryExpr	6.1.3, p. 59	3.2.17, p. 17
boolean logical	15.21.2	BinaryExpr	6.1.3, p. 59	3.2.17, p. 17
conditional and	15.22	CondBinExpr	6.1.3, p. 59	3.2.19, p. 17
conditional or	15.23	CondBinExpr	6.1.3, p. 59	3.2.19, p. 17
conditional	15.24	CondExpr	6.1.3, p. 59	3.2.20, p. 18
simple assignment	15.25.1	LocVarAssign	6.1.5, p. 60	3.2.21, p. 18
simple assignment	15.25.1	SFieldAssign	6.1.5, p. 60	3.2.22, p. 18

name	JLS	our name	semantics	calculus
simple assignment	15.25.1	FieldAssign	6.1.5, p. 60	3.2.23, p. 18
simple assignment	15.25.1	ArrayAssign	6.1.5, p. 61	3.2.24, p. 18
compound assignment	15.25.2	CompAssign	6.1.5, p. 61	3.2.25, p. 19

1.2.2 Java Statements

name	JLS	our name	semantics	calculus
static	12.4	static	6.2.2, p. 68	3.3.1, p. 19
static	12.4	endstatic	6.2.2, p. 68	3.3.2, p. 19
blocks	14.2	block	6.2.1, p. 64	3.3.3, p. 19
locVarDecl	14.3	locVarDecl	6.2.1, p. 65	3.3.3, p. 19
empty stmt	14.5	block	6.2.1, p. 64	3.3.3, p. 19
label	14.6	label	6.2.1, p. 65	3.3.4, p. 20
expression	14.7	exprStm	6.2.1, p. 65	3.3.5, p. 20
if statement	14.8	if	6.2.1, p. 65	3.3.6, p. 20
switch	14.9	switch	6.2.1, p. 66	3.3.7, p. 20
while	14.10	while	6.2.1, p. 65	3.3.8, p. 21
do statement	14.11	do	6.2.1, p. 66	3.3.9, p. 21
for	14.12	for	6.2.1, p. 66	3.3.10, p. 21
break	14.13	break	6.2.1, p. 66	3.3.11, p. 21
continue	14.14	continue	6.2.1, p. 66	3.3.12, p. 21
return	14.15	return	6.2.1, p. 66	3.3.13, p. 21
return	14.15	returnExpr	6.2.1, p. 66	3.3.14, p. 22
return	14.15	target	6.2.2, p. 69	3.3.15, p. 22
return	14.15	targetExpr	6.2.2, p. 68	3.3.16, p. 22
throw	14.16	throw	6.2.1, p. 67	3.3.17, p. 22
synchronize	14.17	–	–	–
try	14.18	try	6.2.1, p. 67	3.3.18, p. 23
try	14.18	catches	6.2.2, p. 69	3.3.19, p. 23
try	14.18	finally	6.2.2, p. 69	3.3.20, p. 23
try	14.18	endfinally	6.2.2, p. 69	3.3.21, p. 23

1.2.3 Additional Rules

name	reference
simplify	3.4.3, p. 24
flatten	3.4.1, p. 24
literalize	3.4.2, p. 24
for induction	3.4.8, p. 25
throwIt	3.4.7, p. 25
execute	3.4.5, p. 25
contract	3.4.6, p. 25
split	3.4.4, p. 24

Chapter 2

Java Programs

2.1 Abstract Syntax

The calculus works on an annotated syntax tree that is obtained after parsing the source code and performing all compile time evaluations (and checks) as described in the language specification.

In the following subsections we present the abstract syntax for expressions, statements, and classes.

2.1.1 Expressions

Notation	**our name**	**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

2.1.2 Additional Types

Type	real type	description
literal	expr	an expression
type	expr	a java type
UnOp	string	`+`, `-`, `~`, `!`
IncDecOp	string	`++`, `--`
CondOp	string	`&&`, `||`
BinOp	string	`+`, `-`, `*`, `&`, `|`, `^`, `<<`, `>>`, `>>>`
ExBinOp	string	`/`, `%`
variable	expr	a variable
fieldspec	expr	a field specification (classname, fieldname, type)
AsgOp	string	`+=`, `-=`, `*=`, `&=`, `|=`, `^=`, `<<=`, `>>=`, `>>>=`
classname	expr	a class name
fieldname	expr	a field name
methodname	string	a method name
invMode		*nonVirtual*(c), *super*(c), *virtual*, *static*(c)
label	expr	a label name
mode	expr	*throw*(val), *break*(l), *return*, *return*(val)

2.1.3 Statements

Notation	our name	arguments
$\{\alpha_1 \dots \alpha_n\}$	Block	javaStm* → javaStm
$ty\ x = e;$	LocVarDecl	type × variable × javaExpr → javaStm
$e;$	Exprstatement	javaExpr → javaStm
if (e) α **else** β	If	javaExpr × javaStm × javaStm → javaStm
$l : \alpha$	Label	label × javaStm → javaStm
while (e) α	While	javaExpr × javaStm → javaStm
do α **while** (e)	Do	javaStm × javaExpr → javaStm
for$(e; e_1 \dots e_n)$ α	For	javaExpr × javaExpr* × javaStm → javaStm
switch (e) $sl_1 \dots sl_n$	Switch	javaExpr × switchLabel* → javaStm
$e_1, \dots, e_n : \alpha$	switchLabel	javaExpr* × javaStm → 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 × catch* × javaStm → javaStm
$ct_1 \dots ct_n$	Catches	catch* → javaStm
catch $c(x)\{\alpha\}$	catch	classname × variable × javaStm → catch
static(c)	Static	classname → javaStm
endstatic(c)	EndStatic	classname → javaStm
target(m)	Target	mode → javaStm
targetExpr(x)	TargetExpr	variable → javaStm
finally$\{\alpha\}$	Finally	javaStm → javaStm
endfinally(m)	EndFinally	mode → javaStm

2.1.4 Classes

javaProgram = TypeDecl*
TypeDecl = ClassDecl | InterfaceDecl
ClassDecl = modifier* × classname × classname* × classname* × MemberDecl*

InterfaceDecl = modifier* × classname × classname* × MemberDecl*
MemberDecl = StaticInit | FieldDecl | MethodDecl | ConstrDecl
StaticInit = javaStm
FieldDecl = modifier* × type × fieldspec
MethodDecl = modifier* × type × methodname × ParamDecl* × javaStm
ConstrDecl = modifier* × classname × ParamDecl* × javaStm
ParamDecl = type × variable
modifier = static, public, private etc.

2.2 Assumptions about the Java Program

We expect a correct Java program, i.e. one that compiles successfully with a normal Java compiler. In addition, we require some implicit assumptions to be made explicit, and have some special requirements ourself.

Every class has a constructor. We require that an implicit default constructor is explicitely defined (JLS 8.6.7).

Constructor begins with this or super. The body of a constructor (except for Object) must begin with either a **this** or **super** call.

In Java, a missing call is implicitly assumed to be `super();` (JLS 8.6.5).

Constructors have return. Every execution path of a constructor ends with the statement `return this;`.

In Java, a return in a constructor is optional, but may have no expression (JLS 8.6.5).

Modifiers for interface fields. All interface fields are implicitly public, final, static. We require these modifiers explicitly (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).

Fields have no initializations. Static fields with initializations are transformed into static fields without initializations, and the initializations are added in their textual order to the static initializer (JLS 12.4.2).

Instance fields with initializations are transformed into instance fields without initializations, and the initializations are added in their textual order to the body of all constructors that begin with a super call (directly after the super call) (JLS 12.5).

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.

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.

Chapter 3

The Calculus

3.1 Overview

3.1.1 The Object Store

We use an explicit *store* for all objects and arrays. This store is specified algebraically. (The specification can be found in a later chapter.) Objects and arrays do not occur explicitly in the store. Instead, an object is represented by a *reference* and its *fields* and *values*. This means that the store contains *keys* that are pairs of a reference and a field specification, written $r - fs$. If we look up a key in a store st, $st[r - fs]$, we obtain the value of the field fs of the object with reference r. $st[r - fs, val]$ updates the field with the new value val. Arrays are represented by a reference and their indices, i.e. an index is seen as a field. If r is an array reference, $st[r - i]$ returns the value at array index i.

jvmref is a special, predefined reference. We use this reference to store static fields and other needed information, e.g. the initialization state of classes, the fact that a jump (an 'abrupt transfer of control') occurs, and references to the class objects. For jumps we have the *mode*. The mode can be either *normal* (no jump, normal execution of statements and expressions), *return* or *return*(e), *break(l)*, or *throw(e)* (denoting a jump due to a `return`, `break`, or `throw`, respectively). $st[mode]$ returns the mode. A field specification is a triple with a classname (the class defining the field), its type, and its field name. For example, $jvmref - [c, int, x]$ denotes a static field named `x` of type `int` in class `c`. The initialization state for a class `c` can be found under $jvmref - [c, void, initstate]$. For every reference r a special field *type* exists, that records the type of the reference (for objects and arrays). Array references also have a *length* field. (Internally, the special fields have a name that is illegal in Java to avoid name conflicts.) Class- and fieldnames are no further structured. Method names do not occur in the store. For references we only know that there are infinitely many of them.

Several operations are defined on the store (i.e. specified algebraically), and used in the proof rules. They are described as they occur. Their formal specification can be found in a later chapter. Java contains a number of predefined operations on its primitive data types, for example multiplication $*$, or bitwise and & on integers, or casts from integer to short and byte. These primitive types and their operations are also specified algebraically. Because we are mainly interested in JavaCard programs, bytes and shorts are correctly specified, as well as all bitwise operations. However, we omitted floats (and doubles), longs, and strings. While byte and short are specified with their defined range (including all effects of conversions), we use arbitrary large integers for Java integers. The reason is that JavaCard does not support integers, so that integer overflows cannot occur. Of course, it is simple to add bounded integers to out specification, if it is useful.

3.1.2 Dynamic Logic

We use a dynamic logic (DL, [Har79]) for the verification of JavaCard programs. DL extends first order logic with two modal operators, a box [.]. and a diamond ⟨.⟩ .. The box (diamond) contains a program, afterwards follows again a DL formula. In our case, the box (diamond) contains a pair 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. Their definition is:

$$\mathcal{A}, v \models [st'/\alpha]\ \varphi \Leftrightarrow \forall\ w.v_{st}^{v(st')}[\![\alpha]\!]w \rightarrow \mathcal{A}, w_{st'}^{w(st)} \models \varphi$$

$$\mathcal{A}, v \models \langle st'/\alpha\rangle\ \varphi \Leftrightarrow \exists\ w.v_{st}^{v(st')}[\![\alpha]\!]w \wedge \mathcal{A}, w_{st'}^{w(st)} \models \varphi$$

If α does not terminate, no w with $[\![\alpha]\!]w$ exists, and for the box the precondition is trivially true, and $\mathcal{A}, v \models [st/\alpha]\ \varphi$ is true regardless of φ. For a diamond, $\mathcal{A}, v \models \langle st/\alpha\rangle\ \varphi$ is false for every φ, if α does not terminate. $[st/\alpha]\ \varphi \Leftrightarrow \neg\ \langle st/\alpha\rangle\ \neg\ \varphi$ holds. Note that JavaCard is deterministic; if w exists with $v[\![\alpha]\!]w$ it is unique. We include the store variable in the box (diamond), because we want to 'talk' about different stores that may have some kind of relation, e. g. $st \equiv st_0 \rightarrow (\langle st/x = m(y)\rangle\ x = 3 \leftrightarrow \langle st_0/x = m(y)\rangle\ x = 3)$ (here $\equiv$ is assumed to define some similarity relation on stores). This example also shows that dynamic logic allows to formulate equivalences between programs. In our semantic we use a reserved variable st for the store (and assume that the program does not contain a variable st). This explains why we do not begin with state v but with $v_{st}^{v(st')}$, i.e. with v where st is set to the value of st', and end with $w_{st'}^{w(st)}$ (φ may contain st', and we assume that it does not contain st).

The verification framework is a sequent calculus. $\varphi_1, ..., \varphi_m \vdash \psi_1, ..., \psi_n$ 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 also 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:

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

Since both sides contain DL formulas, both sides may contain Java programs. All proof rules that follow are formulated only for diamonds in the succedent, but hold also – except where indicated – for boxes and for the antecedent. In the sequel we will mostly talk about diamonds, but boxes are implicitly included. Hoare's calculus is a very special case of dynamic logic. A Hoare triple $\varphi\{\alpha\}\psi$ is written $\varphi \vdash [\alpha]\psi$ in a DL sequent.

3.1.3 Main Features of the Calculus

result of expressions. 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.

evaluation of expressions. Nested expressions are flattened by introducing auxiliary variables for 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. Our literals are not only constants, but may contain variables and algebraic terms.

program state. The store and the local variables contain the complete program state. The fact that a jump occurs is recorded in the store. The store contains references for objects and their fields. Objects do not occur explicitely.

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

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. This property is the key for inductive proofs.

3.2 Expressions

This section contains the proof rules for expressions. Every rule is written with an assignment to a local variable, $\langle st/x = e;\rangle\ \varphi$, but is also applicable for the expression itself, $\langle st/e;\rangle\ \varphi$. Every rule for expressions has a first premise for the jump case. If the current mode $st[mode]$ is a jump ($st[mode] \neq normal$) the expression is not evaluated, but skipped, i.e. discarded. Some expressions include premises for a *first active use*: class or array creation, static field access, static field assignment, and static method invocation. A first active use will create two goals: One goal for an erroneous class state (see JLS 12.4.2), the second goal for an uninitialized class state. The remaining goals deal with the normal execution of the expression.

3.2.1 Literal

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

l is the literal, x a variable, y a new variable, i.e. a variable that does not occur in φ at all, and not free in Γ and Δ. $\varphi[x/y]$ is the replacement of x with y in φ. The definitions of variables, free variables, replacement, and substitution can be found elsewhere. For $\langle st/l;\rangle\ \varphi$ the statement is simply discarded.

3.2.2 Array Initializer

For simplicity, we treat an array initializer as a normal expression, because a variable or field declaration becomes an assignment.

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

An array initialization cannot be a *first active use*, and it cannot cause an *ArrayStoreException*. The expression list *exprs* contains only literals or local variables. r, st_0 are new variables. ty is the element type of the array. $newref(r, st)$ is a predicate that is true if the reference r does not occur in 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 $r - -i$ for $0 \leq i < \#(exprs)$ are added with their correct values, and that a *type* and *length* field is added for r.

3.2.3 Expression List

There is no special rule for expression lists. They are handled by the simplification rules (see 3.4, p. 24).

3.2.4 Local Variable

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

z is the (local) variable, x a variable, y a new variable. For $\langle st/z; \rangle\ \varphi$ the statement is simply discarded.

3.2.5 Class instance creation

See JLS 12.5 and 15.8. What happens is:

1. The object is created and its fields are initialized with their default values. This includes the fields of the super classes.
2. The arguments are evaluated.
3. The constructor is called. We assume that every constructor begins with an explicit constructor invocation either of the current class or of the super class. After this invocation follow field initializers as assignments.
4. All predefined classes are assumed to be initialized and have an empty constructor.

$$\frac{\begin{array}{ll}1. & \Gamma, st[mode] \neq normal \vdash \varphi, \Delta \\ 2. & \Gamma, st[mode] = normal, initerror(c, st) \\ & \vdash \langle st/\texttt{throw new ClassDefNotFoundError()} \rangle\ \varphi, \Delta \\ 3. & \Gamma, st[mode] = normal, initundone(c, st), initdone(c, st_0), st_0 = addclass(c, sfields, st) \\ & \vdash \langle st_0/\texttt{static}(c) \rangle\ \langle st_0/\alpha \rangle\ \langle st_0/\texttt{endstatic}(c) \rangle\ \langle st_0/x = \texttt{new}\ c(e_1 ... e_n) \rangle\ \varphi[st/st_0], \Delta \\ 4. & \Gamma, st[mode] = normal, initdone(c, st), st_0 = addobj(r, c, ifields, st), newref(r, st) \\ & \vdash \langle st_0/x = r.c(e_1 ... e_n) \rangle\ \varphi[st/st_0], \Delta\end{array}}{\Gamma \vdash \langle st/x = \texttt{new}\ c(e_1 ... e_n) \rangle\ \varphi, \Delta}$$

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 deals with the possibility that the class state is *erroneous* (see JLS 12.4.2), premise 3 is for a first active use. Premise 4 deals with the case that the class is already initialized. The `new` expression is simply transformed into an explicit constructor call. **static**(c) is only used to initialize the super class of c. an ExceptionInInitializerError, and set the class state to *erroneous*. r is a new reference for the object ($newref(r, st)$), *addobj* adds the fields of the object to the store. Note: The JLS states explicitely that the object is created before the arguments $e_1, ..., e_n$ are evaluated. This makes a difference only if an OutOfMemory can occur. However, modern Java implementations often do not behave as the given example (JLS 15.8.2), and we assume an arbitrary large memory.

3.2.6 Array creation

We make a difference between one-dimensional and multi-dimensional arrays. First the version for one-dimensional arrays:

$$\frac{\begin{array}{ll}1. & \Gamma, st[mode] \neq normal \vdash \varphi, \Delta \\ 2. & \Gamma, st[mode] = normal, initerror(c, st) \\ & \quad \vdash \langle st/\texttt{throw new } \text{ClassDefNotFoundError}() \rangle\ \varphi, \Delta \\ 3. & \Gamma, st[mode] = normal, initundone(c, st), initdone(c, st_0), st_0 = addclass(c, sfields, st) \\ & \quad \vdash \langle st_0/\texttt{static}(c) \rangle\ \langle st_0/\alpha \rangle\ \langle st_0/\texttt{endstatic}(c) \rangle\ \langle st_0/x = \texttt{new } c[e] \rangle\ \varphi[st/st_0], \Delta \\ 4. & \Gamma, st[mode] = normal, initdone(c, st), e < 0 \\ & \quad \vdash \langle st/\texttt{throw new } \text{NegativeArraySizeException}(); \rangle\ \varphi, \Delta \\ 5. & \Gamma, st[mode] = normal, initdone(c, st), \neg\ e < 0, st_0 = addarray(r, c, e, st), newref(r, st) \\ & \quad \vdash \langle st_0/x = r; \rangle\ \varphi[st/st_0], \Delta \end{array}}{\Gamma \vdash \langle st/x = \texttt{new } c[e] \rangle\ \varphi, \Delta}$$

Here, c is a class type. If an array with a primitive type is created, premises 2. and 3. are omitted because we cannot have a first active use. e must be a basic expression. $addarray(r, c, e, st)$ creates the array of length e in the store and initializes the indices with the correct default values for the type c. r is a new reference.

Note: Every array type has its own class object that can be accessed with getClass. They should be present in the store.

The multi-dimensional version:

$$\frac{\begin{array}{ll}1. & \Gamma, st[mode] \neq normal \vdash \varphi, \Delta \\ 2. & \Gamma, st[mode] = normal, initerror(c, st) \\ & \quad \vdash \langle st/\texttt{throw new } \text{ClassDefNotFoundError}() \rangle\ \varphi, \Delta \\ 3. & \Gamma, st[mode] = normal, initundone(c, st), initdone(c, st_0), st_0 = addclass(c, sfields, st) \\ & \quad \vdash \langle st_0/\texttt{static}(c) \rangle\ \langle st_0/\alpha \rangle\ \langle st_0/\texttt{endstatic}(c) \rangle\ \langle st_0/x = \texttt{new } c[e_1]...[e_n][dims] \rangle\ \varphi[st/st_0], \Delta \\ 4. & \Gamma, st[mode] = normal, initdone(c, st), e_1 < 0 \vee ... \vee e_n < 0 \\ & \quad \vdash \langle st/\texttt{throw new } \text{NegativeArraySizeException}(); \rangle\ \varphi, \Delta \\ 5. & \Gamma, st[mode] = normal, initdone(c, st), \neg\ (e_1 < 0 \vee ... \vee e_n < 0), \\ & \quad st_0 = addarraymultlist(r, ty, st, refs, e_1 + \ldots + e_n, e_1 + \ldots + e_n, dims), newref(r, st), \\ & \quad is\_newref\_list(refs, st), \neg\ r \in refs, n{\rightarrow}i(\#refs) = countrefs(e_1 + \ldots + e_n) \\ & \quad \vdash \langle st_0/x = r; \rangle\ \varphi[st/st_0], \Delta \end{array}}{\Gamma \vdash \langle st/x = \texttt{new } c[e_1]...[e_n][dims] \rangle\ \varphi, \Delta}$$

We need the extra dimensions *dims* to determine the default value for the indices. A multi-dimensional array creates several new arrays and needs several new references *refs* (a list of references). This list of references is new (*is_newref_list(refs,st)*), and has the correct length ($n{\rightarrow}i(\#refs) = countrefs(e_1 + \ldots + e_n)$). Here, # is the length of the list, $n{\rightarrow}i$ converts a natural to an integer, and *countrefs* computes how many references will be needed for the subarrays. The function *addarraymultlist* itself is rather complex.

3.2.7 Instance field access

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

x must be a basic expression, z is a new variable. f is the field specification.

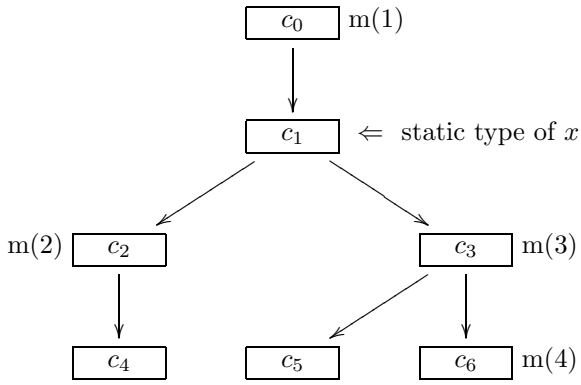
3.2.8 Static field access

$$\frac{\begin{array}{ll}1. & \Gamma, st[mode] \neq normal \vdash \varphi, \Delta \\ 2. & \Gamma, st[mode] = normal, initerror(c, st) \\ & \quad \vdash \langle st/\texttt{throw new ClassDefNotFoundError()}\rangle\ \varphi, \Delta \\ 3. & \Gamma, st[mode] = normal, initundone(c, st), initdone(c, st_0), st_0 = addclass(c, sfields, st) \\ & \quad \vdash \langle st_0/\texttt{static}(c)\rangle\ \langle st_0/\alpha\rangle\ \langle st_0/\texttt{endstatic}(c)\rangle\ \langle st_0/y = c.f;\rangle\ \varphi[st/st_0], \Delta \\ 4. & \Gamma, st[mode] = normal, initdone(c, st), z = st[jvmref - c.f] \vdash \varphi[y/z], \Delta \end{array}}{\Gamma \vdash \langle st/y = c.f;\rangle\ \varphi, \Delta}$$

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

3.2.9 Instance method invocation

The invoker e and arguments es must be basic expressions. The correct method body depends on the runtime type of e. The following picture gives an example:



Let $e.m(es)$ the method invocation, c_1 the static type of e. 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). At run time the runtime class R of e is c_1 or one of its subclasses (because of Java's type safety). 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 runtime classes we obtain: $\mathrm{R} = c_4 \vee c_2 \Rightarrow \mathrm{m}(2)$, $\mathrm{R} = c_5 \vee c_3 \Rightarrow \mathrm{R} = \mathrm{m}(3)$, $\mathrm{R} = c_6 \Rightarrow \mathrm{m}(4)$, $\mathrm{R} = c_1 \Rightarrow \mathrm{m}(1)$. The first premise is for jumps, the second for a null invoker, and the third states that R is indeed one of the possible seven classes. We need this latter premise even for correct java programs because R is stored in the store, and it is not possible to guarantee the 'correctness' of the store. The remaining four premises are for the possible method bodies:

$$\frac{\begin{array}{ll}1. & \Gamma, mode(st) \neq normal \vdash \varphi, \Delta \\ 2. & \Gamma, e = null, mode(st) = normal \vdash \langle st/\textbf{throw}\ new\ NPE()\rangle\ \varphi, \Delta \\ 3. & \Gamma, e \neq null, mode(st) = normal \vdash classOf(e, st) = c_0 \vee \ \ldots\ \vee classOf(e, st) = c_6 \\ 4. & \Gamma, e \neq null, mode(st) = normal, classOf(e, st) = c_4 \vee classOf(e, st) = c_2, \\ & this' = e, \underline{z} = es \vdash \langle st/\alpha_2'\rangle\ \langle st/\textbf{targetexpr}(x)\rangle\ \varphi, \Delta \\ 5. & \Gamma, e \neq null, mode(st) = normal, classOf(e, st) = c_5 \vee classOf(e, st) = c_3, \\ & this' = e, \underline{z} = es \vdash \langle st/\alpha_3'\rangle\ \langle st/\textbf{targetexpr}(x)\rangle\ \varphi, \Delta \\ 6. & \Gamma, e \neq null, mode(st) = normal, classOf(e, st) = c_6, \\ & this' = e, \underline{z} = es \vdash \langle st/\alpha_4'\rangle\ \langle st/\textbf{targetexpr}(x)\rangle\ \varphi, \Delta \\ 7. & \Gamma, e \neq null, mode(st) = normal, classOf(e, st) = c_4 \vee classOf(e, st) = c_1, \\ & this' = e, \underline{z} = es \vdash \langle st/\alpha_1'\rangle\ \langle st/\textbf{targetexpr}(x)\rangle\ \varphi, \Delta \end{array}}{\Gamma \vdash \langle st/x = e.m(es)\rangle\ \varphi, \Delta}$$

α_i is the corresponding method body for $m(i)$, α_i' is α_i with the formal parameters and **this** replaced by new variables $\underline{z}, this'$. Return statements in α_i are 'caught' by **targetexpr**(x),

and assigned to x. A method invocation without assignment (i.e. $\langle st/e.m(es);\rangle\ \varphi$) produces a `target`($return$) instead of `targetexpr`($x$). *classOf* looks up the class of the reference e in the store, $classOf(e, st) = (st[e - type]).class$.

Note: In general it is not possible to shift the assignment $x = \ldots$ into the method body (e.g. by **return** e; ⇒ **return** x = e;), because the **return** may be followed by finally clauses that can cause an exception. However, optimizations are possible.

3.2.10 Static method invocation

$$\frac{\begin{array}{ll}1. & \Gamma, st[mode] \neq normal \vdash \varphi, \Delta\\ 2. & \Gamma, st[mode] = normal, initerror(c, st)\\ & \quad \vdash \langle st/\texttt{throw new ClassDefNotFoundError()}\rangle\ \varphi, \Delta\\ 3. & \Gamma, st[mode] = normal, initundone(c, st), initdone(c, st_0), st_0 = addclass(c, sfields, st)\\ & \quad \vdash \langle st_0/\texttt{static}(c)\rangle\ \langle st_0/\alpha\rangle\ \langle st_0/\texttt{endstatic}(c)\rangle\ \langle st_0/x = c.m(es);\rangle\ \varphi[st/st_0], \Delta\\ 4. & \Gamma, st[mode] = normal, initdone(c, st), \underline{z} = es\\ & \quad \vdash \langle st/\alpha[\underline{y}/\underline{z}]\rangle\ \langle st/\texttt{targetexpr}(x)\rangle\ \varphi, \Delta\end{array}}{\Gamma \vdash \langle st/x = c.m(es);\rangle\ \varphi, \Delta}$$

α is the method body, $\underline{y}$ its formal parameters, $\underline{z}$ new variables. es must be basic expressions. A method invocation without assignment (i.e. $\langle st/c.m(exprs);\rangle\ \varphi$) produces a `target`($return$) instead of `targetexpr`($x$).

3.2.11 Explicit constructor invocation

$$\frac{\begin{array}{ll}1. & \Gamma, st[mode] \neq normal \vdash \varphi, \Delta\\ 2. & \Gamma, st[mode] = normal, \underline{z} = es, this' = r\\ & \quad \vdash \langle st/\alpha[y, this/z, this']\rangle\ \langle st/\texttt{targetexpr}(r)\rangle\ \varphi, \Delta\end{array}}{\Gamma \vdash \langle st/r.m(es);\rangle\ \varphi, \Delta}$$

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`(es) call in class c is transformed into this.c(es), a `super`(es) call is transformed into this.s(es) if s is the superclass of c.) This means that r is always a basic expression, a reference. m is the correct class name for the constructor call, α its body with formal parameters $\underline{y}$. es must be basic expressions, $\underline{y}$ and $this'$ are new variables. We assume that every execution path in the constructor ends with a `return this;` so that `targetexpr`(r) will correctly bind r to the object.

3.2.12 Array access

See JLS 15.12. Both arguments are evaluated, then the first argument is checked to be not null (otherwise *NullPointerException*), then the index is checked (otherwise *IndexOutOfBoundsException*).

$$\frac{\begin{array}{ll}1. & \Gamma, st[mode] \neq normal \vdash \varphi, \Delta\\ 2. & \Gamma, st[mode] = normal, x = \texttt{null}\\ & \quad \vdash \langle st/\texttt{throw new NullPointerException()}\rangle\ \varphi, \Delta\\ 3. & \Gamma, st[mode] = normal, x \neq \texttt{null}, i < 0 \vee i \geq st[x - \_length]\\ & \quad \vdash \langle st/\texttt{throw new IndexOutOfBoundsException()}\rangle\ \varphi, \Delta\\ 4. & \Gamma, st[mode] = normal, z = st[x - i] \vdash \varphi[y/z], \Delta\end{array}}{\Gamma \vdash \langle st/y = x[i];\rangle\ \varphi, \Delta}$$

x and i must be basic expressions, z is a new variable.

3.2.13 Inc/Decrement operations

We show the rule for post increment.

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

v must be a variable in the Java sense, i.e. either a local variable, a static field, an instance field with a basic expression as invoker, or an array access with both arguments basic expressions. Since all arguments are already evaluated there is no problem with multiple evaluations. For post decrement we get $\langle st/x = v; v = v - 1; \rangle\ \varphi$, for pre increment $\langle st/v = v + 1; x = v; \rangle\ \varphi$, for pre decrement $\langle st/v = v - 1; x = v; \rangle\ \varphi$.

3.2.14 Unary operations

There is no special rule for unary operations. They are handled by the simplification rules (see 3.4, p. 24).

3.2.15 Cast

Reference cast: JLS 15.15 and 5.5. 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 ok if the expression e is assignment compatible with ty. (JLS 5.5 states 'A cast conversion must check, at run time, that the class R is assignment compatible with the type T, using the algorithm specified in Σ5.2 but using the class R instead of the compile-time type S as specified there.' Σ5.2 describes assignment conversion.)

$$\frac{\begin{array}{ll}1. & \Gamma, st[mode] \neq normal \vdash \varphi, \Delta \\ 2. & \Gamma, st[mode] = normal, \neg\ asgcomp\_fma \vdash \langle st/\texttt{throw new ClassCastException();} \rangle\ \varphi, \Delta \\ 3. & \Gamma, st[mode] = normal, asgcomp\_fma \vdash \langle st/x = e; \rangle\ \varphi, \Delta\end{array}}{\Gamma \vdash \langle st/x = (ty)e; \rangle\ \varphi, \Delta}$$

asgcomp_fma is the formula that is true iff e is assignment compatible with ty for the given Java program. Note, that we do not know the the run time type of e explicitly (it is accessible as $st[e - \_type]$, but we cannot use this value when the rule is applied), but we can examine the static type of e.

1. e has a primitive static type ⇒ *asgcomp_fma* = true (otherwise a compile time error would have occurred).
2. e has a static type that is an array type with a primitive element type ⇒ *asgcomp_fma* = true
3. This means e has as static type a class (or interface) type or an array type with a class or interface type as element type.
4. $e = null$ is allowed. *asgcomp_fma* = e = null ∨ *sub_fma*. *sub_fma* depends on the run time type of e:
   (a) e has a class type ⇒ ty must be a class type, and the class of e must be a subclass of ty. Let $c_1 \ldots c_n$ be the subclasses of ty (including ty). Let $sub\_cs = st[e - \_type] = mkclasstype(c_1) \vee \ldots \vee st[e - \_type] = mkclasstype(c_n)$.
   (b) e has an array type ($is\_arraytype(st[e - \_type]) \wedge \ldots$). ty can be Object ($ty = mkclasstype(Object) \vee \ldots$), or the interface Cloneable ($ty = mkclasstype(Cloneable) \vee \ldots$, or an array type with the same dimension as e's type. In the latter case let T be ty's element type, and R e's element type ($st[e - \_type].jtclass$). If R is an interface, T

must Object or super interface of R (including R). If R is a class, R must be a subclass of T. The formulas for subclasses or super interfaces are constructed as described in the previous item.

Primitive cast: no special rule, handled by simplification rules.

3.2.16 Instanceof

$$\frac{\begin{array}{ll}1. & \Gamma, st[mode] \neq normal \vdash \varphi, \Delta \\ 2. & \Gamma, st[mode] = normal, \neg\ nonNullasgcomp\_fma \vdash \langle st/x = false; \rangle\ \varphi, \Delta \\ 3. & \Gamma, st[mode] = normal, nonNullasgcomp\_fma \vdash \langle st/x = true; \rangle\ \varphi, \Delta\end{array}}{\Gamma \vdash \langle st/x = e\ \texttt{instanceof}\ ty; \rangle\ \varphi, \Delta}$$

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.

3.2.17 Binary operations

There is no special rule for simple binary operations. They are handled by the simplification rules (see 3.4, p. 24).

3.2.18 Exception binary operation

/ and % throw an *ArithmeticException* if the divisor is 0.

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

e_1, e_2 must be basic Java expressions. We select their value with *expr* (possibly converting byte and short to integer), apply the corresponding algebraic operation on the values, and construct a literal with *mkliteral*. Of course, the algebraic integer division and modulo operations must be specified as required by Java.

3.2.19 Conditional binary operation

Conditional binary operations 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:

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

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

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.

3.2.20 Conditional expression

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

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

3.2.21 Local variable assignment

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

y must be a basic expression, z is a new variable. $\varphi[x/z]$ is the replacement of x by z in φ. Actually this rule is identical to the literal rule.

3.2.22 Static field assignment

$$\frac{\begin{array}{ll}1. & \Gamma, st[mode] \neq normal \vdash \varphi, \Delta \\ 2. & \Gamma, st[mode] = normal, initerror(c, st) \vdash \langle st/\texttt{throw new}\ \text{ClassDefNotFoundError}() \rangle\ \varphi, \Delta \\ 3. & \Gamma, st[mode] = normal, initundone(c, st), initdone(c, st_0), st_0 = addclass(c, sfields, st) \\ & \quad \vdash \langle st_0/\texttt{static}(c) \rangle\ \langle st_0/\alpha \rangle\ \langle st_0/\texttt{endstatic}(c) \rangle\ \langle st_0/c.f = y; \rangle\ \varphi[st/st_0], \Delta \\ 4. & \Gamma, st[mode] = normal, initdone(c, st), st_0 = st[jvmref - c.f][y] \vdash \varphi[st/st_0], \Delta\end{array}}{\Gamma \vdash \langle st/c.f = y; \rangle\ \varphi, \Delta}$$

y must be a literal or a local variable, st_0 is a new variable. c is the class of the static field, f the field name. $st[jvmref - c.f][y]$ modifies the store and sets the key $jvmref - c.f$ to the new value y.

3.2.23 Instance field assignment

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 (and nonsense). Both arguments are evaluated, then e is checked to be not null.

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

x, y must be basic expressions, st_0 is a new variable.

3.2.24 Array assignment

$$\frac{\begin{array}{ll}1. & \Gamma, st[mode] \neq normal \vdash \varphi, \Delta \\ 2. & \Gamma, st[mode] = normal, x = \texttt{null} \\ & \quad \vdash \langle st/\texttt{throw new NullPointerException}() \rangle\ \varphi, \Delta \\ 3. & \Gamma, st[mode] = normal, x \neq \texttt{null}, i < 0 \vee i \geq st[x - \_length] \\ & \quad \vdash \langle st/\texttt{throw new IndexOutOfBoundsException}() \rangle\ \varphi, \Delta \\ 4. & \Gamma, st[mode] = normal, x \neq \texttt{null}, \neg\ (i < 0 \vee i \geq st[x - \_length]), \\ & \quad \neg\ asgcomp\_fma \vdash \langle st/\texttt{throw new ArrayStoreException}() \rangle\ \varphi, \Delta \\ 5. & \Gamma, st[mode] = normal, x \neq \texttt{null}, \neg\ (i < 0 \vee i \geq st[x - \_length]), \\ & \quad asgcomp\_fma, st_0 = st[x - i][y] \vdash \varphi[st/st_0], \Delta\end{array}}{\Gamma \vdash \langle st/x[i] = y; \rangle\ \varphi, \Delta}$$

x, i, y must be either literals or local variables, st_0 is a new variable, the element type of x is a reference type. The rule looks surprisingly complex for a simple array assignment, but compare it to the description in the Java language specification (15.25.1).

3.2.25 Compound assignment

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

e_1, e_2 must be basic expressions.

3.3 Statements

3.3.1 Static Initializer (static)

$$\frac{\begin{array}{ll}1. & \Gamma, st[mode] \neq normal \vdash \varphi, \Delta \\ 2. & \Gamma, st[mode] = normal, initerror(s, st) \vdash \langle st/\texttt{throw new ClassDefNotFoundError()} \rangle\ \varphi, \Delta \\ 3. & \Gamma, st[mode] = normal, initundone(s, st), initdone(s, st_0), st_0 = addclass(s, sfields, st) \\ & \quad \vdash \langle st_0/static(s) \rangle\ \langle st_0/\alpha \rangle\ \langle st_0/endstatic(s) \rangle\ \varphi[st/st_0], \Delta \\ 4. & \Gamma, st[mode] = normal, initdone(s, st) \vdash \varphi, \Delta\end{array}}{\Gamma \vdash \langle st/static(c) \rangle\ \varphi, \Delta}$$

s is the superclass of c, α the static initializer for class s, and *sfields* the static fields of s. If $c =$ Object, the statement is simply discarded. `static` is only used to initialize the super classes of a class. It can cause a first active use of the super class, but has no other effect.

3.3.2 Static Initializer (endstatic)

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 error is re-thrown.

$$\frac{\begin{array}{ll}1. & \Gamma, is\_throw\_mode(st[mode]), leError\_fma, \\ & st_0 = st[jvmref - mkfs(c, void, initstate)][initerror] \vdash \varphi_{[}st/st_0], \Delta \\ 2. & \Gamma, is\_throw\_mode(st[mode]), \neg\ leError\_fma, \\ & st_0 = st[jvmref - mkfs(c, void, initstate)][initerror] \\ & \quad \vdash \langle st_0/\texttt{throw new ExceptionInInitializerError()} \rangle\ \varphi[st/st_0], \Delta \\ 3. & \Gamma, \neg\ is\_throw\_mode(st[mode]) \vdash \varphi, \Delta\end{array}}{\Gamma \vdash \langle st/endstatic(c) \rangle\ \varphi, \Delta}$$

leError_fma is true if the type of the thrown object $st[mode].type$ is a subclass of Error. This formula is constructed in the usual manner. Note that is not possible to throw an object of class Throwable in a static initializer. It must be of class Error, or Exception, or one of their subclasses. On the other hand the user can extend class Error and throw an Object of the new Error subclass. The JLS does not specify if an ExceptionInInitializerError contains a string or the old Throwable object. jdk stores the old throwable object, and the string is null. We simply omit the argument.

3.3.3 Blocks and local variable declarations

$$\frac{1. \quad \Gamma \vdash \langle st/\alpha'_1 \rangle\ ... \langle st/\alpha'_n \rangle\ \varphi, \Delta}{\Gamma \vdash \langle st/\{\alpha_1 ... \alpha_n\} \rangle\ \varphi, \Delta}$$

α_i are the toplevel statements of the block. If α_i is a local variable declaration $tyx = e$, the variable x is replaced by a new one y in $\alpha_{i+1}...\alpha_n$, and α_i becomes an assignment $y = e$. Note that this is not legal Java if e is an array initializer, but we treat it as a normal expression. The rule is always applicable, even in case of a jump.

Note: See JLS 4.5.3 (topic 7) for a special case concerning the switch statement.

3.3.4 Labeled statement

$$\frac{1.\quad \Gamma \vdash \langle st/\alpha\rangle\ \langle st/target(break(l))\rangle\ \varphi, \Delta}{\Gamma \vdash \langle st/l : \alpha\rangle\ \varphi, \Delta}$$

Remember that our Java programs have no `continue`'s. $target(break(\mathrm{l}))$ catches `break`'s occurring in α.

3.3.5 Expression statement

There is no single rule for an expression statement; rather one rule for every expression. See above.

3.3.6 If statement

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

b must be a basic expression.

3.3.7 The switch statement

$$\frac{\begin{array}{ll}1a. & \Gamma, st[mode] \neq normal \vdash \varphi, \Delta\\ 1b. & \Gamma, st[mode] = normal \vdash \bigwedge e_i \neq e_j, \Delta\\ 1. & \Gamma, st[mode] = normal, e = e_1 \vdash \langle st/\{\alpha_1...\alpha_m\ \alpha\ \alpha_{m+1}...\alpha_n\}\rangle\ \varphi, \Delta\\ & \vdots\\ m. & \Gamma, st[mode] = normal, e = e_m \vdash \langle st/\{\alpha_m\ \alpha\ \alpha_{m+1}...\alpha_n\}\rangle\ \varphi, \Delta\\ m+1. & \Gamma, st[mode] = normal, e = e_m \vdash \langle st/\{\alpha_{m+1}...\alpha_n\}\rangle\ \varphi, \Delta\\ & \vdots\\ n. & \Gamma, st[mode] = normal, e = e_m \vdash \langle st/\{\alpha_n\}\rangle\ \varphi, \Delta\\ n+1. & \Gamma, st[mode] = normal, e \neq e_1, ..., e \neq e_n \vdash \langle st/\{\alpha\ \alpha_{m+1}...\alpha_n\}\rangle\ \varphi, \Delta\end{array}}{\Gamma \vdash \langle st/\texttt{switch}\ e\ \{s_1...s_m\ \texttt{default} : \alpha\ s_{m+1}...s_n\}\rangle\ \varphi, \Delta}$$

- s_i is a switchLabelStatement, $s_i = \texttt{case}\ e_i\texttt{:}\ \alpha_i$
- Every switchLabelStatement must have one or more labels. The default case is a switchLabelStatement with no labels.
- In case of more than one label the precondition becomes a disjunction:

  $s_i = \texttt{case}\ e_{i1}\texttt{:}\ldots\ \texttt{case}\ e_{ik_i}\texttt{:}\alpha_i \Rightarrow e = e_{i1} \vee\ \ldots\ \vee e = e_{ik_i}$

  and the default case includes the negation of all expressions.
- If the switch statement has no default case the last goal becomes

  $\Gamma, st[mode] = normal, e \neq e_1, \ldots, e \neq e_n \vdash \varphi, \Delta$
- e must be a basic expression.

- The case expressions must be different. This is captured in premise 1a. because it is not clear that this can be checked syntactically.

Note that a switch statement 'falls through', and that the default case may appear anywhere in the list. See JLS 4.5.3 (topic 7) for a special case concerning local variable declarations.

3.3.8 While statement

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

Note that this rule allows to prove theorems with while loops by noetherian induction on some data structure. We would like to have some invariant rule, and induction on the number of iterations. (The latter is needed for a complete calculus.)

This rule is correct only if the body does not contain `continue` statements. e can be an arbitrary Java expression. It is not possible to flatten this expression.

3.3.9 Do statement

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

3.3.10 For statement

Our `for` statement contains no initialization, but only the termination test e, the updates es and the body.

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

for$(e, es)\ \alpha =$ **while**$(e)\{\alpha; es;\}$ should hold if no `continue` appears.

3.3.11 Break

$$\frac{\begin{array}{ll}1. & \Gamma, st[mode] \neq normal \vdash \varphi, \Delta \\ 2. & \Gamma, st[mode] = normal, st_0 = st[mode][break(l)] \vdash \varphi[st/st_0], \Delta\end{array}}{\Gamma \vdash \langle st/\texttt{break}\ l;\rangle\ \varphi, \Delta}$$

We can optimize this rule by omitting Java statements in φ that cannot catch a `break`.

3.3.12 Continue

Continue statements are transformed into break statements (with the introduction of additional labels). Therefore there is no rule for continue.

3.3.13 Empty return

$$\frac{\begin{array}{ll}1. & \Gamma, st[mode] \neq normal \vdash \varphi, \Delta \\ 2. & \Gamma, st[mode] = normal, st_0 = st[mode][return] \vdash \varphi[st/st_0], \Delta\end{array}}{\Gamma \vdash \langle st/\texttt{return};\rangle\ \varphi, \Delta}$$

We can optimize this rule by omitting Java statements in φ that cannot catch a `return`.

3.3.14 Return value

$$\frac{\begin{array}{ll}1. & \Gamma, st[mode] \neq normal \vdash \varphi, \Delta \\ 2. & \Gamma, st[mode] = normal, st_0 = st[mode][return(e)] \vdash \varphi[st/st_0], \Delta\end{array}}{\Gamma \vdash \langle st/\texttt{return}e; \rangle\ \varphi, \Delta}$$

e must be a basic expression. We store the value in the mode for a `targetexpr`. We can optimize this rule by omitting Java statements in φ that cannot catch a `return`.

3.3.15 Target

$$\frac{\begin{array}{ll}1. & \Gamma, st[mode] = mo, st_0 = st[mode][normal] \vdash \varphi[st/st_0], \Delta \\ 2. & \Gamma, st[mode] \neq mo \vdash \varphi, \Delta\end{array}}{\Gamma \vdash \langle st/\texttt{target}(mo) \rangle\ \varphi, \Delta}$$

mo is the mode to catch, either *break*(l) or *return*.

3.3.16 Target value

$$\frac{\begin{array}{ll}1. & \Gamma, is\_return(st[mode]), st_0 = st[mode][normal] \vdash \langle st_0/x = st[mode].val; \rangle\ \varphi[st/st_0], \Delta \\ 2. & \Gamma, \neg\ is\_return(st[mode]) \vdash \varphi, \Delta\end{array}}{\Gamma \vdash \langle st/\texttt{targetexpr}(x); \rangle\ \varphi, \Delta}$$

A `targetexpr` catches returns and assigns the returned value to the variable x.

3.3.17 Throw

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

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 (this case is missing in JLS 14.16).

Instead of setting the mode, the following optimization is possible. The rule discards the statements after the `throw` statement until a possible catcher for the throw is reached. A possible catcher is either a `finally` statement, a `catches` statement, or an `endstatic` statement. If the following statements do not contain a possible catcher all statements are discarded and the mode is set to throw as in the previous case.

We get four subcases:

1. Next possible catcher is a `finally`, $\alpha = \alpha'$ `finally` { β } γ

   `finally` always catches a throw. We apply the `finally` rule and record the mode in the `endfinally` statement.

$$\frac{\begin{array}{ll}1. & \Gamma, st?\_mode \neq normal \vdash \langle st/\alpha \rangle\ \varphi, \Delta \\ 2. & \Gamma, st?\_mode = normal, r = null \\ & \quad \vdash \langle st/\texttt{throw new } NullPointerException();\ \alpha \rangle\ \varphi, \Delta \\ 3. & \Gamma, st?\_mode = normal, r \neq null \vdash \langle st/\beta\ \texttt{endfinally}(throw(r))\ \gamma \rangle\ \varphi, \Delta\end{array}}{\Gamma \vdash \langle st/\texttt{throw } r;\ \alpha \rangle\ \varphi, \Delta}$$

2. Next possible catcher is a `catches` statement, $\alpha = \alpha'$ **catches**$(catch_1, ..., catch_n)$ β

   A `catches` statement catches the throw if the class of the thrown reference r is a subclass of a caught exception class. Then the first catching clause applies. Otherwise the throw continues, i.e. we re-throw the exception. This allows again to apply the efficient version of the rule.

3. Next possible catcher is a `endstatic` statement, $\alpha = \alpha'$ **endstatic**(c) β. We just discard α and apply the first rule.

4. There is no possible catcher in α. Discard α and apply the first rule.

3.3.18 Try

$$\frac{\begin{array}{ll}1. & \Gamma, st[mode] \neq normal \vdash \varphi, \Delta \\ 2. & \Gamma, st[mode] = normal \vdash \langle st/\alpha \texttt{ catches finally}(\beta)\rangle\ \varphi, \Delta\end{array}}{\Gamma \vdash \langle st/\texttt{try } \alpha \textit{ catches } \texttt{finally } \beta\rangle\ \varphi, \Delta}$$

The list of catch clauses *catches* is transformed into an additional Java statement *Catches*, the finally clause is transformed into an additional Java statement `finally` (see below).

3.3.19 Catches

$$\frac{\begin{array}{ll}1. & \Gamma, is\_throw\_mode(st[mode]), st[mode].type \leq c_1, st_0 = st[mode][normal] \\ & \vdash \langle st_0/\{x_1' = st[mode].val; \alpha_1'\}\rangle\ \varphi[st/st_0], \Delta \\ & \vdots \\ n. & \Gamma, is\_throw\_mode(st[mode]), \neg\ st[mode].type \leq c_1, ..., \neg\ st[mode].type \leq c_{n-1}, \\ & st[mode].type \leq c_n, st_0 = st[mode][normal] \vdash \langle st_0/\{x_n' = st[mode].val; \alpha_n'\}\rangle\ \varphi[st/st_0], \Delta \\ n+1. & \Gamma, \quad \neg\ is\_throw\_mode(st[mode]) \\ & \quad \vee (\neg\ st[mode].type \leq c_1 \wedge ... \wedge \neg\ st[mode].type \leq c_n) \vdash \varphi, \Delta\end{array}}{\Gamma \vdash \langle st/\ \texttt{catches}(c_1(x_1)\alpha_1 ... c_n(x_n)\alpha_n)\rangle\ \varphi, \Delta}$$

$st[mode].type \leq c$ is again a formula computed in the usual manner.

3.3.20 finally

$$\frac{1.\quad \Gamma, st_0 = st[mode][normal] \vdash \langle st_0/\alpha\rangle\ \langle st_0/\ \texttt{endfinally}(st[mode])\rangle\ \varphi[st/st_0], \Delta}{\Gamma \vdash \langle st/\texttt{finally}(\alpha)\rangle\ \varphi, \Delta}$$

3.3.21 endfinally

$$\frac{\begin{array}{ll}1. & \Gamma, st[mode] \neq normal \vdash \varphi, \Delta \\ 2. & \Gamma, st[mode] = normal, st_0 = st[mode][m] \vdash \varphi[st/st_0], \Delta\end{array}}{\Gamma \vdash \langle st/\texttt{endfinally}(m)\rangle\ \varphi, \Delta}$$

Second version:

$$\frac{\begin{array}{ll}1. & \Gamma \vdash m = normal \vee is\_break\_mode(m) \vee is\_return\_mode(m) \vee is\_throw\_mode(m), \Delta \\ 2. & \Gamma, m = normal \vdash \varphi, \Delta \\ 3. & \Gamma, is\_break\_mode(m) \vdash \langle st/\texttt{break}(m.lab)\rangle\ \varphi, \Delta \\ 4. & \Gamma, is\_return\_mode(m) \vdash \langle st/\texttt{return } m.val;\rangle\ \varphi, \Delta \\ 5. & \Gamma, is\_throw\_mode(m) \vdash \langle st/\texttt{throw}m.val;\rangle\ \varphi, \Delta\end{array}}{\Gamma \vdash \langle st/\texttt{endfinally}(m)\rangle\ \varphi, \Delta}$$

This version is correct since the `break`, `return`, or `throw` statements will be discarded if the current mode is not normal. The second version allows for an optimization that discards non-catching statements.

3.4 Additional rules

3.4.1 flatten

The rule *flatten* deals with nested expressions by introducing auxiliary variables. For example, $\langle st/x = m(f(g(y)), h(z)); \rangle\ \varphi$ becomes $\langle st/x_0 = g(y); x = m(f(x_0), h(z)); \rangle\ \varphi$. The rule only flattens the first occurrence of a nested expression. The rule is always applicable, even in case of a jump. Of course, the evaluation order of Java must be preserved.

3.4.2 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 considered 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 (i→b etc.).

unary expression `!` (logical negation), `~` (bitwise complement), `+` , `-` can be literalized if the argument can be literalized.

binary expression this does not include / and % (because they can cause an exception), and not `||`, `&&` (because they have a different evaluation order), and not `instanceof` (because it needs a store lookup), but `==`, `+` , `-`, `*`, `<`, `>`, `<=`, `>=`, `&`, `^`, `|`, `<<`, `>>`, `>>>`. Both arguments must be literalizable. (e_1 `!=` e_2 is transformed into `!` (e_1 `==` e_2).)

The rule is always applicable, even in case of a jump.

3.4.3 simplify

This rule is a (recursive) combination of *block*, *flatten*, and *literalize*, and transforms a double assignment to local variables $\langle st/x = (y = e); \rangle\ \varphi$ into $\langle st/y = e; x = y; \rangle\ \varphi$.

3.4.4 Split

$$\frac{1.\quad \langle st/\alpha \rangle\ (st = st_0 \wedge \underline{x} = \underline{y}), \varphi^{st_0,\underline{y}}_{st,\underline{x}}, \Gamma \vdash \Delta}{\langle st/\alpha \rangle\ \varphi, \Gamma \vdash \Delta}$$

This rule assumes the program on the left side of the sequent because we need the termination of α. $\underline{x}$ are the assigned (local) variables (not fields!) of the program α, i.e. all variables that may change their value. st_0 and $\underline{y}$ are new variables. Actually, st is an assigned variable, but included here explicitely for clarity. Note, that in the statement $x = y.m(z)$; the assigned variables are $\{x\}$, independent of the method implementation: The method m cannot modify *variables*, only *fields*, i.e. the store st. If m throws an exception (or we are in jump mode) no assignment occurs, but it is save to include too many variables in the assigned variables. The rule for the right hand side includes an additional termination goal and shifts the program to the left:

$$\frac{\begin{array}{ll} 1. & \Gamma \vdash \langle st/\alpha \rangle\ true, \Delta \\ 2. & \langle st/\alpha \rangle\ (st = st_0 \wedge \underline{x} = \underline{y}), \Gamma \vdash \varphi^{st_0,\underline{y}}_{st,\underline{x}}, \Delta \end{array}}{\Gamma \vdash \langle st/\alpha \rangle\ \varphi, \Delta}$$

The split rule in this form is not very useful because we normally need at least some information about the store after α, but it is the basis of the following, very important, rules.

3.4.5 Execute Program

This rule comes in different flavours, depending on the optimizations that are built into it. The simplest version assumes two syntactically equal programs and the same store in the antecedent and the succedent of the sequent:

$$\frac{1.\quad \langle st/\alpha\rangle\ \underline{x} = \underline{y}, \varphi^{\underline{y}}_{\underline{x}}, \Gamma \vdash \psi^{\underline{y}}_{\underline{x}}, \Delta}{\langle st/\alpha\rangle\ \varphi, \Gamma \vdash \langle st/\alpha\rangle\ \psi, \Delta}$$

Obviously, α in the antecedent and the succedent compute the same results (the two stores are equal!). This means we can discard the program in the succedent and continue with ψ if we substitute all assigned variables $\underline{x}$ by their new values $\underline{y}$.

The second version allows renamings of variables. This is necessary because other proof rules may rename variables. α' is α, except that some variables are renamed (different variables to different new variables).

$$\frac{\begin{array}{ll}1. & \Gamma \vdash \underline{z} = \underline{z}', \Delta \\ 2. & \langle st/\alpha\rangle\ \underline{x} = \underline{y}, \varphi^{\underline{y}}_{\underline{x}}, \Gamma \vdash \psi[\underline{z}'/\underline{z}]^{\underline{y}}_{\underline{x}}, \Delta\end{array}}{\langle st/\alpha\rangle\ \varphi, \Gamma \vdash \langle st'/\alpha'\rangle\ \psi, \Delta}$$

$\underline{z}' \to \underline{z}$ is the variable renaming, i.e. $\alpha = \alpha'[\underline{z}'/\underline{z}]$. Note that there is no connection between the renamed variables $\underline{z}$ and the assigned variables of α. The first premise guarantees that the renamed variables have the same values.

3.4.6 Contract Program

We can do the same as execute program does if we have the two programs in the antecedent:

$$\frac{1.\quad \langle st/\alpha\rangle\ \underline{x} = \underline{y}, \varphi^{\underline{y}}_{\underline{x}}, \psi^{\underline{y}}_{\underline{x}}, \Gamma \vdash \Delta}{\langle st/\alpha\rangle\ \varphi, \langle st/\alpha\rangle\ \psi, \Gamma \vdash \Delta}$$

$$\frac{\begin{array}{ll}1. & \Gamma \vdash \underline{z} = \underline{z}', \Delta \\ 2. & \langle st/\alpha\rangle\ \underline{x} = \underline{y}, \varphi^{\underline{y}}_{\underline{x}}, \psi[\underline{z}'/\underline{z}]^{\underline{y}}_{\underline{x}}, \Gamma \vdash \Delta\end{array}}{\langle st/\alpha\rangle\ \varphi, \Gamma \vdash \langle st'/\alpha'\rangle\ \psi, \Delta}$$

3.4.7 ThrowIt

A special rule for JavaCard `ISOException.throwIt`.

3.4.8 For-Induction

A specialized rule to simplify induction for `for` loops.

3.5 Optimizations

There are three types of optimizations that can be performed.

1. The *simplify* rule can be applied implicitely after every rule application that modifies the Java Program.
2. Jumps can 'discard' statements until a catcher is found.

3. If a method (or statement) is known to leave the store unchanged (or to raise no exception) by static program analysis this fact can be included in the *split* and *execute* rules.

Further optimizations:

- Under certain conditions we can eliminate return/target constructs: No statements after returns and no try's.(?)
- $\langle st/\{x = m(y);\}\rangle\ \varphi \Rightarrow \langle st/\{x = m(y);\}\rangle\ (st = st0 \wedge x = x0)$

  Note that x is in general transient: It's value can change, but it also can remain unchanged.

  x remains unchanged, if $st[mode] \neq normal$ or the method m throws an exception (it is not possible that m ends with a return- or break-mode).

  $\Rightarrow$ we can't execute call in the following sequent

  $\langle st/\{x = m(y);\}\rangle\ \varphi \vdash \langle st/\{x_0 = m(y);\}\rangle\ \psi$

  Either we must show that $x = x_0$ holds or that the mode in st and the resulting st is normal (i.e. no jump).

  Of course, we can prove this property, and use it as a lemma.
- For very simple programs we can compute that no exception can occur: No instance fields, no try's, no throws, no arrays, no division, no breaks – what about class initialization? all classes initialized? no static fields and initializers? And all returns can be eliminated (see second item).

  In this case the store is never modified and we can modify the split rule accordingly.

Chapter 4

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
$\vdash$ ⟨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). The call rule has a special case for System.out.print(ln) that 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). Applying the rule for static method invocation (3.2.10, p. 15) 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, initerror(returnt, st) ⊢ ⟨st/throw new NoClassDefFoundError();⟩ flatten(st[_out]) = noval ++ intval(1)
3.	st[_out] = noval, initundone(returnt, st), st[mode] = normal, st[mode] = normal, initundone(returnt, st), initdone(returnt, st_0), st_0 = addclass(returnt, returnt.x × intval(0), st) ⊢ ⟨st_0/static(returnt)⟩ ⟨st_0/{ returnt.x = 1; }⟩ ⟨st_0/endstatic(returnt)⟩ ⟨st_0/returnt.main();⟩ flatten(st_0[_out]) = noval ++ intval(1)
4.	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 second goal is for the case that the class is in error state (initerror(returnt, st)), but initundone(returnt, st) is true, and this goal is also closed immediately. The fourth goal is for the case that `returnt` is already initialized, which is also false. We continue with the third goal that deals with the uninitialized case. The class and its static fields are added to the store (with `addclass`), and the static initializer that assigns `x` its initial value 1 is added to the program (enclosed in a `static`, `endstatic` pair).

The `static` statement is simply omitted because the super class of returnt is Object, which is assumed to be initialized. The rule for an assignment to a static field has again four premises, but only the last one is applicable (the other premises are closed immediately):

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(); }
ArithmeticException (e) { object._out(returnt.x); }⟩ ⟨st_1/return;⟩
⟨st_1/target(return)⟩ flatten(st_1[_out]) = noval ++ intval(1)

The `main` call is replaced by its body (with an explicit `return ;` statement added), and a `target` statement is added that will catch returns. (Since we have only one `return` in the main method that is at the end of the body, it is a simple optimization to omit the return/target pair.) 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/ArithmeticException (e) { object._out(returnt.x); }⟩ ⟨st_1/return;⟩
⟨st_1/target(return(refval(jvmref), void_type))⟩ 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/ArithmeticException (e) { object._out(returnt.x); }⟩
⟨st_1/return;⟩ ⟨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/ArithmeticException (e) { object._out(returnt.x); }⟩
⟨st_1/return;⟩
⟨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/ArithmeticException (e) { object._out(returnt.x); }⟩
⟨st_1/return;⟩
⟨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/ArithmeticException (e) { object._out(returnt.x); }⟩
⟨st_1/return;⟩
⟨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/ArithmeticException (e) { object._out(returnt.x); }⟩
⟨st_0/return;⟩
⟨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/{ ArithmeticException e = r_0; object._out(returnt.x); } ⟩ ⟨st_0/return;⟩
⟨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/return;⟩ ⟨st_1/target(return)⟩ flatten($st_1[\_out]$) = noval ++ intval(1)

A last application of the return rule finishes the proof.

Chapter 5

Test Programs

The following programs are handled correctly by the calculus, i.e. it is automatically proved that they compute the same result as a run of the programs with java. They can be regarded as a (admittedly not very systematic) test suite.

```
/* Example by Kurt Stenzel, ExArray/ex1 */

class Ex1 {
        public static void main(String[] args) {

          int[] x = new int[2];
          x[0] = 1;
          x[1] = 2;
          for(int i = 0; i <= 2; i++) {
          System.out.println(x[i]);
          }
        }
}

/* prints
1
2
java.lang.ArrayIndexOutOfBoundsException: 2
        at Ex1.main(Ex1.java:8)
*/
```

```
/* Example by Kurt Stenzel, ExArray/ex2 */

class Super { }

class Sub1 extends Super { }
class Sub2 extends Super { }
class SubSub extends Sub1 { }

class Ex2 {
```

```
    public static void main(String[] args)
    {
        Super[] x = new Sub1[2];
        x[0] = new SubSub();
        System.out.println(x.length);
        x[1] = new Sub2();
    }
}

/*
prints:


2

before causing an ArrayStoreException

*/
```

```
/* Language Specification main/section10.6_1, ExArray/ex3 */

class Ex3 {
        public static void main(String[] args) {
                int ia[][] = { {1, 2}, null };
                for (int i = 0; i < 2; i++)
                        for (int j = 0; j < 2; j++)
                                System.out.println(ia[i][j]);
        }
}

/*

prints:


1
2

before causing a NullPointerException
*/
```

```
/* Language Specification main/section10.10_1, ExArray/ex4 */

class Point { int x, y; }
class ColoredPoint extends Point { int color; }


class Ex4 {
```

```
        public static void main(String[] args) {
                ColoredPoint[] cpa = new ColoredPoint[10];
                Point[] pa = cpa;
                System.out.println(pa[1] == null);
                try {
                        pa[0] = new Point();
                } catch (NullPointerException e) {
                        System.out.println(e);
                }
        }
}

/*
produces the output:


true
before throwing an ArrayStoreException
*/
```

```
/* Example by Dominik Haneberg */

class MultEx1 {
        public static void main(String[] args) {

          int[][] x = new int[2][2];
          x[0][0] = 1;
          x[0][1] = 2;
          for(int i = 0; i <= 1; i++) {
          System.out.println(x[0][i]);
          }
          for(int i = 0; i <= 1; i++) {
          System.out.println(x[1][i]);
          }
        }
}

/* prints
1
2
0
0
java.lang.ArrayIndexOutOfBoundsException: 2
        at Ex1.main(Ex1.java:8)
*/
```

```
/* from langspec/main/section4.2.2_1 */

class Test {
```

```
        public static void main(String[] args) {
                int i = 2;
                System.out.println(i * i);
                int l = i;
                System.out.println(l * l);
                System.out.println(1 / (l - i));
        }
}

/*
prints: 4 4

and throws an ArithmeticException
*/
```

```
/* by Kurt Stenzel, ExException/Finally_Test */

class Finally_Test {
  static void do_something() {
    System.out.println(1);
  }

  static void the_test() {
    try {
      do_something();
    }
    catch (ArithmeticException e) {
      System.out.println(3);
    }
    finally {
      System.out.println(4);
    }
    System.out.println(5);
  }

  public static void main (String[] args) {
    the_test();
    System.out.println(6);
  }
}

/*
prints 1 4 5 6
*/
```

```
public class jump {

    public static int me (int x, boolean b1, boolean b2) {
```

```
        try {
        lab1_label : {
            try {
                if (b1 == true) { int i = 5/x; } else break lab1_label;
                return 3;
            }
            finally { if (b1 == b2) { int j = 4/x; } else
                { System.out.println(0); if (b2 == true) return 4; } }
        }
        return 6;
        }
        finally { System.out.println(1); }
    }


    public static void main (String[] argv) {

        try { System.out.println(me(0,true,false)); }
        catch (ArithmeticException e) {System.out.println(1);}
        try { System.out.println(me(0,false,true)); }
        catch (ArithmeticException e) {System.out.println(2);}
        try { System.out.println(me(0,false,false)); }
        catch (ArithmeticException e) {System.out.println(3);}
        try { System.out.println(me(1,true,false)); }
        catch (ArithmeticException e) {System.out.println(4);}
        try { System.out.println(me(1,false,false)); }
        catch (ArithmeticException e) {System.out.println(5);}
        try { System.out.println(me(0,true,true)); }
        catch (ArithmeticException e) {System.out.println(6);}
        try { System.out.println(me(1,true,true)); }
        catch (ArithmeticException e) {System.out.println(7);}
        try { System.out.println(me(1,false,true)); }
        catch (ArithmeticException e) {}

    }

}

/* prints: 0 1 1/ 0 1 4/ 1 3/ 0 1 3/ 1 6/ 1 6/ 1 3/ 0 1 4
*/
```

```
/* by Kurt Stenzel, ExException/Nullpointer_Test */

class Something {
  int i;
  Something (int j) {
    i=j;
  }
  void method1() {
    System.out.println(i);
  }
}
```

```
class Nullpointer_Test {
  static void create_something() {
    Something so;
    so=null;
    so.method1();
  }

  public static void main (String[] args) {
    try {
      create_something();
    }
    catch (NullPointerException e) {
      System.out.println(2);
    }
  }
}

/*
prints: 2
*/
```

```
/* by Kurt Stenzel, ExException/Throw_Test */

class Throw_Test {
  static void throw_something() {
    System.out.println(1);
    throw new ArithmeticException();
  }

  public static void main (String[] args) {
    try {
      throw_something();
    }
    catch (ArithmeticException e) {
      System.out.println(3);
    }
  }
}

/*
prints: 1 3
output = mkival(1i)'ol ÿol mkival(1i +i 1i +i 1i)'ol
*/
```

```
/* by Kurt Stenzel, ExException/Try_Catch_Test */

class Try_Catch_Test {
  static void do_something() {
    System.out.println(1);
  }
```

```
  static void throw_something() {
    System.out.println(2);
    throw new ArithmeticException();
  }

  static void the_test() {
    try {
      do_something();
    }
    catch (ArithmeticException e) {
      System.out.println(3);
    }
    System.out.println(4);
    try {
      throw_something();
    }
    catch (ArithmeticException e) {
      System.out.println(5);
    }
    System.out.println(6);
  }

  public static void main (String[] args) {
    the_test();
    System.out.println(7);
  }
}

/*
prints: 1 4 2 5 6 7
*/
```

```
/* by Kurt Stenzel, ExException/Try_Catch_Test2 */

class Try_Catch_Test2 {
  static void throw_something() {
    System.out.println(0);
    throw new NullPointerException();
  }

  static void the_test() {
    try {
      throw_something();
    }
    catch (ArithmeticException e) {
      System.out.println(7);
    }
    catch (ClassCastException e) {
      System.out.println(6);
    }
    System.out.println(1);
```

```
  }

  public static void main (String[] args) {
    try {
      the_test();
    }
    catch (NullPointerException e) {
      System.out.println(2);
    }
  }
}

/*
prints: 0 2
*/
```

```
/* Example by Kurt Stenzel */

class myError extends Error {

static int id(int x) { System.out.println(5); throw new myError(); }

static int i = 1 / id(0);

}

public class ErrorInit {

public static void main (String[] args) {

    try {
          myError x = new myError();
          System.out.println(0);
        }
    catch (ExceptionInInitializerError i) {
          System.out.println(1);
          myError y = new myError();
          System.out.println(1);
          System.out.println(myError.i);
          }
    catch (Throwable i) {
          System.out.println(2);
          myError y = new myError();
          }

    finally { System.out.println(3); };

    System.out.println(4);

}

}
```

```
/*
prints: 5 2 3
before throwing a NoClassDefFoundError

*/
```

```
/* Language specification section12.4.1_1, ExInit/ex1 */

class Super {
  static { System.out.print(0); } // instead of "Super "
}

class One {
        static { System.out.print(1); } // instead of "One "
}

class Two extends Super {

    static int x = 2;
    static int y = x;
    static { System.out.print(y); } // instead of "Two "
}

class Test {
        public static void main(String[] args) {
                One o = null;
                Two t = new Two();
                System.out.println((Object)o == (Object)t);
        }
}

/*
prints: 0 2 false
*/
```

```
/* Example by Kurt Stenzel */

class ErrorInInit {

static int id(int x) { return x; }

static int i = 1 / id(0);

}

public class ExInit {
```

```
public static void main (String[] args) {

    try {
          ErrorInInit x = new ErrorInInit();
          System.out.println(0);
        }
    catch (ExceptionInInitializerError i) {
          System.out.println(1);
          ErrorInInit y = new ErrorInInit();
          System.out.println(1);
          System.out.println(ErrorInInit.i);
          }
    catch (ArithmeticException i) {
          System.out.println(2);
          ErrorInInit y = new ErrorInInit();
          }

    finally { System.out.println(3); };

    System.out.println(4);

}

}

/* prints: 1 3
before throwing a NoClassDefFoundError
*/
```

```
/* Example by Kurt Stenzel */

class StatInit {

  static int i, j;

  static { { int k = 3; i = k; } {int k = 4; j = k; } }

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

/* prints
3
4

*/
```

```
interface Colorable { }

class ColoredPoint implements Colorable { }

public class Ex1 {

    public static void main(String[] args) {
        Colorable[] ar;
        ColoredPoint[] ar0;
        ar0 = new ColoredPoint[2];
        ar = (Colorable[])ar0;
        ar = new Colorable[3];
        System.out.println(1);
        ar = (ColoredPoint[])ar;
    }
}

/* prints

1

before throwing a ClassCastException
*/
```

```
/* from langspec, section 12.4.1, third example */

interface I {
        int i = 1, ii = Test.out(1, 2);
}

interface J extends I {
        int j = Test.out(2, 3), jj = Test.out(3, 4);
}

interface K extends J {
        int k = Test.out(4, 5);
}

class Test {

        public static void main(String[] args) {
            // J.i is a compile-time constant
            System.out.println(J.i);
            System.out.println(K.j);
        }

        static int out(int s, int i) {

                System.out.println(s);
                System.out.println(i);
                return i;
        }
```

```
}

/*
Only interface J is initialized.

prints    1 2 3 3 4 3

*/
```

```
/* by Kurt Stenzel, ExMath/gcd64 */

public class byteshort {

  public static void main (String[] args) {

       short b1 = 20000, c = 10, b3,b4,b5,b6;
       b4 = (short)((short)(b1 * 5) / c);

       b5 = (short)(b1 * 5);
       b6 = (short)(b5 / c);
       b3 = (short)((b1 * 5) / c);

       System.out.println((short)0x1234);
       System.out.println(b4);
       System.out.println(b5);
       System.out.println(b6);
       System.out.println(b3);
  }
}
/* prints:

4660
-3107
-31072
-3107
10000
 */
```

```
/* by Kurt Stenzel, ExMath/gcd64 */

public class bytetest {

  public static void main (String[] args) {

       System.out.println((byte)(128));
       System.out.println((byte)(-129));
       System.out.println((byte)(123));
       System.out.println((byte)(-126));
       System.out.println((byte)(-511));
       System.out.println((byte)(-512));
```

```
      System.out.println((byte)(511));
      System.out.println((byte)(512));
  }
}
/* prints: -128 127 123 -126 1 0 -1 0
 */
```

```
public class test {

    public static void main (String[] args) {

        for(int i = -3; i < 3; i++)
            System.out.println( ((~ (byte)i) + 1) == (- (byte)i));
    }
}
```

```
public class intdiv {

  public static void main (String[] args) {
      //      System.out.println((5 / 3)   + " % = " + (5 % 3));
      //      System.out.println((5 / -3)  + " % = " + (5 % -3));
      //      System.out.println((-5 / 3)  + " % = " + (-5 % 3));
      //      System.out.println((-5 / -3) + " % = " + (-5 % -3));
      System.out.println(( 5 /  3));
      System.out.println(( 5 %  3));
      System.out.println(( 5 / -3));
      System.out.println(( 5 % -3));
      System.out.println((-5 /  3));
      System.out.println((-5 %  3));
      System.out.println((-5 / -3));
      System.out.println((-5 % -3));
  }
}

/* prints: 1 2 -1 2 -1 -2 1 -2

 1 % = 2
-1 % = 2
-1 % = -2
 1 % = -2
*/
```

```
/* by Kurt Stenzel, ExMath/gcd64 */

public class gcd {
```

```
  public static int gcd (int x, int y) {
    if (x == 0) return y; else
      if (y == 0) return x; else
        {
          while (x != y) {
            while (x < y) y = y - x;
            while (y < x) x = x - y;
          };
          return x;
        }
  }

  public static void main (String[] args) {
    System.out.println(gcd(6,4));
  }
}
/* prints 2
 */
```

```
/* langspec/main/section4.3.1_2 */

class Value { int vali; } // val => vali


class Test {
        public static void main(String[] args) {
                int i1 = 3;
                int i2 = i1;
                i2 = 4;
                System.out.println(i1);
                System.out.println(i2);
                Value v1 = new Value();
                v1.vali = 5;
                Value v2 = v1;
                v2.vali = 6;
                System.out.println(v1.vali);
                System.out.println(v2.vali);
        }
}

/* prints 3 4 6 6
*/
```

```
/* langspec/main/section4.5.4_1 */

class Point {
        static int npoints;
        int x, y;
        Point root;
```

```
}

class Test {
        public static void main(String[] args) {
                System.out.println(Point.npoints);
                Point p = new Point();
                System.out.println(p.x);
                System.out.println(p.y);
                System.out.println(p.root);
        }
}

/*
prints: 0 0 0 null
*/
```

```
/* langspec/main/section6.3.1_1 */

class Test {
        static int x = 1;
        public static void main(String[] args) {
                int x = 0;
                System.out.println(x);
                System.out.println(Test.x);
        }
}

/* prints 0 1
*/
```

```
/* langspec/main/section8.3.1.1_1 */

class Point {
        int x, y, useCount;
        Point(int x, int y) { this.x = x; this.y = y; }
        final static Point origin = new Point(0, 0);
}

class Test {
        public static void main(String[] args) {
                Point p = new Point(1,1);
                Point q = new Point(2,2);
                p.x = 3; p.y = 3; p.useCount++; p.origin.useCount++;
                System.out.println(q.x);
                System.out.println(q.y);
                System.out.println(q.useCount);
                System.out.println(q.origin == Point.origin);
                System.out.println(q.origin.useCount);
        }
}

/* prints: 2 2 0 true 1 */
```

```
/* langspec/main/section8.3.1.1_1, simplied */

class Point {
        int x, y, useCount;
        Point(int x, int y) { this.x = x; this.y = y; }
}

class Test {
        public static void main(String[] args) {
                Point q = new Point(2,2);
                System.out.println(q.x);
                System.out.println(q.y);
        }
}

/* prints: 2 2
*/
```

```
/* langspec/main/section8.3.2_1 */

class Point {
        int x = 1, y = 5;
}
class Test {
        public static void main(String[] args) {
                Point p = new Point();
                System.out.println(p.x);
                System.out.println(p.y);
        }
}

/* prints: 1 5
*/
```

```
/* langspec/main/section8.3.3.1_1, one line modified */

class Point {
        static int x = 2;
}

class Test extends Point {

        int x = 4;
        public static void main(String[] args) {

                new Test().printX();
        }
        void printX() {
```

```
                System.out.println(x);
                System.out.println(super.x);
        }
}

/* prints: 4 2
*/
```

```
/* langspec/main/section12.5_2 */

class Super {
        Super() { printThree(); }
        void printThree() { System.out.println(1); }
}


class Test extends Super {
        int indiana = 3;

        public static void main(String[] args) {
                Test t = new Test();
                t.printThree();
        }
        void printThree() { System.out.println(indiana); }
}

/* prints: 0 3
*/
```

```
class I { boolean x; }
public class J extends I {
    int x = 1;
    J() {
        x = 5;
        System.out.println(x);
        System.out.println(super.x);
    }
    public static void main(String[] args) { new J(); }
}

/* prints: 5 false
 */
```

```
class acast {

    public void printsome() { System.out.println(0); }
```

```
}

public class acast2 extends acast {

     public void printsome() { System.out.println(1); }

    public static void main (String[] argv) {
        acast2 ox = new acast2();
        ((acast)ox).printsome();
        acast oy = new acast();
        ((acast2)oy).printsome();
    }
}

/* prints: 1
and
java.lang.ClassCastException: acast
        at acast2.main(acast2.java:17)
*/
```

```
/* by Kurt Stenzel, ExObject/Explicit_Constructor_Call_Test */

class The_One {
  int i;

  The_One (int j) {
    i=j;
  }

  The_One (int l, int mvar) {
    this(l);
  }

  void And_Only() {
    System.out.println(i);
  }
}

class explicit_constrcall {
  public static void main(String[] args) {
    The_One t;

    t=new The_One(3,4);
    t.And_Only();
  }
}

/*
prints: 3
*/
```

```
/* by Kurt Stenzel, ExObject/Method_Overloading_Test */

class The_One {
  void do_something() {
    System.out.println(0);
  }

  void do_something2() {
    do_something();
    System.out.println(1);
  }

  void do_something3() {
    System.out.println(3);
  }
}

class And_Only extends The_One{
  void do_something() {
  System.out.println(2);
  }

  void do_something3() {
    System.out.println(4);
  }

  void do_something4() {
    super.do_something2();
  }
}

class Method_Overloading_Test {
  public static void main(String[] args) {
    The_One t;
    And_Only a;

    t=new The_One();
    a=new And_Only();
    t.do_something3();
    a.do_something3();
    t.do_something2();
    a.do_something2();
    a.do_something4();
  }
}

/*
prints: 3 4 0 1 2 1 2 1
*/
```

```
/*
```

```
   The following example demonstrates that it is illegal to handle
   x = me() by shifting the assignment in front of return's, at
   least for field assignments.
*/
public class returnshift {

    static int x;

    public static int me1 () {
        int y = 3;
        try { return y; }
        finally { return y / (y - 3); }
    }

    public static int me2 () {
        int y = 3;
        try { { x = y; return x; } }
        finally { { x = y / (y - 3); return x; } }
    }

    public static int me3 () {
        int y = 3;
        try { return x = y; }
        finally { return x = y / (y - 3); }
    }

    public static void main (String[] argv) {
        try { x = me1(); }
        catch (ArithmeticException e) { System.out.println(x); }

        try { me2(); }
        catch (ArithmeticException e) { System.out.println(x); }

        try { me3(); }
        catch (ArithmeticException e) { System.out.println(x); }
    }
}

/* prints 0 3 3
*/
```

```
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); }

    }
}

/* prints: 1

*/
```

```
/* by Kurt Stenzel, ExStatement/CondExp_Test1 */

class CondExp_Test1 {
        public static void main(String[] args) {
                int i = 3;
                int j = 5;

                System.out.println(i<j?i:j);
        }
}

/*
prints: 3
output = mkival(1i +i 1i +i 1i) 'ol
*/
```

```
/* by Kurt Stenzel, ExStatement/CondExp_Test2 */

class CondExp_Test2 {
        public static void main(String[] args) {
                int i = 6;
                int j = 5;

                System.out.println(i<j?i:j);
        }
}
/*
prints: 5
output = mkival(1i +i 1i +i 1i +i 1i +i 1i) 'ol
*/
```

```
/* by Kurt Stenzel, ExStatement/CondExp_Test3 */

class CondExp_Test3 {
        public static void main(String[] args) {
                int i = 6;
                int j = 0;
```

```
                System.out.println((i/j)>1?i:j);
        }
}

/*
should throw an exception (division by zero)
output = @ol
*/
```

```
/* by Kurt Stenzel, ExStatement/Break_Continue_Test */

class Break_Continue_Test {
  static void the_test() {
    int i;
    int j;

    i=2;
    lab6: while (true) {
      if(i==0) break lab6;
      System.out.println(i);
      i=i-1;
    }

    i=0;
    lab7: while (i<=5) {
      i=i+1;
      if (i<=3) continue lab7;
      System.out.println(i);
    }

    i=0;
    lab1: while (i<=5) {
      i=i+1;
      j=0;
      System.out.println(i);
      while (true) {
        if (j>=i) break lab1;
        j=j+1;
      };
    }
    i=0;
    lab3: while (i<=2) {
      i=i+1;
      j=0;
      System.out.println(i);
      lab4: while (true) {
        if (j>=i) break lab4;
        j=j+1;
      };
    }
    i=0;
    lab2: while (i<=5) {
```

```
      i=i+1;
      j=0;
      System.out.println(i);
      lab5: while (true) {
        if (j>=i) continue lab2;
        j=j+1;
      };
    }
  }

  public static void main (String[] args) {
    the_test();
  }
}

/*
prints: 2 1 4 5 6 1 1 2 3 1 2 3 4 5 6
 */
```

```
class for_test {
    static void the_test() {
        int i;
        for(i=0; i<3; i=i+1) {
            System.out.println(i);
            i=i+1;
        }
    }

    public static void main (String[] args) {
        the_test();
    }
}
/* prints: 0 2
;;; output = mkival(0i)'ol ÿol mkival(1i +i 1i)'ol
*/
```

```
class lab_for {
    static void the_test() {
        int i;
        lab6: for(i=1; i<3; i=i+1) {
            System.out.println(i);
            if (i<=1) {
                continue lab6;
            }
            System.out.println(0);
        }
    }

    public static void main (String[] args) {
        the_test();
    }
```

```
}
/* prints: 1 2 0
*/
```

```
class lab_for_break {
    static void the_test() {
        int i;
        lab6: for(i=0; i<3; i=i+1) {
            System.out.println(i);
            if (i>=1) {
                break lab6;
            }
            System.out.println(0);
        }
    }

    public static void main (String[] args) {
        the_test();
    }
}
/* prints: 0 0 1
*/
```

```
public class simplify {

    short f = -1, g = -2;

    public simplify (int i) { g = (short)i; }
    public simplify () { }

    public byte m(short s, int i, boolean b, byte by) {
        return (byte)((short)(s + i + by));
    }

    public static int m1 (short s) { return s - 2; }

    public static void main(String[] argv) {

        int x = 1, y = 2, i = 4;
        boolean b;
        simplify r1 = new simplify();
        if (b = ((new simplify(2)).m((short)(x = y++), r1.f, i < r1.g + 3,
                                     (byte)(m1(r1.g--))) !=
                 (new simplify(-1)).m((short)(x = y++), r1.f, i < r1.g + 3,
                                      (byte)(m1(r1.g--)))))
            {
                System.out.println(b);
                System.out.println(i);
                System.out.println(x);
                System.out.println(y);
            }
```

```
        else
            {
                System.out.println(b);
                System.out.println(i);
                System.out.println(x);
                System.out.println(y);
            }
    }
}

/* prints: false 4 3 4
*/
```

```
class switch_nodefault {
    public static void main (String[] argv) {
        int i;

        i=1;
        switch (i) {
        case 0: System.out.println(0);
        }
        System.out.println(1);
    }
}
/* prints: 1
*/
```

```
public class switchtest {

    static void sw (int x) {
        switch (x) {

        default: System.out.println(0);
        case 1: int y = 2; System.out.println(1);
        case 2: y = 2; System.out.println(y); break;
        case 3: System.out.println(3);
        case 4: System.out.println(4);
        case 6: case 7: System.out.println(6);
        case 8: case 9:
        }

    }

    public static void main (String[] argv) {

        sw(0); sw(1); sw(2); sw(3); sw(4); sw(5); sw(6);
    }
}
/* prints: 0 1 2/ 1 2/ 2/ 3 4 6/ 4 6/ 0 1 2/ 6
*/
```

Chapter 6

Semantic

The semantic of a Java statement or expression is defined by a relation between variable mappings. If v is a variable mapping $X \to value$, i.e. a function from variables to values, then $v[\![\alpha]\!]v_0$ defines the semantic of the java statement α by 'transforming' an initial mapping v into a final mapping v_0. This means we have an input/output semantic. The relation .[![.]!]. is defined inductively, i.e. by a set of reduction rules. $v[\![\alpha]\!]v_0$ is true iff $v[\![\alpha]\!]v_0$ is member of the smallest set that is closed under the reduction rules presented in the following sections. From the form of the rules it is clear that this smallest set is well defined and not empty.

We use the reserved variables st for the store and r for the result value of expressions. This means that st and r always appear in our variable mapping, but not in the Java program. (Of course we could also use other names.) We will use st as a shortcut for $v(st)$, st_0 for $v_0(st)$, r for $v(r)$ etc., and m for $v(st)[mode]$.

6.1 Expression Semantic

Let e be an expression, es a list of expressions.

6.1.1 Jumps

An expression is evaluated only if no jump occurs. This is captured by the following rule:

$$\frac{m \neq normal}{v[\![e]\!]v}$$

We will assume $m = normal$ for all following rules for expressions (but not for statements).

6.1.2 First Active Use

A *first active use* can occur in case of a

1. new (class or array creation)
2. static field access
3. static field assignment
4. static method invocation

Let α be the static initializer of class c. If c does not exist we assume $\alpha = \{\}$ (empty statement). **static**(c) and **endstatic**(c) are additional Java statements.

1. new class

$$\frac{initundone(c, st) \quad v[\![\{\textbf{static}(c)\ \alpha\ \textbf{endstatic}(c)\}]\!]v_0 \quad v_0[\![new\ c(e_1, \ldots, e_n)]\!]v_1}{v[\![new\ c(e_1, \ldots, e_n)]\!]v_1} \tag{6.1}$$

2. new array

$$\frac{initundone(c, st) \quad v[\![\{\textbf{static}(c)\ \alpha\ \textbf{endstatic}(c)\}]\!]v_0 \quad v_0[\![new\ c[e_1] \ldots [e_n][i]]\!]v_1}{v[\![new\ c[e_1] \ldots [e_n][i]]\!]v_1} \tag{6.2}$$

   Note: c is actually a type. However, a *first active use* occurs only if c is a reference type, not a primitive type.

3. static field access

$$\frac{initundone(c, st) \quad v[\![\{\textbf{static}(c)\ \alpha\ \textbf{endstatic}(c)\}]\!]v_0 \quad v_0[\![c.f]\!]v_1}{v[\![c.f]\!]v_1} \tag{6.3}$$

4. static field assignment

$$\frac{initundone(c, st) \quad v[\![\{\textbf{static}(c)\ \alpha\ \textbf{endstatic}(c)\}]\!]v_0 \quad v_0[\![c.f = e]\!]v_1}{v[\![c.f = e]\!]v_1} \tag{6.4}$$

5. static method invocation

$$\frac{initundone(c, st) \quad v[\![\{\textbf{static}(c)\ \alpha\ \textbf{endstatic}(c)\}]\!]v_0 \quad v_0[\![c.m(e_1, \ldots, e_n)]\!]v_1}{v[\![c.m(e_1, \ldots, e_n)]\!]v_1} \tag{6.5}$$

Remarks:

1. **static**(c) (and possibly **endstatic**(c)) always modifies the initialization state, i.e. for v_0 holds $\neg\ initundone(c, st_0)$.
2. If the mode in v_0 is not normal the expression will be skipped in the second precondition.
3. We need **static**(c) to set the initialization state to *done* and to start the initialization of the super class.
4. We need **endstatic**(c) to catch exceptions (that will be transformed into an ExceptionInInitializerError and set the initstate to *error*).

If the initialization state is *error* a NoClassDefFoundError must be thrown, i.e. we get five more rules.

1. new class

$$\frac{initerror(c, st) \quad v[\![\textbf{throw new}\ \text{NoClassDefFoundError}();]\!]v_0}{v[\![new\ c(e_1, \ldots, e_n)]\!]v_0} \tag{6.6}$$

2. new array

$$\frac{initerror(c, st) \quad v[\![\textbf{throw new}\ \text{NoClassDefFoundError}();]\!]v_0}{v[\![new\ c[e_1] \ldots [e_n][i]]\!]v_0} \tag{6.7}$$

3. static field access

$$\frac{initerror(c, st) \quad v[\![\textbf{throw new}\ \text{NoClassDefFoundError}();]\!]v_0}{v[\![c.f]\!]v_0} \tag{6.8}$$

4. static field assignment

$$\frac{initerror(c, st) \quad v[\![\textbf{throw new}\ \text{NoClassDefFoundError}();]\!]v_0}{v[\![c.f = e]\!]v_0} \tag{6.9}$$

5. static method invocation

$$\frac{initerror(c, st) \quad v[\![\textbf{throw new}\ \text{NoClassDefFoundError}();]\!]v_0}{v[\![c.m(e_1, \ldots, e_n)]\!]v_0} \tag{6.10}$$

6.1.3 Normal evaluation of expressions

We assume $m = normal$ for all following rules, and $initdone(c, st)$ for all possible cases of *first active use*. The following rules all have the form

$$\frac{v[\![e_0]\!]v_0 \quad \ldots \quad v_{n-1}[\![e_n]\!]v_n}{v[\![e]\!]v'}$$

where v' is computed in some manner from $v, v_0, \ldots, v_n$. If during the evaluation of a subexpression e_i an exception occurs the following expressions will not be evaluated, i.e. $v_{i+1} = \ldots v_n = v_i$. In this case we have $v' = v_n$. For the sake of brevity we will assume that the mode after the last expression is normal, $v_n(st)[mode] = normal$, in the following rules.

Expression lists

Expressionlists $es = e_1, \ldots, e_n$ are evaluated from left to right.

Literal

literal l

$$\frac{}{v[\![l]\!]v_r^{eval(l)}} \tag{6.11}$$

A literal in our context is an algebraic term. The value of the result variable r is set to this value $eval(l)$, i.e. the variable mapping v is modified for r (denoted by $v_r^{eval(l)}$).

Unary operation

Unary operation, one of +, -, ~, !

$$\frac{v[\![e]\!]v_0}{v[\![\oplus e]\!](v_0)_r^{eval(\oplus r_0)}} \tag{6.12}$$

Remember that we assume $m = m_0 = normal$. The expression e is evaluated, and the unary operation applied to its result r_0 ($eval(\oplus r_0)$), but only if the evaluation of e did not raise an exception ($m_0 = normal$).

Cast

Primitive cast:

JLS 5.5. A primitive cast changes the type of the argument and the argument itself, for example, a cast from integer to byte cuts off the upper 24 bits. A primitive cast never raises an exception.

$$\frac{v[\![e]\!]v_0}{v[\![(ty)e]\!](v_0)_r^{convert(r_0,ty)}} \tag{6.13}$$

Reference type cast:

JLS 15.15 and 5.5. A reference type cast does not change the runtime type (or rather class) of an object, just checks the class. See JLS 15.11.4.10 for an example. If the check fails a *ClassCastException* is thrown. A null value is accepted.

$$\frac{v[\![e]\!]v_0 \quad r_0 = null \vee asgcomp(r_0, ty)}{v[\![(ty)e]\!]v_o} \tag{6.14}$$

$$\frac{v[\![e]\!]v_0 \quad r_0 \neq null \wedge \neg\, asgcomp(r_0, ty) \quad v_0[\![\textbf{throw}\ new\ ClassCastEx();]\!]v_1}{v[\![(ty)e]\!]v_1} \tag{6.15}$$

instanceof

e *instanceof* ty. 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.

$$\frac{v[\![e]\!]v_0 \quad r_0 \neq null \wedge asgcomp(r_0, ty)}{v[\![e\ instanceof\ ty]\!](v_0)_r^{\text{true}}} \tag{6.16}$$

$$\frac{v[\![e]\!]v_0 \quad r_0 = null \vee \neg\ asgcomp(r_0, ty)}{v[\![e\ instanceof\ ty]\!](v_0)_r^{\text{false}}} \tag{6.17}$$

Conditional operator

In $e_0?e_1 : e_2$, first e_0 is evaluated, then either e_1 or e_2.

$$\frac{v[\![e_0]\!]v_0 \quad r_0 = \text{true} \quad v_0[\![e_1]\!]v_1}{v[\![e_0?e_1 : e_2]\!]v_1} \tag{6.18}$$

$$\frac{v[\![e_0]\!]v_0 \quad r_0 = \text{false} \quad v_0[\![e_2]\!]v_2}{v[\![e_0?e_1 : e_2]\!]v_2} \tag{6.19}$$

Conditional binary operation

These are `&&` and `||`. 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 `||`).

$$\frac{v[\![e_1]\!]v_0 \quad op = \texttt{\&\&} \wedge r_0 = \text{false} \vee op = \texttt{||} \wedge r_0 = \text{true}}{v[\![e_1\ op\ e_2]\!]v_0} \tag{6.20}$$

$$\frac{v[\![e_1]\!]v_0 \quad op = \texttt{\&\&} \wedge r_0 = \text{true} \vee op = \texttt{||} \wedge r_0 = \text{false} \quad v_0[\![e_2]\!]v_1}{v[\![e_1\ op\ e_2]\!](v_1)_r^{op(r_0,r_1)}} \tag{6.21}$$

Simple binary operation

simple binop

These are ==, !=, *, +, -, `<<`, `>>`, `>>>`, `>`, `<`, `<=`, `>=`, `&`, `^`, `|`. Arguments and result are either bool or integer (except for ==, !=, that accept arbitrary arguments of the same type). These binary operations do not raise exceptions.

$$\frac{v[\![e_1]\!]v_0 \quad v_0[\![e_2]\!]v_1}{v[\![e_1\ op\ e_2]\!](v_1)_r^{op(r_0,r_1)}} \tag{6.22}$$

Exception binary operation

/ and % raise an *ArithmeticException* if the divisor is zero.

$$\frac{v[\![e_1]\!]v_0 \quad v_0[\![e_2]\!]v_1 \quad r_1 \neq 0}{v[\![e_1\ op\ e_2]\!](v_1)_r^{op(r_0,r_1)}} \tag{6.23}$$

$$\frac{v[\![e_1]\!]v_0 \quad v_0[\![e_2]\!]v_1 \quad r_1 = 0 \quad v_1[\![\textbf{throw}\ new\ ArithmeticException()]\!]v_2}{v[\![e_1\ op\ e_2]\!]v_2} \tag{6.24}$$

6.1.4 Accesses

Local variable access

$$\frac{}{v[\![x]\!]v_r^{v(x)}} \tag{6.25}$$

Static field access

$$\frac{}{v[\![c.f]\!]v_r^{st[jvmref-c.f]}} \tag{6.26}$$

Instance field access $e.f$

e may not be *null*, otherwise a *NullPointerException* is thrown.

$$\frac{v[\![e]\!]v_0 \quad r_0 \neq null}{v[\![e.f]\!](v_0)_r^{st_0[r_0-f]}} \tag{6.27}$$

$$\frac{v[\![e]\!]v_0 \quad r_0 = null \quad v_0[\![\textbf{throw } new\ NullPointerException();]\!]v_1}{v[\![e.f]\!]v_1} \tag{6.28}$$

Array access $e_0[e_1]$

JLS 15.12.1. Both expressions are evaluated, then e_0 is checked to be not *null* (otherwise *NullPointerException*), then the index is checked (otherwise *IndexOutOfBoundsException*).

$$\frac{v[\![e_0]\!]v_0 \quad v_0[\![e_1]\!]v_1 \quad r_0 \neq null \quad 0 \leq r_1 < st_1[r_0 - length]}{v[\![\ e_0[e_1]\ ]\!](v_1)_r^{st_1[r_0-r_1]}} \tag{6.29}$$

$$\frac{\begin{array}{c}v[\![e_0]\!]v_0 \quad v_0[\![e_1]\!]v_1 \quad r_0 \neq null \quad \neg\ 0 \leq r_1 < st_1[r_0 - length] \\ v_1[\![\textbf{throw } new\ IndexOutOfBoundsEx()]\!]v_2\end{array}}{v[\![\ e_0[e_1]\ ]\!]v_2} \tag{6.30}$$

$$\frac{v[\![e_0]\!]v_0 \quad v_0[\![e_1]\!]v_1 \quad r_0 = null \quad v_1[\![\textbf{throw } new\ NPE()]\!]v_2}{v[\![\ e_0[e_1]\ ]\!]v_2} \tag{6.31}$$

6.1.5 Assignments

Local variable assignment $x = e$

$$\frac{v[\![e]\!]v_0}{v[\![x = e]\!](v_0)_x^{r_0}} \tag{6.32}$$

The value for x is updated to the value of e (stored in r_0), and the result of the assignment is also r_0. (We assume that all primitive conversions have been made explicit.)

Static field assignment $c.f = e$

$$\frac{v[\![e]\!]v_0}{v[\![c.f = e]\!](v_0)_{st}^{st_0[\text{jvmref}-c.f][r_0]}} \tag{6.33}$$

Instance field assignment $e.f = e_0$

The Java language specification JLS 15.26.1 is buggy because 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, this makes no sense, and JDK does not work this way. Correct is: e and e_0 are evaluated, then e is checked to be not null.

$$\frac{v[\![e]\!]v_0 \quad v_0[\![e_0]\!]v_1 \quad r_0 = null \quad v_1[\![\textbf{throw } new\ NullPointerEx()]\!]v_2}{v[\![e.f = e_0]\!]v_2} \tag{6.34}$$

$$\frac{v[\![e]\!]v_0 \quad v_0[\![e_0]\!]v_1 \quad r_0 \neq null}{v[\![e.f = e_0]\!](v_1)_{st}^{st_1[r_0-f][r_1]}} \tag{6.35}$$

Array assignment $e_0[e_1] = e_2$

JLS 15.25.1. All three expressions are evaluated, then e_0 is checked to be not null (otherwise *NullPointerException*), then the index e_1 is checked (otherwise *IndexOutOfBoundsException*), and finally the runtime type is checked to be *assignment compatible* (otherwise *ArrayStoreException*).

$$\frac{v[\![e_0]\!]v_0 \quad v_0[\![e_1]\!]v_1 \quad v_1[\![e_2]\!]v_2 \quad r_0 \neq null \wedge 0 \leq r_1 < st_2[r_0 - length] \wedge asgcomp(r_2, st_2[r_0 - type])}{v[\![e_0[e_1] = e_2]\!](v_2)_{st}^{st_2[r_0 - r_1][r_2]}} \tag{6.36}$$

$$\frac{v[\![e_0]\!]v_0 \quad v_0[\![e_1]\!]v_1 \quad v_1[\![e_2]\!]v_2 \quad r_0 \neq null \wedge 0 \leq r_1 < st_2[r_0 - length] \wedge \neg \ asgcomp(r_2, st_2[r_0 - type]) \quad v_2[\![\textbf{throw } new \ ArrayStoreEx()]\!]v_3}{v[\![e_0[e_1] = e_2]\!]v_3} \tag{6.37}$$

$$\frac{v[\![e_0]\!]v_0 \quad v_0[\![e_1]\!]v_1 \quad v_1[\![e_2]\!]v_2 \quad r_0 \neq null \wedge \neg \ \ 0 \leq r_1 < st_2[r_0 - length] \quad v_2[\![\textbf{throw } new \ IndexOutOfBoundsEx()]\!]v_3}{v[\![e_0[e_1] = e_2]\!]v_3} \tag{6.38}$$

$$\frac{v[\![e_0]\!]v_0 \quad v_0[\![e_1]\!]v_1 \quad v_1[\![e_2]\!]v_2 \quad r_0 = null \quad v_2[\![\textbf{throw } new \ NullPointerEx()]\!]v_3}{v[\![e_0[e_1] = e_2]\!]v_3} \tag{6.39}$$

Prefix/postfix increment/decrement operation

`++` and `--`. The argument must be a variable access, i.e. either a static or instance field, a local variable, or an array component. For a postfix operation the result is the value of the variable, as a side effect it is modified. For every operation we get four rules, one for each kind of access, e.g.

$$\frac{v[\![x]\!]v_0}{v[\![x\texttt{++}]\!](v_0)_x^{r+1}} \qquad \frac{v[\![c.f]\!]v_0}{v[\![c.f\texttt{++}]\!](v_0)_{st}^{st_0[\text{jvmref}-c.f][st_0[\text{jvmref}-c.f]+1]}} \tag{6.40}$$

Compound assignment

$e_1 = \oplus e_2$. e_1 must be a variable access, either a static or instance field, a local variable, or an array component. Exceptions are thrown before e_2 is evaluated, because a compound assignment works like $e_1 = e_1 \oplus e_2$ except that e_1 is evaluated only once. E.g. for an instance field

$$\frac{v[\![e_1]\!]v_0 \quad r_0 \neq null \quad v_0[\![e_2]\!]v_1}{v[\![e_1.f = \oplus e_2]\!](v_1)_{st}^{st_1[r_0 - f][r_1]}} \tag{6.41}$$

$$\frac{v[\![e_1]\!]v_0 \quad r_0 = null \quad v_0[\![\textbf{throw } new \ NullPointerEx();]\!]v_1}{v[\![e_1.f = \oplus e_2]\!]v_1} \tag{6.42}$$

6.1.6 Method invocations and new

New class

See JLS LR 12.5 and 15.8. 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.
2. The arguments $e_1, \ldots, e_n$ are evaluated.
3. A case distinction follows: if the body of the constructor begins with a constructor call for the same class (i.e. with a **this** call), it is executed, and afterwards the rest of the constructor is executed.

4. Otherwise the constructor of the super class is called, the fields of the class c are initialized, and the rest of the constructor is executed.

We assume that

1. the body of each constructor begins with an explicit constructor call (except for class Object),
2. the initialization of fields is handled by assignments to the fields that follow a **super** call,
3. and that every execution path ends with a **return**(this) statement (this assumption differs from the JLS)

We ignore *OutofMemory*.

$$\frac{\neg\, ref \in st \quad v_{st}^{addobj(ref,fs,st)}[\![e_1]\!]v_1 \quad \ldots \quad v_{n-1}[\![e_n]\!]v_n \quad (v_n)_{x_1,\ldots,x_n,this}^{r_1,\ldots,r_n,ref}[\![\alpha]\!]v_0 \quad is\_return\_mode(st_o[mode])}{v[\![new\ c(e_1,\ldots,e_n)]\!](v_0)_{x_1,\ldots,x_n,this,r,st}^{v_n(x_1),\ldots,v_n(x_n),v_n(this),ref,st_0[mode][normal]}} \tag{6.43}$$

First we obtain a new reference ($\neg\, ref \in st$), add a new object to the store ($v_{st}^{addobj(ref,fs,st)}$) and evaluate the arguments of the call. α is the body of the constructor with formal parameters $x_1,\ldots,x_n$. We set the local variables to the values of the call arguments and **this** to the new reference ref, and evaluate the constructor body. Afterward the old values for $x_1,\ldots,x_n,this$ are restored, the result r is bound to the new reference, and the mode is set back to normal (because α will end with a `return`).

If the mode of st_0 is not *return*, but an exception, $x_1,\ldots,x_n,this$ are still restored, but r and st remain unchanged.

$$\frac{\neg\, ref \in st \quad v_{st}^{addobj(ref,fs,st)}[\![e_1]\!]v_1 \quad \ldots \quad v_{n-1}[\![e_n]\!]v_n \quad (v_n)_{x_1,\ldots,x_n,this}^{r_1,\ldots,r_n,ref}[\![\alpha]\!]v_0 \quad \neg\, is\_return\_mode(st_o[mode])}{v[\![new\ c(e_1,\ldots,e_n)]\!](v_0)_{x_1,\ldots,x_n,this}^{v_n(x_1),\ldots,v_n(x_n),v_n(this)}} \tag{6.44}$$

New array

$new\ ty[e_1]\ldots[e_n][n]$, ty is a type, $e_1,\ldots,e_n$ the list of arguments for the dimensions, n the number of additional dimensions (needed to determine the correct default values for the array components: **null**, if $n>0$, the default value for ty otherwise).

JLS 15.9.1 specifies:

1. All arguments are evaluated. If one argument is <0 a *NegativeArraySizeException* is thrown.
2. The arrays are created. (Here an *OutOfMemoryError* can occur.)
3. Finally the arrays are initialized with the correct default values.

First the onedimensional case:

$$\frac{v[\![e]\!]v_0 \quad 0 \le r_0 \quad \neg\, ref \in st_0}{v[\![new\ ty[e]]\!](v_0)_{r,st}^{ref,addarray(ref,ty,r_0,st_0)}} \tag{6.45}$$

$$\frac{v[\![e]\!]v_0 \quad r_0 < 0 \quad v_0[\![\textbf{throw}\ new\ NegativeArraySizeException();]\!]v_1}{v[\![new\ ty[e]]\!]v_1} \tag{6.46}$$

$$\frac{\begin{array}{c} v[\![e_1]\!]v_1 \quad \ldots \quad v_{n-1}[\![e_n]\!]v_n \quad 0 \le r_1 \wedge \ \ldots\ \wedge 0 \le r_n \\ \neg\, ref \in st_0 \wedge is\_newref\_list(refs, st_0) \wedge \neg\, r \in refs \\ n \to i(\#refs) = countrefs(r_1 + \ldots + r_n) \end{array}}{v[\![new\ ty[e_1]\ldots[e_n][n]]\!](v_n)_{r,st}^{ref,addarraymultlist(ref,ty,st_n,refs,r_1+\ldots+r_n,r_1+\ldots+r_n,n)}} \tag{6.47}$$

$$\frac{v[\![e_1]\!]v_1 \quad \ldots \quad v_{n-1}[\![e_n]\!]v_n \quad \neg\,(0 \le r_1 \wedge \ \ldots \ \wedge 0 \le r_n) \quad v_n[\![\textbf{throw}\ new\ NegativeArraySizeException();]\!]v_0}{v[\![new\ ty[e_1]\ldots[e_n][n]]\!]v_0} \tag{6.48}$$

Every array type has its own class object, i.e. if an array of type int[][] is used in a Java program, exactly one class object for this type exists. All class objects should be present in the store.

array initializer

$\{e_1, \ldots, e_n\}$. Note: An array initializer can occur only on the right hand side of a variable or field declaration. If the evaluation of the expressions raises an exception there is no way to access the variable or field: In case of a static field an ExceptionInInitializerError (and afterwards a NoClassDefFoundError) is thrown, das Feld zugreift), in case of an instance field no object is created, and a local variable is no longer visible if the exception is caught. Note also that an array initializer cannot cause an ArrayStoreException.

$$\frac{v[\![e_1]\!]v_1 \quad \ldots \quad v_{n-1}[\![e_n]\!]v_n \quad \neg\ ref \in st_n}{v[\![\{e_1, \ldots, e_n\}]\!](v_n)^{ref, addarray(ref, ty, r_1+\ldots+r_n, st_n)}_{r,st}} \tag{6.49}$$

explicit constructor invocation

In a correct Java program an *explicit constructor invocation* can occur only 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 **this** and adds the correct class name so that the call is $e.c(e_1, \ldots, e_n)$.

$$\frac{v[\![e]\!]v_0 \quad v_0[\![e_1]\!]v_1 \quad \ldots \quad v_{n-1}[\![e_n]\!]v_n \quad (v_n)^{r_1,\ldots,r_n,r_0}_{x_1,\ldots,x_n,this}[\![\alpha]\!]v_{n+1} \quad is\_return\_mode(st_{n+1}[mode])}{v[\![e.c(e_1, \ldots, e_n)]\!](v_{n+1})^{v_n(x_1),\ldots,v_n(x_n),v_n(this),v_n(this),st_{n+1}[mode][normal]}_{x_1,\ldots,x_n,this,r,st}} \tag{6.50}$$

α is the body of the constructor with formal parameters $x_1, \ldots, x_n$. We set the local variables to the values of the call arguments and **this** to the invoking reference r_0, and evaluate the constructor body. Afterward the old values for $x_1, \ldots, x_n, this$ are restored, the result r is bound to the invoking reference, and the mode is set back to normal (because α will end with a `return`). If the mode of st_{n+1} is not *return*, but an exception, $x_1, \ldots, x_n, this$ are still restored, but r and st remain unchanged.

$$\frac{v[\![e]\!]v_0 \quad v_0[\![e_1]\!]v_1 \quad \ldots \quad v_{n-1}[\![e_n]\!]v_n \quad (v_n)^{r_1,\ldots,r_n,r_0}_{x_1,\ldots,x_n,this}[\![\alpha]\!]v_{n+1} \quad \neg\ is\_return\_mode(st_{n+1}[mode])}{v[\![e.c(e_1, \ldots, e_n)]\!](v_{n+1})^{v_n(x_1),\ldots,v_n(x_n),v_n(this)}_{x_1,\ldots,x_n,this}} \tag{6.51}$$

static method invocation

$c.m(e_1, \ldots, e_n)$, c is the class name of the method, m the method name (i.e. the *methodspec*, the method name and formal argument types, because reference widening conversion can occur). We assume that every execution path ends with a **return**.

$$\frac{v[\![e_1]\!]v_1 \quad \ldots \quad v_{n-1}[\![e_n]\!]v_n \quad (v_n)^{r_1,\ldots,r_n}_{x_1,\ldots,x_n}[\![\alpha]\!]v_0 \quad is\_return\_mode(st_0[mode])}{v[\![c.m(e_1, \ldots, e_n)]\!](v_0)^{v_n(x_1),\ldots,v_n(x_n),st_0[mode].val,st_0[mode][normal]}_{x_1,\ldots,x_n,r,st}} \tag{6.52}$$

α is the body of the method with formal parameters $x_1, \ldots, x_n$. We set the local variables to the values of the call arguments and evaluate the method body. Afterward the old values for $x_1, \ldots, x_n$ are restored, the result r is bound to the return value stored in the mode ($st_0[mode].val$), and the

mode is set back to normal (because α will end with a `return`). If the mode of st_0 is not *return*, but an exception, $x_1, \ldots, x_n$ are still restored, but r and st remain unchanged.

$$\frac{v[\![e_1]\!]v_1 \quad \ldots \quad v_{n-1}[\![e_n]\!]v_n \quad (v_n)^{r_1,\ldots,r_n}_{x_1,\ldots,x_n}[\![\alpha]\!]v_0 \quad \neg\, is\_return\_mode(st_0[mode])}{v[\![c.m(e_1,\ldots,e_n)]\!](v_0)^{v_n(x_1),\ldots,v_n(x_n)}_{x_1,\ldots,x_n}} \tag{6.53}$$

(instance) method invocation

See JLS 15.11.4. For an instance method $e.m(e_1, \ldots, e_n)$ the expression e and the arguments are evaluated. e must be a reference $\neq null$. The *accessibility* (JLS 15.11.4.3) is guaranteed since we do not consider dynamic loading et. al. Then the correct method body is searched (dynamic method lookup depending on the invocation *kind*). Every execution path must end with a **return**.

$$\frac{\begin{array}{c} v[\![e]\!]v_0 \quad v_0[\![e_1]\!]v_1 \quad \ldots \quad v_{n-1}[\![e_n]\!]v_n \quad r_0 \neq null \\ (v_n)^{r_1,\ldots,r_n,r_0}_{x_1,\ldots,x_n,this}[\![\alpha]\!]v_{n+1} \quad is\_return\_mode(st_{n+1}[mode]) \end{array}}{v[\![e.m(e_1,\ldots,e_n)]\!](v_{n+1})^{v_n(x_1),\ldots,v_n(x_n),v_n(this),st_{n+1}[mode].val,st_{n+1}[mode][normal]}_{x_1,\ldots,x_n,this,r,st}} \tag{6.54}$$

α is the correct body of the method with formal parameters $x_1, \ldots, x_n$. We set the local variables to the values of the call arguments, **this** to the invoking reference, and evaluate the method body. Afterward the old values for $x_1, \ldots, x_n, this$ are restored, the result r is bound to the return value stored in the mode ($st_0[mode].val$), and the mode is set back to normal (because α will end with a `return`). If the mode of st_0 is not *return*, but an exception, $x_1, \ldots, x_n, this$ are still restored, but r and st remain unchanged.

$$\frac{\begin{array}{c} v[\![e]\!]v_0 \quad v_0[\![e_1]\!]v_1 \quad \ldots \quad v_{n-1}[\![e_n]\!]v_n \quad r_0 \neq null \\ (v_n)^{r_1,\ldots,r_n,r_0}_{x_1,\ldots,x_n,this}[\![\alpha]\!]v_{n+1} \quad \neg\, is\_return\_mode(st_{n+1}[mode]) \end{array}}{v[\![e.m(e_1,\ldots,e_n)]\!](v_{n+1})^{v_n(x_1),\ldots,v_n(x_n),v_n(this)}_{x_1,\ldots,x_n,this}} \tag{6.55}$$

In case the invoking expression is null:

$$\frac{v[\![e]\!]v_0 \quad v_0[\![e_1]\!]v_1 \ldots v_{n-1}[\![e_n]\!]v_n \quad r_0 = null \quad v_n[\![\textbf{throw}\ new\ NullPointerEx();]\!]v_{n+1}}{v[\![e.m(e_1,\ldots,e_n)]\!]v_{n+1}} \tag{6.56}$$

6.2 Semantic of Statements

Statements cannot cause a *first active use* (i.e. the initialization state of classes is irrelevant), but may catch jumps. These are the *labeled statement*, *try (catch finally)* and the additionally introduced statements. We do not consider **continue**, or **break** without label. (Otherwise the iteration statements could catch jump, too.) Statements have no value, so the special variable r is not needed. The evaluation of expressions may change the value of r. We reset r after the evaluation of a statement to its original value. This means that the value of r remains unchanged. (Otherwise we would have a problem with the correctness of our calculus.) We omit this modification to the result state in the following rules.

6.2.1 Java Statements

blocks

A block $\{\alpha_1 \ldots \alpha_n\}$ 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. (Java forbids the hiding of local variables, but in our state every variable always has a value.)

$$\frac{v[\![\alpha_1]\!]v_1 \ldots v_{n-1}[\![\alpha_n]\!]v_n}{v[\![\{\alpha_1 \ldots \alpha_n\}]\!](v_n)^{v(\underline{x})}_{\underline{x}}} \tag{6.57}$$

$\underline{x}$ are the local variables declared in the block. See JLS 4.5.3 (item 7) for a speciality concerning the switch statement.

local variable declaration

$ty\ x = e;$ We assume that every variable has an explicit initialization. Since the end of the surrounding block restores the old value of the variable we can simply set the variable to its new value.

$$\frac{v[\![x = e]\!]v_0}{v[\![ty\ x = e;]\!]v_0} \tag{6.58}$$

The semantic of a local variable declaration is reduced to the semantic of a local variable assignment. If $m \neq normal$ then also $m_0 \neq normal$. This means the declaration is evaluated only if not in a jump.

expression statement

$e;$ is just evaluated unless in a jump. The result of e is simply discarded.

$$\frac{v[\![e]\!]v_0}{v[\![e;]\!]v_0} \tag{6.59}$$

if statement

We get three rules:

$$\frac{v[\![e]\!]v_0 \quad m_0 \neq normal}{v[\![\textbf{if}\ (e)\ \alpha_1\ \textbf{else}\ \alpha_2]\!]v_0} \tag{6.60}$$

$$\frac{v[\![e]\!]v_0 \quad m_0 = normal \wedge r_0 = \text{true} \quad v_0[\![\alpha_1]\!]v_1}{v[\![\textbf{if}\ (e)\ \alpha_1\ \textbf{else}\ \alpha_2]\!]v_1} \tag{6.61}$$

$$\frac{v[\![e]\!]v_0 \quad m_0 = normal \wedge r_0 = \text{false} \quad v_0[\![\alpha_2]\!]v_1}{v[\![\textbf{if}\ (e)\ \alpha_1\ \textbf{else}\ \alpha_2]\!]v_1} \tag{6.62}$$

labeled statement

$l : \alpha$ Labeled statements with the same label may not be nested, but in different blocks the same labels can be used. A label l catches the jump $st[mode] = break(l)$ and ends normal. Otherwise nothing happens. We get two rules:

$$\frac{v[\![\alpha]\!]v_0 \quad m \neq normal \vee m_0 \neq break(l)}{v[\![l : \alpha]\!]v_0} \tag{6.63}$$

$$\frac{v[\![\alpha]\!]v_0 \quad m = normal \wedge m_0 = break(l)}{v[\![l : \alpha]\!](v_0)^{st[mode][normal]}_{st}} \tag{6.64}$$

while loop

$$\frac{v[\![e]\!]v_0 \quad m_0 \neq normal \vee r_0 = \text{false}}{v[\![\textbf{while}\ (e)\ \textbf{do}\ \alpha]\!]v_0} \tag{6.65}$$

$$\frac{v[\![e]\!]v_0 \quad m_0 = normal \wedge r_0 = \text{true} \quad v_0[\![\textbf{while}\ (e)\ \textbf{do}\ \alpha]\!]v_1}{v[\![\textbf{while}\ (e)\ \textbf{do}\ \alpha]\!]v_1} \tag{6.66}$$

This inductive definition is well defined even if the loop does not terminate. In this case the relation for the statement is empty, i.e. there are no v, v_1 such that $v[\![\textbf{while}\ (e)\ \textbf{do}\ \alpha]\!]v_1$.

do loop

$$\frac{v[\![\alpha]\!]v_0 \quad v_0[\![e]\!]v_1 \quad m_1 \neq normal \vee r_1 = \text{false}}{v[\![\textbf{do } \alpha \textbf{ while } (e);]\!]v_1} \tag{6.67}$$

If $m \neq normal$ or the evaluation of α or e causes a jump the do loop is terminated.

$$\frac{v[\![\alpha]\!]v_0 \quad v_0[\![e]\!]v_1 \quad m_1 = normal \wedge r_1 = \text{true} \quad v_1[\![\textbf{do } \alpha \textbf{ while } (e);]\!]v_2}{v[\![\textbf{do } \alpha \textbf{ while } (e);]\!]v_2} \tag{6.68}$$

for loop

Our for loop has no variable initialization, but only a termination test e and updates $e_1, \ldots, e_n$.

$$\frac{v[\![e]\!]v_0 \quad m_0 \neq normal \vee r_0 = \text{false}}{v[\![\textbf{for}(e; e_1, \ldots, e_n)\ \alpha]\!]v_0} \tag{6.69}$$

$$\frac{v[\![e]\!]v_0 \qquad m_0 = normal \wedge r_0 = \text{true} \quad v_0[\![\alpha\ e_1; \ldots e_n; \textbf{for}(e; e_1, \ldots, e_n)\ \alpha]\!]v_2}{v[\![\textbf{for}(e; e_1, \ldots, e_n)\ \alpha]\!]v_2} \tag{6.70}$$

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 for loop will be terminated.

switch statement

$$\frac{v[\![e]\!]v_0 \quad m_0 = normal \quad v_0[\![cases']\!]v_1}{v[\![switch(e)\ \{cases\}]\!]v_1} \tag{6.71}$$

Here is $cases' = findmatchingcases(r_0, cases)$ that selects the correct case and deletes all labels. The second rule is for $m_0 \neq normal$.

$$\frac{v[\![e]\!]v_0 \quad m_0 \neq normal}{v[\![switch(e)\{cases\}]\!]v_0} \tag{6.72}$$

break statement

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

$$\frac{m \neq normal}{v[\![break(l)]\!]v} \tag{6.73}$$

$$\frac{m = normal}{v[\![break(l)]\!]v_{st}^{st[mode][break(l)]}} \tag{6.74}$$

continue statement

We do not consider the **continue** statement.

return statement

A return statement with an expression evaluates the expression and sets the mode to return unless the current mode is already a jump.

$$\frac{v[\![e]\!]v_0 \quad m_0 = normal}{v[\![return\ e;]\!](v_0)_{st}^{st[mode][return(r_0)]}} \tag{6.75}$$

$$\frac{v[\![e]\!]v_0 \quad m_0 \neq normal}{v[\![return\ e;]\!]v_0} \tag{6.76}$$

The result value of e is stored in the mode. An empty return works in the same manner.

$$\frac{m = normal}{v[\![return;]\!]v_{st}^{st_0[mode][return]}} \tag{6.77}$$

$$\frac{m \neq normal}{v[\![return\ e;]\!]v} \tag{6.78}$$

throw statement

If the evaluation of e yields $r_0 = null$, a NullPointerException is thrown (this case is missing in JLS 14.16). Otherwise a throw works like a return.

$$\frac{v[\![e]\!]v_0 \quad m_0 \neq normal}{v[\![throw\ e;]\!]v_0} \tag{6.79}$$

$$\frac{v[\![e]\!]v_0 \quad m_0 = normal \wedge r_0 = null \quad v_0[\![\textbf{throw}\ new\ NullPointerEx();]\!]v_1}{v[\![throw\ e;]\!]v_1} \tag{6.80}$$

$$\frac{v[\![e]\!]v_0 \quad m_0 = normal \wedge r_0 \neq null}{v[\![throw\ e;]\!](v_0)_{st}^{st_0[mode][throw(r_0)]}} \tag{6.81}$$

try statement

The statement is skipped if the mode is not *normal*.

$$\frac{m \neq normal}{v[\![\textbf{try}\ \alpha\ catches\ \textbf{finally}\ \alpha_0]\!]v} \tag{6.82}$$

If the try block does not end with a throw or with a throw that has no handler, the finally block is executed. If the finally block ends normally the original mode is restored:

$$\frac{v[\![\alpha]\!]v_0 \quad (v_0)_{st}^{st_0[mode][normal]}[\![\alpha_0]\!]v_1}{v[\![\textbf{try}\ \alpha\ catches\ \textbf{finally}\ \alpha_0]\!](v_1)_{st}^{st_1[mode][m_0]}} \quad \left\langle \begin{array}{l} m = normal, \\ \neg\ throw(m_0) \vee \\ nocatcher(m_0.val), \\ m_1 = normal \end{array} \right. \tag{6.83}$$

If the try block does not end with a throw or with a throw that has no handler, the finally block is executed. If the finally block does not end normal, this mode is kept:

$$\frac{v[\![\alpha]\!]v_0 \quad (v_0)_{st}^{st_0[mode][normal]}[\![\alpha_0]\!]v_1}{v[\![\textbf{try}\ \alpha\ catches\ \textbf{finally}\ \alpha_0]\!]v_1} \quad \left\langle \begin{array}{l} m = normal, \\ \neg\ throw(m_0) \vee nocatcher(m_0.val), \\ m_1 \neq normal \end{array} \right. \tag{6.84}$$

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 correct catch clause for the thrown reference ($\alpha_c = catcher(m_0.val)$).

$$\frac{v[\![\alpha]\!]v_0 \quad (v_0)_m^{normal}[\![\alpha_c]\!]v_1 \quad (v_1)_m^{normal}[\![\alpha_0]\!]v_2}{v[\![\textbf{try}\ \alpha\ catches\ \textbf{finally}\ \alpha_0]\!](v_2)_{st}^{st_2[mode][m_1]}} \quad \left\langle \begin{array}{l} m = normal, \\ throw(m_0), \\ m_2 = normal \end{array} \right. \tag{6.85}$$

$$\frac{v[\![\alpha]\!]v_0 \quad (v_0)_{st}^{st_0[mode][normal]}[\![\alpha_c]\!]v_1 \quad (v_1)_{st}^{st_1[mode][normal]}[\![\alpha_0]\!]v_2}{v[\![\textbf{try}\ \alpha\ catches\ \textbf{finally}\ \alpha_0]\!]v_2} \quad \left\langle \begin{array}{l} m = normal, \\ throw(m_0), \\ m_2 \neq normal \end{array} \right. \tag{6.86}$$

The 'corresponding' catch block is the first block with an assignment compatible type to the throw type. The declaration is transformed into a local variable declaration.

6.2.2 Additional Statements

The calculus introduces additional statements to deal with blocks, jumps, and class initialization.

static

static(c) handles the initialization of the super classes:

$$\frac{m \neq normal \vee c = Object \vee initdone(super(c))}{v[\![\textbf{static}(c)]\!]v} \tag{6.87}$$

$super(c)$ is the super class of c.

$$\frac{\begin{array}{c} m = normal \wedge c \neq Object \wedge initundone(super(c)) \\ v_{st}^{addclass(super(c),fis,st)}[\![\textbf{static}(super(c)); \alpha; \textbf{endstatic}(super(c))]\!]v_0 \end{array}}{v[\![static(c)]\!]v_0} \tag{6.88}$$

α is the static initializer of $super(c)$. *addclass* adds a class with its static fields to the store.

$$\frac{\begin{array}{c} m = normal \wedge c \neq Object \wedge initerror(super(c)) \\ v[\![\textbf{throw new } \text{ClassDefNotFoundError}()]\!]v_0 \end{array}}{v[\![static(c)]\!]v_0} \tag{6.89}$$

static(c) works like a first active use for the superclass of c, but has no other effect.

endstatic

endstatic(c) catches exceptions. If during static initialization an exception or error occurs the class object is marked 'erroneous', and an exception is transformed in an *ExceptionInInitializerError*. Otherwise nothing happens.

$$\frac{\neg\, throw(m)}{v[\![endstatic(c)]\!]v} \tag{6.90}$$

$$\frac{\begin{array}{c} throw(m) \wedge is\_exception(m.val) \\ v_{st}^{st[mode][normal]}[\![\textbf{throw new } \text{ExceptionInInitializerError}()]\!]v_0 \end{array}}{v[\![endstatic(c)]\!]v_0} \tag{6.91}$$

$is\_exception(m.val)$ is true if the throw reference $m.val$ is of a subclass of Exception.

$$\frac{throw(m) \wedge \neg\, is\_exception(m.val)}{v[\![endstatic(c)]\!]v_{st}^{st[\text{jvmref}-\text{c.initstate}][\text{initerror}]}} \tag{6.92}$$

If an error was thrown the class is marked 'error' and the mode remains unchanged.

targetexpr

targetexpr(x) catches returns and sets x to the returned value $m.val$.

$$\frac{is\_returnexpr(m)}{v[\![\textbf{targetexpr}(x)]\!]v_{x,st}^{m.val,st[mode][normal]}} \tag{6.93}$$

$$\frac{\neg\, is\_returnexpr(m)}{v[\![\textbf{targetexpr}(x)]\!]v} \tag{6.94}$$

target

target(mo) catches jumps with mode *mo* (empty returns or breaks), and does nothing otherwise.

$$\frac{m = mo}{v[\![target(mo)]\!]v_{st}^{st[mode][normal]}} \tag{6.95}$$

$$\frac{m \neq mo}{v[\![target(mo)]\!]v} \tag{6.96}$$

catches

A list of catch clauses catches exceptions (or errors). It is necessary to keep all clauses in one list, because otherwise exceptions in the body of a clause could be caught by one of the following clauses.

$$\frac{throw(m) \wedge catches(m.val) \quad v_{st}^{st[mode][normal]}[\![\alpha_c]\!]v_0}{v[\![catches]\!]v_0} \tag{6.97}$$

α_c is the correct catcher from the list *catches* for the thrown reference $m.val$.

$$\frac{\neg\, throw(m) \vee nocatcher(m.val)}{v[\![catches]\!]v} \tag{6.98}$$

finally

A **finally** block catches all jumps.

$$\frac{v_{st}^{st[mode][normal]}[\![\alpha]\!]v_0}{v[\![\textbf{finally } \alpha]\!]v_0} \tag{6.99}$$

endfinally

endfinally(mo) raises a jump with mode *mo* if the initial mode is *normal*. This statement is used to re-raise the mode at the end of a finally block.

$$\frac{m = normal}{v[\![\textbf{endfinally}(mo)]\!]v_{st}^{st[mode][mo]}} \tag{6.100}$$

$$\frac{m \neq normal}{v[\![\textbf{endfinally}(mo)]\!]v} \tag{6.101}$$

Chapter 7

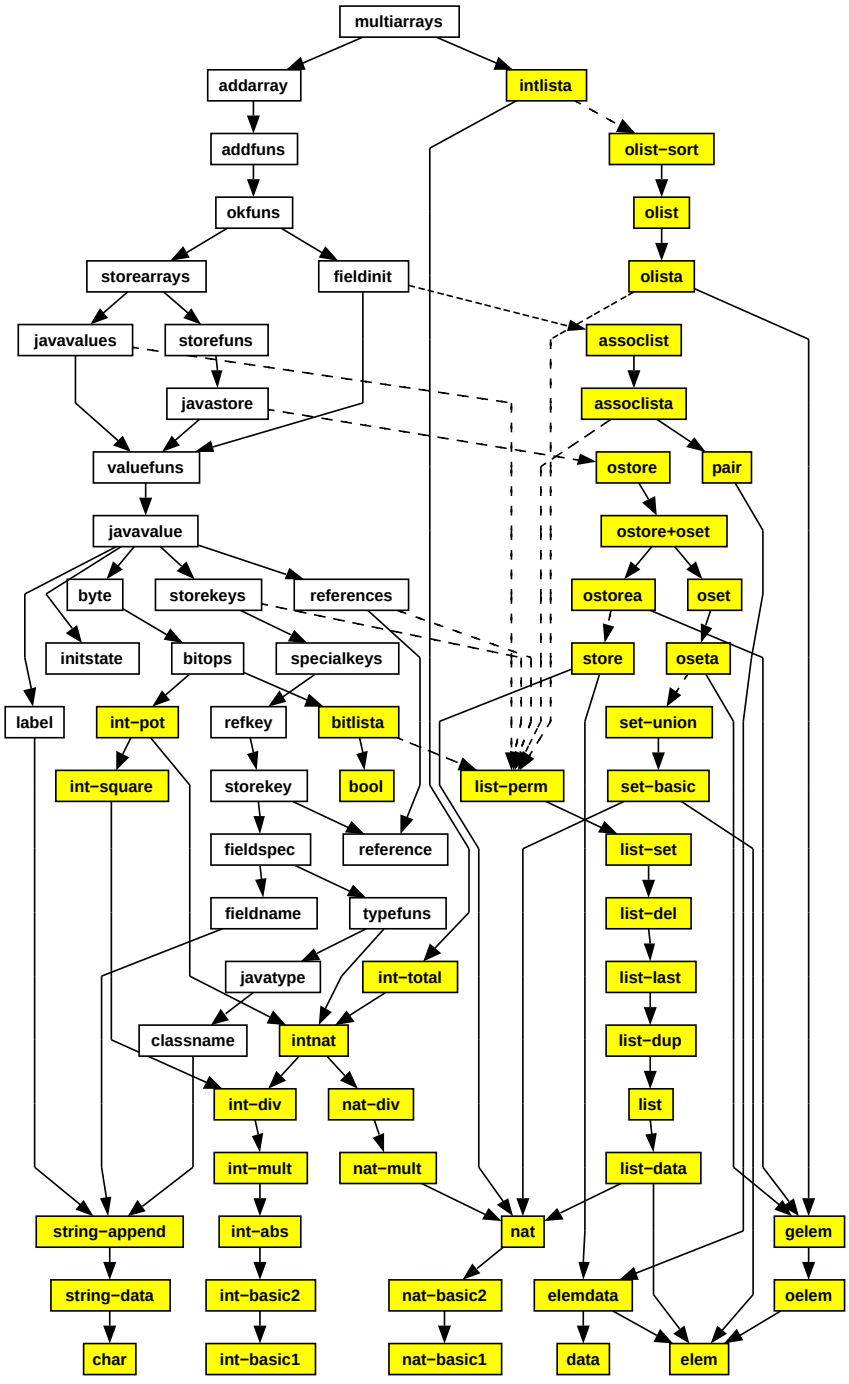
The Specifications

This chapter contains without further comment the algebraic specifications the calculus is based on. (Details about the algebraich specifications used in KIV can be found e.g. in [RSSB98][Rei95] and in the KIV documentation[1].) These specifications contain the exact definitions of *newref*, *addclass*, *addobj*, etc. The structure of the specification is shown on the next page. The yellow boxes are library specifications, i.e. specifications not specific for our calculus. The white boxes contain the specific Java specifications.

The java store is a normal store with special keys; it is built in *javastore*. *addmultiarray* can be found in *multiarray*, *addarray* in *addarray*, *addclass*, *addobj* in *addfuns*, most of the other functions (e.g. *newref*, *initerror*, *initdone*, *initundone*) in *storefuns*. The bitwise operations on integers are specified in *bitops* (their specification is more complicated than one would expect), bytes and shorts in *byte*.

A list with all symbols and the page with their specification can be found at the end on p. 115. The following list shows the page on which each specification can be found.

[1] http://www.Informatik.Uni-Augsburg.DE/swt/fmg/tools/kiv/description.html



Structure of the specifications

multiarrays =
enrich intlista, addarray **with**
functions

addarraymult	:	reference × javatype × store × intlist × intlist × int	→	store	;
addarraymultlist	:	reference × javatype × store × references × intlist × intlist × int	→	store	;
countrefs	:	intlist	→	int	;
$\#_{ints}$	:	intlist	→	nat	;

axioms

intssize-empty : $\#_{ints}$(@) = 0;

intssize-rec : $\#_{ints}$(i ' + ints) = 1 + i→n(i) + $\#_{ints}$(ints);

countrefs_one : # ints ≤ 1 → countrefs(ints) = 0;

countrefs_more_le0 : # ints > 1 ∧ ints.first ≤ 0 → countrefs(ints) = 0;

countrefs_more_gr0 :
ints > 1 ∧ ints.first > 0
→ countrefs(ints) = ints.first + ints.first ∗ countrefs(ints.rest);

addarraymultlist_zero :
ints = 0 → addarraymultlist(r, ty, st, refs, ints, ints$_0$, i) = st;

addarraymultlist_one_0 :
ints = 1 ∧ ints.first ≤ 0
→ addarraymultlist(r, ty, st, refs, ints, ints$_0$, i)
= st[r – _type, typeval(mktype_from_dims(ty, n→i(# ints) + i))][r – _length, intval(ints$_0$.first)];

addarraymultlist_one_not_0 :
ints = 1 ∧ ints.first > 0
→ addarraymultlist(r, ty, st, refs, ints, ints$_0$, i)
= addarray(r, mktype_from_dims(ty, i), ints$_0$.first, st);

addarraymultlist_more_0 :
ints > 1 ∧ ints.first ≤ 0
→ addarraymultlist(r, ty, st, refs, ints, ints$_0$, i)
= st[r – _type, typeval(mktype_from_dims(ty, n→i(# ints) + i))][r – _length, intval(ints$_0$.first)];

addarraymultlist_more_2 :
ints > 1 ∧ ints.first ≥ 1 + 1
→ addarraymultlist(r, ty, st, refs, ints, ints$_0$, i)
= addarraymultlist(refs.first, ty, addarraymultlist(r, ty, st[r – (ints.first - 1)', refval(refs.first)][r – _type, typeval(mktype_from_dims(ty, i + n→i(# ints)))][r – _length, intval(ints$_0$.first)], refs.rest, (ints.first - 1) + ints.rest, ints$_0$, i), restn(i→n(countrefs((ints.first - 1) + ints.rest)), refs.rest), ints.rest, ints$_0$.rest, i);

addarraymultlist_more_1 :
ints > 1 ∧ ints.first = 1
→ addarraymultlist(r, ty, st, refs, ints, ints$_0$, i)
= addarraymultlist(refs.first, ty, st[r – (ints.first - 1)', refval(refs.first)][r – _type, typeval(mktype_from_dims(ty, i + n→i(# ints)))][r – _length, intval(ints$_0$.first)], refs.rest, ints.rest, ints$_0$.rest, i);

end enrich

---

addarray =
enrich addfuns **with**
functions

addarray	:	reference × javatype × int × store	→	store	;
addarray	:	reference × javavalue × int × store	→	store	;
addarray	:	reference × javatype × javavalues × store	→	store	;
addarray	:	reference × javavalues × store	→	store	;

axioms

addarray-def :
addarray(r, ty, i, st)
= addarray(r, initial_value(ty), i, st[r – _type, typeval(mkarraytype(ty))][r – _length, intval(i)]);

addarray-base : i ≤ 0 → addarray(r, val, i, st) = st;

addarray-rec :
i > 0 → addarray(r, val, i, st) = addarray(r, val, i - 1, st)[r – (i - 1)', val];

addarrayinit-def :
addarray(r, ty, vals, st)
= addarray(r, vals, st[r – _type, typeval(mkarraytype(ty))][r – _length, intval(n→i(# vals))]);

addarrayinit-base : addarray(r, @, st) = st;

addarrayinit-rec :
addarray(r, vals + val ', st) = addarray(r, vals, st)[r – n→i(# vals)', val];

end enrich

---

addfuns =
enrich okfuns **with**
functions

addobj	:	reference × classname × fieldinits × store	→	store	;
addclass	:	classname × fieldinits × store	→	store	;

predicates is_obj : reference × classname × storekeys × store;

axioms

addobj-base : addobj(r, $class_1$, @, st) = st[r – _type, typeval(mkclasstype($class_1$))];

addobj-rec :
addobj(r, $class_1$, (sk × val)' + fis, st) = addobj(r, $class_1$, fis, st)[r – sk, val];

addclass-base :
addclass($class_1$, @, st)
= st[jvmref – mkfs($class_1$, void_type, "initstate".field)', initval(done)];

addclass-yes :
addclass($class_1$, (mkfs($class_1$, ty, fieldvar)' × val)' + fis, st)
= addclass($class_1$, fis, st)[jvmref – mkfs($class_1$, ty, fieldvar)', val];

addclass-no1 :
$class_1 \neq class_2$
→ addclass($class_1$, (mkfs($class_2$, ty, fieldvar)' × val)' + fis, st)
= addclass($class_1$, fis, st);

addclass-no2 : addclass($class_1$, (i ' × val)' + fis, st) = addclass($class_1$, fis, st);

is_obj :

is_obj(r, class$_1$, sks, st)
$\leftrightarrow$ r $\neq$ jvmref
$\wedge$ r – _type $\times$ typeval(mkclasstype(class$_1$)) $\in$ st
$\wedge$ goodfieldsandtypes(r, sks, st)
$\wedge$ ($\forall$ sk. r – sk $\in$ st $\rightarrow$ sk = _type $\vee$ sk $\in$ sks);

end enrich

---

okfuns =
enrich fieldinit, storearrays **with**
predicates

eqref	:	reference $\times$ reference $\times$ store $\times$ store;
eqval	:	javavalue $\times$ javavalue $\times$ store $\times$ store;
okstore	:	store;
okclass	:	javavalue $\times$ classname $\times$ store;
okreftype	:	javavalue $\times$ javatype $\times$ store;
oktype	:	javavalue $\times$ javatype $\times$ store;
goodfieldandtype	:	reference $\times$ storekey $\times$ store;
goodfieldsandtypes	:	reference $\times$ storekeys $\times$ store;
. $\in$.	:	fieldinits $\times$ store;
okarray	:	reference $\times$ javatype $\times$ store;
okarray	:	javavalue $\times$ javatype $\times$ int $\times$ store;
okarraytype	:	javavalue $\times$ javatype $\times$ store;
okarrays	:	reference $\times$ fieldinits $\times$ store;

axioms

okstore :
okstore(st)
$\leftrightarrow$ _mode $\in$ st
$\wedge$ _out $\in$ st
$\wedge$ ($\forall$ r. r $\neq$ jvmref $\wedge$ r $\in$ st $\rightarrow$ r – _type $\in$ st $\wedge$ is_typevalue(st[r – _type]));

okclass :
okclass(val, class$_1$, st)
$\leftrightarrow$ reftypep(val, st)
$\wedge$ (val.val $\neq$ jvmref
$\rightarrow$ val.val – _type $\in$ st
$\wedge$ is_typevalue(st[val.val – _type])
$\wedge$ st[val.val – _type].type $\leq$ mkclasstype(class$_1$));

okreftype :
okreftype(val, ty, st)
$\leftrightarrow$ reftypep(val, st)
$\wedge$ (val.val $\neq$ jvmref
$\rightarrow$ val.val – _type $\in$ st
$\wedge$ is_typevalue(st[val.val – _type])
$\wedge$ st[val.val – _type].type $\leq$ ty);

good-base : goodfieldsandtypes(r, @, st);

good-rec :
goodfieldsandtypes(r, sk ' + sks, st)
$\leftrightarrow$ goodfieldandtype(r, sk, st) $\wedge$ goodfieldsandtypes(r, sks, st);

good-field :
goodfieldandtype(r, mkfs(class$_1$, ty, fieldvar)', st)

$\leftrightarrow$ r – mkfs(class$_1$, ty, fieldvar)' $\in$ st
$\wedge$ oktype(st[r – mkfs(class$_1$, ty, fieldvar)'], ty, st);

good-index : $\neg$ goodfieldandtype(r, i ', st);

oktype-bool : oktype(val, boolean_type, st) $\leftrightarrow$ is_boolvalue(val);

oktype-int : oktype(val, int_type, st) $\leftrightarrow$ is_integervalue(val);

oktype-short : oktype(val, short_type, st) $\leftrightarrow$ is_shortvalue(val);

oktype-byte : oktype(val, byte_type, st) $\leftrightarrow$ is_bytevalue(val);

oktype-class : oktype(val, mkclasstype(class$_1$), st) $\leftrightarrow$ okclass(val, class$_1$, st);

oktype-array : oktype(val, mkarraytype(ty), st) $\leftrightarrow$ okreftype(val, mkarraytype(ty), st);

oktype-void : oktype(val, void_type, st);

oktype-abstract : oktype(val, abstract_type, st);

instore-base : @ $\in$ st;

instore-rec : (sk $\times$ val)' + fis $\in$ st $\leftrightarrow$ jvmref – sk $\times$ val $\in$ st $\wedge$ fis $\in$ st;

okarraytype :
okarraytype(val, ty, st)
$\leftrightarrow$ is_referencevalue(val) $\wedge$ (val.val $\neq$ jvmref $\rightarrow$ okarray(val.val, ty, st));

okarray :
okarray(r, ty, st)
$\leftrightarrow$ okreftype(refval(r), mkarraytype(ty), st)
$\wedge$ r $\neq$ jvmref
$\wedge$ r – _length $\in$ st
$\wedge$ is_integervalue(st[r – _length])
$\wedge$ 0 $\leq$ st[r – _length].val
$\wedge$ is_arrayref(r, st[r – _length].val, st)
$\wedge$ ($\forall$ j. 0 $\leq$ j $\wedge$ j $<$ st[r – _length].val $\rightarrow$ oktype(st[r – j '], ty, st));

okarray :
okarray(val, ty, i, st)
$\leftrightarrow$ okreftype(val, mkarraytype(ty), st)
$\wedge$ val.val $\neq$ jvmref
$\wedge$ val.val – _length $\times$ intval(i) $\in$ st
$\wedge$ is_arrayref(val.val, i, st)
$\wedge$ ($\forall$ j. 0 $\leq$ j $\wedge$ j $<$ i
$\rightarrow$ oktype(st[val.val – j '], ty, st)
$\wedge$ ($\forall$ sk. val.val – sk $\in$ st
$\rightarrow$ sk = _type
$\vee$ sk = _length
$\vee$ is_indexkey(sk) $\wedge$ 0 $\leq$ sk.index $\wedge$ sk.index $<$ i));

okarrays-base : okarrays(r, @, st);

okarrays-rec :
okarrays(r, (sk $\times$ val)' + fis, st)
$\leftrightarrow$ goodfieldandtype(r, sk, st)
$\wedge$ is_integervalue(val)
$\wedge$ okarray(st[r – sk], sk.fs.type.type, val.val, st)
$\wedge$ okarrays(r, fis, st);

eqref :
eqref(r, r$_0$, st, st$_0$)
$\leftrightarrow$ r = jvmref $\wedge$ r$_0$ = jvmref

$\vee \quad r \neq \text{jvmref}$
$\quad \wedge\ r_0 \neq \text{jvmref}$
$\quad \wedge\ (\forall\ sk. \quad (r - sk \in st \leftrightarrow r_0 - sk \in st_0)$
$\qquad \wedge\ (r - sk \in st \rightarrow \text{eqval}(st[r - sk], st_0[r_0 - sk], st, st_0)));$

eqval-intl : $\text{is\_integervalue}(val) \rightarrow (\text{eqval}(val, val_0, st, st_0) \leftrightarrow val = val_0);$

eqval-intr : $\text{is\_integervalue}(val_0) \rightarrow (\text{eqval}(val, val_0, st, st_0) \leftrightarrow val = val_0);$

eqval-bytel : $\text{is\_bytevalue}(val) \rightarrow (\text{eqval}(val, val_0, st, st_0) \leftrightarrow val = val_0);$

eqval-byter : $\text{is\_bytevalue}(val_0) \rightarrow (\text{eqval}(val, val_0, st, st_0) \leftrightarrow val = val_0);$

eqval-shortl : $\text{is\_shortvalue}(val) \rightarrow (\text{eqval}(val, val_0, st, st_0) \leftrightarrow val = val_0);$

eqval-shortr : $\text{is\_shortvalue}(val_0) \rightarrow (\text{eqval}(val, val_0, st, st_0) \leftrightarrow val = val_0);$

eqval-booll : $\text{is\_boolvalue}(val) \rightarrow (\text{eqval}(val, val_0, st, st_0) \leftrightarrow val = val_0);$

eqval-boolr : $\text{is\_boolvalue}(val_0) \rightarrow (\text{eqval}(val, val_0, st, st_0) \leftrightarrow val = val_0);$

eqval-refl :
$\text{is\_referencevalue}(val)$
$\rightarrow (\quad \text{eqval}(val, val_0, st, st_0)$
$\quad \leftrightarrow \text{is\_referencevalue}(val_0) \wedge \text{eqref}(val.val, val_0.val, st, st_0));$

eqval-refr :
$\text{is\_referencevalue}(val_0)$
$\rightarrow (\text{eqval}(val, val_0, st, st_0) \leftrightarrow \text{is\_referencevalue}(val) \wedge \text{eqref}(val.val, val_0.val, st, st_0));$

eqval-concl :
$\text{is\_valuelist}(val)$
$\rightarrow (\quad \text{eqval}(val, val_0, st, st_0)$
$\quad \leftrightarrow \quad \text{is\_valuelist}(val_0)$
$\qquad \wedge\ \text{eqval}(val.firstval, val_0.firstval, st, st_0)$
$\qquad \wedge\ \text{eqval}(val.restval, val_0.restval, st, st_0));$

eqval-concr :
$\text{is\_valuelist}(val_0)$
$\rightarrow (\quad \text{eqval}(val, val_0, st, st_0)$
$\quad \leftrightarrow \quad \text{is\_valuelist}(val)$
$\qquad \wedge\ \text{eqval}(val.firstval, val_0.firstval, st, st_0)$
$\qquad \wedge\ \text{eqval}(val.restval, val_0.restval, st, st_0));$

eqval-novall : $\text{is\_novalue}(val) \rightarrow (\text{eqval}(val, val_0, st, st_0) \leftrightarrow val = val_0);$

eqval-novalr : $\text{is\_novalue}(val_0) \rightarrow (\text{eqval}(val, val_0, st, st_0) \leftrightarrow val = val_0);$

eqval-typel : $\text{is\_typevalue}(val) \rightarrow (\text{eqval}(val, val_0, st, st_0) \leftrightarrow val = val_0);$

eqval-typer : $\text{is\_typevalue}(val_0) \rightarrow (\text{eqval}(val, val_0, st, st_0) \leftrightarrow val = val_0);$

eqval-initl : $\text{is\_initvalue}(val) \rightarrow (\text{eqval}(val, val_0, st, st_0) \leftrightarrow val = val_0);$

eqval-initr : $\text{is\_initvalue}(val_0) \rightarrow (\text{eqval}(val, val_0, st, st_0) \leftrightarrow val = val_0);$

eqval-breakl : $\text{is\_break\_mode}(val) \rightarrow (\text{eqval}(val, val_0, st, st_0) \leftrightarrow val = val_0);$

eqval-breakr : $\text{is\_break\_mode}(val_0) \rightarrow (\text{eqval}(val, val_0, st, st_0) \leftrightarrow val = val_0);$

eqval-continuel : $\text{is\_continue\_mode}(val) \rightarrow (\text{eqval}(val, val_0, st, st_0) \leftrightarrow val = val_0);$

eqval-continuer : $\text{is\_continue\_mode}(val_0) \rightarrow (\text{eqval}(val, val_0, st, st_0) \leftrightarrow val = val_0);$

eqval-returnl : $\text{is\_return\_mode}(val) \rightarrow (\text{eqval}(val, val_0, st, st_0) \leftrightarrow val = val_0);$

eqval-returnr : $\text{is\_return\_mode}(val_0) \rightarrow (\text{eqval}(val, val_0, st, st_0) \leftrightarrow val = val_0);$

eqval-throwl : $\text{is\_throw\_mode}(val) \rightarrow (\text{eqval}(val, val_0, st, st_0) \leftrightarrow val = val_0);$

eqval-throwr : $\text{is\_throw\_mode}(val_0) \rightarrow (\text{eqval}(val, val_0, st, st_0) \leftrightarrow val = val_0);$

end enrich

---

fieldinit =
actualize assoclist **with** valuefuns **by morphism**
elem → storekey; data → javavalue; pair → fieldvalue; assoclist → fieldinits; @ → @; × → ×; .1 →.1; .2 →.2; + → +; .first →.first; .rest →.rest; # → #; ' →'; + → +; + → +; + → +; ++ → ++; rmdup → rmdup; .last →.last; .butlast →.butlast; rev → rev; mklist → mklist; -$_l$ → -$_l$; -1$_l$ → -1$_l$; -1$_l$ → -1$_l$;] →]; pos → pos;] →]; sublist → sublist; firstn → firstn; restn → restn; lastn → lastn; frome → frome; ∪ → ∪; \ → \; filter → filter; #$_{oc}$ → #$_{oc}$;] →];] →];] →]; < → <; ∈ → ∈; dups → dups; disj → disj; ⊑ → ⊑; ⊒ → ⊒; ⊆ → ⊆; perm → perm; ⊆$_m$ → ⊆$_m$; ∈ → ∈; unique_al → unique_al; a → sk; a$_0$ → sk$_2$; b → sk$_0$; c → sk$_1$; d → val; d$_0$ → val$_0$; d$_1$ → val$_1$; d$_2$ → val$_2$; p → fv; p$_0$ → fv$_0$; p$_1$ → fv$_1$; p$_2$ → fv$_2$; ax → fis; ay → fis$_0$; az → fis$_1$; ax$_0$ → fis$_2$; az$_0$ → fis$_{10}$; ay$_0$ → fis$_3$; ax$_1$ → fis$_4$; ay$_1$ → fis$_5$; az$_1$ → fis$_6$; ax$_2$ → fis$_7$; ay$_2$ → fis$_8$; az$_2$ → fis$_9$
end actualize

---

storearrays =
enrich storefuns, javavalues **with**

functions

arraycopy	:	reference × int × reference × int × int × store	→	store	;
getarrayv	:	reference × int × int × store	→	javavalue	;
getarrayv	:	reference × store	→	javavalue	;
getarray	:	reference × store	→	javavalues	;
getarray	:	reference × int × int × store	→	javavalues	;
intval	:	javavalues	→	int	;

predicates

is_arrayref	:	reference × int × store;
arraycomp	:	reference × int × reference × int × int × store;

axioms

is_arrayref : is_arrayref(r, i, st) ↔ (∀ j. 0 ≤ j ∧ j < i → r – j ' ∈ st);

arrayComp :
arraycomp(r$_1$, i$_1$, r$_2$, i$_2$, j, st)
↔ ∀ i. 0 ≤ i ∧ i < j → st[r$_1$ – (i$_1$ + i)'] = st[r$_2$ – (i$_2$ + i)'];

arrayCopy-base : j ≤ 0 → arraycopy(r$_1$, i$_1$, r$_2$, i$_2$, j, st) = st;

arrayCopy-rec :
0 < j
→ arraycopy(r$_1$, i$_1$, r$_2$, i$_2$, j, st)
= arraycopy(r$_1$, i$_1$, r$_2$, i$_2$, j - 1, st[r$_2$ – (i$_2$ + j - 1)', st[r$_1$ – (i$_1$ + j - 1)']]);

getArray-zero : j ≤ 0 → getarray(r, i, j, st) = @;

getArray-one : getarray(r, i, 1, st) = st[r – i '] ';

getArray-rec :
1 < j → getarray(r, i, j, st) = getarray(r, i, j - 1, st) + st[r – (i + j - 1)'];

getArray-all : getarray(r, st) = getarray(r, 0, st[r – _length].val, st);

getArrayv-zero : j ≤ 0 → getarrayv(r, i, j, st) = noval;

getArrayv-one : getarrayv(r, i, 1, st) = st[r – i '];

getArrayv-rec :
1 < j → getarrayv(r, i, j, st) = getarrayv(r, i, j - 1, st) ++ st[r – (i + j - 1)'];

getArrayv-all : getarrayv(r, st) = getarrayv(r, 0, st[r – _length].val, st);

end enrich

---

assoclist =
enrich assoclista **with**
functions

. [.]	:	assoclist × elem	→	data	**prio** 2;
. [.]	:	assoclist × data	→	elem	**prio** 2;
. [.]	:	assoclist × elem × data	→	assoclist	;

predicates

. ∈ .	:	elem × assoclist;
unique_al	:	assoclist;

axioms

assoc-yes : (a × d)' + ax[a] = d;

assoc-no : a ≠ b → (b × d)' + ax[a] = ax[a];

rassoc-yes : (a × d)' + ax[d] = a;

rassoc-no : d ≠ d_0 → (a × d_0)' + ax[d] = ax[d];

inassoc : a ∈ ax ↔ (∃ d. a × d ∈ ax);

put-empty : @[a, d] = (a × d)';

put-yes : (a × d_0)' + ax[a, d] = (a × d)' + ax;

put-no : a ≠ b → (b × d_0)' + ax[a, d] = (b × d_0)' + ax[a, d];

unique-e : unique_al(@);

unique-r : unique_al((a × d) + ax) ↔ ¬ a ∈ ax ∧ unique_al(ax);

end enrich

---

javavalues =
actualize list-perm **with** valuefuns **by morphism**
elem → javavalue; list → javavalues; @ → @; + → +; .first →.first; .rest →.rest; # → #; ' →'; + → +; + → +; + → +; ++ → ++; rmdup → rmdup; .last →.last; .butlast →.butlast; rev → rev; mklist → mklist; -$_l$ → -$_l$; -1$_l$ → -1$_l$; -1$_l$ → -1$_l$;] →]; pos → pos;] →]; sublist → sublist; firstn → firstn; restn → restn; lastn → lastn; frome → frome; ∪ → ∪; \ → \; filter → filter; #$_{oc}$ → #$_{oc}$; < → <; ∈ → ∈; dups → dups; disj → disj; ⊑ → ⊑; ⊒ → ⊒; ⊆ → ⊆; perm → perm; ⊆$_m$ → ⊆$_m$; a → val; a_0 → val_0; b → val_1; c → val_2; x → vals; x_0 → $vals_0$; y → $vals_1$; z → $vals_2$; y_0 → $vals_3$; z_0 → $vals_4$; x_1 → $vals_5$; y_1 → $vals_6$; z_1 → $vals_7$; x_2 → $vals_8$; y_2 → $vals_9$; z_2 → $vals_{10}$
end actualize

---

storefuns =
enrich javastore **with**
functions

. − .	:	store × reference	→	store	**prio** 9 **left**;
_out	:	store	→	javavalue	;

predicates

okrefs : store;
okval : javavalue × store;
normalmode : store;
exception : classname × store;
newref : reference × store;
classof : reference × classname × store;
typeof : reference × javatype × store;
. ∈ . : reference × store;
reftypep : javavalue × store;
initerror : classname × store;
initdone : classname × store;
initundone : classname × store;
init : store;
is_newref_list : references × store;
. $=_{mode}$. : store × store;
eqexmode : store × store × reference;
eqex : store × store × reference;
eqkeys : store × store;
sameobj : reference × store × store;

variables i_1, i_2: int;

axioms

eqkeys : eqkeys(st, st_0) ↔ (∀ rk. rk ∈ st ↔ rk ∈ st_0);

okrefs : okrefs(st) ↔ (∀ rk. rk ∈ st → okval(st[rk], st));

okval : okval(val, st) ↔ (∀ r. r ∈ val → r ∈ st);

list_ok : is_newref_list(refs, st) ↔ ¬ dups(refs) ∧ (∀ r. r ∈ refs → newref(r, st));

instore : r ∈ st ↔ (∃ sk. r – sk ∈ st);

deleteref :
¬ r ∈ st – r
∧ (∀ r_0, sk. r ≠ r_0
→ (r_0 – sk ∈ st ↔ r_0 – sk ∈ st – r) ∧ st[r_0 – sk] = st – r[r_0 – sk]);

out : _out(st) = flatten(st[_out]);

normalmode : normalmode(st) ↔ is_normal_mode(st[_mode]);

newref : newref(r, st) ↔ r ≠ jvmref ∧ (∀ rk. rk ∈ st → rk.ref ≠ r);

reftypep :
reftypep(val, st) ↔ is_referencevalue(val) ∧ (val.val = jvmref ∨ val.val ∈ st);

classOf :
classof(r, $class_1$, st)
↔ r ≠ jvmref ∧ r – _type ∈ st ∧ st[r – _type] = typeval(mkclasstype($class_1$));

typeOf : typeof(r, ty, st) ↔ r ≠ jvmref ∧ r – _type ∈ st ∧ st[r – _type] = typeval(ty);

initerror :
initerror($class_1$, st)
↔ jvmref – mkfs($class_1$, void_type, "initstate".field)' ∈ st
∧ st[jvmref – mkfs($class_1$, void_type, "initstate".field)'] = initval(error);

initundone :
initundone($class_1$, st)
↔ jvmref – mkfs($class_1$, void_type, "initstate".field)' ∈ st
∧ st[jvmref – mkfs($class_1$, void_type, "initstate".field)'] = initval(undone);

initdone : initdone($class_1$, st) ↔ ¬ initerror($class_1$, st) ∧ ¬ initundone($class_1$, st);

init : init(st) $\leftrightarrow$ ($\forall$ $class_1$. initdone($class_1$, st));

exception :
exception($class_1$, st) $\leftrightarrow$ is_throw_mode(st[_mode]) $\wedge$ st[_mode].type = mkclasstype($class_1$);

equalXmode :
st $=_{mode}$ st_0
$\leftrightarrow$ st = st_0
$\vee$ ($\exists$ r, $class_1$. st = st_0[r – _type, typeval(mkclasstype($class_1$))][_mode, throw(r, mkclasstype($class_1$))]
$\wedge$ newref(r, st_0));

sameobj :
sameobj(r, st, st_0)
$\leftrightarrow$ ($\forall$ sk. r – sk $\in$ st $\leftrightarrow$ r – sk $\in$ st_0)
$\wedge$ (r – _type $\in$ st $\rightarrow$ st[r – _type] = st_0[r – _type])
$\wedge$ (r – _length $\in$ st $\rightarrow$ st[r – _length] = st_0[r – _length]);

eqexmode : eqexmode(st, st_0, r) $\leftrightarrow$ st – r $=_{mode}$ st_0 – r;

eqex : eqex(st, st_0, r) $\leftrightarrow$ st – r = st_0 – r;

end enrich

---

javastore =
actualize ostore **with** valuefuns **by morphism**
elem $\rightarrow$ refkey; data $\rightarrow$ javavalue; ++ $\rightarrow$ ++; } $\rightarrow$}; – $\rightarrow$–; .min $\rightarrow$.min; .max $\rightarrow$.max; $\times$ $\rightarrow$ $\times$;] $\rightarrow$];] $\rightarrow$]; – $\rightarrow$–; .min $\rightarrow$.min; < $\rightarrow$ <; $\in$ $\rightarrow$ $\in$; < $\rightarrow$ <; < $\rightarrow$ <; $\in$ $\rightarrow$ $\in$; a $\rightarrow$ rk; a_0 $\rightarrow$ rk_0; b $\rightarrow$ rk_1; c $\rightarrow$ rk_2; d $\rightarrow$ val; d_0 $\rightarrow$ val_0; d_1 $\rightarrow$ val_1; d_2 $\rightarrow$ val_2
end actualize

---

valuefuns =
enrich javavalue **with**
functions

s$\rightarrow$r	:	string	$\rightarrow$	reference	;
flatten	:	javavalue	$\rightarrow$	javavalue	;
shiftright	:	javavalue	$\rightarrow$	javavalue	;
initial_value	:	javatype	$\rightarrow$	javavalue	;
get	:	nat $\times$ javavalue	$\rightarrow$	javavalue	;
getrange	:	nat $\times$ nat $\times$ javavalue	$\rightarrow$	javavalue	;

predicates

is_normal_mode	:	javavalue;
refp	:	javavalue;
. $\in$.	:	reference $\times$ javavalue;

axioms

string2ref : s$\rightarrow$r(str) = s$\rightarrow$r(str_0) $\leftrightarrow$ str = str_0;

init_val_bool : initial_value(boolean_type) = boolval(false);

init_val_int : initial_value(int_type) = intval(0);

init_val_short : initial_value(short_type) = shortval(0_s);

init_val_byte : initial_value(byte_type) = byteval(0_b);

init_val_class : initial_value(mkclasstype($class_1$)) = refval(jvmref);

init_val_array : initial_value(mkarraytype(ty)) = refval(jvmref);

refp : refp(val) $\leftrightarrow$ is_referencevalue(val) $\wedge$ val.val $\neq$ jvmref;

normal_mode : is_normal_mode(val) $\leftrightarrow$ val = noval;

base : $\neg$ is_valuelist(val) $\rightarrow$ flatten(val) = val;

flatten-shift : $\neg$ is_valuelist(val_0) $\rightarrow$ flatten(val ++ val_0) = flatten(val) ++ val_0;

flatten-rec : flatten(val ++ (val_0 ++ val_1)) = flatten(val ++ val_0 ++ val_1);

shiftright-shift :
$\neg$ is_valuelist(val) $\rightarrow$ shiftright(val ++ val_0) = shiftright(val) ++ val_0;

shiftright-rec : shiftright(val ++ val_0 ++ val_1) = shiftright(val ++ (val_0 ++ val_1));

refin-int : $\neg$ r $\in$ intval(i);

refin-byte : $\neg$ r $\in$ byteval(by);

refin-short : $\neg$ r $\in$ shortval(sho);

refin-ref : r $\in$ refval(r_0) $\leftrightarrow$ r = r_0;

refin-bool : $\neg$ r $\in$ boolval(boolvar);

refin-list : r $\in$ val ++ val_0 $\leftrightarrow$ r $\in$ val $\vee$ r $\in$ val_0;

refin-noval : $\neg$ r $\in$ noval;

refin-type : $\neg$ r $\in$ typeval(ty);

refin-initval : $\neg$ r $\in$ initval(istate);

refin-break : $\neg$ r $\in$ break(lbl);

refin-continue : $\neg$ r $\in$ continue(lbl);

refin-return : r $\in$ return(val, ty) $\leftrightarrow$ r $\in$ val;

refin-throw : r $\in$ throw(r_0, ty) $\leftrightarrow$ r = r_0;

getRange-base : getrange(m, m, val) = get(m, val);

getRange-rec : m < n $\rightarrow$ getrange(m, n, val) = getrange(m, n -1, val) ++ get(n, val);

get-zero : get(0, val) = shiftright(val).firstval;

get-rec : get(m +1, val) = get(m, shiftright(val).restval);

end enrich

---

javavalue =
data specification
using label, initstate, byte, storekeys, references
javavalue = intval (. .val : int ;) **with** is_integervalue
| byteval (. .val : byte ;) **with** is_bytevalue
| shortval (. .val : short ;) **with** is_shortvalue
| refval (. .val : reference ;) **with** is_referencevalue
| boolval (. .val : bool ;) **with** is_boolvalue
| stringval (. .val : string ;) **with** is_stringvalue
| . ++ . (. .firstval : javavalue ; . .restval : javavalue ;) **prio** 9 **left**
with is_valuelist
| noval **with** is_novalue
| typeval (. .type : javatype ;) **with** is_typevalue
| initval (. .istate : initstate ;) **with** is_initvalue
| break (. .label : label ;) **with** is_break_mode
| continue (. .label : label ;) **with** is_continue_mode

| return (. .val : javavalue ; . .type : javatype ;) **with** is_return_mode
| throw (. .ref : reference ; . .type : javatype ;) **with** is_throw_mode
;

variables val, val_0, val_1, val_2: javavalue;

end data specification

Generated axioms:

javavalue **freely generated by** noval, intval, byteval, shortval, refval, boolval, stringval, ++, typeval, initval, break, continue, return, throw;

disj : intval(i) $\neq$ byteval(by);

disj : intval(i) $\neq$ shortval(sho);

disj : intval(i) $\neq$ refval(r);

disj : intval(i) $\neq$ boolval(boolvar);

disj : intval(i) $\neq$ stringval(stringvar);

disj : intval(i) $\neq$ val ++ val_0;

disj : intval(i) $\neq$ noval;

disj : intval(i) $\neq$ typeval(ty);

disj : intval(i) $\neq$ initval(istate);

disj : intval(i) $\neq$ break(lbl);

disj : intval(i) $\neq$ continue(lbl);

disj : intval(i) $\neq$ return(val, ty);

disj : intval(i) $\neq$ throw(r, ty);

disj : byteval(by) $\neq$ shortval(sho);

disj : byteval(by) $\neq$ refval(r);

disj : byteval(by) $\neq$ boolval(boolvar);

disj : byteval(by) $\neq$ stringval(stringvar);

disj : byteval(by) $\neq$ val ++ val_0;

disj : byteval(by) $\neq$ noval;

disj : byteval(by) $\neq$ typeval(ty);

disj : byteval(by) $\neq$ initval(istate);

disj : byteval(by) $\neq$ break(lbl);

disj : byteval(by) $\neq$ continue(lbl);

disj : byteval(by) $\neq$ return(val, ty);

disj : byteval(by) $\neq$ throw(r, ty);

disj : shortval(sho) $\neq$ refval(r);

disj : shortval(sho) $\neq$ boolval(boolvar);

disj : shortval(sho) $\neq$ stringval(stringvar);

disj : shortval(sho) $\neq$ val ++ val_0;

disj : shortval(sho) $\neq$ noval;

disj : shortval(sho) $\neq$ typeval(ty);

disj : shortval(sho) $\neq$ initval(istate);

disj : shortval(sho) $\neq$ break(lbl);

disj : shortval(sho) $\neq$ continue(lbl);
disj : shortval(sho) $\neq$ return(val, ty);
disj : shortval(sho) $\neq$ throw(r, ty);
disj : refval(r) $\neq$ boolval(boolvar);
disj : refval(r) $\neq$ stringval(stringvar);
disj : refval(r) $\neq$ val ++ val_0;
disj : refval(r) $\neq$ noval;
disj : refval(r) $\neq$ typeval(ty);
disj : refval(r) $\neq$ initval(istate);
disj : refval(r) $\neq$ break(lbl);
disj : refval(r) $\neq$ continue(lbl);
disj : refval(r) $\neq$ return(val, ty);
disj : refval(r) $\neq$ throw(r_0, ty);
disj : boolval(boolvar) $\neq$ stringval(stringvar);
disj : boolval(boolvar) $\neq$ val ++ val_0;
disj : boolval(boolvar) $\neq$ noval;
disj : boolval(boolvar) $\neq$ typeval(ty);
disj : boolval(boolvar) $\neq$ initval(istate);
disj : boolval(boolvar) $\neq$ break(lbl);
disj : boolval(boolvar) $\neq$ continue(lbl);
disj : boolval(boolvar) $\neq$ return(val, ty);
disj : boolval(boolvar) $\neq$ throw(r, ty);
disj : stringval(stringvar) $\neq$ val ++ val_0;
disj : stringval(stringvar) $\neq$ noval;
disj : stringval(stringvar) $\neq$ typeval(ty);
disj : stringval(stringvar) $\neq$ initval(istate);
disj : stringval(stringvar) $\neq$ break(lbl);
disj : stringval(stringvar) $\neq$ continue(lbl);
disj : stringval(stringvar) $\neq$ return(val, ty);
disj : stringval(stringvar) $\neq$ throw(r, ty);
disj : val ++ val_0 $\neq$ noval;
disj : val ++ val_0 $\neq$ typeval(ty);
disj : val ++ val_0 $\neq$ initval(istate);
disj : val ++ val_0 $\neq$ break(lbl);
disj : val ++ val_0 $\neq$ continue(lbl);
disj : val ++ val_0 $\neq$ return(val_1, ty);
disj : val ++ val_0 $\neq$ throw(r, ty);
disj : noval $\neq$ typeval(ty);
disj : noval $\neq$ initval(istate);
disj : noval $\neq$ break(lbl);
disj : noval $\neq$ continue(lbl);

disj : noval $\neq$ return(val, ty);
disj : noval $\neq$ throw(r, ty);
disj : typeval(ty) $\neq$ initval(istate);
disj : typeval(ty) $\neq$ break(lbl);
disj : typeval(ty) $\neq$ continue(lbl);
disj : typeval(ty) $\neq$ return(val, ty_0);
disj : typeval(ty) $\neq$ throw(r, ty_0);
disj : initval(istate) $\neq$ break(lbl);
disj : initval(istate) $\neq$ continue(lbl);
disj : initval(istate) $\neq$ return(val, ty);
disj : initval(istate) $\neq$ throw(r, ty);
disj : break(lbl) $\neq$ continue(lbl_0);
disj : break(lbl) $\neq$ return(val, ty);
disj : break(lbl) $\neq$ throw(r, ty);
disj : continue(lbl) $\neq$ return(val, ty);
disj : continue(lbl) $\neq$ throw(r, ty);
disj : return(val, ty) $\neq$ throw(r, ty_0);
sel : throw(r, ty).type = ty;
sel : throw(r, ty).ref = r;
sel : return(val, ty).type = ty;
sel : return(val, ty).val = val;
sel : continue(lbl).label = lbl;
sel : break(lbl).label = lbl;
sel : initval(istate).istate = istate;
sel : typeval(ty).type = ty;
sel : (val ++ val_0).restval = val_0;
sel : (val ++ val_0).firstval = val;
sel : stringval(stringvar).val = stringvar;
sel : boolval(boolvar).val $\leftrightarrow$ boolvar = true;
sel : refval(r).val = r;
sel : shortval(sho).val = sho;
sel : byteval(by).val = by;
sel : intval(i).val = i;
test : is_throw_mode(throw(r, ty));
test : $\neg$ is_throw_mode(return(val, ty));
test : $\neg$ is_throw_mode(continue(lbl));
test : $\neg$ is_throw_mode(break(lbl));
test : $\neg$ is_throw_mode(initval(istate));
test : $\neg$ is_throw_mode(typeval(ty));
test : $\neg$ is_throw_mode(noval);
test : $\neg$ is_throw_mode(val ++ val_0);

test : ¬ is_throw_mode(stringval(stringvar));
test : ¬ is_throw_mode(boolval(boolvar));
test : ¬ is_throw_mode(refval(r));
test : ¬ is_throw_mode(shortval(sho));
test : ¬ is_throw_mode(byteval(by));
test : ¬ is_throw_mode(intval(i));
test : ¬ is_return_mode(throw(r, ty));
test : is_return_mode(return(val, ty));
test : ¬ is_return_mode(continue(lbl));
test : ¬ is_return_mode(break(lbl));
test : ¬ is_return_mode(initval(istate));
test : ¬ is_return_mode(typeval(ty));
test : ¬ is_return_mode(noval);
test : ¬ is_return_mode(val ++ val_0);
test : ¬ is_return_mode(stringval(stringvar));
test : ¬ is_return_mode(boolval(boolvar));
test : ¬ is_return_mode(refval(r));
test : ¬ is_return_mode(shortval(sho));
test : ¬ is_return_mode(byteval(by));
test : ¬ is_return_mode(intval(i));
test : ¬ is_continue_mode(throw(r, ty));
test : ¬ is_continue_mode(return(val, ty));
test : is_continue_mode(continue(lbl));
test : ¬ is_continue_mode(break(lbl));
test : ¬ is_continue_mode(initval(istate));
test : ¬ is_continue_mode(typeval(ty));
test : ¬ is_continue_mode(noval);
test : ¬ is_continue_mode(val ++ val_0);
test : ¬ is_continue_mode(stringval(stringvar));
test : ¬ is_continue_mode(boolval(boolvar));
test : ¬ is_continue_mode(refval(r));
test : ¬ is_continue_mode(shortval(sho));
test : ¬ is_continue_mode(byteval(by));
test : ¬ is_continue_mode(intval(i));
test : ¬ is_break_mode(throw(r, ty));
test : ¬ is_break_mode(return(val, ty));
test : ¬ is_break_mode(continue(lbl));
test : is_break_mode(break(lbl));
test : ¬ is_break_mode(initval(istate));
test : ¬ is_break_mode(typeval(ty));
test : ¬ is_break_mode(noval);

test : ¬ is_break_mode(val ++ val_0);
test : ¬ is_break_mode(stringval(stringvar));
test : ¬ is_break_mode(boolval(boolvar));
test : ¬ is_break_mode(refval(r));
test : ¬ is_break_mode(shortval(sho));
test : ¬ is_break_mode(byteval(by));
test : ¬ is_break_mode(intval(i));
test : ¬ is_initvalue(throw(r, ty));
test : ¬ is_initvalue(return(val, ty));
test : ¬ is_initvalue(continue(lbl));
test : ¬ is_initvalue(break(lbl));
test : is_initvalue(initval(istate));
test : ¬ is_initvalue(typeval(ty));
test : ¬ is_initvalue(noval);
test : ¬ is_initvalue(val ++ val_0);
test : ¬ is_initvalue(stringval(stringvar));
test : ¬ is_initvalue(boolval(boolvar));
test : ¬ is_initvalue(refval(r));
test : ¬ is_initvalue(shortval(sho));
test : ¬ is_initvalue(byteval(by));
test : ¬ is_initvalue(intval(i));
test : ¬ is_typevalue(throw(r, ty));
test : ¬ is_typevalue(return(val, ty));
test : ¬ is_typevalue(continue(lbl));
test : ¬ is_typevalue(break(lbl));
test : ¬ is_typevalue(initval(istate));
test : is_typevalue(typeval(ty));
test : ¬ is_typevalue(noval);
test : ¬ is_typevalue(val ++ val_0);
test : ¬ is_typevalue(stringval(stringvar));
test : ¬ is_typevalue(boolval(boolvar));
test : ¬ is_typevalue(refval(r));
test : ¬ is_typevalue(shortval(sho));
test : ¬ is_typevalue(byteval(by));
test : ¬ is_typevalue(intval(i));
test : ¬ is_novalue(throw(r, ty));
test : ¬ is_novalue(return(val, ty));
test : ¬ is_novalue(continue(lbl));
test : ¬ is_novalue(break(lbl));
test : ¬ is_novalue(initval(istate));
test : ¬ is_novalue(typeval(ty));

test : is_novalue(noval);
test : ¬ is_novalue(val ++ val_0);
test : ¬ is_novalue(stringval(stringvar));
test : ¬ is_novalue(boolval(boolvar));
test : ¬ is_novalue(refval(r));
test : ¬ is_novalue(shortval(sho));
test : ¬ is_novalue(byteval(by));
test : ¬ is_novalue(intval(i));
test : ¬ is_valuelist(throw(r, ty));
test : ¬ is_valuelist(return(val, ty));
test : ¬ is_valuelist(continue(lbl));
test : ¬ is_valuelist(break(lbl));
test : ¬ is_valuelist(initval(istate));
test : ¬ is_valuelist(typeval(ty));
test : ¬ is_valuelist(noval);
test : is_valuelist(val ++ val_0);
test : ¬ is_valuelist(stringval(stringvar));
test : ¬ is_valuelist(boolval(boolvar));
test : ¬ is_valuelist(refval(r));
test : ¬ is_valuelist(shortval(sho));
test : ¬ is_valuelist(byteval(by));
test : ¬ is_valuelist(intval(i));
test : ¬ is_stringvalue(throw(r, ty));
test : ¬ is_stringvalue(return(val, ty));
test : ¬ is_stringvalue(continue(lbl));
test : ¬ is_stringvalue(break(lbl));
test : ¬ is_stringvalue(initval(istate));
test : ¬ is_stringvalue(typeval(ty));
test : ¬ is_stringvalue(noval);
test : ¬ is_stringvalue(val ++ val_0);
test : is_stringvalue(stringval(stringvar));
test : ¬ is_stringvalue(boolval(boolvar));
test : ¬ is_stringvalue(refval(r));
test : ¬ is_stringvalue(shortval(sho));
test : ¬ is_stringvalue(byteval(by));
test : ¬ is_stringvalue(intval(i));
test : ¬ is_boolvalue(throw(r, ty));
test : ¬ is_boolvalue(return(val, ty));
test : ¬ is_boolvalue(continue(lbl));
test : ¬ is_boolvalue(break(lbl));
test : ¬ is_boolvalue(initval(istate));

test : ¬ is_boolvalue(typeval(ty));
test : ¬ is_boolvalue(noval);
test : ¬ is_boolvalue(val ++ val_0);
test : ¬ is_boolvalue(stringval(stringvar));
test : is_boolvalue(boolval(boolvar));
test : ¬ is_boolvalue(refval(r));
test : ¬ is_boolvalue(shortval(sho));
test : ¬ is_boolvalue(byteval(by));
test : ¬ is_boolvalue(intval(i));
test : ¬ is_referencevalue(throw(r, ty));
test : ¬ is_referencevalue(return(val, ty));
test : ¬ is_referencevalue(continue(lbl));
test : ¬ is_referencevalue(break(lbl));
test : ¬ is_referencevalue(initval(istate));
test : ¬ is_referencevalue(typeval(ty));
test : ¬ is_referencevalue(noval);
test : ¬ is_referencevalue(val ++ val_0);
test : ¬ is_referencevalue(stringval(stringvar));
test : ¬ is_referencevalue(boolval(boolvar));
test : is_referencevalue(refval(r));
test : ¬ is_referencevalue(shortval(sho));
test : ¬ is_referencevalue(byteval(by));
test : ¬ is_referencevalue(intval(i));
test : ¬ is_shortvalue(throw(r, ty));
test : ¬ is_shortvalue(return(val, ty));
test : ¬ is_shortvalue(continue(lbl));
test : ¬ is_shortvalue(break(lbl));
test : ¬ is_shortvalue(initval(istate));
test : ¬ is_shortvalue(typeval(ty));
test : ¬ is_shortvalue(noval);
test : ¬ is_shortvalue(val ++ val_0);
test : ¬ is_shortvalue(stringval(stringvar));
test : ¬ is_shortvalue(boolval(boolvar));
test : ¬ is_shortvalue(refval(r));
test : is_shortvalue(shortval(sho));
test : ¬ is_shortvalue(byteval(by));
test : ¬ is_shortvalue(intval(i));
test : ¬ is_bytevalue(throw(r, ty));
test : ¬ is_bytevalue(return(val, ty));
test : ¬ is_bytevalue(continue(lbl));
test : ¬ is_bytevalue(break(lbl));

test : $\neg$ is_bytevalue(initval(istate));

test : $\neg$ is_bytevalue(typeval(ty));

test : $\neg$ is_bytevalue(noval);

test : $\neg$ is_bytevalue(val ++ val_0);

test : $\neg$ is_bytevalue(stringval(stringvar));

test : $\neg$ is_bytevalue(boolval(boolvar));

test : $\neg$ is_bytevalue(refval(r));

test : $\neg$ is_bytevalue(shortval(sho));

test : is_bytevalue(byteval(by));

test : $\neg$ is_bytevalue(intval(i));

test : $\neg$ is_integervalue(throw(r, ty));

test : $\neg$ is_integervalue(return(val, ty));

test : $\neg$ is_integervalue(continue(lbl));

test : $\neg$ is_integervalue(break(lbl));

test : $\neg$ is_integervalue(initval(istate));

test : $\neg$ is_integervalue(typeval(ty));

test : $\neg$ is_integervalue(noval);

test : $\neg$ is_integervalue(val ++ val_0);

test : $\neg$ is_integervalue(stringval(stringvar));

test : $\neg$ is_integervalue(boolval(boolvar));

test : $\neg$ is_integervalue(refval(r));

test : $\neg$ is_integervalue(shortval(sho));

test : $\neg$ is_integervalue(byteval(by));

test : is_integervalue(intval(i));

inj : throw(r, ty) = throw(r_0, ty_0) $\leftrightarrow$ r = r_0 $\wedge$ ty = ty_0;

inj : return(val, ty) = return(val_0, ty_0) $\leftrightarrow$ val = val_0 $\wedge$ ty = ty_0;

inj : continue(lbl) = continue(lbl_0) $\leftrightarrow$ lbl = lbl_0;

inj : break(lbl) = break(lbl_0) $\leftrightarrow$ lbl = lbl_0;

inj : initval(istate) = initval(istate_0) $\leftrightarrow$ istate = istate_0;

inj : typeval(ty) = typeval(ty_0) $\leftrightarrow$ ty = ty_0;

inj : val ++ val_0 = val_1 ++ val_2 $\leftrightarrow$ val = val_1 $\wedge$ val_0 = val_2;

inj : stringval(stringvar) = stringval(stringvar_0) $\leftrightarrow$ stringvar = stringvar_0;

inj : boolval(boolvar) = boolval(boolvar_0) $\leftrightarrow$ boolvar = true $\leftrightarrow$ boolvar_0 = true;

inj : refval(r) = refval(r_0) $\leftrightarrow$ r = r_0;

inj : shortval(sho) = shortval(sho_0) $\leftrightarrow$ sho = sho_0;

inj : byteval(by) = byteval(by_0) $\leftrightarrow$ by = by_0;

inj : intval(i) = intval(i_0) $\leftrightarrow$ i = i_0;

case :
  val = intval(val.val)
$\vee$ val = byteval(val.val)
$\vee$ val = shortval(val.val)
$\vee$ val = refval(val.val)

$\vee$ val = boolval(val.val)
$\vee$ val = stringval(val.val)
$\vee$ val = val.firstval ++ val.restval
$\vee$ val = noval
$\vee$ val = typeval(val.type)
$\vee$ val = initval(val.istate)
$\vee$ val = break(val.label)
$\vee$ val = continue(val.label)
$\vee$ val = return(val.val, val.type)
$\vee$ val = throw(val.ref, val.type);

ex : is_throw_mode(val) $\leftrightarrow$ ($\exists$ r, ty. val = throw(r, ty));

ex : is_return_mode(val) $\leftrightarrow$ ($\exists$ val_0, ty. val = return(val_0, ty));

ex : is_continue_mode(val) $\leftrightarrow$ ($\exists$ lbl. val = continue(lbl));

ex : is_break_mode(val) $\leftrightarrow$ ($\exists$ lbl. val = break(lbl));

ex : is_initvalue(val) $\leftrightarrow$ ($\exists$ istate. val = initval(istate));

ex : is_typevalue(val) $\leftrightarrow$ ($\exists$ ty. val = typeval(ty));

ex : is_novalue(val) $\leftrightarrow$ val = noval;

ex : is_valuelist(val) $\leftrightarrow$ ($\exists$ val_0, val_1. val = val_0 ++ val_1);

ex : is_stringvalue(val) $\leftrightarrow$ ($\exists$ stringvar. val = stringval(stringvar));

ex : is_boolvalue(val) $\leftrightarrow$ ($\exists$ boolvar. val = boolval(boolvar));

ex : is_referencevalue(val) $\leftrightarrow$ ($\exists$ r. val = refval(r));

ex : is_shortvalue(val) $\leftrightarrow$ ($\exists$ sho. val = shortval(sho));

ex : is_bytevalue(val) $\leftrightarrow$ ($\exists$ by. val = byteval(by));

ex : is_integervalue(val) $\leftrightarrow$ ($\exists$ i. val = intval(i));

elim : is_throw_mode(val) $\rightarrow$ (r = val.ref $\wedge$ ty = val.type $\leftrightarrow$ val = throw(r, ty));

elim : is_return_mode(val) $\rightarrow$ (val_0 = val.val $\wedge$ ty = val.type $\leftrightarrow$ val = return(val_0, ty));

elim : is_continue_mode(val) $\rightarrow$ (lbl = val.label $\leftrightarrow$ val = continue(lbl));

elim : is_break_mode(val) $\rightarrow$ (lbl = val.label $\leftrightarrow$ val = break(lbl));

elim : is_initvalue(val) $\rightarrow$ (istate = val.istate $\leftrightarrow$ val = initval(istate));

elim : is_typevalue(val) $\rightarrow$ (ty = val.type $\leftrightarrow$ val = typeval(ty));

elim :
is_valuelist(val) $\rightarrow$ (val_0 = val.firstval $\wedge$ val_1 = val.restval $\leftrightarrow$ val = val_0 ++ val_1);

elim : is_stringvalue(val) $\rightarrow$ (stringvar = val.val $\leftrightarrow$ val = stringval(stringvar));

elim : is_boolvalue(val) $\rightarrow$ (boolvar = true $\leftrightarrow$ val.val $\leftrightarrow$ val = boolval(boolvar));

elim : is_referencevalue(val) $\rightarrow$ (r = val.val $\leftrightarrow$ val = refval(r));

elim : is_shortvalue(val) $\rightarrow$ (sho = val.val $\leftrightarrow$ val = shortval(sho));

elim : is_bytevalue(val) $\rightarrow$ (by = val.val $\leftrightarrow$ val = byteval(by));

elim : is_integervalue(val) $\rightarrow$ (i = val.val $\leftrightarrow$ val = intval(i));

---

byte =
enrich bitops **with**
sorts byte, short, bint;

constants

0_b : byte;
0_s : short;
0_{bi} : bint;

functions

b→i	:	byte	→	int	;
b→s	:	byte	→	short	;
b→bi	:	byte	→	bint	;
s→i	:	short	→	int	;
s→b	:	short	→	byte	;
s→bi	:	short	→	bint	;
bi→i	:	bint	→	int	;
bi→b	:	bint	→	byte	;
bi→s	:	bint	→	short	;
i→b	:	int	→	byte	;
i→s	:	int	→	short	;
i→bi	:	int	→	bint	;

predicates

. $\in_{byte}$	:	int;
. $\in_{short}$	:	int;
. $\in_{bint}$	:	int;

variables

by, byi, byj: byte;
sho, shoi, shoj: short;
bi, bj: bint;

axioms

zero-byte : 0_b = i→b(0);
zero-short : 0_s = i→s(0);
zero-bint : 0_{bi} = i→bi(0);
inbyte : i $\in_{byte}$ ↔-128 ≤ i ∧ i ≤ 127;
inshort : i $\in_{short}$ ↔-32768 ≤ i ∧ i ≤ 32767;
inbint : i $\in_{bint}$ ↔-2147483648 ≤ i ∧ i ≤ 2147483647;
ibi : i→b(b→i(by)) = by;
isi : i→s(s→i(sho)) = sho;
ibii : i→bi(bi→i(bi)) = bi;
byte2int-in : b→i(by)$\in_{byte}$;
short2int-in : s→i(sho)$\in_{short}$;
bint2int-in : bi→i(bi)$\in_{bint}$;
bib-in : i $\in_{byte}$ → b→i(i→b(i)) = i;
sis-in : i $\in_{short}$ → s→i(i→s(i)) = i;
biibi-in : i $\in_{bint}$ → bi→i(i→bi(i)) = i;
int2byte-out : ¬ i $\in_{byte}$ → i→b(i) = i→b(bits2int(lastn(8, int2bits(i))));
int2short-out : ¬ i $\in_{short}$ → i→s(i) = i→s(bits2int(lastn(16, int2bits(i))));
int2bint-out : ¬ i $\in_{bint}$ → i→bi(i) = i→bi(bits2int(lastn(32, int2bits(i))));
byte2short : b→s(by) = i→s(b→i(by));

byte2bint : b→bi(by) = i→bi(b→i(by));

short2byte : s→b(sho) = i→b(s→i(sho));

short2bint : s→bi(sho) = i→bi(s→i(sho));

end enrich

---

initstate =
data specification
initstate = done
| error
| undone
;
variables istate: initstate;
end data specification

Generated axioms:

initstate **freely generated by** done, error, undone;
disj : done ≠ error;

disj : done ≠ undone;

disj : error ≠ undone;

case : istate = done ∨ istate = error ∨ istate = undone;

---

label =
data specification
using string-append
label = . .label (. .label : string ;);
variables lbl, lbl_0, lbl_1, lbl_2: label;
end data specification

Generated axioms:

label **freely generated by** .label;
sel : stringvar.label.label = stringvar;

inj : stringvar.label = $stringvar_0$.label ↔ stringvar = $stringvar_0$;

case : lbl.label.label = lbl;

elim : stringvar = lbl.label ↔ lbl = stringvar.label;

---

references =
actualize list-perm **with** reference **by morphism**
elem → reference; list → references; @ → @; + → +; .first →.first; .rest →.rest; # → #; ' →'; + → +; + → +; + → +; ++ → ++; rmdup → rmdup; .last →.last; .butlast →.butlast; rev → rev; mklist → mklist; -$_l$ → -$_l$; -1$_l$ → -1$_l$; -1$_l$ → -1$_l$;] →]; pos → pos;] →]; sublist → sublist; firstn → firstn; restn → restn; lastn → lastn; frome → frome; ∪ → ∪; \ → \; filter → filter; #$_{oc}$ → #$_{oc}$; < → <; ∈ → ∈; dups → dups; disj → disj; ⊑ → ⊑; ⊒ → ⊒; ⊆ → ⊆; perm → perm; ⊆$_m$ → ⊆$_m$; a → r; a_0 → r_0; b → r_1; c → r_2; x → refs; x_0 → $refs_0$; y → $refs_1$; z → $refs_2$; y_0 → $refs_3$; z_0 → $refs_4$; x_1 → $refs_5$; y_1 → $refs_6$; z_1 → $refs_7$; x_2 → $refs_8$; y_2 → $refs_9$; z_2 → $refs_{10}$

end actualize

---

storekeys =
actualize list-perm **with** specialkeys **by morphism**
elem → storekey; list → storekeys; @ → @; + → +; .first →.first; .rest →.rest; # → #; ' →'; + → +; + → +; + → +; ++ → ++; rmdup → rmdup; .last →.last; .butlast →.butlast; rev → rev; mklist → mklist; -$_l$ → -$_l$; -1$_l$ → -1$_l$; -1$_l$ → -1$_l$;] →]; pos → pos;] →]; sublist → sublist; firstn → firstn; restn → restn; lastn → lastn; frome → frome; ∪ → ∪; \ → \; filter → filter; #$_{oc}$ → #$_{oc}$; < → <; ∈ → ∈; dups → dups; disj → disj; ⊑ → ⊑; ⊒ → ⊒; ⊆ → ⊆; perm → perm; ⊆$_m$ → ⊆$_m$; a → sk; a$_0$ → sk$_0$; b → sk$_1$; c → sk$_2$; x → sks; x$_0$ → sks$_0$; y → sks$_1$; z → sks$_2$; y$_0$ → sks$_3$; z$_0$ → sks$_4$; x$_1$ → sks$_5$; y$_1$ → sks$_6$; z$_1$ → sks$_7$; x$_2$ → sks$_8$; y$_2$ → sks$_9$; z$_2$ → sks$_{10}$
end actualize

---

bitops =
enrich int-pot, bitlista **with**
functions

int2bits	:	int	→	bitlist	;
uint2bits	:	int	→	bitlist	;
bitsinvert	:	bitlist	→	bitlist	;
bits2int	:	bitlist	→	int	;
bits2uint	:	bitlist	→	int	;
. +1	:	bitlist	→	bitlist	;
bitsand	:	bitlist × bitlist	→	bitlist	;
bitsxor	:	bitlist × bitlist	→	bitlist	;
bitsor	:	bitlist × bitlist	→	bitlist	;
band	:	int × int	→	int	;
bxor	:	int × int	→	int	;
bor	:	int × int	→	int	;
bcompl	:	int	→	int	;
. <<$_i$.	:	int × int	→	int	**prio** 9;
. >>$_i$.	:	int × int	→	int	**prio** 9;
. >>>$_i$.	:	int × int	→	int	**prio** 9;

predicates intrep : bitlist;

axioms

intrep : intrep(bits) ↔ (∃ i. int2bits(i) = bits);

int2bits-pos : 0 ≤ i → int2bits(i) = uint2bits(i);

int2bits-neg : i < 0 → int2bits(i) = bitsinvert(uint2bits(abs(i)))+1;

uint2bits-zero : uint2bits(0) = false ';

uint2bits-one : uint2bits(1) = false + true;

uint2bits-rec : 1 < i → uint2bits(i) = uint2bits(i / 2) + (i % 2 = 1);

bitsinvert-base : bitsinvert(@) = @;

bitsinvert-zero : bitsinvert(false ' + bits) = true + bitsinvert(bits);

bitsinvert-one : bitsinvert(true ' + bits) = false + bitsinvert(bits);

succ-zero : (bits + false ')+1 = bits + true;

succ-one : true ' +1 = true + false;

succ-rec : (bits + boolvar ' + true ')+1 = (bits + boolvar)+1 + false;

bits2int-pos : bits2int(false ' + bits) = bits2uint(bits);

bits2int-neg : bits2int(true ' + bits) = ~ bits2uint(bitsinvert(true + bits)+1);

bits2uint-base : bits2uint(@) = 0;

bits2uint-zero : bits2uint(bits + false ') = bits2uint(bits) * 2;

bits2uint-one : bits2uint(bits + true ') = bits2uint(bits) * 2 + 1;

bitsand-onel : bitsand(true ', bits + boolvar ') = bits + boolvar;

bitsand-oner : bitsand(bits + boolvar ', true ') = bits + boolvar;

bitsand-zerol : bitsand(false ', bits + boolvar ') = false ';

bitsand-zeror : bitsand(bits + boolvar ', false ') = false ';

bitsand-rec :
  bitsand(bits + boolvar ' + $boolvar_0$ ', $bits_0$ + $boolvar_1$ ' + $boolvar_2$ ')
= bitsand(bits + boolvar, $bits_0$ + $boolvar_1$) + ($boolvar_0$ and $boolvar_2$);

bitsxor-onel : bitsxor(true ', bits + boolvar ') = bitsinvert(bits + boolvar);

bitsxor-oner : bitsxor(bits + boolvar ', true ') = bitsinvert(bits + boolvar);

bitsxor-zerol : bitsxor(false ', bits + boolvar ') = bits + boolvar;

bitsxor-zeror : bitsxor(bits + boolvar ', false ') = bits + boolvar;

bitsxor-rec :
  bitsxor(bits + boolvar ' + $boolvar_0$ ', $bits_0$ + $boolvar_1$ ' + $boolvar_2$ ')
= bitsxor(bits + boolvar, $bits_0$ + $boolvar_1$) + not($boolvar_0$ <-> $boolvar_2$);

bitsor-onel : bitsor(true ', bits + boolvar ') = true ';

bitsor-oner : bitsor(bits + boolvar ', true ') = true ';

bitsor-zerol : bitsor(false ', bits + boolvar ') = bits + boolvar;

bitsor-zeror : bitsor(bits + boolvar ', false ') = bits + boolvar;

bitsor-rec :
  bitsor(bits + boolvar ' + $boolvar_0$ ', $bits_0$ + $boolvar_1$ ' + $boolvar_2$ ')
= bitsor(bits + boolvar, $bits_0$ + $boolvar_1$) + ($boolvar_0$ or $boolvar_2$);

band-def : band(i, j) = bits2int(bitsand(int2bits(i), int2bits(j)));

bxor-def : bxor(i, j) = bits2int(bitsxor(int2bits(i), int2bits(j)));

bor-def : bor(i, j) = bits2int(bitsor(int2bits(i), int2bits(j)));

bcompl-def : bcompl(i) = bits2int(bitsinvert(int2bits(i)));

shiftleft :
  i $<<_i$ j
= bits2int(int2bits(i) + mklist(false, i→n(bits2int(bitsand(int2bits(j), false + true + true + true + true + true)))));

shiftrsign :
    n = i→n(bits2int(bitsand(int2bits(j), false + true + true + true + true + true)))
  ∧ bits = int2bits(i)
→ i $>>_i$ j = (# bits ≤ n ⊃ (i < 0 ⊃ -1; 0); bits2int(sublist(0, # bits - n, bits)));

shiftrzero : i $>>>_i$ j = (i < 0 ⊃ (i $>>_i$ j) + (2 $<<_i$ ~ j); (i $>>_i$ j));

end enrich

specialkeys =
enrich refkey **with**
constants
_out : refkey;
_mode : refkey;
_type : storekey;
_length : storekey;

predicates . < . : refkey × refkey;

axioms

out : _out = jvmref – mkfs("Object".class, void_type, "out".field)';
mode : _mode = jvmref – mkfs("Object".class, void_type, "mode".field)';
type : _type = mkfs("Object".class, void_type, "type".field)';
length : _length = mkfs("Array".class, int_type, "length".field)';
irref : ¬ rk < rk;
trans : rk < rk_0 ∧ rk_0 < rk_1 → rk < rk_1;
total : rk < rk_0 ∨ rk = rk_0 ∨ rk_0 < rk;

end enrich

---

bitlista =
actualize list-perm **with** bool **by morphism**
elem → bool; list → bitlist; @ → @; + → +; .first →.first; .rest →.rest; # → #; ' →'; + → +; + → +; + → +; ++ → ++; rmdup → rmdup; .last →.last; .butlast →.butlast; rev → rev; mklist → mklist; -$_l$ → -$_l$; -1_l → -1_l; -1_l → -1_l;] →]; pos → pos;] →]; sublist → sublist; firstn → firstn; restn → restn; lastn → lastn; frome → frome; ∪ → ∪; \ → \; filter → filter; $\#_{oc}$ → $\#_{oc}$; < → <; ∈ → ∈; dups → dups; disj → disj; ⊑ → ⊑; ⊒ → ⊒; ⊆ → ⊆; perm → perm; $\subseteq_m$ → $\subseteq_m$; a → boolvar; a_0 → $boolvar_0$; b → $boolvar_1$; c → $boolvar_2$; x → bits; x_0 → $bits_0$; y → $bits_1$; z → $bits_2$; y_0 → $bits_3$; z_0 → $bits_4$; x_1 → $bits_5$; y_1 → $bits_6$; z_1 → $bits_7$; x_2 → $bits_8$; y_2 → $bits_9$; z_2 → $bits_{10}$
end actualize

---

refkey =
data specification
using storekey
refkey = . – . **prio** 9 **left** (. .ref : reference ; . .key : storekey ;) **prio** 9 **left**;
variables rk, rk_0, rk_1, rk_2: refkey;
end data specification

Generated axioms:

refkey **freely generated by** –;
sel : (r – sk).key = sk;
sel : (r – sk).ref = r;
inj : r – sk = r_0 – sk_0 ↔ r = r_0 ∧ sk = sk_0;
case : rk.ref – rk.key = rk;
elim : r = rk.ref ∧ sk = rk.key ↔ rk = r – sk;

---

storekey =
data specification
using fieldspec, reference
storekey = . ' (. .fs : fieldspec ;) **with** is_fskey
| . ' (. .index : int ;) **with** is_indexkey
;
variables sk, sk_0, sk_1: storekey;
end data specification

Generated axioms:

storekey **freely generated by** ', ';
disj : fs ' ≠ i ';
sel : i '.index = i;
sel : fs '.fs = fs;
test : is_indexkey(i ');
test : ¬ is_indexkey(fs ');
test : ¬ is_fskey(i ');
test : is_fskey(fs ');
inj : i ' = i_0 ' ↔ i = i_0;
inj : fs ' = fs_0 ' ↔ fs = fs_0;
case : sk = sk.fs ' ∨ sk = sk.index ';
ex : is_indexkey(sk) ↔ (∃ i. sk = i ');
ex : is_fskey(sk) ↔ (∃ fs. sk = fs ');
elim : is_indexkey(sk) → (i = sk.index ↔ sk = i ');
elim : is_fskey(sk) → (fs = sk.fs ↔ sk = fs ');

---

fieldspec =
data specification
using fieldname, typefuns
fieldspec = mkfs (. .class : classname ; . .type : javatype ; . .field : fieldname ;);
variables fs, fs_0, fs_1: fieldspec;
end data specification

Generated axioms:

fieldspec **freely generated by** mkfs;
sel : mkfs(class, ty, fieldvar).field = fieldvar;
sel : mkfs(class, ty, fieldvar).type = ty;
sel : mkfs(class, ty, fieldvar).class = class;
inj :
mkfs(class, ty, fieldvar) = mkfs($class_0$, ty_0, $fieldvar_0$)
↔ class = $class_0$ ∧ ty = ty_0 ∧ fieldvar = $fieldvar_0$;
case : mkfs(fs.class, fs.type, fs.field) = fs;
elim :
class = fs.class ∧ ty = fs.type ∧ fieldvar = fs.field ↔ fs = mkfs(class, ty, fieldvar);

---

reference =
specification
sorts reference;
constants jvmref : reference;
predicates . < . : reference × reference;
variables r, r_0, r_1, r_2, this: reference;

axioms

irreflexivity : ¬ r < r;

transitivity : r < r_0 ∧ r_0 < r_1 → r < r_1;

totality : r < r_0 ∨ r = r_0 ∨ r_0 < r;

no-maximum : ∀ r. ∃ r_0. r < r_0;

end specification

---

typefuns =
enrich javatype, intnat **with**
functions

. .jtclass	:	javatype	→	classname ;
mktype_from_dims	:	javatype × int	→	javatype ;

predicates

. ≤ . : javatype × javatype;
. ≤ . : classname × classname;

axioms

mktype_base : mktype_from_dims(ty, 0) = ty;

mktype_rec : 0 < i → mktype_from_dims(ty, i) = mkarraytype(mktype_from_dims(ty, i - 1));

reflexivity : $class_1$ ≤ $class_1$;

transitivity : $class_1$ ≤ $class_2$ ∧ $class_2$ ≤ $class_3$ → $class_1$ ≤ $class_3$;

javatype-class : mkclasstype($class_1$).jtclass = $class_1$;

javatype-array : mkarraytype(ty).jtclass = ty.jtclass;

subclass : mkclasstype($class_1$) ≤ mkclasstype($class_2$) ↔ $class_1$ ≤ $class_2$;

subarray1 : mkarraytype(ty) ≤ mkarraytype(ty_0) ↔ ty ≤ ty_0;

subarray2 : mkarraytype(ty) ≤ mkclasstype($class_1$) ↔ $class_1$ = “object”.class;

subarray3 : ¬ mkclasstype($class_1$) ≤ mkarraytype(ty);

subbasic1 : ¬ is_classtype(ty) ∧ ¬ is_arraytype(ty) → (ty ≤ ty_0 ↔ ty = ty_0);

subbasic2 : ¬ is_classtype(ty_0) ∧ ¬ is_arraytype(ty_0) → (ty ≤ ty_0 ↔ ty = ty_0);

end enrich

---

fieldname =
data specification
using string-append
fieldname = . .field (. .field : string ;);
variables fieldvar, $fieldvar_0$, $fieldvar_1$, $fieldvar_2$, $fieldvar_3$: fieldname;
end data specification

Generated axioms:

fieldname **freely generated by** .field;
sel : stringvar.field.field = stringvar;
inj : stringvar.field = stringvar_0.field ↔ stringvar = stringvar_0;
case : fieldvar.field.field = fieldvar;
elim : stringvar = fieldvar.field ↔ fieldvar = stringvar.field;

---

classname =
data specification
using string-append
classname = . .class (. .class : string ;);
variables class, class_0, class_1, class_2, class_3: classname;
end data specification

Generated axioms:

classname **freely generated by** .class;
sel : stringvar.class.class = stringvar;
inj : stringvar.class = stringvar_0.class ↔ stringvar = stringvar_0;
case : class.class.class = class;
elim : stringvar = class.class ↔ class = stringvar.class;

---

javatype =
data specification
using classname
javatype = boolean_type
| int_type
| short_type
| byte_type
| mkclasstype (. .class : classname ;) **with** is_classtype
| mkarraytype (. .type : javatype ;) **with** is_arraytype
| void_type **with** is_void_type
| abstract_type
;
variables ty: javatype;
end data specification

Generated axioms:

javatype **freely generated by** boolean_type, int_type, short_type, byte_type,
void_type, abstract_type, mkclasstype, mkarraytype;
disj : boolean_type ≠ int_type;
disj : boolean_type ≠ short_type;
disj : boolean_type ≠ byte_type;
disj : boolean_type ≠ mkclasstype(class);
disj : boolean_type ≠ mkarraytype(ty);
disj : boolean_type ≠ void_type;

disj : boolean_type ≠ abstract_type;
disj : int_type ≠ short_type;
disj : int_type ≠ byte_type;
disj : int_type ≠ mkclasstype(class);
disj : int_type ≠ mkarraytype(ty);
disj : int_type ≠ void_type;
disj : int_type ≠ abstract_type;
disj : short_type ≠ byte_type;
disj : short_type ≠ mkclasstype(class);
disj : short_type ≠ mkarraytype(ty);
disj : short_type ≠ void_type;
disj : short_type ≠ abstract_type;
disj : byte_type ≠ mkclasstype(class);
disj : byte_type ≠ mkarraytype(ty);
disj : byte_type ≠ void_type;
disj : byte_type ≠ abstract_type;
disj : mkclasstype(class) ≠ mkarraytype(ty);
disj : mkclasstype(class) ≠ void_type;
disj : mkclasstype(class) ≠ abstract_type;
disj : mkarraytype(ty) ≠ void_type;
disj : mkarraytype(ty) ≠ abstract_type;
disj : void_type ≠ abstract_type;
sel : mkarraytype(ty).type = ty;
sel : mkclasstype(class).class = class;
test : ¬ is_void_type(abstract_type);
test : is_void_type(void_type);
test : ¬ is_void_type(mkarraytype(ty));
test : ¬ is_void_type(mkclasstype(class));
test : ¬ is_void_type(byte_type);
test : ¬ is_void_type(short_type);
test : ¬ is_void_type(int_type);
test : ¬ is_void_type(boolean_type);
test : ¬ is_arraytype(abstract_type);
test : ¬ is_arraytype(void_type);
test : is_arraytype(mkarraytype(ty));
test : ¬ is_arraytype(mkclasstype(class));
test : ¬ is_arraytype(byte_type);
test : ¬ is_arraytype(short_type);
test : ¬ is_arraytype(int_type);
test : ¬ is_arraytype(boolean_type);
test : ¬ is_classtype(abstract_type);

test : ¬ is_classtype(void_type);

test : ¬ is_classtype(mkarraytype(ty));

test : is_classtype(mkclasstype(class));

test : ¬ is_classtype(byte_type);

test : ¬ is_classtype(short_type);

test : ¬ is_classtype(int_type);

test : ¬ is_classtype(boolean_type);

inj : mkarraytype(ty) = mkarraytype(ty_0) ↔ ty = ty_0;

inj : mkclasstype(class) = mkclasstype($class_0$) ↔ class = $class_0$;

case :
  ty = boolean_type
∨ ty = int_type
∨ ty = short_type
∨ ty = byte_type
∨ ty = mkclasstype(ty.class)
∨ ty = mkarraytype(ty.type)
∨ ty = void_type
∨ ty = abstract_type;

ex : is_void_type(ty) ↔ ty = void_type;

ex : is_arraytype(ty) ↔ (∃ ty_0. ty = mkarraytype(ty_0));

ex : is_classtype(ty) ↔ (∃ class. ty = mkclasstype(class));

elim : is_arraytype(ty) → (ty_0 = ty.type ↔ ty = mkarraytype(ty_0));

elim : is_classtype(ty) → (class = ty.class ↔ ty = mkclasstype(class));

---

intlista =
actualize olist-sort **with** int-total **by morphism**
elem → int; list → intlist; @ → @; + → +; .first →.first; .rest →.rest; # → #; ' →'; + → +; + → +; + → +; ++ → ++; rmdup → rmdup; .last →.last; .butlast →.butlast; rev → rev; mklist → mklist; -$_l$ → -$_l$; -1$_l$ → -1$_l$; -1$_l$ → -1$_l$;] →]; pos → pos;] →]; sublist → sublist; firstn → firstn; restn → restn; lastn → lastn; frome → frome; ∪ → ∪; \ → \; filter → filter; #$_{oc}$ → #$_{oc}$; ins → ins; merge → merge; sort → sort; < → <; < → <; ∈ → ∈; dups → dups; disj → disj; ⊑ → ⊑; ⊒ → ⊒; ⊆ → ⊆; perm → perm; ⊆$_m$ → ⊆$_m$; ≤$_{ordered}$ → ≤$_{ordered}$; <$_{ordered}$ → <$_{ordered}$; a → i; a_0 → i_0; b → i_1; c → i_2; x → ints; x_0 → $ints_0$; y → $ints_1$; z → $ints_2$; y_0 → $ints_3$; z_0 → $ints_4$; x_1 → $ints_5$; y_1 → $ints_6$; z_1 → $ints_7$; x_2 → $ints_8$; y_2 → $ints_9$; z_2 → $ints_{10}$
end actualize

---

int-total =
enrich intnat **with**

axioms

totality : i < j ∨ i = j ∨ j < i;

end enrich

---

olist-sort =
enrich olist **with**
functions sort : list → list ;

axioms

Ordered : $\leq_{ordered}$(sort(x));

Perm : perm(x, sort(x));

end enrich

---

olist =
enrich olista **with**
functions

ins	:	elem × list	→	list ;
merge	:	list × list	→	list ;

predicates

$\leq_{ordered}$	:	list;
$<_{ordered}$	:	list;

axioms

le-e : $\leq_{ordered}$(@);

le-o : $\leq_{ordered}$(a ');

le-r : $\leq_{ordered}$(a ' + b ' + x) ↔ ¬ b < a ∧ $\leq_{ordered}$(b + x);

ls-e : $<_{ordered}$(@);

ls-o : $<_{ordered}$(a ');

ls-r : $<_{ordered}$(a ' + b ' + x) ↔ a < b ∧ $<_{ordered}$(b + x);

ins-e : ins(a, @) = a ';

ins-y : ¬ b < a → ins(a, b ' + x) = a + b + x;

ins-n : b < a → ins(a, b ' + x) = b + ins(a, x);

end enrich

---

olista =
actualize list-perm **with** gelem **by morphism**

end actualize

---

assoclista =
actualize list-perm **with** pair **by morphism**
elem → pair; list → assoclist; @ → @; + → +; .first →.first; .rest →.rest; # → #; ' →'; + → +; + → +; + → +; ++ → ++; rmdup → rmdup; .last →.last; .butlast →.butlast; rev → rev; mklist → mklist; -$_l$ → -$_l$; -1$_l$ → -1$_l$; -1$_l$ → -1$_l$;] →]; pos → pos;] →]; sublist → sublist; firstn → firstn; restn → restn; lastn → lastn; frome → frome; ∪ → ∪; \ → \; filter → filter; $\#_{oc}$ → $\#_{oc}$; < → <; ∈ → ∈; dups → dups; disj → disj; ⊑ → ⊑; ⊒ → ⊒; ⊆ → ⊆; perm → perm; $\subseteq_m$ → $\subseteq_m$; a → p; b → p$_0$; c → p$_1$; a$_0$ → p$_2$; x → ax; x$_0$ → ax$_0$; x$_1$ → ax$_1$; x$_2$ → ax$_2$; y → ay; y$_0$ → ay$_0$; y$_1$ → ay$_1$; y$_2$ → ay$_2$; z → az; z$_0$ → az$_0$; z$_1$ → az$_1$; z$_2$ → az$_2$
end actualize

---

ostore =
enrich ostore+oset **with**
functions
. .min : store → elem ;
. .keys : store → set ;

axioms

Min-in : st ≠ ∅ → st.min ∈ st;

Min-min : a ∈ st → ¬ a < st.min;

Keys : a ∈ st.keys ↔ a ∈ st;

end enrich

---

ostore+oset = oset + ostorea

---

pair =
generic data specification
parameter elemdata
pair = . × . **prio** 9 (. .1 : elem ; . .2 : data ;) **prio** 9;
variables p, p_0, p_1, p_2: pair;
end generic data specification

Generated axioms:

pair **freely generated by** ×;
sel : (a × d).2 = d;

sel : (a × d).1 = a;

inj : a × d = a_0 × d_0 ↔ a = a_0 ∧ d = d_0;

case : p.1 × p.2 = p;

elim : a = p.1 ∧ d = p.2 ↔ p = a × d;

---

ostorea =
actualize store **with** gelem **by morphism**

end actualize

---

oset =
enrich oseta **with**
functions
. .min : set → elem ;
. .butmin : set → set ;
. .max : set → elem ;
. .butmax : set → set ;
predicates
. < . : elem × set;
. < . : set × elem;

axioms

Min-in : $s \neq \emptyset \rightarrow s.min \in s$;

Min-min : $a \in s \rightarrow \neg\ a < s.min$;

Butmin : $s \neq \emptyset \rightarrow s.butmin = s - s.min$;

Max-in : $s \neq \emptyset \rightarrow s.max \in s$;

Max-max : $a \in s \rightarrow \neg\ s.max < a$;

Butmax : $s \neq \emptyset \rightarrow s.butmax = s - s.max$;

Less : $a < s \leftrightarrow (\forall\ b.\ b \in s \rightarrow a < b)$;

Greater : $s < a \leftrightarrow (\forall\ b.\ b \in s \rightarrow b < a)$;

end enrich

---

oseta =
actualize set-union **with** gelem **by morphism**

end actualize

---

store =
generic specification
parameter elemdata **using** nat **target**
sorts store, elemdata;
constants $\emptyset$: store;
functions

. × .	:	elem × data	→	elemdata	**prio** 9;
. [.]	:	store × elem × data	→	store	;
. [.]	:	store × elem	→	data	**prio** 2;
# .	:	store	→	nat	;
. − .	:	store × elem	→	store	**prio** 9 **left**;

predicates

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

variables
st, st_0, st_1, st_2: store;
elemdatavar: elemdata;

axioms

store **generated by** $\emptyset$,];
elemdata **freely generated by** ×;
Extension : $st_1 = st_2 \leftrightarrow (\forall\ a.\ (a \in st_1 \leftrightarrow a \in st_2) \wedge st_1[a] = st_2[a])$;

In-empty : $\neg\ a \in \emptyset$;

In-insert : $a \in st[b, d] \leftrightarrow a = b \vee a \in st$;

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

At-other : $a \neq b \rightarrow st[b, d][a] = st[a]$;

In-store : $a \times d \in st \leftrightarrow a \in st \wedge st[a] = d$;

Subset : $st_1 \subseteq st_2 \leftrightarrow (\forall\ a.\ a \in st_1 \rightarrow a \in st_2 \wedge st_1[a] = st_2[a])$;

Size-empty : $\#\ \emptyset = 0$;

Size-insert : $\neg\ a \in st \rightarrow \#\ st[a, d] = \#\ st + 1$;

Del-in : $a \in st - b \leftrightarrow a \neq b \wedge a \in st$;

Del-at : $a \neq b \rightarrow st - b[a] = st[a]$;

end generic specification

---

set-union =
enrich set-basic **with**
functions

{ . }	:	elem	→	set	;
. ∪ .	:	set × set	→	set	**prio** 9;
. ∩ .	:	set × set	→	set	**prio** 9;
. \ .	:	set × set	→	set	**prio** 9;
. − .	:	set × elem	→	set	**prio** 9 **left**;

axioms

One : {a} = ∅ ++ a;

Union : $a \in s_1 \cup s_2 \leftrightarrow a \in s_1 \vee a \in s_2$;

Intersect : $a \in s_1 \cap s_2 \leftrightarrow a \in s_1 \wedge a \in s_2$;

Difference : $a \in s_1 \setminus s_2 \leftrightarrow a \in s_1 \wedge \neg\, a \in s_2$;

Delete : $a \in s - b \leftrightarrow a \neq b \wedge a \in s$;

end enrich

---

set-basic =
generic specification
parameter elem **using** nat **target**
sorts set;
constants ∅ : set;
functions

. ++ .	:	set × elem	→	set	**prio** 9 **left**;
# .	:	set	→	nat	;

predicates

. ∈ .	:	elem × set;
. ⊆ .	:	set × set;

variables s, s_0, s_1, s_2: set;

axioms

set **generated by** ∅, ++;

Extension : $s_1 = s_2 \leftrightarrow (\forall\, a.\ a \in s_1 \leftrightarrow a \in s_2)$;

In-empty : $\neg\, a \in \emptyset$;

In-insert : $a \in s \mathbin{++} b \leftrightarrow a = b \vee a \in s$;

Size-empty : $\#\, \emptyset = 0$;

Size-insert : $\neg\, a \in s \rightarrow \#(s \mathbin{++} a) = \#\, s + 1$;

Subset : $s_1 \subseteq s_2 \leftrightarrow (\forall\, a.\ a \in s_1 \rightarrow a \in s_2)$;

end generic specification

---

list-perm =
enrich list-set **with**
functions $\#_{oc}$: elem × list → nat ;
predicates

perm	:	list × list;
. $\subseteq_m$.	:	list × list;

axioms

oc-e : $\#_{oc}$(a, @) = 0;

oc-y : $\#_{oc}$(a, a ' + x) = $\#_{oc}$(a, x) + 1;

oc-n : a ≠ b → $\#_{oc}$(a, b ' + x) = $\#_{oc}$(a, x);

msubset : x $\subseteq_m$ y ↔ (∀ a. $\#_{oc}$(a, x) ≤ $\#_{oc}$(a, y));

perm : perm(x, y) ↔ x $\subseteq_m$ y ∧ y $\subseteq_m$ x;

end enrich

---

list-set =
enrich list-del **with**
functions

. ∪ .	:	list × list	→	list	**prio** 9;
. \ .	:	list × list	→	list	**prio** 9;
filter	:	list × list	→	list	;

predicates . ⊆ . : list × list;

axioms

subset : x ⊆ y ↔ (∀ a. a ∈ x → a ∈ y);

union : x ∪ y = rmdup(x + y);

diff-e : @ \ y = @;

diff-y : a ∈ y → (a ' + x) \ y = x \ y;

diff-n : ¬ a ∈ y → (a ' + x) \ y = a + x \ y;

filt-e : filter(@, y) = @;

filt-y : a ∈ y → filter(a ' + x, y) = a ' + filter(x, y);

filt-n : ¬ a ∈ y → filter(a ' + x, y) = filter(x, y);

end enrich

---

list-del =
enrich list-last **with**
functions

. -$_l$.	:	list × elem	→	list	**prio** 9;
. -1$_l$.	:	list × elem	→	list	**prio** 9;
. -1$_l$.	:	list × nat	→	list	**prio** 9;
. [.]	:	list × nat	→	elem	**prio** 2;
pos	:	elem × list	→	nat	;
. [.]	:	list × nat × elem	→	list	;
sublist	:	nat × nat × list	→	list	;
firstn	:	nat × list	→	list	;
restn	:	nat × list	→	list	;
lastn	:	nat × list	→	list	;
frome	:	list × elem	→	list	;

axioms

del-e : @ -$_l$ a = @;

del-y : (a ' + x) -$_l$ a = x -$_l$ a;

del-n : a ≠ b → (b ' + x) -$_l$ a = b ' + x -$_l$ a;

del1-e : @ -1$_l$ a = @;

del1-y : (a ' + x) -1$_l$ a = x;

del1-n : a ≠ b → (b ' + x) -1$_l$ a = b ' + x -1$_l$ a;

delpos-empty : @ -1$_l$ n = @;

delpos-base : (a ' + x) -1$_l$ 0 = x;

delpos-rec : (a ' + x) -1$_l$ n + 1 = a + x -1$_l$ n;

get-zero : a ' + x[0] = a;

get-succ : a ' + x[n + 1] = x[n];

pos-e : pos(a, @) = 0;

pos-y : pos(a, a ' + x) = 0;

pos-n : a ≠ b → pos(a, b ' + x) = pos(a, x) + 1;

put-zero : b ' + x[0, a] = a + x;

put-succ : b ' + x[n + 1, a] = b + x[n, a];

sublist-good : sublist(# x, # y, x + y + z) = y;

firstN-zero : firstn(0, x) = @;

firstN-rec : firstn(n + 1, x) = x.first + firstn(n, x.rest);

restN-zero : restn(0, x) = x;

restN-rec : restn(n + 1, x) = restn(n, x.rest);

lastN-zero : lastn(0, x) = @;

lastN-rec : lastn(n + 1, x) = lastn(n, x.butlast) + x.last;

fromE-empty : frome(@, a) = @;

fromE-yes : frome(a ' + x, a) = a ' + x;

fromE-no : a ≠ b → frome(a ' + x, b) = frome(x, b);

end enrich

---

list-last =
enrich list-dup **with**
functions

. .last	:	list	→	elem ;
. .butlast	:	list	→	list ;
rev	:	list	→	list ;
mklist	:	elem × nat	→	list ;

predicates

. ⊑ .	:	list × list;
. ⊒ .	:	list × list;

axioms

last : x ≠ @ → x.butlast + x.last = x;

rev-e : rev(@) = @;

rev-r : rev(a ' + x) = rev(x) + a;

mk-len : # mklist(a, n) = n;

mk-elem : a ∈ mklist(b, n) → a = b;

prefix : x ⊑ y ↔ (∃ z. x + z = y);

postfix : x ⊒ y ↔ (∃ z. z + x = y);

end enrich

---

list-dup =
enrich list **with**
functions

. ++ .	:	list × elem	→	list	**prio** 9 **left**;
rmdup	:	list	→	list	;

predicates

dups	:	list;
disj	:	list × list;

axioms

rmdup-e : rmdup(@) = @;

rmdup-y : a ∈ x → rmdup(a ' + x) = rmdup(x);

rmdup-n : ¬ a ∈ x → rmdup(a ' + x) = a + rmdup(x);

adjoin-in : a ∈ x → x ++ a = x;

adjoin-notin : ¬ a ∈ x → x ++ a = a + x;

dups : dups(x) ↔ (∃ a, x_0, y, z. x = x_0 + a + y + a + z);

disjoint : disj(x, y) ↔ (∀ a. ¬ (a ∈ x ∧ a ∈ y));

end enrich

---

list =
enrich list-data **with**
functions

. '	:	elem	→	list	;
. + .	:	list × list	→	list	**prio** 9;
. + .	:	list × elem	→	list	**prio** 9;
. + .	:	elem × elem	→	list	**prio** 9;

predicates . ∈ . : elem × list;

axioms

Nil : @ + x = x;

Cons : (a + x) + y = a + x + y;

One : a ' = a + @;

Last : x + a = x + a ';

Two : a + b = a ' + b ';

in : a ∈ x ↔ (∃ y, z. x = y + a + z);

end enrich

---

list-data =
generic data specification
parameter elem **using** nat
list = @
| . + . **prio** 9 (. .first : elem ; . .rest : list ;) **prio** 9
;
variables x, y, z, x_0, y_0, z_0, x_1, y_1, z_1, x_2, y_2, z_2: list;
size functions # . : list → nat ;
order predicates . < . : list × list;
end generic data specification

Generated axioms:

list **freely generated by** @, +;
disj : @ ≠ a + x;
sel : (a + x).rest = x;
sel : (a + x).first = a;
inj : a + x = a_0 + x_0 ↔ a = a_0 ∧ x = x_0;
case : x = @ ∨ x = x.first + x.rest;
size : #(a + x) = (# x)+1;
size : # @ = 0;
ref : ¬ x < x;
trans : x < x_0 ∧ x_0 < x_1 → x < x_1;
less : x_0 < a + x ↔ x_0 = x ∨ x_0 < x;
less : ¬ x < @;
elim : x ≠ @ → (a = x.first ∧ x_0 = x.rest ↔ x = a + x_0);

---

int-pot =
enrich int-square, intnat **with**
functions . ^ . : int × int → int **prio** 12;

axioms

Pot-zero : j ^ 0 = 1;
Pot-pos : 0 < i → j ^ i = j ^ i -1 ∗ j;
Pot-neg : i < 0 → j ^ i = 1 / j ^ abs(i);

end enrich

---

int-square =
enrich int-div **with**
functions
. ^2 : int → int ;
sqrt : int → int ;

axioms

Square-def : i ^2 = i * i;

Sqrt-def : 0 ≤ i → sqrt(i)^2 ≤ i ∧ i < sqrt(i)+1 ^2;

end enrich

---

intnat =
enrich int-div, nat-div **with**
functions

i→n	:	int	→	nat	;
n→i	:	nat	→	int	;

axioms

Nat2int-zero : n→i(0) = 0;

Nat2int-succ : n→i(n +1) = n→i(n)+1;

Int2nat-zero : i→n(0) = 0;

Int2nat-pos : 0 < i → i→n(i) = i→n(i - 1) + 1;

Int2nat-neg : i < 0 → i→n(i) = i→n(~ i);

end enrich

---

int-div =
enrich int-mult **with**
functions

. / .	:	int × int	→	int	**prio** 11;
. % .	:	int × int	→	int	**prio** 11;

axioms

Div-def : 0 ≤ i ∧ 0 < j → i / j * j ≤ i ∧ i < (i / j)+1 * j;

Div-left : j ≠ 0 → ~ i / j = ~(i / j);

Div-right : j ≠ 0 → i / ~ j = ~(i / j);

Mod-def : j ≠ 0 → i = i / j * j + i % j;

end enrich

---

int-mult =
enrich int-abs **with**
functions . * . : int × int → int **prio** 10;

axioms

Mult-zero : i * 0 = 0;

Mult-succ : i * j +1 = i * j + i;

Mult-pred : i * j -1 = i * j - i;

end enrich

---

int-abs =
enrich int-basic2 **with**
functions

. - .	:	int × int	→	int	**prio** 8 **left**;
~ .	:	int	→	int	;
abs	:	int	→	int	;

predicates

. ≤ .	:	int × int;
. > .	:	int × int;
. ≥ .	:	int × int;

axioms

Sub-def : i + (j - i) = j;

Minus-def : ~ i = 0 - i;

Abs-neg : j < 0 → abs(j) = 0 - j;

Abs-pos : ¬ j < 0 → abs(j) = j;

i ≤ j ↔ ¬ j < i;

i > j ↔ j < i;

i ≥ j ↔ ¬ i < j;

end enrich

---

int-basic2 =
enrich int-basic1 **with**
functions . + . : int × int → int **prio** 9;

axioms

i + 0 = i;

i + j +1 = (i + j)+1;

i + j -1 = (i + j)-1;

end enrich

---

int-basic1 =
specification
sorts int;
functions

. +1	:	int	→	int	;
. -1	:	int	→	int	;

predicates . < . : int × int;
variables i, i_0, j, j_0: int;

axioms

int **generated by** 0, +1, -1;

succpred : i +1 -1 = i;

predsucc : i -1 +1 = i;

irref : ¬ i < i;

trans : i < j ∧ j < j_0 → i < j_0;

lsrec : i < j ↔ i +1 = j ∨ i +1 < j;

lssucc : i +1 < j ↔ i < j -1;

end specification

---

nat-div =
enrich nat-mult **with**
functions
. / . : nat × nat → nat **prio** 11;
. % . : nat × nat → nat **prio** 11;

axioms

Divdef : n ≠ 0 → m / n ∗ n ≤ m ∧ m < (m / n)+1 ∗ n;

Moddef : n ≠ 0 → m = m / n ∗ n + m % n;

end enrich

---

nat-mult =
enrich nat **with**
functions . ∗ . : nat × nat → nat **prio** 10;

axioms

m ∗ 0 = 0;

m ∗ n +1 = m ∗ n + m;

end enrich

---

nat =
enrich nat-basic2 **with**
functions . - . : nat × nat → nat **prio** 8 **left**;
predicates
. ≤ . : nat × nat;
. > . : nat × nat;
. ≥ . : nat × nat;

axioms

m - 0 = m;

m - n +1 = (m - n)-1;

m ≤ n ↔ ¬ n < m;

m > n ↔ n < m;

m ≥ n ↔ ¬ m < n;

end enrich

---

nat-basic2 =
enrich nat-basic1 **with**
functions . + . : nat × nat → nat **prio** 9;
variables m, n_0: nat;

axioms

n + 0 = n;

m + n +1 = (m + n)+1;

$n < n_0 \vee n = n_0 \vee n_0 < n$;

1 = 0 +1;

$0 \neq 1$;

end enrich

---

nat-basic1 =
data specification
nat = 0
| . +1 (. -1 : nat ;)
;
variables n: nat;
order predicates . < . : nat × nat;
end data specification

Generated axioms:

nat **freely generated by** 0, +1;

disj : $0 \neq n +1$;

sel : n +1 -1 = n;

inj : $n +1 = n_0 +1 \leftrightarrow n = n_0$;

case : $n = 0 \vee n = n -1 +1$;

ref : $\neg\, n < n$;

trans : $n < n_0 \wedge n_0 < n_1 \rightarrow n < n_1$;

less : $n_0 < n +1 \leftrightarrow n_0 = n \vee n_0 < n$;

less : $\neg\, n < 0$;

elim : $n \neq 0 \rightarrow (n_0 = n -1 \leftrightarrow n = n_0 +1)$;

---

string-append =
enrich string-data **with**
functions
. + . : string × string → string **prio** 9;
. ’ : char → string ;
variables stringvar, stringvar_0: string;

axioms

chartostring : char ’ = char + “”;

append-base : “” + str = str;

append-rec : (char + str) + str_0 = char + str + str_0;

end enrich

---

string-data =
data specification
using char
string = “”
| . + . **prio** 9 (. .char1 : char ; . .rstring : string ;) **prio** 9
;
variables str, str_0, str_1, str_2, str_3: string;
end data specification

Generated axioms:

string **freely generated by** “”, +;
disj : “” $\neq$ $char_0$ + str;
sel : ($char_0$ + str).rstring = str;
sel : ($char_0$ + str).char1 = $char_0$;
inj : $char_0$ + str = $char_1$ + str_0 $\leftrightarrow$ $char_0$ = $char_1$ $\wedge$ str = str_0;
case : str = “” $\vee$ str = str.char1 + str.rstring;
elim : str $\neq$ “” $\rightarrow$ ($char_0$ = str.char1 $\wedge$ str_0 = str.rstring $\leftrightarrow$ str = $char_0$ + str_0);

---

char =
specification
sorts char;
variables char, $char_0$, $char_1$, $char_2$: char;

axioms

char **generated by** “a”, “b”, “c”, “d”, “e”, “f”, “g”, “h”, “i”, “j”, “k”, “l”, “m”, “n”, “o”, “p”, “q”, “r”, “s”, “t”, “u”, “v”, “w”, “x”, “y”, “z”, “A”, “B”, “C”, “D”, “E”, “F”, “G”, “H”, “I”, “J”, “K”, “L”, “M”, “N”, “O”, “P”, “Q”, “R”, “S”, “T”, “U”, “V”, “W”, “X”, “Y”, “Z”, “!”, “@”, “#”, “$”, “%”, “^”, “&”, “*”, “_”, “-”, “+”, “=”, “~”, “<”, “>”, “?”, “/”, “0”, “1”, “2”, “3”, “4”, “5”, “6”, “7”, “8”, “9”;

end specification

---

bool =
data specification
bool = true
| false
;
variables boolvar, $boolvar_0$: bool;
end data specification

Generated axioms:

bool **freely generated by** true, false;
disj : $\neg$ (true $\leftrightarrow$ false);
case : (boolvar = true $\leftrightarrow$ true) $\vee$ (boolvar = true $\leftrightarrow$ false);

---

elemdata = elem + data

---

gelem =
generic specification
    parameter oelem **target**
end generic specification

---

data =
specification
    sorts data;
    variables d, d_0, d_1, d_2: data;
end specification

---

oelem =
enrich elem **with**
    predicates . < . : elem × elem;

axioms

    irreflexivity : ¬ a < a;

    transitivity : a < b ∧ b < c → a < c;

    totality : a < b ∨ a = b ∨ b < a;

end enrich

---

elem =
specification
    sorts elem;
    variables a, b, c: elem;
end specification

$\subseteq$ 92
$\subseteq$ m 78
$\subseteq$ m 77
$\subseteq$ m 100
$\subseteq$ m 92
$\subseteq$ m 95
$\subseteq$ m 93
$\times$ 77
$\times$ 80
$\sqsubseteq$ 78
$\sqsubseteq$ 100
$\sqsubseteq$ 77
$\sqsubseteq$ 95
$\sqsubseteq$ 92
$\sqsubseteq$ 93
$\sqsupseteq$ 78
$\sqsupseteq$ 100
$\sqsupseteq$ 77
$\sqsupseteq$ 95
$\sqsupseteq$ 92
$\sqsupseteq$ 93
$\emptyset$ 104
$\emptyset$ 103

Bibliography

[AF99] J. Alves-Foss, editor. *Formal Syntax and Semantics of Java*. Springer LNCS 1523, 1999.

[Bec00] B. Beckert. A Dynamic Logic for Java Card. In *Proceedings, 2nd ECOOP Workshop on Formal Techniques for Java Programs, Cannes, France*, June 2000.

[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. Springer, 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.

[DEJ+00] S. Drossopoulou, S. Eisenbach, B. Jacobs, G. T. Leavens, P. Müller, and A. Poetzsch-Heffter, editors. *Formal Techniques for Java Programs. Proceedings of the ECOOP 2000 Workshop.* Technical Report 269, Fernuniversität Hagen, 2000. Available from `www.informatik.fernuni-hagen.de/pi5/publications.html`.

[GJS96] J. Gosling, B. Joy, and G. Steele. *The Java Language Specification, Edition 1.0*. Addison-Wesley, 1996.

[Har79] D. Harel. *First Order Dynamic Logic*. LNCS 68. Springer, Berlin, 1979.

[HJ00] M. Huisman and B. Jacobs. Java Program Verification via a Hoare Logic with Abrupt Termination. In T. Maibaum, editor, *Fundamental Approaches to Software Engineering (FASE'00)*. Springer LNCS 1783, 2000.

[Jav00] *Java Card 2.1.1 Specification*, 2000. http://java.sun.com/products/javacard/.

[JLMPH99] B. Jacobs, G. T. Leavens, P. Müller, and A. Poetzsch-Heffter, editors. *Formal Techniques for Java Programs*. Technical Report 251, Fernuniversität Hagen, 1999.

[JvdBH+98] B. Jacobs, J. van den Berg, M. Huisman, M. van Berkum, U. Hensel, and H. Tews. Reasoning about Java Classes. In *In Proceedings of Object-Oriented Programming Systems, Languages and Applications (OOPSLA'98)*, October 1998.

[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.

[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, Berlin, 1995.

[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*. Kluwer Academic Publishers, Dordrecht, 1998.

[vO00] David von Oheimb. Axiomatic semantics for Java^{light} 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, Fernuniversität Hagen, 2000.

[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.