Formal Syntax and Semantics of Java

Article *in* Lecture Notes in Computer Science · January 1999

CITATIONS
39

READS
1,496

1 author:

Jim Alves-Foss
University of Idaho
148 PUBLICATIONS **1,347** CITATIONS

Lecture Notes in Computer Science 1523

Edited by G. Goos, J. Hartmanis and J. van Leeuwen

Springer
Berlin
Heidelberg
New York
Barcelona
Hong Kong
London
Milan
Paris
Singapore
Tokyo

Jim Alves-Foss (Ed.)

Formal Syntax and Semantics of Java

Springer

Series Editors

Gerhard Goos, Karlsruhe University, Germany
Juris Hartmanis, Cornell University, NY, USA
Jan van Leeuwen, Utrecht University, The Netherlands

Volume Editor

Jim Alves-Foss
University of Idaho
Center for Secure and Dependable Software
Moscow, ID 83844-1010, USA
E-mail: jimaf@cs.uidaho.edu

Cataloging-in-Publication data applied for

Die Deutsche Bibliothek - CIP-Einheitsaufnahme

Formal syntax and semantics of Java / Jim Alves-Foss (ed.). - Berlin ; Heidelberg ; New York ; Barcelona ; Hong Kong ; London ; Milan ; Paris ; Singapore ; Tokyo : Springer, 1999
(Lecture notes in computer science ; Vol. 1523)
ISBN 3-540-66158-1

CR Subject Classification (1998): D.3, F.3

ISSN 0302-9743
ISBN 3-540-66158-1 Springer-Verlag Berlin Heidelberg New York

Typesetting: Camera-ready by author
SPIN: 10692825 06/3142 – 5 4 3 2 1 0 Printed on acid-free paper

Preface

The Java language began as a program for embedded devices, specifically consumer electronics. The driving concepts behind the language included portability, enabling reuse as the underlying processors were changed in new versions of the device; and simplicity, to keep the best aspects of related languages and to throw out the fluff.

As the World Wide Web blasted onto the scene, the Java development team realized that with some additional functionality (specifically a GUI interface API) a language such as Java could readily be used to enable the specification of executable content on web pages. The inclusion of Java support in the Netscape 2.0 browser provided enough publicity and support for the language that it immediately became the de facto standard language for programming executable content.

Although Java has its roots in embedded systems and the web, it is important to realize that it is a fully functional high-level programming language that can provide users with a wide range of functionality and versatility. In "The Java Language: A White Paper," Sun Microsystems developers describe Java as:

> Java: A simple, object-oriented, distributed interpreted, robust, secure, architecture neutral, aportable, high-performance, multithreaded, and dynamic language.

This list of terms describes Sun's design goals for the Java language and portrays some of the most important features of the language. Simplicity refers to a small language learning curve, similarity to C and C++, and removal of some standard (but dangerous) features of languages such as C++. For example, Java does not use pointers as in C and C++, but rather references as in Pascal. This avoids potential pointer manipulation errors by the programmer. In addition, Java provides no header files as in C and C++, thus enabling more automated bookkeeping (although compilers are not currently utilizing the full potential of such a system, resulting in sometimes awkward coding constructs).

As an object-oriented language Java supports the concepts of abstract data typing as encapsulated in many object-oriented programming languages. The user defines a class which specifies a collection of data items and methods to operate on those items. Data and methods of the class can exist with the class (one instance per program execution) or with specific instances (objects) of the class. Classes are arranged in a hierarchy to provide mechanisms for inheritance and reuse. Java classes are further arranged into packages that provide some additional protection and bookkeeping for Java programmers. To assist the programmer, there exists a set of predefined classes that are provided with Java development/execution environments. Some of these classes are essential as they provide a bridge between the portable Java code and the underlying native operating system.

As a distributed language, the Java API provides functionality for inter process communication and remote data access. It is important to understand that

this distributed nature of Java is solely the benefit of good API classes and not any inherent distributed or parallel programming capabilities.

As an interpreted language, Sun means that the user creates an intermediate file containing the "bytecode" implementation of the program. It is the byte code that is subsequently interpreted and not the raw Java source code. The Java interpreter and the supporting run-time system implement what is called the Java Virtual Machine.

As a robust language, Java is strongly typed and does not use pointers. These two features greatly reduce the possibility of very common software flaws. In addition, Java has both built-in automatic garbage collection routine to prevent memory leakage and exception handling. The exception handling allows almost all errors to be caught and managed by the software.

As a secure language, Java provides for access control restrictions on class or object methods and data items. These may be implemented as part of the basic protection attribute of the items or through a run-time security monitor.

As architecture neutral and portable, Java functionality does not rely on any underlying architecture specifics, thus allowing the code (or even the byte code) to be executed on any machine with a virtual machine implementation.

As a high-performance language, Java is meant to execute well with respect to similar high-level languages. The use of special "just in time" compilers or other features may improve performance even more.

As a multithreaded language, Java permits the development of user specified concurrent threads of control, as well as synchronization mechanisms to establish consistency between the users.

As a dynamic language, Java is intended to be able to dynamically load code from the network and execute the new version of the code in the current virtual machine as opposed to recompiling the whole project.

The above list of features describes Sun's view of the Java language, a view that is shared by many users. We will assume that these features represent the base capabilities of the language. In the rest of this part we describe various features of Java, highlighting their challenges for formal method specifications of the language.

Java Basic Data Types

The Java language includes several built-in basic data types. These include boolean, char and numeric types: byte, short, integer, long for (8, 16, 32 and 64-bit integer calculations) and float and double for (32 and 64-bit floating point operations). Java provides standard operations for these types with the few special features discussed below. In addition Java has a reference data type for use of objects and a special reference data type, the array. Like Pascal and unlike C, the Java reference data type can only be copied; no increment or other operations can be performed on it.

Java is strongly typed and only permits limited type casting or automatic conversions. This strengthens the reliability of the language. The only problems is that the explicit casting of integer values (with either 32 or 64-bits) to smaller

integers such as byte results in truncation of the high-order bits, resulting in information loss and even potential change in sign.

Java Classes

All user-defined Java data types are specified using a class definition. A class defines the fields and methods of the object and their appropriate access modifiers. With Java 1.1 and later, users have the ability to define subclasses and anonymous classes within their own classes.

Java Files and Packages

A Java program consists of one or more packages, each of which consist of a collection of Java classes. A class within a single package has a stronger trust relationship with other classes in that package than with those outside of the package. In addition, the package relationship provides the ability to utilize a hierarchical naming convention for Java classes.

Each Java class is defined within a single file. Although a file may contain more than one class definition, only one file in that class may be declared public (and must be named the same as the source file). Classes within the same file are implicitly within the same package.

Exception Handling

Java provides a flexible exception handling capability. Any time an exception occurs the violating routine can throw a named exception, abruptly terminating the statement. All Java statements can be encapsulated within a try-catch statement. If the enclosed statement throws an exception that is specified within the catch clause, the violating statement is terminated and the code in the catch clause is executed. Otherwise, the thrown exception is propagated up the call hierarchy.

The Java Virtual Machine

All Java programs are compiled into an intermediate form, the Java Byte Code. The Java Virtual Machine (JVM) reads and executes the byte code. In addition the JVM is responsible for downloading and verifying byte code from local and remote sources. The virtual machine checks access rights to class fields and methods, provides links to native code libraries and even implements security monitors for further limited access.

Formal Methods

"Formal methods" is a term that refers to the application of formal mathematical models to computer systems and subsystems. The intent of this book is

to provide a forum for the presentation of a variety of approaches to formal specifications, execution models and analysis of Java programs.

There are several styles of formal methods, a few of which are used in this book. The most common approach to specifying the meaning of a program currently in use is operational semantics. The purpose behind an operational semantics is to provide an abstract model of the internal state of the computer (as referenced by the program) and to specify the modifications of that state with respect to program statements and expressions. A typical semantic clause could be of the form:

$$\frac{< c, \sigma > \rightarrow < c', \sigma' >}{< c; c_1, \sigma > \rightarrow < c'; c_1, \sigma' >}$$

This clause states that if partial execution of command c while in state ?, will result the remaining command c' to be executed in state ?', then the semantics of the sequential composition of c and c1 will behave similarly.

Another type of semantics used in this book is denotational semantics. In this form of semantics, each statement, expressions and other programming language constructs are mapped into functions. These functions are defined as mapping semantic domains to semantic domains. These domains may represent anything from the basic data values stored in variables to the effects of complex recursive functions on the state of the system.

Acknowledgements

I would like to thank all of the contributors to this volume, who patiently waited for me to assemble this document. They also helped edit and review each other's work as well as their own. In addition, I would like to thank the following reviewers for their help: B. Auernheimer, J. Buffenbarger, P. Ciancarini, C. Pusch, P. Sestoft, E.T. Schubert, A. Sobel, and A. Wabenhorst.

Jim Alves-Foss

Table of Contents

Formal Grammar for Java

Jim Alves-Foss and Deborah Frincke

Center for Secure and Dependable Software, Department of Computer Science
University of Idaho, Moscow ID 83844-1010, USA

1 Introduction

This chapter presents an attribute grammar for the Java programming language (v. 1.1). This grammar is derived from the LALR grammar presented in the *Java Language Specification* (JLS) [1]. The purpose of this grammar is to formally specify not only the syntactic structure of Java programs, but also their static semantics. Specifically, in this chapter we try to formally capture all aspects of the language that would result in compile-time errors. These errors include, but are not limited to:

- Type checking for assignment statements, ensuring that the type of the right-hand side of the statement is assignment compatible with the left hand side.
- Type checking expression operands, ensuring that they are of compatible types.
- Type checking method parameters, ensuring that they are the correct type and number.
- Checking for duplicate variable and method names.
- Checking for undefined variables.

We do not actually capture all errors, but a sufficient body of them to demonstrate the approach we are using. We have left comments within the syntax for portions where we believe addition semantic checks are needed, as an exercise to the reader. The grammar is written using a BNF-like notation of the form of productions:

```
<NonTerm> ::= exp1
                  semantic action 1
            | exp2
                  semantic action 2
```

where the left hand side (LHS) non-terminal <NonTerm> can be defined in terms of either right hand side (RHS) expression *exp1* or *exp2*. Within the productions we use some abbreviations to shorten the specification. We define the following abbreviations: $exp^?$ to specify optional inclusion of the expression, exp^+ to specify one or more occurrences of the expression, exp^* to specify zero or more occurrences of the expression. On the lines immediately following the RHS expression are the semantic actions for that production. These actions involve propagation of the attributes, up and down the parse tree, and static correctness

Jim Alves-Foss (Ed.): Formal Syntax and Semantics of Java, LNCS 1523, pp. 1–40, 1999.

Table 1. Language unit caller/callee relationship.

	Callees												
Callers	names & lits	pkgs	types	mods	inter decl	class decl	meth decl	field decl	var decl	inits	cnstr	blocks & stmts	exprs
names & lits	X												
pkgs	X	X					X	X					
types	X		X										
mods				X									
inter decl	X			X	X	X	X	X					
class decl	X			X	X	X	X	X			X		
meth decl	X		X	X					X			X	
field decl	X		X	X					X				
var decl													
inits										X		?	X
cnstr	X			X								X	?
blocks & stmts	X		X						X			X	X
exprs	X		X		X	X			X			X	X

checks (compile-time errors). Associated with every potential compiler-time error we have placed the semantic action **ERROR** which displays a string error message related to the compile time error.

1.1 Logical Units of the Grammar

The full grammar is broken down into several logical units, each consisting of a collection of productions that define non-terminals in the grammar. Table 1 depicts the hierarchical relationship between these units. A logical unit is said to *call* another logical unit if it uses a non-terminal of the other logical unit in the RHS of one of its productions. The called logical unit is the *callee*. Note that there are several self- and circular-references in this table. These logical units are defined as follows:

names and literals - these define the lowest level constructs in the Java language and provide abstract representations of the low-level syntax of the Java language.

packages - these define the overall structure of the Java source code files, package and import specifications.

types - these define the type definition facilities of Java, which includes primitive types, reference types, class types, array types and interface types.

modifiers - these define the modifiers of various Java constructs. Such modifiers include protection modes (e.g., public, private) and status (e.g., static, final).

interface declarations - these define the form and structure of interfaces specifications.
class declarations - these define the form and structure of class specifications.
method declarations - these define the form and structure of method specifications.
field declarations - these define the form and structure of class and interface field specifications.
variable declarations - these define the form and structure of local variable specifications.
initializers - these define the initialization expressions for variable (including array) initializations.
constructors - these define the constructor statements for classes.
blocks and statements - these define the instruction and scoping constructs of the Java language.
expressions - these define all expressions of the Java language.

1.2 Attributes

To specify the semantic aspects of the grammar, we define a set of attributes that are used during the traversal of the parse tree specified by the grammar. For simplicity sake, we define the attributes using Java field and method use notation (e.g., `non.in.env` defines the inherited environment from the non-terminal non). An attribute is considered *inherited* if it is passed down from the non-terminal (root of the subtree) and it is *synthesize* if is created by the right-hand side expression (child nodes). We assume that all inherited attributes are included as fields of the inherited object `in`, which is specified as a field of the non-terminal of the production. We assume that all synthesize attributes are included as fields of the synthesized object `out` which is specified as a field of the non-terminal of the right-hand side expression.

This section describes the attributes of the grammar. The use of these attributes by the logical units of the language is as depicted in Table 2.

context This defines the code type being executed, whether it is a static or normal method, etc. This attribute is only inherited. The methods of this attribute are:

- `addPackage(`*`name`*`)` which adds the specified package *name* to the current context.
- `addClass(`*`name`*`, `*`mods`*`)` which adds the specified class *name* with its correct modifiers *mods* to the current context.
- `addInterface(`*`name`*`, `*`mods`*`)` which adds the specified interface *name* with its correct modifiers *mods* to the current context.
- `addMethod(`*`name`*`, `*`mods`*`)` which adds the specified method *name* with its correct modifiers *mods* to the current context.
- `addSwitchExpr(`*`type`*`)` stores the type of the current switch expression.
- `switchExpr()` returns the type of the current switch expression.

Table 2. Use of attributes in logical units of the language

	context	env	vars	type	value	mods	ids
	Inher.	Both	Both	Synth.	Synth.	Synth.	Synth.
names & lits				S	S		
pkgs		SI					
types				S			
mods						S	
inter decl	CI	CI				U	
class decl	CI	SI				U	
meth decl	CI	SI				U	SU
field decl	CI	SI	UI	U		U	
var decl							SU
inits	I	I	I				
cnstr	CI	SI				U	
blocks & stmts	UI	SUI	SUI	SU			
exprs	UI	UI	UI	SU	U		

C = creates I = Inherits S = Synthesizes
U = used for static error check

- `isInstanceMethod()` which returns true if the current context is within an instance method.
- `isClassMethod()` which returns true if the current context is within a class method.
- `isConstructor()` which returns true if the current context is within a constructor method.
- `className()` which returns the string representation of the current class.
- `getClass()` which returns the reference of the current class.
- `getSuper()` which returns the reference of the *super* class of the current class.

env This defines the "environment" of the program, basically the definition of all types, class fields and class definitions accessible by the current code. This attribute is inherited by code, but synthesized by the declarations aspects of the code. For a truly correct environment, the compiler must first parse all relevant declarations to build the top-level environment. Then the compiler can use this information in the second pass to evaluate expressions and statements. Without these two passes, all information must be declared prior to its use. To compress the presentation of the grammar in this chapter, we have combined the two passes of the compiler into one presentation and have greatly simplified the operations of the first pass of the compiler. The method `new()` defined below activates the first pass of the compiler and returns its results for the second pass. The methods of the env attribute are:

- `new(`*`CompUnit`*`)` which runs the first pass of the compiler on the code, producing a top-level environment which is used for the second pass of the compiler. In this environment are definitions of all classes, their fields

and methods, imported classes, and compiler defined environment information (e.g., classes defined in other files specified on the same command line to the compiler). This method, in effect, runs the attributed grammar by ignoring all error checks and returning the environment output by CompUnit.
- `typeCheck(`*type*$_1$`,` *type*$_2$`)` which returns true if *type*$_2$ is of the type specified by *type*$_1$.
- `lookupFieldType(`*ref*`,` *id*`)` which returns the type of the field *id* from the reference *ref*.
- `lookupFieldValue(`*ref*`,` *id*`)` which returns the value of the field *id* from the reference *ref* if that field is final and was initialized with a constant expression, otherwise it returns *undef*.
- `isDefined(`*name*`)` which returns true if *name* is defined in the current environment.
- `idCheck(`*PrimaryType*`,` *IdType*`)` which returns true if *IdType* is unambiguous and acceptable for *PrimaryType*.
- `isLabel(`*name*`)` which returns true if the specified *name* is a current statement label.
- `addLabel(`*name*`)` which stores *name* as a named label in the current environment.

vars This defines the set of local variable declarations and their types. This attribute is typically only inherited, the exception being the local variable declaration statement which modifies this attribute synthesizing a new one.

type This attribute is only synthesized to perform the necessary type checking. It is synthesized by variable declarations and expressions. The methods of this attribute are:
- `insert(item)` which adds `item` to the list
- `equals(`*type*`)` which returns true if the argument type is the same as the current attributes type. This is used by the typeCheck method of env.
- `promotableTo(`*type*`)` which returns true if the current attribute type is promotable to the argument type.
- `inc()` which takes the current array type, increments the number of dimensions and returns the new array type. If the current type is not an array type, this method creates a one-dimensional array of the current type. Note that in this method we do not keep track of the actual size of each dimension (that is a run-time check.)
- `inc(`*num*`)` which takes the current array type, increments the number of dimensions by *num* and returns the new array type. If the current type is not an array type, this method creates a *num*-dimensional array of the current type. Note that in this method we do not keep track of the actual size of each dimension (that is a run-time check.)

value This attribute is synthesized from the low-level syntax of the language and is used to return the actual value associated with language literals, specifically identifier names and numeric, boolean, string, and character literals and the null constant. The methods of this attribute are:

- `defined()` returns true if the value is not `undefined`.
- `XX(value)` [for XX one of LT, GT, GE, LE, EQ] returns true if the value compares correctly with the parameter *value* (e.g., the value is less than the parameter for operation LT), and false otherwise.
- `bitXX(value)` returns the numeric result of performing the specified bitwise operation (bitAND, bitOR or bitXOR) on the numeric value and the numeric parameter *value*.
- `bitNOT()` returns the numeric result of performing the bitwise complement operation on the numeric value.
- `XX(value)` [for XX one of AND, OR or XOR] returns the boolean result of performing the specified logical operation (AND, OR or XOR) on the boolean value and the boolean parameter *value*.
- `NOT()` returns the boolean result of performing the logical complement operation boolean value.
- `XX(value)` [for XX one of LS, RSS, RSZ] returns the numeric result of performing the specified shift operation (<<, >, or >>>) on the numeric value and the numeric parameter *value*.

ids This attribute is only synthesized by variable declarations and is a list of declared variable ids.

mods This is the list of modifiers for classes, methods, fields and interfaces. The methods of this attribute are:

- `exclusive(list)` which returns true if the attribute only contains modifiers specified in *list*.
- `containsmod)` which returns true if the attribute contains the modifiers specified by *mod*.
- `insert(mod)` which adds `mod` to the list of modifiers.

The following methods are part of the output attribute of a term. They are part of a specific output attribute, since they utilize results of more than one attribute.

- `assignableTo(type)` returns true if the current expression can be converted to the specified *type* by assignment conversion.
- `isExpression(name)` returns true if the parameter *name* refers to a local variable or a field accessible in the current context.
- `getType(name)` returns the type of the parameter *name* within the current context (or undef if the type is unresolvable).
- `getValue(name)` returns the value of the parameter *name* within the current context if name refers to a final variable who's initializer was a constant expression, otherwise it returns undef.

In addition, the following auxiliary functions are used in this grammar

- `binaryNumericConversion(t1, t2)` which returns the resultant type after applying binary numeric conversion [1] to the two argument types `t1` and `t2`.

- `unaryNumericConversion(type)` which returns the resultant type after applying unary numeric conversion [1] to the argument type.
- `mkArrayType(`*`type`*`)` which returns the type equivalent to an array of the parameter *type*.
- `unmkArrayType(`*`type`*`)` which returns the type equivalent to a single element of the array specified by the parameter *type*.

2 The Grammar

In this section we present the full attributed grammar, for each of the logical units of the language defined above. A brief discussion of the attributes of each logical unit is provided.

2.1 Names and Literals

The following grammar specifies the syntax of names and literals in the Java language. Specific formatting details of these are not presented here, but rather are assumed to be those defined in the Java Language Specification [1]. The name entity in this specification returns a string representation of the name that is used by the higher level production to determine the appropriate type. The resulting name/type is returned in the type attribute. Literals, on the other hand return the appropriate literal type in the type attribute. Integer literals also return a value in the value attribute that can be evaluated in the assignment statement. This permits a direct assignment of a *small* integer to shorts, chars and bytes.

```
<Name> ::=
    <SimpleName>
      Name.out := SimpleName.out
  | <QualifiedName>
      Name.out := QualifiedName.out

<SimpleName>::=
    <Id>
      SimpleName.out.type := Id.out.value

<QualifiedName>::=
    <Name> . <Id>
      QualifiedName.type := Name.out.type+“.”+Id.out.value

<Literal> ::=
    <IntLit>
      Literal.out.type := int
      Literal.out.value := IntLit.out.value
  | <FloatLit>
      Literal.out.type := float
```

```
    Literal.out.value := FloatLit.out.value
  | <BoolLit>
    Literal.out.type := bool
    Literal.out.value := BoolLit.out.value
  | <CharLit>
    Literal.out.type := char
    Literal.out.value := CharLit.out.value
  | <StringLit>
    Literal.out.type := java.lang.String
    Literal.out.value := StringLit.out.value
  | <NullLit>
    Literal.out.type := null
```

2.2 Packages

The following grammar defines the high-level file syntax of Java programs. Specifically this aspect of the grammar is responsible for defining package membership, class imports and the top-level class and interface specifications. It is important to remember that all of the type-checking performed within the method bodies is performed only after all of these top-level definitions are parsed in the first pass. All the attributes at this level are just passed up and down the parse tree with the only changes being made are: the name of the current package is placed into the context (if no package is defined, the current package is the default package) and class definition imports are added to the environment.

Note that in this specification, there are some optional non-terminals on the RHS of the productions. The question arises as to how the attribute grammar handles the synthesized attributes of non-selected optional non-terminals. In this case, we adopt the convention that all synthesized-only attributes of an non-selected optional non-terminal are null, and that all inherited and synthesized attributes take on the value of the inherited attribute.

```
<Goal> ::=
    <CompUnit>
      CompUnit.in.env := env.new(<CompUnit>)
      CompUnit.in.context := new context()

<CompUnit> ::=
    <PackageDecl>? <ImportDeclList>? <TypeDeclList>?
      PackageDecl.in := CompUnit.in
      ImportDeclList.in.context :=
          CompUnit.in.context.addPackage(PackageDecl.out.type)
      ImportDeclList.in.env := PackageDecl.out.env
      TypeDeclList.in.context := ImportDeclList.in.context
      TypeDeclList.in.env := ImportDeclList.out.env
      CompUnit.out.env := TypeDeclList.out.env
```

```
<PackageDecl> ::=
    package <Name> ;
      PackageDecl.out.type := Name.out.type

<ImportDeclList> ::=
    <ImportDecl>
      ImportDecl.in := ImportDeclList.in
      ImportDeclList.out.env := ImportDecl.out.env
  | <ImportDeclList1> <ImportDecl>
      ImportDeclList1.in := ImportDeclList.in
      ImportDecl.in.context := ImportDeclList.in.context
      ImportDecl.in.env := ImportDeclList1.out.env
      ImportDeclList.out.env := ImportDecl.out.env

<TypeDeclList> ::=
    <TypeDecl>
      TypeDecl.in := TypeDeclList.in
      TypeDeclList.out.env := TypeDecl.out.env
  | <TypeDeclList> <TypeDecl>
      TypeDeclList1.in := TypeDeclList.in
      TypeDecl.in.context := TypeDeclList.in.context
      TypeDecl.in.env := TypeDeclList1.out.env
      TypeDeclList.out := TypeDecl.out.env

<ImportDecl> ::=
    <SingleTypeImportDecl>
      SingleTypeImportDecl.in := ImportDecl.in
      ImportDecl.out.env := SingleTypeImportDecl.out.env
  | <TypeImportOnDemandDecl>
      TypeImportOnDemandDecl.in := ImportDecl.in
      ImportDecl.out.env := TypeImportOnDemandDecl.out.env

<SingleTypeImportDecl> ::=
    import <Name> ;
      SingleTypeImportDecl.out.env :=
          SingleTypeImportDecl.in.env.import(Name.out.type)

<TypeImportOnDemandDecl> ::=
    import <Name> . * ;
      SingleTypeImportDecl.out.env :=
          SingleTypeImportDecl.in.env.importOnDemand(Name.out.type)

<TypeDecl> ::=
    ;
      TypeDecl.out.env := TypeDecl.in.env
  | <ClassDecl>
      ClassDecl.in := TypeDecl.in
      TypeDecl.out.env := ClassDecl.out.env
  | <InterfaceDecl>
      InterfaceDecl.in := TypeDecl.in
```

TypeDecl.out.env := InterfaceDecl.out.env

2.3 Types

The following grammar presents the syntax of type definitions in Java. These productions simply pass back up the generated type of the term. If a reference type is expected, a compile-time check is made to ensure that the reference is defined, otherwise an error occurs. The same is true of array types.

<Type> ::=
    <PrimType>
      Type.out.type := PrimType.out.type
  | <RefType>
      Type.out.type := RefType.out.type

<PrimType> ::=
    <NumType>
      PrimType.out.type := NumType.out.type
  | **boolean**
      PrimType.out.type := boolean

<NumType> ::=
    <IntType>
      NumType.out.type := IntType.out.type
  | <FloatType>
      NumType.out.type := FloatType.out.type

<IntType> ::=
    byte
      IntType.out.type := byte
  | **short**
      IntType.out.type := short
  | **int**
      IntType.out.type := int
  | **long**
      IntType.out.type := long
  | **char**
      IntType.out.type := char

<FloatType> ::=
    float
      FloatType.out.type := float
  | **double**
      FloatType.out.type := double

<RefType> ::=
    <ClassInterfaceType>

```
      RefType.out.type := ClassInterfaceType.out.type
  | <ArrayType>
      RefType.out.type := ArrayType.out.type

<ClassInterfaceType>
    <Name>
      ClassInterfaceType.out.type := Name.out.type
      if not(ClassInterfaceType.in.env.isDefined(Name.out.type))
          ERROR ("Undefined Name" + Name.out.type)
          ClassInterfaceType.out.type := null

<ClassType> ::=
    <ClassInterfaceType>
      ClassType.out.type := ClassInterfaceType.out.type

<InterfaceType> ::=
    <ClassInterfaceType>
      InterfaceType.out.type := ClassInterfaceType.out.type

<ArrayType> ::=
    <PrimType> [ ]
      ArrayType.out.type := mkArrayType(PrimType.out.type)
  | <Name> [ ]
      ArrayType.out.type :=
          mkArrayType(ArrayType.out.env.lookupType(Name.out.value))
  | <ArrayType> [ ]
      ArrayType.out.type := mkArrayType(ArrayType.out.type)
**** Type check these **
```

2.4 Modifiers

The following grammar presents the syntax of modifiers, which return the *mods* attribute as a list of defined modifiers. These modifiers are used for classes, fields and methods in a Java file. It was decided to include all modifiers in a single grammatical structure here, and to perform restriction checking at a higher level; such as the illegal modification of an interface declaration with the **volatile** modifier. This structure does check for illegal duplicate modifiers, a condition that is not permitted in any use of modifiers in the Java language.

```
<Modifiers> ::=
    <Modifier>
      Modifiers.out.mods := Modifer.out.mods
  | <Modifiers> <Modifier>
      if Modifiers1.out.mods.contains(Modifer.out.value)
          ERROR ("The modifiers should contain only one instance of"+
            Modifier.out.value)
```

```
        Modifiers.out.mods := Modifiers1.out.mods
      else
        Modifiers.out.mods := Modifiers.out.mods.insert(Modifier1.out.value)
      endif

<Modifier> ::=
    public
      Modifer.out.value := public
  | private
      Modifer.out.value := private
  | protected
      Modifer.out.value := protected
  | static
      Modifer.out.value := static
  | abstract
      Modifer.out.value := abstract
  | final
      Modifer.out.value := final
  | native
      Modifer.out.value := native
  | synchronized
      Modifer.out.value := synchronized
  | transient
      Modifer.out.value := transient
  | volatile
      Modifer.out.value := volatile
```

2.5 Interface Declarations

The following grammar presents the syntax for interface declarations.

```
<InterfaceDecl> ::=
    <Modifiers>? <UnmodInterfaceDecl>
**** Modifiers abstract or public

<UnmodInterfaceDecl> ::=
  interface <Id> <Extends>? <InterfaceBody>

<Extends>::=
    extends <InterfaceType>
  | <Extends> , <InterfaceType>

<InterfaceBody> ::=
    { <InterfaceMemberDeclList>? }

<InterfaceMemberDeclList> ::=
    <InterfaceMemberDecl>
```

| <InterfaceMemberDeclList> <InterfaceMemberDecl>

<InterfaceMemberDecl> ::=
<ClassDecl>
| <InterfaceDecl> | <ConstDecl>
| <AbsMethodDecl>

<ConstDecl> ::=
<FieldDecl>
**** Public, static and/or final. Field declaration in body of interface is all 3

<AbsMethodDecl> ::=
<MethodHdr> **;**

2.6 Class Declarations

The following grammar presents class declarations.

<ClassDecl> ::=
<Modifiers> <UnmodClassDecl>
if not(Modifiers.out.mods.exclusive([public, abstract, final]))
ERROR "Classes may only be public, abstract and/or final")
endif
if not(Modifiers.out.mods.contains(abstract) and
Modifiers.out.mods.contains(final))
ERROR ("Classes can not be both abstract and final")
endif
UnmodClassDecl.in.context :=
ClassDecl.context.addClassMods(Modifiers.out.mods)
UnmodClassDecl.in.env := ClassDecl.in.env

<UnmodClassDecl> ::=
class <Id> <Super>$^{?}$ <Interfaces>$^{?}$ <ClassBody>
let con = UnmodClassDecl.in.context.addClassName(Id.out.value) in
let con1 = con.addSuper(Super.out.type)in
let con2 = con1.addInterfaces(Interfaces.out.type) in
ClassBody.in.context := con2
ClassBody.in.env := UnmodClassDecl.in.env

<Super> ::=
extends <ClassType>
Super.out.type := ClassType.out.type

<Interfaces> ::=
implements <InterfaceTypeList>
Interfaces.out.type := InterfaceTypeList.out.type

```
<InterfaceTypeList> ::=
    <InterfaceType>
      InterfaceTypeList.out.type := InterfaceType.out.type
  | <InterfaceTypeList>1 , <InterfaceType>

      InterfacesTypeList.out.type :=
         InterfaceTypeList.out.type.insert(InterfaceType1.out.type)

<ClassBody> ::=
    { <ClassBodyDeclList>? }        ClassBodyDeclList.in := ClassBody.in
      ClassBody.out := ClassBodyDeclList.out

<ClassBodyDeclList> ::=
    <ClassBodyDecl>
      ClassBodyDecl.in := ClassBodyDeclList.in
      ClassBodyDeclList.out := ClassBodyDecl.out
  | <ClassBodyDeclList1> <ClassBodyDecl>
      ClassBodyDeclList1.in := ClassBodyDeclList.in
      ClassBodyDecl.in := ClassBodyDeclList1.out
      ClassBodyDeclList.out := ClassBodyDecl.out

<ClassBodyDecl> ::=
    <ClassDecl>
      ClassDec.in := ClassBodyDecl.in
      ClassBudyDecl.out := ClassDecl.out
**** Nested classes may be static, abstract, final, public, protected, or private **
  | <InterfaceDecl>
      ClassDec.in := ClassBodyDecl.in
      ClassBudyDecl.out := ClassDecl.out
  | <ClassMemberDecl>
      ClassMemberDecl.in := ClassBodyDecl.in
      ClassBudyDecl.out := ClassMemberDecl.out
  | <StaticInit>
      StaticInit.in := ClassBodyDecl.in
      ClassBudyDecl.out := StaticInit.out
  | <ConstrDecl>
      ConstrDecl.in := ClassBodyDecl.in
      ClassBudyDecl.out := ConstrDecl.out

<ClassMemberDecl> ::=
    <FieldDecl>
      FieldDecl.in := ClassMemberDecl.in
      ClassMemberDecl.out := FieldDecl.out
  | <MethodDecl>        MethodDecl.in := ClassMemberDecl.in
      ClassMemberDecl.out := MethodDecl.out
```

2.7 Method Declarations

The following grammar presents the syntax for class method declarations.

<MethodDecl> ::=
    <MethodHdr> <MethodBody>

<MethodHdr> ::=
    <Modifiers>? <Type> <MethodDef> <Throws>?
  | <Modifiers>? **void** <MethodDef> <Throws>?

<MethodDef> ::=
    <Id> **(** <FormalParmList>? **)**
  | <MethodDef> [ ]

<FormalParmList> ::=
    <FormalParam>
  | <FormalParmList> **,** <FormalParam>

<FormalParam> ::=
    <Modifier> <Type> <VarDeclId>
**** Modifier may be final **

<Throws> ::=
    throws <ClassTypeList>

<ClassTypeList> ::=
    <ClassType>
  | <ClassTypeList> **,** <ClassType>

<MethodBody> ::=
    ;
  | <Block>

2.8 Field and Variable Declarations

The following grammar presents the syntax for class field declarations, and variable declarations.

<FieldDecl> ::=
    <Modifiers>? <Type> <VarDecl> **;**
**** Modifiers one of (public, protected, private) final, static, transient, volatile

<VarDeclList> ::=
    <VarDecl>

```
  | <VarDeclList> , <VarDecl>

<VarDecl> ::=
    <VarDeclId>
  | <VarDeclId> = <VarInit>
      *** Need Declared before used check here for Field inits***

<VarDeclId> ::=
    <Id>
  | <VarDeclId> [ ]
```

2.9 Initializers

The following grammar presents the syntax for variable and array initializers.

```
<StaticInit> ::=
    static <Block>

<VarInits> ::=
    <VarInit>
  | <VarInits> , <VarInit>
      *** Need Declared before used check here for Field inits***

<ArrayInit> ::=
    { <VarInits>? ,? }

<VarInit> ::=
    <Expr>
  | <ArrayInit>

    myline
```

2.10 Constructor Declarations

The following grammar presents the syntax for constructors.

```
<ConstrDecl> ::=
    <Modifiers>? <ConstrDef> <Throws>? <ConstrBody>
**** Modifiers one of public, provate, protected

<ConstrDef> ::=
    <SimpleName> ( <FormalParmList>? )

<ConstrBody> ::=
    { <ExplConstrInv>? <BlockStmtList>? }
```

<ExplConstrInv> ::=
  this (<ArgList>$^{?}$) ;
| **super** (<ArgList>$^{?}$) ;

2.11 Blocks and Statements

The following grammar presents the syntax for statements and blocks in the Java language. The pertinent attributes of blocks and statements are the environment (*env*) and local variable (*vars*) attributes.

<Block> ::=
  { <BlockStmtList>$^{?}$ }
    BlockStmtList.in.context := Block.in.context
    BlockStmtList.in.env := Block.in.env
    BlockStmtList.in.vars := BlockStmtList.in.vars.newBlock()
    Block.out.vars := Block.in.vars
    Block.out.env := Block.in.env

<BlockStmtList> ::=
  <BlockStmt>
    BlockStmt.in := BlockStmtList.in
    BlockStmtList.out := BlockStmt.out
| <BlockStmtList$_1$> <BlockStmt>
    BlockStmtList$_1$.in := BlockStmtList.in
    BlockStmt.in.context := BlockStmtList.in.context
    BlockStmt.in.vars := BlockStmtList$_1$.out.vars
    BlockStmt.in.end := BlockStmtList$_1$.out.env
    BlockStmtList.out := BlockStmt.out

<BlockStmt> ::=
  <LocalVarDeclStmt>
    LocalVarDeclStmt.in := BlockStmt.in
    BlockStmt.out := LocalVarDeclStmt.out
| <Statement>
    Statement.in := BlockStmt.in
    BlockStmt.out := Statement.out
| <UnmodClassDecl>
    UnmodClassDecl.in := BlockStmt.in
    BlockStmt.out := UnmodClassDecl.out

<LocalVarDeclStmt> ::=
  <LocalVarDecl> ;
    LocalVarDecl.in := LocalVarDeclStmt.in
    LocalVarDeclStmt.out := LocalVarDecl.out

<LocalVarDecl> ::=
  <Type> <VarDeclList>

```
      Type.in := LocalVarDecl.in
      VarDeclList.in := LocalVarDecl.in
      LocalVarDecl.out.vars :=
         LocalVarDecl.in.vars.insert(Type.out.type, VarDeclList.out.ids)
      if DeclConflict(Type.out.type, VarDeclList.out.ids, LocalVarDecl.in.vars)
         ERROR ("Illegal Local Variable Declaration")
      endif
      LocalVarDecl.out.env := LocalVarDecl.in.env

<Statement> ::=
    <StmtNoTrailing>
      StmtNoTrailing.in := Statement.in
  | <LabeledStmt>
      LabeledStmt.in := Statement.in
  | <IfStmt>
      IfStmt.in := Statement.in
  | <IfElseStmt>
      IfElseStmt.in := Statement.in
  | <WhileStmt>
      WhileStmt.in := Statement.in
  | <ForStmt>
      ForStmt.in := Statement.in

<StmtNoShortIf> ::=
    <StmtNoTrailing>
      StmtNoTrailing.in := StmtNoShortIf.in
  | <LabeledStmtNoShortIf>
      LabeledStmtNoShortIf.in := StmtNoShortIf.in
  | <IfElseStmtNoShortIf>
      IfElseStmtNoShortIf.in := StmtNoShortIf.in
  | <WhileStmtNoShortIf>
      WhileStmtNoShortIf.in := StmtNoShortIf.in
  | <ForStmtNoShortIf>
      ForStmtNoShortIf.in := StmtNoShortIf.in

<StmtNoTrailing>
    <Block>
      Block.in := StmtNoTrailing.in
  | <EmptyStmt>
      EmptyStmt.in := StmtNoTrailing.in
  | <ExprStmt>
      ExprStmt.in := StmtNoTrailing.in
  | <SwitchStmt>
      SwitchStmt.in := StmtNoTrailing.in
  | <DoStmt>
      DoStmt.in := StmtNoTrailing.in
  | <BreakStmt>
      BreakStmt.in := StmtNoTrailing.in
  | <ContStmt>
      ContStmt.in := StmtNoTrailing.in
```

```
  | <RetStmt>
      RetStmt.in := StmtNoTrailing.in
  | <SynchStmt>
      SynchStmt.in := StmtNoTrailing.in
  | <ThrowStmt>
      ThrowStmt.in := StmtNoTrailing.in
  | <TryStmt>
      TryStmt.in := StmtNoTrailing.in

<EmptyStmt> ::=
    ;

<LabeledStmt> ::=
    <Id> : <Statement>
      Statement.in.context := LabeledStmt.in.context
      Statement.in.vars := LabeledStmt.in.vars
      if not(LabeledStmt.in.env.isLabel(Id.out.value))
          ERROR("Label "+Id.out.value+" already in use.")
          Statement.in.env := LabeledStmt.in.env
      else
          Statement.in.env := LabeledStmt.in.env.addLabel(Id.out.value)
      endif

<LabeledStmtNoShortIf> ::=
    <Id> : <StmtNoShortIf>
      StmtNoShortIf.in.context := LabeledStmtNoShortIf.in.context
      StmtNoShortIf.in.vars := LabeledStmtNoShortIf.in.vars
      if not(LabeledStmtNoShortIf.in.env.isLabel(Id.out.value))
          ERROR("Label "+Id.out.value+" already in use.")
          StmtNoShortIf.in.env := LabeledStmtNoShortIf.in.env
      else
          StmtNoShortIf.in.env := LabeledStmtNoShortIf.in.env.addLabel(Id.out.value)
      endif

<ExprStmt> ::=
    <Assign>
      Assign.in := ExprStmt.in
  | <PreIncExpr>
      PreIncExpr.in := ExprStmt.in
  | <PreDecExpr>
      PreDecExpr.in := ExprStmt.in
  | <PostIncExpr>
      PostIncExpr.in := ExprStmt.in
  | <PostDecExpr>
      PostDecExpr.in := ExprStmt.in
  | <MethodInv>
      MethodInv.in := ExprStmt.in
  | <ClassInstCreationExpr>
      ClassInstCreationExpr.in := ExprStmt.in
```

```
<IfStmt> ::=
    if ( <Expr> ) <Statement>
      Expr.in := IfStmt.in
      Statement.in := IfStmt.in
      if not (Expr.out.type.equals(boolean))
          ERROR("Condition of if statement must be boolean")
      endif

<IfElseStmt> ::=
    if ( <Expr> ) <StmtNoShortIf> else <Statement>
      Expr.in := IfElseStmt.in
      Statement.in := IfElseStmt.in
      StmtNoShortIf.in := IfElseStmt.in
      if not (Expr.out.type.equals(boolean))
          ERROR("Condition of if statement must be boolean")
      endif

<IfElseStmtNoShortIf> ::=
    if ( <Expr> ) <StmtNoShortIf> else <StmtNoShortIf>
      Expr.in := IfElseStmtNoShortIf.in
      Statement.in := IfElseStmtNoShortIf.in
      StmtNoShortIf.in := IfElseStmtNoShortIf.in
      if not (Expr.out.type.equals(boolean))
          ERROR("Condition of if statement must be boolean")
      endif

<SwitchStmt> ::=
    switch ( <Expr> ) <SwitchBlock>
      Expr.in := SwitchStmt.in
      SwitchBlock.in.env := SwitchStmt.in.env
      if not(Expr.out.type.equals(integral))
          ERROR("Switch statement expression must be integral")
          SwitchBlock.in.context := SwitchStmt.in.context.addSwitchExpt(int)
      else
          SwitchBlock.in := SwitchStmt.in.context.addSwitchExpfr(Expr.out.type)
      endif

<SwitchBlock> ::=
    { <SwitchBlockStmtList>? <SwitchLabelList>? }
      SwitchBlockStmtList.in := SwitchBlock.in
      SwitchLabelList.in := SwitchBlock.in

<SwitchBlockStmtList> ::=
    <SwitchBlockStmt>
      SwitchBlockStmt.in := SwitchBlockStmtList.in
  | <SwitchBlockStmtList1> <SwitchBlockStmt>
      SwitchBlockStmtList1.in := SwitchBlockStmtList.in
      SwitchBlockStmt.in := SwitchBlockStmtList.in
```

```
<SwitchBlockStmt> ::=
    <SwitchLabelList> <BlockStmtList>
      SwitchLabelList.in := SwitchBlockStmt.in
      BlockStmtList.in := SwitchBlockStmt.in

<SwitchLabelList> ::=
    <SwitchLabel>
      SwitchLabel.in := SwitchLabelList.in
  | <SwitchLabelList1> <SwitchLabel>
      SwitchLabelList1.in := SwitchLabelList.in
      SwitchLabel.in := SwitchLabelList.in

<SwitchLabel> ::=
    case <ConstExpr>
      ConstExpr.in := SwitchLabel.in
      if not(ConstExpr.out.assignableTo(SwitchLabel.in.context.switchExpr()))
          ERROR("Case label must be compatible with switch expression type.")
      endif
  | default :

<WhileStmt> ::=
    while ( <Expr> ) <Statement>
      Expr.in := WhileStmt.in
      Statement.in := WhileStmt.in
      if not(Expr.out.type.equals(boolean))
          ERROR("While statement expression must be boolean")
      endif

<WhileStmtNoShortIf> ::=
    while ( <Expr> ) <StmtNoShortIf>
      Expr.in := WhileStmt.in
      StmtNoShortIf.in := WhileStmt.in
      if not(Expr.out.type.equals(boolean))
          ERROR("While statement expression must be boolean")
      endif

<DoStmt> ::=
    do <Statement> while ( <Expr> )
      Expr.in := DoStmt.in
      Statement.in := DoStmt.in
      if not(Expr.out.type.equals(boolean))
          ERROR("Do statement expression must be boolean")
      endif

<ForStmt> ::=
    for ( <ForInit>? ; <Expr>? ; <ForUpdate>? ) <Statement>
      ForInit.in := ForStmt.in
      Expr.in.context := ForStmt.in.context
      Expr.in.env := ForStmt.in.env
```

```
      Expr.in.vars := ForInit.out.vars
      ForUpdate.in.context := ForStmt.in.context
      ForUpdate.in.env := ForStmt.in.env
      ForUpdate.in.vars := ForInit.out.vars
      Statement.in.context := ForStmt.in.context
      Statement.in.env := ForStmt.in.env
      Statement.in.vars := ForInit.out.vars

<ForStmtNoShortIf> ::=
    for ( <ForInit>? ; <Expr>? ; <ForUpdate>? ) <StmtNoShortIf>
      ForInit.in := ForStmt.in
      Expr.in.context := ForStmt.in.context
      Expr.in.env := ForStmt.in.env
      Expr.in.vars := ForInit.out.vars
      ForUpdate.in.context := ForStmt.in.context
      ForUpdate.in.env := ForStmt.in.env
      ForUpdate.in.vars := ForInit.out.vars
      StmtNoShortIf.in.context := ForStmt.in.context
      StmtNoShortIf.in.env := ForStmt.in.env
      StmtNoShortIf.in.vars := ForInit.out.vars

<ForInit> ::=
    <ExprStmtList>
      ExprStmtList.in := ForInit.in
      ForInit.in.vars := ExprStmtList.out.vars
  | <LocalVarDecl>
      LocalVarDecl.in := ForInit.in
      ForInit.out.vars := LocalVarDecl.out.vars

<ForUpdate> ::=
    <ExprStmtList>
      ExprStmtList.in := ForUpdate.in

<ExprStmtList> ::=
    <ExprStmt>
      ExprStmt.in := ExprStmtList.in
  | <ExprStmtList1> , <ExprStmt>
      ExprStmt.in := ExprStmtList.in
      ExprStmtList1.in := ExprStmtList.in

<BreakStmt> ::=
    break <Id>? ;
      Id.in := BreakStmt.in
      if not(BreakStmt.in.env.isLabel(Id.out.value))
         ERROR("Undefined Label "+Id.out.value+" in Break statement")
      endif

<ContStmt> ::=
    continue <Id>? ;
```

```
      Id.in := ContStmt.in
      if not(ContStmt.in.env.isLabel(Id.out.value))
         ERROR("Undefined Label "+Id.out.value+" in Continue statement")
      endif

<RetStmt> ::=
    return <Expr>? ;
      Expr.in := RetStmt.in
      if not(Expr.out.assignableTo(RetStmt.in.context.returnType())
         ERROR(Expr.out.type+ " not compatible with return type")
      endif

<ThrowStmt> ::=
    throw <Expr> ;
      Expr.in := RetStmt.in
      if not(RetStmt.in.context.throws(Expr.out.type))
         ERROR("Statment does not throw exception: "+Expr.out.type)
      endif

<SynchStmt> ::=
    synchronized ( <Expr> ) <Block>
      Expr.in := SynchStmt.in
      Block.in := SynchStmt.in
      if not(Expr.out.type.equals(ref))
         ERROR("Argument of synchronized statement must be reference type")
      endif

<TryStmt> ::=
    try <Block> <Catches>
      Block.in := TryStmt.in
      Catches.in := TryStmt.in
  | try <Block> <Catches>? <Finally>
      Block.in := TryStmt.in
      Catches.in := TryStmt.in
      Finally.in := TryStmt.in

<Catches> ::=
    <CatchClause>
      CatchClause.in := Catches.in
  | <Catches1> <CatchClause>
      Catches1.in := Catches.in
      CatchClause.in := Catches.in

<CatchClause> ::=
    catch ( <FormalParam> ) <Block>
      FormalParam.in := CatchClause.in
      Block.in := CatchClause.in
      if not(FormalParam.out.type.promotableTo(Throwable))
         ERROR("Catch clause parameter must be of type throwable.")
```

```
    endif

<Finally> ::=
    finally <Block>
      Block.in := Finally.in
```

2.12 Expressions

The following grammar presents the syntax for expressions in the Java language. For expressions, the pertinent output (synthesized) attributes are *types* and *values*, the input (inherited) attributes are *context*, *environment* and *variables*.

The JLS specifies the types of expressions, dependent on the types of the subexpressions and the form of the expression. However, in the case where there is a compile-time error (e.g., a type mismatch error), the JLS does not specify either a default or calculated return type. This enables compiler writers to make their own interpretation of the return type, resulting in incompatible behavior during compilation when compile-time errors are present. In this specification we have chosen return types that either follow the convention of the Sun JDK, or result in a relatively intuitive result. For select expressions, the return type and value are both *undef* an undefined value. For type checking methods, an undefined type is compatible with all types. For these errors, we have made the following decisions:

- For the conditional expression <CondExpr>, experimentation with the Sun JDK indicates that the resulting type is the type of the right most expression <CondExpr$_1$>. We followed that precedent in this specification.
- For the overloaded operators |, **&**, and $\wedge$, which can be used for either boolean operations or for numeric bit-wise operations, we follow the convention that the expected and return types are boolean.
- for shift, addition, subtraction and multiplication operations, the default return type is int.
- for the or, and, and xor operations the default value is boolean (even if the programmer intended on a bitwise operation).

```
<ConstExpr> ::=
    <Expr>
      Expr.in := ConstExpr.in
      ConstExpr.out := Expr.out

<Expr> ::=
    <AssignExpr>
      AssignExpr.in := Expr.in
      Expr.out := AssignExpr.out
```

```
<AssignExpr> ::=
    <Assign>
      Assign.in := AssignExpr.in
      AssignExpr.out := Assign.out
  | <CondExpr>
      CondExpr.in := AssgnExpr.in
      AssignExpr.out := CondExpr.out

<Assign> ::=
    <LHS> <AssignOp> <AssignExpr>
      LHS.in := Assign.in
      AssignExpr.in := Assign.in
      if (AssignOp.out.value == EQ)
         if not(AssignExpr.out.assignableTo(LHS.out.type))
            ERROR("Assignment conversion error, cannot convert" +
               AssignExpr.out.type + "to" + LHS.out.type)
         endif
      else if (AssignOp.out.value == NumEQ)
         if not(LHS.out.type.equals(numeric) and
               AssignExpr.out.type.equals(numeric))
            ERROR("Operands of "+AssignOp+" must be numeric")
         endif
      else // AssignOp.out.value == BitEQ
         if not(LHS.out.type.equals(numeric) and
               AssignExpr.out.type.equals(numeric)) or
               not(LHS.out.type.equals(boolean) and
               AssignExpr.out.type.equals(boolean))
            ERROR ("Operands of "+AssignOp+" must be
               both either boolean or numeric")
         endif
      endif
      Assign.out.type := LHS.out.type
      Assign.out.value := undef

<LHS> ::=
    <Name>
      Name.in := LHS.in
      LHS.out := Name.out
  | <FieldAccess>
      Name.in := FieldAccess.in
      FieldAccess.out := Name.out
  | <ArrayAccess>
      Name.in := ArrayAccess.in
      ArrayAccess.out := Name.out

<AssignOp> ::=
    =
      AssignOp.out.value := EQ
  | * =
      AssignOp.out.value := NumEQ
```

```
  | /=
      AssignOp.out.value := NumEQ
  | %=
      AssignOp.out.value := NumEQ
  | +=
      AssignOp.out.value := NumEQ
  | −=
      AssignOp.out.value := NumEQ
  | <<=
      AssignOp.out.value := NumEQ
  | >>=
      AssignOp.out.value := NumEQ
  | >>>=
      AssignOp.out.value := NumEQ
  | &=
      AssignOp.out.value := BitEQ
  | ^=
      AssignOp.out.value := BitEQ
  | |=
      AssignOp.out.value := BitEQ

<CondExpr>::=
    <CondOrExpr>
      CondOrExpr.in := CondExpr.in
      CondExpr.out := CondOrExpr.out
  | <CondOrExpr> ? <Expr> : <CondExpr1>
      CondOrExpr.in := CondExpr.in
      Expr.in := CondExpr.in
      CondExpr1.in := CondExpr.in
      // Check type of conditional expression and evaluate
      if not(CondOrExpr.out.type.equals(boolean))
         ERROR ("Expression on LHS of ? must be boolean")
      else if (CondOrExpr.out.value == undef)
            CondOrExpr.out.value := undef
         else
         if (CondOrExpr.out.value == true)
            CondExpr.out.value := Expr.out.value
         else
            CondExpr.out.value := CondExpr1.out.value
         endif
      endif
      // Handle case if both right-hand subexpressions are boolean
      if (Expr.out.type.equals(boolean) and
            CondExpr1.out.type.equals(boolean))
         CondExpr.out.type := boolean
      // Handle case if both right-hand subexpressions are numeric
      else if (Expr.out.type.equals(numeric) and
            CondExpr1.out.type.equals(numeric))
         if (Expr.out.type.equals(CondExpr1.out.type))
            CondExpr.out.type := Expr.out.type
```

```
        else if ((Expr.out.type.equals(byte) and
                CondExpr1.out.type.equals(short)) or
                (Expr.out.type.equals(short) and
                CondExpr1.out.type.equals(byte))
            CondExpr.out.type := short
        else if (Expr.out.type.inList([short;char;byte]) and
                CondExpr1.out.assignableTo(Expr.out.type))
            ConExpr.out.type := Expr.out.type
        else if (CondExpr1.out.type.inList([short;char;byte]) and
                Expr.out.assignableTo(CondExpr1.out.type))
            CondExpr.out.type := CondExpr1.out.type
        else
            CondExpr.out.type :=
                binaryNumericConversion(Expr.out.type, CondExpr1.out.type)
        endif
    // Handle case if both right-hand subexpresions are references
    else if (Expr.out.type.equals(ref) and
            CondExpr1.out.type.equals(ref))
        if (Expr.out.type.promotableTo(CondExpr1.out.type))
            CondExpr.out.type := CondExpr1.out.type
        else if (CondExpr1.out.type.promotableTo(Expr.out.type))
            CondExpr.out.type := Expr.out.type
        else
            ERROR("Can't convert "+Expr.out.type+"to "+CondExpr1.out.type)
            CondExpr.out.type := Expr.out.type
        endif
    else
        ERROR("Can't convert "+Expr.out.type+"to "+CondExpr1.out.type)
        CondExpr.out.type := Expr.out.type
    endif

<CondOrExpr>::=
    <CondAndExpr>
      CondAndExpr.in := CondOrExpr.in
      CondOrExpr.out := CondAndExpr.out
  | <CondOrExpr1> || <CondAndExpr>
      CondOrExpr1.in := CondOrExpr.in
      CondAndExpr.in := CondOrExpr.in
      if not(CondOrExpr1.out.type.equals(boolean) and
            CondAndExpr.out.type.equals(boolean))
        ERROR("Both arguments to || must be boolean")
        CondOrExpr.out.value := undef
      else if not (CondOrExpr1.out.value.defined())
        CondOrExpr.out.value := undef
      else if (CondOrExpr1.out.value == true)
        CondOrExpr.out.value := true
      else if not (CondAndExpr.out.value.defined())
        CondOrExpr.out.value := undef
      else if (CondAndExpr.out.value == true)
        CondOrExpr.out.value := true
```

```
        else
            CondOrExpr.out.value := false
        endif
        CondOrExpr.out.type := boolean

<CondAndExpr>::=
    <IncOrExpr>
        IncOrExpr.in := CondAndExpr.in
        CondAndExpr.out := IncOrExpr.out
  | <CondAndExpr1> && <IncOrExpr>
        CondAndExpr1.in := CondAndExpr.in
        IncOrExpr.in := CondAndExpr.in
        if not(CondAndExpr1.out.type.equals(boolean) and
                IncOrExpr.out.type.equals(boolean))
            ERROR("Both arguments to && must be boolean")
            CondAndExpr.out.value := undef
        else if not (CondAndExpr1.out.value.defined())
            CondAndExpr.out.value := undef
        else if (CondAndExpr1.out.value == false)
            CondAndExpr.out.value := false
        else if not (IncOrExpr.out.value.defined())
            CondAndExpr.out.value := undef
        else if (IncOrExpr.out.value == true)
            CondAndExpr.out.value := true
        else
            CondAndExpr.out.value := false
        endif
        CondAndExpr.out.type := boolean

<IncOrExpr>::=
    <XORExpr>
        XORExpr.in := IncOrExpr.in
        IncOrExpr.out := XORExpr.out
  | <IncOrExpr1> | <XORExpr>
        IncOrExpr1.in := IncOrExpr.in
        XORExpr.in := IncOrExpr.in
        if (IncOrExpr1.out.type.equals(integral) and
                XORExpr.out.type.equals(integral))
            IncOrExpr.out.type :=
                    binaryNumericConversion(IncOrExpr1.out.type,XORExpr.out.type)
            IncOrExpr.out.value := IncOrExpr1.out.value.bitOR(XORExpr.out.value)
        else if not(IncOrExpr1.out.type.equals(boolean) and
                XORExpr.out.type.equals(boolean))
            ERROR("Both arguments to | must be boolean or numeric")
            IncOrExpr.out.value := undef
            IncOrExpr.out.type := boolean
        else
            IncOrExpr.out.value := IncOrExpr1.out.value.OR(XORExpr.out.type)
            IncOrExpr.out.type := boolean
        endif
```

```
<XORExpr>::=
    <AndExpr>
      AndExpr.in := XORExpr.in
      XORExpr.out := AndExpr.out
  | <XORExpr1> ^ <AndExpr>
      XORExpr1.in := XORExpr.in
      AndExpr.in := XORExpr.in
      if (XOROrExpr1.out.type.equals(integral) and
            AndExpr.out.type.equals(integral))
         XORExpr.out.type :=
               binaryNumericConversion(XORExpr1.out.type,AndExpr.out.type)
         XORExpr.out.value := XORExpr1.out.value.bitXOR(AndExpr.out.value)
      else if not(XORExpr1.out.type.equals(boolean) and
            AndExpr.out.type.equals(boolean))
         ERROR("Both arguments to ∧ must be boolean or numeric")
         XORExpr.out.value := undef
         XORExpr.out.type := boolean
      else
         XORExpr.out.value := XORExpr1.out.value.XOR(AndExpr.out.type)
         XORExpr.out.type := boolean
      endif

<AndExpr>::=
    <EqualExpr>
      EqualExpr.in := AndExpr.in
      AndExpr.out := EqualExpr.out
  | <AndExpr1> & <EqualExpr>
      AndExpr1.in := AndExpr.in
      EqualExpr.in := AndExpr.in
      if (AndExpr1.out.type.equals(integral) and
            EqualExpr.out.type.equals(integral))
         AndExpr.out.type :=
               binaryNumericConversion(AndExpr1.out.type,EqualExpr.out.type)
         AndExpr.out.value := AndExpr1.out.value.bitAND(EqualExpr.out.value)
      else if not(AndExpr1.out.type.equals(boolean) and
            EqualExpr.out.type.equals(boolean))
         ERROR("Both arguments to & must be boolean or numeric")
         AndExpr.out.value := undef
         AndExpr.out.type := boolean
      else
         AndExpr.out.value := AndExpr1.out.value.AND(EqualExpr.out.type)
         AndExpr.out.type := boolean
      endif

<EqualExpr> ::=
    <RelatExpr>
      RelatExpr.in := EqualExpr.in
      EqualExpr.out := RelatExpr.out
  | <EqualExpr1> == <RelatExpr>
```

```
      EqualExpr.out.type := boolean
      if not(EqualsExpr1.out.type.compatibleWith(RelatExpr.out.type))
         ERROR("Operands of == must be compatible types")
         EqualExpr.out.value := undef
      else
         EqualExpr.out.value := EqualExpr1.out.value.EQ(RelatExpr.out.value)
      endif
  | <EqualExpr1> != <RelatExpr>
      EqualExpr.out.type := boolean
      if not(EqualsExpr1.out.type.compatibleWith(RelatExpr.out.type))
         ERROR("Operands of != must be compatible types")
         EqualExpr.out.value := undef
      else
         EqualExpr.out.value := not(EqualExpr1.out.value.EQ(RelatExpr.out.value))
      endif

<RelatExpr>::=
    <ShiftExpr>
      ShiftExpr.in := RelatExpr.in
      RelatExpr.out := ShiftExpr.out
  | <RelatExpr1> < <ShiftExpr>
      RelatExpr1.in := RelatExpr.in
      ShiftExpr.in := RelatExpr.in
      if (TypeCheck(numeric, RelatExpr1.out.type) and
            TypeCheck(numeric, ShiftExpr.out.type))
         RelatExpr.out.type := boolean
         RelatExpr.out.value := RelatExpr1.out.value.LT(ShiftExpr.out.value)
      else
         ERROR ("Both arguments to < must be numeric type")
         RelatExpr.out.type := boolean
         RelatExpr.out.value := undef
      endif
  | <RelatExpr1> > <ShiftExpr>
      RelatExpr1.in := RelatExpr.in
      ShiftExpr.in := RelatExpr.in
      if (TypeCheck(numeric, RelatExpr1.out.type) and
            TypeCheck(numeric, ShiftExpr.out.type))
         RelatExpr.out.type := boolean
         RelatExpr.out.value := RelatExpr1.out.value.GT(ShiftExpr.out.value)
      else
         ERROR ("Both arguments to > must be numeric type")
         RelatExpr.out.type := boolean
         RelatExpr.out.value := undef
      endif
  | <RelatExpr1> <= <ShiftExpr>
      RelatExpr1.in := RelatExpr.in
      ShiftExpr.in := RelatExpr.in
      if (TypeCheck(numeric, RelatExpr1.out.type) and
            TypeCheck(numeric, ShiftExpr.out.type))
         RelatExpr.out.type := boolean
```

```
            RelatExpr.out.value := RelatExpr1.out.value.LE(ShiftExpr.out.value)
        else
            ERROR ("Both arguments to <= must be numeric type")
            RelatExpr.out.type := boolean
            RelatExpr.out.value := undef
        endif
    | <RelatExpr1> >= <ShiftExpr>
        RelatExpr1.in := RelatExpr.in
        ShiftExpr.in := RelatExpr.in
        if (TypeCheck(numeric, RelatExpr1.out.type) and
                TypeCheck(numeric, ShiftExpr.out.type))
            RelatExpr.out.type := boolean
            RelatExpr.out.value := RelatExpr1.out.value.GE(ShiftExpr.out.value)
        else
            ERROR ("Both arguments to >= must be numeric type")
            RelatExpr.out.type := boolean
            RelatExpr.out.value := undef
        endif
    | <RelatExpr1> instanceof <RefType>
        RelatExpr1.in := RelatExpr.in
        RefType.in := RelatExpr.in
        if (typeCheck(refOrNull, RelatExpr1.out.type) and
            typeCheck(ref, ShiftExpr.out.type) and
                RefType.out.type.promotableTo(RelatExpr1.out.type)
            RelatExpr.out.type := boolean
            RelatExpr.out.type := undef
        else
            ERROR ("Impossible for "+RelatExpr1.out.type+
                " to be instance of "+RefType.out.type)
            RelatExpr.out.type := boolean
            RelatExpr.out.value := undef
        endif

<ShiftExpr> ::=
    <AddExpr>
        AddExpr.in := ShiftExpr.in
        ShiftExpr.out := AddExpr.out
    | <ShiftExpr1> << <AddExpr>
        ShiftExpr1.in := ShiftExpr.in
        AddExpr.in := ShiftExpr.in
        if (TypeCheck(integral, ShiftExpr1.out.type) and
                TypeCheck(integral, AddExpr.out.type))
            Shift.out.type := promote(ShiftExpr1.out.type, AddExpr.out.type)
        else ERROR ("Both arguments to << must be integral type")
            ShiftExpr.out.type := int
            Shift.out.value := Shift1.out.value.LS(AddExpr.out.value)
        endif
    | <ShiftExpr1> >> <AddExpr>
        ShiftExpr1.in := ShiftExpr.in
        AddExpr.in := ShiftExpr.in
```

```
      if (TypeCheck(integral, ShiftExpr1.out.type) and
            TypeCheck(integral, AddExpr.out.type))
         Shift.out.type := promote(ShiftExpr1.out.type, AddExpr.out.type)
         Shift.out.value := Shift1.out.value.RSS(AddExpr.out.value)
      else ERROR ("Both arguments to >> must be integral type")
         ShiftExpr.out.type := int
         ShiftExpr.out.value := undef
      endif
  | <ShiftExpr1> >>> <AddExpr>
      ShiftExpr1.in := ShiftExpr.in
      AddExpr.in := ShiftExpr.in
      if (TypeCheck(integral, ShiftExpr1.out.type) and
            TypeCheck(integral, AddExpr.out.type))
         Shift.out.type := promote(ShiftExpr1.out.type, AddExpr.out.type)
         Shift.out.value := Shift1.out.value.RSZ(AddExpr.out.value)
         ShiftExpr.out.type := int
         ShiftExt.out.value := undef
      else ERROR ("Both arguments to >>> must be integral type")
         ShiftExpr.out.type := int
         ShiftExt.out.value := undef
      endif

<AddExpr> ::=
    <MultExpr>
      MultExpr.in := AddExpr.in
      AddExpr.out := MultExpr.out
  | <AddExpr1> + <MultExpr>
      AddExpr1.in := AddExpr.in
      MultExpr.in := AddExpr.in
      if (TypeCheck(string, AddExpr1.out.type) or
            TypeCheck(string, MultExpr.out.type))
         AddExpr.out.type := string
         AddExpr.out.value := AddExpr1.out.value.string+( MultExpr.out.value)
      else if (TypeCheck(numeric, AddExpr1.out.type) and
            TypeCheck(numeric, MultExpr.out.type))
         AddExpr.out.type := promote(AddExpr1.out.type, MultExpr.out.type)
         AddExpr.out.value := AddExpr1.out.value + MultExpr.out.value
      else ERROR ("Both arguments to + must be numeric, or one a string")
         AddExpr.out.type := int
         AddExpr.out.value := undef
      endif
  | <AddExpr> - <MultExpr>
      AddExpr1.in := AddExpr.in
      MultExpr.in := AddExpr.in
      if (TypeCheck(numeric, AddExpr1.out.type) and
            TypeCheck(numeric, MultExpr.out.type))
         AddExpr.out.type := promote(AddExpr1.out.type, MultExpr.out.type)
         AddExpr.out.value := AddExpr1.out.value - MultExpr.out.value
      else ERROR ("Both arguments to + must be NumType")
         AddExpr.out.type := int
```

```
        AddExpr.out.value := undef
      endif

<MultExpr> ::=
    <UnaryExpr>
      UnaryExpr.in := MultExpr.in
      MultExpr.out := UnaryExpr.out
  | <MultExpr1> * <UnaryExpr>
      MultExpr1.in := MultExpr.in
      UnaryExpr.in := MultExpr.in
      if (typeCheck(numeric, MultExpr1.out.type) and
            typeCheck(numeric,UnaryExpr.out.type))
        MultExpr.out.type := promote(MultExpr1.out.type, UnaryExpr.out.type)
        AddExpr.out.value := MultExpr1.out.value - AddExpr.out.value
      else ERROR ("Both arguments to * must be numeric")
        MultExpr.out.type := int
        MultExpr.out.value := undef
      endif
  | <MultExpr1> / <UnaryExpr>
      MultExpr1.in := MultExpr.in
      UnaryExpr.in := MultExpr.in
      if (typeCheck(numeric, MultExpr1.out.type) and
            typeCheck(numeric,UnaryExpr.out.type))
        MultExpr.out.type := promote(MultExpr1.out.type, UnaryExpr.out.type)
        MultExpr.out.value := MultExpr1.out.value - AddExpr.out.value
      else ERROR ("Both arguments to / must be numeric")
        MultExpr.out.type := int
        MultExpr.out.value := undef
      endif
  | <MultExpr1> % <UnaryExpr>
      MultExpr1.in := MultExpr.in
      UnaryExpr.in := MultExpr.in
      if (typeCheck(numeric, MultExpr1.out.type) and
            typeCheck(numeric,UnaryExpr.out.type))
        MultExpr.out.type := promote(MultExpr1.out.type, UnaryExpr.out.type)
        MultExpr.out.value := MultExpr1.out.value % AddExpr.out.value
      else ERROR ("Both arguments to % must be numeric")
        MultExpr.out.type := int
        MultExpr.out.value := undef
      endif

<UnaryExprNotPlusMinus> ::=
    <CastExpr>
      CastExpr.in := UnaryExprNotPlusMinus.in
      UnaryExprNotPlusMinus.out := CastExpr.out
  | <PostExpr>
      PostExpr.in := UnaryExprNotPlusMinus.in
      UnaryExprNotPlusMinus.out := PostExpr.out
  | ~ <UnaryExpr>
      UnaryExpr.in := UnaryExprNotPlusMinus.in
```

```
      if not(UnaryExpr.out.type.equals(integral))
         ERROR(" Argument of ~ must be primitive Integral Type")
         UnaryExprNotPlusMinus.out.type := int
         UnaryExprNotPlusMinus.out.value := undef
      else
         UnaryExprNotPlusMinus.out.type := UnaryExpr.out.type
         UnaryExprNotPlusMinus.out.value := UnaryExpr.out.value.bitNOT()
      endif
  | ! <UnaryExpr>
      if not(UnaryExpr.out.type.equals(integral))
         ERROR(" Argument of ! must be boolean")
         UnaryExprNotPlusMinus.out.type := boolean
         UnaryExprNotPlusMinus.out.value := undef
      else
         UnaryExprNotPlusMinus.out.type := UnaryExpr.out.type
         UnaryExprNotPlusMinus.out.value := UnaryExpr.out.value.NOT()
      endif

<CastExpr>::=
******* Needs to check types for validity of cast **********
    ( <PrimType> <Dims>? ) <UnaryExpr>
      PrimType.in := CastExpr.in
      Dims.in := CastExpr.in
      UnaryExpr.in := CastExpr.in
      CastExpr.out.type := array(PrimType.out.type,Dims.out.type)
  | ( <Expr> ) <UnaryExprNotPlusMinus>
      Expr.in := CastExpr.in
      UnaryExprNotPlusMinus.in := CastExpr.in
      CastExpr.out.type := Expr.out.type
  | ( <Name Dims> ) <UnaryExprNotPlusMinus>
      Name.in := CastExpr.in
      Dims.in := CastExpr.in
      UnaryExprNotPlusMinus.in := CastExpr.in
      CastExpr.out.type := array(Name.out.type, Dims.out.type)

<PostExpr> ::=
    <Primary>
      Primary.in := PostExpr.in
      PostExpr.out := Primary.out
  | <Name>
      Name.in := PostExpr.in
      if (Name.out.isExpression())
         PostExpr.out.type := Name.out.getType()
         PostExpr.out.value := Name.out.getValue()
      else
         ERROR("Undefined variable "+ Name.out.value)
         PostExpr.out.type := undef
         PostExpr.out.value := undef
      endif
```

```
  | <PostIncExpr>
      PostIncExpr.in := PostExpr.in
      PostExpr.out := PostIncExpr.out
  | <PostDecExpr>
      PostDecExpr.in := PostExpr.in
      PostExpr.out := PostDecExpr.out

<PostIncExpr> ::=
    <PostExpr> ++
      PostExpr.in := PostIncExpr.in
      if not(PostExpr.out.type.equals(numeric))
          ERROR("Postfix Expr must be a variable of numeric type")
          PostIncExpr.out.type := int
          PostIncExpr.out.value := undef
      else
          PostIncExpr.out.type := PostExpr.out.type
          if (PostExpr.out.value.defined())
              PostIncExpr.out.value := PostExpr.out.value + 1
          else
              PostIncExpr.out.value := undef
          endif
      endif

<PostDecExpr> ::=
    <PostExpr> - -
      PostExpr.in := PostDecExpr.in
      if not(PostExpr.out.type.equals(numeric))
          ERROR("Postfix Expr must be a variable of numeric type")
          PostDecExpr.out.type := int
          PostDecExpr.out.value := undef
      else
          PostDecExpr.out.type := PostExpr.out.type
          if (PostExpr.out.value.defined())
              PostDecExpr.out.value := PostExpr.out.value - 1
          else
              PostDecExpr.out.value := undef
          endif
      endif

<UnaryExpr>::=
    <PreIncExpr>
      PreIncExpr.in := UnaryExpr.in
      UnaryExpr.out := PreIncExpr.out
  | <PreDecExpr>
      PreDecExpr.in := UnaryExpr.in
      UnaryExpr.out := PreDecExpr.out
  | + <UnaryExpr1>
      UnaryExpr1.in := UnaryExpr.in
      if not(UnaryExpr1.out.type.equals(numeric))
          ERROR('Argument of unary + must be numeric')
```

```
        UnaryExpr.out.type := int
        UnaryExpr.out.value := undef
    else
        UnaryExpr.out := UnaryExpr1.out
    endif
| - <UnaryExpr1>
    UnaryExpr1.in := UnaryExpr.in
    if not(UnaryExpr1.out.type.equals(numeric))
        ERROR("Argument of unary - must be numeric')
        UnaryExpr.out.type := int
        UnaryExpr.out.value := undef
    else
        UnaryExpr.out.type := UnaryExpr1.out.type
        if (UnaryExpr1.out.value.defined())
            UnaryExpr.out.value := 0 - UnaryExpr1.out.value
        else
            UnaryExpr.out.value := undef
        endif
    endif
| <UnaryExprNotPlusMinus>
    UnaryExprNotPlusMinus.in := UnaryExpr.in
    UnaryExpr.in := UnaryExprNotPlusMinus.in

<PreIncExpr> ::=
  ++ <UnaryExpr>
    UnaryExpr.in := PreIncExpr.in
    PreIncExpr.out.value := undef
    if not(UnaryExpr.out.type.equals(numeric))
        ERROR("Preincrement expr must be a variable of numeric type")
        PreIncExpr.out.type := int
    else
        PreIncExpr.out.type := UnaryExpr.out.type
        endif
    endif

<PreDecExpr> ::=
  − <UnaryExpr>
    UnaryExpr.in := PreDecExpr.in
    PreDecExpr.out.value := undef
    if not(UnaryExpr.out.type.equals(numeric))
        ERROR("Predecrement expr must be a variable of numeric type")
        PreDecExpr.out.type := int
    else
        PreDecExpr.out.type := UnaryExpr.out.type
        endif
    endif

<Primary> ::=
  <PrimaryNoNewArray>
    PrimaryNoNewArray.in := Primary.in
```

```
      Primary.out := PrimaryNoNewArray.out
  | <ArrayCreationExpr>
      ArrayCreationExpression := Primary.in
      Primary.out := ArrayCreationExpression.out

<PrimaryNoNewArray> ::=
    <Literal>
      Literal.in := PrimaryNoNewArray.in
      PrimaryNoNewArray.out := Literal.out
  | this
      if not(PrimaryNoNewArray.context.isInstanceMethod() or
            PrimaryNoNewArray.context.isConstructor())
        ERROR("this permitted only in an instance method or constructor")
      endif
      PrimaryNoNewArray.out.value := undef
      PrimaryNoNewArray.out.type = Primary.NoNewArray.in.context.getClass()
  | ( <Expr> )
      Expr.in := PrimaryNoNewArray.in
      PrimaryNoNewArray.out := Expr.out
  | <ClassInstCreationExpr>
      ClassInstCreationExpr.in := PrimaryNoNewArray.in
      PrimaryNoNewArray.out := ClassInstCreationExpr.out
  | <FieldAcc>
      FieldAcc.in := PrimaryNoNewArray.in
      PrimaryNoNewArray.out := FieldAcc.out
  | <MethodInv>
      MethodInv.in := PrimaryNoNewArray.in
      PrimaryNoNewArray.out := MethodInv.out
  | <ArrayAccess>
      ArrayAccess.in := PrimaryNoNewArray.in
      PrimaryNoNewArray.out := ArrayAccess.out

<ArrayCreationExpr> ::=
    new <PrimType> <DimExprList> <Dims>?
      PrimType.in := ArrayCreationExpr.in
      DimExprList.in := ArrayCreationExpr.in
      Dims.in := ArrayCreationExpr.in
      ArrayCreationExpr.out.value := undef
      ArrayCreationExpr.out.type :=
        PrimType.out.type.inc(DimExprList.out.value + Dims.out.value)
  | new <ClassInterfaceType> <DimExprList> <Dims>?
      ClassInterfaceType.in := ArrayCreationExpr.in
      DimExprList.in := ArrayCreationExpr.in
      Dims.in := ArrayCreationExpr.in
      ArrayCreationExpr.out.value := undef
      ArrayCreationExpr.out.type :=
        PrimType.out.type.inc(DimExprList.out.value + Dims.out.value)

<ClassInstCreationExpr>::=
```

```
    new <ClassType> ( <ArgList>? ) <ClassBody>?
      ClassType.in := ClassInstCreationExpr.in
      ArgList.in := ClassInstCreationExpr.in
      ClassBody.in := ClassInstCreationExpr.in
      ClassInstCreationExpr.out.type := ClassType.out.type
      ClassInstCreationExpr.out.value := undef
********* Finish this to check argument types for constructor **********
  | new <InterfaceType> () <ClassBody>
      InterfaceType.in := ClassInstCreationExpr.in
      ArgList.in := ClassInstCreationExpr.in
      ClassBody.in := ClassInstCreationExpr.in
      ClassInstCreationExpr.out.type := InterfaceType.out.type
      ClassInstCreationExpr.out.value := undef
********* Finish this to check types **********

<FieldAcc> ::=
    <Primary> . <Id>
      Primary.in := FieldAcc.in
      Id.in := FieldAcc.in
      if not (FieldAcc.in.env.typeCheck(ref,Primary.out.type))
         ERROR(Primary.out.type + "must be a reference type")
         FieldAcc.out.type := null
      else if not(FieldAcc.in.env.idCheck(Primary.out.type,Id.out.type)))
         ERROR(Id.out.value + "must be non-ambiguous and accessible")
         FieldAcc.out.type := null
      else
         FieldAcc.out.type :=
            FieldAcc.in.env.lookupFieldType(Primary.out.type, Id.out.type)
         FieldAcc.out.value :=
            Field.in.env.lookupFieldValue(Primary.out.type, Id.out.type)
      endif
  | super . <Id>
      Id.in := FieldAcc.in
      FieldAcc.out := Id.out
      if FieldAcc.context.className() == "Java.lang.Object"
         Error("Term super not permitted in class Object")
      else if not(PrimaryNoNewArray.context.isInstanceMethod() or
            PrimaryNoNewArray.context.isConstructor())
         ERROR("super permitted only in an instance method or constructor")
      else
         FieldAcc.out.type := FieldAcc.in.env.lookupFieldType
            (FieldAcc.in.context.getSuper(), Id.out.type)
         FieldAcc.out.value := FieldAcc.in.env.lookupFieldValue
            (FieldAcc.in.context.getSuper(), Id.out.type)
      endif

<MethodInv> ::=
    <Name> ( <ArgList>? )
      *** 15.11.1 Type Name ID not interface
```

```
  | <Primary> . <Id> ( <ArgList>? )
      *** Id must be non-ambiguous and accessible
  | super . <Id> ( <ArgList>? )
      if FieldAcc.context.className() == "Java.lang.Object"
         Error("Term super not permitted in class Object")
         FieldAcc.out := FieldAcc.in
      else if not(PrimaryNoNewArray.context.isInstanceMethod() or
         PrimaryNoNewArray.context.isConstructor())
         ERROR("super permitted only in an instance method or constructor")
      else
         FieldAcc.out.type = FieldAcc.out.context.getSuper() + Id.out.type
      endif

<ArrayAccess> ::=
    <Name> [ <Expr> ]
      Name.in := ArrayAccess.in
      Expr.in := ArrayAccess.in
      ArrayAccess.out.value := undef
      if not(Expr.out.type.promotableTo(int))
         ERROR"Array indicies must be integers")
      endif
      if not(typeCheck(array, ArrayAccess.env.lookupType(Name.out.type)))
         ERROR(Name.value+"must be of array type")
         ArrayAccess.out.type := undef
      else
         ArrayAccess.out.type =
            unmkArrayType(ArrayAccess.env.lookupType(Name.out.type))
      endif
  | <PrimaryNoNewArray> [ <Expr> ]
      PrimaryNoNewArray.in := ArrayAccess.in
      Expr.in := ArrayAccess.in
      ArrayAccess.out.value := undef
      if not(Expr.out.type.promotableTo(int))
         ERROR("Array indicies must be integers")
      endif
      if not(typeCheck(array, PrimaryNoNewArray.type)))
         ERROR(Name.value+"must be of array type")
         ArrayAccess.out.type := undef
      else
         ArrayAccess.out.type =
            unmkArrayType(PrimaryNoNewArray.out.type)
      endif

<ArgList> ::=
    <Expr>
      Expr.in := ArgList.in
      ArgList.out := Expr.out
  | <ArgList1> , <Expr>
      ArgList1.in := ArgList.in
```

```
      Expr.in := ArgList1.out
      ArgList.out := Expr.out

<DimExprList> ::=
    <DimExpr>
      DimExpr.in := DimExprList.in
      DimExprList.out := DimExpr.out
  | <DimExprList1> <DimExpr>
      DimExprList1.in := DimExprList.in
      DimExpr.in := DimExprList.in
      DimExprList.out.type := undef
      DimExprList.out.value := DimExprList1.out.value + 1

<DimExpr> ::=
    [ <Expr> ]
      Expr.in := DimExpr.in
      if not (typeCheck(integral, Expr.out.type))
         ERROR ("Dimension declaration must be IntType")
      endif
      DimExpr.out := Expr.out
      DimExpr.out.type := undef
      DimExpr.out.value := 1

<Dims> ::=
    [ ]
      Dims.out := Dims.in
      Dims.out.type := undef
      Dims.out.value := 1
  | <Dims1> [ ]
      Dims1.in := Dims.in
      Dims.out.value := Dims1.out.value + 1
      Dims.out.type := undef
```

References

1. J. Gosling, B. Joy, and G. Steele. *The Java Language Specification*. Addison-Wesley, 1996.

Describing the Semantics of Java and Proving Type Soundness

Sophia Drossopoulou and Susan Eisenbach

Department of Computing
Imperial College of Science, Technology and Medicine

1 Introduction

Java combines the experience from the development of several object oriented languages, such as C++, Smalltalk and CLOS. The philosophy of the language designers was to include only features with already known semantics, and to provide a small and simple language.

Nevertheless, we feel that the introduction of some new features in Java, as well as the specific combination of features, justifies a study of the Java formal semantics. The use of interfaces, reminiscent of [10,6] is a simplification of the signatures extension for C++ [4] and is – to the best of our knowledge – novel. The mechanism for dynamic method binding is that of C++, but we know of no formal definition. Java adopts the Smalltalk [15] approach whereby all object variables are implicitly pointers.

Furthermore, although there are a large number of studies of the semantics of isolated programming language features or of minimal programming languages [1], [31],[34], there have not been many studies of the formal semantics of *actual* programming languages. In addition, the interplay of features which are very well understood in isolation, might introduce unexpected effects.

Experience confirms the importance of formal studies of type systems early on during language development. Eiffel, a language first introduced in 1985, was discovered to have a loophole in its type system in 1990 [9,22]. Given the growing usage of Java, it seems important that if there are loopholes in the type system they be discovered early on.

We argue that the type system of Java is sound, in the sense that unless an exception is raised, the evaluation of any expression will produce a value of a type "compatible" with the type assigned to it by the type system.

We were initially attracted to Java because of its elegant combination of several tried language features. For this work we were guided by the language description in [17]. Any question relating to semantics could be answered unambiguously by [17]. However, we discovered some rules to be more restrictive than necessary, and the reasons for some design decisions were not obvious. We hope that the language authors will publish a language design rationale soon.

Jim Alves-Foss (Ed.): Formal Syntax and Semantics of Java, LNCS 1523, pp. 41–80, 1999.

1.1 The Java Subset Considered so Far

In this paper we consider the following parts of the Java language: primitive types, classes and inheritance, instance variables and instance methods, interfaces, shadowing of instance variables, dynamic method binding, object creation with `new`, the `null` value, arrays, exceptions and exception handling.

We chose this Java subset because we consider the Java way of combining classes, interfaces and dynamic method binding to be both novel and interesting. Furthermore, we chose an imperative subset right from the start, because the extension of type systems to the imperative case has sometimes uncovered new problems, (*e.g.* multi-methods for functional languages [8], and for imperative languages in [5], the Damas and Milner polymorphic type systems for functional languages [11], and for the imperative extension [30]). We considered arrays, because of the known requirement for run time type checking.

In contrast with our previous work [12,13,14] we follow the language description in [17] rather than the more general approach outlined in older versions of the language description.

1.2 Our Approach

We define Java$_s$, a provably *safe* subset of Java containing the features listed previously, a term rewrite system to describe the operational semantics and a type inference system to describe compile-time type checking. We prove that program execution preserves the types up to the subclass/subinterface relationship.

$$\begin{array}{ccccccc} \text{Java} & \supset \text{Java}_s & \xrightarrow[\mathcal{C}]{} & \text{Java}_{se} & \subset \text{Java}_r & \rightsquigarrow_{\mathtt{p}} & \text{Java}_r \\ & \downarrow & & \downarrow & \downarrow & & \downarrow \\ & \text{Type} & = & \text{Type} & = \text{Type} & \geq_{wdn} & \text{Type} \end{array}$$

We aimed to keep the description straightforward, and so we have removed some of the syntactic sugar in Java, *e.g.* we require instance variable access to have the form `this.var` as opposed to `var`, and we require the last statement in a method to be a `return` statement. These restrictions simplify the type inference and term rewriting systems, but do not diminish the applicability to Java itself. It only takes a simple tranformation to turn a Java program from the domain under cosideration to the corresponding Java$_s$ program.

The type system is described in terms of an inference system. In contrast with many type systems for object oriented languages, it does not have a subsumption rule, a crucial property when type checking message expressions, *c.f.* section 5.2. Contrary to Java, Java$_s$ statements have a type – and thus we can type check the return values of method bodies.

The execution of Java programs requires some type information at run-time (*e.g.* method descriptors as in chapter 15.11 in [17]). For this reason, we define Java$_{se}$, an *enriched* version of Java$_s$ containing compile-time type information to be used for method call and field access.

During execution, these terms may be rewritten to terms which are not described by the enriched language, Java$_{se}$. We therefore extend the langauge, obtaining Java$_r$, that describes *run-time* terms. In previous work [12,13,14] we did not distinguish between Java$_{se}$ and Java$_r$; instead, we only conisdered one enriched and extended language. However, as Don Syme pointed out early on, the two different reasons for language modifications should naturally lead to distinct languages. Also, such a distinction allows a clearer description of the concepts. Last not least, this distinction is necessary for the formalization of the notions around binary compatibility [33].

The operational semantics is defined as a ternary rewrite relationship between configurations, programs and configurations. Configurations are tuples of Java$_r$ terms and states. The terms represent the part of the original program remaining to be executed. We describe method calls through textual substitution.

We have been able to avoid additional structures such as program counters and higher order functions. The Java$_s$ simplifications of eliminating block structure and local variables allow the definition of the state as a flat structure, where addresses are mapped to objects and global variables are mapped to primitive values or addresses. Objects carry their classes (similar to the Smalltalk abstract machine [19], thus we do not need store types [1], or location typings [18]). Objects are labelled tuples, where each label contains the class in which it was declared. Array values are tuples too, and they are annotated by their type and their dimension.

There are strong links between our work and that described in the next two chapters of that book [29,32] Don Syme describes in chapter 4 the formalization of a large part of this work using his theorem checker, DECLARE. During this process he uncovered a major flaw in our work, which will be described later on. A close collaboration ensued.

David von Oheimb and Tobias Nipkow have encoded their formalization of an enriched language similar to Java$_{se}$ into the theorem prover Isabelle. Thus the treatment of the original language, Java$_s$, is omitted. Their description of most language constructs is similar to ours, except for exceptions, for which they use a dedicated component of the run-time configuration. More importantly, they used a large-step operational semantics. This turned out to have incisive influence on the necessary proofs, and to allow for spectacular simplifications. Thus, in the large step semantics inconsistent intermediate states need not be considered and most lemmas could be significantly simplified. This difference came as a surprise to all authors. On the other hand, strictly speaking, large step semantics cannot make any promise about non terminating programs not breaking the type system, nor is it yet clear how large step semantics could adequately describe coroutines.

The rest of this chapter is organized as follows: In section 2 we give an example in Java, which we use to illustrate the concepts introduced in the subsequent sections. In section 3 we give the syntax of Java$_s$. In section 4 we define the language Java$_{se}$. In section 5 we define the static types for Java$_s$, and the mapping

from Java_s to Java_{se}. In section 6 we describe the types of Java_{se} terms, whereas in section 7 we describe the types of Java_r terms. In section 8 we describe states, configurations and the operational semantics for Java_r. In section 9 we state properties of the operational semantics and in particular the Subject Reduction Theorem. In section 10 we draw some conclusions.

2 An Example in Java

The following, admittedly contrived, Java program serves to demonstrate some of the Java features that we tackle, and will be used in later sections to illustrate our approach. It can have the following interpretation: Philosophers like truths. When a philosopher thinks about a problem together with another philosopher, then, after some deliberation, they refer the problem to a third philosopher. When a philosopher thinks together with a French philosopher, they produce a book. French philosophers like food; they too may think together with another philosopher, and finally refer the question to another philosopher.

Assuming previous definitions of classes `Book`, `Food` and `Truth`, consider the classes `Phil`, `FrPhil` defined as:

```
class Phil {
     Truth like ;
     Phil think(Phil y){ ...}
     Book think(FrPhil y){ ...}
}
class FrPhil extends Phil {
     Food like ;
     Phil think(Phil y){like=oyster;...}
}
```

Consider the following declarations and expressions:

```
     Phil aPhil ; FrPhil pascal = new FrPhil ;
     ...aPhil.like
     ...aPhil.think(pascal)...aPhil.think(aPhil)
     ...pascal.like
     ...pascal.think(pascal)...pascal.think(aPhil)
```

The above example demonstrates:

- Recursive scopes, *e.g.* the class `FrPhil` is visible inside the class `Phil`, that is before its declaration.
- Shadowing of instance variables by static types, *e.g.* `pascal.like` is an object of class `Food`, whereas `aPhil.like` indicates an object of class `Truth`, even after the assignment `aPhil:=pascal`.
- Method binding according to the dynamic class of the receiver, and the static class of the arguments: The call `aPhil.think(pascal)` will result in calling the method `Phil::think(FrPhil)` (*i.e.* the `think` method declared in class `Phil` and which takes a `FrPhil` argument, and returns a `Book`), even if `aPhil`

contains a pointer to a `FrPhil` object. The call `aPhil.think(aPhil)` will result in calling the method `Phil::think(Phil)` if `aPhil` is an object of class `Phil`, and it will result in calling `FrPhil::think(Phil)`, if `aPhil` is an object of class `FrPhil`. The call `pascal.think(pascal)` is ambiguous, because the methods `Phil::think(FrPhil)` and `FrPhil::think(Phil)` are applicable, and neither is more special than the other.

3 The Language Java$_s$

Java$_s$ is a subset of Java, which includes classes, instance variables, instance methods, inheritance of instance methods and variables, shadowing of instance variables, interfaces, widening, method calls, assignments, object creation and access, the `null`value, instance variable access and the exception `NullPE`, arrays, array creation and the exceptions `ArrStoreE`, `NegSzeE` and `IndOutBndE`. The features we have not yet considered include initializers, constructors, finalizers, class variables and class methods, local variables, class modifiers, final/abstract classes and methods, `super`, strings, numeric promotions and widenings, concurrency, packages and separate compilation.

There are slight differences between the syntax of Java$_s$ and Java which were introduced to simplify the formal description. A Java program contains both type (*i.e.* variable declarations, parameter and result types for methods, interfaces of classes) and evaluation information (*i.e.* statements in method bodies). In Java$_s$ this information is split into two: type information is contained in the environment (usually represented by a Γ), whereas evaluation information is in the program (usually represented by a `p`).

We follow the convention that Java$_s$ keywords appear as `keyword`, identifiers as `identifier`, nonterminals appear in italics as *Nonterminal*, and the metalanguage symbols appear in Roman (*e.g.* ::=, (,*,)). Identifiers with the suffix `Id` (*e.g.* `VarId`) indicate the identifiers of newly declared entities, whereas identifiers with the suffix `Name` (*e.g.* `VarName`) are entities that have been previously declared.

3.1 Java$_s$ Programs

A program, as described in figure 1, consists of a sequence of class bodies. Class bodies consist of a sequence of method bodies. Method bodies consist of the method identifier, the names and types of the arguments, and a statement sequence. We require that there is exactly one `return` statement in each method body, and that it is the last statement. This simplifies the Java$_s$ operational semantics without restricting the expressiveness, since it requires at most a minor transformation to enable any Java method body to satisfy this property.

We only consider conditional statements, assignments, method calls, `try` and `throw` statements. This is because `loop`, `break`, `continue` and `case` statements can be coded in terms of conditionals and recursion.

$$
\begin{array}{lll}
\mathit{Program} & ::= & (\ \mathit{ClassBody}\ )^* \\
\mathit{ClassBody} & ::= & \texttt{ClassId ext ClassName}\ \{(\ \mathit{MethBody}\ )^*\} \\
\mathit{MethBody} & ::= & \texttt{MethId is}\ \ (\lambda\ \texttt{ParId:}\mathit{VarType}.)^*\ \{\mathit{Stmts}\texttt{; return}\ [\mathit{Expr}]\} \\
\mathit{Stmts} & ::= & \epsilon\ \mid \mathit{Stmts}\ \texttt{;}\ \mathit{Stmt} \\
\mathit{Stmt} & ::= & \texttt{if}\ \ \mathit{Expr}\ \ \texttt{then}\ \ \mathit{Stmts}\ \ \texttt{else}\ \ \mathit{Stmts} \\
 & \mid & \ \ \mathit{Var}\ \texttt{:=}\ \mathit{Expr}\ \mid\ \mathit{Expr}\texttt{.MethName(}\mathit{Expr}^*\texttt{)}\ \mid\ \texttt{throw}\ \mathit{Expr} \\
 & \mid & \ \ \texttt{try}\ \mathit{Stmts}\ (\texttt{catch ClassName Id}\ \mathit{Stmts})^*\ \texttt{finally}\ \mathit{Stmts} \\
 & \mid & \ \ \texttt{try}\ \mathit{Stmts}\ (\texttt{catch ClassName Id}\ \mathit{Stmts})^+ \\
\mathit{Expr} & ::= & \mathit{Value}\ \mid\ \mathit{Var}\ \mid\ \mathit{Expr}\texttt{.MethName (}\ \mathit{Expr}^*\texttt{)}\ \mid\ \texttt{new ClassName} \\
 & \mid & \ \ \texttt{new}\mathit{SimpleType}\ (\texttt{[}\ \mathit{Expr}\ \texttt{]})^+(\texttt{[}\ \texttt{]})^* \\
\mathit{Var} & ::= & \texttt{Name}\ \mid\ \mathit{Var}\texttt{.VarName}\ \mid\ \mathit{Var}\texttt{[}\mathit{Expr}\texttt{]}\ \mid\ \ \texttt{this} \\
\mathit{Value} & ::= & \mathit{PrimValue}\ \mid\ \texttt{null} \\
\mathit{PrimValue} & ::= & \mathit{intValue}\ \mid\ \mathit{charValue}\ \mid\ \mathit{byteValue}\ \mid\ \ldots \\
\mathit{VarType} & ::= & \mathit{SimpleType}\ \mid\ \mathit{ArrayType} \\
\mathit{SimpleType} & ::= & \mathit{PrimType}\ \mid\ \texttt{ClassName}\ \mid\ \texttt{InterfaceName} \\
\mathit{ArrayType} & ::= & \mathit{SimpleType}\texttt{[ ]}\ \mid\ \mathit{ArrayType}\texttt{[ ]} \\
\mathit{PrimType} & ::= & \texttt{bool}\ \mid\texttt{char}\ \mid\texttt{int}\ \mid\ \ldots
\end{array}
$$

Fig. 1. Java$_s$ programs

We consider values, method calls, and instance variable access. Java values are primitive (*e.g.* literals such as `true`, `false`, `3`, $'\texttt{c}'$ *etc*), references or arrays. References are `null`, or pointers to objects. The expression `new C` creates a new object of class `C`, whereas the expression `new T`$[\texttt{e}_1]\ldots[\texttt{e}_\texttt{n}][]_1\ldots[]_k$, $\texttt{n} \geq 1, k \geq 0$ creates a `n+k`-dimensional array value. Pointers to objects are implicit. We distinguish variable types (sets of possible run-time values for variables) and method types, as can be seen in figure 3.

Java$_s$ programs contain the class hierarchy. Thus, from a program `p` we can deduce the $\sqsubseteq$ relationship, which is the transitive closure of the immediate superclass relation, and also applies to arrays whose component types are subclasses of each other. This relation is defined in figure 2. We use the notation $\texttt{p} = \texttt{p}', \texttt{C ext C}'\{\ldots\}, \texttt{p}''$ to indicate that `p` contains a declaration of class `C` as a subclass of $\texttt{C}'$. The assertion $\texttt{p} \vdash \texttt{C} \sqsubseteq \texttt{C}'$ indicates that given program `p`, `C` is a subclass of $\texttt{C}'$.

$$
\frac{\texttt{p} = \texttt{p}', \texttt{C ext C}'\{\ldots\}, \texttt{p}''}{\begin{array}{l}\texttt{p} \vdash \texttt{C} \sqsubseteq \texttt{C} \\ \texttt{p} \vdash \texttt{C} \sqsubseteq \texttt{C}' \\ \texttt{p} \vdash \texttt{nil} \sqsubseteq \texttt{C}\end{array}}
\qquad
\frac{\texttt{p} \vdash \texttt{C} \sqsubseteq \texttt{C}' \quad \texttt{p} \vdash \texttt{C}' \sqsubseteq \texttt{C}''}{\texttt{p} \vdash \texttt{C} \sqsubseteq \texttt{C}''}
\qquad
\frac{\texttt{p} \vdash \texttt{C} \sqsubseteq \texttt{C}'}{\texttt{p} \vdash \texttt{C}[] \sqsubseteq \texttt{C}'[]}
$$

Fig. 2. Subclasses deduced from programs `p`

Given a program p we define the functions p(C), which looks up a class body with identifier C in p, and *Classes*(p), which is the set of the identifiers of all classes defined in p.

Definition 1 *For a program* p*, we define* p(C)*, and Classes*(p) *as follows:*

- $\mathtt{p(C)} = \mathtt{cBody}$ *iff* $\mathtt{p} = \mathtt{p}', \mathtt{cBody}, \mathtt{p}''$, *and* $\mathtt{cBody} = \mathtt{C\ ext\ C'\ impl\ ...I_n\{...\}}$,
- $\mathtt{p(C)} = \mathtt{Undef}$, *otherwise.*
- $\mathtt{C} \in \mathit{Classes}(\Gamma)$ *iff* $p \vdash \mathtt{C} \sqsubseteq \mathtt{C}$.

3.2 Environments

The environment, described in figure 3, usually denoted by a Γ, contains both the subclass and interface hierarchies and variable type declarations. It also contains the type definitions of all variables and methods of a class and its interface. *StandardEnv* should include all the predefined classes, and all the classes described in chapters 20-22 of [17], *e.g.* the exception classes Exception, NullPE, ArrStoreE, IndOutBndE, NegSzeE and others – we do not need to distinguish between checked and unchecked exceptions. Declarations may be class declarations, interface declarations or identifier declarations.

Env	::=	*StandardEnv* | *Env* ; *Decl*
StandardEnv	::=	Exception ext Object ...NullPE ext Exception ...; ...
Decl	::=	ClassId ext ClassName impl (InterfName)*
		{(VarId : *VarType*)* (MethId : *MethType*)*}
	|	InterfId ext InterfName*{(MethId : *MethType*)*}
	|	VarId : *VarType*
MethType	::=	*ArgType* → (*VarType* | void)
ArgType	::=	[*VarType* (×*VarType*)*]
VarType	::=	*SimpleType* | *ArrayType*
SimpleType	::=	*PrimType* | ClassName | InterfaceName
ArrayType	::=	*SimpleType*[ ] | *ArrayType*[ ]
PrimType	::=	bool | char | int | ...
Type	::=	*VarType* | void | nil | *MethType* | ClassName-Thrn

Fig. 3. Java$_s$ environments

A class declaration introduces a new class as a subclass of another class (if no explicit superclass is given, then Object will be assumed), a sequence of component declarations, and optionally, interfaces implemented by the class. Component declarations consist of field identifiers and their types, and method identifiers and their signatures. Since method bodies are not declarations, they are found in the program part rather than the environment.

An interface declaration introduces a new interface as a subinterface of several other interfaces and a sequence of components. The only interface components

in Java$_s$ are methods, because interface variables are implicitly static, and have not been considered. Variable declarations introduce variables of a given type.

3.3 The Example in Java$_s$

The Java philosophers classes from section 2 correspond to the Java$_s$ program $\mathtt{p}_s$:

```
p_s = Phil  ext  Object {
          think  is  λy:Phil.{...}
          think  is  λy:FrPhil.{...}
      }
      FrPhil  ext  Phil {
          think  is  λy:Phil
                     .{this.like :=oyster;...}
      }
```

The corresponding Java$_s$ environment Γ_0 is:

$$
\begin{array}{l}
\Gamma_0 = \mathtt{Phil\ ext\ Object}\ \{\ \mathtt{like\ :\ Truth},\\
\qquad\qquad\qquad\quad \mathtt{think\ :\ Phil{\rightarrow}Phil},\\
\qquad\qquad\qquad\quad \mathtt{think\ :\ FrPhil{\rightarrow}Book},\},\\
\qquad \mathtt{FrPhil\ ext\ Phil}\ \{\ \mathtt{like\ :\ Food},\\
\qquad\qquad\qquad\quad \mathtt{think\ :\ Phil{\rightarrow}Phil}\},\\
\qquad \mathtt{aPhil\ :\ Phil}, \mathtt{pascal\ :\ FrPhil}
\end{array}
$$

3.4 Subclasses, Subinterfaces, Widening

The subclass $\sqsubseteq$ and the implements $:_{imp}$ relations deduced from an environment Γ are defined by the inference rules in figure 4.

$$
\frac{\Gamma = \Gamma',\ \mathtt{C\ \ ext\ \ C'\ \ impl\ ...I...\{...\}}, \Gamma''}{\Gamma \vdash \mathtt{C} \sqsubseteq \mathtt{C},\quad \Gamma \vdash \mathtt{C} \sqsubseteq \mathtt{C'},\quad \Gamma \vdash \mathtt{C} :_{imp} \mathtt{I}}
$$

$$
\frac{\Gamma = \Gamma',\ \mathtt{I\ \ ext\ ...,\ I'\ ,...\{...\}}, \Gamma''}{\Gamma \vdash \mathtt{I} \leq \mathtt{I},\qquad \Gamma \vdash \mathtt{I} \leq \mathtt{I'}}
$$

$$
\frac{}{\vdash \mathtt{Object} \sqsubseteq \mathtt{Object}}
\qquad
\frac{\Gamma \vdash \mathtt{C} \sqsubseteq \mathtt{C'} \quad \Gamma \vdash \mathtt{C'} \sqsubseteq \mathtt{C''}}{\Gamma \vdash \mathtt{C} \sqsubseteq \mathtt{C''}}
\qquad
\frac{\Gamma \vdash \mathtt{I} \leq \mathtt{I'} \quad \Gamma \vdash \mathtt{I'} \leq \mathtt{I''}}{\Gamma \vdash \mathtt{I} \leq \mathtt{I''}}
$$

Fig. 4. subclasses and subinterfaces

By the assertion $\Gamma = \Gamma', def, \Gamma''$ we indicate that Γ contains the definition def. Every class introduced in Γ is its own subclass, and the assertion $\Gamma \vdash \mathtt{C} \sqsubseteq \mathtt{C}$

indicates that C is defined in the environment Γ as a class. The direct superclass of a class is indicated in its declaration. Object is a predefined class. The assertion $\Gamma \vdash \texttt{C} :_{imp} \texttt{I}$ indicates that the class C was declared in Γ as providing an implementation for interface I. The subclass relationship is transitive. Every interface is its own subinterface and the assertion $\Gamma \vdash \texttt{I} \leq \texttt{I}$ indicates that I is defined in the environment Γ as an interface. The superinterface of an interface is indicated in its declaration. The subinterface relationship is transitive.

$$\frac{\Gamma \vdash \texttt{C} \sqsubseteq \texttt{C}}{\Gamma \vdash \texttt{C} \diamond_{VarType}} \qquad \frac{\Gamma \vdash \texttt{I} \leq \texttt{I}}{\Gamma \vdash \texttt{I} \diamond_{VarType}}$$

$$\frac{\Gamma \vdash \texttt{T} \diamond_{VarType}}{\Gamma \vdash \texttt{T[]} \diamond_{VarType}} \qquad \begin{array}{l} \vdash \texttt{int} \diamond_{VarType} \\ \vdash \texttt{char} \diamond_{VarType} \\ \vdash \texttt{bool} \diamond_{VarType} \end{array}$$

$$\frac{\begin{array}{l} \Gamma \vdash \texttt{T} \diamond_{VarType} \quad or \quad \texttt{T} = \texttt{void} \\ \Gamma \vdash \texttt{T}_\texttt{i} \diamond_{VarType} \qquad \texttt{i} \in \{\texttt{1...n}\}, \texttt{n} \geq 0 \end{array}}{\begin{array}{l} \Gamma \vdash \texttt{T}_1 \times ... \times \texttt{T}_\texttt{n} \diamond_{ArgType} \\ \Gamma \vdash \texttt{T}_1 \times ... \times \texttt{T}_\texttt{n} \rightarrow \texttt{T} \diamond_{MethType} \end{array}}$$

Fig. 5. Variable and method types

Variable types, *i.e.* primitive types, interfaces, classes and arrays, are defined in figure 5 and are required in type declarations. Method types, *i.e.* n argument types, with n≥0, and a result type, are defined in figure 5 and are required in method declarations. The assertion $\Gamma \vdash \texttt{T} \diamond_{VarType}$ means that T is a variable type, $\Gamma \vdash \texttt{AT} \diamond_{ArgType}$ means that AT is a method argument type, and $\Gamma \vdash \texttt{MT} \diamond_{MethType}$ means that MT is a method type. Note that we do not keep track of potentially throwable exceptions in the method type. However, in future work method types should be extended to do so, and a stronger subject reduction theorem should be proven, stating that a checked exception can only be thrown during execution of a method that mentions this exception's class (or superclass) in the method's type.

The widening relationship, described in figure 6, exists between variable types. If a type T can be widened to a type T′ (expressed as $\Gamma \vdash \texttt{T} \leq_{wdn} \texttt{T}'$), then a value of type T can be assigned to a variable of type T′ without any run-time casting or checking taking place. This is defined in chapter 5.1.4 [20]; chapter 5.1.2 in [20] defines widening of primitive types, but here we shall only be concerned with widening of references. Furthermore, for the null value, we introduce the type nil which can be widened to any array, class or interface.

$$\frac{\Gamma \vdash \mathtt{T} \diamond_{VarType}}{\Gamma \vdash \mathtt{T} \leq_{wdn} \mathtt{T}} \qquad \frac{\Gamma \vdash \mathtt{T} \leq_{wdn} \mathtt{Object}}{\Gamma \vdash \mathtt{nil} \leq_{wdn} \mathtt{T}}$$

$$\frac{\Gamma \vdash \mathtt{T} \leq \mathtt{T}}{\Gamma \vdash \mathtt{T} \leq_{wdn} \mathtt{Object}} \qquad \frac{\Gamma \vdash \mathtt{T} \sqsubseteq \mathtt{T}'}{\Gamma \vdash \mathtt{T} \leq_{wdn} \mathtt{T}'}$$

$$\frac{\Gamma \vdash \mathtt{T} \leq \mathtt{T}'}{\Gamma \vdash \mathtt{T} \leq_{wdn} \mathtt{T}'} \qquad \frac{\Gamma \vdash \mathtt{T} \leq_{wdn} \mathtt{T}'}{\Gamma \vdash \mathtt{T}[] \leq_{wdn} \mathtt{T}'[]}$$

$$\frac{\Gamma \vdash \mathtt{T} \sqsubseteq \mathtt{T}' \quad \Gamma \vdash \mathtt{T}' :_{imp} \mathtt{T}'' \quad \Gamma \vdash \mathtt{T}'' \leq \mathtt{T}'''}{\Gamma \vdash \mathtt{T} \leq_{wdn} \mathtt{T}'''} \qquad \frac{\Gamma \vdash \mathtt{T} \diamond_{VarType}}{\Gamma \vdash \mathtt{T}[] \leq_{wdn} \mathtt{Object}}$$

Fig. 6. The widening relationship

3.5 Well-Formed Declarations and Environments

The relations $\sqsubseteq$, $:_{imp}$, $\leq$ and $\leq_{wdn}$ are computable for any environment – as can be straightforwardly shown. In figure 7 we describe the Java requirements for variable, class and interface declarations to be well-formed.

We indicate by $\Gamma \vdash \Gamma' \diamond$, that the declarations in environment Γ' are well-formed, under the declarations of the larger environment Γ. We need to consider a larger environment Γ because Java allows forward declarations (*e.g.* in the philosophers example, class `Phil` uses the class `FrPhil` whose declaration follows that of `Phil`). We shall call Γ well-formed, iff $\Gamma \vdash \Gamma \diamond$, in which case we use the shorthand $\Gamma \vdash \diamond$, *c.f.* the third rule in figure 7. The assertion $\Gamma \vdash \Gamma' \diamond$ is checked in two stages: The first stage establishes the relations $\sqsubseteq$, $:_{imp}$, $\leq$ and $\leq_{wdn}$ for the complete environment Γ and establishes that $\sqsubseteq$ and $\leq$ are acyclic; if this is the case, then the second stage establishes that the declarations in Γ' are well-formed one by one, according to the rules in this section.

Not surprisingly, the empty environment is well-formed, *c.f.* the first rule in figure 7.

We need the notion of definition table lookup, *i.e.* $\Gamma(\mathtt{Id})$, which returns the definition of the identifier `Id` in Γ, if it has one.

Definition 2 *For an environment Γ, with unique definitions for every identifier, define $\Gamma(\mathtt{id})$ as follows:*

- $\Gamma(\mathtt{x}) = \mathtt{T}$ *iff* $\Gamma = \Gamma', \mathtt{x} : \mathtt{T}, \Gamma''$
- $\Gamma(\mathtt{C}) = \mathtt{C\ ext\ C'\ impl\ I_1, ...I_n}\ \{\mathtt{v_1 : T_1, ...v_m : T_m, m_1 : MT_1, ...m_k : MT_k}\}$ *iff* $\Gamma = \Gamma', \mathtt{C\ ext\ C'\ impl\ I_1, ...I_n}\ \{\mathtt{v_1 : T_1, ...v_m : T_m, m_1 : MT_1, ...m_k : MT_k}\}, \Gamma''$
- $\Gamma(\mathtt{I}) = \mathtt{I\ ext\ I_1, ...I_n}\{\mathtt{m_1 : MT_1, ...m_k : MT_k}\}$ *iff* $\Gamma = \Gamma'', \mathtt{I\ ext\ I_1, ...I_n}\{\mathtt{m_1 : MT_1, ...m_k : MT_k}\}, \Gamma''$
- $\Gamma(\mathtt{id}) = \mathtt{Undef}$ *otherwise*

Furthermore, Classes(Γ) and Interfaces(Γ) contains the identifiers of all classes or interfaces declared in Γ, i.e.

- $\mathtt{C} \in \mathit{Classes}(\Gamma)$ *iff* $\Gamma \vdash \mathtt{C} \sqsubseteq \mathtt{C}$.
- $\mathtt{I} \in \mathit{Interfaces}(\Gamma)$ *iff* $\Gamma \vdash \mathtt{I} \leq \mathtt{I}$.

A variable should be declared to have a variable type and it should be declared only once, *c.f.* the second rule in figure 7. The type declaration for `T` may follow textually that of the variable `x`, as for example in:

```
A x; ...class A ....
```

We now consider when class declarations are well-formed. For this we shall need several auxiliary concepts. The following auxiliary definition allows the extraction of the argument types and the result type from a method type and helps us describe restrictions imposed on variable and method definitions for classes or interfaces, given in chapters 8.2 and 9 in [17].

Definition 3 *For a method type* $\mathtt{MT} = \mathtt{T_1} \times ... \times \mathtt{T_n} \to \mathtt{T}$*, we define the* argument types *and the* result type*:*

- $\mathit{Args}(\mathtt{MT}) = \mathtt{T_1} \times ... \times \mathtt{T_n}$
- $\mathit{Res}(\mathtt{MT}) = \mathtt{T}$

Next we introduce some functions to find the class components:

- $\mathit{FDec}(\Gamma, \mathtt{C}, \mathtt{v})$ indicates the nearest superclass of `C` (possibly `C` itself) which contains a declaration of the instance variable `v` and its declared type;
- $\mathit{FDecs}(\Gamma, \mathtt{C}, \mathtt{v})$ indicates all the field declarations for `v`, which were declared in a superclass of `C`, and possibly hidden by `C`, or another superclass.
- $\mathit{MDecs}(\Gamma, \mathtt{C}, \mathtt{m})$ indicates all method declarations (*i.e.* both the class of the declaration and the signature) for method `m` in class `C`, or inherited from one of its superclasses, and not hidden by any of its superclasses;
- $\mathit{MSigs}(\Gamma, \mathtt{C}, \mathtt{m})$ returns all signatures for method `m` in class `C`, or inherited and not hidden by any of its superclasses.

Note that shadowed variables are treated differently from overridden methods. Namely, shadowed variables are part of the set *FDecs*, whereas overridden methods are not part of the set *MDecs*. The reason for the difference is that shadowed variables need to be stored in the objects of subclasses (*e.g.* a `FrPhil` object contains a `like` field inherited from the class `Phil`, even though this field is shadowed in `FrPhil`), whereas overridden methods are never called by objects of the subclasses (*e.g.* for `FrPhil` objects the only `think` method with a `Phil` argument is that from `FrPhil`, whereas that defined in `Phil` is of no interest to `FrPhil` objects).

From now on, we implicitly expect Γ to have unique declarations and the relations $\sqsubseteq$ and $\leq$ to be acyclic up to reflexivity. Thus the functions *FDec*, *FDecs*, *MDecs* and *MSigs* are well-defined, *c.f.* [26].

Definition 4 *For an environment Γ, with a class declaration for* C*, i.e.* $\Gamma = \Gamma', \texttt{C ext C' impl } \texttt{I}_1,...\texttt{I}_n\{\texttt{v}_1 : \texttt{T}_1,...\texttt{v}_k : \texttt{T}_k,\ \texttt{m}_1 : \texttt{MT}_1,...\texttt{m}_l : \texttt{MT}_l\},\ \Gamma''$*, define:*

- $FDec(\Gamma, \texttt{Object}, \texttt{v}) = \texttt{Undef}$ *for any* v
  $FDec(\Gamma, \texttt{C}, \texttt{v}) = (\texttt{C}, \texttt{T}_j)$ *iff* $\texttt{v} = \texttt{v}_j$
  $FDec(\Gamma, \texttt{C}, \texttt{v}) = FDec(\Gamma, \texttt{C}', \texttt{v})$ *iff* $\texttt{v} \neq \texttt{v}_j\ \forall \texttt{j} \in \{\texttt{1...k}\}$
- $FDecs(\Gamma, \texttt{Object}, \texttt{v}) = \emptyset$
  $FDecs(\Gamma, \texttt{C}, \texttt{v}) = \{(C', T') \mid (C', T') = FDec(\Gamma, \texttt{C}, \texttt{v})\} \cup FDecs(\Gamma, \texttt{C}', \texttt{v})$
- $MDecs(\Gamma, \texttt{Object}, \texttt{m}) = \emptyset$
  $MDecs(\Gamma, \texttt{C}, \texttt{m}) = \{\ (\texttt{C}, \texttt{MT}_j) \mid \texttt{m} = \texttt{m}_j\ \} \cup$
  $\{\ (\texttt{C}'', \texttt{MT}'') \mid (\texttt{C}'', \texttt{MT}'') \in MDecs(\Gamma, \texttt{C}', \texttt{m}),$ *and*
  $\forall \texttt{j} \in \{\texttt{1...l}\} : \texttt{m} = \texttt{m}_j \implies Args(\texttt{MT}_j) \neq Args(\texttt{MT}'')\ \}$
- $MSigs(\Gamma, \texttt{C}, \texttt{m}) = \{\ \texttt{MT} \mid \exists \texttt{C}''$ *with* $(\texttt{C}'', \texttt{MT}) \in MDecs(\Gamma, \texttt{C}, \texttt{m})\ \}$

The sets $FDecs(\Gamma, \texttt{Object}, \texttt{v})$ and $MDecs(\Gamma, \texttt{Object}, \texttt{m})$ should contain the entities described in chapter 20.1 of [17]. We defined them as empty sets for simplicity.

For the philosophers example the above functions are:

$FDec(\Gamma_0, \texttt{Phil}, \texttt{like}) = \texttt{(Phil,Truth)}$
$FDec(\Gamma_0, \texttt{FrPhil}, \texttt{like}) = \texttt{(FrPhil,Food)}$
$FDecs(\Gamma_0, \texttt{Phil}, \texttt{like}) = \{\texttt{(Phil,Truth)}\}$
$FDecs(\Gamma_0, \texttt{FrPhil}, \texttt{like}) = \{\texttt{(Phil,Truth)}, \texttt{(FrPhil,Food)}\}$
$MDecs(\Gamma_0, \texttt{Phil}, \texttt{think}) = \{\texttt{(Phil,Phil}\rightarrow\texttt{Phil)}, \texttt{(Phil,FrPhil}\rightarrow\texttt{Book)}\}$
$MDecs(\Gamma_0, \texttt{FrPhil}, \texttt{think}) = \{\texttt{(FrPhil,Phil}\rightarrow\texttt{Phil)}, \texttt{(Phil,FrPhil}\rightarrow\texttt{Book)}\}$
$MSigs(\Gamma_0, \texttt{Phil}, \texttt{think}) = \{\texttt{Phil}\rightarrow\texttt{Phil}, \texttt{FrPhil}\rightarrow\texttt{Book}\}$

Similar to classes, we introduce the following functions to look up the interface components: $MDecs(\Gamma, \texttt{I}, \texttt{m})$ contains all method declarations (*i.e.* the interface of the declaration and the signature) for method m in interface I, or inherited – and not hidden – from any of its superinterfaces; $MSigs(\Gamma, \texttt{I}, \texttt{m})$ returns all signatures for method m in interface I, or inherited – and not hidden – from a superinterface.

Definition 5 *For an environment Γ, containing an interface declaration for* I*, i.e.* $\Gamma = \Gamma', \texttt{I ext } \texttt{I}_1,...\texttt{I}_n\{\texttt{m}_1 : \texttt{MT}_1,...\texttt{m}_k : \texttt{MT}_k\}, \Gamma''$*, we define:*

- $MDecs(\Gamma, \texttt{I}, \texttt{m}) = \{\ (\texttt{I}, \texttt{MT}_j) \mid \texttt{m} = \texttt{m}_j\ \} \cup$
  $\{\ (\texttt{I}', \texttt{MT}') \mid \exists \texttt{j} \in \{\texttt{1...n}\} : \quad (\texttt{I}', \texttt{MT}') \in MDecs(\Gamma, \texttt{I}_j, \texttt{m})$
  and $\forall \texttt{i} \in \{\texttt{1...k}\}\ \texttt{m} = \texttt{m}_i \implies Args(\texttt{MT}') \neq Args(\texttt{MT}_i)\ \}$
- $MSigs(\Gamma, \texttt{I}, \texttt{m}) = \{\ \texttt{MT} \mid \exists \texttt{I}' : \quad (\texttt{I}', \texttt{MT}) \in MDecs(\Gamma, \texttt{I}, \texttt{m})\ \}$

The following lemma says that if a type T inherits a method signature from another type T′ *i.e.* if $(\texttt{T}', \texttt{MT}) \in MDecs(\Gamma, \texttt{T}, \texttt{m})$, then T′ is either a class or an interface exporting that method and no other superclass of T, which is a subclass of T′ exports a method with the same identifier and argument types. Also, if a class C inherits a field declaration for v, then there exists a C′, a superclass of C

$$\frac{}{\Gamma \vdash \epsilon \diamond} \qquad \frac{\Gamma \vdash \Gamma' \diamond \quad \Gamma \vdash \mathtt{T} \diamond_{VarType} \quad \Gamma'(\mathtt{x}) = \mathtt{Undef}}{\Gamma \vdash \Gamma', \mathtt{x} : \mathtt{T} \diamond} \qquad \frac{\Gamma \vdash \Gamma \diamond}{\Gamma \vdash \diamond}$$

$$\frac{\begin{array}{l}\mathtt{n} \geq 0,\ \mathtt{k} \geq 0,\ \mathtt{l} \geq 0 \\ \Gamma \vdash \Gamma' \diamond \\ \Gamma'(\mathtt{C}) = \mathtt{Undef} \\ NOT \quad \Gamma \vdash \mathtt{C}' \sqsubseteq \mathtt{C} \\ \Gamma \vdash \mathtt{C}' \sqsubseteq \mathtt{C}' \\ \Gamma \vdash \mathtt{I_j} \leq \mathtt{I_j} \quad \mathtt{j} \in \{\mathtt{1...n}\} \\ \Gamma \vdash \mathtt{T_j} \diamond_{VarType} \quad \mathtt{j} \in \{\mathtt{1...k}\} \\ \Gamma \vdash \mathtt{MT_j} \diamond_{MethType} \quad \mathtt{j} \in \{\mathtt{1...l}\} \\ \mathtt{v_i} = \mathtt{v_j} \implies \mathtt{i} = \mathtt{j} \quad \mathtt{j}, \mathtt{i} \in \{\mathtt{1...k}\} \\ \mathtt{m_i} = \mathtt{m_j} \implies \mathtt{i} = \mathtt{j} \text{ or } Args(\mathtt{MT_i}) \neq Args(\mathtt{MT_j}) \quad \mathtt{j}, \mathtt{i} \in \{\mathtt{1...l}\} \\ \forall \mathtt{j} \in \{\mathtt{1...l}\} \mathtt{MT} \in MSigs(\Gamma, \mathtt{C}', \mathtt{m_j}), Args(\mathtt{MT}) = Args(\mathtt{MT_j}) \implies \\ \quad Res(\mathtt{MT_j}) = Res(\mathtt{MT}) \\ \forall \mathtt{m}, \forall \mathtt{j} \in \{\mathtt{1...n}\} \quad \mathtt{AT} \to \mathtt{T} \in MSigs(\Gamma, \mathtt{I_j}, \mathtt{m}) \implies \\ \quad \exists \mathtt{T}' \text{ with } \mathtt{AT} \to \mathtt{T}' \in MSigs(\Gamma, \mathtt{C}, \mathtt{m}), \Gamma \vdash \mathtt{T}' \leq_{wdn} \mathtt{T}\end{array}}{\Gamma \vdash \Gamma', \mathtt{C\ ext\ C'\ impl\ I_1, ...I_n\{v_1 : T_1, ...v_k : T_k, m_1 : MT_1, ...m_l : MT_l\}} \diamond}$$

$$\frac{\begin{array}{l}\mathtt{n} \geq 0, \mathtt{l} \geq 0 \\ \Gamma \vdash \Gamma' \diamond \\ \Gamma'(\mathtt{I}) = \mathtt{Undef} \\ NOT \quad \Gamma \vdash \mathtt{I_i} \leq \mathtt{I} \quad \mathtt{j} \in \{\mathtt{1...n}\} \\ \Gamma \vdash \mathtt{I_j} \leq \mathtt{I_j} \quad \mathtt{j} \in \{\mathtt{1...n}\} \\ \Gamma \vdash \mathtt{MT_j} \diamond_{MethType} \quad \mathtt{j} \in \{\mathtt{1...l}\} \\ \mathtt{m_i} = \mathtt{m_j} \implies \mathtt{i} = \mathtt{j} \text{ or } Args(\mathtt{MT_i}) \neq Args(\mathtt{MT_j}) \\ \mathtt{i} \in \{\mathtt{1...n}\}, \mathtt{j} \in \{\mathtt{1...l}\} \quad \mathtt{MT} \in MSigs(\Gamma, \mathtt{I_i}, \mathtt{m_j}),\ Args(\mathtt{MT}) = Args(\mathtt{MT_j}) \\ \qquad \implies Res(\mathtt{MT_j}) = Res(\mathtt{MT}) \\ \forall \mathtt{i}, \mathtt{j} \in \{\mathtt{1...n}\} \quad \mathtt{MT_1} \in MSigs(\Gamma, \mathtt{I_1}, \mathtt{m}), \quad \mathtt{MT_2} \in MSigs(\Gamma, \mathtt{I_2}, \mathtt{m}) : \\ \quad Args(\mathtt{MT_1}) : Args(\mathtt{MT_2}) \implies Res(\mathtt{MT_1}) = Res(\mathtt{MT_2})\end{array}}{\Gamma \vdash \Gamma', \mathtt{I\ ext\ I_1, ...I_n\{\ m_1 : MT_1, ...m_l : MT_l\}} \diamond}$$

Fig. 7. Well-formed environments

which contains the declaration of v. This lemma is needed later in the subject reduction theorem when proving that there exists a redex in any well-typed non-ground term.

Lemma 1 *For any environment* Γ*, types* T, T′ *and identifiers* v *and* m:

- $(\mathtt{T}', \mathtt{MT}) \in MDecs(\Gamma, \mathtt{T}, \mathtt{m}) \implies$
  - $\Gamma \vdash \mathtt{T} \sqsubseteq \mathtt{T}'$ *and* $\Gamma(\mathtt{T}') = \mathtt{T}'$ ext ... impl ...{...m : MT...} *and* $\forall \mathtt{T}'', \mathtt{C} \neq \mathtt{T}'$ *with* $\Gamma \vdash \mathtt{C} \sqsubseteq \mathtt{T}'$, $\Gamma \vdash \mathtt{T} \sqsubseteq \mathtt{C}$: $\Gamma(\mathtt{C}) \neq \mathtt{C}$ ext ... impl ...{...m : $Args(\mathtt{MT}) \to \mathtt{T}''$}
    or

  - $\Gamma \vdash \mathtt{T} \leq \mathtt{T}'$ *and* $\Gamma(\mathtt{T}') = \mathtt{T}'\ \mathtt{ext}\ ...\{...\mathtt{m} : \mathtt{MT}...\}$ *and* $\forall \mathtt{T}'', \mathtt{I} \neq \mathtt{T}'$ *with* $\Gamma \vdash \mathtt{I} \leq \mathtt{T}', \Gamma \vdash \mathtt{T} \leq \mathtt{I} :\quad \Gamma(\mathtt{I}) \neq \mathtt{I}\ \mathtt{ext}\ ...\{...\mathtt{m} : \mathit{Args}(\mathtt{MT}) \rightarrow \mathtt{T}''\}$
- $\mathit{FDec}(\Gamma, \mathtt{C}, \mathtt{v}) = (\mathtt{C}', \mathtt{T}') \implies$
  $\Gamma(\mathtt{C}') = \mathtt{C}'...\{...\mathtt{v} : \mathtt{T}...\}$ *and* $\Gamma \vdash \mathtt{C} \sqsubseteq \mathtt{C}'$ *and* $\forall \mathtt{T}', \mathtt{C}'' \neq \mathtt{C}'$ *with* $\Gamma \vdash \mathtt{C} \sqsubseteq \mathtt{C}'', \Gamma \vdash \mathtt{C}'' \sqsubseteq \mathtt{C}' :\quad \Gamma(\mathtt{C}'') \neq \mathtt{C}''\ \mathtt{ext}\ ...\mathtt{impl}...\{...\mathtt{v} : \mathtt{T}''\}$

The language description [17] imposes the following requirements, when a new class C is declared as

$$\mathtt{C}\ \ \mathtt{ext}\ \ \mathtt{C}'\ \mathtt{impl}\ \ \mathtt{I}_1, ...\mathtt{I}_n\{\mathtt{v}_1 : \mathtt{T}_1, ...\mathtt{v}_k : \mathtt{T}_k, \mathtt{m}_1 : \mathtt{MT}_1, ...\mathtt{m}_l : \mathtt{MT}_l\}$$

- there can be sequences of superinterfaces, instance variable declarations, and instance method declarations;
- the previous declarations are well-formed;
- there is no prior declaration of C
- there are no cyclic subclass dependencies between $\mathtt{C}'$ and C
- the declarations of the class $\mathtt{C}'$, interfaces $\mathtt{I_j}$ and variable types $\mathtt{T_j}$ may precede or *follow* the declaration for C – this is why we require $\Gamma \vdash \mathtt{C}' \sqsubseteq \mathtt{C}'$, rather than $\Gamma' \vdash \mathtt{C}' \sqsubseteq \mathtt{C}'$;
- the $\mathtt{MT_j}$ are method types;
- instance variable identifiers are unique;
- instance methods with the same identifier must have different argument types;
- a method overriding an inherited method must have the same result type as the overridden method;
- "unless a class is abstract, the declarations of methods defined in each direct superinterface must be implemented either by a declaration in this class, or by an existing method declaration inherited from a superclass".

These requirements are formalized in the fourth rule in figure 7. Similar requirements for interfaces are given in [17], and their formalization is also given in the fifth rule in figure 7.

3.6 Properties of Well-Formed Environments

It is straightforward to state and prove the following properties of well-formed environments: Two types that are in the subclass relationship are classes, $\sqsubseteq$ is reflexive, transitive and antisymmetric, and the subclass hierarchy forms a tree. Also, two types that are in the subinterface relationship are interfaces, and $\leq$ is transitive, reflexive and antisymmetric. Unlike $\sqsubseteq$, $\leq$ does not form a tree.

Widening is reflexive, transitive and antisymmetric. If an interface widens to another type, then the second type is a superinterface of the first. If a type widens to a class, then the type is a subclass of that class. If a class widens to an interface I, then the class implements a subinterface of I. If an interface widens to another type, then the interface is identical to the type, or one of its immediate superinterfaces is a subinterface of that type.

Finally, the following lemma states that if a type T widens to another type $\mathtt{T}'$, and $\mathtt{T}'$ has a method m, then there exists in T a unique method m with the

same argument types, and whose return type is the same as that of the method from $\mathtt{T}'$.

Lemma 2 *If* $\Gamma \vdash \diamond$, $\Gamma \vdash \mathtt{T} \leq_{wdn} \mathtt{T}'$, *then* $MSigs(\Gamma, \mathtt{T}, \mathtt{m}) \subseteq MSigs(\Gamma, \mathtt{T}', \mathtt{m})$

From now on we implicitly assume that all environments are well-formed.

4 Java$_{se}$, Enriching Java$_s$

Java$_{se}$ is an enriched version of Java$_s$ which provides compile-time type information necessary at run-time. It is a subset of Java$_R$ from [29], and corresponds to Java$_{light}$ from[32]. The syntax of Java$_{se}$ programs is described in figure 8. The process of enriching Java$_s$ terms is described by the mapping $\mathcal{C}$:

$$\mathcal{C} : \text{Environment} \times \text{Java}_s \longrightarrow \text{Java}_{se}$$

Program	::=	(*ClassBody*)*
ClassBody	::=	`ClassId ext ClassName {` (*MethBody*)* `}`
MethBody	::=	`MethId is` (λ `ParId :` *VarType.*)* {*Stmts* `; return` [*Expr*] }
Stmts	::=	ϵ | *Stmts* `;` *Stmt*
Stmt	::=	`if` *Expr* `then` *Stmts* `else` *Stmts*
	|	*Var* `:=` *Expr* | *Expr*.`MethName(`*Expr**`)` | `throw` *Expr*
	|	`try` *Stmts* (`catch ClassName Id` *Stmts*)* `finally` *Stmts*
	|	`try` *Stmts* (`catch ClassName Id` *Stmts*)$^+$
Type	::=	*VarType* | `void` | `nil` | `ClassName-Thrn`
Expr	::=	*Value* | *Var*
	|	*Expr*.[*ArgType*]`MethName(`*Expr**`)`
	|	`new ClassName` $\ll$(`VarName ClassName Value`)$^*\gg$
	|	`new` *SimpleType* (`[`*Expr*`]`)$^+$(`[]`)$^*[\![$*Value*$]\!]$
Var	::=	`Name` | *Var*[*Expr*] | `this`
	|	*Var*.[`ClassName`]`VarName`
Value	::=	*PrimValue* | `null`
PrimValue	::=	*intValue* | *charValue* | *byteValue* | ...
VarType	::=	*SimpleType* | *ArrayType*
SimpleType	::=	*PrimType* | `ClassName` | `InterfaceName`
ArrayType	::=	*SimpleType*`[ ]` | *ArrayType*`[ ]`
PrimType	::=	`bool` | `char` | `int` | ...

Fig. 8. Java$_{se}$ programs

Java$_{se}$ can be obtained from Java$_s$ by applying enrichments in four cases. Method calls are enriched by the signature of the most special applicable method available at compile-time. Thus, the Java$_s$ syntax *Expr*.`MethName(`*Expr**`)`, is replaced in Java$_{se}$ by the syntax *Expr*.`[`*ArgType*`]MethName(`*Expr**`)`. Instance

variable accesses are enriched by the class containing the field declaration. Thus, *Expr*.`VarName` is replaced in Java$_{se}$ by *Expr*.`[ClassName]VarName`. Object creation is enriched by the names of all its fields, the classes they were declared in and their initial, default values. Therefore, the Java$_s$ syntax `new ClassName` is replaced by the Java$_{se}$ syntax `new ClassName` $\ll$(`VarName ClassName Value`)$^*\gg$. Finally, array creation is enriched by the initial values to be stored in each component of the new array. Therefore, the Java$_s$ syntax `new` *SimpleType* (`[` *Expr* `]`)$^+$(`[ ]`)* is replaced in Java$_{se}$ by `new` *SimpleType* (`[`*Expr*`]`)$^+$(`[]`)$^*[\![$*Value*$]\!]$. Examples of enriching of method call, of instance variable access and of object creation can be seen in section 4.1. The Java$_s$ array creation `new int[3]` would be represented in Java$_{se}$ as `new int[3]`$[\![$`0`$]\!]$.

4.1 The Example in Java$_{se}$

The program $\mathtt{p}_s$ from section 2 would be mapped to the Java$_{se}$ program $\mathtt{p}_{se}$:

$\mathtt{p}_{se} = \mathcal{C}\{\!\{\Gamma_0, \mathtt{p}_s\}\!\} =$

```
     Phil ext Object{
              think is λy:Phil.{...}
              think  is  λ y:FrPhil.{...}
     }
     FrPhil ext Phil {
              think is λy:Phil.
              { this.[FrPhil]like :=oyster; ...}
     }
```

The terms would be represented as:

```
...  pascal := new FrPhil ≪ like Phil nil, like FrPhil nil ≫
...  aPhil.[Phil]like ...
...  aPhil.[Phil]think (aPhil)
...  aPhil.[FrPhil]think (pascal) ...
...  pascal.[FrPhil]like ...
...  pascal.think(pascal) !! ambiguous call
...  pascal.[Phil]think (aPhil) ...
```

5 Java$_s$ Types

The type rules for Java$_s$ are given in figures 9, 10, and 11. They correspond to the type checking phase of a Java compiler and have the form $\Gamma \vdash \mathtt{t} : \mathtt{T}$, which means that term `t` has the type `T` in the environment Γ. The assertion $\Gamma \vdash \mathtt{p} \diamond$ signifies that program `p` is well-formed under the environment Γ (*i.e.* that all expression are type correct, and that all classes conform to their definitions), whereas $\Gamma \vdash \mathtt{p} \diamondsuit$ signifies that `p` is complete, (*i.e.* well-formed, and it provides a class body for each class declared in Γ).

In parallel with type checking, Java$_s$ terms are enriched with type information Thus, each type rule is followed by a an enrichment equation of the form $\mathcal{C}\{\!|\Gamma, t|\!\} = \mathtt{t}'$ meaning that the Java$_s$ term `t` is enriched to the equivalent Java$_{se}$ term $\mathtt{t}'$. The enrichment rules are given together with the type rules because in some cases (*i.e.* for method call and field access) the enrichments use type information.

Figure 9 describes the types for variables, primitive values, `null`, statements, newly created objects and arrays, and field and array access.

According to the first rule, character literals have character type, integer literals have integer type *etc.* According to the second rule, a statement sequence has the same type as its last statement. A return statement has `void` type, or the same type as the expression it returns. An expression of type $\mathtt{T}'$ can be assigned to a variable of a type `T` if $\mathtt{T}'$ can be widened to `T`. A conditional consists of two statement sequences not necessarily of the same type.

For a class `C`, the expression `new C` has type `C`. For a simple type `T`, the expression `new T`$[\mathtt{e}_1]...[\mathtt{e}_n][]_1...[]_k$ is a `n+k`-dimensional array of elements of type `T`. Array and object creation expressions are enriched with initialization information that determine the values for component initialization. Initial values are defined in ch. 4.5.5. of [17], and here in the following definition:

Definition 6 *The* initial value *of a simple type is*:

- `0` *is the initial value of* `int`
- `' '` *is the initial value of* `char`
- `false` *is the initial value of* `bool`
- `null` *is the initial value of classes, interfaces or* `nil`

For an array access `v[e]`, the variable `v` should have an array type `T[]`, and `e` should be of integer type. For a field access `v.f`, the variable `v` should have a class type `T`, (because so far we only consider non-static fields, in Java$_s$ only instances have fields) one of whose superclasses (`C`) should contain a field declaration for `f` of type $\mathtt{T}'$, *i.e.* $FDec(\Gamma, \mathtt{T}, \mathtt{f}) = (\mathtt{C}, \mathtt{T}')$, in which case the field access expression has type $\mathtt{T}'$, and the information from which superclass the field declaration is inherited is stored in the corresponding Java$_{se}$ expression, *i.e.* $\mathcal{C}\{\!|\Gamma, v.f|\!\} = \mathcal{C}\{\!|\Gamma, \mathtt{v}|\!\}.[\mathtt{C}]\mathtt{f}$.

Figure 9 also contains the type rules for method bodies and method calls, as in ch. 15.11, [20]: A method is *applicable* if the actual parameter types can be widened to the corresponding formal parameter types. A signature is *more special* than another signature, if and only if it is defined in a subclass or subinterface and all argument types can be widened to the argument types of the second signature; this defines a partial order. The most special signatures are the minima of the "more special" partial order.

Definition 7 *For an environment* Γ*, identifier* `m`*, variable types* `T`, $\mathtt{T}_1$, ... $\mathtt{T}_n$, *the* most special declarations *are defined as follows:*

- $ApplMeths(\Gamma, \mathtt{m}, \mathtt{T}, \mathtt{T}_1 \times ... \times \mathtt{T}_n) = \{(\mathtt{T}', \mathtt{MT}') \mid (\mathtt{T}', \mathtt{MT}') \in MDecs(\Gamma, \mathtt{T}, \mathtt{m})$ *and* $\mathtt{MT}' = \mathtt{T}'_1 \times ... \times \mathtt{T}'_n \to \mathtt{T}'_{n+1}$ *and* $\Gamma \vdash \mathtt{T}_i \leq_{wdn} \mathtt{T}'_i$ *for* $\mathtt{i} \in \{\mathtt{1}...\mathtt{n}\}\}$

$\Gamma \vdash \mathtt{null} : \mathtt{nil}$, $\Gamma \vdash \mathtt{true} : \mathtt{bool}$, $\Gamma \vdash \mathtt{false} : \mathtt{bool}$,
$\Gamma \vdash \mathtt{i} : \mathtt{int}$, $\Gamma \vdash \mathtt{c} : \mathtt{char}$, $\Gamma \vdash \mathtt{x} : \Gamma(\mathtt{x})$
$\mathcal{C}\{\!|\Gamma, z|\!\} = \mathtt{z}$ if $\mathtt{z}$ is integer, character, identifier, `null`, `true`, or `false`

$$\frac{\Gamma \vdash \mathtt{stmts} : \mathtt{void} \qquad \Gamma \vdash \mathtt{stmts}' : \mathtt{void} \qquad \Gamma \vdash \mathtt{stmt} : \mathtt{T}'}{\Gamma \vdash \mathtt{stmts\ ;\ stmt} : \mathtt{T}'}$$

$\mathcal{C}\{\!|\Gamma, \mathtt{stmts\ ;\ stmt}|\!\} = \mathcal{C}\{\!|\Gamma, \mathtt{stmts}|\!\}\ ;\ \mathcal{C}\{\!|\Gamma, \mathtt{stmt}|\!\}$
$\Gamma \vdash \mathtt{if\ e\ then\ stmts\ else\ stmts}' : \mathtt{void}$
$\mathcal{C}\{\!|\Gamma, \mathtt{if}\ e\ \mathtt{then\ stmts\ else\ stmts}'|\!\} =$
$\mathtt{if}\ \mathcal{C}\{\!|\Gamma, e|\!\}\ \mathtt{then}\ \mathcal{C}\{\!|\Gamma, \mathtt{stmts}|\!\}\ \mathtt{else}\ \mathcal{C}\{\!|\Gamma, \mathtt{stmts}'|\!\}$

$$\frac{\Gamma \vdash \mathtt{v} : \mathtt{T} \quad \Gamma \vdash \mathtt{e} : \mathtt{T}' \quad \Gamma \vdash \mathtt{T}' \leq_{wdn} \mathtt{T}}{\Gamma \vdash \mathtt{v} := \mathtt{e} : \mathtt{void}}$$

$\mathcal{C}\{\!|\Gamma, v := e|\!\} = \mathcal{C}\{\!|\Gamma, v|\!\} := \mathcal{C}\{\!|\Gamma, e|\!\}$

$$\frac{\Gamma \vdash \mathtt{e} : \mathtt{T}}{\Gamma \vdash \mathtt{return\ e} : \mathtt{T}}$$

$\mathcal{C}\{\!|\Gamma, \mathtt{return}\ e|\!\} = \mathtt{return}\ \mathcal{C}\{\!|\Gamma, e|\!\}$

$$\frac{}{\Gamma \vdash \mathtt{return} : \mathtt{void}}$$

$\mathcal{C}\{\!|\Gamma, \mathtt{return}|\!\} = \mathtt{return}$

$$\frac{\Gamma \vdash \mathtt{C} \sqsubseteq \mathtt{C}\forall \mathtt{f}, \mathtt{C}', \mathtt{T}' \text{ with } (\mathtt{C}', \mathtt{T}') \in \mathit{FDecs}(\Gamma, \mathtt{C}, \mathtt{f}) : \exists \mathtt{i} \in \{\mathtt{1...n}\} : \mathtt{f_i} = \mathtt{f}, \mathtt{C_i} = \mathtt{C}', \mathtt{T_i} = \mathtt{T}' \quad \mathtt{v_i} \text{ initial for } \mathtt{T_i} \quad \mathtt{i} \in \{\mathtt{1...n}\}}{\Gamma \vdash \mathtt{new\ C} : \mathtt{C}}$$

$\mathcal{C}\{\!|\Gamma, \mathtt{new}\ C|\!\} = \mathtt{new\ C}\ll \mathtt{f_1\ C_1\ v_1, ...f_n\ C_n\ v_n} \gg$

$$\frac{\Gamma \vdash \mathtt{T} \diamond_{VarType}, \quad NOT \quad \Gamma \vdash \mathtt{T} \leq \mathtt{T} \quad \Gamma \vdash \mathtt{e_i} : \mathtt{int} \quad \mathtt{i} \in \{\mathtt{1...n}\}, \mathtt{n} \geq 1, \mathtt{k} \geq 0 \quad \mathtt{v} \text{ is initial for } \mathtt{T}}{\Gamma \vdash \mathtt{new\ T}[\mathtt{e_1}]...[\mathtt{e_n}][]_1...[]_{\mathtt{k}} : \mathtt{T}[]_1...[]_{\mathtt{n+k}}}$$

$\mathcal{C}\{\!|\Gamma, \mathtt{new}\ T[\mathtt{e_1}]...[\mathtt{e_n}][]_1...[]_k|\!\} =$
$\mathtt{new\ T}[\mathcal{C}\{\!|\Gamma, \mathtt{e_1}|\!\}]...[\mathcal{C}\{\!|\Gamma, \mathtt{e_n}|\!\}][]_1...[]_{\mathtt{k}}[\mathtt{v}]$

$$\frac{\Gamma \vdash \mathtt{v} : \mathtt{T}[] \quad \Gamma \vdash \mathtt{e} : \mathtt{int}}{\Gamma \vdash \mathtt{v}[\mathtt{e}] : \mathtt{T}}$$

$\mathcal{C}\{\!|\Gamma, v[e]|\!\} = \mathcal{C}\{\!|\Gamma, v|\!\}[\mathcal{C}\{\!|\Gamma, e|\!\}]$

$$\frac{\Gamma \vdash \mathtt{v} : \mathtt{T} \quad \mathit{FDec}(\Gamma, \mathtt{T}, \mathtt{f}) = (\mathtt{C}, \mathtt{T}')}{\Gamma \vdash \mathtt{v.f} : \mathtt{T}'}$$

$\mathcal{C}\{\!|\Gamma, v.f|\!\} = \mathcal{C}\{\!|\Gamma, \mathtt{v}|\!\}.[\mathtt{C}]\mathtt{f}$

$$\frac{\Gamma \vdash \mathtt{e_i} : \mathtt{T_i} \quad \mathtt{i} \in \{\mathtt{1...n}\}, \mathtt{n} \geq 1 \quad \mathit{MostSpec}(\Gamma, \mathtt{m}, \mathtt{T_1}, \mathtt{T_2} \times ... \times \mathtt{T_n}) = \{(\mathtt{T}, \mathtt{MT})\}}{\Gamma \vdash \mathtt{e_1.m(e_2...e_n)} : \mathit{Res}(\mathtt{MT})}$$

$\mathcal{C}\{\!|\Gamma, \mathtt{e_1.m(e_2...e_n)}|\!\} =$
$\mathcal{C}\{\!|\Gamma, e_1|\!\}.[\mathit{Args}(\mathtt{MT})]\mathtt{m}(\mathcal{C}\{\!|\Gamma, e_2|\!\}...\mathcal{C}\{\!|\Gamma, e_n|\!\})$

Fig. 9. Types for Java$_s$ expressions and statements

– $(\mathtt{T}, \mathtt{T}_1 \times ... \times \mathtt{T}_n \to \mathtt{T}_{n+1})$ *is* more special than $(\mathtt{T}', \mathtt{T}'_1 \times ... \times \mathtt{T}'_n \to \mathtt{T}'_{n+1})$
iff $\Gamma \vdash \mathtt{T} \leq_{wdn} \mathtt{T}'$ *and* $\forall \mathtt{i} \in \{\mathtt{1}...\mathtt{n}\}\ \Gamma \vdash \mathtt{T_i} \leq_{wdn} \mathtt{T'_i}$
– $MostSpec(\Gamma, \mathtt{m}, \mathtt{T}, \mathtt{T}_1 \times ... \times \mathtt{T}_n) = \{(\mathtt{T}', \mathtt{MT}') \mid$
$(\mathtt{T}', \mathtt{MT}') \in ApplMeths(\Gamma, \mathtt{m}, \mathtt{T}, \mathtt{T}_1 \times ... \times \mathtt{T}_n)$ *and*
if $(\mathtt{T}'', \mathtt{MT}'') \in ApplMeths(\Gamma, \mathtt{m}, \mathtt{T}, \mathtt{T}_1 \times ... \times \mathtt{T}_n)$
and $(\mathtt{T}'', \mathtt{MT}'')$ *is* more special than $(\mathtt{T}', \mathtt{MT}')$
then $\mathtt{T}'' = \mathtt{T}'$ *and* $\mathtt{MT}' = \mathtt{MT}''\}$

The signatures of the more specific applicable methods are contained in the set *MostSpec*. A message expression is type-correct when this set contains exactly one pair. The argument types of the signature of this pair is stored as the *method descriptor*, *c.f.* ch.15.11 in [17], and the result type of the signature is the type of the message expression.

Figure 10 describes the types for program, method or class bodies. The first rule describes the type of a method body with parameters $\mathtt{x}_1$, ..., $\mathtt{x}_n$, consisting of the statements `stmts`. The renaming of variables in the method body, namely $\mathtt{stmts}[\mathtt{z}_1/\mathtt{x}_1, ..., \mathtt{z}_n/\mathtt{x}_n]$, is necessary in order to avoid name clashes and, also, in order for lemma 8 to hold – as pointed out in [25]. It is worth noticing that the rules describing method bodies do not determine `T` – instead, the expected return type of the method, `T`, is taken from the environment Γ when applying the next rule of the figure, which describes class bodies.

The second rule in figure 10 describes the type of a class body consisting of method bodies $\mathtt{mBody}_1$, ... $\mathtt{mBody}_n$. Note that each $\mathtt{mBody}_i$ is type checked in the environment Γ, $\mathtt{this} : \mathtt{C}$, which does not contain the instance variable declarations $\mathtt{v}_1 : \mathtt{T}_1$..., $\mathtt{v}_k$: $\mathtt{T}_k$. Thus, through the type system, we force the use of the expression $\mathtt{this}.\mathtt{v}_j$ as opposed to $\mathtt{v}_j$.

A program $\mathtt{p} = \mathtt{cBody}_1, ...\mathtt{cBody}_n$ is well-formed, *i.e.* $\Gamma \vdash \mathtt{p}\ \diamond$, if it contains no more than one class body for each identifier, and if all class bodies, $\mathtt{cBody}_i$, are well-typed and satisfy their declarations. Furthermore, each class is transformed by $\mathcal{C}$. Finally, as described in the last rule of figure 10, a program is complete, if it is well formed, and it provides a class body for each of the classes declared in the environment Γ. This is indicated by $\Gamma \vdash \mathtt{p}\ \diamondsuit$.[1]

The following two functions will be needed for the operational semantics. In a class body `cBody` the function $MethBody(\mathtt{m}, \mathtt{AT}, \mathtt{cBody})$ finds the method body with identifier `m` and argument types `AT`, if it exists. From the requirements for classes in figure 10, it follows that for a well-formed environment Γ, the function $MethBody(\mathtt{m}, \mathtt{AT}, \mathtt{cBody})$ returns either an empty set or a set with one element. In a program `p` the function $MethBody(\mathtt{m}, \mathtt{AT}, \mathtt{C}, \mathtt{p})$ finds the method body with identifier `m` and argument types `AT`, in the nearest superclass of class `C` – if it exists. It returns a single pair consisting of the class with the appropriate method body, and the method body itself or the empty set if none exists.

Definition 8 *Given a class body* $\mathtt{cBody} = \mathtt{C\ ext\ C'}\ \{\mathtt{mBody}_1, ...\mathtt{mBody}_n\}$*, argument types* `AT`*, and a program* `p`*, we define* method look up *as follows:*

[1] Notice, that in previous work, we did not distinguish between well-formed and complete, and the assertion $\Gamma \vdash p\diamond$ signified both.

$$\frac{\begin{array}{l}\texttt{mBody} = \texttt{m is } \lambda \texttt{x}_1 : \texttt{T}_1 ... \lambda \texttt{x}_n : \texttt{T}_n . \{\texttt{stmts}\} \\ \texttt{x}_i \neq \texttt{this} \quad \texttt{i} \in \{1...n\} \\ \texttt{z}_1, ..., \texttt{z}_n \text{ are new variables in } \Gamma \\ \texttt{stmts}' = \texttt{stmts}[\texttt{z}_1/\texttt{x}_1, ..., \texttt{z}_n/\texttt{x}_n] \\ \Gamma, \texttt{z}_1 : \texttt{T}_1 ... \texttt{z}_n : \texttt{T}_n \vdash \texttt{stmts}' \ : \ \texttt{T}' \\ \Gamma \vdash \texttt{T}' \leq_{wdn} \texttt{T}\end{array}}{\begin{array}{l}\Gamma \vdash \texttt{mBody} \ : \ \texttt{T}_1 \times ... \times \texttt{T}_n \rightarrow \texttt{T} \\ \mathcal{C}\{\!|\Gamma, mBody|\!\} = \ \texttt{m is } \lambda \texttt{x}_1 : \texttt{T}_1 ... \lambda \texttt{x}_n : \texttt{T}_n . \{\mathcal{C}\{\!|\Gamma, stmts|\!\}\}\end{array}}$$

$$\frac{\begin{array}{l}\texttt{n}, \texttt{k}, \texttt{ml} \geq 0, \quad \Gamma \vdash \diamond \\ \Gamma(\texttt{C}) = \texttt{C ext C}' \ \texttt{impl I}_1 ... \texttt{I}_n \{\texttt{v}_1 : \texttt{T}_1 ... \texttt{v}_k : \texttt{T}_k, \texttt{m}_1 : \texttt{MT}_1 ... \texttt{m}_l : \texttt{MT}_l\} \\ \texttt{cBody} = \texttt{C ext C}' \ \{\texttt{mBody}_1, ... \texttt{mBody}_l\} \\ \Gamma(\texttt{this} : \texttt{C}) = \texttt{Undef} \\ \texttt{mBody}_i = \texttt{m}_i \texttt{ is mPrsSts}_i \quad \texttt{i} \in \{1...l\} \\ \Gamma, \texttt{this} : \texttt{C} \vdash \texttt{mBody}_i \ : \ \texttt{MT}_i \quad \texttt{i} \in \{1...l\}\end{array}}{\begin{array}{l}\Gamma \vdash \texttt{cBody} \ : \ \Gamma(\texttt{C}) \\ \mathcal{C}\{\!|\Gamma, cBody|\!\} = \ \texttt{C ext C}' \ \{\mathcal{C}\{\!|\Gamma, \texttt{this} : \texttt{C}, mBody_1|\!\} ... \mathcal{C}\{\!|\Gamma, \texttt{this} : \texttt{C}, mBody_l|\!\}\}\end{array}}$$

$$\frac{\begin{array}{l}\texttt{n} \geq 0, \quad \texttt{p} = \texttt{cBody}_1, ... \texttt{cBody}_n \\ \texttt{cBody}_i = \texttt{C ext } ..., \texttt{cBody}_j = \texttt{C ext } ... \\ \qquad \Longrightarrow \quad \texttt{i} = \texttt{j} \quad \texttt{i}, \texttt{j} \in \{1...n\} \\ \Gamma \vdash \texttt{cBody}_i \diamond \quad \texttt{i} \in \{1...n\}\end{array}}{\begin{array}{l}\Gamma \vdash \texttt{p} \diamond \\ \mathcal{C}\{\!|\Gamma, p|\!\} = \mathcal{C}\{\!|\Gamma, cBody_1|\!\} ... \mathcal{C}\{\!|\Gamma, cBody_n|\!\}\end{array}} \qquad \frac{\begin{array}{l}Classes(\Gamma) = Classes(\texttt{p}) \\ \Gamma \vdash \texttt{p} \diamond\end{array}}{\Gamma \vdash \texttt{p} \ll\gg}$$

Fig. 10. Types for Java$_s$ method bodies, class bodies, and program bodies

- $MethBody(\texttt{m}, \texttt{AT}, \texttt{cBody}) = \{\ \texttt{mBody}_j \mid \texttt{mBody}_j = \texttt{m is } \ \lambda \texttt{x}_1 : \texttt{T}_1 ... \ \lambda \texttt{x}_k : \texttt{T}_k . \{...\}$
  $\quad$ *and* $\ \texttt{AT} = \texttt{T}_1 \times ... \times \texttt{T}_k \ \}$
- $MethBody(\texttt{m}, \texttt{AT}, \texttt{Object}, \texttt{p}) = \emptyset$
- $MethBody(\texttt{m}, \texttt{AT}, \texttt{C}, \texttt{p}) = (\texttt{C}, \texttt{mBody}) \quad$ *iff*
  $\quad MethBody(\texttt{m}, \texttt{AT}, \texttt{cBody}) = \{\texttt{mBody}\}, \quad$ *where* $\texttt{cBody} = \texttt{p}(\texttt{C})$
- $MethBody(\texttt{m}, \texttt{AT}, \texttt{C}, \texttt{p}) = MethBody(\texttt{m}, \texttt{AT}, \texttt{C}', \texttt{p}) \quad$ *iff*
  $\quad MethBody(\texttt{m}, \texttt{AT}, \texttt{cBody}) = \emptyset, \quad$ *where* $\texttt{p}(\texttt{C}) = \texttt{C ext C}' ...$

In figure 11 we define the typing rules for exceptions. A `throw` statement has the type `void` if the expression following the `throw` indicates an exception. We require the expression not to be an address. For addresses the rules for Java$_{se}$ found in figure 12 apply. Similarly the `try ... catch ... finally` statements have the type `void`, provided that the constituent statement lists are well-typed, and that the names of exception classes and new variables appear after each `catch`. The additional Java requirements, that no class $\texttt{E}_i$ should appear more than once, and that no class should appear preceded by a subclass are expressed in [21] but are omitted here, since they do not affect the subject reduction property.

$$\frac{\Gamma \vdash \mathtt{e} : \mathtt{E}, \quad \mathtt{e} \neq \iota_{\mathtt{i}} \qquad \Gamma \vdash \mathtt{E} \sqsubseteq \mathtt{Exception}}{\Gamma \vdash \mathtt{throw\ e} : \mathtt{void}}$$

$$\mathcal{C}\{\Gamma, \mathtt{throw}\ e\} = \mathtt{throw}\ \mathcal{C}\{\Gamma, e\}$$

$$\frac{\begin{array}{l}\mathtt{n} \geq 0, \quad \mathtt{v_i}, \mathtt{z_i} \text{ new in } \Gamma \quad \mathtt{i} \in \{1...\mathtt{n}\} \\ \Gamma \vdash \mathtt{E_i} \sqsubseteq \mathtt{Exception} \quad \mathtt{i} \in \{1...\mathtt{n}\} \\ \Gamma, \mathtt{z_i} : \mathtt{E_i} \vdash \mathtt{stmts_i}[\mathtt{z_i}/\mathtt{v_i}] : \mathtt{void} \quad \mathtt{i} \in \{1...\mathtt{n}\} \\ \Gamma \vdash \mathtt{stmts_{n+1}} : \mathtt{void}\end{array}}{\begin{array}{l}\Gamma \vdash \mathtt{try\ stmts_0\ catch\ E_1\ v_1\ stmts_1\ ...\ catch\ E_n\ v_n\ stmts_n} : \mathtt{void} \\ \Gamma \vdash \mathtt{try\ stmts_0\ catch\ E_1\ v_1\ stmts_1\ ...\ catch\ E_n\ v_n\ stmts_n} \\ \quad \mathtt{finally\ stmts_{n+1}} : \mathtt{void} \\ \mathcal{C}\{\Gamma, \mathtt{try\ stmts_0\ catch\ E_1\ v_1\ stmts_1\ ...\ catch\ E_n\ v_n\ stmts_n}\} \\ \quad = \mathtt{try}\ \mathcal{C}\{\Gamma, stmts_0\}\ \mathtt{catch\ E_1\ v_1}\ \mathcal{C}\{\Gamma, stmts_1\}\ ... \\ \qquad \mathtt{catch\ E_n\ v_n}\ \mathcal{C}\{\Gamma, stmts_n\} \\ \mathcal{C}\{\Gamma, \mathtt{try}\ stmts_0 \mathtt{catch\ E_1 v_1 stmts_1 ... catch\ E_n\ v_n\ stmts_n\ finally\ stmts_{n+1}}\} \\ \quad = \mathtt{try}\ \mathcal{C}\{\Gamma, stmts_0\}\ \mathtt{catch\ E_1\ v_1}\ \mathcal{C}\{\Gamma, stmts_1\}\ ... \\ \qquad \mathtt{catch\ E_n\ v_n}\ \mathcal{C}\{\Gamma, stmts_n\} \mathtt{finally}\ \mathcal{C}\{\Gamma, stmts_{n+1}\}\end{array}}$$

Fig. 11. Java$_s$ types for exceptions

5.1 Properties of the Java$_s$ Type System

The following lemma says that the Java$_s$ type system is deterministic, and that in a complete Java$_s$ program any class that widens to a superclass or superinterface provides an implementation for each method exported by the superclass or superinterface.

Lemma 3 *For any well-formed environment Γ, variable types* $\mathtt{T}$, $\mathtt{T_1}$, ...,$\mathtt{T_n}$, $\mathtt{T_{n+1}}$, *class* $\mathtt{C}$, *Java$_s$ program* $\mathtt{p}$, *with* $\Gamma \vdash \mathtt{p} \Diamond$:

– *If* $\mathtt{p} \vdash \mathtt{C} \sqsubseteq \mathtt{C'}$ *then* $\Gamma \vdash \mathtt{C} \leq_{wdn} \mathtt{C'}$

Furthermore, if

– $\Gamma \vdash \mathtt{C} \leq_{wdn} \mathtt{T}$
– $\mathtt{T_1} \times ... \mathtt{T_n} \rightarrow \mathtt{T_{n+1}} \in MSigs(\Gamma, \mathtt{T}, \mathtt{m})$

then $\exists \mathtt{T'_{n+1}}, \mathtt{C'}$:

– $(\mathtt{C'}, \mathtt{T_1} \times ... \mathtt{T_n} \rightarrow \mathtt{T_{n+1}}) \in MDecs(\Gamma, \mathtt{C}, \mathtt{m})$, *and* $\Gamma \vdash \mathtt{C} \sqsubseteq \mathtt{C'}$
– $MethBody(\mathtt{m}, \mathtt{T_1} \times ... \mathtt{T_n}, \mathtt{p}, \mathtt{C}) = (\mathtt{C'}, \lambda \mathtt{x_1} : \mathtt{T_1}, ... \lambda \mathtt{x_n} : \mathtt{T_n}.\{\mathtt{stmts}\})$ *and* $\Gamma, \mathtt{this} : \mathtt{C'}, \mathtt{x_1} : \mathtt{T_1}, ... \mathtt{x_n} : \mathtt{T_n} \vdash \mathtt{stmts} : \mathtt{T'_{n+1}}$ *and* $\Gamma \vdash \mathtt{T'_{n+1}} \leq_{wdn} \mathtt{T_{n+1}}$

5.2 Absence of the Subsumption Rule

The *subsumption rule* says that any expression of type $\mathtt{T}$ also has type $\mathtt{T'}$ if $\mathtt{T}$ is a subtype of $\mathtt{T'}$. In the case of Java, where subtypes are expressed by the $\leq_{wdn}$

$$\frac{\Gamma \vdash_{se} \mathtt{v} : \mathtt{T} \quad \Gamma \vdash \mathtt{T} \leq_{wdn} \mathtt{C} \quad FDec(\Gamma, \mathtt{C}, \mathtt{f}) = (\mathtt{C}, \mathtt{T}')}{\Gamma \vdash_{se} \mathtt{v}.[\mathtt{C}]\mathtt{f} : \mathtt{T}'}$$

$$\frac{\Gamma \vdash_{se} \mathtt{e_i} : \mathtt{T}'_i \quad \mathtt{i} \in \{1...\mathtt{n}\}, \mathtt{n} \geq 0 \quad \Gamma \vdash \mathtt{T}'_i \leq_{wdn} \mathtt{T_i} \quad \mathtt{i} \in \{2...\mathtt{n}\} \quad FirstFit(\Gamma, \mathtt{m}, \mathtt{T}'_1, \mathtt{T_2} \times ... \times \mathtt{T_n}) = \{(\mathtt{T}, \mathtt{MT})\}}{\Gamma \vdash_{se} \mathtt{e_1}.[\mathtt{T_2} \times ... \times \mathtt{T_n}]\mathtt{m}(\mathtt{e_2}...\mathtt{e_n}) : Res(\mathtt{MT})}$$

$$\frac{\Gamma \vdash \mathtt{C} \sqsubseteq \mathtt{C} \quad \forall \mathtt{f}, \mathtt{C}', \mathtt{T}' \text{ with } (\mathtt{C}', \mathtt{T}') \in FDecs(\Gamma, \mathtt{C}, \mathtt{f}) : \exists \mathtt{i} \in \{1...\mathtt{n}\} : \mathtt{f_i} = \mathtt{f}, \mathtt{C_i} = \mathtt{C}', \mathtt{T_i} = \mathtt{T}' \quad \Gamma \vdash_{se} \mathtt{v_i} : \mathtt{T_i} \; \mathtt{i} \in \{1...\mathtt{n}\}}{\Gamma \vdash_{se} \mathtt{new\ C} \ll \mathtt{f_1\ C_1\ v_1}, ...\mathtt{f_n\ C_n\ v_n} \gg : \mathtt{C}}$$

$$\frac{\mathtt{n} \geq 1, \mathtt{k} \geq 0 \quad \Gamma \vdash \mathtt{T} \diamond_{VarType}, \quad NOT\ \Gamma \vdash \mathtt{T} \leq \mathtt{T} \quad \Gamma \vdash_{se} \mathtt{e_i} : \mathtt{int} \quad \mathtt{i} \in \{1...\mathtt{n}\} \quad \Gamma \vdash_{se} \mathtt{v} : \mathtt{T}}{\Gamma \vdash_{se} \mathtt{new\ T}[\mathtt{e_1}]...[\mathtt{e_n}][]_1...[]_k[\![\mathtt{v}]\!] : \mathtt{T}[]_1...[]_{n+k}}$$

Fig. 12. Differences between Java$_{se}$ types and Java$_s$ types

relation, it would have had the form:

$$\frac{\Gamma \vdash \mathtt{e} : \mathtt{T} \quad \Gamma \vdash \mathtt{T} \leq_{wdn} \mathtt{T}'}{\Gamma \vdash \mathtt{e} : \mathtt{T}'}$$

The type system introduced in this paper does not obey the subsumption rule. For instance, the type of `aPhil.like` is `Phil`, but the type of `pascal.like` is `Food`, though $\Gamma_0 \vdash \mathtt{aPhil} : \mathtt{Phil}$, $\Gamma_0 \vdash \mathtt{pascal} : \mathtt{FrPhil}$, and $\Gamma_0 \vdash \mathtt{FrPhil} \leq_{wdn} \mathtt{Phil}$. In fact, introduction of the subsumption rule would make this type system non-deterministic – although [7] develops a system for Java which has a subsumption rule, and in which the types of method call and field access are determined by using the *minimal types* of the expressions.

6 Extending the Type Rules to Java$_{se}$

After giving types to Java$_s$ terms, we also give types to Java$_{se}$ terms. However, the rationale for typing the two languages is different: Java$_s$ typing corresponds to typing performed by a Java compiler, and it determines whether a term is well-formed. Java$_{se}$ typing, on the other hand, does not correspond to type checking actually performed, it is needed in order to express the subject reduction theorem. A Java$_{se}$ term that has emerged by enriching a well-typed Java$_s$ term will be well-typed too, and will have the same type as the latter, *c.f.* lemma 5. Therefore, the Java$_{se}$ type rules correspond to Java$_s$ type rules, except where the expressions have different syntax.

Figure 12 contains the four cases where Java$_{se}$ syntax differs from that of Java$_r$, and therefore, where Java$_{se}$ types differ from Java$_s$ types. The assertion $\Gamma \vdash_{se} \mathtt{t} : \mathtt{T}$ signifies that the Java$_{se}$ term `t` has type `T` in the Java$_{se}$ type system. Thus, we use the subscript *se* to distinguish between type systems.

The first rule describes field access. The difference between the type of a field access expression in Java$_s$ and Java$_{se}$ is, that in Java$_{se}$ the type depends on the descriptor (*i.e.* `C`) instead of the type of the variable on the left of the field access (*i.e.* `T`).

In the second rule we consider Java$_{se}$ method calls: we search for appropriate methods using the descriptor signature, $(\mathtt{T}_2 \times ... \times \mathtt{T}_n)$, instead of the types of the actual expressions, $(\mathtt{T}'_2, ...\mathtt{T}'_n)$. For this search we first examine the class of the receiver expression for a method body with appropriate argument types, and then *its* superclasses:

Definition 9 *For environment Γ, identifier* $\mathtt{m}$*, type* $\mathtt{T}_1$*, argument types* $\mathtt{AT}$*, we define:*
$FirstFit(\Gamma, \mathtt{m}, \mathtt{T}_1, \mathtt{AT}) =$
$\{(\mathtt{T}, \mathtt{MT}) \mid (\mathtt{T}, \mathtt{MT}) \in MDecs(\Gamma, \mathtt{T}_1, \mathtt{m}) \quad and \quad Args(\mathtt{MT}) = \mathtt{AT}\}$

The last two rules describe object and array creation. The requirements are the same as those for Java$_s$, except that we additionally require the initialization values to be of the appropriate type.

6.1 Properties of the Java$_{se}$ Type System

The following lemma states that no more than one signature with argument types $\mathtt{AT}$ can be found for a type $\mathtt{T}$. This signature will always be found in a superclass or superinterface of $\mathtt{T}$. Also, once such a signature is found, the same signature can be found for any subclass or subinterface of $\mathtt{T}$.

Lemma 4 *For a well-formed environment Γ, types* $\mathtt{T}$*,* $\mathtt{T}'$*,* $\mathtt{T}''$*, and argument types* $\mathtt{AT}$*:*

- $card(FirstFit(\Gamma, \mathtt{m}, \mathtt{T}, \mathtt{AT})) \leq 1$
- $\exists \mathtt{MT} : FirstFit(\Gamma, \mathtt{m}, \mathtt{T}, \mathtt{AT}) = \{(\mathtt{T}', \mathtt{MT})\} \implies \Gamma \vdash \mathtt{T} \leq_{wdn} \mathtt{T}'$
- $\exists \mathtt{MT} : FirstFit(\Gamma, \mathtt{m}, \mathtt{T}, \mathtt{AT}) = \{(\mathtt{T}', \mathtt{MT})\} \quad and \quad \Gamma \vdash \mathtt{T}'' \leq_{wdn} \mathtt{T}$
  $\implies \quad \exists \mathtt{T}''' : FirstFit(\Gamma, \mathtt{m}, \mathtt{T}', \mathtt{AT}) = (\mathtt{T}''', \mathtt{MT})$ *and* $\Gamma \vdash \mathtt{T}''' \leq_{wdn} \mathtt{T}'$

Not surprisingly, a well-typed Java$_s$ expression of type $\mathtt{T}$ is enriched into a Java$_{se}$ expression which has the type $\mathtt{T}$ as well.

Lemma 5 *For types* $\mathtt{T}$*,* $\mathtt{T}'$*, environment Γ, Java$_s$ term* $\mathtt{t}$*:*

$$\Gamma \vdash \mathtt{t} : \mathtt{T} \implies \Gamma \vdash_{se} \mathcal{C}\{\Gamma, \mathtt{t}\} : \mathtt{T}$$

7 Java$_r$, the Run Time Language

As we said in the previous section, Java$_{se}$ is an enriched version of Java$_s$, enriched with compile-time type information necessary at run-time. However, at run time, new terms may be reached, whose syntax is not covered by Java$_{se}$. For this, we further extend Java$_{se}$, to obtain Java$_r$, the run time language. Java$_r$ is a pure superset of Java$_{se}$, it corresponds to Don Syme's Java$_R$ from [29], with the difference that Java$_r$ also allows for exceptions. Java$_r$ is a superset of Java$_{light}$ from [32], because Java$_{light}$ does not describe additional artifacts that may arise at run-time only.

$$
\begin{array}{lll}
\mathit{Program} & ::= & (\ \mathit{ClassBody}\ )^* \\
\mathit{ClassBody} & ::= & \texttt{ClassId ext ClassName}\ \{(\ \mathit{MethBody}\ )^*\} \\
\mathit{MethBody} & ::= & \texttt{MethId is}\ (\lambda\ \texttt{ParId} : \mathit{VarType}.)^*\ \{\mathit{Stmts}\ \texttt{;}\ \texttt{return}\ [\mathit{Expr}]\ \} \\
\mathit{Stmts} & ::= & \epsilon\ \mid \mathit{Stmts}\ \texttt{;}\ \mathit{Stmt} \\
\mathit{Stmt} & ::= & \texttt{if}\ \mathit{Expr}\ \texttt{then}\ \mathit{Stmts}\ \texttt{else}\ \mathit{Stmts} \\
 & \mid & \mathit{Var}\ \texttt{:=}\ \mathit{Expr} \mid \mathit{Expr}.\texttt{MethName}(\mathit{Expr}^*) \mid \texttt{throw}\ \mathit{Expr} \\
 & \mid & \texttt{try}\ \mathit{Stmts}\ (\texttt{catch ClassName Id}\ \mathit{Stmts})^*\ \texttt{finally}\ \mathit{Stmts} \\
 & \mid & \texttt{try}\ \mathit{Stmts}\ (\texttt{catch ClassName Id}\ \mathit{Stmts})^+ \\
\mathit{Type} & ::= & \mathit{VarType} \mid \texttt{void} \mid \texttt{nil} \mid \texttt{ClassName-Thrn} \\
\mathit{Expr} & ::= & \mathit{Value} \mid \mathit{Var} \\
 & \mid & \mathit{Expr}.[\mathit{ArgType}]\texttt{MethName}(\mathit{Expr}^*) \\
 & \mid & \texttt{new ClassName} \ll(\texttt{VarName ClassName Value})^* \gg \\
 & \mid & \texttt{new}\ \mathit{SimpleType}\ (\texttt{[}\mathit{Expr}\texttt{]})^+(\texttt{[]})^*[\![\mathit{Value}]\!] \\
 & \mid & \mathit{Stmts} \\
\mathit{Var} & ::= & \texttt{Name} \mid \mathit{Var}\texttt{[}\mathit{Expr}\texttt{]} \mid \texttt{this} \\
 & \mid & \mathit{Var}.[\texttt{ClassName}]\texttt{VarName} \\
 & \mid & \iota_{\texttt{i}}.[\texttt{ClassName}]\texttt{VarName} \mid \iota_{\texttt{i}}\texttt{[}\mathit{Expr}\texttt{]} \qquad \texttt{i}\ \textit{an integer} \\
 & \mid & \texttt{null}.[\texttt{ClassName}]\texttt{VarName} \mid \texttt{null[}\mathit{Expr}\texttt{]} \\
\mathit{Value} & ::= & \mathit{PrimValue} \mid \texttt{null} \mid \mathit{RefValue} \\
\mathit{RefValue} & ::= & \iota_{\texttt{i}} \qquad \texttt{i}\ \textit{an integer} \\
\mathit{PrimValue} & ::= & \mathit{intValue} \mid \mathit{charValue} \mid \mathit{byteValue} \mid \ldots \\
\mathit{VarType} & ::= & \mathit{SimpleType} \mid \mathit{ArrayType} \\
\mathit{SimpleType} & ::= & \mathit{PrimType} \mid \texttt{ClassName} \mid \texttt{InterfaceName} \\
\mathit{ArrayType} & ::= & \mathit{SimpleType}\texttt{[ ]} \mid \mathit{ArrayType}\texttt{[ ]} \\
\mathit{PrimType} & ::= & \texttt{bool} \mid \texttt{char} \mid \texttt{int} \mid \ldots
\end{array}
$$

Fig. 13. Java$_r$ programs

These additional artifacts that may arise at run-time and are not part of Java$_{se}$, but are part of Java$_r$ arise through addresses, the `null` value, and statements as expressions. Addresses have the form $\iota_{\texttt{i}}$; they represent references to objects and arrays, and may appear wherever a value is expected, as well as in array and field accesses. Therefore, Java$_{se}$ variables may have the form $\iota_{\texttt{i}}.[\texttt{ClassName}]\texttt{VarName}$, or $\iota_{\texttt{i}}[\mathit{Expr}]$, and expressions may have the form $\iota_{\texttt{i}}$. An access to `null` may arise during evaluation of array or field access variables, therefore Java$_{se}$ expressions may have the form $\texttt{null}.[\texttt{ClassName}]\texttt{VarName}$, or $\texttt{null}[\mathit{Expr}]$. Furthermore, in order to describe method evaluation through inline expansion rather than closures and stacks, in Java$_{se}$ we allow an expression to consist of a sequence of statements, so that in the operational semantics a method call can be rewritten to a statement sequence.

7.1 Extending the Type Rules to Java$_r$

As stated in the previous section, we gave types to Java$_{se}$ terms, in order to be able to formulate a subject reduction theorem. We shall now have to extend these to cover the types of Java$_r$. The Java$_r$ type rules correspond to Java$_{se}$ type rules,

$$\frac{\sigma(\iota_{\mathtt{i}}) = \ll ... \gg^{\mathtt{C}}}{\Gamma, \sigma \vdash_r \iota_{\mathtt{i}} \ : \ \mathtt{C}} \qquad \frac{\sigma(\iota_{\mathtt{i}}) = [\![...]\!]^{\mathtt{T}[]_1 ... []_n}}{\Gamma, \sigma \vdash_r \iota_{\mathtt{i}} \ : \ \mathtt{T}[]_1 ... []_n} \qquad \frac{\Gamma \vdash \mathtt{T} \leq_{wdn} \mathtt{Object}}{\Gamma, \sigma \vdash_r \mathtt{null} \ : \ \mathtt{T}}$$

$$\frac{\Gamma, \sigma \vdash_r \mathtt{v}[\mathtt{e}] \ : \ \mathtt{T} \quad \Gamma, \sigma \vdash_r \mathtt{e}' \ : \ \mathtt{T}' \quad \mathtt{T}, \mathtt{T}' \neq \mathtt{E\text{-}Thrn}}{\Gamma, \sigma \vdash_r \mathtt{v}[\mathtt{e}] := \mathtt{e}' \ : \ \mathtt{void}} \qquad \frac{\mathtt{v} \neq \mathtt{v}'[\mathtt{e}'] \ \text{ for any } \mathtt{v}', \mathtt{e}' \quad \Gamma, \sigma \vdash_r \mathtt{v} \ : \ \mathtt{T} \quad \Gamma, \sigma \vdash_r \mathtt{e} \ : \ \mathtt{T}' \quad \Gamma \vdash \mathtt{T}' \leq_{wdn} \mathtt{T}}{\Gamma, \sigma \vdash_r \mathtt{v} := \mathtt{e} \ : \ \mathtt{void}}$$

$$\frac{\Gamma \vdash \mathtt{E} \sqsubseteq \mathtt{Exception} \quad \sigma(\iota_{\mathtt{i}}) = \ll ... \gg^{\mathtt{E}}}{\Gamma, \sigma \vdash_r \mathtt{throw} \ \iota_{\mathtt{i}} \ : \ \mathtt{E\text{-}Thrn}} \qquad \frac{\mathtt{cont} \text{ is a context} \quad \Gamma, \sigma \vdash_r \mathtt{t} \ : \ \mathtt{E\text{-}Thrn}}{\Gamma, \sigma \vdash_r \mathtt{cont}\sqsubset \mathtt{t} \sqsupset \ : \ \mathtt{E\text{-}Thrn}}$$

Fig. 14. Difference between Java$_r$ and Java$_{se}$ types

except where Java$_r$ introduces new syntax, or, where necessities of the subject reduction theorem proof require otherwise.

The type of an address ($\iota_{\mathtt{i}}$) depends on the object or array pointed at in the current state σ (states are introduced in section 8); therefore, the type of a Java$_r$ term depends on both the environment *and* the state, and this is why type assertions for Java$_r$ terms `t` have the form $\Gamma, \sigma \vdash_r \mathtt{t} \ : \ \mathtt{T}$. Again, we use a subscript in order to distinguish between the three type systems in our approach.

Figure 14 contains the seven cases where Java$_r$ types differ from Java$_{se}$ types. The reasons for the differences can be classified into three categories. Firstly, those that give types to expressions that may only arise during program execution but do not involve exceptions (*i.e.* the rules for addresses and for `null`). Secondly, those that give types to terms enclosing a thrown exception (the last two rules). Thirdly, those that give types to terms that would be type incorrect in Java$_s$(*i.e.* typing of assignments, and the rules for `null`). The rules in the first category give the same type as that given if the address or `null` were replaced by an identifier of an appropriate class or array type. The rules in the second category make type-correct terms which would have been type-incorrect in Java$_{se}$. However, the evaluation of such terms does not corrupt the integrity of the system, since the operational semantics requires run-time checks to be performed, and exceptions to be thrown, if certain conditions are not satisfied. The rules in the third category involve the type `E-Thrn`, a type which was not available in Java$_{se}$, or Java$_s$.

We now discuss these seven rules in more detail.

The first two rules in figure 14 describe the types of addresses. If an object is stored at address $\iota_{\mathtt{i}}$, *i.e.* $\sigma(\iota_{\mathtt{i}}) = \ll ... \gg^{\mathtt{C}}$, then its class, `C`, is the type of $\iota_{\mathtt{i}}$. If a `k`-dimensional array of `T` is stored at such an address, *i.e.* $\sigma(\iota_{\mathtt{i}}) = [\![...]\!]^{\mathtt{T}[]_1 ... []_k}$, then $\mathtt{T}[]_1 ... []_k$ is the type of this reference.

The third rule says that `null` has any reference type. This rule is required in order to be able to give a type to terms like `null[j-4]`, which, although type incorrect in Java$_s$, may arise during execution of Java$_r$ terms. Such terms ultimately lead to exceptions, but they do not *immediately* raise the exception `NullPE`, because the Java semantics requires other parts of the expression to be evaluated first – in our example, `j-4` has to be evaluated first. In order to be able to prove the subject reduction theorem, such expressions need a type. The effect of this rule is, that Java$_r$ terms do not have unique types.

The fifth and sixth rule describe assignments. The Java$_r$ array assignment rule, suggested to us by Don Syme [27,28], only requires the left hand side and the right hand side to be type-correct. It is weaker than the corresponding assignment type rule in Java$_s$, or Java$_{se}$: it does not require the right hand side to be of a type that can be widened to that of the left hand side. The reason for this weaker requirement is, that the type of an array component may become narrower during evaluation. For example, if `z` is a one dimensional array of `Phil`, then the assignment `z[3]:=aPhil` is type-correct. However, if at run-time `z` happens to contain a reference to an array of `FrPhil`, *i.e.* $\sigma(\texttt{z}) = \iota_{\texttt{i}}$ and $\sigma(\iota_{\texttt{i}}) = [\![...]\!]^{\texttt{FrPhil}[]}$, then `z[3]:=aPhil` will be rewritten to $\iota_{\texttt{i}}$`[3] := aPhil`. Should this term be considered type correct? A term `y`$[3] :=$ `aPhil` would be type incorrect if `y` was declared as an array of `FrPhil`. On the other, hand the evaluation of the term $\iota_{\texttt{i}}[3] :=$ `aPhil` will not stop here. The right hand hand side, in that case `aPhil`, will be evaluated, and if it returns a value which is of a subclass of `FrPhil` then the assignment will be performed, otherwise an exception will be thrown. Therefore, in order to be able to prove subject reduction, the intermediate term $\iota_{\texttt{i}}[3] :=$ `aPhil` has to be considered type correct in Java$_r$. Interestingly, such a distinction between types for array assignments and other assignments is not necessary when using large steps operational semantics [23].

Finally, the last two rules in figure 14 deal with exceptions that have actually been thrown. Namely, the term `throw new E`$\ll\gg$ indicates *potential* throwing of an exception, and would be rewritten to the term `throw` $\iota_{\texttt{i}}$, where $\iota_{\texttt{i}}$ is the address of an object of class `E`. The latter term indicates an exception which has *actually* been thrown, and, according to the rules, it has the type `E-Thrn`. The *context* of an exception, defined in figure 15, encompasses all enclosing terms up to the nearest enclosing `try`...`catch` close, *i.e.* up to the first possible position at which the exception might be handled. According to the last rule in figure 14, the type of a term which is a context for a thrown exception of class `E` is `E-Thrn`. This rule allows the typing of a message expression one of whose arguments threw an exception, assignments whose left hand or right hand side threw an exception, *etc.*

$$
\begin{array}{lll}
\mathit{Context} & ::= & \mathit{ExprCont} \mid \mathit{VarCont} \mid \mathit{StmtCont} \\
\mathit{VarCont} & ::= & \mathit{VarCont}.\texttt{[ClassName]VarName} \mid \mathit{VarCont}\ \texttt{[}\mathit{Expr}\texttt{]} \\
 & \mid & \mathit{Var}\ \texttt{[}\mathit{ExprCont}\texttt{]} \mid \sqsubset\cdot\sqsupset \\
\mathit{ExprCont} & ::= & \texttt{new}\ \mathit{VarType}\ \texttt{[}\mathit{Expr}\texttt{]}_1\ \ldots\ \texttt{[}\mathit{ExprCont}\texttt{]}_k \ldots \texttt{[}\mathit{Expr}\texttt{]}_n \\
 & \mid & \mathit{ExprCont}.\texttt{[}\mathit{ArgType}\texttt{]MethName (}\mathit{Expr}_1\texttt{,}\ldots\mathit{Expr}_n\texttt{)} \\
 & \mid & \mathit{Expr}.\texttt{[}\mathit{ArgType}\texttt{]MethName (}\mathit{Expr}_1\texttt{,}\ldots\mathit{ExprCont}_k\texttt{,}\ldots\mathit{Expr}_n\texttt{)} \\
 & & \quad \text{where}\ \ n \geq 1, 1 \leq k \leq n \\
 & \mid & \sqsubset\cdot\sqsupset \\
\mathit{StmtCont} & ::= & \mathit{VarCont}\ \texttt{:=}\ \mathit{Expr} \mid \mathit{Var}\ \texttt{:=}\ \mathit{ExprCont} \\
 & \mid & \texttt{if}\ \mathit{ExprCont}\ \texttt{then}\ \mathit{Stmts}\ \texttt{else}\ \mathit{Stmts} \\
 & \mid & \mathit{StmtCont}\ \texttt{;}\ \mathit{Stmt} \mid \texttt{return}\ \mathit{ExprCont} \mid \texttt{throw}\ \mathit{ExprCont} \\
 & \mid & \sqsubset\cdot\sqsupset
\end{array}
$$

Fig. 15. Java$_r$ exception contexts

7.2 Properties of the Java$_r$ Type System

Trivially, any well-typed Java$_{se}$ expression retains its type for any state σ.

Lemma 6 *For types* `T`*, environment* Γ*, Java*$_{se}$ *term* `t`*:*

$$\Gamma \vdash_{se} \texttt{t} : \texttt{T} \implies \forall\ \textit{states}\ \sigma : \quad \Gamma, \sigma \vdash_r \texttt{t} : \texttt{T}$$

Notice, that the opposite direction does not hold. For example, for a variable `diningFrPhils` of type `FrPhil[]`, the Java$_r$ term `diningFrPhils[3]:=aPhil` is type correct, but the corresponding Java$_{se}$ term, `diningFrPhils[3]:=aPhil` is not. Furthermore, Java$_r$ expressions may have more than one type.

The type `E-Thrn` characterizes Java$_r$ terms that contain actually thrown exceptions. Thus, the type `E-Thrn` can only be encountered when typing Java$_r$ terms.

Lemma 7 *For any Java*$_r$ *term* `t`*:* $\Gamma, \sigma \vdash_r$ `t : E-Thrn` $\implies$
$\exists$ *context* $\texttt{t}'\sqsubset\cdot\sqsupset$*, and reference* $\iota_{\texttt{i}}$*:* $\texttt{t} = \texttt{t}'\sqsubset\texttt{throw}\ \iota_{\texttt{i}}\sqsupset$*, and* $\sigma(\iota_{\texttt{i}}) = \ll \ldots \gg^{\texttt{E}}$.

8 The Operational Semantics

Figure 16 describes the run-time model for the operational semantics. For a given program `p`, the operational semantics maps configurations to new configurations. Configurations are tuples of Java$_r$ terms and states, or just states. The operational semantics is a mapping from programs and configurations to configurations.

The state is flat; it consists of mappings from identifiers to primitive values or to references, and from references to objects or arrays. Note that references may point to objects, or arrays, but they may not point to other references, primitive values, or `null`– this is so, because pointers in Java are implicit, and there are no pointers to pointers.

An object is annotated by its class, and it consists of a sequence of labels and values. Each label also carries the class in which it was defined; this is needed for labels shadowing labels from superclasses, *c.f.* [17] ch. 9.5. For the philosophers example, $\ll$ `like Phil:` ι_2, `like FrPhil: null` $\gg^{\texttt{FrPhil}}$ is an object of class `FrPhil`. It inherits the field `like` from `Phil`, and has the field `like` from `FrPhil`.

The following state σ_0 contains mappings according to the philosophers example:

$\sigma_0(\texttt{aPhil}) = \iota_1$
$\sigma_0(\texttt{oyster}) = \iota_3$
$\sigma_0(\iota_1) = \ll\texttt{like Phil: } \iota_2\texttt{, like FrPhil: null}\gg^{\texttt{FrPhil}}$
$\sigma_0(\iota_2) = \ll ... \gg^{\texttt{Truth}}$
$\sigma_0(\iota_3) = \ll ... \gg^{\texttt{Food}}$

Arrays carry their dimension and type information, and they consist of a sequence of values for the first dimension. For example, $[\![\texttt{3}, \texttt{5}, \texttt{8}, \texttt{11}]\!]^{\texttt{int}[]}$ is a one dimensional array of integers.

Configuration	::=	$\langle \text{Java}_r \text{ term}, \mathit{state}\rangle \cup \langle \mathit{state}\rangle$
$\rightsquigarrow$	:	Java$_r$ program $\longrightarrow$ *Configuration* $\longrightarrow$ *Configuration*
$\rightsquigarrow_{\texttt{p}}$	:	*Configuration* $\longrightarrow$ *Configuration*
State	::=	$(\texttt{Ident} \longrightarrow \mathit{Value})^* \cup (\mathit{RefValue} \longrightarrow \mathit{ObjectOrArray})^*$
ObjectOrArray	::=	*Object* | *Array*
Object	::=	$\ll(\texttt{LabelName ClassName} : \mathit{Value})^* \gg^{\texttt{ClassName}}$
Array	::=	$[\![(\mathit{Value})^*]\!]^{\mathit{ArrayType}}$

Fig. 16. Java$_r$ run-time model

8.1 State, Object Operations, Ground Terms

In this section we define operations on objects, arrays and states. These operations are well-defined, only if the object, array or state "conforms" to the types expected by the environment, a requirement introduced in definition 12.

Definition 10 *For object,* $\texttt{obj} = \ll \texttt{l}_1\ \texttt{C}_1\texttt{:val}_1, \texttt{l}_2\ \texttt{C}_2 : \texttt{val}_2, ..., \texttt{l}_\texttt{n}\ \texttt{C}_\texttt{n}\texttt{:val}_\texttt{n} \gg^{\texttt{C}'}$*, state* σ*, value* `val`*, reference* $\iota_\texttt{i}$*, identifier or reference* `z`*, class* `C`*, field identifier* `f`*, integers* `m`*,* `k` *with* $\texttt{m} \geq 0$*, array* $\texttt{arr} = [\![\texttt{val}_0, ...\texttt{val}_{\texttt{n}-1}]\!]^{\texttt{T}[]_1 \cdots []_\texttt{m}}$*, we define:*

- *the* access to field `f` declared in class `C` *as* `obj(f,C)`:
  $\texttt{obj}(\texttt{f}, \texttt{C}) = \texttt{val}_\texttt{i}$ *if* $\texttt{f} = \texttt{l}_\texttt{i}$ *and* $\texttt{C} = \texttt{C}_\texttt{i}$

- *the* access to component `f`, `C` of an object *stored at reference* $\iota_\texttt{i}$*, in state* σ :
  $\sigma(\iota_\texttt{i}, \texttt{f}, \texttt{C}) = \sigma(\iota_\texttt{i})(\texttt{f}, \texttt{C})$

– *the* access to the k^{th} component *of* $\mathtt{arr}$, $\mathtt{arr}[k]$:
  $\mathtt{arr}[k] = \mathtt{val}_k$ *if* $0 \leq k \leq n-1$
– *a new state,* $\sigma' = \sigma[z \mapsto val]$*, such that:*
  $\sigma'(z) = val$
  $\sigma'(z') = \sigma(z')$ *for* $z' \neq z$:
– *a new object,* $obj' = obj[f, C \mapsto val]$*, a new state,* $\sigma' = \sigma[\iota_i, f, C \mapsto val]$:
  $obj'(f, C) = val$
  $obj'(f', C') = obj(f', C')$ *if* $f \neq f'$ *or* $C \neq C'$
  $\sigma' = \sigma[\iota_i \mapsto \sigma(\iota_i)[f, C \mapsto val]]$
– *a new array,* $arr' = arr[k \mapsto val]$*, and a new state,* $\sigma' = \sigma[\iota_i, k \mapsto val]$:
  $arr'[k] = val$
  $arr'[j] = arr[j]$ *if* $j \neq k$
  $\sigma' = \sigma[\iota_i \mapsto \sigma(\iota_i)[k \mapsto val]]$

We distinguish ground terms which cannot be further rewritten, and l-ground terms, which are "almost ground" and may not be further rewritten if they appear on the left hand side of an assignment:

Definition 11 *A Java$_r$ term* t *is*

– ground *iff* t *is a primitive value, or* $t = null$ *, or* $t = \iota_i$ *for some* i*;*
– l-ground *iff* $t = id$ *for some identifier* id*, or* $t = \iota_i.[C]f$ *for a class* C *and a field* f $t = null.[C]f$ *, or* $t = \iota_i[k]$ *or* $t = null\ [k]$ *for some integer* k*.*

8.2 Program Execution

Figures 18, 17, 19 and 21 describe rewriting of Java$_r$terms. We chose small step semantics because we found this more intuitive. Interestingly, it turns out that large step semantics allow for a simpler proof of subject reduction, and in particular, do not require different type rules for Java$_r$ assignment to array components and the other assignments statements [23]. On the other hand, this allows the description of co-routines [21]. In figure 17 we describe the evaluation of variables, field and array access, and the creation of new objects or arrays.

Figure 18 describes statement execution. Statement sequences are evaluated from left to right. In conditional statements the condition is evaluated first; if it evaluates to `true`, then the first branch is executed, otherwise the second branch is executed. A `return` statement terminates execution. A statement returning an expression evaluates this expression until ground and replaces itself by this ground value – thus modeling methods returning values.

Variables (*i.e.* identifiers, instance variable access or array access) are evaluated from left to right. The rules about assignment in figure 19 prevent an expression like x, or $\iota_i[C]v$, appearing on the left hand side of an assignment from being rewritten further. They allow an expression of the form $u[C1].w[C2].x[C3].y$ to be rewritten to an expression of the form $\iota_j[C3].y$ for some j. Furthermore, there is *no* rule of the form $\langle \iota_j, \sigma \rangle \rightsquigarrow_p \langle \sigma(\iota_j), \sigma \rangle$. This is because there is no explicit dereferencing operator in Java. Objects are passed as references, and they are dereferenced only implicitly, when their fields are accessed.

$$\frac{}{\langle \mathtt{id}, \sigma\rangle \rightsquigarrow_{\mathtt{p}} \langle \sigma(\mathtt{id}), \sigma\rangle}$$

$$\frac{}{\langle \iota_{\mathtt{i}}.[\mathtt{C}]\mathtt{f}, \sigma\rangle \rightsquigarrow_{\mathtt{p}} \langle \sigma(\iota_{\mathtt{i}}, \mathtt{f}, \mathtt{C}), \sigma\rangle}$$

$$\frac{\langle \mathtt{v}, \sigma\rangle \rightsquigarrow_{\mathtt{p}} \langle \mathtt{v}', \sigma'\rangle}{\begin{array}{l}\langle \mathtt{v}[\mathtt{e}], \sigma\rangle \rightsquigarrow_{\mathtt{p}} \langle \mathtt{v}'[\mathtt{e}], \sigma'\rangle \\ \langle \mathtt{v}.[\mathtt{C}]\mathtt{f}, \sigma\rangle \rightsquigarrow_{\mathtt{p}} \langle \mathtt{v}'.[\mathtt{C}]\mathtt{f}, \sigma'\rangle\end{array}}$$

$$\frac{\langle \mathtt{e}, \sigma\rangle \rightsquigarrow_{\mathtt{p}} \langle \mathtt{e}', \sigma'\rangle}{\begin{array}{l}\langle \iota_{\mathtt{i}}[\mathtt{e}], \sigma\rangle \rightsquigarrow_{\mathtt{p}} \langle \iota_{\mathtt{i}}[\mathtt{e}'], \sigma'\rangle \\ \langle \mathtt{null}[\mathtt{e}], \sigma\rangle \rightsquigarrow_{\mathtt{p}} \langle \mathtt{null}[\mathtt{e}'], \sigma'\rangle\end{array}}$$

$$\frac{\mathtt{k} \text{ is integer value}}{\langle \iota_{\mathtt{i}}[\mathtt{k}], \sigma\rangle \rightsquigarrow_{\mathtt{p}} \langle \sigma(\iota_{\mathtt{i}})[\mathtt{k}], \sigma\rangle}$$

$$\frac{\mathtt{k} \text{ is integer value} \quad \langle \ \mathtt{new\ \ NullPE}\ll\gg, \sigma\rangle \rightsquigarrow_{\mathtt{p}} \langle \iota_{\mathtt{i}}, \sigma'\rangle}{\begin{array}{l}\langle \mathtt{null}[\mathtt{k}], \sigma\rangle \rightsquigarrow_{\mathtt{p}} \langle \mathtt{throw}\ \ \iota_{\mathtt{i}}, \sigma'\rangle \\ \langle \mathtt{null}.[\mathtt{C}]\mathtt{f}, \sigma\rangle \rightsquigarrow_{\mathtt{p}} \langle \mathtt{throw}\ \ \iota_{\mathtt{i}}, \sigma'\rangle\end{array}}$$

$$\frac{\begin{array}{l}\iota_{\mathtt{i}} \text{ is new in } \sigma \\ \sigma' = \sigma[\iota_{\mathtt{i}} \mapsto \ll \mathtt{f}_1\ \mathtt{C}_1 : \mathtt{v}_1, \ldots \mathtt{f}_{\mathtt{n}}\ \mathtt{C}_{\mathtt{n}} : \mathtt{v}_{\mathtt{n}} \gg^{\mathtt{C}}]\end{array}}{\begin{array}{l}\langle \ \mathtt{new\ \ C}\ll \mathtt{f}_1\ \mathtt{C}_1\ \mathtt{v}_1, \ldots \mathtt{f}_{\mathtt{n}}\ \mathtt{C}_{\mathtt{n}}\ \mathtt{v}_{\mathtt{n}} \gg, \sigma\rangle \\ \rightsquigarrow_{\mathtt{p}} \langle \iota_{\mathtt{i}}, \sigma'\rangle\end{array}}$$

$$\frac{\begin{array}{l}\iota_{\mathtt{i}} \text{ is new in } \sigma \\ \mathtt{v} = \mathtt{v}_0 = \mathtt{v}_1 \ldots = \mathtt{v}_{\mathtt{n}-1}, \quad \mathtt{n} \geq 0 \\ \sigma' = \sigma[\iota_{\mathtt{i}} \mapsto [\![\mathtt{v}_0, \ldots \mathtt{v}_{\mathtt{n}-1}]\!]^{\mathtt{T}[]}]\end{array}}{\langle \ \mathtt{new\ \ T}[\mathtt{n}][\![\mathtt{v}]\!], \sigma\rangle \rightsquigarrow_{\mathtt{p}} \langle \iota_{\mathtt{i}}, \sigma'\rangle}$$

$$\frac{\begin{array}{l}1 \leq \mathtt{j} \leq \mathtt{k}, \quad \mathtt{k} \geq 1, \quad \mathtt{m} \geq 0 \\ \mathtt{n}_{\mathtt{i}} \geq 0 \qquad \mathtt{i} \in \{1 \ldots \mathtt{j}-1\} \\ \langle \mathtt{n}_{\mathtt{j}}, \sigma\rangle \rightsquigarrow_{\mathtt{p}} \langle \mathtt{n}'_{\mathtt{j}}, \sigma'\rangle\end{array}}{\begin{array}{l}\langle \mathtt{new\ T}[\mathtt{n}_1] \ldots [\mathtt{n}_{\mathtt{j}}] \ldots [\mathtt{n}_{\mathtt{k}}][]_1 \ldots []_{\mathtt{m}}[\![\mathtt{v}]\!], \sigma\rangle \\ \rightsquigarrow_{\mathtt{p}} \langle \mathtt{new\ T}[\mathtt{n}_1] \ldots [\mathtt{n}'_{\mathtt{j}}] \ldots [\mathtt{n}_{\mathtt{k}}][]_1 \ldots []_{\mathtt{m}}[\![\mathtt{v}]\!], \sigma'\rangle\end{array}}$$

$$\frac{\begin{array}{l}\mathtt{m} \geq 1, \mathtt{n} \geq 0, k \geq 2 \\ \iota_{\mathtt{i}} \text{ new in } \sigma \\ \sigma' = \sigma[\iota_{\mathtt{i}} \mapsto [\![\mathtt{null}_0, \ldots \mathtt{null}_{\mathtt{n}-1}]\!]^{\mathtt{T}[]_1 \cdots []_{\mathtt{k}}}]\end{array}}{\langle \mathtt{new\ T}[\mathtt{n}][]_2 \ldots []_{\mathtt{k}}[\![\mathtt{v}]\!], \sigma\rangle \rightsquigarrow_{\mathtt{p}} \langle \iota_{\mathtt{i}}, \sigma'\rangle}$$

$$\frac{\begin{array}{l}\mathtt{k} \geq 1, \mathtt{m} \geq 0 \\ \mathtt{n}_{\mathtt{i}} \text{ ground} \quad \mathtt{i} \in \{1 \ldots \mathtt{k}\} \\ \mathtt{n}_{\mathtt{j}} < 0 \text{ for some} \quad \mathtt{j} \in \{1 \ldots \mathtt{k}\} \\ \langle \ \mathtt{new\ \ NegSzeE}\ll\gg, \sigma\rangle \rightsquigarrow_{\mathtt{p}} \langle \iota_{\mathtt{i}}, \sigma'\rangle\end{array}}{\begin{array}{l}\langle \mathtt{new\ T}[\mathtt{n}_1] \ldots [\mathtt{n}_{\mathtt{k}}][]_1 .. []_{\mathtt{m}}[\![\mathtt{v}]\!], \sigma\rangle \\ \rightsquigarrow_{\mathtt{p}} \langle \mathtt{throw}\ \ \iota_{\mathtt{i}}, \sigma'\rangle\end{array}}$$

$$\frac{\begin{array}{l}\mathtt{n}_1 \geq 0, \ \mathtt{k} \geq 2, \ \mathtt{m} \geq 0, \ \sigma_0 = \sigma \\ \mathtt{T} \text{ is a simple type} \\ \langle \ \mathtt{new\ \ T}[\mathtt{n}_2] \ldots [\mathtt{n}_{\mathtt{k}}][]_1 \ldots []_{\mathtt{m}}[\![\mathtt{v}]\!], \sigma_i\rangle \\ \rightsquigarrow_{\mathtt{p}} \langle \iota_{\mathtt{j}_{\mathtt{i}}}, \sigma_{i+1}\rangle \\ \qquad \text{for all} \quad \mathtt{i} \in \{0 \ldots \mathtt{n}_1 - 1\} \\ \iota_{\mathtt{j}_{\mathtt{i}}} \text{ is new in } \sigma_{\mathtt{n}_{\mathtt{i}}} \qquad \mathtt{i} \in \{0 \ldots \mathtt{n}_1\} \\ \sigma' = \sigma_{n_1}[\iota_{\mathtt{j}_{n_1}} \mapsto [\![\iota_{\mathtt{j}_0}, \ldots, \iota_{\mathtt{j}_{\mathtt{n}_1 - 1}}]\!]^{\mathtt{T}[]_1 \cdots []_{\mathtt{k}+\mathtt{m}}}]\end{array}}{\langle \ \mathtt{new\ \ T}[\mathtt{n}_1] \ldots [\mathtt{n}_{\mathtt{k}}][]_1 \ldots []_{\mathtt{m}}[\![\mathtt{v}]\!], \sigma\rangle \rightsquigarrow_{\mathtt{p}} \langle \iota_{\mathtt{j}_{\mathtt{n}_1}}, \sigma'\rangle}$$

Fig. 17. Expression execution

Array access as described here adheres to the rules in ch. 15.12 of [17], which require full evaluation of the expression to the left of the brackets. Thus, with our operational semantics, the term $\mathtt{a}[(\mathtt{a} := \mathtt{b})[3]]$ corresponds to the term $\mathtt{a}[\mathtt{b}[3]]; \mathtt{a} := \mathtt{b}$.

The last six rules in figure 17 describe the creation of new objects or arrays, *c.f.* ch. 15.8-15.9 of [17]. Essentially, a new value of the appropriate array or class type is created, and its address is returned. The fields of the array, and the components of the object are assigned initial values (calculated at compile time, cf definition 6) of the type to which *they* belong.

For example, for a state σ_{00} the expression $\mathtt{new\ int}[2][3][][][\![0]\!]$ would be executed as: $\langle \ \mathtt{new\ int}[2][3][][][\![0]\!], \sigma_{00}\rangle \rightsquigarrow_{\mathtt{p}'} \langle \iota_7, \sigma_{01}\rangle$ where ι_5, ι_6, and ι_7 are new

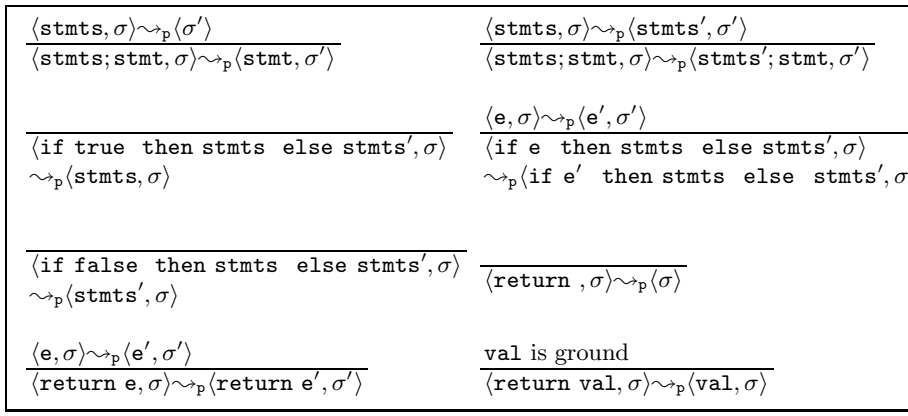


Fig. 18. Statement Execution

in σ_{00}, and they have the following contents in σ_{01}:
$\sigma_{01}(\iota_5) = [\![\texttt{null}, \texttt{null}, \texttt{null}]\!]^{\texttt{int}[][][]}$
$\sigma_{01}(\iota_6) = [\![\texttt{null}, \texttt{null}, \texttt{null}]\!]^{\texttt{int}[][][]}$
$\sigma_{01}(\iota_7) = [\![\iota_5, \iota_6]\!]^{\texttt{int}[][][][]}$

Figure 19 describes the evaluation of assignments. According to the first rule, the left hand side is evaluated first, until it becomes l-ground. Then, according to the next rule, the right hand side of the assignment is evaluated, up to the point of obtaining a ground term. Assignment to variables or to object components modifies the state accordingly.

The last three rules describe assignment to array components where the index being within bounds has to be checked first (if not, `IndOutBndE` is thrown), then the value has to fit the array (if not, `ArrStoreE` is thrown), and, if the two above requirements are satisfied, then the assignment is performed. Fitting, a requirement which ensures that an object or array value is of a type that can be appropriately stored into another array, is described in the definition 11.

Other exceptions (*e.g.* null access) need not be considered in these rules, because they would be checked by the variable rules (figure 18), and then propagated by the exception rules from figure 21. Also, we have *no* rule of the form $\langle \iota_j := \texttt{value}, \sigma\rangle \rightsquigarrow_p \ldots$. This is because in Java overwriting of objects is not possible – only sending messages to them, or overwriting selected instance variables.

Definition 12 *A value* `val` fits *a type* $\texttt{T} = \texttt{T}'[]$ *in a program* `p`*, iff* `val` *is primitive, or* `val`=`null` *, or* $\sigma(\texttt{val}) = \ll\ldots\gg^{\texttt{C}}$ *and* $\texttt{p} \vdash \texttt{C} \sqsubseteq \texttt{T}'$*, or* $\sigma(\texttt{val}) = [\![\ldots]\!]^{\texttt{T}''}$ *and* $\texttt{p} \vdash \texttt{T}'' \sqsubseteq \texttt{T}'$.

Note that a primitive value fits any array type, *e.g.* 4 fits the type `FrPhil[][][]`. This is so, because when primitive values are assigned to array components no run time check needs to be performed, *c.f.* lemma 11. Also, note that in the above

$$\frac{\texttt{v} \text{ is not l-ground} \quad \langle \texttt{v}, \sigma\rangle \rightsquigarrow_{\texttt{p}} \langle \texttt{v}', \sigma'\rangle}{\langle \texttt{v:=e}, \sigma\rangle \rightsquigarrow_{\texttt{p}} \langle \texttt{v}':=\texttt{e}, \sigma'\rangle} \qquad \frac{\texttt{v} \text{ is l-ground} \quad \langle \texttt{e}, \sigma\rangle \rightsquigarrow_{\texttt{p}} \langle \texttt{e}', \sigma'\rangle}{\langle \texttt{v:=e}, \sigma\rangle \rightsquigarrow_{\texttt{p}} \langle \texttt{v:=}\texttt{e}', \sigma'\rangle}$$

$$\frac{\texttt{val} \text{ is ground} \quad \texttt{id} \text{ is an identifier}}{\langle \texttt{id:=val}, \sigma\rangle \rightsquigarrow_{\texttt{p}} \langle \sigma[\texttt{id} \mapsto \texttt{val}]\rangle} \qquad \frac{\texttt{val}, \texttt{k} \text{ are ground} \quad \sigma(\iota_{\texttt{i}}) = [\![\texttt{val}_0 ... \texttt{val}_{\texttt{n}-1}]\!]^{\texttt{T}[]_1 ... []_{\texttt{m}}} \quad 0 > \texttt{k}, \text{ or } \texttt{k} > n-1 \quad \langle \texttt{new IndOutBndE}\ll\gg, \sigma\rangle \rightsquigarrow_{\texttt{p}} \langle \iota_{\texttt{j}}, \sigma'\rangle}{\langle \iota_{\texttt{i}}[\texttt{k}]:=\texttt{val}, \sigma\rangle \rightsquigarrow_{\texttt{p}} \langle \texttt{throw } \iota_{\texttt{j}}, \sigma'\rangle}$$

$$\langle \iota_{\texttt{i}}.[\texttt{C}]\texttt{v:=val}, \sigma\rangle \rightsquigarrow_{\texttt{p}} \langle \sigma[\iota_{\texttt{i}}, \texttt{v}, \texttt{C} \mapsto \texttt{val}]\rangle$$

$$\frac{\texttt{val}, \texttt{k} \text{ are ground} \quad \sigma(\iota_{\texttt{i}}) = [\![\texttt{val}_0 ... \texttt{val}_{\texttt{n}-1}]\!]^{\texttt{T}[]_1 ... []_{\texttt{m}}} \quad 0 \leq \texttt{k} \leq n-1 \quad \texttt{val} \text{ does not fit } \texttt{T}[]_1 ... []_{\texttt{m}} \text{ in } \texttt{p}, \sigma \quad \langle \texttt{new ArrStoreE}\ll\gg, \sigma\rangle \rightsquigarrow_{\texttt{p}} \langle \iota_{\texttt{j}}, \sigma'\rangle}{\langle \iota_{\texttt{i}}[\texttt{k}]:=\texttt{val}, \sigma\rangle \rightsquigarrow_{\texttt{p}} \langle \texttt{throw } \iota_{\texttt{j}}, \sigma'\rangle}$$

$$\frac{\texttt{val}, \texttt{k} \text{ are ground} \quad \sigma(\iota_{\texttt{i}}) = [\![\texttt{val}_0 ... \texttt{val}_{\texttt{n}-1}]\!]^{\texttt{T}[]_1 ... []_{\texttt{m}}} \quad 0 \leq \texttt{k} \leq n-1 \quad \texttt{val} \text{ fits } \texttt{T}[]_1 ... []_{\texttt{m}} \text{ in } \texttt{p}, \sigma}{\langle \iota_{\texttt{i}}[\texttt{k}]:=\texttt{val}, \sigma\rangle \rightsquigarrow_{\texttt{p}} \langle \sigma[\iota_{\texttt{i}}, \texttt{k} \mapsto \texttt{val}]\rangle}$$

Fig. 19. assignment execution

definition the types $\texttt{T}'$ and $\texttt{T}''$ may be array types themselves, and remember that the subclass relationship is monotonic with the array type constructor (*i.e.* $\texttt{p} \vdash \texttt{C} \sqsubseteq \texttt{C}'$ implies that $\texttt{p} \vdash \texttt{C}[] \sqsubseteq \texttt{C}'[]$).

$$\frac{\texttt{val}_{\texttt{i}} \text{ is ground for } \texttt{i} \in \{1...\texttt{k}-1\}, \texttt{n} \geq \texttt{k} \geq 1 \quad \langle \texttt{e}_{\texttt{k}}, \sigma\rangle \rightsquigarrow_{\texttt{p}} \langle \texttt{e}'_{\texttt{k}}, \sigma'\rangle}{\langle \texttt{val}_1.[\texttt{AT}]\texttt{m}(\texttt{val}_2, ..., \texttt{val}_{\texttt{k}-1}, \texttt{e}_{\texttt{k}}, ...\texttt{e}_{\texttt{n}}), \sigma\rangle \rightsquigarrow_{\texttt{p}} \langle \texttt{val}_1.[\texttt{AT}]\texttt{m}(\texttt{val}_2, ..., \texttt{val}_{\texttt{k}-1}, \texttt{e}'_{\texttt{k}}, ...\texttt{e}_{\texttt{n}}), \sigma'\rangle}$$

$$\frac{\texttt{val}_{\texttt{i}} \quad \text{is ground for } \texttt{i} \in \{2...\texttt{n}\}, \texttt{n} \geq 1}{\langle \texttt{null}.[\texttt{AT}]\texttt{m}(\texttt{val}_2, ...\texttt{val}_{\texttt{n}}), \sigma\rangle \rightsquigarrow_{\texttt{p}} \langle \texttt{throw new NullPE}\ll\gg, \sigma\rangle}$$

$$\frac{\texttt{n} \geq 1 \quad \texttt{val}_{\texttt{i}} \quad \text{is ground for } \texttt{i} \in \{1...\texttt{n}\} \quad \sigma(\texttt{val}_1) = \ll ... \gg^{\texttt{C}} \quad \texttt{AT} = \texttt{T}_2 \times ... \times \texttt{T}_{\texttt{n}} \quad \mathit{MethBody}(\texttt{m}, \texttt{AT}, \texttt{C}, \texttt{p}) = (\texttt{C}', \texttt{m is } \lambda \texttt{x}_2 : \texttt{T}_2 ... \lambda \texttt{x}_{\texttt{n}} : \texttt{T}_{\texttt{n}}.\{\texttt{stmts}\}) \quad \texttt{z}_{\texttt{i}} \text{ are new identifiers in } \sigma \quad \sigma' = \sigma[\texttt{z}_1 \mapsto \texttt{val}_1]...[\texttt{z}_{\texttt{n}} \mapsto \texttt{val}_{\texttt{n}}] \quad \texttt{stmts}' = \texttt{stmts}[\texttt{z}_1/\texttt{this}, \texttt{z}_2/\texttt{x}_2, ...\texttt{z}_{\texttt{n}}/\texttt{x}_{\texttt{n}}]}{\langle \texttt{val}_1.[\texttt{AT}]\texttt{m}(\texttt{val}_2, ...\texttt{val}_{\texttt{n}}), \sigma\rangle \rightsquigarrow_{\texttt{p}} \langle \texttt{stmts}', \sigma'\rangle}$$

Fig. 20. Evaluation of method call

Figure 20 describes the evaluation of method calls. The receiver and argument expressions are evaluated left to right, *c.f.* ch. 9.3 in [20]. The first rule describes rewriting the $\mathtt{k}^{th}$ expression, where all the previous expressions (*i.e.* $\mathtt{val_i}, \mathtt{i} \in \{\mathtt{1}...\mathtt{k}-\mathtt{1}\}$) are ground. The second rule requires the exception `NullPE` to be thrown if the receiver is `null`. The third rule describes dynamic method look up, taking into account the argument types, and the statically calculated method descriptor `AT`. The term $\mathtt{t}[\mathtt{t}'/\mathtt{x}]$ has the usual meaning of replacing the variable `x` by the term $\mathtt{t}'$ in the term `t`.

Execution of the method call `aPhil.[Phil]think(aPhil)` results in the following rewrites:

$$\langle \mathtt{aPhil.[Phil]think(aPhil)}, \sigma_0\rangle \leadsto_{\mathtt{p}'} \langle \iota_1.\mathtt{[Phil]think(aPhil)}, \sigma_0\rangle \leadsto_{\mathtt{p}'}$$
$$\langle \iota_1.\mathtt{[Phil]think}(\iota_1), \sigma_0\rangle \leadsto_{\mathtt{p}'} \langle (\mathtt{w.[FrPhil]like{:=}oyster}; ...), \sigma_1\rangle \leadsto_{\mathtt{p}'}$$
$$\langle (...), \sigma_2\rangle$$

where σ_1, σ_2 are:

$$\begin{aligned}
\sigma_1(\mathtt{aPhil}) &= \sigma_0(\mathtt{aPhil}) = \iota_1\\
\sigma_1(\mathtt{oyster}) &= \sigma_0(\mathtt{oyster}) = \iota_3\\
\sigma_1(\mathtt{w}) &= \iota_1\\
\sigma_1(\mathtt{w}') &= \iota_1\\
\sigma_1(\iota_1) &= \sigma_0(\iota_1) = \ll \mathtt{like\ Phil:}\ \iota_2,\ \mathtt{like\ FrPhil:\ null} \gg^{\mathtt{FrPhil}}\\
\sigma_1(\iota_2) &= \sigma_0(\iota_2) = \ll ... \gg^{\mathtt{Truth}}\\
\sigma_1(\iota_3) &= \sigma_0(\iota_3) = \ll ... \gg^{\mathtt{Food}}\\
\sigma_2(\mathtt{z}) &= \sigma_1(\mathtt{z}) \qquad \forall \mathtt{z} \neq \iota_1\\
\sigma_2(\iota_1) &= \ll \mathtt{like\ Phil:}\ \iota_2, \mathtt{like\ FrPhil:}\ \iota_3 \gg^{\mathtt{FrPhil}}
\end{aligned}$$

The rules in figure 21 describe the operational semantics for propagation and handling exceptions. Thus, $\langle \mathtt{try\ throw\ new\ E; f(x)...catch\ E_1\ v_1\ stmts_1}, \sigma\rangle$ would rewrite to $\langle \mathtt{try\ throw}\ \iota_\mathtt{i}\ \mathtt{catch\ E_1\ v_1\ stmts_1}, \sigma\rangle$ then to $\langle \mathtt{stmts}_1', \sigma\rangle$, if `E` is a subclass of $\mathtt{E_1}$. During execution the term maintains its type, which is `void`; the subterm $\mathtt{throw}\ \iota_\mathtt{i}$ has the type `E-Thrn`.

9 Soundness of the Java$_s$ Type System

9.1 Conforming Environments and States

We require objects to be constructed according to their class, array values to conform to their dimension and to consist of values of appropriate types, and variables to contain values of the appropriate type. Furthermore, an environment that contains all definitions from another environment, plus possibly some additional variable definitions is said to conform to the second environment.

Definition 13 *A value* `val` weakly conforms to a type `T` *in an environment* Γ *and a state* σ *iff:*

- `val` *is a primitive value,* `T` *is a primitive type, and* $\mathtt{val} \in \mathtt{T}$*, or*
- $\mathtt{val} = \mathtt{null}$*, and* `T` *is a class, interface or array type, or*
- $\mathtt{val} = \iota_\mathtt{j}$*,* $\sigma(\iota_\mathtt{j}) = \ll ... \gg^{\mathtt{C}}$*, and* $\Gamma \vdash \mathtt{C} \leq_{wdn} \mathtt{T}$*, or*

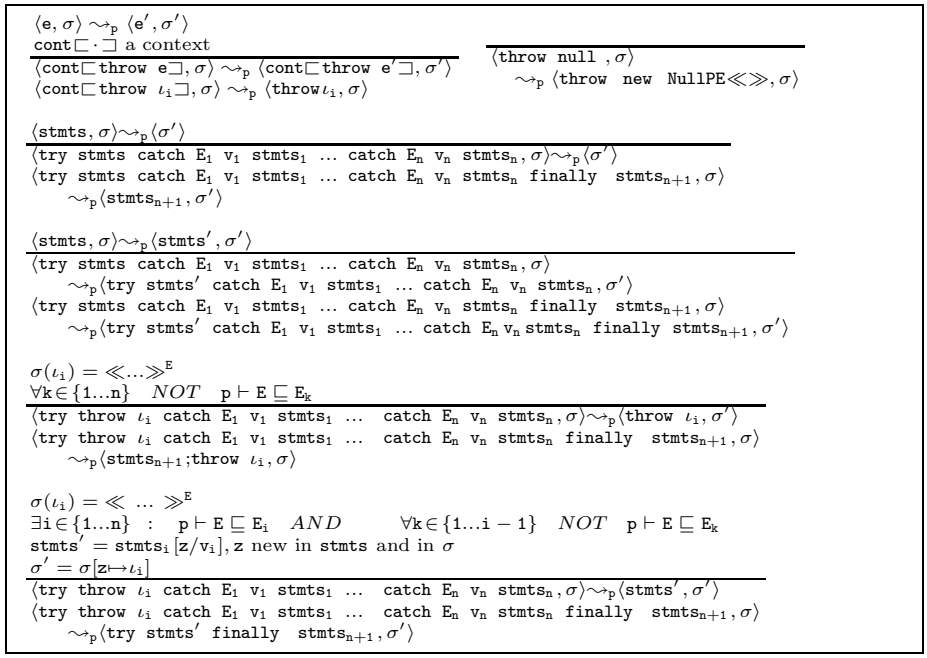


Fig. 21. exception throwing, propagation and handling

– $\mathtt{val}{=}\iota_{\mathtt{j}}$, $\sigma(\iota_{\mathtt{j}}) = [\![\ldots]\!]^{\mathtt{T}'[]_1\ldots[]_{\mathtt{k}}}$ *and* $\Gamma \vdash \mathtt{T}'[]_1\ldots[]_{\mathtt{k}} \leq_{wdn} \mathtt{T}$.

A value $\mathtt{val}$ conforms to a type $\mathtt{T}$ *in an environment* Γ *and a state* σ *iff* $\mathtt{val}$ *weakly conforms to* $\mathtt{T}$ *in* Γ *and* σ *and*

– $\mathtt{val}{=}\iota_{\mathtt{j}}$, $\sigma(\iota_{\mathtt{j}}) = \ll \mathtt{v_1\ C_1} : \mathtt{val_1}, \ldots \mathtt{v_n\ C_n} : \mathtt{val_n} \gg^{\mathtt{C}}$, *and* $\forall$ *labels* $\mathtt{v}$, *classes* $\mathtt{C}'$, *types* $\mathtt{T}'$ *with* $(\mathtt{C}', \mathtt{T}') \in FDecs(\Gamma, \mathtt{C}, \mathtt{v})$, $\exists \mathtt{k}\in\{\mathtt{1\ldots n}\}$ *with* $\mathtt{v_k} = \mathtt{v}$, $\mathtt{C_k} = \mathtt{C}'$, *and* $\mathtt{val_k}$ *weakly conforms to* $\mathtt{T}'$ *in* Γ *and* σ*; or*
– $\mathtt{val}{=}\iota_{\mathtt{j}}$, $\sigma(\iota_{\mathtt{j}}) = [\![\mathtt{val_0}, \ldots \mathtt{val_n}]\!]^{\mathtt{T}'[]_1\ldots[]_{\mathtt{k}}}$, *and* $\forall \mathtt{i} \in \{\mathtt{0\ldots n}\} : \mathtt{val_i}$ *weakly conforms to* $\mathtt{T}'[]_2\ldots[]_k$.

Furthermore, a state σ conforms to an environment Γ *iff for all identifiers* $\mathtt{x}$, *and integers* $\mathtt{i}$

– *if* $\Gamma(\mathtt{x}) \neq \mathtt{Undef}$ *then* $\sigma(\mathtt{x})$ *conforms to* $\Gamma(\mathtt{x})$ *in* Γ,σ*;*
– *if* $\sigma(\iota_{\mathtt{i}}) = \ll\ldots\gg^{\mathtt{C}}$, *then* $\iota_{\mathtt{i}}$ *conforms to* $\mathtt{C}$ *in* Γ, σ*;*
– *if* $\sigma(\iota_{\mathtt{i}}) = [\![\ldots]\!]^{\mathtt{T}[]_1\ldots[]_{\mathtt{n}}}$, *then* $\iota_{\mathtt{i}}$ *conforms to* $\mathtt{T}[]_1\ldots[]_{\mathtt{n}}$ *in* Γ, σ.

Finally, an environment Γ conforms to environment Γ' *iff for any identifier* $\mathtt{x}$*:*

– $\Gamma'(\mathtt{x}) \neq \mathtt{Undef}$ *implies* $\Gamma(\mathtt{x}) = \Gamma'(\mathtt{x})$;
– $\Gamma'(\mathtt{x}) = \mathtt{Undef} \neq \Gamma(\mathtt{x})$, *implies that* $\Gamma(\mathtt{x})$ *is a variable.*

For example, the state σ_0 from section 8 conforms to the environment Γ_0. The "fitting" requirement from definition 11 is weaker than conforming. Also, conforming is defined in terms of an environment, whereas fitting is defined in terms of the more restricted information that is available in the program.

The following lemma states that conforming environments preserve all properties.

Lemma 8 *Given environments Γ, Γ', where Γ conforms to Γ', any term* $\mathtt{t}$*, types* $\mathtt{T}$*,* $\mathtt{t}'$ *program* $\mathtt{p}$*, and argument types* $\mathtt{AT} = \mathtt{T}_2 \times ... \times \mathtt{T}_n$*:*

- $\Gamma \vdash \diamond \implies \Gamma' \vdash \diamond$*;*
- $\Gamma' \vdash \mathtt{p}\ \diamond \implies \Gamma \vdash \mathtt{p}\ \diamond$*;*
- $\Gamma \vdash \mathtt{T} \leq_{wdn} \mathtt{T}' \iff \Gamma' \vdash \mathtt{T} \leq_{wdn} \mathtt{T}'$*;*
- $\Gamma' \vdash \mathtt{t} : \mathtt{T} \implies \Gamma \vdash \mathtt{t} : \mathtt{T}$*;*
- $\mathit{FirstFit}(\Gamma, \mathtt{m}, \mathtt{T}', \mathtt{AT}) = \mathit{FirstFit}(\Gamma', \mathtt{m}, \mathtt{T}', \mathtt{AT})$*;*
- $\Gamma' \vdash_{se} \mathtt{t} : \mathtt{T} \implies \Gamma \vdash_{se} \mathtt{t} : \mathtt{T}$*;*
- $\Gamma', \sigma \vdash_r \mathtt{t} : \mathtt{T} \implies \Gamma, \sigma \vdash_r \mathtt{t} : \mathtt{T}$.

9.2 Properties of Term Evaluation

The operational semantics is deterministic up to renaming of addresses and identifiers. A term containing an actually thrown exception not included by a `try` statement, *i.e.* one with the type `E-Thrn`, will either not terminate, or it will terminate in a `throw` statement. Rewriting variables on the left hand side of assignments does not make their type more special, except for arrays. Program execution may modify the contents of arrays and objects, but will not change their type or class:

Lemma 9 *For a state σ conforming to a well-formed environment Γ, a Java$_{se}$ program with $\Gamma \vdash \mathtt{p}\ \diamond$, a well-typed Java$_{se}$ term* $\mathtt{t}$*:*

- $\langle \mathtt{t}, \sigma \rangle \rightsquigarrow_{\mathtt{p}} \langle \mathtt{t}', \sigma' \rangle$ *and* $\langle \mathtt{t}, \sigma \rangle \rightsquigarrow_{\mathtt{p}} \langle \mathtt{t}'', \sigma'' \rangle$ *implies that* $\mathtt{t}' = \mathtt{t}''$*,* $\sigma' = \sigma''$ *up to renaming of addresses and identifiers. Also,* $\langle \mathtt{t}, \sigma \rangle \rightsquigarrow_{\mathtt{p}} \langle \sigma' \rangle$ *and* $\langle \mathtt{t}, \sigma \rangle \rightsquigarrow_{\mathtt{p}} \langle \sigma'' \rangle$ *implies that* $\sigma' = \sigma''$ *up to renaming of addresses and identifiers. Furthermore, it is impossible to have* $\langle \mathtt{t}, \sigma \rangle \rightsquigarrow_{\mathtt{p}} \langle \mathtt{t}'', \sigma'' \rangle$ and $\langle \mathtt{t}, \sigma \rangle \rightsquigarrow_{\mathtt{p}} \langle \sigma' \rangle$.
- *If* $\Gamma, \sigma \vdash_r \mathtt{t} : \mathtt{E\text{-}Thrn}$*, then, either* $\langle \mathtt{t}, \sigma \rangle \rightsquigarrow_{\mathtt{p}}^*$ *does not terminate, or* $\langle \mathtt{t}, \sigma \rangle \rightsquigarrow_{\mathtt{p}}^* \langle \mathtt{throw}\ \iota_i, \sigma \rangle$
- *For Java$_r$ variables* $\mathtt{v}$*,* $\mathtt{v}'$*, if* $\langle \mathtt{v}, \sigma \rangle \rightsquigarrow_{\mathtt{p}} \langle \mathtt{v}', \sigma' \rangle$*, and* $\Gamma, \sigma \vdash_r \mathtt{v} : \mathtt{T}$*, and* $\mathtt{v}$ *is not l-ground, then* $\Gamma, \sigma \vdash_r \mathtt{v}' : \mathtt{T}'$*,* $\Gamma \vdash \mathtt{T}' \leq_{wdn} \mathtt{T}$ *and* $\mathtt{v}'$ *is not ground. Furthermore, if* $\mathtt{v}$ *is not an array access, then* $\mathtt{T} = \mathtt{T}'$.
- *If* $\langle \mathtt{t}, \sigma \rangle \rightsquigarrow_{\mathtt{p}} \langle \mathtt{t}', \sigma' \rangle$*, then for any* ι_i*, if* $\sigma(\iota_i) = [\![...]\!]^{\mathtt{T}[]_1 \cdots []_n}$ *then* $\sigma'(\iota_i) = [\![...]\!]^{\mathtt{T}[]_1 \cdots []_n}$*, and if* $\sigma(\iota_i) = \ll ... \gg^{\mathtt{C}}$ *then* $\sigma'(\iota_i) = \ll ... \gg^{\mathtt{C}}$.

Don Syme pointed out to us [27,28] that a lemma stating that program execution preserves types up to widening, is necessary for the proof of subject reduction. Interestingly, it turned out that a stronger lemma, than that originally suggested and used in the subject reduction theorem, is possible, namely,

execution of a program does not have *any* effect the type of an expression. This lemma is easier to prove, and considerably facilitates the proof of subject reduction.

Lemma 10 *For Java$_r$ terms* $\mathtt{t}$, $\mathtt{t}'$, $\mathtt{t}''$, *states* σ, σ', *environments* Γ, Γ', *type* $\mathtt{T}''$, *Java$_s$ program* $\mathtt{p}$ *and Java$_r$ program* $\mathtt{p}' = \mathcal{C}\{\!|\Gamma, p|\!\}$, *if*

- $\Gamma \vdash \mathtt{p} \diamond$ *and* $\Gamma, \sigma \vdash_r \mathtt{t} : \mathtt{T}$ *and* $\Gamma, \sigma \vdash_r \mathtt{t}'' : \mathtt{T}''$;
- σ *conforms to* Γ *and* Γ' *conforms to* Γ
- $\langle \mathtt{t}, \sigma \rangle \rightsquigarrow_{\mathtt{p}'} \langle \mathtt{t}', \sigma' \rangle$, *or* $\langle \mathtt{t}, \sigma \rangle \rightsquigarrow_{\mathtt{p}'} \langle \sigma' \rangle$

then

- $\Gamma', \sigma' \vdash_r \mathtt{t}'' : \mathtt{T}''$.

The lemma may be surprising: As stated later in the subject reduction theorem, a term $\mathtt{t}$ when rewritten to a new term $\mathtt{t}'$ has, possibly, a *narrower* type; therefore, one would expect evaluation of the term $\mathtt{t}$ to affect the type of a third term $\mathtt{t}''$. However, according to the above lemma, even if $\mathtt{t}''$ should contain $\mathtt{t}$ as a subterm, its type does not change. The lemma is proven by structural induction over term execution (*i.e.* on $\langle \mathtt{t}, \sigma \rangle \rightsquigarrow_{\mathtt{p}'} \langle \mathtt{t}', \sigma' \rangle$, or $\langle \mathtt{t}, \sigma \rangle \rightsquigarrow_{\mathtt{p}'} \langle \sigma' \rangle$), and then, each case by structural induction on the typing of $\mathtt{t}''$ (*i.e.* on $\Gamma, \sigma \vdash_r \mathtt{t}'' : \mathtt{T}''$). The interesting cases are those where the state changes, *i.e.* the application of the three different assignment rules from figure 19. Assignments do not change the types of variables (these are looked up in the environment). They do not change the type of addresses (as shown in lemma 9). They do not change the type of array access because this depends on the type of the array and not on the type of the actual array component. And they do not change the type of object access because this too depends on the type of the object and the class stored in the descriptor and not on the value stored in the object field.

The *array property*, introduced in the following definition, ensures that checking for fitting when executing array assignments will be sufficient to preserve conformance of the state.

Definition 14 *A Java$_r$ term* $\mathtt{t}$ *has the* array property *for a program* $\mathtt{p}$ *and for a state* σ, *iff for any subterm of* $\mathtt{t}$ *with the form* $\mathtt{v[e]} := \mathtt{e}'$, *with* $\Gamma, \sigma \vdash_r \mathtt{v[e]} : \mathtt{T}$ *and* $\Gamma, \sigma \vdash_r \mathtt{e}' : \mathtt{T}'$, *if NOT* $\Gamma \vdash \mathtt{T}' \leq_{wdn} \mathtt{T}$, *then for appropriate* $\mathtt{n} \geq 0$, $\mathtt{T} = \mathtt{C}[]_1 ... []_n$, $\mathtt{T}' = \mathtt{C}'[]_1 ... []_n$, *and NOT* $\mathtt{p} \vdash \mathtt{C}' \sqsubseteq \mathtt{C}$.

The array property is trivially guaranteed in type correct Java$_{se}$ terms, and thus in any Java$_r$ terms that are the result of enriching type-correct Java$_s$ terms, and it is preserved by the execution of Java$_r$ terms.

Lemma 11 *For an environment* Γ *under which the Java$_s$ term* $\mathtt{t}$ *and Java$_{se}$ term* $\mathtt{t}'$ *are well typed, Java$_{se}$ program* $\mathtt{p}$ *with* $\Gamma \vdash \mathtt{p} \diamond$, $\mathtt{p}' = \mathcal{C}\{\!|\Gamma, p|\!\}$:

- $\mathtt{t}'$ *has the array property for* $\mathtt{p}$ *and any state* σ.
- $\mathcal{C}\{\!|\Gamma, t|\!\}$ *has the array property for* $\mathtt{p}$ *and any state* σ.
- *If* σ *conforms to* Γ, $\mathtt{t}'$ *has the array property for* $\mathtt{p}'$, σ, *and* $\langle \mathtt{t}', \sigma \rangle \rightsquigarrow_{\mathtt{p}'} \langle \mathtt{t}'', \sigma' \rangle$, *then* $\mathtt{t}''$ *has the array property for* $\mathtt{p}'$ *and* σ'.
- $\forall$ *Java$_r$ terms* $\mathtt{t}''$, *states* σ: $\mathtt{t}''$ *has the array property for* $\mathtt{p}$ *and* $\sigma \implies$ $\mathtt{t}''$ *has the array property for* $\mathtt{p}'$ *and* σ

9.3 Subject Reduction and Soundness

The subject reduction theorem says that any non-ground well-typed Java$_{se}$ term either rewrites to another well-typed term of a type that can be widened to the type of the original term, or it rewrites to an exception. Furthermore, the state remains consistent with the environment. The subject reduction theorem of this paper is stronger than usual subject reduction theorems: not only does it guarantee that rewriting preserves types, but it also guarantees that a rewrite step exists for any well-formed, non-ground term. (In that sense it combines the safety and soundness property described in chapter 4 of this book.) In particular, it guarantees for statically type-correct expressions, that the situation where an object cannot execute a message (the Smalltalk counterpart to "`object does not understand message`") will never occur. On the other hand, it does not preclude the usual run-time errors like index out of bound, or wrong assignment to array components; however, it *does* guarantee that such erroneous situations will raise an exception, as opposed to going unnoticed and corrupting the run-time environment.

Theorem 1 Subject Reduction *For a state σ that conforms to an environment Γ, a Java$_{se}$ program* `p` *with $\Gamma \vdash_{se}$* `p` $\gg$*, a non-ground Java$_r$ term* `t` *with the array property for* `p` *and σ, and a type* `T` *with $\Gamma, \sigma \vdash_r$* `t : T`*, there exist σ', Γ',* `t'`*,* `T'` *such that*:

- $\langle$`t`$, \sigma\rangle \rightsquigarrow_{\mathtt{p}} \langle$`t'`$, \sigma'\rangle$*, and $\Gamma', \sigma' \vdash_r$* `t' : T'`*, and* `t'` *has the array property for* `p` *and σ', and*:
  - `T'`=`E-Thrn`*,* `E` *an exception, σ' conforms to Γ, $\Gamma' = \Gamma$*
    or
  - *$\Gamma \vdash$* `T'` $\leq_{wdn}$ `T`*, Γ' conforms to Γ, σ' conforms to Γ'*

  or
- $\langle$`t`$, \sigma\rangle \rightsquigarrow_{\mathtt{p}} \langle \sigma' \rangle$ *and* *σ' conforms to Γ*

Furthermore, if `t` *is a non l-ground variable, then* $\langle$`t`$, \sigma\rangle \rightsquigarrow_{\mathtt{p}} \langle$`t'`$, \sigma'\rangle$ *and* `t'` *is not ground. Also, if* `t` *is a non l-ground variable which isn't an array access, then* `T` $=$ `T'`.
The theorem is proven by structural induction over the derivation of $\Gamma, \sigma \vdash_r$ `t : T`.

When the method call `aPhil.[Phil]think(aPhil)` was evaluated in the philosophers example, then after the third rewrite step, the "environment extension" required by the subject reduction theorem is $\Gamma' = \Gamma_0, \mathtt{w} : \mathtt{FrPhil}, \mathtt{w'} : \mathtt{FrPhil}$. The states σ_1, σ_2 conform to Γ'.

Finally, the soundness theorem states that execution of a well-typed Java$_s$ program will produce a uniquely defined value of the expected type in a state conforming to the definitions, or it will throw an exception which will be propagated to the outermost level, or it will not terminate.

Theorem 2 Soundness *Take any Java$_s$ term* `t`*, a well-formed environment Γ, a type* `T` *with $\Gamma \vdash$* `t : T`*, a Java$_s$ program* `p` *with $\Gamma \vdash$* `p` $\gg$*, and a state σ*

that conforms to Γ*. Then for the Java*$_{se}$ *program* $\mathtt{p}'$, $\mathtt{p}' = \mathcal{C}\{\!\{\Gamma, p\}\!\}$, *there exists a unique Java*$_r$ *term* $\mathtt{t}'$, *and a state* σ', *such that:*

- $\mathtt{T} \neq \mathtt{void}$, $\langle \mathcal{C}\{\!\{\Gamma, \mathtt{t}\}\!\}, \sigma\rangle \rightsquigarrow_{\mathtt{p}'}{}^{*} \langle \mathtt{t}', \sigma'\rangle$, $\mathtt{t}'$ *is ground,*
  $\exists \mathtt{T}' :\ \Gamma, \sigma' \vdash_r \mathtt{t}' : \mathtt{T}'$, $\Gamma \vdash \mathtt{T}' \leq_{wdn} \mathtt{T}$ *and* σ' *conforms to* Γ *or*
- $\mathtt{T} = \mathtt{void}$, *and* $\langle \mathcal{C}\{\!\{\Gamma, \mathtt{t}\}\!\}, \sigma\rangle \rightsquigarrow_{\mathtt{p}'}{}^{*} \langle \sigma'\rangle$ *and* σ' *conforms to* Γ *or*
- $\langle \mathcal{C}\{\!\{\Gamma, \mathtt{t}\}\!\}, \sigma\rangle \rightsquigarrow_{\mathtt{p}'}{}^{*}$ *does not terminate or*
- $\langle \mathcal{C}\{\!\{\Gamma, \mathtt{t}\}\!\}, \sigma\rangle \rightsquigarrow_{\mathtt{p}'}{}^{*} \langle \mathtt{throw}\ \iota_{\mathtt{i}}, \sigma'\rangle$, *and* $\sigma(\iota_{\mathtt{i}}) = \ll ... \gg^{\mathtt{E}}$, *and* $\Gamma \vdash \mathtt{E} \sqsubseteq \mathtt{Exception}$

10 Conclusions

We have given a formal description of the operational semantics and type system for a substantial subset of Java. We believe this subset is reasonably rich and contains many of the features which together might have led to difficulties in the Java type system. By applying some simplifications we obtained a straightforward system, which, we think, does not diminish the application of our results.

Close scrutiny of the language description [17] showed that the semantic issues related to the scope of our investigation are unambiguously answered by [17]. However, we found areas that could have been defined more generally (*e.g.* the return types of methods override those from superclasses and superinterfaces) and others that could have been defined more concisely (*e.g.* the descriptions of widening and of exceptions). Furthermore, in [21,33] we describe problems related to the definition of binary compatibility, and attempt a formalization of this concept.

We believe that the formal system we have developed is very near to Java and to programmers' intuitive ideas about program execution. On the other hand, we now have a large system, and the proofs of the lemmas require the consideration of many cases. The system grew and evolved through many iterations, and during which some omissions crept into the argumentation. The most significant omissions were uncovered by Don Syme and are described earlier on in this paper, and also, in the next chapter of this book[29]. With the modifications he suggested, he was able to validate the subject reduction theorem using his theorem checker. This gives us greater confidence in our results, but it also underlines the importance of the use of theorem checkers for such, rather large systems.

Another proof of the soundness of the Java type system, using Isabelle in a large step semantics is described in [23,32]. Applications of theorem provers for programming language properties are also described in [16,26,31].

We aim to extend the language subset to describe a larger part of Java, and we also hope that our approach may serve as the basis for other studies on the language and its possible extensions [24,3,2]. We are also looking at further language properties such as an abstraction property and binary compatibility [33].

Acknowledgments

We would also like to express our appreciation to Bernie Cohen for awakening our interest in the application of formal methods to Java and especially to all our students whose overwhelming interest in Java convinced us that this work needed to be undertaken.

We are greatly indebted to many people:w for valuable feedback: Peter Sellinger, David von Oheimb, Yao Feng, Sarfraz Khurshid, Guiseppe Castagna, anonymous TAPOS referees, and most particularly to Don Syme.

References

1. M. Abadi and L. Cardelli. A Semantics of Object Types. In *LICS'94 Proceedings*, 1994.
2. Ole Ageson, Stephen Freunds, and John C. Mitchell. Adding paraleterized types to Java. In *OOPSLA'97 Proceedings*, 1997.
3. Joseph A. Bank, Barbara Liskov, and Andrew C. Myers. Parameterized Types and Java. In *POPL'97 Proceedings*, January 1997.
4. Gerald Baumgartner and Vincent F. Russo. Signatures: A language extension for improving type abstraction and subtype polymorphism in C++. *Software–Practice & Experience*, 25(8):863–889, August 1995.
5. John Boyland and Giuseppe Castagna. Type-Safe Compilation of Covariant Specialization: A Practical Case. In *ECOOP'96 Proceedings*, July 1996.
6. P. Canning, William Cook, and William Olthoff. Interfaces for object-oriented programming. In *OOPLSA'89*, pages 457–467, 1989.
7. Giuseppe Castagna. Parasitic Methods: Implementation of Multimethods for Java. Technical report, C.N.R.S, November 1996.
8. Giuseppe Castagna, Giorgio Ghelli, and Giuseppe Longo. A calculus for overloaded functions with subtyping. *Information and Computation*, 117(1):115–135, 15 February 1995.
9. William Cook. A Proposal for making Eiffel Type-safe. In S. Cook, editor, *ECOOP'87 Proceedings*, pages 57–70. Cambridge University Press, July 1989.
10. William Cook, Walter Hill, and Peter Canning. Inheritance is not Subtyping. In *POPL'90 Proceedings*, January 1990.
11. Luis Damas and Robin Milner. Principal Type Schemes for Functional Languages. In *POPL'82 Proceedings*, 1982.
12. Sophia Drossopoulou and Susan Eisenbach. Is the Java type system sound? In *Fourth International Workshop on Foundations of Object-Oriented Languages*, January 1997.
13. Sophia Drossopoulou and Susan Eisenbach. Java is type safe — probably. In *Proceedings of the European Conference on Object-Oriented Programming*, June 1997.
14. Sophia Drossopoulou, Susan Eisenbach, and Sarfraz Khurshid. Is Java Sound? *Theory and Practice of Object Systems*, 1998. to appear, available at http://www-dse.doc.ic.ac.uk/projects/slurp/.
15. A. Goldberg and D. Robson. *SmallTalk-80: The Language and Its Implementation.* Addison-Wesley, 1983.
16. M. Gordon and T.F. Melhams, editors. *Introduction to HOL: A Theorem Proving Environment for Higher Order Logic.* Cambridge University Press, 1993.

17. James Gosling, Bill Joy, and Guy Steele. *The Java Language Specification*. Addison-Wesley, August 1996.
18. R. Harper. A simplified account of polymorphic references. Technical Report CMU-CS-93-169, Carnegie Mellon University, 1993.
19. Daniel Ingalls. The Smalltalk-76 programming system design and implementation. In *POPL'78 Proceedings*, pages 9–15, January 1978.
20. The Java Language Specification, May 1996.
21. Sarfraz Khurshid. Some Aspects of Type Soundness for Java, 1997. BSc thesis.
22. Bertrand Meyer. Static typing and other mysteries of life, http://www.eiffel.com 1995.
23. Tobias Nipkow and David von Oheimb. Java$_{\ell ight}$ is type-safe — definitely. In *POPL Proceedings*, 1998.
24. Martin Odersky and Philip Wadler. Pizza into Java: Translating theory into practice. In *POPL'97 Proceedings*, January 1997.
25. Peter Sellinger. private communication, October 1996.
26. Donald Syme. DECLARE: A Prototype Declarative Proof System for Higher Order Logic. Technical Report 416, Cambridge University, March 1997.
27. Donald Syme. Private Communication, 1997.
28. Donald Syme. Proving Java Type Sound. Technical Report 427, Cambridge University, June 1997.
29. Donald Syme. Proving Java Type Sound. In Jim Alves-Foss, editor, *Formal Syntax and Semantics of Java*, Lecture Notes in Computer Science. Springer-Verlag, 1998. Chapter 4 of this volume.
30. Mads Tofte. Type Inference for Polymorphic References. In *Information and Computation'80 Conference Proceedings*, pages 1–34, November 1980.
31. Myra VanInwegen. Towards Type Preservation in Core SML. Technical report, Cambridge University, 1997.
32. David von Oheimb and Tobias Nipkow. Machine-checking the Java Specification: Proving Type-Safety. In Jim Alves-Foss, editor, *Formal Syntax and Semantics of Java*, Lecture Notes in Computer Science. Springer-Verlag, 1998. Chapter 5 of this volume.
33. David Wragg, Sophia Drossopoulou, and Susan Eisenbach. Java Binary Compatibility is Almost Correct. Technical report, Imperial College, 1998. http://www-dse/projects/SLURP/bc.
34. Andrew Wright and Matthias Felleisen. A Syntactic Approach to Type Soundness. *Information and Computation*, 115(1), 1994.

Proving Java Type Soundness

Don Syme

University of Cambridge Computer Laboratory,
New Museums Site, Pembroke Street
Cambridge CB2 3QG, U.K.
drs1004@cl.cam.ac.uk

1 Introduction

This chapter describes a machine checked proof of the type soundness of a subset of Java (we call this subset Java$_S$). In Chapter 3, a formal semantics for approximately the same subset was presented by Drossopoulou and Eisenbach. The work presented here serves two roles: it complements the written semantics by correcting and clarifying some details; and it demonstrates the utility of formal, machine checking when exploring a large and detailed proof based on operational semantics.[1]

This work contributes to three distinct fields of formal reasoning:

- *The Formal Study of Java*: We contribute a detailed analysis of a significant property of Java, and provide corrections to proofs that are interesting in their own right.
- *Tools for Formal Methods*: This work is a major case study in so-called 'declarative' proof techniques. The tool we use, called DECLARE [Sym97], has been developed by the author to demonstrate the utility of these techniques.
- *Formally Checked Properties of Languages*: This work contributes a tool and a methodology for the general task of machine checking properties of languages.

Most of this chapter should be clear to readers with a basic understanding of operational semantics, formal specification and the results presented in Chapter 3.

Our main aim has not been to find errors. However, a significant error in the original formulation adopted by Drossopoulou and Eisenbach [DE97] was discovered during our work. We also independently rediscovered a significant error in the Java Language Specification [GJS96]. Both errors are described in Section 6.

[1] The latest version of the proofs and specifications described in this document are available on the World Wide Web at http://www.cl.cam.ac.uk/users/drs1004/java-proofs.html. This will be updated to reflect further work on the formalization.

Jim Alves-Foss (Ed.): Formal Syntax and Semantics of Java, LNCS 1523, pp. 83–118, 1999.

The first error resulted in an interesting collaboration between Drossopoulou, Eisenbach and the present author, and led to a deeper understanding of the problems involved. It is as a result of this exercise that we discuss methodology in this chapter, because the methodology we adopted enabled us to find errors quickly and to provide good feedback to the authors of Chapter 3. This demonstrates the positive role that machine checking can play when used in conjunction with existing techniques.

1.1 Outline

This chapter is organized as follows. The rest of this introduction describes, in general terms, just *what* we have proved, and *how* we have gone about doing it. Section 2 delves into the technical content of our model of Java$_S$, and puts into place the building blocks necessary for the proof. As our semantics is based heavily on that of Chapter 3, we only give details where our analysis departs from theirs.

In Sections 4 and 5 we describe the process of machine checking this proof in detail, taking us from a higher-order logic formalization of the problem to a completed proof script. In Section 6 the errors we have mentioned are described, and we summarize and discuss related work in Section 7.

1.2 What Have We Proved?

An introduction to the notion of *type soundness* has already been given at the beginning of Part 2. Briefly, type soundness states that a well-typed Java program will not 'go wrong' at runtime, in the sense that it will never reach a state that violates conditions implied by the typing rules. To illustrate, one aspect of type soundness is captured in the following statement that is taken directly from the Java Language Specification [GJS96]:

> The type [of a variable or expression] limits the possible values that the variable can hold or the expression can produce at runtime. If a runtime value is a reference that is not `null`, it refers to an object or array that has a class ... that will necessarily be compatible with the compile-time type.

In this study we are concerned with the Java language itself, rather than the Java Virtual Machine (JVM). The two are closely related but the difference is non-trivial: for example there are JVM bytecodes that do not correspond to any Java text. Thus it remains a challenge to formalize and verify the corresponding type soundness property for the JVM. However, unlike many high-level/low-level language combinations (e.g. C++/assembler) the type systems of Java and the JVM are closely related, and a comprehensive study of the former is a useful precursor to the study of the latter (see also [Qia97]). Of course, even if an abstract model of Java and/or the JVM is verified, this does not guarantee the soundness of a particular implementation.

The precise formulation of type soundness we use is described in Section 3, but, not surprisingly, it must be expressed in terms of the *inner* workings of a runtime machine, in our case the execution model we use for Java$_S$. This helps explain why it takes so much infrastructure before we can even state type soundness explicitly. Ultimately we would like to verify various "security properties" that are independent of the inner workings of the particular runtime model, but it is beyond the scope of this work to demonstrate such properties.

The Java subset we consider here is that covered in version 2.01 of Drossopoulou and Eisenbach's paper.[2] It includes primitive types, classes with inheritance, instance variables and instance methods, interfaces, shadowing of instance variables, dynamic method binding, statically resolvable overloading of methods, object creation, null pointers, arrays and a minimal treatment of exceptions. An advantage of the approach to formalization we take in this work is that as new features of the language are treated it will be possible to incrementally adjust existing definitions and proofs.

1.3 Five Steps to a Formalized, Machine Checked, Human Readable Proof of Java Type Soundness

In this chapter we are largely concerned with *how* we prove type soundness, *to the point that a machine can check our proof.* Here we step back to look at the methodology in general, to understand what we learn at each stage of the process. The end result of the methodology is a proof outline that is machine checkable, human readable and maintainable as further features are added to our language. A feature of the methodology is that valuable feedback is provided to language researchers at each step.

The steps of the methodology are as follows:

1. **Understand the Problem**
   This first step is so obvious it should hardly need stating: we *must* develop a strong understanding of the problem before we proceed. Like all theorem provers, the tool we use, called DECLARE [Sym97], should only be used when this has been achieved.[3]
2. **Develop a Machine Acceptable Model**
   This involves developing a machine acceptable model of the system, in our case as a DECLARE specification. This process typically uncovers many significant errors and omissions in the original specification, and complications arise, e.g. those that arise from the use of a formal logic rather than informal mathematical notation.

[2] This version was distributed only on the WWW, and is no longer directly available. If a version is needed for reference please contact the authors.

[3] We state this explicitly because some previous attempts at machine-checked language formalization have held that machine formalization should (somehow) be used to reveal the underlying theory (this can be seen by the fact that the theory was not worked out in significant detail prior to using the machine). The two can be done concomitantly but one should not be pursued at the expense of the other.

3. **Validate the Model by Generating an Interpreter and Running Test Cases**
   If specifications were always perfect, then systems probably would be as well, and there would be little need for formal methods. However, specifications nearly always contain mistakes, and thus some process of validation is required. Thus, we must attempt to check that the logical specifications represent a valid model of the Java$_S$ language. This validation is of course non-trivial, and the tools required to perform validation (notably the ability to execute specifications when they fall in an executable subset) are rarely provided by the theorem proving community. Researchers will often rely on the process of *proof* to debug their specifications, a tedious exercise that is not particularly effective.

   We have used two main techniques for validation: typechecking (which is easy as DECLARE is based on higher order logic), and the automatic generation of ML code for an interpreter, directly from the specification. It is not possible to remove all mistakes in the specification via these techniques, but are surprising number are caught.

4. **Formulate All Key Properties**
   We should now have a valid model of the Java$_S$ language, in a form that the computer can accept. We now write the properties that we expect to hold of the specification. Though this may seem simple, it invariably isn't: formulating properties can take as much work as formulating a model, especially for properties of programming languages. Because writing in a formal logic requires attention to detail, this process can uncover many mistakes.

5. **Sketch Outlines of the Proofs and Refine the Proofs Until the Proof Checker is Convinced**
   We now turn to the formal proofs of the propositions we have developed. This involves writing 'rough' proof outlines in a format close to that accepted by DECLARE, and repeatedly refining these proofs until they are accepted as correct by DECLARE's automated proof checker. DECLARE supports the expression of proofs in a simple case-decomposition language that resembles the style used by mathematicians. Most importantly, it supports a migratory path from a rough outline to a machine acceptable outline.

The methodology is like the 'waterfall' methodology of software development: each step can require a return to previous steps, and we iterate until the task is complete. Some steps (e.g. validation) can be highly automated or skipped in later iterations. The methodology differs substantially from that applied to many previous theorem proving projects: it is *top-down*, especially when we write proofs. The advantages of such an approach are well understood from software engineering, and our tool, DECLARE has been developed especially with the aim of supporting it.

Surprisingly, the process of writing rough proof sketches was the most valuable stage in the work. It was here that the flaw in the original proof was discovered (see Section 6). An important by-product of this stage is identifying the key lemmas about component constructs that support the argument. Our methodology supports this elegantly: unless you are formalizing a well-established corpus of mathematics, the necessary lemmas are not at all obvious *a priori*, even if the general direction is clear. Thus support for top-down proof development is essential.

2 Our Model of Java$_S$

With issues of methodology out of the way, we move on to our proof of Java type soundness. However, before we get to the proof itself, we present the details of our model of Java$_S$. We will inherit much from Chapter 3, so we concentrate on the areas where our model differs. The material in this section is quite technical and there are many "building-blocks" to consider: the reader is encouraged to refer back as needed.

The aim of the type correctness proof is to bridge the gap between:

- A model of the *static checks* performed on Java$_S$ programs; and
- A model of the *runtime execution* of Java$_S$ programs.

This section is devoted to describing these two components, which we will connect in Section 3. A picture of the components of the semantics is shown in Figure 1. The "annotated" language Java$_A$ is the result of the static checking process and the "machine-code" language Java$_R$ is the code executed at runtime. Our semantics were originally based on that developed by Drossopoulou and

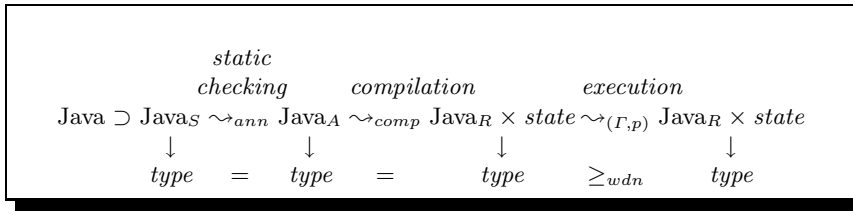


Fig. 1. Components of the Semantics and their Relationships

Eisenbach in version 2.01 of their paper [DE97]. The main differences between our semantics and this version are outlined below. Some of these suggestions have been incorporated into the version presented in Chapter 3.

- We correct minor mistakes, such as missing rules for null pointers, some definitions that were not well-founded (e.g. those for MSigs, FDecs and FDec),

some typing mistakes and some misleading/ambiguous definitions (e.g. the definition of MethBody, and the incorrect assertion that any primitive type widens to the null type).
- We choose different representations for environments, based on tables (partial functions) rather than lists of declarations.
- We differentiate between the source language Java$_S$, the annotated language Java$_A$and the 'runtime terms' Java$_R$. The latter are used to model execution and enjoy subtly different typing rules.
- We adopt a suggestion by von Oheimb (see Chapter 5) that well-formedness for environments be specified without reference to a declaration order.
- We allow the primitive class `Object` to have an arbitrary set of methods (In Chapter 3 `Object` has no methods). It was when considering this extension that one mistake in the Java Language Specification was discovered (see Section 6).
- We do not use substitution during typing, as it turns out to be unnecessary given our representation of environments.
- At runtime we do not choose arbitrary new names for local variables when calling a procedure, but use a system of 'frames' of local variables that makes reasoning about substitution easier (and is also closer to a real implementation based on stacks and offsets).
- The modeling of multi-dimensional arrays in version 2.01 of Drossopoulou and Eisenbach's paper was not faithful to the Java Language Specification, where sub-array dimensions are not constant.
- Arrays in Java support the methods supported by the class `Object` (e.g. `hashValue()`). We include this in our model (with non-trivial consequences). However our model of arrays is still incomplete, as Java arrays support certain array-specific methods and fields, whereas in our treatment they do not.

Figure 2 presents the abstract syntax of Java$_S$ programs.

2.1 The Static Semantics: Environments, Widening, and Visibility

Chapter 3 has already covered the basic components of the static semantics for Java$_S$. The complicating factors for the static semantics are:

- Java allows the use of classes before they are defined. A non-circular class and interface hierarchy must result. Thus *type-checking environments* were defined, extracted from all the classes and interfaces that make up a program. A well-formedness condition is required for these.
- Java allow subtyping in a typical object-oriented fashion, which leads to the *widening* ($\leq_{wdn}$) relation.
- Defining well-formedness for type-checking environments requires knowledge of what identifiers are visible from subclasses. Visibility is defined by relations for traversing the class and interface hierarchies.

prog	= $class_1; \ldots; class_n;$	(programs)
class	= C `extends` C_{sup} `implements` $I_1, \ldots, I_n$ {	(class declaration)
	$field_1; \ldots; field_n;$	
	$method_1; \ldots; method_m;$	
	}	
field	= *type field-name*	(field declaration)
method	= *expr-type method* `(`*type* $x_1, \ldots,$ *type* x_n`)` {	(method declarations)
	$stmt_1; \ldots; stmt_m$	
	`return` *expr*?	
	}	
stmt	= `if` *expr* `then` *stmt* `else` *stmt*	(conditionals)
	| *var* `:=` *expr*	(assignment)
	| { $stmt_1; \ldots; stmt_n;$ }	(blocks)
	| *expr*	(evaluation)
var	= *id*	(local variable)
	| *expr*.field-name	(object field)
	| *expr*`[`*expr*`]`	(array element)
expr	= *prim*	(literal value)
	| *var*	(dereferencing)
	| *expr*.*method-name*`(`*expr*`+)`	(method call)
	| `new` C	(object creation)
	| `new` *comptype*`[`*expr*`]+[]*`	(array creation)

Fig. 2. The Abstract Syntax of $Java_S$

primitive-type	= `bool` | `char` | `short` | `int` |	
	`long` | `float` | `double`	
simple-reference-type	= *class-name* | *interface-name*	
component-type	= *simple-reference-type* | *primitive-type*	
array-type	= *component-type*`[]`^{n}	$(n > 0)$
reference-type	= *simple-reference-type* | *array-type* | `nullT`	
type	= *primitive-type* | *reference-type*	
expr-type	= *type* | `void`	
arg-type	= list of *type*	
method-type	= *arg-type* $\rightarrow$ *expr-type*	

Fig. 3. Types

– Java implementations disambiguate field and method references at compile-time. Method calls may be statically overloaded (not to be confused with the object oriented late-binding mechanism), and fields may be hidden by superclasses.

Type checking environments contain several components (Figure 4). Always present are tables of class and interface declarations, and when typechecking inside method bodies we add a table of variable declarations. We write environments as records ($\langle \ldots \rangle$) , and omit record tag names when it is obvious which record field is being referred to.[4] In the machine acceptable model tables are represented as partial functions, and sets as predicates:

$$\begin{array}{rcl} \alpha \overset{table}{\longmapsto} \beta & \equiv & \alpha \longrightarrow \beta \ \textbf{option} \\ \text{set of } \alpha & \equiv & \alpha \longrightarrow \textbf{bool} \end{array}$$

where the type function **option** has the standard definition.

$$\begin{array}{lll} \textit{class-env} & = \textit{class-names} \overset{table}{\longmapsto} \textit{class-dec} & \\ \textit{interface-env} & = \textit{interface-names} \overset{table}{\longmapsto} \textit{interface-dec} & \\ \textit{variable-env} & = \textit{variable-names} \overset{table}{\longmapsto} \textit{type} & \\ \textit{class-dec} & = \langle \text{ super:} & \textit{class-name}, \\ & \quad \text{interfaces:} & \text{set of } \textit{interface-names}\ , \\ & \quad \text{fields:} & \textit{field-names} \overset{table}{\longmapsto} \textit{type}, \\ & \quad \text{methods:} & \textit{method-names} \times \textit{arg-types} \overset{table}{\longmapsto} \textit{expr-type} \rangle \\ \textit{interface-dec} & = \langle \text{ superinterfaces:} & \text{set of } \textit{interface-names}\ , \\ & \quad \text{methods:} & \textit{method-names} \times \textit{arg-types} \overset{table}{\longmapsto} \textit{type} \rangle \end{array}$$

Fig. 4. Type checking environments

We use Γ for a composite environment, Γ^V, Γ^C and Γ^I its respective components, and $\Gamma(x)$ for the lookup of x in the appropriate table. We also use $x \in \Gamma$ to indicate that x is defined in the relevant table in Γ.

Component types, array types, reference types and regular types are said to be well-formed, written $\Gamma \vdash \textit{object} \diamond_{syntax-category}$ (e.g. `.. |- .. wf_class` in the DECLARE specification) if all classes and interfaces are in scope.

Next we define the subclass ($\sqsubseteq_{class} \equiv$ `subclass_of`), subinterface ($\sqsubseteq_{int} \equiv$ `subinterface_of`) and implements ($:_{imp} \equiv$ `implements`) relations as shown below. All classes are a subclass of the special class `Object`, though we do not have to mention this explicitly as the well-formedness conditions for environments will ensure it.

[4] In the machine acceptable model we do not use such conveniences: records are represented as tuples.

$$\frac{\Gamma \vdash C \diamond_{class-name}}{\Gamma \vdash C \sqsubseteq_{class} C} \ (reflC) \qquad \frac{\Gamma(C).\text{super} = C_{sup} \quad \Gamma \vdash C_{sup} \sqsubseteq_{class} C'}{\Gamma \vdash C \sqsubseteq_{class} C'} \ (stepC)$$

$$\frac{I \in \Gamma}{\Gamma \vdash I \sqsubseteq_{int} I} \ (reflI) \qquad \frac{I_k \in \Gamma(I).\text{interfaces} \quad \Gamma \vdash I_k \sqsubseteq_{int} I'}{\Gamma \vdash I \sqsubseteq_{int} I'} \ (stepI)$$

$$\frac{I_k \in \Gamma(C).\text{interfaces}}{\Gamma \vdash C :_{imp} I_k} \ (implements)$$

In the DECLARE specification, the $\diamond_{class-name}$ and $\sqsubseteq_{class}$ relations are defined by the DECLARE text

```
inductive_relation wf_class
(Object) [rw,prolog]
            ------------------------------------
            TE |- "Object" wf_class

(Decl) [rw,prolog]
            Cdec(TE,C) = SOME(classdec)
            ------------------------------------
            TE |- C wf_class

inductive_relation subclass_of
(Refl) [rw,prolog]

            ------------------------------------
            TE |- C subclass_of C

(Step) [prolog]
            Cdec(TE,C) = SOME(CLASS(C',_,_,_)) &
            TE |- C' subclass_of C''
            ------------------------------------
            TE |- C subclass_of C''
```

Here *TE* is the type environment, and contains a partial function from class-names to class declarations. Extra syntactic detail is required such as `SOME` to indicate that the class is actually in the domain of *CE*.

Keywords such as `rw` and `prolog` are "pragmas" that provide interpretative information to proof tools when the specification is used as a set of logical axioms: in particular `rw` indicates that the rule can be safely used as a (conditional) rewrite, and `prolog` that the rule can be safely used as a backchaining Prolog-style rule.

Subtyping in Java is the combination of the subclass, subinterface and implements relations, and is called *widening*. Defining widening accurately in the machine acceptable model turns out to be a tedious but instructive process: we define it incrementally over the different kinds of types, i.e. over simple reference types ($\leq_{sref}$) then component types ($\leq_{comp}$) then array types ($\leq_{arr}$) and so on through to regular types ($\leq_{wdn}$). We have to be careful about this to avoid errors

that creep in by other approaches: e.g. in the original formulation it appeared that all primitive types are subtypes of `Object`, when in fact only reference types are.

The full rules for widening are given in Appendix B. Note the co-variant rule for arrays eventually leads to the need for runtime typechecking.

The functions FDec, FDecs and MSigs tell us what fields are visible from a given class or interface. They traverse the subclass/subinterface graphs, starting at a particular class/interface.

- FDec: Finds the 'first visible' definition of a field starting at a particular class. A set is returned, with at most one element when the environment is well-formed.
- MSigs: Finds all the methods visible from a reference type. Methods with identical argument descriptors hide methods further up the hierarchy, though return types may be different.
- FDecs: Finds all fields, including hidden ones. This is used to determine the runtime fields of an object.

In Drossopoulou and Eisenbach's original formulation these definitions were given as recursive functions, and only make sense for well-formed environments, as the search may not terminate for circular class and interface hierarchies. Unfortunately the constructs are themselves used in the definition of well-formedness below. To avoid this problem we define the constructs as inductively defined sets. Our definitions are given in Appendix C.

MSigs is defined by first defining MSigs_C MSigs_I and MSigs_A for the visible methods from the three different reference types. The methods visible from arrays and interfaces include all methods found in the type `Object`. Whether this should be the case for interfaces is the subject of discussion in Section 6.1.

2.2 The Static Semantics: Well-Formedness, Type Checking, and Java$_A$

Well-formedness for type checking environments is essential ($\vdash\ \Gamma\ \diamond_{tyenv}\ \equiv$ `wf_tyenv`) to ensure that subclasses provide methods compatible with their superclasses. Drossopoulou and Eisenbach originally formulated this by an incremental process, where the environment is constructed from a sequence of definitions. We originally followed this formulation, but von Oheimb (See Chapter 5) has pointed out that this is not necessary, since the definition is independent of any ordering constraints (however a finiteness constraint is needed to ensure no infinite chains of classes not terminating in 'Object' exist).

In the machine acceptable model, every class declaration in a well-formed environment must satisfy:

- Its superclass and implemented interfaces must be defined and no circularities can occur in the hierarchy;
- Any methods that override inherited methods (by having the same name and argument types) must have a narrower return type;

– All interfaces must be implemented by methods that have narrower return types.

In addition the class `Object` must be defined and have no superclass, superinterfaces or fields. In the DECLARE specification these are written as follows (the italicized labels are used in proofs to refer to facts that are deducible from the well-formedness of an environment).

```
Cdec(TE,C) = SOME(CLASS(Csup,Is,fields,methods)) →
   ∀Csup'. Csup' = SOME(Csup) →
    ~(TE |- Csup' subclass_of C) [no_circular_classes] &
    (∀m at rt1.  MSigsC(TE,Csup')(m,MT(at,rt1)) →
    (∃rt2. MSigsC(TE,C)(m,MT(at,rt2)) &
       TE |- rt2 expty_widens_to rt1) [class_return_types_wider])

   Csup = NONE →
        C = "Object" [only_Object_has_no_superclass] &
        fields = { } [Object_has_no_fields] &
        Is = { } [Object_implements_no_interfaces]

   ∀m mt1. (m,mt1) :: methods →
   ∀mt2. (m,mt2) :: methods →
      Args(mt1) = Args(mt2) → mt1 = mt2 [class_argtypes_unique]
```

Several constraints mentioned in Chapter 3 are guaranteed by the types of the constructs we have used to represent environments. A similar set of constraints must hold for each interface declaration.

We can now define the static checks performed on Java$_S$ programs, and can assume we are operating with a well-formed type environment. Our rules differ from Chapter 3 only in that we translate to a simpler, annotated version of Java$_S$ called Java$_A$ rather than the runtime language Java$_R$. We later prove that the compilation process preserves types.

We do not give the full details of the typing rules here, since they follow the rules given in Chapter 3 very closely. The rules give rise to a series of relations for Java$_S$ (`var_hastype`, `exp_hastype`, `stmt_hastype` through to `prog_hastype`) and similar relations for Java$_A$ (`avar_hastype` through to `aprog_hastype`). The annotation process is also described by a relation ($\rightsquigarrow_{ann}$ ≡ `prog_annotates_to`). As an example, the typing rule for references to local (stack) variables in both the unannotated and annotated languages is:

```
 PLOOKUP(VE)(x) = SOME(vt)
-------------------------------------------
 (TE,VE) |- Id(x) var_hastype vt
```

The typing rule for method calls in the unannotated language is:

```
LEN(args) = n &
LEN(argtys) = n &
(∀j. j < n → (TE,VE) |- EL(j)(args) exp_hastype SOME(EL(j)(argtys))) &
(TE,VE) |- e exp_hastype SOME(vt) &
MostSpecC(TE,vt,m,argtys)(mt) &
(∀y. MostSpecC(TE,vt,m,argtys)(y) → mt = y)
-----------------------------------------------------------------
(TE,VE) |- Call(e,m,args) exp_hastype Res(mt)
```

Note that iterated constructs are replaced by (bounded) universal quantification: the first three lines of the rule correspond to a side condition like:

$$\text{each } arg_i \text{ has some type } ty_i \ (1 \leq i \leq n)$$

Note the index change to take advantage of the inbuilt theory of natural numbers and zero-indexed lists and the use of the inbuilt list operators `EL` and `LEN`. The definition of MostSpec can be found in Chapter 3. The typing rule for the same construct in the annotated language is:

```
LEN(args) = nargs &
LEN(AT) = nargs &
(TE,VE) |- e aexp_hastype SOME(vt) &
MSigs(TE,vt)(m,MT(AT,rt))
-------------------------------------------------
(TE,VE) |- CallA(e,AT,m,args) aexp_hastype rt
```

This completes our presentation of the static checks performed for the Java_S language. We now move onto the runtime model of execution.

2.3 The Runtime Semantics: Configurations, Runtime Terms, and State

Chapter 3 models execution by a transition semantics, i.e. a 'small step' rewrite system [Plo91]. A *configuration* (s, t) of the runtime system has a *state* s and a *runtime term* (rterm) t. The rterm represents expressions yet to be evaluated and the partial results of terms evaluated so far. The configuration is progressively modified by making reductions. The rewrite system specifies an abstract machine, which is an inefficient but simple interpreter for Java_S.

A small step system is chosen over a 'big step' (evaluation semantics) since we want to reason about non-terminating programs, and later will want to model non-determinism and concurrency. Using a small-step system imposes significant overheads in the type soundness proof (e.g. with a big-step rewrite system certain intermediary configurations need not be considered), but this seems unavoidable.

Runtime terms (the language Java_R) are Java_A programs with the addition of addresses, exception packets and the method bodies that have been called. There are three types of rterms: expressions, variables and statements, and thus there are really three different types of configurations. As an intuition for what

we mean by this, consider the "top level" configuration: it always contains an expression (and a state) since Java begins execution with the `main` static method from a given class, and this eventually evaluates to an integer.

In our model, the program state consists of two components: a *list of frames* of local variables and a *heap* containing objects and arrays. Neither are garbage collected[5]. Heap objects are annotated with types for runtime typechecking (in

$$
\begin{array}{lll}
\mathit{state} & = \langle \text{ frames: list of } (\mathit{id} \overset{table}{\longmapsto} \mathit{val}), & \\
 & \quad \text{heap: } \mathit{addr} \overset{table}{\longmapsto} \mathit{heap\text{-}object} \rangle & \\
 & & \\
\mathit{heap\text{-}object} = & \ll \mathit{fld}_1 \mapsto \mathit{val}_1, \ldots, \mathit{fld}_n \mapsto \mathit{val}_n \gg^C & (\mathit{object}) \\
 \mid & \texttt{[[}\mathit{val}_1, \ldots, \mathit{val}_n\texttt{]]}^{type} & (\mathit{array})
\end{array}
$$

Fig. 5. State

the case of arrays this is the type of values stored in the array). We use the symbol $\oplus$ to indicate adding a new frame at the next available frame index, $s(id)$ and $s(addr)$ for looking up local variables and objects, and $s(id) \leftarrow val$ and $s(addr) \leftarrow heap\text{-}obj$ for assigning things into the respective components of the state.

2.4 The Rewrite System

The reduction of rterms is specified by three relations, one for each syntax category: $\overset{exp}{\rightsquigarrow}_{(\Gamma,p)}$, $\overset{var}{\rightsquigarrow}_{(\Gamma,p)}$ and $\overset{stmt}{\rightsquigarrow}_{(\Gamma,p)}$ ($\equiv$ `exp_reduces_to`, `var_reduces_to` etc.). Global parameters are an environment Γ (containing the class and interface hierarchies, needed for runtime typechecking) and the program p being executed. p contains Java$_A$ terms: each time a method is executed we create a Java$_R$ term for the body of that method.

A term is *ground* if it is in normal form, i.e. no further reduction can be made. Being ground is a *syntactic test*, and the test can depend on the syntax category "from which a term is viewed". A local variable *id* is ground if *id* is a variable, but not ground if *id* is an expression (this is the standard distinction between lvalues and rvalues in C). Formally, *ground* is defined as follows:

- A value is ground *iff* it is a primitive value or an address.
- An expression is ground *iff* it is a ground value.

[5] In future versions of the semantics a garbage collection rule collecting inaccessible items at any time may be added. Garbage collection is semantically visible in Java because of the presence of 'finally' methods that get called before an object is deallocated.

- A variable is ground *iff* all its component expressions are ground.
- A statement is ground *iff* it is an empty block of statements or a ground expression.

There are 36 rules in our rewrite system. 15 of them are "redex" rules that specify the reduction of expressions in the cases where sub-expressions have reductions. A sample from the DECLARE specification is:

```
 (stmt,s) stmt_reduce(TE,p) (stmt',s')
------------------------------------------------------------------------
 (RBlock(stmt#stmts),s) stmt_reduce(TE,p) (RBlock(stmt'#stmts),s')‘
```

11 of the rules specify the generation of exceptions: 5 for null pointer dereferences, 4 for bad array index bounds, one for a bad size when creating a new array and one for runtime type checking when assigning to arrays. A simple example is:

```
 exp_ground(exp) = T &
 val_ground(val) = T‘
-----------------------------------------------------------
 (RAssign(RAccess(RValue(RAddr(NONE)),exp),RValue(val)),s)
                                                    stmt_reduce(TE,p)
 (RExpr(RValue(RExn("NullPointExc"))),s)‘
```

The array creation rule is:

```
 ndims = LEN(dims) &
 (∀j. j < ndims → exp_ground(EL(j)(dims))) &
 LEN(dimns) = ndims &
 (∀j. j < ndims →
  ∃i32. EL(j)(dims) = RValue(RPrim(Int(i32))) &
  dest_int32(i32) >>= 0 &
  dest_int32(i32) = EL(j)(dimns)) &
  s = (frames,heap) &
  alloc(heap,st,dimns,ext) = (val,heap') &
  (frames,heap') = s'
-------------------------------------------------------------
 (RNewArray(st,dims,ext),s) exp_reduce(TE,p) (RValue(val),s')
```

Here alloc represents the recursive process of allocating $k_1 \times \ldots k_{n-1}$ arrays that eventually point to initial values appropriate for the type *type*. This process is described in detail in [GJS96].[6]

[6] This model of array creation would need to be modified if threads or constructors are considered. Array creation is not atomic with respect to thread execution, It may execute constructors (and thus may not even terminate), and may raise an out-of-memory exception.

2.5 Runtime Typechecking

Java performs runtime typechecks at just two places: during array assignment, and when casting reference values. Runtime typechecking is needed for array assignment because of the well-known problem with a co-variant array typing rule. Casts are not covered in this chapter: they are a trivial extension once runtime checking for arrays is in place.

Runtime typechecking is performed by simply checking that the real (i.e. runtime) type of any reference object, as stored in the state, is narrower than the real type of the array cell it is being assigned to. This means the runtime system must have access to the program class/interfaces hierarchies (as the JVM does).

> *An aside:* The notion of runtime type checking from Chapter 3 (weak conformance) is a little too strong: it allows the runtime machine to check the conformance of primitive values to primitive types. No realistic implementation of Java checks at runtime that a primitive type such as `int` fits in a given slot. The problem stems from the fact that conformance is used for two purposes:
>
> – to represent the procedure invoked at runtime to do runtime typechecking
> – as an abstract concept used to formulate type soundness.

The function typecheck checks that a stored type is compatible with a given type. It succeeds for an address *addr*, a type *ty* in a heap h if:

– $h(addr) = \ll ... \gg^C$ and *ty* is wider than C
– or $h(addr) = \texttt{[[...]]}^{ty'}$ and *ty* is wider than $ty'\texttt{[]}$

In future versions of the semantics this will not perform compatibility checks for primitive or `null` values.

2.6 The Model as a DECLARE Specification

DECLARE specifications can be interpreted as axioms in an appropriate logic, or, if executable, as a specification of an interpreter. The documents we have seen fragments from are *abstracts*, i.e. summaries of theories that are checked to be consistent extensions of higher order logic. The declaration forms available are simple (non-recursive) definitions, recursive datatype definitions (mutually recursive and recursive through positive type functions like *list*), inductive relations (again mutually recursive, with any monotonic operators), and recursive functions with a well-founded measure.[7]

The syntax classes and semantic objects (*exp*, *type*, *state* etc.) are easily defined in DECLARE using logical datatypes, partial functions and sets — we will not give an example here. As we have seen inductive relations [CM92,Pau94] are formulated by specifying a set of rules, and giving a name to each. When

[7] Not all the features listed here are fully implemented in the version of DECLARE used for this work, for example monotonicity conditions are not currently checked.

treated as a logical specification, DECLARE generates the appropriate axioms for the least fixed point of the set of rules.

Formalizing the runtime rewrite system as an inductive relation is relatively straight forward given DECLARE's collection of background theories.

The machine-acceptable specification runs to around 2500 lines in total. The use of three similar versions of the language results in some unfortunate duplication that seems hard to avoid: e.g. we have three sets of typing rules that are very similar. Perhaps most importantly, the specification was easily read and understood by the authors of Chapter 3 when shown to them.

3 Formulating Key Properties

In this section we formulate, in the terms of the model just developed, the type soundness property we will prove. The main substantive differences with Drossopoulou and Eisenbach's original formulation are:

- We distinguish between the *safety* property (type soundness) and a *liveness* property, that says that the runtime machine can always proceed if the result is not yet ground. This formulation is correct for non-deterministic language constructs.
- We correct the rule for typing array assignments at runtime.

Loosely speaking, type soundness says that as evaluation progresses the configuration of our rewrite system always conforms to the types we expect, and that terms only ever narrow in type.

A *frame typing* F is a list of tables of types for local variables. A frame typing indicates what types we expect local variables to conform to. We require other auxiliary concepts too:

- typing for rterms (`rexp_hastype`, `rvar_hastype` etc.);
- self-consistency of a heap ($\diamond_{heap} \equiv$ `wf_heap`);
- conformance between frame typings and the local variables in a state ($\rightleftharpoons_{frames} \equiv$ `frames_conform_to`);
- conformance between two heaps ($\rightleftharpoons_{heap} \equiv$ `heap_conforms_to`);
- widening between two frame typings ($\leq_{ftyp} \equiv$ `ftyenv_leq`)

and define these in the sections that follow.

Theorem 1. Type Soundness *For a well-formed type environment Γ, an annotated program p that typechecks and a state s_0 that conforms to some frame typing F_0, if a well-typed term t_0 rewrites to some t_1 and a new state s_1, then either t_1 represents a raised exception, or there exists a new, larger frame typing F_1 such that t_1 has some narrower type than t_0 in the new state and environment, and s_1 conforms to F_1. That is, if*

- $\vdash \Gamma \diamond$
- $\Gamma \vdash p \diamond$

- $\Gamma \vdash h_0 \diamond$
- $\Gamma, h_0 \vdash f_0 \rightleftharpoons F_0$
- $\Gamma, h_0, F_0 \vdash t_0 : ty_0$ *and*
- $t_0, s_0 \rightsquigarrow_{(\Gamma,p)} t_1, s_1$ *with* $s_0 = (f_0, h_0)$ *and* $s_1 = (f_1, h_1)$

then t_1 *represents an exception or there exists* F_1 *and* ty_1 *such that*

- $\Gamma \vdash h_1 \diamond$
- $\Gamma, h_1 \vdash f_1 \rightleftharpoons F_1$
- $\Gamma, h_1, F_1 \vdash t_1 : ty_1$ *and*
- $\Gamma \vdash ty_1 \leq_{wdn} ty_0$.

Note we assume a reduction is made, rather than proving that one exists. This is distinguishes the safety property from the liveness property. In the presence of non-determinism it is not sufficient to prove that a safe transition exists: we want to show that all possible transitions are safe.

Type soundness is in fact three properties, one for each syntax category within rterms. For variables we write the property in DECLARE as follows (a similar property is used for expressions and statements):

```
  TE wf_tyenv ∧
  TE |- p aprog_hastype ∧
  TE |- heap0 wf_heap ∧
  (TE,heap0) |- frames0 frames_conform_to F0 ∧
  (TE,F0,heap0) |- var0 rvar_hastype ty0 ∧
  s0 = (frames0,heap0) ∧
  s1 = (frames1,heap1) ∧
  (var0,s0) var_reduce(TE,p) (var1,s1)
→
   exceptional_var(var1)
∨ ∃F1 ty1.
     TE |- heap1 wf_heap ∧
     (TE,heap1) |- frames1 frames_conform_to F1 ∧
     (TE,F1,heap1) |- var1 rvar_hastype ty1 ∧
     TE |- ty1 widens_to ty0
```

The proof of type soundness is by induction on the derivation of the typing judgment for t_0. The outline originally sketched by Drossopoulou and Eisenbach is a good guide, but is certainly 'rough around the edges.' The latter two syntax categories do not have types, so the statements are simpler. In the proof we strengthen the induction invariant in the following ways:

- $\Gamma \vdash h_0 \rightleftharpoons_{heap} h_1$,
- $F_1 \leq_{ftyp} F_0$,
- If t_0 is a field variable, then t_1 is also and $ty_0 = ty_1$. This is needed because field types on the left of assignments cannot narrow, otherwise runtime type-checking would be needed.
- If t_0 is an array variable, then t_1 is also, and similarly for stack variables.

3.1 Typing for Rterms and Conformance

As in Chapter 3, the typing rules at runtime are those for annotated expressions, with the addition of rules for addresses. These make the typing relation dependent on the current heap.

Note that we no longer demand unique typing: null values can be considered to have any reference type. The rule for assignments must also be different: the new rule drops the requirement that the source type be narrower than the target type in the case of array assignments, since this will be checked by runtime type checking. We will return to this issue in Section 6. The new rule for arrays is:

$$\frac{\begin{array}{c}\Gamma, h \vdash e : ty_e \\ \Gamma, h \vdash arr\,\texttt{[}idx\texttt{]} : ty_v\end{array}}{\Gamma, h \vdash arr\,\texttt{[}idx\texttt{]}\ \texttt{:=}\ e : \texttt{void}}$$

The definitions of conformance we use are similar to those in Chapter 3: A value v *weakly conforms* to a type ty with a heap h and type environment Γ if

- ty is a primitive type and v is an element of that primitive type; or
- ty is a reference type and v is a null pointer; or
- v is an address whose value upon dereferencing $h(v)$ is an instance of a class type C and $\Gamma \vdash C \leq_{wdn} ty$; or
- v is an address whose value upon dereferencing $h(v)$ is an array with elements of type $ty'\texttt{[]}^n$ and $\Gamma \vdash ty'\texttt{[]}^{n+1} \leq_{wdn} ty$.

Value conformance states that the components of an object or array weakly conform. A value v *conforms* to a type ty with heap h and type environment Γ if v weakly conforms to ty and

- if v is an address then $h(v) = \ll \mathit{fldvals} \gg^C$ and for each $(\mathit{field}, \mathit{idx}, ty') \in \mathsf{FDecls}(C)$ $\mathit{fldvals}(\mathit{field})$ is defined and weakly conforms to ty'; and
- if v is an address then $h(v) = \texttt{[[}vec\texttt{]]}^{ty'}$ and each $val \in vec$ weakly conforms to ty'.

A heap h is *self-consistent* in Γ, written $\Gamma \vdash h\,\diamond$ if these hold:

- if $addr$ is an address and $h(addr) = \ll \mathit{fldvals} \gg^C$ then $addr$ conforms to C.
- if $addr$ is an address and $h(addr) = \texttt{[[}vec\texttt{]]}^{ty'}$ then $addr$ conforms to $ty'\texttt{[]}$.

A set of frames f *conforms* to a frame typing F (with a heap h and in Γ), written $\Gamma, h \vdash f \rightleftharpoons F$ if

- every local variable in every frame of f conforms to the corresponding type given in F;

We expect each new heap to maintain value conformance in the following way: for in environment Γ a heap h_1 conforms with a heap h_0 at a set of addresses A, written $\Gamma, A \vdash h_1 \rightleftharpoons_{heap} h_0$ if

– for every *addr* in both A and h_0, if *addr* conformed to some type ty in the context of h_0, then *addr* also conforms to ty in the context of h_1.

We restrict the definition to a set of addresses A to allow for the possibility of garbage collection: we would then demand continued conformance only at a set of 'active addresses'. Our current working definition makes A universal.

3.2 Key Lemmas

The following is a selective list of the lemmas that form the basis for the type soundness proof. These have been proved using DECLARE.

`Object` is the least class
If $\vdash \Gamma \diamond_{tyenv}$ and $\Gamma \vdash \texttt{Object} \sqsubseteq_{class} C$ then $C = \texttt{Object}$.

Widening is transitive and reflexive
These hold for the $\sqsubseteq_{class}$, $\sqsubseteq_{int}$, $\leq_{ref}$, $\leq_{comp}$, $\leq_{arr}$ and $\leq_{wdn}$ relations. The transitivity results only hold for well-formed environments.

Compatible fields and methods exist at subtypes
Methods and fields visible at one type must still be visible at narrower types, though with possibly narrower return types. Put formally, if
$\vdash \Gamma \diamond_{tyenv}$ and
$\Gamma \vdash C_1 \sqsubseteq_{class} C_0$ and
$((C_f, \mathit{fld}), ty_{fld}) \in \mathsf{FDecls}(\Gamma, C_0)$
then $((C_f, \mathit{fld}), ty_{fld}) \in \mathsf{FDecls}(\Gamma, C_1)$.
Similarly if $\Gamma \vdash ty_1 \leq_{ref} ty_0$ and
$(m, ty_{args} \to ty_{ret}) \in \mathsf{MSigs}(\Gamma, ty_0)$
then there exists some ty'_{ret} with
$\Gamma \vdash ty'_{ret} \leq_{wdn} ty_{ret}$ and
$(m, ty_{args} \to ty'_{ret}) \in \mathsf{MSigs}(\Gamma, ty_1)$.

Method fetching behaves correctly
Assuming $\vdash \Gamma \diamond_{tyenv}$ and $\Gamma \vdash \mathit{prog} \diamond$, then if
$(m, ty_{args} \to ty_{ret}) \in \mathsf{MSigs}(\Gamma, ty)$ and
$\mathsf{MethBody}(m, ty_{args}, ty, p) = \mathit{method}$
then $\Gamma \vdash \mathit{method} : ty_{ret}$. That is, fetching the annotated body of a method from the annotated program results in a method of the type we expect.

Compilation behaves correctly
If $\vdash \Gamma \diamond_{tyenv}$ and
$\Gamma \vdash \mathit{mbody} : ty_{ret}$ and
$\Gamma \vdash \mathit{mbody} \rightsquigarrow_{comp} \mathit{rmbody}$
then $\Gamma \vdash \mathit{rmbody} : ty_{ret}$. Note compilation is an almost trivial process in the current system, so this lemma is not difficult.

Relations are preserved under $\rightleftharpoons_{heap}$ and $\leq_{ftyp}$
This holds for the typing, value conformance and frame conformance relations.

Atomic state/frame manipulations preserve $\rightleftharpoons_{heap}$, $\rightleftharpoons_{frames}$ and $\diamond_{heap}$
We prove this for all primitive state manipulations, including object and array allocation, field, array and local variable assignment, and method call. The case for array allocation involves a double induction because of the nested loop used to allocate multi-dimensional arrays.

3.3 Annotation and Liveness

To complement the type soundness proof, we prove that the process of annotation preserves types:

```
   TE wf_tyenv ∧
   TE |- p prog_hastype ∧
   TE |- p prog_annotates_to p'
→  TE |- p' aprog_hastype
```

This property is proved by demonstrating that a similar property holds for all syntax classes from expressions through to class bodies. We also prove liveness, which is again three properties, the one for variables being:

```
   TE wf_tyenv ∧
   TE |- p aprog_hastype ∧
   TE |- heap wf_heap ∧
   (TE,heap) |- frames frames_conform_to F ∧
   (TE,F,heap) |- var0 rvar_hastype ty ∧
   ~var_ground var0 ∧
   s0 = (frames,heap)
→  ∃var1 s1. (var0,s0) var_reduce(TE,p) (var1,s1)
```

4 Validating the Model

We claim the specification we have developed so far is a correct formulation of the semantics of the Java subset we have in mind. But how do we know this, indeed how do we know our definitions are even logically consistent?

Because of the style of definition we have used (least fixed points and simple recursive definitions), consistency is essentially trivial. Validity is harder: we have to measure this against the Java language standard [GJS96] and our own understanding of the meaning of constructs in the subset.

We use two techniques to validate the specification:

1. Type checking of higher order logic;
2. Compiling to ML and running test cases.

Here we concentrate on the second. Essentially we compile 'manifestly executable' specifications to Objective CaML code, thus generating an interpreter for the language based directly on our definitions. The interpreter is able to typecheck and execute concrete Java$_S$ programs if given a concrete environment. The interpreter is not efficient, but is sufficient to test small programs.

An example is required. The $\sqsubseteq_{class}$ relation shown in Section 4 compiles to a ML function that is semantically equivalent to the following (we use CaML syntax):

```
let rec subclass_of CE C =
 (fun () -> if wf_class(C) then seq_cons(C,seq_nil) else fail()) seq_then
 (fun () -> match (PLOOKUP CE C) with
              NONE -> fail()
            | SOME(C',_,_,_) -> subclass_of CE C');
```

where `seq_nil`, `seq_cons` and the infix operator `seq_then` are the obvious operations on lazy lists, used to implement backtracking. Thus `subclass_of` will return a lazy list of identifiers and acts as a non-standard model of the relation defined by the inductive rules. Likewise we translate recursive functions to ML code, though no backtracking is needed here.

Of course, not all inductive relations or higher order logic terms are executable under this scheme. The executable subset is large and fairly straightforward to define, however only inductive relations that satisfy strict *mode constraints* are admitted at present. That is, arguments must be divisible into inputs and outputs, and inputs must always be defined by previous inputs or generated outputs. This concept is familiar from Prolog: the mode constraints for the $\sqsubseteq_{class}$ relation are `(+,+,-)`. We do not to translate directly to Prolog rules: this is clearly possible but unification is almost never required when expressing 'manifestly executable' rules, and indeed the elimination of all implicit unification steps is a typical way of proving the existence of an algorithm for the relations defined.

Thus, DECLARE produces a CaML module for each specification we have written. The modules are compiled together and linked against a module which implements core functionality. Test cases are expressed as higher order logic expressions, though better would be the ability to parse, compile and run Java programs directly from Java source code.

Approximately 40 errors were discovered by using these techniques. The breakdown of these was as follows:

- Around 30 typing mistakes which led to mode violations.
- Around 5 logical mistakes in the typing rules.
- Around 5 logical mistakes in the runtime rules.

From our experience with this technique, we would recommend that every system used for reasoning about programming language semantics include similar functionality.

5 The DECLARE Proof

We now outline the DECLARE proof of type soundness. The reader should keep in mind that when this proof was begun, the only guide available was the rough proof outline in [DE97], and this was based on a formulation of the problem that was subsequently found to contain errors. Thus the process is one of proof discovery rather than proof transcription!

The proof of type soundness proceeds by induction on the derivation of the typing judgment for the term t_0. We have one case for each rule in the mutually recursive inductive relations that define rterm expression, statement and variable typing judgments. The three mutually recursive goals are specified in DECLARE as follows (`var_types`, `exp_types` and `stmt_types` are macros for the induction invariants):

```
if  "TE wf_tyenv"                                      <auto>
    "TE |- p aprog_hastype"                            <p_types>
    "TE |- heap0 wf_heap"                              <heap0_conforms>
    "(TE,heap0) |- frames0 frames_conform_to FT0"      <frames0_conform>
    "s0 = (frames0,heap0)"                             <auto>
    "s1 = (frames1,heap1)"                             <auto>
then
  if "(var0,s0) var_reduce(TE,p) (var1,s1)"            <var0_reduces>
     "(TE,FT0,heap0) |- var0 rvar_hastype var0_ty"     <var0_welltyped>
  then "var_types (TE,FT0,heap0) var0 var0_ty (var1,s1)"
  or   "exceptional_var(var1)"                         <var1_exceptional>
and
  if "(exp0,s0) exp_reduce(TE,p) (exp1,s1)"             <exp0_reduces>
     "(TE,FT0,heap0) |- exp0 rexp_hastype exp0_ty"     <exp0_welltyped>
  then "exp_types (TE,FT0,heap0) exp0_ty (exp1,s1)"
  or   "exceptional_exp(exp1)"                          <exp1_exceptional>
and
  if "(stmt0,s0) stmt_reduce(TE,p) (stmt1,s1)"         <stmt0_reduces>
     "(TE,FT0,heap0) |- stmt0 rstmt_hastype"           <stmt0_welltyped>
  then "stmt_types (TE,FT0,heap0) (stmt1,s1)"
  or "exceptional_stmt(stmt1)"                          <stmt1_exceptional>
```

Note how we name facts, and can have multiple (disjunctive) goals. The name `<auto>` indicates a fact should be implicitly included in all future justifications for this branch of the proof. The induction step of the proof is specified by:

```
proceed by rule induction on
    <var0_welltyped>, <exp0_welltyped>, <stmt0_welltyped>
    with TE,heap0,frames0,FT0,s0,heap1,frames1,s1,p constant;
```

This step hides a great deal of complexity: DECLARE determines the correct induction theorem to use based on the given judgments, and computes the induction predicate based on the problem statement. The `with ... constant`

construct tells DECLARE that the given local constants do not 'vary during the induction', i.e. the induction hypotheses do not need to be general over these. DECLARE will warn the user if a case of the induction is skipped, and can present the cases remaining if asked to do so. The user is free to write the cases in any order.

The first case we consider is very easy: it is when var_0 is a local variable. Local variables are ground, so there are no reductions possible, and we get an immediate contradiction. The proof is:

```
case StackVar
  "var0 = RStackVar(frame,id)" <auto>;

  contradiction by rule cases on <var0_reduces>;
```

That is, by considering the different possibilities of how the fact `var0_reduces` is derived, we get an immediate contradiction. We have given a *claim* (in this case the simple claim that we can derive a contradiction, though in general we assert a complex set of facts, possibly introducing new variables) and a *justification* (`by ...`). The construct `rule cases` is a simple function that is an example of the way we specify hints that help the proof checker (other examples are simply quoting facts directly, or giving a fact with some explicit instantiations). DECLARE combines 'forward' and 'backward' proof: each step specifies something we have to prove, given our current context (in this case a contradiction). The automation uses a combination of proof techniques to prove the result, and the justification hints we give can involve little 'forward' proofs in their own right.

The next case we will consider is where $stmt_0$ assigns to an array. DECLARE informs us of the available inductive hypotheses, though we choose our own names for the new variables and facts:

```
case AssignToArray
   "lval0 = RAccess(arr0,idx0)"                          <auto>
   "stmt0 = RAssign(RAccess lval0,rexp0)"                <auto>
   "∀exp1. (rexp0,s0) exp_reduce(TE,p) (exp1,s1)
      → exp_types (TE,FT0,heap0) rexp0_ty (exp1,s1)
        | exceptional_exp exp1"                          <ihyp_for_rexp0>
   "(TE,FT0,heap0) |- rexp0 rexp_hastype rexp0_ty"       <rexp0_types_in_s0>
   "∀var1. (RAccess lval0,s0) var_reduce(TE,p) (var1,s1)
      → var_types (TE,FT0,heap0) (RAccess lval0) lval0_ty (var1,s1)
        | exceptional_var var1"                          <ihyp_for_lval0>
   "(TE,FT0,heap0) |- RAccess(arr0,idx0) rvar_hastype lval0_ty"
                                                         <lval0_types_in_s0>;
```

This case can be decomposed into three sub-cases as follows, because there are only three interesting reductions that can occur when our top term is an assignment:

```
cases by rule cases on <stmt0_reduces>,
             not <stmt1_exceptional>,
             <exceptional>;
   // the lvalue reduces
   case "(lval0,s0) var_reduce(TE,p) (lval1,s1)" <lval0_reduces>
        "stmt1 = RAssign(lval1,rexp0)" <auto>;
     ...
   // the rvalue reduces
   case "(rexp0,s0) exp_reduce(TE,p) (rexp1,s1)" <rexp0_reduces>
        "stmt1 = RAssign(lval0,rexp1)" <auto>;
     ...
   // both are ground, so the assignment happens
   case "arr0 = RValue(RAddr(SOME(taddr)))" <auto>
        "idx0 = RValue(RPrim(Int(k32)))" <auto>
        "rexp0 = RValue(val)" <auto>
        "heap0(taddr) = cell" <cell>
        "cell = SOME(ARRAY(arrty,vec))" <lookup>
        "idx = dest_int32(k32)" <auto>
        "idx >>= 0" <auto>
        "idx < LEN(vec)" <auto>
        "typecheck((TE,heap0),val,arrty)" <val_fits>
        "heap1 = heap0 <++ (taddr,ARRAY(arrty,REPL idx vec val))" <auto>
        "stmt1 = RExpr(RVoid)" <auto>
        "frames1 = frames0" <auto>;
```

Here we see DECLARE's third (and final!) proof language construct: decomposition into cases, perhaps introducing new constants in each case. Those used to tactic based theorem provers may find it difficult to believe that these three constructs are sufficient to express any proof, and even harder to believe that proofs end up simpler: this is discussed further in Appendix A.

The cases may look daunting, but consider how far we have come with this case-split: the justification for the split is based on the possibilities for how the $stmt_0$ could have reduced (`rule cases on <stmt0_reduces>`), on the fact that in the cases where an exception is produced the proof is trivial (`not <stmt1_exceptional>`), and on the definition of what it means for a value to be exceptional (`<exceptional>`). Thus we have eliminated all the cases in array assignment where exceptions arise (there are three), as well as the 30 cases where the reduction rules do not apply to array assignments (each of these require some proof).

Of the remaining cases, the first two correspond to redex rules for arrays, and their proofs use the induction hypotheses. The final case is the most interesting one: it is where both the left and right of the assignment are ground. The rest of the proof for this case is shown in Appendix D.

The above proof text was arrived at by repeatedly refining an approximate proof script, and also cut-and-pasting some reasoning steps from previously proved cases. This process was repeated for all 36 major cases of the type soundness proof. Typically, a first pass at formally checking a proof will result in roughly:

- 50% of the steps (i.e. logical leaps) in the proof being accepted immediately;
- 25% of the steps requiring the addition of one or two supporting facts, and perhaps some explicit instantiations;
- 25% of the cases requiring more thought than anticipated.

The success rate increases for cases that are very similar to previous ones. Machine checking the proofs up to the lemmas that were outlined in Section 3.2 took around 30 minutes to one hour per case: more for complex cases such as procedure call. The lemmas were initially assumed, and proofs given to them at a later stage. This sometimes involved refining the original proof script further, or adapting the model where it was found to be inadequate.

The reader should note that although a *very* powerful automated routine may be able to find the entire proof for one of these cases *after the fact*, the very process of writing the proof corrected significant errors in the rough draft that would have confounded even the best prover. Increased automation gives us a diminishing return as we venture into areas where the correct formulation takes care to find.

6 Errors Discovered

In this section we describe an error in the Java language specification that we independently rediscovered during the course of this work. We also describe one major error and a noteworthy omission in Drossopoulou and Eisenbach's original presentation of the type soundness proof.

6.1 An Error in the Java Language Specification

In the process of finishing the proofs of the lemmas described in Section 3.2 we independently rediscovered a significant flaw in the Java language specification that had recently been found by developers of a Java implementation [PB97]. In theory the flaw does not break type soundness, but the authors of the language specification have confirmed that the specification needs alteration.

The problem is this: in Java, all interfaces and arrays are considered subtypes of the type `Object`, in the sense that a cast from an interface or array type to `Object` is permitted. The type `Object` supports several "primitive" methods, such as `<object>.hashValue()` and `<object>.getClass()` (there are 11 in total). The question is whether expressions whose *static* type is an interface support these methods.

By rights, interfaces should indeed support the `Object` methods - any class that actually implements the interface will support these methods by virtue of being a subclass of `Object`, or an array. Indeed, the Sun JDK toolkit allows calling these methods from static interface types, as indicated by the successful compilation (but not execution) of the code:

```
public interface I { }
public class Itest {
  public static void main(String args[]) {
    I a[] = { null, null };
    a[0].hashCode();
    a[0].getClass();
    a[0].equals(a[1]);
  }
}
```

However, the existing language specification states explicitly that interfaces *only support those methods listed in the interface or its superinterfaces*, and that there is no 'implicit' superinterface (i.e. there is no corollary to the 'mother-of-all-classes' `Object` for interfaces. To quote:

> The members of an interface are all of the following:
> – Members declared from any direct superinterfaces
> – Members declared in the body of the interface.
> ...
> There is no analogue of the class `Object` for interfaces; that is, while every class is an extension of class `Object`, there is no single interface of which all interfaces are extensions.
>
> [GJS96], pages 87 and 185

The error was detected when trying to prove the existence of compatible methods and fields as we move from a type to a subtype, in particular from the type `Object` to an interface type.

6.2 Runtime Typechecking, Array Assignments, and Exceptions

In Drossopoulou and Eisenbach's original formulation the type soundness property was stated along the following lines (emphasis added):

Theorem 2. *If a well-typed term t is not ground, then it rewrites to some t' (and a new state s and environment Γ). Furthermore, either t'* **eventually rewrites** *to an exception, or t' has some narrower type than t, in the new state and environment.*

The iterated rewriting was an attempted fix for a problem demonstrated by the following program:

```
void silly(C arr[], C s) {
   arr[1] = s;
}
```

At runtime, `arr` may actually be an array of some narrower type, say `C'` where `C'` is a subclass of `C`. Then the array assignment appears to become badly typed

before the exception is detected, because during the rewriting the left side becomes a narrower type than the right. Thus they allow the exception to appear after a number of additional steps.

However, `arr` can become narrower, and then subsequently fail to terminate! Then an exception is never raised, e.g.

```
arr[loop()] = s;
```

The problem occurs in even simpler cases, e.g. when both `arr` and `s` have some narrower types `C'[]` and `C'`. Then, after the left side is evaluated, the array assignment appears badly typed, but will again be well typed after the right side is evaluated.

Fixing this problem requires a different understanding of the role of the types we assign to rterms. Types for intermediary rterms only exist to help express the type soundness invariant of the abstract machine, i.e. to define the allowable states that a well-typed execution can reach. In particular, the array assignment rule must be relaxed to allow what appear to be badly typed assignments, but which later get caught by the runtime typechecking mechanism.

This problem is an interesting case where the attempted re-use of typing rules in a different setting (i.e. the runtime setting rather than the typechecking setting) led to a subtle error, and one which we believe would only have been detected with the kind of detailed analysis that machine formalization demands. The mistake could not be missed in that setting!

6.3 Side Effects on Types

A significant omission in Drossopoulou and Eisenbach's original proof was as follows: when a term has two or more subterms, e.g. *arr*`[`*idx*`]` `:=` *e*, and *arr* makes a reduction to *arr*', then the types of *idx* and *e* may change (become narrower) due to side effects on the state. This possibility had not originally been considered by Drossopoulou and Eisenbach, and requires a proof that heap locations do not change type (our notion of heap conformity suffices). We also need lemmas stating that typing is monotonic with respect to the $\leq_{frame}$ and $\rightleftharpoons_{heap}$ relationship, up to the $\leq_{wdn}$ relationship. The foremost of these lemmas has been mentioned in Section 3.2. This problem was only discovered while doing detailed machine checking of the rough proof outline.

7 Summary

This chapter has presented corrections to the semantics of Java$_S$, a machine formalization of this semantics, a technique to partially validate the semantics, and an example of the use of new mechanized proof techniques to prove the type soundness property for that language.

The work demonstrates how formal techniques can be used to help specify a major language. Java itself is far more complicated than Java$_S$, but we have

still covered a non trivial subset. The formalization in Chapter 3 was the original inspiration for this work. We suggest that in the long run theorem prover specifications may provide a better format for the formalization, especially when flexible tools are provided to read, execute and reason about it.

The disciplined approach enforced when writing a proof to be accepted by a mechanized tool ensures errors like those described in Section 6 are detected. The declarative proof language played a very useful role: it allowed the author to think clearly about the language while preparing the proof outlines for the computer. The first error was found when simply preparing the proof outline, rather than when checking it in detail. During this process of preparation the question 'will a machine accept this proof?' was always in the back of the author's mind, and this ensured that unwarranted logical leaps were not made.

The independent rediscovery of the mistake in the Java language specification described in Section 6.1 indicates that such errors can indeed be discovered by the process of formal proof. However the mistake had already been discovered by implementors attempting to follow the language specification precisely.

It is commonly accepted that *formal specification* in a logic is of value when debugging specifications. This work has demonstrated that *proof sketching* and *proof checking* can also be of value, even while the theoretical framework for the language is still under development. It is interesting to note that of the three major errors, two were discovered at a late stage in the work.

7.1 Related Work

In the following chapter, Tobias Nipkow and David von Oheimb present their work on developing a proof of the type soundness property for a similar subset of Java in the Isabelle theorem prover. I am extremely grateful for the chance to meet with them and have adopted suggestions they have made. These two works are valuable 'modern' case studies of theorem proving methods applied this kind of problem. Isabelle is a mature system and has complementary strengths to DECLARE, notably strong generic automation and manifest soundness. A tool which unites these strengths with DECLARE's is an exciting prospect.

7.2 Future Work

The model presented in this article has scope to be extended in many directions. The treatment could be expanded to encompass features such as exceptions, constructors, access modifiers, static fields and static methods without major problems, although this would involve a significant expansion in the size of the proofs. Features such as threads and Java's semantically visible garbage collection pose greater problems, but should also be possible.

The work began as a case study for the application of a declarative proof language to operational reasoning, and there are ways in which DECLARE (or similar systems) could be improved based on this experience. The most necessary features are the implementation of decision procedures for ground equational

reasoning (as in PVS [ORR+96]) and a small amount of 'Computer Aided Proof Writing', as described briefly in [Sym97].

Acknowledgments

I would like to thank Sophia Drossopoulou and Sarfraz Khurshid for an excellent day at Imperial spent discussing this work and its possible relevance to their project. I also thank David von Oheimb, Tobias Nipkow, Mark Staples, Michael Norrish, Mike Gordon, John Rushby and Natarajan Shankar and Frank Pfennig for the useful discussions I have had with them concerning DECLARE and this work. I especially thank John Harrison, whose system HOL-lite has contributed greatly to the development of DECLARE, including some source code, and whose work has provided a wealth of stimulating insights regarding higher order logic theorem proving.

References

[CM92] Juanito Camilleri and Tom Melham. Reasoning with inductively defined relations in the HOL theorem prover. Technical Report 265, University of Cambridge Computer Laboratory, Cambridge, CB2 3QG, U.K., August 1992.

[DE97] Sophia Drossopoulou and Susan Eisenbach. Is the Java type system sound? (version 2.01). Technical report, Imperial College, University of London, Cambridge, CB2 3QG, U.K., January 1997. This version was distributed on the Internet. Please contact the authors if a copy is required for reference.

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

[GM93] M.J.C Gordon and T.F Melham. *Introduction to HOL: A Theorem Proving Assistant for Higher Order Logic*. Cambridge University Press, 1993.

[Har96] John Harrison. HOL light: A tutorial introduction. In Mandayam Srivas and Albert Camilleri, editors, *Proceedings of the First International Conference on Formal Methods in Computer-Aided Design (FMCAD'96)*, volume 1166 of *Lecture Notes in Computer Science*, pages 265–269. Springer-Verlag, 1996.

[Har97] John R. Harrison. Proof Style. Technical Report 410, University of Cambridge Computer Laboratory, Cambridge, CB2 3QG, U.K., January 1997.

[Lov68] D. W. Loveland. Mechanical Theorem Proving by Model Elimination. *Journal of the ACM*, 15:236–251, 1968.

[ORR+96] S. Owre, S. Rajan, J.M. Rushby, N. Shankar, and M.K. Srivas. PVS: Combining specification, proof checking and model checking. In Rajeev Alur and Thomas A. Henzinger, editors, *Computer-Aided Verification, CAV '96*, volume 1102, pages 411–414. Springer-Verlag, July/August 1996.

[Pau90] L. C. Paulson. Isabelle: The next 700 theorem provers. In P. Odifreddi, editor, *Logic and Computer Science*, pages 361–385. Academic Press, 1990.

[Pau94] Lawrence C. Paulson. A Fixed Point Approach to Implementing (Co)inductive Definitions. *18th International Conference on Automated Deduction*, pages 148–161, 1994.

[PB97] Roly Perera and Peter Bertelsen. The Unofficial Java Bug Report, June 1997. Published on the WWW at http://www2.vo.lu/homepages/gmid/java.htm.

[Plo91] Gordon D. Plotkin. A structural approach to operational semantics. Technical report, Computer Science Department, Aarhus University, DK-8000 Aarhus C. Denmark, September 1991.

[Qia97] Zhenyu Qian. A Formal Specification of Java Virtual Machine Instructions. Technical report, Universität Bremen, FB3 Informatik, D-28334 Bremen, Germany, November 1997.

[Rud92] P. Rudnicki. An Overview of the MIZAR Project, 1992. Unpublished; available by anonymous FTP from `menaik.cs.ualberta.ca` as `pub/Mizar/Mizar_Over.tar.Z`.

[Sym97] Don Syme. DECLARE: A prototype declarative proof system for higher order logic. Technical Report 416, University of Cambridge Computer Laboratory, Cambridge, CB2 3QG, U.K., March 1997.

A A Brief Introduction to DECLARE

DECLARE is a proof checker for simple, polymorphic, higher order logic. It is designed to aid in the production of clear, readable, maintainable specification and proof documents. DECLARE is not a complete or polished system, and has been developed with the aim of testing various features that could be incorporated in existing, supported theorem provers such as HOL, Isabelle and PVS. It has been influenced heavily by Mizar, HOL, HOL-lite, Isabelle and PVS [Rud92,ORR+96,GM93,Har96,Pau90]. It is *not* an LCF-style system: deductions are not reduced to a primitive logical framework, though in principle we are confident this is possible. The features of interest here are:

- The declarative language used to express proof outlines.
- The support for modularization, separate processing and top-down formalization, which leads to a well-structured, efficient working environment.
- The automated proof support.

A.1 The Proof Language

We try to achieve, by the simplest means possible, results that are both *machine checkable* and *human readable*. [8]

DECLARE's proof language was originally inspired by Mizar and work by Harrison [Har97], but has been considerably streamlined. The language was demonstrated by example in Section 5. and uses just three main constructs:

- Induction;
- Case-decomposition; and
- Justifications.

[8] Some researchers take the view that human readable proofs should be generated as output from mechanized proofs: this may be possible, but it is a highly complex process and the results are not yet convincing. Our approach is to make the input readable in the first place.

Several other constructs are degenerate forms of these constructs: e.g. asserting new facts, perhaps involving new local constants, is a degenerate form of case-decomposition where there is only one case. Similarly introducing an abbreviation is a degenerate form of asserting new facts, where there is one new fact and one new local constant.

Those used to tactic based theorem provers may find it difficult to believe that these three constructs are sufficient to express any proof, and even harder to believe that proofs end up simpler. It is clear that any higher order logic proof can indeed be expressed: we simply have to implement the basic proof rules of the logic within the default proof obligation checker. The key reasons why proofs end up *simpler* with DECLARE are:

- (a) It provides excellent support for specifying complex reasoning deep within a logical context;
- (b) Case splits may be based on a complex argument, rather than some simple syntactic criteria (as is usually the case in a tactic based theorem prover). Many trivial cases disappear without thought.
- (c) The proof style encourages extensive use of abbreviations, as in written proofs, and gives easy control over variable and fact naming. A common accusation levied at declarative proof languages is that in large verifications terms get too large to be written out by hand. However, we would claim the exact opposite: *in large interactive proofs, terms get so complex it is essential that a human be in charge of keeping the complexity under control.* This can be done through definitions, abbreviations and other conveniences both logical and notational. These are essential to the production of an elegant, clean and maintainable proof.

Of course, none of DECLARE's constructs are *incompatible* with tactics, but our experience indicates that adding more traditional tactic constructs into the proof language does not gain much, and has the potential to destroy many of the useful properties the language enjoys.

Such proof languages are called *declarative*, to place them in contrast to 'procedural' (often tactic based) mechanisms for specifying proofs. The main feature of a declarative language is that the machine works out the vast majority of the syntactic manipulations necessary to achieve a proof (especially those associated with propositional connectives, first-order quantifiers and associative/commutative operators), leaving the user free to simply declare a semantic intent. The use of a declarative proof language has clear advantages:

- Declarative proofs are more readable than tactic proofs.
- Proof interpretation always terminates, unlike tactic proofs which are expressed in a Turing-complete language. In particular guaranteed termination makes error recovery in proof checking more tractable.
- Declarative proofs are potentially more maintainable under changes to the specification and the prover.
- Declarative proofs are potentially more portable. Specification and proof documents developed with DECLARE are, in principle, portable to other

proof systems and may even be interpreted in other sufficiently powerful logics.
- A declarative style appeals to a wider class of users, helping to deliver automated reasoning and formal methods to mathematicians and others.

A.2 The Working Environment

When using DECLARE, large bodies of work are broken into a series of *articles*, each of which may have an interface called an *abstract*. Articles are checked relative to the abstracts they import, and must 'implement' the abstract they export. Abstracts may be pre-compiled, which, in combination with the `make` system, gives us a simple, yet light-weight and effective means for maintaining the coherence of large collections of specifications and proofs. This approach also means DECLARE typically uses only 5-6 MB of memory when executing.

A.3 Automation

DECLARE proofs are only proof outlines, and require automation to fill in the gaps in the argument. In this way the proof language acts as a bridge between the human and the automated capabilities of the proof checker.

The automation we use in this chapter is fairly straightforward:

- We use Boyer-Moore/Isabelle style simplification with conditional, higher-order rewriting to normalize expressions. Simplification is performed under the auspices of a two-sided sequent calculus like that used by PVS. During simplification we:
  - Apply safe introduction and elimination rules, e.g. choosing witnesses for existentials in assumptions; splitting disjuncts in goal formulae; and transferring negated formulae across the sequent.
  - Apply 'unwinding' rules to eliminate local constants from existential and universal formulae, including the sequent itself.
  - 'Untuple' all pair, tuple and record values.
  - Apply a large background set of (conditional) rewrites collected from imported abstracts;
  - Normalize arithmetic expressions;
  - Case-split on constructs such as conditionals and quantified structural variables (booleans, options etc.).
  - Use exploratory unwinding of some definitions, in the style of PVS.
  - Use controlled left-right simplification of certain 'program-like' constructs, which helps implement partial evaluation and avoids common causes of non-terminating rewriting strategies.
- After simplification we use a simple model elimination [Lov68] prover (with time, variable and depth limited iterative deepening) to search for values for unknowns.

This level of automation has been sufficient during exploratory proof development, since in this most important stage we are content with guiding the prover through the proof without expecting complex steps, such as inductions, to be automated. The only significant problems arise when we venture into problem spaces that requires significant equality *and* proof-search reasoning (this is still a major research area), or equality reasoning not amenable to rewriting (adding congruence closure will solve this).

Automation in DECLARE is guided by 'pragmas': lemmas are given once-only 'how to use me' declarations, and no weightings or other obscure hints are specified when a lemma is used. This helps ensure that proof documents are not overly reliant on quirks of the underlying prover, and are robust as the prover itself changes.

B The Full Widening Rules

These rules determine the widening (subtype) relation.

$$\frac{\Gamma \vdash C \sqsubseteq_{class} C'}{\Gamma \vdash C \leq_{sref} C'} \qquad \frac{\Gamma \vdash I \sqsubseteq_{int} I'}{\Gamma \vdash I \leq_{sref} I'} \qquad \frac{I \in \Gamma}{\Gamma \vdash I \leq_{sref} \texttt{Object}} \qquad \frac{\begin{array}{l}\Gamma \vdash C \sqsubseteq_{class} C' \\ \Gamma \vdash C' :_{imp} I \\ \Gamma \vdash I \sqsubseteq_{int} I'\end{array}}{\Gamma \vdash C \leq_{sref} I'}$$

$$\frac{ty \in \textit{prim-types}}{\Gamma \vdash ty \leq_{comp} ty} \qquad \frac{ty, ty' \in \textit{simple-ref-types} \quad \Gamma \vdash ty \leq_{sref} ty'}{\Gamma \vdash ty \leq_{comp} ty'}$$

$$\frac{ty \in \textit{component-types} \quad n > 0}{\Gamma \vdash ty\texttt{[]}^n \leq_{arr} \texttt{Object}} \qquad \frac{n > 0 \quad \Gamma \vdash ty \leq_{comp} ty'}{\Gamma \vdash ty\texttt{[]}^n \leq_{arr} ty'\texttt{[]}^n}$$

$$\frac{ty, ty' \in \textit{array-types} \quad \Gamma \vdash ty \leq_{arr} ty'}{\Gamma \vdash ty \leq_{ref} ty'} \qquad \frac{ty, ty' \in \textit{simple-ref-types} \quad \Gamma \vdash ty \leq_{sref} ty'}{\Gamma \vdash ty \leq_{ref} ty'} \qquad \frac{ty \in \textit{ref-types}}{\Gamma \vdash \texttt{nullT} \leq_{ref} ty}$$

$$\frac{ty \in \textit{prim-types}}{\Gamma \vdash ty \leq_{wdn} ty} \qquad \frac{ty, ty' \in \textit{ref-types} \quad \Gamma \vdash ty \leq_{ref} ty'}{\Gamma \vdash ty \leq_{wdn} ty'}$$

C The Full Traversal Rules

These rules determine what methods and fields are visible from a given class. The relations evaluate graphs, which in well-formed environments determine partial functions.

$$\frac{\Gamma(C).\text{flds}(\textit{field}) = ty}{\Gamma \vdash (C, ty) \in \mathsf{FDec}(C, \textit{field})}$$

$$\frac{\Gamma(C) = \langle C_{sup}, \mathit{flds}, \ldots\rangle \quad \mathit{flds}(\mathit{field}) = \bot \quad \Gamma \vdash (C', \mathit{ty}) \in \mathsf{FDec}(C_{sup}, \mathit{field})}{\Gamma \vdash (C', \mathit{ty}) \in \mathsf{FDec}(C, \mathit{field})}$$

$$\frac{\Gamma(C).\mathrm{flds}(\mathit{field}) = \mathit{ty}}{\Gamma \vdash ((C, \mathit{field}), \mathit{ty}) \in \mathsf{FDecs}(C)} \quad \frac{\Gamma(C) = \langle C_{sup}, \ldots\rangle \quad \Gamma \vdash ((C', \mathit{field}), \mathit{ty}) \in \mathsf{FDecs}(C_{sup})}{\Gamma \vdash ((C', \mathit{field}), \mathit{ty}) \in \mathsf{FDecs}(C)}$$

$$\frac{\Gamma(C).\mathrm{meths}(\mathit{meth}, \mathit{at}) = \mathit{rt}}{\Gamma \vdash ((\mathit{meth}, \mathit{at}), \mathit{rt}) \in \mathsf{MSigs}_C(C)}$$

$$\frac{\Gamma(C) = \langle C_{sup}, \mathit{meths}\rangle \quad \mathit{meths}(\mathit{meth}, \mathit{at}) = \bot \quad \Gamma \vdash ((\mathit{meth}, \mathit{at}), \mathit{rt}) \in \mathsf{MSigs}_C(C_{sup})}{\Gamma \vdash ((\mathit{meth}, \mathit{at}), \mathit{rt}) \in \mathsf{MSigs}_C(C)}$$

D The Proof for Ground Array Assignments

```
case AssignToArray
  "lval0 = RAccess(arr0,idx0)"                    <auto>
  "stmt0 = RAssign(RAccess lval0,rexp0)"          <auto>
  "∀exp1. (rexp0,s0) exp_reduce(TE,p) (exp1,s1)
     → exp_types (TE,FT0,heap0) rexp0_ty (exp1,s1)
       | exceptional_exp exp1"                    <ihyp_for_rexp0>
  "(TE,FT0,heap0) |- rexp0 rexp_hastype rexp0_ty"
                                                  <rexp0_types_in_s0>
  "∀var1. (RAccess lval0,s0) var_reduce(TE,p) (var1,s1)
     → var_types (TE,FT0,heap0) (RAccess lval0) lval0_ty (var1,s1)
       | exceptional_var var1"                    <ihyp_for_lval0>
  "(TE,FT0,heap0) |- RAccess(arr0,idx0) rvar_hastype lval0_ty"
                                                   <lval0_types_in_s0>;
 cases by rule cases on <stmt0_reduces>,
              not <stmt1_exceptional>,
              <exceptional>;
    // Case 1: the lvalue reduces
   case "(lval0,s0) var_reduce(TE,p) (lval1,s1)" <lval0_reduces>
        "stmt1 = RAssign(lval1,rexp0)"           <auto>;
     ...
    // Case 2: the rvalue reduces
   case "(rexp0,s0) exp_reduce(TE,p) (rexp1,s1)" <rexp0_reduces>
        "stmt1 = RAssign(lval0,rexp1)"           <auto>;
     ...
    // Case 3: both are ground, so the assignment happens
   case "arr0 = RValue(RAddr(SOME(taddr)))"      <auto>
        "idx0 = RValue(RPrim(Int(k32)))"         <auto>
        "rexp0 = RValue(val)"                    <auto>
        "heap0(taddr) = cell"                    <cell>
        "cell = SOME(ARRAY(arrty,vec))"          <lookup>
        "idx = dest_int32(k32)"                  <auto>
        "idx >>= 0"                              <auto>
        "idx < LEN(vec)"                         <auto>
        "typecheck((TE,heap0),val,arrty)"        <val_fits>
```

```
    "heap1 = heap0 <++ (taddr,ARRAY(arrty,REPL idx vec val))" <auto>
    "stmt1 = RExpr(RVoid)"                    <auto>
    "frames1 = frames0"                       <auto>;

 // Because the lvalue is well-typed, its constituents must
 // also be well-typed.  We need these facts to derive interesting
 // things about the content of the array we are assigning into.
consider simpty,dim,ndim st
 "(TE,FT0,heap0) |- arr0 rexp_hastype
                    SOME(VT(simpty,dim))"       <arr0_types>
 "0N < dim"                                     <auto>
 "(TE,FT0,heap0) |- idx0 rexp_hastype
                    SOME(VT(PrimT(intT),0N))"   <idx0_types>
 "ndim = dim-1N"                                <auto>
 "lval0_ty = VT(simpty,ndim)"                   <auto>
        by rule cases on <lval0_types_in_s0>;

 // The type of the target address
 // correlates with that of the vector.
have "(TE,heap0) |- RAddr(SOME(taddr)) rval_hastype
                    VT(simpty,dim)"             <taddr_types>
     by rule cases on <arr0_types>;

 // The stuff stored at the target address is an array...
consider dim',vec' st
 "dim = dim'+1N"                                 <auto>
 "cell = SOME(ARRAY(VT(simpty,dim'),vec'))"      <auto>
      by rule cases on <taddr_types>,<cell>;

 // And the array that's stored there looks exactly like we expect...
then have "arrty = VT(simpty,dim')"              <auto>
          "vec = vec'" <auto> by <lookup>;

 // Now the rhs: it's ground so it's really a value...
consider rexp0_vty st
   "rexp0_ty = SOME(rexp0_vty)"                  <auto>
   "(TE,heap0) |- val rval_hastype rexp0_vty"    <val_types>
     by rule cases on <rexp0_types_in_s0>;

 // Now we have everything we need to invoke our lemma that
 // assigning into an array maintains the necessary heap properties.
have "TE |- heap1 wf_heap"                  <heap1_wf>
     "TE |- heap0 heap_conforms_to heap1"  <heap1_conforms>
  by <array-assign-conforms-lemma> ["TE","heap1","heap0"],
     <heap0_conforms>,<val_fits>,<val_types>,<lookup>,<cell>;

 // And similarly for the frames: this is easy because they
 // don't change, and we just invoke the property that
 // frames_conforms_to is monotonic under heap_conforms_to.
have "(TE,heap1) |- frames0 frames_conform_to
```

```
                        FT0" <frames0_conform_in_s1>
          by <frames_conform-mono-lemma>, <frames0_conform>,
             <heap1_larger>,<heap0_wf>,<heap1_conforms>;

     // Finally we can derive the typing judgment for stmt1...
    have "(TE,FT0,heap1) |- stmt1 rstmt_hastype"
          by <rstatics__Expr> ["NONE"],<rstatics__Void>;

     // And we have everything we need to show the induction invariant
     // still holds.
    then qed by <heap1_wf>, <frames0_conform_in_s1>,
                <heap1_conforms>,<stmt_types>;
end
```

Machine-Checking the Java Specification: Proving Type-Safety*

David von Oheimb** and Tobias Nipkow

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

Abstract. In this article we present Bali, the formalization of a large (hitherto sequential) sublanguage of Java. We give its abstract syntax, type system, well-formedness conditions, and an operational evaluation semantics. Based on these definitions, we can express soundness of the type system, an important design goal claimed to be reached by the designers of Java, and prove that Bali is indeed type-safe.
All definitions and proofs have been done formally in the theorem prover Isabelle/HOL. Thus this article demonstrates that machine-checking the design of non-trivial programming languages has become a reality.

1 Introduction

Bali is a large subset of Java [GJS96]. This article presents its formalization and the proof of a key property, namely the soundness of its type system — specified and verified in the theorem prover Isabelle/HOL [Pau94].

On the face of it, this article is mostly about Bali, its abstract syntax, type system, well-formedness conditions, and operational semantics, formalized as a hierarchy of Isabelle theories, and the structure of the machine-checked proof of type soundness and its implications. Although these technicalities do indeed take up much of the space, there is a meta-theme running through the article, which we consider even more important: the technology for producing machine-checked programming language designs has arrived.

We emphasize that by 'machine-checked' we do not just mean that it has passed some type checker, but that some non-trivial properties of the language have been established with the help of a (semi-automatic) theorem prover. The latter process is still not a piece of cake, but it has become more than just feasible. Therefore any programming language intended for serious applications should strive for such a machine-checked design. The benefits are not just greater reliability, but also greater maintainability because the theorem prover keeps track of the impact that changes have on already established properties.

* This is a completely revised and extended version of [NO98].
** Research supported by DFG SPP *Deduktion*.

Jim Alves-Foss (Ed.): Formal Syntax and Semantics of Java, LNCS 1523, pp. 119–156, 1999.

Note that the type-safety of Java is not sufficient to guarantee secure execution of bytecode programs on the Java Virtual Machine, because the bytecode might be tampered with, produced by a faulty compiler, or not be related to any Java source program at all. This was the main reason for introducing the Bytecode Verifier in the JVM, which checks the integrity, in particular type-correctness, of any bytecode before execution. Of course similar security problems arise for any other high-level languages as well. Nevertheless, the investigation of type-safety at source level is worthwhile: it checks whether the language design is sound, which means that at least all the necessary conditions expressible at that level are fulfilled. In particular static typing loses much of its *raison d'être* if the language is not type-safe.

1.1 Related Work

The history of type soundness proofs goes back to the subject reduction theorem for typed λ-calculus but starts in earnest with Milner's slogan 'Well-typed expressions do not go wrong' [Mil78] in the context of ML. Milner uses a denotational semantics, in contrast to most of the later work, including ours. The question of type-safety came to prominence with the discovery of its failure in Eiffel [Coo89]. Ever since, many designers of programming languages (especially OO ones) have been at pains to prove type-safety of their languages (see, for example, the series of papers by Bruce et al. [Bru93,BCM+93,BvGS95]).

Directly related to our work is that by Drossopoulou and Eisenbach [DE98] who prove (on paper) type-safety of a subset of Java very similar to Bali. Although we were familiar with an earlier version [DE97] of their work and have certainly profited from it, our work is not a formalization of theirs in Isabelle/HOL but differs in many respects from it, for example in the representation of programs and the use of an evaluation (aka "big-step") semantics instead of a transition (aka "small-step") semantics. Simultaneously with our work, Syme [Sym98] formalized the paper [DE97] as far as possible, uncovering two significant mistakes, both in connection with the use of transition semantics. Syme uses his own theorem prover DECLARE, also based on higher-order logic.

There are two other efforts to formalize aspects of Java in a theorem prover. Dean [Dea97] studies the interaction of static typing with dynamic linking. His simple PVS specification addresses only the linking aspect and requires a formalization of Java (such as our work provides) to turn his lemmas about linking into theorems about the type-safety of dynamically linked programs. Cohen [Coh97] has formalized the semantics of large parts of the Java Virtual Machine, essentially by writing an interpreter in Common Lisp. He used ACL2, the latest version of the Boyer-Moore theorem prover [BM88]. No proofs have been reported yet.

2 Overview

Bali includes the features of Java that we believe to be important for an investigation of the semantics of a practical imperative object-oriented language:

- class and interface declarations with instance fields and methods,
- subinterface, subclass, and implementation relations with inheritance, overriding, and hiding,
- method calls with static overloading and dynamic binding,
- some primitive types, objects (including arrays),
- exception throwing and handling.

This portion of Java is very similar to that covered by [DE98] and [Sym98].

We do not consider Java packages and separate compilation. For the moment, we also leave out several features of Java like class variables and static methods, constructors and finalizers, the visibility of names, and concurrency, but we aim to include at least part of them in later stages of our project. Some constructs are simplified without limiting the expressiveness of the language (see §4.1).

In developing the formalization of Bali and investigating its properties, we aim to meet the following design goals:

- faithfulness to the official language specification,
- succinctness and simplicity,
- maintainability and extendibility,
- adequacy for the theorem prover.

It might be interesting to keep these goals in mind while reading §4 on the formalization of Bali and §5 on our proofs and check how far we have reached them. We comment on our experience in pursuing these goals in §6.

3 The Basics of Isabelle/HOL

Before we present the formalization of Bali, we briefly introduce the underlying theorem proving system Isabelle/HOL.

Isabelle/HOL is the instantiation of the generic interactive theorem prover Isabelle [Pau94] with Church's version of Higher-Order Logic and is very close to Gordon's HOL system [GM93]. In this article HOL is short for Isabelle/HOL.

The appearance of formulas is standard, e.g. '$\longrightarrow$' is the (right-associative) infix implication symbol. Predicates are functions with Boolean result. Function application is written in curried style. For descriptions we apply Hilbert's choice operator ε, where $\varepsilon x.\ P\ x$ denotes some value x satisfying P, or an arbitrary value if no such x exists.

Logical constants are declared by giving their name and type, separated by '::'. Primitive recursive function definitions are written as usual. Non-recursive definitions are written with '$\stackrel{\mathsf{def}}{=}$'.

Types follow the syntax of ML, except that the function arrow is '$\Rightarrow$'. Type abbreviations are introduced simply as equations. A free datatype is defined by listing its constructors together with their argument types, separated by '|'.

There are the basic types *bool* and *int*, as well as the polymorphic type $(\alpha)set$ of homogeneous sets for any type α. Occasionally we apply the infix 'image' operator lifting a function over a set, defined as $f``S \stackrel{\mathsf{def}}{=} \{y.\ \exists x{\in}S.\ y = f\,x\}$.

The product type $\alpha \times \beta$ comes with the projection functions fst and snd. Tuples are pairs nested to the right, e.g. $(a,b,c) = (a,(b,c))$.

As the list type $(\alpha)list$ is defined via its constructors [] denoting the empty list and the infix 'cons' operator '#', it can be introduced by the datatype declaration

$(\alpha)list = [] \mid \alpha\#(\alpha)list$

The concatenation operator on lists is written as the infix symbol '@'. There is a functional map :: $(\alpha \Rightarrow \beta) \Rightarrow (\alpha)list \Rightarrow (\beta)list$ applying a function to all elements of a list, as well as a conversion function set :: $(\alpha)list \Rightarrow (\alpha)set$.

We frequently use the datatype

$(\alpha)option = \mathsf{None} \mid \mathsf{Some}\ \alpha$

It has an unpacking function the :: $(\alpha)option \Rightarrow \alpha$ such that the (Some x) $= x$ and the None $=$ arbitrary, where arbitrary is an unknown value defined as $\varepsilon x.\,\mathsf{False}$. There is a simple function mapping o2s :: $(\alpha)option \Rightarrow (\alpha)set$ converting an optional value to a set, with the characteristic equations o2s (Some x) $= \{x\}$ and o2s None $= \{\}$.

Most of the HOL text shown in this article is directly taken from the input files. However, it has been massaged by hand to hide Isabelle idiosyncrasies, increase readability, and adapt the layout. Minor typos may have been introduced in the process.

We adopt the following typographic conventions: Java keywords like `catch` appear in typewriter font, the names of logical constants like cfield appear in sans serif, while type names like *state* and meta-variables like v appear in italics.

4 The Formalization of Bali

This section presents all aspects of our formalization of Bali[1].

As far as Bali is a subset of Java, it strictly adheres to the Java language specification [GJS96], with several generalizations:

- we allow the result type of a method overriding another method to widen to the result type of the other method instead of requiring it to be identical.
- if a class or an interface inherits more than one method with the same signature, the methods need not have identical return types.
- no check of result types in dynamic method lookup.
- the type of an assignment is determined by the right-hand side, which can be more specific than the left-hand side.

[1] The Isabelle sources are available from the Bali project page `http://www.in.tum.de/~isabelle/bali/`

We found several issues concerning exceptions not specified in [GJS96] and therefore define a reasonable behavior that seems to be consistent with current implementations:

- given a Null reference, the `throw` statement raises a `NullPointer` exception.
- each system exception thrown yields a fresh exception object.
- if there is not enough memory even to allocate an `OutOfMemory` error, program execution simply halts. (Our experiments showed erratic behavior of some implementation in this case, ranging from sudden termination without executing `finally` blocks, over hangup, to repeated invocation of a single exception handler!)

To illustrate our approach, we use the following (artificial) example.

```
class Base {
  boolean vee;
  Base foo(Base x) {
    return x;
  }
}

class Ext extends Base{
  int vee;
  Ext foo(Base x) {
    ((Ext)x).vee=1;
    return null;
  }
}

Base e;
e=new Ext();
try {e.foo(null); }
catch (NullPointerException x) {throw x;}
```

This program fragment consists of two simple but complete class declarations and a block of statements that might occur in any method that has access to these declarations. It contains the following features of Bali:

- class declarations with inheritance, hiding of fields, and overriding of methods (with refined result type),
- return expressions, parameter access,
- sequential composition, expression statements, field assignment, type cast, local accesses, literal values, exception propagation,
- local assignment, instance creation,
- try & catch statement, method call (with dynamic binding), throw statement

4.1 Abstract Syntax

First, we describe how we represent the syntax of Bali and which abstractions we have introduced thereby.

Programs. A Bali program is a pair of lists of interface and class declarations:

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

Throughout the article, the symbol 'Γ' denotes a Bali program, as we use programs as part of the static type context defined in §4.2.

Each declaration is a pair of a name and the defined entity. Some names, like those of predefined classes (including those of system exceptions *xname*), have a predefined meaning and are therefore handled extra. We do not specify the structure of names further, but use the opaque HOL types *tname0*, *mname*, and *ename0* for user-defined type names, method names, and "expression names" (i.e. field and variable identifiers).

$xname =$ `Throwable`
$\quad \mid$ `NullPointer` $\mid$ `OutOfMemory` $\mid$ `ClassCast`
$\quad \mid$ `NegArrSize` $\mid$ `IndOutBound` $\mid$ `ArrStore`

$tname =$ `Object`	name of the top of the class hierarchy
$\quad \mid$ `SXcpt` *xname*	name of a system exception
$\quad \mid$ `TName` *tname0*	other class or interface name

$ename =$ `this`	special name for `this` pointer
$\quad \mid$ `EName` *ename0*	expression name

An interface (*iface*) contains lists of superinterface names and method declarations. A *class* specifies the names of an optional superclass and implemented interfaces, and lists of field and method declarations.

$iface = (tname)list \times (sig \times mhead)list$
$idecl = tname \times iface$
$class = (tname)option \times (tname)list \times (fdecl)list \times (mdecl)list$
$cdecl = tname \times class$

A field declaration (*fdecl*) simply gives the field type (*ty*, see §4.2). A method declaration ($sig \times mhead$ for interfaces or *mdecl* for classes) consists of a "signature" [GJS96, 8.4.2] (i.e. the method name and the list of parameter types, excluding the result type) followed by *mhead*, consisting of the list of parameter names and the result type, and (if it appears within a class) the method body (*mbody*). The latter consists of the list of local variables, a statement *stmt* as body, and a return expression *expr* (see below). As in [DE98], the separate return expression saves us from dealing with return statements occurring in arbitrary positions within the method body. Such statements may be replaced by

assignments to a suitable result variable followed by a control transfer to the end of the method body, using the result variable as return expression. We provide a dummy result type and value for `void` methods. For simplicity, up to now each method has exactly one parameter; multiple parameters can be simulated by a single parameter object with multiple fields.

field	$= ty$	field type
fdecl	$= ename \times field$	
sig	$= mname \times ty$	method name and parameter type
mhead	$= ename \times ty$	parameter name and result type
lvar	$= ename \times ty$	local variable name and type
mbody	$= (lvar)list \times stmt \times expr$	local vars, block, and return expression
methd	$= mhead \times mbody$	method (of a class)
mdecl	$= sig \times methd$	

In the abstract syntax given above, the formalization of our example program fragment looks like this:

```
BaseC ≝ (Base, (Some Object,
                []),
                [(vee, PrimT boolean)],
                [((foo,Class Base),(x,Class Base), ([],Skip,x))]))
ExtC  ≝ (Ext,  (Some Base,
                [],
                [(vee, PrimT int)],
                [((foo,Class Base),(x,Class Ext), ([],
                   Expr({ClassT Ext}(Class Ext)x.vee:=Lit (Intg 1)),
                   Lit Null))]))
classes ≝ [ObjectC,
           SXcptC Throwable,
           SXcptC NullPointer, SXcptC OutOfMemory, SXcptC ClassCast,
           SXcptC NegArrSize, SXcptC IndOutBound, SXcptC ArrStore,
           BaseC, ExtC]
tprg ≝ ([],classes)
test ≝ Expr(e:=new Ext);
       try Expr(e.foo({Class Base}Lit Null))
       catch((SXcpt NullPointer) x) (throw x)
```

where *Base* stands for TName *Base_*, *Ext* for TName *Ext_*, and similarly for *vee*, *x*, and *e*. The constants *Base_*, *Ext_*, etc. are all distinct. The sequence of statements *test* could have been embedded in *tprg*, which we have left out for simplicity.

Representation of Lookup Tables. The representation of declarations as lists gives an implicit finiteness constraint, which turns out to be necessary for the well-foundedness of the subclass and subinterface relation. The list representation also enables an explicit check whether the declared entities are named uniquely, implemented with the function unique given below. This function does not check for duplicate definitions, which is harmless.

$$\mathsf{unique} :: (\alpha \times \beta)\mathit{list} \Rightarrow \mathit{bool}$$
$$\mathsf{unique}\ t \stackrel{\mathsf{def}}{=} \forall (x_1,y_1) \in \mathsf{set}\ t.\ \forall (x_2,y_2) \in \mathsf{set}\ t.\ x_1 = x_2 \longrightarrow y_1 = y_2$$

For the lookup of declared entities, we transform declaration lists into abstract tables. They are realized in HOL as "partial" functions mapping names to values:

$$(\alpha,\beta)\mathit{table} \;=\; \alpha \Rightarrow (\beta)\mathit{option}$$

The empty table, pointwise update, extension of one table by another, the function converting a declaration list into a table, and an auxiliary predicate relating entries of two tables, are defined easily:

$$\begin{array}{ll}
\mathsf{empty} & :: (\alpha,\beta)\mathit{table} \\
_[_\mapsto_] & :: (\alpha,\beta)\mathit{table} \Rightarrow \alpha \Rightarrow \beta \Rightarrow (\alpha,\beta)\mathit{table} \\
_ \oplus _ & :: (\alpha,\beta)\mathit{table} \Rightarrow (\alpha,\beta)\mathit{table} \Rightarrow (\alpha,\beta)\mathit{table} \\
\mathsf{table_of} & :: (\alpha\times\beta)\mathit{list} \Rightarrow (\alpha,\beta)\mathit{table} \\
_\ \mathsf{hiding}\ _ & \\
\quad \mathsf{entails}\ _ & :: (\alpha,\beta)\mathit{table} \Rightarrow (\alpha,\gamma)\mathit{table} \Rightarrow (\beta \Rightarrow \gamma \Rightarrow \mathit{bool}) \Rightarrow \mathit{bool}
\end{array}$$

$$\begin{array}{ll}
\mathsf{empty} & \stackrel{\mathsf{def}}{=} \lambda k.\ \mathsf{None} \\
t[x\mapsto y] & \stackrel{\mathsf{def}}{=} \lambda k.\ \mathsf{if}\ k = x\ \mathsf{then}\ \mathsf{Some}\ y\ \mathsf{else}\ t\ k \\
s \oplus t & \stackrel{\mathsf{def}}{=} \lambda k.\ \mathsf{case}\ t\ k\ \mathsf{of}\ \mathsf{None} \Rightarrow s\ k \mid \mathsf{Some}\ x \Rightarrow \mathsf{Some}\ x
\end{array}$$

$$\begin{array}{lll}
\mathsf{table_of}\ [] & = & \mathsf{empty} \\
\mathsf{table_of}\ ((k,x)\#t) & = & (\mathsf{table_of}\ t)[k\mapsto x]
\end{array}$$

$$t\ \mathsf{hiding}\ s\ \mathsf{entails}\ R \stackrel{\mathsf{def}}{=} \forall k\ x\ y.\ t\ k = \mathsf{Some}\ x \longrightarrow s\ k = \mathsf{Some}\ y \longrightarrow R\ x\ y$$

For the union of tables, we also need the type of non-unique tables,

$$(\alpha,\beta)\mathit{tables} \;=\; \alpha \Rightarrow (\beta)\mathit{set}$$

together with a union operator and straightforward variants of two of the notions defined above:

$$\begin{array}{ll}
_ \oplus\oplus _ & :: (\alpha,\beta)\mathit{tables} \Rightarrow (\alpha,\beta)\mathit{tables} \Rightarrow (\alpha,\beta)\mathit{tables} \\
\mathsf{Un_tables} & :: ((\alpha,\beta)\mathit{tables})\mathit{set} \Rightarrow (\alpha,\beta)\mathit{tables} \\
_\ \mathsf{hidings}\ _ & \\
\quad \mathsf{entails}\ _ & :: (\alpha,\beta)\mathit{tables} \Rightarrow (\alpha,\gamma)\mathit{tables} \Rightarrow (\beta \Rightarrow \gamma \Rightarrow \mathit{bool}) \Rightarrow \mathit{bool}
\end{array}$$

$$\mathsf{Un\_tables}\ ts \quad \stackrel{\mathsf{def}}{=}\ \lambda k.\ \bigcup t{\in}ts.\ t\ k$$
$$s \oplus\oplus t \quad \stackrel{\mathsf{def}}{=}\ \lambda k.\ \mathsf{if}\ t\ k = \{\}\ \mathsf{then}\ s\ k\ \mathsf{else}\ t\ k$$
$$t\ \mathsf{hidings}\ s\ \mathsf{entails}\ R \stackrel{\mathsf{def}}{=} \forall k.\ \forall x{\in}t\ k.\ \forall y{\in}s\ k.\ R\ x\ y$$

A simple application of type *table* is the translation of programs to tables indexed by interface and class names:

$$\mathsf{iface} :: prog \Rightarrow (tname,\ iface)table \stackrel{\mathsf{def}}{=} \mathsf{table\_of} \circ \mathsf{fst}$$
$$\mathsf{class} :: prog \Rightarrow (tname,\ class)table \stackrel{\mathsf{def}}{=} \mathsf{table\_of} \circ \mathsf{snd}$$

More interesting are the following functions that traverse the type hierarchy of a program, collecting the methods and fields into a table (the types *tname* and *ref_ty* are defined in §4.2):

$$\begin{aligned}
&\mathsf{imethds} :: prog \Rightarrow tname \Rightarrow (sig,\ &&ref_ty\ \times mhead)tables\\
&\mathsf{cmethd}\ \ :: prog \Rightarrow tname \Rightarrow (sig,\ &&ref_ty\ \times methd)table\\
&\mathsf{fields}\ \ \ :: prog \Rightarrow tname \Rightarrow ((ename \times ref_ty) \times field\ )list
\end{aligned}$$

Note that imethds collects a non-unique table of method declarations allowing for inheritance of more than one method with the same signature.

As Syme [Sym98] points out, a naive recursive definition of these functions is not possible in HOL because the class hierarchy might be cyclic, which is ruled out for well-formed programs (see §4.3) only. This leads to partial functions, which HOL does not support directly. Syme defines these functions as relations instead. In contrast, we have chosen to define them as proper functions, based on Slind's work on well-founded recursion [Sli96]. We do not give their definitions, but only the recursion equations, which we derive as easy consequences:

$$\begin{aligned}
&\mathsf{wf_prog}\ \Gamma \wedge \mathsf{iface}\ \Gamma\ I = \mathsf{Some}\ (is,ms) \longrightarrow\\
&\mathsf{imethds}\ \Gamma\ I = \mathsf{Un_tables}\ ((\lambda J.\ \mathsf{imethds}\ \Gamma\ J)\text{`` set}\ is) \oplus\oplus\\
&\qquad\qquad (\mathsf{o2s} \circ \mathsf{table_of}\ (\mathsf{map}\ (\lambda(s,mh).\ (s,\mathsf{IfaceT}\ I,mh))\ ms))
\end{aligned}$$

$$\begin{aligned}
&\mathsf{wf_prog}\ \Gamma \wedge \mathsf{class}\ \Gamma\ C = \mathsf{Some}\ (sc,si,fs,ms) \longrightarrow\\
&\mathsf{cmethd}\ \Gamma\ C = (\mathsf{case}\ sc\ \mathsf{of\ None} \Rightarrow \mathsf{empty} \mid \mathsf{Some}\ D \Rightarrow \mathsf{cmethd}\ \Gamma\ D) \oplus\\
&\qquad\qquad \mathsf{table_of}\ (\mathsf{map}\ (\lambda(s,m).\ (s,(\mathsf{ClassT}\ C,m)))\ ms)
\end{aligned}$$

$$\begin{aligned}
&\mathsf{wf_prog}\ \Gamma \wedge \mathsf{class}\ \Gamma\ C = \mathsf{Some}\ (sc,si,fs,ms) \longrightarrow\\
&\mathsf{fields}\ \Gamma\ C = \mathsf{map}\ (\lambda(fn,ft).\ ((fn,\mathsf{ClassT}\ C),ft))\ fs\ @\\
&\qquad\qquad (\mathsf{case}\ sc\ \mathsf{of\ None} \Rightarrow [] \mid \mathsf{Some}\ D \Rightarrow \mathsf{fields}\ \Gamma\ D)
\end{aligned}$$

The structure of the three equations is the same: the tables are constructed recursively from the corresponding tables of the superinterfaces or the superclass (if any), which models inheritance, augmented — with overriding — by the newly declared items. All declared items receive an extra label, namely their defining interface or class.

In our example, we obtain

```
fields tprg Base = [((vee, ClassT Base), PrimT boolean)]
fields tprg Ext  = [((vee, ClassT Ext ), PrimT int),
                    ((vee, ClassT Base), PrimT boolean)]
cmethd tprg Base = empty[(foo, Class Base)↦
                         (ClassT Base, (x, Class Base), ([], Skip, x))]
cmethd tprg Ext  = empty[(foo, Class Base)↦
                         (ClassT Ext , (x, Class Ext ), ([],
                           Expr({ClassT Ext}(Class Ext)x.vee:=Lit (Intg 1)),
                           Lit Null))]
```

Terms. We define statements (appearing in method bodies), expressions (appearing in statements), and values (appearing in expressions) as recursive datatypes.

Statements are reduced to their bare essentials. We do not formalize syntactic variants of conditionals and loops. Neither do we consider jumps like the `break` statement.

For a more modular description, we divide the `try _ catch _ finally _` statement into a `try _ catch _` statement and a `_ finally _` statement, which might be used in any context. Our version of the `try _ catch _` statement has exactly one `catch` clause. Multiple `catch` clauses can be simulated with cascaded `if _ else _` statements applying the `_ instanceof _` operator.

```
stmt = Skip
     | Expr expr
     | stmt; stmt
     | if    (expr) stmt else stmt
     | while(expr) stmt
     | throw(expr)
     | try stmt catch(tname ename) stmt
     | stmt finally stmt
```

Skip denotes the empty statement. The "expression statement" Expr is a conversion from expressions to statements causing evaluation for side effects only. Assignments and method calls, which are expressions because they yield a value, can be turned into statements this way. In contrast to Java, for simplicity we allow this conversion to be applied to any kind of expression.

Concerning expressions, our formalization leaves out the standard unary and binary operators as their typing and semantics is straightforward. The `this` expression is modeled as a special non-assignable local variable named `this`. The `super` construct can be simulated with a `this` expression that is cast to the superclass of the current class. Creation of multi-dimensional arrays can be simulated with nested array creation, while access and assignment to multi-dimensional arrays is nested anyway.

It might be reasonable to introduce the general notion of variables (i.e. left-hand sides of assignments) in order to factor out common behavior of local variables, class instance variables, and array components. But we have chosen not to do so because the semantic treatment of these three variants of variables differs considerably. This decision leads to some redundancy between access and assignment, especially in the semantics for arrays.

$expr =$ `new` $tname$	class instance creation
$\mid$ `new` $ty[expr]$	array creation
$\mid (ty)expr$	type cast
$\mid expr$ `instanceof` ref_ty	type comparison operator
$\mid$ Lit val	literal
$\mid ename$	local/parameter access
$\mid ename$`:=`$expr$	local/parameter assignment
$\mid \{ref\_ty\}expr.ename$	field access
$\mid \{ref\_ty\}expr.ename$`:=`$expr$	field assignment
$\mid expr[expr]$	array access
$\mid expr[expr]$`:=`$expr$	array assignment
$\mid expr.mname(\{ty\}expr)$	method call

The terms in braces $\{\ldots\}$ above are called *type annotations*. Strictly speaking, they are not part of the input language but serve as auxiliary information (computed by the type checker) that is crucial for the static binding of fields and the resolution of method overloading. Distinguishing between the actual input language and the augmented language would lead to a considerable amount of redundancy. We avoid this by assuming that the annotations are added beforehand by a kind of preprocessor. The correctness of the annotations is checked by the typing rules (see §4.2).

The definition of values is straightforward. It relies on the standard HOL types of Boolean values (*bool*) and integers (*int*), whereas the type *loc* of locations, i.e. abstract non-null addresses of objects, is not further specified.

$val =$ Unit	dummy result of `void` methods
$\mid$ Bool $bool$	
$\mid$ Intg int	
$\mid$ Null	
$\mid$ Addr loc	

The definitions below give some simple destructor functions for *val* with their characteristic properties.

$$\begin{aligned}
&\mathsf{the_Bool} :: val \Rightarrow bool\\
&\mathsf{the_Intg} :: val \Rightarrow int\\
&\mathsf{the_Addr} :: val \Rightarrow loc
\end{aligned}$$

$$\begin{aligned}
&\mathsf{the_Bool}\ (\mathsf{Bool}\ b) = b\\
&\mathsf{the_Intg}\ \ (\mathsf{Intg}\ \ i) = i\\
&\mathsf{the_Addr}\ (\mathsf{Addr}\ a) = a
\end{aligned}$$

4.2 Type System

This section defines types, various ordering relations between types, and the typing rules for statements and expressions.

Types. We formalize Bali types as values of datatype *ty*, dividing them into primitive and reference types:

$$
\begin{array}{lll}
prim\_ty & = & \texttt{void} \\
 & | & \texttt{boolean} \\
 & | & \texttt{int} \\
 & & \\
ref\_ty & = & \mathsf{NullT} \\
 & | & \mathsf{IfaceT}\ tname \\
 & | & \mathsf{ClassT}\ tname \\
 & | & \mathsf{ArrayT}\ ty \\
 & & \\
ty & = & \mathsf{PrimT}\ prim\_ty \\
 & | & \mathsf{RefT}\ \ ref\_ty
\end{array}
$$

`void` is used as a dummy type for methods without result. In the sequel NT stands for RefT NullT, Iface I for RefT(IfaceT I), Class C for RefT(ClassT C), and T[] for RefT(ArrayT T).

An interface or class type is considered as a proper type only if there is a corresponding declaration for its type name in the current program, which is checked by the following predicates:

$$
\begin{array}{l}
\mathsf{is\_iface} :: prog \Rightarrow tname \Rightarrow bool \\
\mathsf{is\_class} :: prog \Rightarrow tname \Rightarrow bool \\
\mathsf{is\_type} :: prog \Rightarrow ty \Rightarrow bool
\end{array}
$$

$$
\begin{array}{lll}
\mathsf{is\_iface}\ \Gamma\ tn & \stackrel{\mathsf{def}}{=} & \mathsf{iface}\ \Gamma\ tn \neq \mathsf{None} \\
\mathsf{is\_class}\ \Gamma\ tn & \stackrel{\mathsf{def}}{=} & \mathsf{class}\ \Gamma\ tn \neq \mathsf{None} \\
\mathsf{is\_type}\ \Gamma\ (\mathsf{PrimT}\ \_) & = & \mathsf{True} \\
\mathsf{is\_type}\ \Gamma\ \mathsf{NT} & = & \mathsf{True} \\
\mathsf{is\_type}\ \Gamma\ (\mathsf{Iface}\ I) & = & \mathsf{is\_iface}\ \Gamma\ I \\
\mathsf{is\_type}\ \Gamma\ (\mathsf{Class}\ C) & = & \mathsf{is\_class}\ \Gamma\ C \\
\mathsf{is\_type}\ \Gamma\ (T[]) & = & \mathsf{is\_type}\ \Gamma\ T
\end{array}
$$

For all types, a default value is defined via

$$
\begin{array}{l}
\mathsf{default\_val} :: ty \Rightarrow val \\
\mathsf{default\_val}\ (\mathsf{PrimT}\ \texttt{void}\ \ ) = \mathsf{Unit} \\
\mathsf{default\_val}\ (\mathsf{PrimT}\ \texttt{boolean}) = \mathsf{Bool\ False} \\
\mathsf{default\_val}\ (\mathsf{PrimT}\ \texttt{int}\ \ ) = \mathsf{Intg\ 0} \\
\mathsf{default\_val}\ (\mathsf{RefT}\ \ r\ \ ) = \mathsf{Null}
\end{array}
$$

Type Relations. The relations between types depend on the interface and class hierarchy of a given program Γ, and are therefore expressed with reference to Γ. The direct subinterface ($\_ \vdash \_ \prec^1_{\mathsf{i}} \_$), subclass ($\_ \vdash \_ \prec^1_{\mathsf{c}} \_$), and implementation ($\_ \vdash \_ \rightsquigarrow^1 \_$) relations are of type $\mathit{prog} \times \mathit{tname} \times \mathit{tname} \Rightarrow \mathit{bool}$ and are defined as follows:

$$\Gamma \vdash I \prec^1_{\mathsf{i}} J \stackrel{\mathsf{def}}{=} \mathsf{is\_iface}\ \Gamma\ I \wedge \mathsf{is\_iface}\ \Gamma\ J \wedge J \in \mathsf{set}\ (\mathsf{fst}(\mathsf{the}(\mathsf{iface}\ \Gamma\ I)))$$
$$\Gamma \vdash C \prec^1_{\mathsf{c}} D \stackrel{\mathsf{def}}{=} \mathsf{is\_class}\ \Gamma\ C \wedge \mathsf{is\_class}\ \Gamma\ D \wedge \mathsf{Some}\ D = \mathsf{fst}(\mathsf{the}(\mathsf{class}\ \Gamma\ C))$$
$$\Gamma \vdash C \rightsquigarrow^1 I \stackrel{\mathsf{def}}{=} \mathsf{is\_class}\ \Gamma\ C \wedge \mathsf{is\_iface}\ \Gamma\ I \wedge I \in \mathsf{set}\ (\mathsf{fst}(\mathsf{snd}(\mathsf{the}(\mathsf{class}\ \Gamma\ C))))$$

The transitive (but not reflexive) closures $\_ \vdash \_ \prec_{\mathsf{i}} \_$ and $\_ \vdash \_ \prec_{\mathsf{c}} \_$ can be defined inductively:

$$\frac{\Gamma \vdash I \prec^1_{\mathsf{i}} K}{\Gamma \vdash I \prec_{\mathsf{i}} K} \qquad \frac{\Gamma \vdash I \prec_{\mathsf{i}} J;\ \ \Gamma \vdash J \prec_{\mathsf{i}} K}{\Gamma \vdash I \prec_{\mathsf{i}} K}$$
$$\frac{\Gamma \vdash C \prec^1_{\mathsf{c}} E}{\Gamma \vdash C \prec_{\mathsf{c}} E} \qquad \frac{\Gamma \vdash C \prec_{\mathsf{c}} D;\ \ \Gamma \vdash D \prec_{\mathsf{c}} E}{\Gamma \vdash C \prec_{\mathsf{c}} E}$$

There is also a kind of transitive closure of $\_ \vdash \_ \rightsquigarrow^1 \_$ defined as

$$\frac{\Gamma \vdash C \rightsquigarrow^1 J}{\Gamma \vdash C \rightsquigarrow J} \qquad \frac{\Gamma \vdash C \rightsquigarrow^1 I;\ \ \Gamma \vdash I \prec_{\mathsf{i}} J}{\Gamma \vdash C \rightsquigarrow J} \qquad \frac{\Gamma \vdash C \prec^1_{\mathsf{c}} D;\ \ \Gamma \vdash D \rightsquigarrow J}{\Gamma \vdash C \rightsquigarrow J}$$

The key relation is widening: $\Gamma \vdash S \preceq T$, where S and T are of type ty, means that S is a syntactic subtype of T, i.e. in any expression context (especially assignments and method invocations) expecting a value of type T, a value of type S may occur. Note that this does not necessarily mean that type S behaves like type T, but only that it has a syntactically compatible set of fields and methods. The widening relation is defined inductively as given below. Note that some rules carry the additional premise that `Object` is a proper class, which will be ensured for well-formed programs.

$$\frac{\mathsf{is\_type}\ \Gamma\ T}{\Gamma \vdash T \preceq T} \qquad \frac{\mathsf{is\_type}\ \Gamma\ (\mathsf{RefT}\ R)}{\Gamma \vdash \mathsf{NT} \preceq \mathsf{RefT}\ R}$$
$$\frac{\Gamma \vdash I \prec_{\mathsf{i}} J}{\Gamma \vdash \mathsf{Iface}\ I \preceq \mathsf{Iface}\ J} \qquad \frac{\mathsf{is\_iface}\ \Gamma\ I;\ \ \mathsf{is\_class}\ \Gamma\ \mathtt{Object}}{\Gamma \vdash \mathsf{Iface}\ I \preceq \mathsf{Class}\ \mathtt{Object}}$$
$$\frac{\Gamma \vdash C \prec_{\mathsf{c}} D}{\Gamma \vdash \mathsf{Class}\ C \preceq \mathsf{Class}\ D} \qquad \frac{\Gamma \vdash C \rightsquigarrow I}{\Gamma \vdash \mathsf{Class}\ C \preceq \mathsf{Iface}\ I}$$
$$\frac{\Gamma \vdash \mathsf{RefT}\ S \preceq \mathsf{RefT}\ T}{\Gamma \vdash (\mathsf{RefT}\ S)[] \preceq (\mathsf{RefT}\ T)[]} \qquad \frac{\mathsf{is\_type}\ \Gamma\ T[];\ \ \mathsf{is\_class}\ \Gamma\ \mathtt{Object}}{\Gamma \vdash T[] \preceq \mathsf{Class}\ \mathtt{Object}}$$

To allow for type casting we also have the casting relation, where $\Gamma \vdash S \preceq_? T$ means that a value of type S may be cast to type T:

$$\frac{\Gamma \vdash S \preceq T}{\Gamma \vdash S \preceq_? T} \qquad \frac{\Gamma \vdash C \prec_{\mathsf{c}} D}{\Gamma \vdash \mathsf{Class}\ D \preceq_? \mathsf{Class}\ C} \qquad \frac{\mathsf{is\_class}\ \Gamma\ C;\ \ \mathsf{is\_iface}\ \Gamma\ I}{\Gamma \vdash \mathsf{Class}\ C \preceq_? \mathsf{Iface}\ I}$$

$$\frac{\Gamma \vdash \mathsf{RefT}\ S \preceq_? \mathsf{RefT}\ T}{\Gamma \vdash (\mathsf{RefT}\ S)[] \preceq_? (\mathsf{RefT}\ T)[]} \qquad \frac{\mathsf{is\_class}\ \Gamma\ \mathtt{Object};\ \ \mathsf{is\_type}\ \Gamma\ T}{\Gamma \vdash \mathsf{Class}\ \mathtt{Object} \preceq_? T[]}$$

$$\frac{\begin{array}{c}\mathsf{is\_iface}\ \Gamma\ J;\ \neg\Gamma \vdash I \prec_{\mathsf{i}} J;\\ \mathsf{imethds}\ \Gamma\ I\ \mathsf{hidings}\ \mathsf{imethds}\ \Gamma\ J\ \mathsf{entails}\\ (\lambda(m_1,(pn_1,rT_1))\ (m_2,(pn_2,rT_2)).\ \Gamma \vdash rT_1 \preceq rT_2)\end{array}}{\Gamma \vdash \mathsf{Iface}\ I \preceq_? \mathsf{Iface}\ J} \qquad \frac{\mathsf{is\_iface}\ \Gamma\ I;\ \ \mathsf{is\_class}\ \Gamma\ C}{\Gamma \vdash \mathsf{Iface}\ I \preceq_? \mathsf{Class}\ C}$$

Typing Rules. Now we come to type-checking itself, which is expressed as a set of constraints on the types of expressions, relative to a type environment.

An *environment* consists of a global part, namely a program Γ, and a local part (written 'Λ'), namely the types of the local variables including the current class, i.e. the type of `this`:

$$\begin{array}{l} lenv = (ename,\ ty)\ table\\ env = prog \times lenv \end{array}$$

$$\begin{array}{l} \mathsf{prg} :: env \Rightarrow prog \stackrel{\mathsf{def}}{=} \lambda(\Gamma,\Lambda).\ \Gamma\\ \mathsf{lcl}\ :: env \Rightarrow lenv \stackrel{\mathsf{def}}{=} \lambda(\Gamma,\Lambda).\ \Lambda \end{array}$$

The well-typedness of statements and the typing of expressions are defined inductively relative to an environment. The typing of expressions is unique, as can be shown easily by rule induction.

$$\begin{array}{ll} \_ \vdash \_ :: \diamond & :: env \Rightarrow stmt \Rightarrow bool\\ \_ \vdash \_ :: \_ & :: env \Rightarrow expr \Rightarrow ty \Rightarrow bool \end{array}$$

The type-checking rules for most statements are standard:

$$\frac{}{E \vdash \mathsf{Skip} :: \diamond} \qquad \frac{E \vdash e :: T}{E \vdash \mathsf{Expr}\ e :: \diamond} \qquad \frac{E \vdash c_1 :: \diamond;\ \ E \vdash c_2 :: \diamond}{E \vdash c_1\ ;\ c_2 :: \diamond}$$

$$\frac{E \vdash e :: \mathsf{PrimT}\ \texttt{boolean};\ \ E \vdash c_1 :: \diamond;\ \ E \vdash c_2 :: \diamond}{E \vdash \texttt{if}(e)\ c_1\ \texttt{else}\ c_2 :: \diamond}$$

$$\frac{E \vdash e :: \mathsf{PrimT}\ \texttt{boolean};\ \ E \vdash c :: \diamond}{E \vdash \texttt{while}(e)\ c :: \diamond} \qquad \frac{E \vdash c_1 :: \diamond;\ \ E \vdash c_2 :: \diamond}{E \vdash c_1\ \texttt{finally}\ c_2}$$

Note the use of the widening relation in the following two rules to ensure that a value thrown or caught as an exception is indeed a exception object.

$$\frac{E \vdash e :: \mathsf{Class}\ tn;\ \ \mathsf{prg}\ E \vdash \mathsf{Class}\ tn \preceq \mathsf{Class}\ (\mathsf{SXcpt}\ \texttt{Throwable})}{E \vdash \texttt{throw}\ e :: \diamond}$$

$$\frac{\begin{array}{c}(\Gamma,\Lambda) \vdash c_1 :: \diamond;\ \ \Gamma \vdash \mathsf{Class}\ tn \preceq \mathsf{Class}\ (\mathsf{SXcpt}\ \texttt{Throwable});\\ \Lambda\ vn = \mathsf{None};\ \ (\Gamma,\Lambda[vn \mapsto \mathsf{Class}\ tn]) \vdash c_2 :: \diamond\end{array}}{(\Gamma,\Lambda) \vdash \texttt{try}\ c_1\ \texttt{catch}(tn\ vn)\ c_2 :: \diamond}$$

The `try _ catch _` statement is the only one that involves a change of the type environment, namely to include typing information for the exception parameter. The name of this parameter is required to be new in the local environment.

The typing rules for the first few of the expressions are straightforward, except for the confusing direction of the casting relation in the type cast rule:

$$\frac{\mathsf{is\_class}\ (\mathsf{prg}\ E)\ C}{E \vdash \texttt{new}\ C :: \mathsf{Class}\ C} \qquad \frac{\mathsf{is\_type}\ (\mathsf{prg}\ E)\ T;\ \ E \vdash i :: \mathsf{PrimT}\ \texttt{int}}{E \vdash \texttt{new}\ T[i] :: T[]}$$

$$\frac{E \vdash e :: T;\ \ \mathsf{prg}\ E \vdash T \preceq_? T'}{E \vdash (T')e :: T'} \qquad \frac{\mathsf{typeof}\ (\lambda a.\ \mathsf{None})\ x = \mathsf{Some}\ T}{E \vdash \mathsf{Lit}\ x :: T}$$

$$\frac{E \vdash e :: \mathsf{RefT}\ T;\ \ \mathsf{prg}\ E \vdash \mathsf{RefT}\ T \preceq_? \mathsf{RefT}\ T'}{E \vdash e\ \texttt{instanceof}\ T' :: \mathsf{PrimT}\ \texttt{boolean}}$$

The rule for Lit prohibits addresses as literal values, which is implemented by supplying $\lambda a.$ None as the "dynamic type" argument in the call of the function

$$\begin{array}{l}
\mathsf{typeof} :: (loc \Rightarrow ty\ option) \Rightarrow val \Rightarrow ty\ option \\
\mathsf{typeof}\ dt\ \ \mathsf{Unit} \quad\ = \mathsf{Some}\ (\mathsf{PrimT}\ \texttt{void}) \\
\mathsf{typeof}\ dt\ (\mathsf{Bool}\ b) = \mathsf{Some}\ (\mathsf{PrimT}\ \texttt{boolean}) \\
\mathsf{typeof}\ dt\ (\mathsf{Intg}\ i) \ = \mathsf{Some}\ (\mathsf{PrimT}\ \texttt{int}) \\
\mathsf{typeof}\ dt\ \ \mathsf{Null} \quad\ = \mathsf{Some}\ (\mathsf{RefT}\ \ \mathsf{NullT}) \\
\mathsf{typeof}\ dt\ (\mathsf{Addr}\ a) = dt\ a
\end{array}$$

This function is reused below with a more interesting value for the parameter dt, namely a function to compute the dynamic type of a reference.

The typings of all three assignment variants are quite similar, except that for local variables additionally an assignment to `this` is forbidden. In any case, as a generalization to the Java specification, the type of the assignment is determined by the right-hand (as opposed to the left-hand) side.

$$\frac{\mathsf{lcl}\ E\ vn = \mathsf{Some}\ T;\ \mathsf{is\_type}\ (\mathsf{prg}\ E)\ T}{E \vdash vn :: T}$$

$$\frac{E \vdash vn :: T;\ \ E \vdash v :: T';\ \ \mathsf{prg}\ E \vdash T' \preceq T;\ \ vn \neq \texttt{this}}{E \vdash vn := v :: T'}$$

$$\frac{E \vdash e :: \mathsf{Class}\ C;\ \ \mathsf{cfield}\ (\mathsf{prg}\ E)\ C\ fn = \mathsf{Some}\ (fd,fT)}{E \vdash \{fd\}e.fn :: fT}$$

$$\frac{E \vdash \{fd\}e.fn :: T;\ \ E \vdash v :: T';\ \ \mathsf{prg}\ E \vdash T' \preceq T}{E \vdash \{fd\}e.fn := v :: T'}$$

$$\frac{E \vdash a :: T[];\ \ E \vdash i :: \mathsf{PrimT}\ \texttt{int}}{E \vdash a[i] :: T}$$

$$\frac{E \vdash a[i] :: T;\ \ E \vdash v :: T';\ \ \mathsf{prg}\ E \vdash T' \preceq T}{E \vdash a[i] := v :: T'}$$

$$\frac{\begin{array}{c} E \vdash e :: \mathsf{RefT}\ T;\ \ E \vdash p :: pT; \\ \mathsf{max\_spec}\ (\mathsf{prg}\ E)\ T\ (mn,pT) = \{((md,(pn,rT)),pT')\} \end{array}}{E \vdash e.mn(\{pT'\}p) :: rT}$$

The function $\mathsf{cfield} :: \mathit{prog} \Rightarrow \mathit{tname} \Rightarrow (\mathit{ename}\ ,\ \mathit{ref\_ty} \times \mathit{field})\mathit{table}$, defined as $\mathsf{cfield}\ \Gamma\ C \stackrel{\mathsf{def}}{=} \mathsf{table\_of}\ ((\mathsf{map}\ (\lambda((n,d),t).\ (n,(d,t))))\ (\mathsf{fields}\ \Gamma\ C))$, is a variant of fields. It implements a field lookup that is based on the field name alone in contrast to a combination of field name and defining class. Thus in the above typing rule for field access, equal field names hide each other, while at run-time all fields are accessible, using the defining class as an additional search key.

The type annotations $\{\ldots\}$ in the above rules for field access and method call are used to implement static binding for fields and to resolve overloaded method names statically. Technically speaking, the typing rules serve as constraints on these annotations during type-checking, but one can also think of the annotations being filled with schematic variables that are instantiated with their correct values in the type-checking process, as is demonstrated in the example overleaf. The value of each annotation is uniquely determined by the value of a function in the premise of the field access and method call rule:

A field access $\{\mathit{fd}\}e.\mathit{fn}$ is annotated with the defining class of the field found when searching the class hierarchy for the name fn (using cfield), starting from the static type $\mathsf{Class}\ C$ of e. The annotation $\{\mathit{fd}\}$ will be used at run-time to access the field via the pair $(\mathit{fn},\mathit{fd})$.

A method call $e.\mathit{mn}(\{pT'\}p)$ is type-correct only if the function max_spec determining the set of "maximally specific" [GJS96, 15.11.2] methods for reference type T (as defined below) yields exactly one method entry. In this case, the method call is annotated by pT', which is the argument type of the most specific method mn applicable according to the static types T of e and pT of p. Thus any static overloading of the method name mn has been resolved and the dynamic method lookup at run-time will be based on the signature (mn,pT').

$$
\begin{array}{lll}
\mathsf{max\_spec} & :: \mathit{prog} \Rightarrow \mathit{ref\_ty} \Rightarrow \mathit{sig} & \Rightarrow((\mathit{ref\_ty} \times \mathit{mhead}) \times \mathit{ty})\ \ \mathit{set} \\
\mathsf{appl\_methds} & :: \mathit{prog} \Rightarrow \mathit{ref\_ty} \Rightarrow \mathit{sig} & \Rightarrow((\mathit{ref\_ty} \times \mathit{mhead}) \times \mathit{ty})\ \ \mathit{set} \\
\mathsf{mheads} & :: \mathit{prog} \Rightarrow \mathit{ref\_ty} \Rightarrow \mathit{sig} & \Rightarrow (\mathit{ref\_ty} \times \mathit{mhead})\ \ \mathit{set} \\
\mathsf{more\_spec} & :: \mathit{prog} \Rightarrow (\mathit{ref\_ty} \times \mathit{mhead}) \times \mathit{ty} \Rightarrow (\mathit{ref\_ty} \times \mathit{mhead}) \times \mathit{ty} \Rightarrow \mathit{bool}
\end{array}
$$

$$
\begin{array}{l}
\mathsf{max\_spec}\ \Gamma\ T\ \mathit{sig} \stackrel{\mathsf{def}}{=} \{m \mid m \in \mathsf{appl\_methds}\ \Gamma\ T\ \mathit{sig} \wedge \\
\qquad\qquad (\forall m' \in \mathsf{appl\_methds}\ \Gamma\ T\ \mathit{sig}. \\
\qquad\qquad\qquad \mathsf{more\_spec}\ \Gamma\ m'\ m \longrightarrow m' = m)\} \\
\mathsf{appl\_methds}\ \Gamma\ T\ (\mathit{mn},\ pT) \stackrel{\mathsf{def}}{=} \{(m,pT') \mid m \in \mathsf{mheads}\ \Gamma\ T\ (\mathit{mn},\ pT') \wedge \\
\qquad\qquad\qquad\qquad\qquad\qquad \Gamma \vdash pT \preceq pT'\} \\
\mathsf{mheads}\ \Gamma\ \ \mathsf{NullT} \qquad = \lambda \mathit{sig}.\ \{\} \\
\mathsf{mheads}\ \Gamma\ (\mathsf{IfaceT}\ \ I) \ = \mathsf{imethds}\ \Gamma\ I \\
\mathsf{mheads}\ \Gamma\ (\mathsf{ClassT}\ \ C) = \mathsf{o2s} \circ \mathsf{option\_map}\ (\lambda(d,(h,b)).(d,h)) \circ \mathsf{cmethd}\ \Gamma\ C \\
\mathsf{mheads}\ \Gamma\ (\mathsf{ArrayT}\ \ T) = \lambda \mathit{sig}.\ \{\} \\
\mathsf{more\_spec}\ \Gamma\ ((md,mh),pT)\ ((md',mh'),pT') \stackrel{\mathsf{def}}{=} \Gamma \vdash \mathsf{RefT}\ md \preceq \mathsf{RefT}\ md' \wedge \\
\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad \Gamma \vdash pT \preceq pT'
\end{array}
$$

where

$$
\begin{array}{l}
\mathsf{option\_map} :: (\alpha \Rightarrow \beta) \Rightarrow (\alpha\ \mathit{option} \Rightarrow \beta\ \mathit{option}) \\
\mathsf{option\_map}\ f \stackrel{\mathsf{def}}{=} \lambda y.\ \mathsf{case}\ y\ \mathsf{of}\ \mathsf{None} \Rightarrow \mathsf{None} \mid \mathsf{Some}\ x \Rightarrow \mathsf{Some}\ (f\ x)
\end{array}
$$

The well-typedness of our example code *test* is derived as given below. For formatting reasons, the derivation tree is cut at several positions, whereby the positions are marked with the labels of the cut subtrees. Irrelevant values in formulas are replaced by _. We use the following abbreviations:

$\Gamma \quad = tprg$
$\Lambda \quad = \mathsf{empty}[e{\mapsto}\mathsf{Class}\ Base]$
$SNP = \mathsf{SXcpt}\ \texttt{NullPointer}$

During the derivation, the schematic variable $?pT'$ is instantiated with $\mathsf{Class}\ Base$, as a result of the function max_spec.

$$\dfrac{\dfrac{\dfrac{\Lambda\ e = \mathsf{Some}\ (\mathsf{Class}\ Base) \quad \mathsf{is\_type}\ \Gamma\ (\mathsf{Class}\ Base)}{(\Gamma,\Lambda) \vdash e{::}\mathsf{Class}\ Base} \quad e \neq \texttt{this} \quad \dfrac{\mathsf{is\_class}\ \Gamma\ Ext}{(\Gamma,\Lambda) \vdash \texttt{new}\ Ext{::}\mathsf{Class}\ Ext} \quad \Gamma \vdash \mathsf{Class}\ Ext \preceq \mathsf{Class}\ Base}{(\Gamma,\Lambda) \vdash (e\texttt{:=new}\ Ext){::}\_}}{(\Gamma,\Lambda) \vdash \mathsf{Expr}(e\texttt{:=new}\ Ext){::}\diamond}\ (LAss)$$

$$\dfrac{\dfrac{\dfrac{\Lambda\ e = \mathsf{Some}\ (\mathsf{Class}\ Base) \quad \mathsf{is\_type}\ \Gamma\ (\mathsf{Class}\ Base)}{(\Gamma,\Lambda) \vdash e{::}\mathsf{Class}\ Base} \quad \dfrac{\mathsf{typeof}\ (\lambda a.\ \mathsf{None})\ \mathsf{Null} = \mathsf{Some}\ \mathsf{NT}}{(\Gamma,\Lambda) \vdash \mathsf{Lit}\ \mathsf{Null}{::}\mathsf{NT}} \quad \mathsf{max\_spec}\ \Gamma\ (\mathsf{ClassT}\ Base)\ (foo, \mathsf{NT}) = \{((\_, \_, \mathsf{Class}\ Base), ?pT')\}}{(\Gamma,\Lambda) \vdash (e.foo(\{?pT'\}\mathsf{Lit}\ \mathsf{Null})){::}\mathsf{Class}\ Base}}{(\Gamma,\Lambda) \vdash \mathsf{Expr}(e.foo(\{?pT'\}\mathsf{Lit}\ \mathsf{Null})){::}\diamond}\ (Call)$$

$$\dfrac{\dfrac{\Lambda[x{\mapsto}\mathsf{Class}\ SNP]\ x = \mathsf{Some}\ (\mathsf{Class}\ SNP) \quad \mathsf{is\_type}\ \Gamma\ (\mathsf{Class}\ SNP)}{(\Gamma,\ \Lambda[x{\mapsto}\mathsf{Class}\ SNP]) \vdash x{::}\mathsf{Class}\ SNP} \quad \Gamma \vdash \mathsf{Class}\ SNP \preceq \mathsf{Class}\ (\mathsf{SXcpt}\ \texttt{Throwable})}{(\Gamma,\ \Lambda[x{\mapsto}\mathsf{Class}\ SNP]) \vdash \texttt{throw}\ x{::}\diamond}\ (Throw)$$

$$\dfrac{(LAss) \qquad \dfrac{(Call) \quad \Gamma \vdash \mathsf{Class}\ SNP \preceq \mathsf{Class}\ (\mathsf{SXcpt}\ \texttt{Throwable}) \quad \Lambda\ x = \mathsf{None} \quad (Throw)}{(\Gamma,\Lambda) \vdash \texttt{try}\ \mathsf{Expr}(e.foo(\{?pT'\}\mathsf{Lit}\ \mathsf{Null}))\ \texttt{catch}(SNP\ x)\ (\texttt{throw}\ x){::}\diamond}}{(\Gamma,\Lambda) \vdash \mathsf{Expr}(e\texttt{:=new}\ Ext)\texttt{;}\ \texttt{try}\ \mathsf{Expr}(e.foo(\{?pT'\}\mathsf{Lit}\ \mathsf{Null}))\ \texttt{catch}(SNP\ x)\ (\texttt{throw}\ x){::}\diamond}$$

4.3 Well-Formedness

A program must satisfy a number of well-formedness conditions concerning global properties of all declarations. The conditions are expressed as predicates on field, method, interface, class, and whole program declarations.

$$\begin{array}{lll}
\mathsf{wf_fdecl} & :: prog \Rightarrow & fdecl \Rightarrow bool \\
\mathsf{wf_mhead} & :: prog \Rightarrow sig & \times mhead \Rightarrow bool \\
\mathsf{wf_mdecl} & :: prog \Rightarrow tname & \Rightarrow mdecl \Rightarrow bool \\
\mathsf{wf_idecl} & :: prog \Rightarrow & idecl \Rightarrow bool \\
\mathsf{wf_cdecl} & :: prog \Rightarrow & cdecl \Rightarrow bool \\
\mathsf{wf_prog} & :: prog \Rightarrow & bool
\end{array}$$

A field declaration is well-formed iff its type exists:

$$\mathsf{wf\_fdecl}\ \Gamma\ (fn,ft) \stackrel{\mathsf{def}}{=} \mathsf{is\_type}\ \Gamma\ ft$$

A method declaration is well-formed only if its argument and result types are defined and the name of the parameter is not `this`. Additionally, if the declaration appears in a class, the names of the local variables must be unique and may not contain the special name `this` nor hide the parameter, all types of the local variables must exist, the method body has to be well-typed (in the static context of its parameter type and the current class), and its result expression must have a type that widens to the result type:

$$\begin{array}{l}
\mathsf{wf_mhead}\ \Gamma \quad ((mn,pT),(pn,rT)) \stackrel{\mathsf{def}}{=} \mathsf{is_type}\ \Gamma\ pT \wedge \mathsf{is_type}\ \Gamma\ rT \wedge pn \neq \texttt{this} \\
\mathsf{wf_mdecl}\ \Gamma\ C\ ((mn,pT),(pn,rT),lvars,blk,res) \stackrel{\mathsf{def}}{=} \\
\quad \mathsf{let}\ ltab = \mathsf{table_of}\ lvars;\ E = (\Gamma,ltab[\texttt{this}\mapsto\mathsf{Class}\ C][pn\mapsto pT]) \\
\quad \mathsf{in}\ \ \mathsf{wf_mhead}\ \Gamma\ ((mn,pT),(pn,rT)) \wedge \\
\qquad \mathsf{unique}\ lvars \wedge ltab\ \texttt{this} = \mathsf{None} \wedge ltab\ pn = \mathsf{None} \wedge \\
\qquad (\forall (vn,T) \in \mathsf{set}\ lvars.\ \mathsf{is_type}\ \Gamma\ T) \wedge \\
\qquad E \vdash blk :: \diamond \wedge \exists T.\ E \vdash res :: T \wedge \Gamma \vdash T \preceq rT
\end{array}$$

Even more complex conditions are required for well-formed interface and class declarations. The name of a well-formed interface declaration is not a class name. All superinterfaces exist and are not subinterfaces at the same time. All methods newly declared in the interface are named uniquely and are well-formed. Furthermore, any method overriding a set of methods defined in some superinterfaces has a result type that widens to all their result types:

$$\begin{array}{l}
\mathsf{wf_idecl}\ \Gamma\ (I,(is,ms)) \stackrel{\mathsf{def}}{=} \neg\ \mathsf{is_class}\ \Gamma\ I \wedge \\
\quad (\forall J \in \mathsf{set}\ is.\ \mathsf{is_iface}\ \Gamma\ J \wedge \neg\ \Gamma \vdash J \prec_{\mathsf{i}} I) \wedge \\
\quad \mathsf{unique}\ ms \wedge (\forall m \in \mathsf{set}\ ms.\ \mathsf{wf_mhead}\ \Gamma\ m \wedge \\
\quad \mathsf{let}\ mtab = \mathsf{Un_tables}\ ((\lambda J.\ \mathsf{imethds}\ \Gamma\ J)\ ``\ \mathsf{set}\ is)\ \mathsf{in} \\
\quad (\mathsf{o2s} \circ \mathsf{table_of}\ ms)\ \mathsf{hidings}\ mtab\ \mathsf{entails} \\
\qquad (\lambda (pn,rT)\ (m,(pn',rT')).\ \Gamma \vdash rT \preceq rT')
\end{array}$$

Similarly, the name of a well-formed class declaration is not an interface name. All implemented interfaces exist, and for any method of such an interface, the class provides an implementing method with a possibly narrower return type. All fields and methods newly declared in the class are named uniquely and are well-formed. If the class is not `Object`, it refers to an existing superclass, which is not a subclass of the current class. Furthermore, any method overriding a method of the superclass has a compatible result type:

$$
\begin{array}{l}
\mathsf{wf\_cdecl}\ \Gamma\ (C,(sc,si,fs,ms)) \stackrel{\mathsf{def}}{=} \neg\ \mathsf{is\_iface}\ \Gamma\ C \wedge \\
\quad (\forall I{\in}\mathsf{set}\ si.\ \mathsf{is\_iface}\ \Gamma\ I \wedge \\
\qquad \forall s.\ \forall(m_1,(pn_1,rT_1)) \in \mathsf{imethds}\ \Gamma\ I\ s. \\
\qquad\quad \exists m_2\ \ pn_2\ rT_2\ b.\ \ \mathsf{cmethd}\ \ \Gamma\ C\ s = \mathsf{Some}\ (m_2,(pn_2,rT_2),b) \wedge \\
\qquad\qquad\qquad\qquad \Gamma \vdash rT_2 \preceq rT_1) \wedge \\
\quad \mathsf{unique}\ fs\ \wedge (\forall f\ {\in}\mathsf{set}\ fs.\ \ \mathsf{wf\_fdecl}\ \ \Gamma\ \ f\ ) \wedge \\
\quad \mathsf{unique}\ ms \wedge (\forall m{\in}\mathsf{set}\ ms.\ \mathsf{wf\_mdecl}\ \Gamma\ C\ m) \wedge \\
\quad (\mathsf{case}\ sc\ \mathsf{of\ None}\ \ \Rightarrow C = \mathtt{Object} \\
\qquad\quad \mid \mathsf{Some}\ D \Rightarrow \mathsf{is\_class}\ \Gamma\ D \wedge \neg\ \Gamma \vdash D \prec_{\mathsf{c}} C \wedge \\
\qquad\qquad\qquad\quad \mathsf{table\_of}\ ms\ \mathsf{hiding\ cmethd}\ \Gamma\ D\ \mathsf{entails} \\
\qquad\qquad\qquad\quad (\lambda((pn_1,rT_1),b)\ (m,((pn_2,rT_2),b')).\ \Gamma \vdash rT_1 \preceq rT_2)
\end{array}
$$

Finally, all interfaces and classes declared in a well-formed program are named uniquely and are in turn well-formed. For uniformity, this includes the predefined class declarations of `Object` and the (flat) hierarchy of system exceptions.

$$
\begin{array}{l}
\mathsf{ObjectC} \stackrel{\mathsf{def}}{=} (\mathtt{Object}\ ,\ (\mathsf{None}\ \ ,\ [],\ [],\ [])) \\
\mathsf{SXcptC}\ xn \stackrel{\mathsf{def}}{=} \mathsf{let}\ sc = \mathsf{if}\ xn{=}\mathtt{Throwable}\ \mathsf{then}\ \mathtt{Object}\ \mathsf{else\ SXcpt}\ \mathtt{Throwable}\ \mathsf{in} \\
\qquad\qquad (\mathsf{SXcpt}\ xn,\ (\mathsf{Some}\ sc,\ [],\ [],\ []))
\end{array}
$$

$$
\begin{array}{l}
\mathsf{wf\_prog}\ \Gamma \stackrel{\mathsf{def}}{=} \mathsf{let}\ is = \mathsf{set}\ (\mathsf{fst}\ \Gamma);\ cs = \mathsf{set}\ (\mathsf{snd}\ \Gamma) \\
\qquad\qquad \mathsf{in\ ObjectC} \in cs \wedge \forall xn.\ \mathsf{SXcptC}\ xn \in cs \wedge \\
\qquad\qquad\quad \mathsf{unique}\ (\mathsf{fst}\ \ \Gamma) \wedge \forall i{\in}is.\ \ \mathsf{wf\_idecl}\ \ \Gamma\ i) \wedge \\
\qquad\qquad\quad \mathsf{unique}\ (\mathsf{snd}\ \Gamma) \wedge \forall c{\in}cs.\ \mathsf{wf\_cdecl}\ \Gamma\ c)
\end{array}
$$

Our example program *tprg* is well-formed. Here is a heavily abstracted derivation tree of our proof of this fact.

$$
\frac{\mathsf{wf\_mdecl}\ tprg\ Base\ ((foo,\ \mathsf{Class}\ Base),\ (x,\ \mathsf{Class}\ Base),\ [],\ \mathsf{Skip},\ x) \qquad \neg(tprg \vdash \mathtt{Object} \prec_{\mathsf{c}} Base)}{\mathsf{wf\_cdecl}\ tprg\ BaseC}
$$

$$
\frac{\begin{array}{c}\mathsf{wf\_mdecl}\ tprg\ Ext\ ((foo,\ \mathsf{Class}\ Base),\\ (x,\ \mathsf{Class}\ Ext),\ [],\ \mathsf{Expr}\ (\{\mathsf{ClassT}\ Ext\}(\mathsf{Class}\ Ext)\\ x.vee{:=}\mathsf{Lit}\ (\mathsf{Intg}\ 1)),\ \mathsf{Lit\ Null})\end{array} \qquad \neg(tprg \vdash Base \prec_{\mathsf{c}} Ext)}{\mathsf{wf\_cdecl}\ tprg\ BaseC}
$$

$$
\frac{\mathsf{wf\_cdecl}\ tprg\ BaseC \quad \mathsf{wf\_cdecl}\ tprg\ ExtC \quad Base \neq Ext}{\mathsf{wf\_}\ tprg\ tprg}
$$

4.4 Operational Semantics

We formalize the semantics of Java in operational style with evaluation rules. This is the natural choice since the language specification itself is given in an operational evaluation-oriented style, which allows for a direct formalization and its straightforward validation. Furthermore, a denotational semantics would require much more difficult mathematical tools, and an axiomatic semantics would be problematic to validate and to use for reasoning on the language as a whole. We prefer an evaluation semantics to a transition semantics in order to obtain a concise description, because we consider a transition semantics less readable and rather low-level, which in particular holds for a formulation as an Abstract State Machine (ASM) like in [BS98].

In this section, we describe the notions of a state and its components and give the evaluation rules for statements and expressions.

State. A *state* consists of an optional exception (of type *xcpt*), a heap, and a current invocation frame, which is the values of the local variables (including method and exception parameters and the `this` pointer):

$state = (xcpt)option \times st$
$st \quad = heap \times locals$
$\mathsf{heap} :: st \Rightarrow heap \quad \stackrel{\mathsf{def}}{=} \lambda(h,l).\ h$
$\mathsf{locals} :: st \Rightarrow locals \stackrel{\mathsf{def}}{=} \lambda(h,l).\ l$

Remember that tuples associative to the right, so if for some state σ we have an equation like $\sigma = (x, \sigma')$, then x is the (optional) exception component alone, while the second projection σ' of the state has (tuple) type *st*, i.e. represents a "small" state excluding the exception entry.

An *exception* is a reference to an instance of some exception class, which is a subclass of `Throwable`. Normally, when an exception is thrown, a fresh exception object is allocated and its location returned to represent the exception. But in the case of system exceptions, we defer their allocation (and just record their names) until an enclosing `catch` block references it. This helps to avoid the subtleties of (conditional) side effects on the heap and out-of-memory conditions. Thus we model exceptions as follows.

$xcpt = \mathsf{XcptLoc}\ loc$
$\quad\ \mid \mathsf{SysXcpt}\ xname$

A *heap* maps locations to objects, while local variables map names to values:

$heap \ = (loc \quad , obj)table$
$locals = (ename,\ val)table$

In our model there is no need to explicitly maintain a stack of invocation frames containing local variables and return addresses for method calls. In this way we also abstract over the finiteness of stack space. On the other hand, we explicitly model the possibility of memory allocation on the heap to fail if there is no free location (i.e. some a with $(\mathsf{heap}\ \sigma)\ a = \mathsf{None}$) available. Memory allocation is loosely, yet deterministically, defined by the function

$$\begin{aligned}
&\mathsf{new_Addr} :: \mathit{heap} \Rightarrow (\mathit{loc} \times (\mathit{xcpt})\mathit{option})\mathit{option}\\
&\mathsf{new_Addr}\ h \stackrel{\mathsf{def}}{=} \varepsilon y.\ (\quad y = \mathsf{None} \wedge\ (\forall a.\ h\ a \neq \mathsf{None})) \vee\\
&\qquad\qquad (\exists a\ x.\ y = \mathsf{Some}\ (a,x) \wedge h\ a = \mathsf{None} \wedge\\
&\qquad\qquad\quad (x = \mathsf{None} \vee x = \mathsf{Some}\ (\mathsf{SysXcpt}\ \texttt{OutOfMemory})))
\end{aligned}$$

This function fails, i.e. returns None, iff there is no free location on the heap, and otherwise gives an unused location. At the latest when there is only one free address left, it returns an `OutOfMemory` exception. In this way it is guaranteed that when an `OutOfMemory` exception is thrown for the first time, there is a free location on the heap to allocate it. Note that we do not consider garbage collection.

An *object* is either a class instance, modeled as a pair of its class name and a table mapping pairs of a field name and the defining class to values, or an array, modeled as a pair of its component type and a table mapping integers to values:

$$\begin{aligned}
\mathit{fields} &= (\mathit{ename} \times \mathit{ref_ty},\ \mathit{val})\mathit{table}\\
\mathit{components} &= (\mathit{int}\ ,\ \mathit{val})\mathit{table}\\
\mathit{obj} &= \mathsf{Obj}\ \mathit{tname}\ \mathit{fields}\\
&\ \ |\ \mathsf{Arr}\ \mathit{ty}\ \mathit{components}
\end{aligned}$$

$$\begin{aligned}
&\mathsf{the_Obj} :: (\mathit{obj})\mathit{option} \Rightarrow \mathit{tname} \times \mathit{fields}\\
&\mathsf{the_Arr} :: (\mathit{obj})\mathit{option} \Rightarrow \mathit{ty} \times \mathit{components}\\
&\mathsf{obj_ty} :: \mathit{obj} \Rightarrow \mathit{ty}
\end{aligned}$$

$$\begin{aligned}
&\mathsf{the_Obj}\ (\mathsf{Some}\ (\mathsf{Obj}\ C\ \mathit{fs})) = (C,\mathit{fs})\\
&\mathsf{the_Arr}\ (\mathsf{Some}\ (\mathsf{Arr}\ T\ \mathit{cs})) = (T,\mathit{cs})\\
&\mathsf{obj_ty}\ (\mathsf{Obj}\ C\ \mathit{fs}) = \mathsf{Class}\ C\\
&\mathsf{obj_ty}\ (\mathsf{Arr}\ T\ \mathit{cs}) = T[]
\end{aligned}$$

Using obj_ty we define the predicate $\Gamma,\sigma \vdash v$ fits T, meaning that in the context of Γ and state σ, the value v is assignable to a variable of type T. This proposition, which is computed at run-time for type casts and array assignments, is a weaker version of the notion of conformance introduced in §5.3.

$$\begin{aligned}
&_\ ,_ \vdash _\ \mathsf{fits}\ _ :: \mathit{prog} \Rightarrow \mathit{st} \Rightarrow \mathit{val} \Rightarrow \mathit{ty} \Rightarrow \mathit{bool}\\
&\Gamma,\sigma \vdash v\ \mathsf{fits}\ T \stackrel{\mathsf{def}}{=} (\exists \mathit{pt}.\ T = \mathsf{PrimT}\ \mathit{pt}) \vee v = \mathsf{Null} \vee\\
&\qquad\qquad \Gamma \vdash \mathsf{obj_ty}\ (\mathsf{the}\ (\mathsf{heap}\ \sigma\ (\mathsf{the_Addr}\ v))) \preceq T
\end{aligned}$$

There is a number of auxiliary functions for constructing and updating the state, namely:

$$\begin{aligned}
&\mathsf{lupd}[_\mapsto_]\ _ :: \mathit{ename} \Rightarrow \mathit{val} \Rightarrow \mathit{st} \Rightarrow \mathit{st}\\
&\mathsf{hupd}[_\mapsto_]\ _ :: \mathit{loc} \Rightarrow \mathit{obj} \Rightarrow \mathit{st} \Rightarrow \mathit{st}\\
&\mathsf{x_case} :: \mathit{xcpt}\ \mathit{option} \Rightarrow \mathit{st} \Rightarrow \mathit{st} \Rightarrow \mathit{state}\\
&\mathsf{lupd}[v\mapsto x\ ]\ (h,l) \stackrel{\mathsf{def}}{=} (h,l[v\mapsto x])\\
&\mathsf{hupd}[a\mapsto \mathit{obj}]\ (h,l) \stackrel{\mathsf{def}}{=} (h[a\mapsto \mathit{obj}],l)\\
&\mathsf{x_case}\quad x\ \sigma'\ \sigma \stackrel{\mathsf{def}}{=} (x,\ \mathsf{if}\ x = \mathsf{None}\ \sigma'\ \mathsf{else}\ \sigma)
\end{aligned}$$

$$
\begin{array}{ll}
\mathsf{init\_vars} & :: (\alpha \times ty)list \Rightarrow (\alpha, val)table \\
\mathsf{init\_Obj} & :: prog \Rightarrow tname \Rightarrow obj \\
\mathsf{init\_Arr} & :: ty \Rightarrow int \Rightarrow obj \\
\mathsf{init\_vars} & \stackrel{\mathsf{def}}{=} \mathsf{table\_of} \circ \mathsf{map}\ (\lambda(n,T).\ (n, \mathsf{default\_val}\ T)) \\
\mathsf{init\_Obj}\ \Gamma\ C & \stackrel{\mathsf{def}}{=} \mathsf{Obj}\ C\ (\mathsf{init\_vars}\ (\mathsf{fields}\ \Gamma\ C)) \\
\mathsf{init\_Arr}\ T\ i & \stackrel{\mathsf{def}}{=} \mathsf{Arr}\ T\ (\lambda j.\ \mathsf{if}\ 0 \leq j \wedge j < i\ \mathsf{then}\ \mathsf{Some}\ (\mathsf{default\_val}\ T)\ \mathsf{else}\ \mathsf{None}) \\
\mathsf{raise\_if} & :: bool \Rightarrow xname \Rightarrow (xcpt)option \Rightarrow (xcpt)option \\
\mathsf{np} & :: val \Rightarrow (xcpt)option \Rightarrow (xcpt)option \\
\mathsf{raise\_if}\ c\ xn\ xo & \stackrel{\mathsf{def}}{=} \mathsf{if}\ c \wedge (xo = \mathsf{None})\ \mathsf{then}\ \mathsf{Some}\ (\mathsf{SysXcpt}\ xn)\ \mathsf{else}\ xo \\
\mathsf{np}\ v & \stackrel{\mathsf{def}}{=} \mathsf{raise\_if}\ (v = \mathsf{Null})\ \texttt{NullPointer}
\end{array}
$$

The definition of raise_if deserves a comment: raise_if c xn xo either propagates an already thrown exception xo or raises the system exception xn if c is true. As an application, np v checks for a null pointer access through the value v and throws a `NullPointer` exception in this case, but any other exception that has already occurred takes precedence.

Evaluation Rule Format. Internally, the evaluation rules are given as mutually inductive sets of tuples. These sets define relations, which we present as predicates of the following form.

- $\Gamma \vdash \sigma\ -c\rightarrow\ \sigma' :: prog \Rightarrow state \Rightarrow stmt \Rightarrow state \Rightarrow bool$
  means that the execution of statement c transforms state σ into σ'.
- $\Gamma \vdash \sigma\ -e\triangleright v\rightarrow\ \sigma' :: prog \Rightarrow state \Rightarrow expr \Rightarrow val \Rightarrow state \Rightarrow bool$
  means that expression e evaluates to v, transforming σ into σ'.

Although defined as relations (for technical reasons), the semantics given below can be shown to be functional, i.e. deterministic.

Strictly speaking it is not necessary to include an exception in the start state of a computation. Similarly, an expression needs only return either a value or an exception, but not both. However, the symmetry achieved by our slightly redundant model simplifies the rules considerably. In particular, we can avoid case distinctions on whether exceptions occur in intermediate states, which would cause the rules to be split. Suppose for example that $\Gamma \vdash \sigma\ -c\rightarrow\ \sigma'$ had the signature $prog \Rightarrow st \Rightarrow stmt \Rightarrow state \Rightarrow bool$, i.e. all rules assume that there is no exception in the start state. Then the rule(s) for sequential composition would look like

$$\frac{\Gamma \vdash \sigma_0\ -c_1\ \rightarrow (\mathsf{None}, \sigma_1);\ \ \Gamma \vdash \sigma_1\ -c_2\rightarrow\ \sigma_2}{\Gamma \vdash \sigma_0\ -c_1\,;\ c_2\rightarrow\ \sigma_2}$$

$$\frac{\Gamma \vdash \sigma_0\ -c_1\ \rightarrow (\mathsf{Some}\ xs, \sigma_1)}{\Gamma \vdash \sigma_0\ -c_1\,;\ c_2\rightarrow (\mathsf{Some}\ xs, \sigma_1)}$$

As a consequence of the design decisions just mentioned, there is exactly one rule for each syntactic construct. Additionally there are general rules defining that exceptions simply propagate when a series of statements is executed or a series of expressions is evaluated:

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

$$\frac{}{\Gamma \vdash (\mathsf{Some}\ xc,\sigma)\ -e\triangleright\mathsf{arbitrary}\rightarrow\ (\mathsf{Some}\ xc,\sigma)}$$

All other rules can assume that in their concerning initial state no exception has been thrown. For such states, we define the abbreviation $\mathsf{Norm}\ \sigma$, which stands for (None,σ).

Execution of Statements. The rules for the statements not explicitly involving exceptions are obvious:

$$\frac{}{\Gamma \vdash \mathsf{Norm}\ \sigma\ -\mathsf{Skip}\rightarrow\ \mathsf{Norm}\ \sigma} \qquad \frac{\Gamma \vdash \mathsf{Norm}\ \sigma_0\ -c_1\ \rightarrow\ \sigma_1;\ \ \Gamma \vdash \sigma_1\ -c_2\rightarrow\ \sigma_2}{\Gamma \vdash \mathsf{Norm}\ \sigma_0\ -c_1\,;\ c_2\rightarrow\ \sigma_2}$$

$$\frac{\Gamma \vdash \mathsf{Norm}\ \sigma_0\ -e\triangleright v\rightarrow\ \sigma_1}{\Gamma \vdash \mathsf{Norm}\ \sigma_0\ -\mathsf{Expr}\ e\rightarrow\ \sigma_1} \qquad \frac{\begin{array}{c}\Gamma \vdash \mathsf{Norm}\ \sigma_0\ -e\triangleright v\rightarrow\ \sigma_1;\\ \Gamma \vdash \sigma_1\ -\mathsf{if\ the\_Bool}\ v\ \mathsf{then}\ c_1\ \mathsf{else}\ c_2\rightarrow\ \sigma_2\end{array}}{\Gamma \vdash \mathsf{Norm}\ \sigma_0\ -\mathtt{if}(e)\ c_1\ \mathtt{else}\ c_2\rightarrow\ \sigma_2}$$

$$\frac{\Gamma \vdash \mathsf{Norm}\ \sigma_0\ -\mathtt{if}(e)\ (c;\ \mathtt{while}(e)\ c)\ \mathtt{else\ Skip}\rightarrow\ \sigma_1}{\Gamma \vdash \mathsf{Norm}\ \sigma_0\ -\mathtt{while}(e)\ c\rightarrow\ \sigma_1}$$

If no other exceptions have occurred while evaluating its argument and testing for a null reference (using `np`), the `throw` statement copies the evaluated location into the exception component of the state:

$$\frac{\begin{array}{c}\Gamma \vdash \mathsf{Norm}\ \sigma_0\ -e\triangleright a'\rightarrow\ (x_1,\sigma_1);\ \ {x_1}' = \mathsf{np}\ a'\ x_1;\\ {x_1}''=(\mathsf{if}\ {x_1}'=\mathsf{None\ then}\ (\mathsf{Some}\ (\mathsf{XcptLoc}\ (\mathsf{the\_Addr}\ a'))))\ \mathsf{else}\ {x_1}')\end{array}}{\Gamma \vdash \mathsf{Norm}\ \sigma_0\ -\mathtt{throw}\ e\rightarrow\ ({x_1}'',\sigma_1)}$$

For the semantics of the `try _ catch _` statement we have to distinguish whether some exception is thrown and then caught by the `catch` clause or not. In the first case, i.e. there is an exception of appropriate dynamic type to be handled, the `catch` clause is executed with its exception parameter set to the caught exception. In the second case the `catch` clause is skipped. Because of technical limitations of the inductive definition package of Isabelle/HOL, even in this case we have to provide an occurrence of the execution relation, which in effect simply sets σ_2 to $({x_1}',{\sigma_1}')$.

$$\frac{\begin{array}{c}\Gamma \vdash \mathsf{Norm}\ \sigma_0\ -c_1\rightarrow\ \sigma_1;\ \ \Gamma \vdash \sigma_1\ -\mathsf{salloc}\rightarrow\ ({x_1}',{\sigma_1}');\\ \mathsf{case}\ {x_1}'\ \mathsf{of\ None}\quad\ \Rightarrow {\sigma_1}'' = ({x_1}',{\sigma_1}') \wedge {c_2}' = \mathsf{Skip}\\ \quad\mid \mathsf{Some}\ xc \Rightarrow \mathsf{let}\ a = \mathsf{Addr}\ (\mathsf{the\_XcptLoc}\ xc)\ \mathsf{in}\\ \qquad\qquad\qquad\ \mathsf{if}\ \Gamma,{\sigma_1}' \vdash a\ \mathsf{fits\ Class}\ tn\\ \qquad\qquad\qquad\ \mathsf{then}\ {\sigma_1}'' = \mathsf{Norm}\ (\mathsf{lupd}[vn\mapsto a]{\sigma_1}') \wedge {c_2}' = c_2\\ \qquad\qquad\qquad\ \mathsf{else}\ \ {\sigma_1}'' = ({x_1}',{\sigma_1}') \wedge {c_2}' = \mathsf{Skip};\\ \Gamma \vdash {\sigma_1}''\ -{c_2}'\rightarrow\ \sigma_2\end{array}}{\Gamma \vdash \mathsf{Norm}\ \sigma_0\ -(\mathtt{try}\ c_1\ \mathtt{catch}(tn\ vn)\ c_2)\rightarrow\ \sigma_2}$$

On the one hand, the exception parameter of the `catch` clause must represent the exception thrown in the `try` block by a reference to its exception object. As on the other hand we defer the allocation of system exceptions when evaluating expressions, we have to ensure that even for such exceptions a suitable exception object is allocated on the heap of σ_1', replacing the SysXcpt entry by an XcptLoc entry in x_1'. This is achieved by the auxiliary relation $\Gamma \vdash \sigma\ {-}\mathsf{salloc}{\rightarrow}\ \sigma' :: \mathit{prog} \Rightarrow \mathit{state} \Rightarrow \mathit{state} \Rightarrow \mathit{bool}$. If no system exception has been thrown, the relation behaves like the identity on the state, and otherwise allocates an exception object and modifies the state accordingly. Note that this allocation step is impossible — and therefore program execution halts — if there is no free address left.

$$\frac{}{\Gamma \vdash \mathsf{Norm}\ \sigma\ {-}\mathsf{salloc}{\rightarrow}\ \mathsf{Norm}\ \sigma}$$

$$\frac{}{\Gamma \vdash (\mathsf{Some}\ (\mathsf{XcptLoc}\ a),\sigma)\ {-}\mathsf{salloc}{\rightarrow}\ (\mathsf{Some}\ (\mathsf{XcptLoc}\ a),\sigma)}$$

$$\frac{\begin{array}{c}\mathsf{new\_Addr}\ (\mathsf{heap}\ \sigma) = \mathsf{Some}\ (a,x);\\ xobj = \mathsf{init\_Obj}\ \Gamma\ (\mathsf{SXcpt}\ (\mathsf{if}\ x = \mathsf{None}\ \mathsf{then}\ xn\ \mathsf{else}\ \texttt{OutOfMemory}))\end{array}}{\Gamma \vdash (\mathsf{Some}\ (\mathsf{SysXcpt}\ xn),\sigma)\ {-}\mathsf{salloc}{\rightarrow}\ (\mathsf{Some}\ (\mathsf{XcptLoc}\ a),\mathsf{hupd}[a{\mapsto}xobj]\sigma)}$$

The `finally` statement is similar to the sequential composition, but executes its second clause regardless whether an exception has been thrown in its first clause or not. If an exception occurs in either clause, it is (re-)raised after the statement, and if both parts throw an exception, the first one takes precedence.

$$\frac{\begin{array}{c}\Gamma \vdash \mathsf{Norm}\ \sigma_0\ {-}c_1\ {\rightarrow}\ (x_1,\sigma_1);\\ \Gamma \vdash \mathsf{Norm}\ \sigma_1\ {-}c_2\ {\rightarrow}\ (x_2,\sigma_2);\\ x_2' = (\mathsf{if}\ x_1 \neq \mathsf{None} \wedge x_2 = \mathsf{None}\ \mathsf{then}\ x_1\ \mathsf{else}\ x_2)\end{array}}{\Gamma \vdash \mathsf{Norm}\ \sigma_0\ {-}(c_1\ \texttt{finally}\ c_2){\rightarrow}\ (x_2',\sigma_2)}$$

Evaluation of Expressions. In contrast to the statement rules, almost all evaluation rules for expressions deserve some comments.

Creating a new class instance means picking a free address a and updating the heap at that address with an object, the fields of which are initialized with default values according to their types. Note that the rule is not applicable — and therefore execution halts — if new_Addr fails.

$$\frac{\mathsf{new\_Addr}\ (\mathsf{heap}\ \sigma) = \mathsf{Some}\ (a,x)}{\Gamma \vdash \mathsf{Norm}\ \sigma\ {-}\texttt{new}\ C{\rhd}\mathsf{Addr}\ a{\rightarrow}\ \mathsf{x\_case}\ x\ (\mathsf{hupd}[a{\mapsto}\mathsf{init\_Obj}\ \Gamma\ C]\sigma)\ \sigma}$$

The same applies for the creation of a new array, where additionally an exception is raised if the length of the array is negative:

$$\frac{\begin{array}{c}\Gamma \vdash \mathsf{Norm}\ \sigma_0\ {-}e{\rhd}i'{\rightarrow}\ (x_1,\sigma_1);\ \ i = \mathsf{the\_Intg}\ i';\\ \mathsf{new\_Addr}\ (\mathsf{heap}\ \sigma_1) = \mathsf{Some}\ (a,x);\\ x_1' = \mathsf{raise\_if}\ (i{<}0)\ \texttt{NegArrSize}\ (\mathsf{if}\ x_1 = \mathsf{None}\ \mathsf{then}\ x\ \mathsf{else}\ x_1)\end{array}}{\Gamma \vdash \mathsf{Norm}\ \sigma_0\ {-}\texttt{new}\ T[e]{\rhd}\mathsf{Addr}\ a{\rightarrow}\ \mathsf{x\_case}\ x_1'\ (\mathsf{hupd}[a{\mapsto}\mathsf{init\_Arr}\ T\ i]\sigma_1)\ \sigma_1}$$

A type cast merely returns its argument value, but raises an exception if the dynamic type happens to be unsuitable:

$$\frac{\Gamma \vdash \mathsf{Norm}\ \sigma_0\ -e\rhd v\rightarrow\ (x_1,\sigma_1);\quad x_1' = \mathsf{raise\_if}(\neg\ \Gamma,\sigma_1 \vdash v\ \mathsf{fits}\ T)\ \mathtt{ClassCast}\ x_1}{\Gamma \vdash \mathsf{Norm}\ \sigma_0\ -(T)e\rhd v\rightarrow\ (x_1',\sigma_1)}$$

The type comparison operator checks if the type of its argument is assignable to the given reference type:

$$\frac{\Gamma \vdash \mathsf{Norm}\ \sigma_0\ -e\rhd v\rightarrow\ \sigma_1;\quad b = (v{\neq}\mathsf{Null} \wedge \Gamma,\mathsf{snd}\ \sigma_1 \vdash v\ \mathsf{fits}\ \mathsf{RefT}\ T)}{\Gamma \vdash \mathsf{Norm}\ \sigma_0\ -e\ \mathtt{instanceof}\ T\rhd\mathsf{Bool}\ b\rightarrow\ \sigma_1}$$

The result of a literal expression is simply the given value:

$$\frac{}{\Gamma \vdash \mathsf{Norm}\ \sigma\ -\mathsf{Lit}\ v\rhd v\rightarrow\ \mathsf{Norm}\ \sigma}$$

An access to a local variable (or the `this` pointer) reads from the local state component:

$$\frac{}{\Gamma \vdash \mathsf{Norm}\ \sigma\ -vn\rhd\mathsf{the}\ (\mathsf{locals}\ \sigma\ vn)\rightarrow\ \mathsf{Norm}\ \sigma}$$

An assignment to a local variable updates the state, unless the evaluation of the subexpression raises an exception:

$$\frac{\Gamma \vdash \mathsf{Norm}\ \sigma_0\ -e\rhd v\rightarrow\ \ (x,\sigma_1);\quad \sigma_1' = (\mathsf{if}\ x = \mathsf{None}\ \mathsf{then}\ \mathsf{lupd}[vn{\mapsto}v]\ \sigma_1\ \mathsf{else}\ \sigma_1)}{\Gamma \vdash \mathsf{Norm}\ \sigma_0\ -vn{:=}e\rhd v\rightarrow\ (x,\sigma_1')}$$

A field access reads from a field of the given object, taking into account the type annotation which yields the defining class of the field as determined statically. It also checks for null pointer access.

$$\frac{\Gamma \vdash \mathsf{Norm}\ \sigma_0\ -e\rhd a'\rightarrow\ (x_1,\sigma_1);\quad v = \mathsf{the}\ (\mathsf{snd}\ (\mathsf{the\_Obj}\ (\mathsf{heap}\ \sigma_1\ (\mathsf{the\_Addr}\ a')))\ (fn,T))}{\Gamma \vdash \mathsf{Norm}\ \sigma_0\ -\{T\}e.fn\rhd v\rightarrow\ (\mathsf{np}\ a'\ x_1,\sigma_1)}$$

A field assignment acts accordingly:

$$\frac{\Gamma \vdash \mathsf{Norm}\ \sigma_0\ -e_1\rhd a'\rightarrow\ (x_1,\sigma_1);\quad a = \mathsf{the\_Addr}\ a';\quad \Gamma \vdash (\mathsf{np}\ a'\ x_1,\sigma_1)\ -e_2\rhd v\rightarrow\ (x_2,\sigma_2);\quad (c,fs) = \mathsf{the\_Obj}\ (\mathsf{heap}\ \sigma_2\ a);\quad obj = \mathsf{Obj}\ c\ (fs[(fn,T){:=}v])}{\Gamma \vdash \mathsf{Norm}\ \sigma_0\ -(\{T\}e_1.fn{:=}e_2)\rhd v\rightarrow\ \mathsf{x\_case}\ x_2\ (\mathsf{hupd}[a{\mapsto}obj]\sigma_2)\ \sigma_2}$$

An array access reads a component from the given array, but raises an exception if the index is invalid:

$$\frac{\Gamma \vdash \mathsf{Norm}\ \sigma_0\ -e_1\rhd a'\rightarrow\ \sigma_1;\quad \Gamma \vdash \sigma_1\ -e_2\rhd i'\rightarrow\ (x_2,\sigma_2);\quad vo = \mathsf{snd}\ (\mathsf{the\_Arr}\ (\mathsf{heap}\ \sigma_2\ (\mathsf{the\_Addr}\ a')))\ (\mathsf{the\_Intg}\ i');\quad x_2' = \mathsf{raise\_if}\ (vo = \mathsf{None})\ \mathtt{IndOutBound}\ (\mathsf{np}\ a'\ x_2)}{\Gamma \vdash \mathsf{Norm}\ \sigma_0\ -e_1[e_2]\rhd\mathsf{the}\ vo\rightarrow\ (x_2',\sigma_2)}$$

Similarly, an array assignment updates the appropriate component, but first has to check the type of the value to be assigned. Note one subtle difference to field assignment: null pointer access is checked after evaluating the right-hand side, whereas in field assignment the check occurs immediately after calculating the reference.

$$\frac{\begin{array}{c}\Gamma \vdash \mathsf{Norm}\ \sigma_0\ -e_1 \rhd a' \rightarrow\ \sigma_1;\ \ a = \mathsf{the\_Addr}\ a';\\ \Gamma \vdash \sigma_1\ -e_2 \rhd i' \rightarrow\ \sigma_2;\ \ i = \mathsf{the\_Intg}\ i';\\ \Gamma \vdash \sigma_2\ -e_3 \rhd v \rightarrow\ (x_3,\sigma_3);\\ (T,cs) = \mathsf{the\_Arr}\ (\mathsf{heap}\ \sigma_3\ a);\ \ obj = \mathsf{Arr}\ T\ (cs[i \mapsto v]);\\ x_3{}' = \mathsf{raise\_if}\ (\neg\ \Gamma,\sigma_3 \vdash v\ \mathsf{fits}\ T)\ \mathtt{ArrStore}\ (\\ \mathsf{raise\_if}\ (cs\ i = \mathsf{None})\ \mathtt{IndOutBound}\ (\mathsf{np}\ a'\ x_3))\end{array}}{\Gamma \vdash \mathsf{Norm}\ \sigma_0\ -(e_1[e_2]\mathtt{:=}e_3) \rhd v \rightarrow\ \mathsf{x\_case}\ x_3{}'\ (\mathsf{hupd}[a \mapsto obj]\sigma_3)\ \sigma_3}$$

The most complex rule is the one for method invocation: after evaluating e to the target location a' and p to the parameter value pv, the block blk and the result expression res of method mn with argument type T are extracted from the program Γ (using the dynamic type $dynT$ of the object stored at a'). For simplicity, we require local variables to be initialized with default values, as the expensive rules for "definite assignment" [GJS96, Ch. 16] merely enable the run-time optimization that variables need not be initialized before being explicitly assigned to. After executing blk and res in the new invocation frame built from the local variables, the parameter pv and a' as the value of `this`, the old invocation frame is restored and the result value v returned:

$$\frac{\begin{array}{c}\Gamma \vdash \mathsf{Norm}\ \sigma_0\ -e \rhd a' \rightarrow\ \sigma_1;\\ \Gamma \vdash \sigma_1\ -p \rhd pv \rightarrow\ (x_2,\sigma_2);\\ dynT = \mathsf{fst}\ (\mathsf{the\_Obj}\ (\mathsf{heap}\ \sigma_2\ (\mathsf{the\_Addr}\ a')));\\ (md,(pn,rT),lvars,blk,res) = \mathsf{the}\ (\mathsf{cmethd}\ \Gamma\ dynT\ (mn,pT));\\ \Gamma \vdash (\mathsf{np}\ a'\ x_2,(\mathsf{heap}\ \sigma_2,\mathsf{init\_vars}\ lvars[\mathtt{this} \mapsto a'][pn \mapsto pv]))\ -blk \rightarrow\ \sigma_3;\\ \Gamma \vdash \sigma_3\ -res \rhd v \rightarrow\ (x_4,\sigma_4)\end{array}}{\Gamma \vdash \mathsf{Norm}\ \sigma_0\ -(e.mn(\{pT\}p)) \rhd v \rightarrow\ (x_4,(\mathsf{heap}\ \sigma_4,\mathsf{locals}\ \sigma_2))}$$

Note that all rules are defined carefully in order to be applicable even in not type-correct situations. For example, in any context where a value v is expected to be an address, we do not use a premise like $v = \mathsf{Addr}\ a$ as this will disable the rule if v happens to be, for example, a null pointer or a Boolean value. Instead, we use an expression like $a = \mathsf{the\_Addr}\ v$, which will yield an arbitrary value if v is not an address, yet will leave the rule applicable. In such cases we could not prove anything useful about a, but during the type soundness proof itself it emerges that for well-formed programs (and statically well-typed statements and expressions) such situations cannot occur. A "defensive" evaluation throwing some artificial exception in case of type mismatches, which would require additional overhead, is therefore not necessary.

Below we give a derivation for the execution of our example code *test*, under the assumptions new_Addr empty = Some (a, None) and $\forall obj.$ new_Addr (empty[a↦obj]) = Some (b, None), which guarantee that there are at least two free locations on the heap.

We use the following abbreviations:

Γ = $tprg$
NP = `NullPointer`
blk = Expr({ClassT Ext}(Class Ext)x.vee:=Lit (Intg 1))
$obj1$ = Obj Ext (empty[(vee, ClassT $Base$)↦Bool False] [(vee, ClassT Ext)↦Intg 0])
$obj2$ = Obj (SXcpt NP) empty
h = empty[a↦$obj1$]
l = empty[e↦Addr a]

Given only the start state σ_0, all other states are computed during the derivation, which results in the sequence

σ_0 = Norm (h, empty)
σ_1 = Norm (h, l)
σ_2 = Norm (h, empty[this↦Addr a][x↦Null])
σ_3 = (Some (SysXcpt NP), h, empty[this↦Addr a][x↦Null])
σ_4 = (Some (SysXcpt NP), h, l)
σ_5 = (Some (XcptLoc b), h[b↦$obj2$], l)
σ_6 = Norm (h[b↦$obj2$], l[x↦Addr b])
σ_7 = (Some (XcptLoc b), h[b↦$obj2$], l[x↦Addr b])

$$\dfrac{\dfrac{\dfrac{\Gamma \vdash \sigma_2 \ -x\triangleright\text{Null}\rightarrow \sigma_2 \quad \Gamma,\text{snd } \sigma_2 \vdash \text{Null fits Class } Ext}{\Gamma \vdash \sigma_2 \ -(\text{Class } Ext)x\triangleright\text{Null}\rightarrow \sigma_2} \quad \sigma_3 = (\text{Some (SysXcpt } NP), \text{ snd } \sigma_2) \quad \Gamma \vdash \sigma_3 \ -\text{Lit (Intg 1)}\triangleright\_\rightarrow \sigma_3}{\Gamma \vdash \sigma_2 \ -(\{\text{ClassT } Ext\}(\text{Class } Ext)x.\, vee\text{:=Lit (Intg 1)})\triangleright\_\rightarrow \sigma_3}}{\Gamma \vdash \sigma_2 \ -\text{Expr}(\{\text{ClassT } Ext\}(\text{Class } Ext)x.\, vee\text{:=Lit (Intg 1)}) \ \rightarrow \sigma_3}\ (Blk)$$

$$\dfrac{\dfrac{\Gamma \vdash \sigma_1 \ -e\triangleright\text{Addr } a\rightarrow \sigma_1 \quad \text{cmethd } \Gamma\ Ext\ (foo, \text{Class } Base) = \text{Some } (\_, (x, \_), [], blk, \text{Lit Null}) \quad \sigma_2 = \text{Norm } (h, \text{empty}[\text{this}\mapsto\text{Addr } a][x\mapsto\text{Null}]) \quad (Blk) \quad \Gamma \vdash \sigma_3 \ -\text{Lit Null}\triangleright\_\rightarrow \sigma_3}{\Gamma \vdash \sigma_1 \ -(e.foo(\{\text{Class } Base\}\text{Lit Null}))\triangleright\_\rightarrow \sigma_4}}{\Gamma \vdash \sigma_1 \ -\text{Expr}(e.foo(\{\text{Class } Base\}\text{Lit Null})) \ \rightarrow \sigma_4}\ (Call')$$

$$\dfrac{\dfrac{\dfrac{\text{new\_Addr empty} = \text{Some } (a, \text{None})}{\Gamma \vdash \sigma_0 \ -\texttt{new } Ext\triangleright a\rightarrow \text{Norm } (h, \text{empty})} \quad \text{new\_Addr } h = \text{Some } (b, \text{None})}{\Gamma \vdash \sigma_0 \ -(e\text{:=}\texttt{new } Ext)\triangleright\_\rightarrow \sigma_1} \quad (Call') \quad \dfrac{\Gamma \vdash \sigma_4 \ -\text{salloc}\rightarrow \sigma_5 \quad \Gamma,\text{snd } \sigma_5 \vdash \text{Addr } b \text{ fits Class (SXcpt } NP) \quad (Throw)}{\Gamma \vdash \sigma_1 \ -(\texttt{try } \text{Expr}(e.foo(\{\text{Class } Base\}\text{Lit Null})) \ \texttt{catch}((\text{SXcpt } NP)\ x)\ (\texttt{throw } x))\rightarrow \sigma_7}}{\Gamma \vdash \sigma_0 \ -(\text{Expr}(e\text{:=}\texttt{new } Ext); \texttt{ try } \text{Expr}(e.foo(\{\text{Class } Base\}\text{Lit Null})) \ \texttt{catch}((\text{SXcpt } NP)\ x)\ (\texttt{throw } x))\rightarrow \sigma_7}$$

$$\dfrac{\Gamma \vdash \sigma_6 \ -x\triangleright\text{Addr } b\rightarrow \sigma_6 \quad \sigma_7 = (\text{Some (XcptLoc } b), \text{ snd } \sigma_6)}{\Gamma \vdash \sigma_6 \ -\texttt{throw } x\rightarrow \sigma_7}\ (Throw)$$

5 The Proof of Type Soundness

In this section we discuss our type soundness theorem together with its crucial lemmas. As we spent almost half of the proof effort deriving properties of the type relations and the structure of well-formed programs, we dedicate to them subsections of their own before introducing helpful notions concerning type soundness, the main theorem itself, and interesting corollaries.

It is not surprising that many of them are similar to those given by Drossopoulou and Eisenbach [DE98] since the necessity of certain lemmas emerges quite naturally. On the other hand, the proof principles we use are sometimes rather different from those outlined in their earlier paper [DE97], some of which were inadequate.

5.1 Lemmas on the Type Relations

There are two non-trivial lemmas concerning the type relations of Bali, namely the well-foundedness wf of the converse subinterface and subclass relations

$$\begin{aligned}\mathsf{wf\_prog}\ \Gamma \longrightarrow\ & \mathsf{wf}\ (\lambda(J,I).\ \Gamma \vdash I \prec_{\mathsf{i}} J) \\ \wedge\ & \mathsf{wf}\ (\lambda(D,C).\ \Gamma \vdash C \prec_{\mathsf{c}} D)\end{aligned}$$

and the frequently used transitivity of the widening relation:

$$\mathsf{wf\_prog}\ \Gamma \wedge \Gamma \vdash S \preceq U \wedge \Gamma \vdash U \preceq T \longrightarrow \Gamma \vdash S \preceq T$$

The two relations are well-founded because they are finite and acyclic, where the former is a consequence of representing class and interface declarations as lists, and the latter follows from the irreflexivity of the relations, which in turn follows from the well-formedness of the classes and interfaces implied by the well-formedness of the whole program.

The well-foundedness facts are necessary for deriving the recursion equations for the functions that traverse the type hierarchy of a program (see §4.1) and also give rise to induction principles for the (direct) subinterface and subclass relations, e.g. the rule

$$\frac{\mathsf{wf\_prog}\ \Gamma;\ P\ \texttt{Object};\quad \forall C\ D.\ C \neq \texttt{Object} \wedge \Gamma \vdash C \prec^1_{\mathsf{c}} D \wedge \ldots \wedge P\ D \longrightarrow P\ C}{\forall E.\ \mathsf{is\_class}\ \Gamma\ E \longrightarrow P\ E}$$

means that for a well-formed program, if some property hold for class `Object` and is preserved by the direct subclass relation, it holds for all classes.

Most lemmas, as well as auxiliary properties for deriving them, typically rely on several well-formedness conditions and are usually proved by rule induction on the type relation involved, or by applying the induction principles just mentioned. For example, the transitivity of $\_ \vdash \_ \preceq \_$ is proved by rule induction on the widening relation. It requires a well-formed program because it uses the properties that every class widens to `Object` and that `Object` has neither a superclass nor a superinterface.

5.2 Lemmas on Fields and Methods

For the type-safety of field accesses and method calls, characteristic lemmas concerning the field lookup and method lookup are required. They are used to relate the (static) types of fields and methods, as determined at compile-time, to the actual (dynamic) types that occur at run-time.

For example, fields correctly referred to at compile-time must be found at run-time. More formally, if a field access $\{T\}e.fn$, where e is of type Class C, statically refers to a field of type fT defined in the reference type T, then within an instance of some class C', which may be a subclass of C, the field can be (dynamically) referred to using the same name and its defining class. In particular, there is no dynamic binding for fields. This fact requires the following lemma:

$$\mathsf{wf\_prog}\ \Gamma \wedge \mathsf{cfield}\ \Gamma\ C\ fn = \mathsf{Some}\ (T,fT) \wedge \Gamma \vdash \mathsf{Class}\ C' \preceq \mathsf{Class}\ C \longrightarrow \mathsf{table\_of}\ (\mathsf{fields}\ \Gamma\ C')\ (fn,T) = \mathsf{Some}\ fT$$

Concerning method calls, a similar requirement preventing 'method not understood' errors can be formalized: if a method call of the form $e.mn(\{pT\}p)$ with $E \vdash e::\mathsf{RefT}\ T$ refers to a method that is statically available for the reference e, the dynamic lookup of the object pointed at by e should yield a method with a compatible result type. The lemma that helps to establish this behavior reads as follows: for a well-formed program, a reference type T, and any class type T_1 that widens to T, if T (statically) supports a method with a given signature, then the (dynamic) type T_1 supports a method with the same signature and whose result type widens to the result type of the first method:

$$\mathsf{wf\_prog}\ \Gamma \wedge (m_1,(pn_1,rT_1)) \in \mathsf{mheads}\ \Gamma\ T\ sig \wedge \Gamma \vdash \mathsf{Class}\ T_1 \preceq \mathsf{RefT}\ T \longrightarrow \exists m_2\ pn_2\ rT_2\ b.\ \mathsf{cmethd}\ \Gamma\ T_1\ sig = \mathsf{Some}\ (m_2,(pn_2,rT_2),b) \wedge \Gamma \vdash rT_2 \preceq rT_1$$

The proofs of these lemmas are lengthy and require many auxiliary theorems that are proved by induction on the direct subclass relation, by case splitting on the right-hand argument of the widening relation and by rule induction on the subinterface, subclass, and implementation relation.

5.3 Type Soundness

Finally, we state and prove the type soundness theorem. We motivate how we express type soundness, comment on the proof of the main theorem, and discuss it consequences.

Goal. Type soundness is a relation between the type system and the semantics of a language meaning that all values produced during any program execution respect their static types. This can be formulated as a preservation property: For all state transformations caused by executing a statement or evaluating an expression, if in the original state the contents of all variables "conform" to their respective types, this holds also for any final state. Additionally, if an expression yields some result, this value "conforms" to the type of the expression. Of course, we can only expect all this to hold if we assume a well-formed program and well-typed statements and expressions.

It remains to specify what we mean exactly by 'conforms', which is inspired by [DE98]. Relative to a given program Γ and a state σ, a value v conforms to a type T, written $\Gamma,\sigma \vdash v::T$, iff the dynamic type of v widens to T. Via two auxiliary conformance concepts, this can be lifted to the notion of a whole state σ conforming to an environment E. The proposition $\sigma ::\preceq E$ means that the value of any accessible variable within the state is compatible with its static type. Formally, these four concepts

- $\_,\_ \vdash \_ ::\preceq \_ :: prog \Rightarrow st \Rightarrow val \Rightarrow ty \Rightarrow bool$
  of a value conforming to a type,
- $\_,\_ \vdash \_[::\preceq]\_ :: prog \Rightarrow st \Rightarrow (\alpha,val)table \Rightarrow (\alpha,ty)table \Rightarrow bool$
  of all values in a table conforming to their respective types,
- $\_,\_ \vdash \_ ::\preceq \diamond :: prog \Rightarrow st \Rightarrow obj \Rightarrow bool$
  of all components of an object conforming to their respective types, and
- $\_ ::\preceq \_ :: state \Rightarrow env \Rightarrow bool$
  of a state conforming to an environment

are defined as follows:

$$\Gamma,\sigma \vdash v\ ::\preceq\ T \ \stackrel{\text{def}}{=}\ \mathsf{let}\ dyn\_ty = \mathsf{option\_map\ obj\_ty} \circ \mathsf{heap}\ \sigma\ \mathsf{in}\ \exists T'.\ \mathsf{typeof}\ dyn\_ty\ v = \mathsf{Some}\ T' \wedge \Gamma \vdash T' \preceq T$$

$$\Gamma,\sigma \vdash vs[::\preceq]Ts \ \stackrel{\text{def}}{=}\ \forall n\ T.\ Ts\ n = \mathsf{Some}\ T \longrightarrow (\exists v.\ vs\ n = \mathsf{Some}\ v \wedge \Gamma,\sigma \vdash v::\preceq T)$$

$$\Gamma,\sigma \vdash \mathsf{Obj}\ C\ fs::\preceq \diamond\ = \Gamma,\sigma \vdash fs[::\preceq]\mathsf{table\_of}\ (\mathsf{fields}\ \Gamma\ C$$

$$\Gamma,\sigma \vdash \mathsf{Arr}\ T\ cs::\preceq \diamond\ = \Gamma,\sigma \vdash cs[::\preceq]\mathsf{option\_map}\ (\lambda i.\ T) \circ cs$$

$$(x,\sigma)::\preceq(\Gamma,\Lambda) \ \stackrel{\text{def}}{=}\ \Gamma,\sigma \vdash \mathsf{locals}\ \sigma[::\preceq]\Lambda \wedge (\forall a\ obj.\ \mathsf{heap}\ \sigma\ a = \mathsf{Some}\ obj \longrightarrow \Gamma,\sigma \vdash obj\ ::\preceq \diamond) \wedge (\forall a.\ x = \mathsf{Some}(\mathsf{XcptLoc}\ a) \longrightarrow \Gamma,\sigma \vdash \mathsf{Addr}\ a::\preceq \mathsf{Class}(\mathsf{SXcpt}\ \texttt{Throwable}))$$

The expression $(\mathsf{option\_map\ obj\_ty} \circ \mathsf{heap}\ \sigma)\ a$ calculates the dynamic type of the object (if any) at address a on the heap. Note that the conformance relation is defined such that it does not take into account inaccessible variables, i.e. values that occur in the state but not in the corresponding component of the static environment. Among others, this frees us from explicitly deallocating exception parameters after a catch clause.

With the help of the notions just introduced, we can express the propositions we aim to prove as follows. In the context of a well-formed program, the execution of a well-typed statement transforms a state conforming to the environment into another state that again conforms to the environment:

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

Analogously, the evaluation of a well-typed expression preserves the conformance of the state to the environment where, unless an exception has occurred, the value of the expression conforms to its static type:

$$E = (\Gamma,\Lambda) \wedge \mathsf{wf\_prog}\ \Gamma \wedge E \vdash e::T \wedge \sigma ::\preceq E \wedge \Gamma \vdash \sigma\ -e\triangleright v\rightarrow\ (x',\sigma') \ \longrightarrow\ (x',\sigma') ::\preceq E \wedge (x' = \mathsf{None} \longrightarrow \Gamma,\sigma' \vdash v::\preceq T)$$

The validity of these two formulas will result as trivial corollaries from the main theorem, given next.

Main Theorem and Proof. To prove the intended type soundness theorems given above, we utilize rule induction on the derivation on the execution of statements and the evaluation of expressions. As these depend on each other, we must deal with statements and expressions simultaneously. In addition, in order to obtain a suitable induction hypothesis, we have to strengthen the propositions by adding the auxiliary "heap extension" predicate $\_ \trianglelefteq \_$ (defined below) and introducing universal quantifications explicitly at several positions. As a result, the main theorem looks quite formidable, yet we attempt to cast it into words:

$$
\begin{array}{l}
\mathsf{wf\_prog}\ \Gamma \longrightarrow \\
\quad (\Gamma \vdash (x,\sigma)\ -c\ \rightarrow (x',\sigma') \longrightarrow \\
\qquad \forall \Lambda.\ (x,\sigma) :: \preceq (\Gamma,\Lambda) \longrightarrow \\
\qquad\quad (\Gamma,\Lambda) \vdash c :: \diamond \longrightarrow \\
\qquad\quad (x',\sigma') :: \preceq (\Gamma,\Lambda) \wedge \sigma \trianglelefteq \sigma' \\
\wedge \\
\quad (\Gamma \vdash (x,\sigma)\ -e \triangleright v \rightarrow (x',\sigma') \longrightarrow \\
\qquad \forall \Lambda.\ (x,\sigma) :: \preceq (\Gamma,\Lambda) \longrightarrow \\
\qquad \forall T.\ (\Gamma,\Lambda) \vdash e :: T \longrightarrow \\
\qquad\quad (x',\sigma') :: \preceq (\Gamma,\Lambda) \wedge \sigma \trianglelefteq \sigma' \wedge (x' = \mathsf{None} \longrightarrow \Gamma,\sigma' \vdash v :: \preceq T))
\end{array}
$$

For a well-formed program Γ, if the execution of a statement transforms one state into another then for all local environments Λ, if the the statement is well-typed according to the environment (Γ,Λ) and the first state conforms to it, so does the second state, and the new heap is an extension of the old one. The same holds for expressions, but additionally the value of the expression conforms to its type, in case there is no exception.

The "heap extension" is a pre-order on states of type $st \Rightarrow st \Rightarrow bool$, where $\sigma \trianglelefteq \sigma'$ means that any object existing on the heap of σ also exists on σ' and has the same type there. (If we considered garbage collection, we would have to restrict this proposition to accessible objects.) The heap extension property holds for any transition of the operational semantics, which turns out to be necessary in our inductive proof.

$$
\begin{array}{l}
\sigma \trianglelefteq \sigma' \stackrel{\mathsf{def}}{=} \forall a\ obj.\ \ \mathsf{heap}\ \sigma\ \ a = \mathsf{Some}\ obj \longrightarrow \\
\qquad\qquad \exists obj'.\ \mathsf{heap}\ \sigma'\ a = \mathsf{Some}\ obj' \wedge \mathsf{obj\_ty}\ obj' = \mathsf{obj\_ty}\ obj
\end{array}
$$

The proof of the main type soundness theorem is by far the heaviest. At the top level, it consists of currently 21 cases, one for each evaluation rule, where

- 8 cases can be solved rather directly (e.g. from the induction hypothesis),
- 7 cases require just simple lemmas on the structure of the state, and
- the remaining 6 cases require extensive reasoning on the characteristic properties of the constructs concerned.

Most of this reasoning is independent of the operational semantics itself and can be tackled separately, which keeps the main proof manageable.

Consequences. A corollary of type soundness is that method calls always execute a suitable method, i.e. a 'method not understood' run-time error is impossible. This property can be stated more formally: for a well-formed program and a state that conforms to the environment, if an expression of reference type (which plays the role of the target expression for the method call considered) evaluates without an exception to a non-null reference, and if for that (static) type and a given signature a method is available, the dynamic method lookup for the same signature according to the class instance pointed at by the reference value yields a proper method body:

$$E = (\Gamma,\Lambda) \wedge \mathsf{wf\_prog}\ \Gamma \wedge E \vdash e::\mathsf{RefT}\ T \wedge \sigma ::\preceq E \wedge \Gamma \vdash \sigma\ -e\triangleright a'\rightarrow \mathsf{Norm}\ \sigma' \wedge a' \neq \mathsf{Val\ Null} \wedge dynT = \mathsf{fst}\ (\mathsf{the\_Obj}\ (\mathsf{heap}\ \sigma'\ (\mathsf{the\_Addr}\ a'))) \wedge \mathsf{mheads}\ \Gamma\ T\ sig \neq \{\} \longrightarrow \exists m.\ \mathsf{cmethd}\ \Gamma\ dynT\ sig = \mathsf{Some}\ m$$

This implies that in a well-formed context, in every instance of the evaluation rule for method calls, the function cmethd returns a proper method body.

As it stands, the type soundness theorem does not directly say anything about non-terminating computations, which might lead to the conclusion that it is useless for the type-safety of reactive systems and looping programs. Fortunately, the theorem guarantees type-safety even in such cases if one accepts the following meta-level reasoning. An infinite computation can be interrupted after any finite number of computation steps, for example by introducing a counter of steps and raising an exception when a given value has been reached. The theorem implies that the state resulting from interrupting the computation after any finite number of statements executed conforms to the environment. Together with the fact that there is no single non-terminating statement, the whole (infinite) computation can be concluded to be type-safe.

In addition to the evaluation semantics, we plan to define a transition semantics and prove both styles equivalent (for finite computations). The transition semantics will be less concise and abstract, but allows type soundness to be formulated as a subject reduction property, which is more natural for infinite computations. More importantly, it seems to be unavoidable to describe concurrency (and I/O).

6 Experience and Statistics

Recalling our design goals stated at the end of §2, we comment how far we have reached them and share some of the lessons learned during the project.

Faithfulness to the official language specification. HOL's expressiveness enables us to formalize the Java specification quite naturally and directly, without facing any severe obstacles. There is almost a one-to-one correspondence between the concepts given in the specification and those defined in Bali. As far as we could tell, all the messy well-formedness conditions inherited from the language specification are actually needed somewhere in the proofs. This inspires confidence in the adequacy of both the specification and our formalization.

We do not yet have tools for automatically generating executable code from our theories, which would be an additional help in validating our formalization. The importance of such a mechanism became very obvious when we uncovered a mistake in our formalization (which was not present in [NO98] but was introduced by modifications) when symbolically executing the example in this article in Isabelle: the list returned by function fields was in reverse order. Although the type soundness proof itself was an excellent debugging mechanism which caught many minor and some major mistakes, it failed to detect the wrong order because type soundness is independent of the order in which fields are inherited. In the original language specification we did not find any significant errors, but some omissions and unneeded restrictions, which we lifted.

Succinctness and simplicity. Our policy to restrict the number of features considered and to make straightforward simplifications that do not diminish the expressiveness of the language has lead to a clear and straightforward formalization. Mixfix syntax and mathematical fonts as offered by Isabelle also contribute greatly to moderately readable definitions and theorems.

The facility to conduct concise proofs strongly depends on the formalization. In our case, the use of the (also more elegant) evaluation semantics saved us from a lot of trouble, while the intricacies of a transition semantics faced by Drossopoulou and Eisenbach [DE97] lead to several mistakes that were finally corrected during Syme's machine-checked proof [Sym97b], but at the expense of additional concepts.

Maintainability and extendibility. Unless the language changes drastically, modifications tend to be of a local nature, but only if both the formalization and the proofs are reasonably structured. As always, modularity is the key issue. But when the formalization is extended, even well-structured proofs need to be modified, which remains a tedious job. Higher-level proof scripts and more automation are some of the answers. A dedicated mechanism for change management exploring and fixing the impact of modifications would also help.

We are reasonably happy with the modularity of our work. For instance, Martin Büchi [BW98] has adpoted the formalization (including the proofs), extended it to handle compound types, and proved the type-safety of the augmented language, all of which worked very smoothy.

Adequacy for the theorem prover. Theorem provers are notoriously sensitive to the precise formulation of definitions and theorems. Thus the two goals of maximal automation of proofs and maximal abstractness of definitions are sometimes in conflict. In a number of cases this meant that although we could start with an abstract definition, we had to derive consequences which were better suited for the available proof procedures. Although we are far from satisfied with the current status of Isabelle's proof procedures (for example, the handling of assumptions during simplification, or the necessity to expand tuples and similar datatypes by hand), they are basically adequate for the task at hand. Nevertheless, more automation is necessary and feasible by extending the capabilities of Isabelle itself.

Statistics. We spent two months (estimated net time) developing and maintaining our formalization, and the Isabelle theory files produced add up to about 1200 lines of well-documented definitions. To conduct and maintain the type soundness proof with all necessary lemmas, it took us roughly three months of work and about 2400 lines of proof scripts.

7 Conclusion

The reader has been exposed to large chunks of a formal language specification and a proof of type soundness and may need to be reminded of the benefits. Even including the slight generalizations mentioned at the beginning of §4, we did not discover a loop-hole in the type system. But we had not seriously expected this either. So what have we gained over and above a level of certainty far beyond any paper-and-pencil proof?

We view our work primarily as an investment for the future. For a start, it can serve as the basis for many other mechanized proofs about Java, e.g. as a foundation for the work by Dean [Dea97] or for compiler correctness. More importantly, we see machine-checked proofs as an invaluable aid in maintaining large language designs (or formal documents of any kind). It is all very well to perform a detailed proof on paper once, but in the face of changes and extensions, the reliability of such proofs begins to crumble. In contrast, we developed the design incrementally, and Isabelle reminded us where proofs needed to be modified. This has shown to be important, for example when we extended Bali with full exception handling. It will continue to help us further: apart from adding the last important Java features missing from Bali, e.g. threads, we also plan to use Bali as a vehicle for experimental extensions of Java such as parameterized types [MBL97,OW97,AFM97].

Despite our general enthusiasm for machine-checked language designs, a few words of warning are in order:

- Bali is still a half-way house: not a toy language any more, but missing many details and some important features of Java.
- The Java type system is, despite subclassing, simpler than that of your average functional language: whereas the type checking rules of Java are almost directly executable, the verification of ML's type inference algorithm against the type system requires a significant effort [NN97]. The key complication there is the presence of free and bound type variables, which requires complex reasoning about substitutions. VanInwegen [Van97] reports similar difficulties in her formalization of the type system and the semantics of ML.
- Theorem provers, and Isabelle is no exception, require a certain learning effort due to the machine-oriented proof style. Recent moves towards a more human-oriented proof style like Syme's DECLARE system [Sym97a] promise to lower this hurdle. However, as Harrison [Har97] points out, both proof styles have their merits, and we are currently investigating a combination.

In a nutshell: although machine-checked language designs for the masses are still some way off, this article demonstrates that they have definitely become a viable option for the expert.

Acknowledgments. We thank Sophia Drossopoulou, Donald Syme and Egon Börger for the very helpful discussions about their related work. We also thank Wolfgang Naraschewski, Markus Wenzel, Andrew Gordon and several anonymous referees for their comments on earlier reports on our project.

References

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

BCM+93. Kim B. Bruce, Jon Crabtree, Thomas P. Murtagh, Robert van Gent, Allyn Dimock, and Robert Muller. Safe and decidable type checking in an object-oriented language. In *Proc. OOPSLA'93*, volume 18 of *ACM SIGPLAN Notices*, pages 29–46, October 1993.

BM88. Robert S. Boyer and J Strother Moore. *A Computational Logic Handbook*. Academic Press, 1988.

Bru93. Kim B. Bruce. Safe type checking in a statically-typed object-oriented programming language. In *Proc. 20th ACM Symp. Principles of Programming Languages*, pages 285–298. ACM Press, 1993.

BS98. Egon Börger and Wolfram Schulte. A programmer friendly modular definition of the dynamic semantics of Java. In Jim Alves-Foss, editor, *Formal Syntax and Semantics of Java*, Lect. Notes in Comp. Sci. Springer-Verlag, 1998. Chapter 11 of this volume.

BvGS95. Kim B. Bruce, Robert van Gent, and Angela Schuett. PolyTOIL: A type-safe polymorphic object-oriented language. In W. Olthoff, editor, *ECOOP '95*, volume 952 of *Lect. Notes in Comp. Sci.*, pages 27–51. Springer-Verlag, 1995.

BW98. Martin Büchi and Wolfgang Weck. Java needs compound types. Technical Report 182, Turku Center for Computer Science, May 1998. `http://www.abo.fi/~mbuechi/publications/CompoundTypes.html`.

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

Coo89. William Cook. A proposal for making Eiffel type-safe. In *Proc. ECOOP'89*, pages 57–70. Cambridge University Press, 1989.

DE97. Sophia Drossopoulou and Susan Eisenbach. Is the Java type system sound? In *Proc. 4th Int. Workshop Foundations of Object-Oriented Languages*, January 1997.

DE98. Sophia Drossopoulou and Susan Eisenbach. Java is type safe — probably. In Jim Alves-Foss, editor, *Formal Syntax and Semantics of Java*, Lect. Notes in Comp. Sci. Springer-Verlag, 1998. Chapter 3 of this volume.

Dea97. Drew Dean. The security of static typing with dynamic linking. In *Proc. 4th ACM Conf. Computer and Communications Security*. ACM Press, 1997.

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

GM93. M.J.C. Gordon and T.F. Melham. *Introduction to HOL: a theorem-proving environment for higher order logic*. Cambridge University Press, 1993.

Har97. John Harrison. Proof style. Technical Report 410, University of Cambridge Computer Laboratory, 1997.

MBL97. Andrew C. Myers, Joseph A. Bank, and Barbara Liskov. Parameterized types for Java. In *Proc. 24th ACM Symp. Principles of Programming Languages*, pages 132–145, 1997.

Mil78. Robin Milner. A theory of type polymorphism in programming. *J. Comp. Sys. Sci.*, 17:348–375, 1978.

NN97. Wolfgang Naraschewski and Tobias Nipkow. Type inference verified: Algorithm W in Isabelle/HOL. In C. Paulin-Mohring, editor, *Proc. Int. Workshop TYPES'96*, volume 1??? of *Lect. Notes in Comp. Sci.* Springer-Verlag, 1997. To appear.

NO98. Tobias Nipkow and David von Oheimb. Java$_{\ell ight}$ is type-safe — definitely. In *Proc. 25th ACM Symp. Principles of Programming Languages*, pages 161–170. ACM Press, 1998.

OW97. Martin Odersky and Philip Wadler. Pizza into Java: Translating theory into practice. In *Proc. 24th ACM Symp. Principles of Programming Languages*, pages 146–159, 1997.

Pau94. Lawrence C. Paulson. *Isabelle: A Generic Theorem Prover*, volume 828 of *Lect. Notes in Comp. Sci.* Springer-Verlag, 1994.

Sli96. Konrad Slind. Function definition in higher order logic. In J. von Wright, J. Grundy, and J. Harrison, editors, *Theorem Proving in Higher Order Logics*, volume 1125 of *Lect. Notes in Comp. Sci.*, pages 381–397. Springer-Verlag, 1996.

Sym97a. Donald Syme. DECLARE: A prototype declarative proof system for higher order logic. Technical Report 416, University of Cambridge Computer Laboratory, 1997.

Sym97b. Donald Syme. Proving Java type soundness. Technical Report 427, University of Cambridge Computer Laboratory, 1997.

Sym98. Donald Syme. Proving Java type soundness. In Jim Alves-Foss, editor, *Formal Syntax and Semantics of Java*, Lect. Notes in Comp. Sci. Springer-Verlag, 1998. Chapter 4 of this volume.

Van97. Myra VanInwegen. Towards type preservation for core SML. University of Cambridge Computer Laboratory, 1997.

Index

An Event-Based Structural Operational Semantics of Multi-threaded Java

Pietro Cenciarelli*, Alexander Knapp, Bernhard Reus, and Martin Wirsing

Ludwig–Maximilians–Universität München
{cenciare,knapp,reus,wirsing}@informatik.uni-muenchen.de

Abstract A structural operational semantics of a significant sublanguage of Java is presented, including the running and stopping of threads, thread interaction via shared memory, synchronization by monitoring and notification, and sequential control mechanisms such as exception handling and return statements. The operational semantics is parametric in the notion of "event space" [6], which formalizes the rules that threads and memory must obey in their interaction. Different computational models are obtained by modifying the well-formedness conditions on event spaces while leaving the operational rules untouched. In particular, we implement the *prescient stores* described in [10, §17.8] which allow certain intermediate code optimizations, and prove that such stores do not affect the semantics of properly synchronized programs.

1 Introduction

The object-oriented programming language Java offers simple and tightly integrated support for concurrent programming. In Java's model of concurrency multiple *threads of control* run in parallel and exchange information by operating on objects which reside in a shared main memory. A precise informal description of this model is given in the Java language specification [10]. Other notable references are [4] and [12].

This paper presents a formal semantics of a significant sublanguage of Java including the running and stopping of threads, thread interaction via shared memory, synchronization by monitoring and notification, and sequential control mechanisms such as exception handling and return statements. Here we focus on the *dynamic* semantics of Java and leave a detailed treatment of the static, type-related aspects of the language, e.g. class declarations, to a followup paper.

Our semantics is given in the style of Plotkin's structural operational semantics (SOS) [15]. In SOS, which has been used in the past for describing SML [13], evaluation is driven by the syntactic structure of programs. This allows a powerful proof technique for semantic analysis: *structural induction*. The idea inspiring the present work is that the semantics of real concurrent languages such as Java, with complex, interacting control features can be given in full detail by means of simple structural rules.

* Research partially supported by the HCM project CHRX-CT94-0591 "De Stijl."

J.Alves-Foss (Ed.): Formal Syntax and Semantics of Java, LNCS 1523, pp.157-200, 1999.

One of the difficulties in modelling concurrent Java programs consists in capturing the complex interplay of memory and thread actions during execution. Each thread of control has, in Java, a private *working memory* in which it keeps its own working copy of variables that it must use or assign. As the thread executes a program, it operates on these working copies. The main memory contains the master copy of each variable. There are rules about when a thread is permitted or required to transfer the contents of its working copy of a variable into the master copy or vice versa. The process of copying is *asynchronous*. There are also rules which regulate the *locking* and *unlocking* of objects, by means of which threads synchronize with each other. All this is described precisely in [10, §17] in terms of eight kinds of low-level actions: *Use*, *Assign*, *Load*, *Store*, *Read*, *Write*, *Lock*, and *Unlock*. Here is an example of a rule from [10, §17.6, p. 407] involving locks and variables. Let T be a thread, V a variable and L a lock:

> "Between an *Assign* action by T on V and a subsequent *Unlock* action by T on L, a *Store* action by T on V must intervene; moreover, the *Write* action corresponding to that *Store* must precede the *Unlock* action, as seen by the main memory."

These rules impose constraints on any implementation of Java so as to allow a correct exchange of information among threads. On the other hand they intentionally leave much freedom to the implementor, thus permitting certain standard hardware and software techniques to improve the speed and efficiency of concurrent code. Therefore, it is *only* on the given rules that the programmer should rely to predict the possible behaviour of a concurrent program. Likewise, it is only the given rules that should constrain the possible execution traces generated by a correct operational semantics.

The above considerations led us to base our semantics on the notion of *event space*. These correspond roughly to *configurations* in Winskel's *event structures* [21] which are denotational, non-interleaving models of concurrent languages. The use of such structures in (interleaving) operational semantics is new. It allows us to give an abstract, "declarative" account of the Java thread model while retaining the virtues of a structural approach. This description is a straight formal paraphrase of the rules of [10]. Event spaces were introduced in [6], where we showed that their use in modelling multi-threading preserves the naive semantics of "sequential" computations (i.e. computations where one thread interacts synchronously with the memory).

Basing our description of Java on the finely grained notion of event allowed us to observe phenomena which may be not readily seen when more abstract approaches are taken. For example, we realized that the asynchrony of communication between main memory and working memories (viz. the *loose* coupling of *Read* and *Load* actions, and similarly of *Store* and *Write*) is actually *observable* in Java. Let threads θ_1 and θ_2, respectively running the code

```
(θ1)   synchronized(p) { p.y = 2; } a = p.x; b = p.y; c = p.y;
(θ2)   synchronized(p) { p.y = 3; p.y = 100; } p.x = 1;
```

share a main memory in which $\mathtt{p.x} = \mathtt{p.y} = 0$, and let their working memories be initially empty. No parallel execution of θ_1 and θ_2 in which main and working memories interact *synchronously* would possibly allow the values 1, 2 and 3 to be assigned respectively to `a`, `b` and `c`. Any model of execution not capable of producing a run with this assignment of values, indeed possible as we show in Section 2.3, provides maybe a correct implementation, but cannot be considered correct as semantics of Java.

The operational semantics presented below is *parametric* in the notion of event space. This allows different computational models to be obtained by modifying the well-formedness conditions on event spaces while leaving the operational rules untouched. To show the flexibility of this approach we study the "prescient" store actions introduced in [10, §17.8]. Such actions allow optimizing compilers to perform certain kinds of code rearrangements. A bisimulation is given to prove that such rearrangements preserve the semantics of properly synchronized programs (see also [17]).

Related work. Several other semantics of sublanguages of Java are available in the literature. Much work has also been done on the semantics of the Java Virtual Machine [7, 16, 18]; this is one half of a formal semantics of the language, the other half being a description of a Java-to-Virtual Machine bytecode compiler, not available to date.

In this volume Drossopoulou and Eisenbach [8] give a "small-step" structural operational semantics which covers roughly the sequential part of our sublanguage of Java; their work, which is mainly concerned with proving type soundness, has been formalized by Syme [19]. Von Oheimb and Nipkow [14] also deal with a sequential sublanguage of Java and give a formal proof of type safety. A noteworthy difference between [8] and [14] is that the latter follows a "big-step" approach. In [9] Flatt, Krishnamurthy and Felleisen investigate the semantics of operators for combining Java classes (so-called "mixins"). All these semantics focus on type soundness for a *sequential* portion of Java.

As for multi-threading, non-structural descriptions based on *abstract state machines* (see [11]) are given by Börger and Schulte [5], and by Wallace [20].

Synopsis. Section 2 describes and formalizes the Java memory-threads communication protocol. Section 3 presents our event-based, structural operational semantics of Java. Section 4 studies the notion of prescient store action. Loose ends and future research are discussed in Section 5.

2 Event Spaces

In this section we describe and formalize the memory-threads communication protocol of Java. This is done by writing the rules of [10, §17] as simple logical clauses (Section 2.2) and by adopting them as well-formedness conditions on structures called event spaces (Section 2.4). The latter are used in the operational judgements to constrain the applicability of some operational rules. An

example of event space is given in Section 2.3, describing the "1-2-3" parallel run of the threads θ_1 and θ_2 introduced above.

2.1 Actions and Events

A formal notion of event is given below in terms of five sets of entities:

- $\{Use, Assign, Load, Store, Read, Write, Lock, Unlock\}$, the action names;
- *Thread_id*, the thread identifiers;
- *Obj*, the objects;
- *LVal*, the left values (or "variables," following [10]) and
- *RVal*, the (right) values.

Intuitively, *Use* and *Assign* actions do just what their names suggest, operating on the private working memories. *Read* and *Load* are used for a loosely coupled copying of data from the main memory to a working memory and dually *Store* and *Write* are used for copying data from a working memory to the main memory. *Lock* and *Unlock* are for synchronizing the access to objects.

Formally, an *action* is either a triple (A, θ, o), where $A \in \{Lock, Unlock\}$, θ is a thread (identifier) and o is an object, or a 4-tuple of the form (A, θ, l, v), where $A \in \{Use, Assign, Load, Store, Read, Write\}$, l is a variable, v is a value and θ is as above. When $A \in \{Use, Assign, Load, Store\}$, the tuple (A, θ, l, v) records that the thread θ performs an A action on l with value v, while, if $A \in \{Read, Write\}$, it records that the main memory performs an A action on l with value v on behalf of θ. If A is *Lock* or *Unlock*, (A, θ, o) records that θ acquires, or respectively relinquishes, a lock on o. Actions with name *Use*, *Assign*, *Load*, *Store*, *Lock* and *Unlock* are called *thread actions*, while *Read*, *Write*, *Lock* and *Unlock* are *memory actions*.

Events are instances of actions, which we think of as happening at different times during execution. We use the same tuple notation for actions and their instances: the context clarifies which one is meant. When no confusion arises we may omit components of an action or event which are not immediately relevant in the context of discourse: so $(Read, l)$ stands for $(Read, \theta, l, v)$, for some θ and v. Given a thread θ, we write $\alpha(\theta)$ for a generic instance of a *thread* action performed by θ. Similarly, $\beta(x)$ indicates a generic instance of a *memory* action involving a location or object x.

2.2 The Rules of interaction

Here we formalize the rules of [10, Chapter 17], to which we refer for a detailed discussion. These rules are translated into logical clauses describing the properties of a poset of events called the "poset of discourse." The events of such a poset, which are thought of as occurring in the given order, are meant to record the activity of memory and threads during the execution of a Java program. We assume that every chain of the poset of discourse can be counted monotonically: $a_0 \leq a_1 \leq a_2 \leq \ldots$. The clauses in our formalization have the form:

$$\forall \mathbf{a} \in \eta \,.\, (\Phi \Rightarrow ((\exists \mathbf{b}_1 \in \eta \,.\, \Psi_1) \vee (\exists \mathbf{b}_2 \in \eta \,.\, \Psi_2) \vee \ldots \ (\exists \mathbf{b}_n \in \eta \,.\, \Psi_n)))$$

where $\mathbf{a}$ and $\mathbf{b}_i$ are lists of events, η is the poset of discourse and $\forall \mathbf{a} \in \eta \,.\, \Phi$ means that Φ holds for all tuples of events in η matching the elements of $\mathbf{a}$ (and similarly for $\exists \mathbf{b}_i \in \eta \,.\, \Psi_i$). The clauses are abbreviated by adopting the following conventions: quantification over $\mathbf{a}$ is left implicit when all events in $\mathbf{a}$ appear in Φ; quantification over $\mathbf{b}_i$ is left implicit when all events in $\mathbf{b}_i$ appear in Ψ_i. Moreover, a rule of the form $\forall \mathbf{a} \in \eta \,.\, (\mathit{true} \Rightarrow \ldots)$ is written $\mathbf{a} \Rightarrow (\ldots)$. When the symbols θ and θ' appear in a rule, we always assume that $\theta \neq \theta'$. Similarly for values v and v', and for events a and a'.

The rules are the following: The actions performed by any one thread are totally ordered, and so are the actions performed by the main memory for any one variable or lock [10, §17.2, §17.5].

$$\alpha(\theta), \alpha'(\theta) \Rightarrow \alpha(\theta) \leq \alpha'(\theta) \vee \alpha'(\theta) \leq \alpha(\theta) \tag{1}$$

$$\beta(x), \beta'(x) \Rightarrow \beta(x) \leq \beta'(x) \vee \beta'(x) \leq \beta(x) \tag{2}$$

Hence, the occurrences of any action (A, θ, x) are totally ordered in the poset of discourse. We write $\eta(A, \theta, x)$ the subposet of η including only instances of (A, θ, x).

A *Store* action by θ on l must intervene between an *Assign* by θ of l and a subsequent *Load* by θ of l. Less formally, a thread is not permitted to lose its most recent assign [10, §17.3]:

$$(\mathit{Assign}, \theta, l) \leq (\mathit{Load}, \theta, l) \Rightarrow (\mathit{Assign}, \theta, l) \leq (\mathit{Store}, \theta, l) \leq (\mathit{Load}, \theta, l) \tag{3}$$

A thread is not permitted to write data from its working memory back to main memory for no reason [10, §17.3]:

$$(\mathit{Store}, \theta, l) \leq (\mathit{Store}, \theta, l)' \Rightarrow (\mathit{Store}, \theta, l) \leq (\mathit{Assign}, \theta, l) \leq (\mathit{Store}, \theta, l)' \tag{4}$$

Threads start with an empty working memory and new variables are created only in main memory and are not initially in any thread's working memory [10, §17.3]:

$$(\mathit{Use}, \theta, l) \Rightarrow (\mathit{Assign}, \theta, l) \leq (\mathit{Use}, \theta, l) \vee (\mathit{Load}, \theta, l) \leq (\mathit{Use}, \theta, l) \tag{5}$$

$$(\mathit{Store}, \theta, l) \Rightarrow (\mathit{Assign}, \theta, l) \leq (\mathit{Store}, \theta, l) \tag{6}$$

A *Use* action transfers the contents of the thread's working copy of a variable to the thread's execution engine [10, §17.1]:

$$\begin{aligned} &(\mathit{Assign}, \theta, l, v) \leq (\mathit{Use}, \theta, l, v') \Rightarrow \\ &\quad (\mathit{Assign}, \theta, l, v) \leq (\mathit{Assign}, \theta, l)' \leq (\mathit{Use}, \theta, l, v') \vee \\ &\quad (\mathit{Assign}, \theta, l, v) \leq (\mathit{Load}, \theta, l) \leq (\mathit{Use}, \theta, l, v') \end{aligned} \tag{7}$$

$$\begin{aligned} &(\mathit{Load}, \theta, l, v) \leq (\mathit{Use}, \theta, l, v') \Rightarrow \\ &\quad (\mathit{Load}, \theta, l, v) \leq (\mathit{Assign}, \theta, l) \leq (\mathit{Use}, \theta, l, v') \vee \\ &\quad (\mathit{Load}, \theta, l, v) \leq (\mathit{Load}, \theta, l)' \leq (\mathit{Use}, \theta, l, v') \end{aligned} \tag{8}$$

A *Store* action transmits the contents of the thread's working copy of a variable to main memory [10, §17.1]:

$$\begin{aligned}(Assign, \theta, l, v) \leq (Store, \theta, l, v') \Rightarrow \\ (Assign, \theta, l, v) \leq (Assign, \theta, l)' \leq (Store, \theta, l, v')\end{aligned} \tag{9}$$

The following rules require some events to be paired in the poset of discourse. Let A and B be posets, and let $f : A \rightleftharpoons B$ indicate that a function f is either a monotonic injection $A \to B$ with downward closed codomain or the *partial* inverse of a monotonic injection $B \to A$ with downward closed codomain. For every poset η satisfying (1) and (2), for every thread θ, left value l and object o, there exist unique functions

$$\begin{aligned}read\_of_{\eta,\theta,l} &: \eta(Load, \theta, l) \rightleftharpoons \eta(Read, \theta, l) \\ store\_of_{\eta,\theta,l} &: \eta(Write, \theta, l) \rightleftharpoons \eta(Store, \theta, l) \\ lock\_of_{\eta,\theta,o} &: \eta(Unlock, \theta, o) \rightleftharpoons \eta(Lock, \theta, o).\end{aligned}$$

These are called the "pairing" functions. Indices are omitted when understood. The function *read_of* matches the n-th occurrence of $(Load, \theta, l)$ in η with the n-th occurrence of $(Read, \theta, l)$ if such an event exists in η and is undefined otherwise. Similarly for *store_of* and *lock_of*.

Each *Load* or *Write* action is uniquely paired with a preceding *Read* or *Store* action respectively. Matching actions bear identical values [10, §17.2, §17.3]:

$$(Load, \theta, l, v) \Rightarrow (Read, \theta, l, v) = read\_of(Load, \theta, l, v) \leq (Load, \theta, l, v) \tag{10}$$

$$(Write, \theta, l, v) \Rightarrow (Store, \theta, l, v) = store\_of(Write, \theta, l, v) \leq (Write, \theta, l, v) \tag{11}$$

Rules (10) and (11) ensure that *read_of* and *store_of* are total. We call *load_of* and *write_of* their partial inverses.

The actions on the master copy of any given variable on behalf of a thread are performed by the main memory in exactly the order that the thread requested [10, §17.3]:

$$(Store, \theta, l) \leq (Load, \theta, l) \Rightarrow write\_of(Store, \theta, l) \leq read\_of(Load, \theta, l) \tag{12}$$

A thread is not permitted to unlock a lock it does not own [10, §17.5]:

$$(Unlock, \theta, o) \Rightarrow lock\_of(Unlock, \theta, o) \leq (Unlock, \theta, o) \tag{13}$$

Rule (13) ensures that *lock_of* is total. We write *unlock_of* its partial inverse.

Only one thread at a time is permitted to lay claim to a lock, and moreover a thread may acquire the same lock multiple times and does not relinquish ownership of it until a matching number of *Unlock* actions have been performed [10, §17.5]:

$$(Lock, \theta, o) \leq (Lock, \theta', o) \Rightarrow unlock\_of(Lock, \theta, o) \leq (Lock, \theta', o) \tag{14}$$

If a thread is to perform an *Unlock* action on any lock, it must first copy all assigned values in its working memory back out to main memory [10, §17.6] (this rule formalizes the quotation in the introduction):

$$\begin{aligned}&(Assign, \theta, l) \leq (Unlock, \theta) \Rightarrow \\ &\quad (Assign, \theta, l) \leq store\_of(Write, \theta, l) \leq (Write, \theta, l) \leq (Unlock, \theta)\end{aligned} \tag{15}$$

A *Lock* action acts as if it flushes all variables from the thread's working memory; before use they must be assigned or loaded from main memory [10, §17.6]:

$$\begin{aligned}&(Lock, \theta) \leq (Use, \theta, l) \Rightarrow \\ &\quad (Lock, \theta) \leq (Assign, \theta, l) \leq (Use, \theta, l) \vee \\ &\quad (Lock, \theta) \leq read\_of(Load, \theta, l) \leq (Load, \theta, l) \leq (Use, \theta, l)\end{aligned} \tag{16}$$

$$(Lock, \theta) \leq (Store, \theta, l) \Rightarrow (Lock, \theta) \leq (Assign, \theta, l) \leq (Store, \theta, l) \tag{17}$$

Discussion. Each of the above rules corresponds to one rule in [10]. Note that the language specification requires any *Read* action to be completed by a corresponding *Load* and similarly for *Store* and *Write*. The above theory does not include clauses expressing such requirements because it must capture "incomplete" program executions (see Section 4). Except for read and store completion, any rule in [10] which we have not included above can be derived in our axiomatization. In particular,

$$(Load, \theta, l) \leq (Store, \theta, l) \Rightarrow (Load, \theta, l) \leq (Assign, \theta, l) \leq (Store, \theta, l) \qquad (*)$$

of [10, §17.3] holds in any model of the axioms. In fact, by (6) there must be some *Assign* action before the *Store*; moreover, one of such *Assign* must intervene in between the *Load* and the *Store*, because otherwise, from (1) and (3), there would be a chain $(Store, \theta, l) \leq (Load, \theta, l) \leq (Store, \theta, l)$ with no *Assign* in between, which contradicts (4). Similarly, the following rule of [10, §17.3] derives from (10) and (11):

$$(Load, \theta, l) \leq (Store, \theta, l) \Rightarrow read\_of(Load, \theta, l) \leq write\_of(Store, \theta, l)$$

Clauses (6) and (17) simplify the corresponding rules of [10, §17.3, §17.6] which include a condition $(Load, \theta, l) \leq (Store, \theta, l)$ to the right of the implication. This would be redundant because of $(*)$.

2.3 Example

We briefly illustrate the above formal rules on the example given in the introduction, where two threads

(θ_1) `synchronized(p) { p.y = 2; } a = p.x; b = p.y; c = p.y;`
(θ_2) `synchronized(p) { p.y = 3; p.y = 100; } p.x = 1;`

start with a main memory where both instance variables `p.x` and `p.y` have value 0, and with empty working memories, and interact so that the values 1, 2 and 3

are eventually assigned to `a`, `b`, and `c` respectively. We shall run part of this example through our operational rules in Section 3.7. Figure 1 describes this run as a poset of events, whose ordering is represented by the arrows. The actions of the two threads and of the main memory on the two instance variables `p.x` and `p.y` are aligned vertically in four columns. We let o be the object denoted by `p`, while x and y stand for the left values of `p.x` and `p.y` respectively.

Since all actions performed by the same thread and by the memory on the same variable must be totally ordered, each column of Figure 1 is a chain. Moreover, some memory actions must occur before or after some thread actions. For example, a $(\mathit{Write}, \theta_1, y, 2)$ must come after $(\mathit{Assign}, \theta_1, y, 2)$ because, as dictated by the structure of the program, an *Unlock* follows the assignment `p.y` $= 2$, and hence, by (15), θ_1's working copy of y must be written in main memory before the *Unlock* and after a corresponding *Store*. Note that not all the assigned values must be stored in main memory. For example, it would have been legal to omit $(\mathit{Store}, \theta_2, y, 3)$ and $(\mathit{Write}, \theta_2, y, 3)$; in this case, however, the value 3 would have never been passed to θ_1. Similarly, not all the values used by a thread must be first loaded from main memory: in the example no $(\mathit{Load}, \theta_1, y, 2)$ precedes $(\mathit{Use}, \theta_1, y, 2)$.

As stated in the introduction, the above assignments to `a`, `b` and `c` would not be possible if communication between main and working memories where "synchronous," that is if no other event were allowed to happen between a *Read* and a corresponding *Load* or, equivalently, if these two actions were executed as a single atomic step (and similarly for *Store* and *Write*). Assume in fact that there is a synchronous run producing `a` $= 1$, `b` $= 2$, and `c` $= 3$. Since 3 must be assigned to `c`, an action $(\mathit{Read}, \theta_1, y, 3)$ must occur, and moreover it must be after θ_2 writes 3 and before it writes 100 in the master copy of y. Hence, by (15), $(\mathit{Read}, \theta_1, y, 3)$ must occur while θ_2 is executing the synchronized block. Again by (15), a $(\mathit{Store}, \theta_1, y, 2)$ must occur before θ_1 exits its synchronized block; moreover this *Store* must occur before $(\mathit{Read}, \theta_1, y, 3)$, otherwise the value 3 would be lost, and therefore θ_1 must enter its synchronized block before θ_2. Then, in order to get the value 1 for `a`, the assignment `a` $=$ `p.x` must occur after θ_2 has left the block, it has assigned, stored and written 1 in x, and after θ_1 has read and loaded such value in its working copy of x. However, by the time θ_1 can load 1 in x, the value of y in its working memory must already be 3, because a $(\mathit{Read}, \theta_1, y, 3)$ occured while θ_2 was executing the synchronized block. Therefore, to assign 2 to `b`, θ_1 can neither rely on the content of it's working copy of y, nor on the master copy in main memory, which, by now, must contain 100.

2.4 Event Spaces

An *event space* is a poset of events every chain of which can be counted monotonically ($a_0 \leq a_1 \leq a_2 \leq \dots$) and satisfying conditions (1) to (17) of Section 2.2.

Event spaces serve two purposes in our operational semantics: On the one hand they provide all the information needed to reconstruct the working memories (which in fact do not appear in the operational judgements). On the other

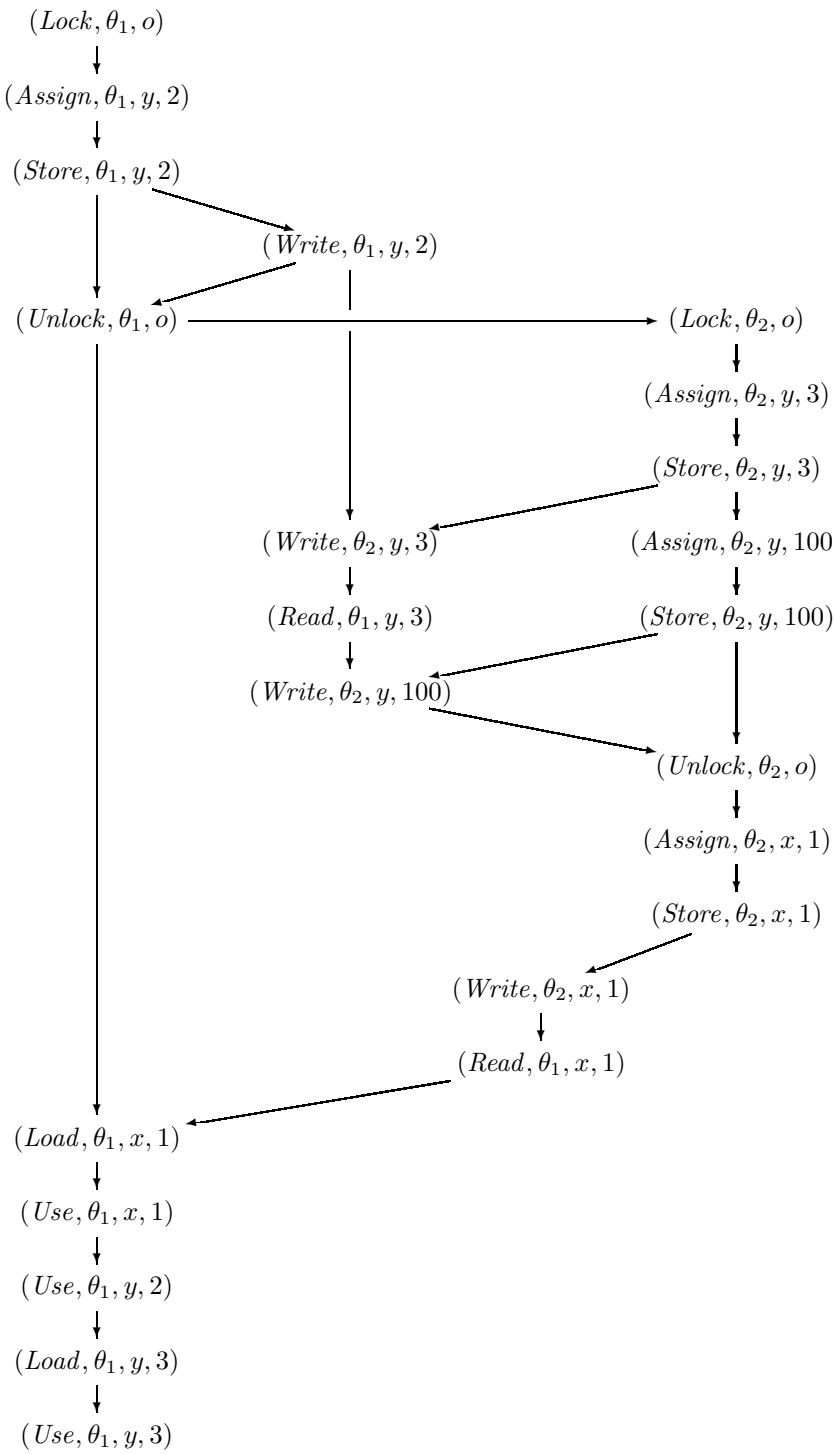


Figure 1. An event space for Example 2.3

hand event spaces record the "historical" information on the computation which constrains the execution of certain actions according to the language specification, and hence the applicability of certain operational rules (see Section 3.4).

Given two event spaces $(X, \leq_X)$ and $(Y, \leq_Y)$, we say that $(X, \leq_X)$ is a *conservative extension* of $(Y, \leq_Y)$ when $Y \subseteq X$ and $\leq_Y \subseteq \leq_X$ and, for all $a, b \in Y$, $a \leq_X b$ implies $a \leq_Y b$.

To adjoin a new event a to an event space $\eta = (X, \leq_X)$, we use an operation $\oplus$ defined as follows: $\eta \oplus a$ denotes nondeterministically an event space $\eta' = (Y, \leq_Y)$ such that:

- η' is a conservative extension of η, with $Y = X \cup \{a\}$;
- if $a = \alpha(\theta)$ is a thread action performed by θ, then $a' \leq_Y a$ for all thread actions $a' = \alpha'(\theta)$ by θ in η';
- if $a = \beta(x)$ is a memory action on x, then $a' \leq a$ for all memory actions $a' = \beta'(x)$ on x in η'.

If no event space η' exists satisfying these conditions, then $\eta \oplus a$ is undefined. For example, by (5), the term $\eta \oplus (\mathit{Use}, \theta, l)$ is defined only if a suitable $(\mathit{Assign}, \theta, l)$ or $(\mathit{Load}, \theta, l)$ occurs in η. If η is an event space and $\mathbf{a} = (a_1, a_2, \ldots, a_n)$ is a sequence of events, we write $\eta \oplus \mathbf{a}$ for $\eta \oplus a_1 \oplus a_2 \oplus \cdots \oplus a_n$.

As little ordering may be added to an event space by the operation $\oplus$ as is required by the rules of interaction: indeed two expressions $\eta \oplus a \oplus b$ and $\eta \oplus b \oplus a$ may denote the same event space. This reflects the fact that the same concurrent activity may be described by different sequences of interleaved events. *More* ordering can also be introduced than strictly dictated by the rules. For example, the expression $(\mathit{Read}, \theta, o) \oplus (\mathit{Lock}, \theta, l, v) \oplus (\mathit{Load}, \theta, l, v)$ may produce an event space $\{(\mathit{Lock}, \theta, o) \leq (\mathit{Read}, \theta, l, v) \leq (\mathit{Load}, \theta, l, v)\}$: although no rule enforces that $(\mathit{Lock}, \theta, o) \leq (\mathit{Read}, \theta, l, v)$, it better be so in view of rule (16) if a $(\mathit{Use}, \theta, l)$ is to be further added to the space.

3 Operational Semantics

The present paper focuses on the dynamic semantics of Java. Of course, the behaviour of a program may depend on type information obtained from static analysis. Part of this information we assume is retrievable at run-time from the main memory (see Section 3.1), part goes to enrich the syntactic terms upon which the operational semantics operates (see Section 3.2).

In Java every variable and every expression has a type which is known at compile-time. The type limits the possible values that the variable can hold or expression can produce at run-time. Adopting the terminology of [10], every object *belongs* to a class (the class *of* the object, the one which is mentioned when the object is created). Moreover, the values contained by a variable or produced by an expression should, by the design of the language, be *compatible* with the type of the variable or expression. A value of primitive type (such as booleans) is only compatible with that type (`boolean`), while a reference to an object is compatible with any class type which is a superclass of the object's

class [10, §4.5.5]. We do not implement run-time compatibility checks in our semantics (they can be added straightforwardly). For example, like in Java, we do not check that the object produced by evaluating the expression e in `throw` e; is compatible with `Throwable`. However, we do use type information wherever it is needed to drive computation. An example is the execution of a `try-catch` statement (see Section 3.8).

Java's *modifiers* are not treated in the present paper. For example, we do not consider *static* fields; these would require minor changes of the semantic machinery. Similarly, *synchronized* methods can be easily implemented by using synchronized statements (see Section 3.7), as remarked in [10, §8.4.3.5].

After introducing in Section 3.1 semantic domains such as stores and environments, we describe a "compilation" function translating Java programs into semantically enriched *abstract syntax* (Section 3.2). Next, we define operational judgements (Section 3.3) and give the SOS rules which generate them. These are presented in homogeneous groups (expressions, statements, exceptions, etc.) in Section 3.4 to 3.10.

3.1 Semantic Domains

Primitive semantic domains. These are the building blocks of our operational semantics, and nothing is assumed on the structure of their elements.

We call *RVal* the primitive domain of (right) values. These are produced by the evaluation of expressions and can be assigned to variables. A distinguished subset *Obj* of *RVal* is also given as primitive; we call its elements (references to) *objects*. In particular, since threads are objects in Java, we choose the domain *Thread_id* of the previous section to be *Obj*. Right values come equipped with a primitive function *value* mapping literals to the corresponding values.

$$value : Literal \to RVal$$

In particular, *null* is the reference to the null object denoted by the literal `null`, that is: $null = value(\texttt{null})$. Similarly, $true = value(\texttt{true})$ and so on.

In Java the object denoted by an expression e may contain several fields with the same name i; then, the type of e decides on which field is actually accessed by the expression $e.i$. An identifier together with a type are therefore a non-ambiguous name for field access. We call *FieldIdentifier*, ranged over by f, the set of such pairs (see Table 1). The domain of left-values introduced in the previous section is *not* primitive: an instance variable is addressed by a non-null object reference o together with a field identifier f, and written $o.f$.

$$LVal = (Obj \setminus \{null\}) \times FieldIdentifier$$

Store is the primitive domain of stores ranged over by μ. This domain comes equipped with the following primitive semantic functions, where *ClassType* is as in Appendix A:

$$new : ClassType \times Store \to Obj \times Store$$

$$upd : LVal \times RVal \times Store \rightharpoonup Store$$
$$rval : LVal \times Store \rightharpoonup RVal.$$

Besides providing storage for variables, stores are assumed to contain information produced by the static analysis of a program; typically: the names and types of fields and methods for each class, the initial values of fields, the subclass relation, and so on. This information does not change during execution and it could alternatively be kept separate from stores.

Given a class type C and a store μ, the function *new* produces a new object of type C with suitably initialized instance variables, and returns it in output together with μ updated with the new object. We write:

$$o \in_\mu C,$$

dropping μ when understood, to mean that o is a reference to an object in μ of a class type which is *compatible* with C. We also assume that the partial function $init : FieldIdentifier \times Store \rightharpoonup RVal$ returns the initial values for an object's fields. The domain of this function is the set of pairs (f, μ) where $f = (i, C)$ and i is an appropriate field for C in μ.

The function *upd* updates a store, while *rval* gets the right-value associated in a store with a given left-value. These functions are partial: they are undefined on the left-values $o.f$ where f is not an appropriate field for o in the given store. We write $\mu[l \mapsto v]$ and $\mu(l)$ for $upd(l, v, \mu)$ and $rval(l, \mu)$ respectively.

A rather weak axiomatization of stores is given below by using a binary predicate $\preceq$ (written infix). The meaning of $e_1 \preceq e_2$ is that if e_1 is defined, then so is e_2 and they denote the same value. By $e_1 \simeq e_2$ we mean that both $e_1 \preceq e_2$ and $e_2 \preceq e_1$ hold.

$$\begin{aligned}
\mu(l) &\preceq \mu'(l) && \text{where } new(C, \mu) = (o, \mu') \\
init((i, C), \mu) &\preceq \mu'(o.(i, C)) && \text{where } new(C, \mu) = (o, \mu') \\
\mu[l \mapsto v](l) &\preceq v \\
\mu[l' \mapsto v](l) &\simeq \mu(l) && \text{if } l \neq l' \\
\mu[l \mapsto v'][l \mapsto v] &\preceq \mu[l \mapsto v] \\
\mu[l' \mapsto v'][l \mapsto v] &\simeq \mu[l \mapsto v][l' \mapsto v'] && \text{if } l \neq l' \\
\mu[l \mapsto \mu(l)] &\preceq \mu
\end{aligned}$$

Finally, *Throws* is the primitive domain of exceptional results. Upon occurrence of an exception, Java allows objects to be passed to handlers as "reasons" for the exception. The primitive function

$$throw : Obj \to Throws$$

turns an object into an exception $throw(o)$ "with reason o." Note that elements of *Throws* are not right values.

Environments and stacks. Environments are pairs (I, ρ) where I is a subset of $\mathit{Identifier} \cup \{\mathit{this}\}$ and ρ is a partial function from I to right values.

$$\mathcal{I} = \mathit{Identifier} \cup \{\mathit{this}\}$$
$$\mathit{Env} = \textstyle\sum_{I \subseteq \mathcal{I}} (I \rightharpoonup \mathit{RVal})$$

The component I of an environment (I, ρ), called the *source* of ρ, is meant to contain the local variables of a block and the formal parameters of a method body or of an exception handler. Environments are also used to store the information on which object's code is currently being executed: $\rho(\mathit{this})$. By abuse of notation, we write ρ for an environment (I, ρ) and indicate with $\mathit{src}(\rho)$ its source I. In particular, we understand that $\rho_\emptyset$ is an empty environment $(I, \rho_\emptyset)$ such that $\rho_\emptyset(i)$ is undefined for all $i \in I$. As usual, $\rho[i \mapsto v](j) = v$ if $i = j$ and $\rho[i \mapsto v](j) \simeq \rho(j)$ otherwise.

Let *Stack* be the domain of stacks of environments, and let the metavariable σ range over this domain. The empty stack is written $\sigma_\emptyset$. The operation $\mathit{push} : \mathit{Env} \times \mathit{Stack} \to \mathit{Stack}$ is the usual one on stacks. An instance variable declaration $i = v$ binds v to i in the topmost environment of a stack σ; we write $\sigma[i = v]$ the result of this operation. The result of assigning v to i in the first environment (I, ρ) of σ such that $i \in I$ is written $\sigma[i \mapsto v]$. The value associated with i in such an environment is denoted by $\sigma(i)$. More precisely:

$$\sigma[i = v] = \begin{cases} \mathit{push}(\rho[i \mapsto v], \sigma') & \text{if } \sigma = \mathit{push}(\rho, \sigma') \text{ and } i \in \mathit{src}(\rho) \\ \mathit{undefined} & \text{otherwise;} \end{cases}$$

$$\sigma[i \mapsto v] = \begin{cases} \mathit{push}(\rho[i \mapsto v], \sigma') & \text{if } \sigma = \mathit{push}(\rho, \sigma') \text{ and } i \in \mathit{src}(\rho) \\ \mathit{push}(\rho, \sigma'[i \mapsto v]) & \text{if } \sigma = \mathit{push}(\rho, \sigma') \text{ and } i \notin \mathit{src}(\rho) \\ \mathit{undefined} & \text{otherwise;} \end{cases}$$

$$\sigma(i) = \begin{cases} \rho(i) & \text{if } \sigma = \mathit{push}(\rho, \sigma') \text{ and } i \in \mathit{src}(\rho) \\ \sigma'(i) & \text{if } \sigma = \mathit{push}(\rho, \sigma') \text{ and } i \notin \mathit{src}(\rho) \\ \mathit{undefined} & \text{otherwise.} \end{cases}$$

3.2 Abstract Terms

The operational semantics presented below does not work directly on the Java syntax of Appendix A, which we call *concrete*, but on the *abstract* terms produced by the grammar of Table 1. We call *A-Term* the set of abstract terms and let t range over this set. Concrete and abstract syntax share the clauses defining *Identifier*, *Literal*, *ReturnType* and *ClassInstanceCreationExpression*.

Some of the abstract terms, those which cannot be further evaluated, play the role of *results* in our operational semantics. There are operational rules which only apply when a result is produced ([assign4] for example). Some of the results are called *abrupt* (see Section 3.8), as specified by the following grammar:

$$\mathit{Results} ::= * \mid \mathit{RVal} \mid \mathit{AbruptResults}$$
$$\mathit{AbruptResults} ::= \mathit{Throws} \mid \mathit{return}\ \mathit{RVal} \mid \mathit{return}$$

The terms *return v* and *return* are results produced by evaluating return statements, respectively with and without a return value.

In most cases, abstract terms look just like their concrete counterparts. Some abstract terms, however, are enriched with semantic information produced by the static analysis of the Java program. For example, abstract blocks, which we write $\{S\}_\rho$, have two components: a sequence S of (abstract) statements and an environment ρ containing the local variables of the block. We leave ρ implicit when irrelevant.

Unlike with field identifiers, the method invoked by a method call $e.i(\dots)$ is only known at run-time, because it depends not only on the static type C of e but on the dynamic class type of the object denoted by e. At compile-time, however, a "most specific compile-time declaration" is chosen for i among the methods of C and of its superclasses. The class where this declaration is found, the types of the parameters and the return type are attached by the compiler to i for later run-time usage (see [10, §15.11] for more detail). This motivates the introduction of the domain *MethodIdentifier* in the abstract syntax. When the rest is understood, we write just the identifier of a method identifier.

A recursive function $(\_)^\circ$ translates concrete into abstract syntax. Terms of the shared domains are translated into themselves. The concrete list-like syntactic domains, such as *BlockStatements*, are translated in the obvious way into abstract domains of the form $\mathcal{K}^*$ and $\mathcal{K}^+$, where:

$$\mathcal{K}^* ::= () \mid \mathcal{K}\,\mathcal{K}^*$$
$$\mathcal{K}^+ ::= \mathcal{K} \mid \mathcal{K}\,\mathcal{K}^* \ .$$

Lists that are *optional* in a concrete term are translated into the empty list () when missing. In writing abstract terms we often omit the empty list.

The translation is generally trivial. For example: $(\texttt{throw}\,(e))^\circ = \texttt{throw}\,(e^\circ)$. All non-trivial cases are listed in Table 2. We understand that a "declaration environment" is implicitly carried along during translation, recording the static information collected from processing class declarations. We express that an expression e has declared type τ (in the current declaration environment) by writing $e : \tau$.

Every syntactic domain $A\text{-}\mathcal{K}$ of the abstract syntax corresponds to a concrete domain $\mathcal{K}$, and the translation is such that $t \in \mathcal{K}$ whenever $t^\circ \in A\text{-}\mathcal{K}$. There are syntactic categories in the abstract syntax which have no counterpart in the concrete; these are: *Obj*, *RVal*, *Throws*, *FieldIdentifier*, *MethodIdentifier* and *ActivationFrame*. Of these only the latter is still to be discussed, which we do in Section 3.5.

3.3 Operational Judgements

Configurations. A configuration represents the state of execution of a multi-threaded Java program; therefore, it may include several abstract terms, one for each thread of execution. Each thread has an associated stack. We call *M-term*

$$
\begin{aligned}
\textit{A-Statement} &::= * \mid \texttt{;} \mid \textit{A-Block} \mid \textit{A-StatementExpression}\,\texttt{;} \\
&\mid \texttt{synchronized(}\,\textit{A-Expression}\,\texttt{)}\ \textit{A-Block} \\
&\mid \textit{A-IfThenStatement} \mid \textit{AbruptResults} \\
&\mid \texttt{throw}\ \textit{A-Expression}\,\texttt{;} \mid \textit{A-TryStatement} \\
&\mid \texttt{return;} \mid \texttt{return}\ \textit{A-Expression}\,\texttt{;} \\
\textit{A-Block} &::= \{\ \textit{A-BlockStatement}^*\ \}\ \textit{Env} \\
\textit{A-BlockStatement} &::= \textit{A-LocalVariableDeclaration}\,\texttt{;} \mid \textit{A-Statement} \\
\textit{A-LocalVariableDeclaration} &::= \textit{Type}\ \textit{A-VariableDeclarator}^+ \\
\textit{A-VariableDeclarator} &::= \textit{Identifier}\ =\ \textit{A-Expression} \\
\textit{A-Expression} &::= \textit{RVal} \mid \textit{Throws} \mid \textit{Literal} \mid \textit{Identifier} \mid \texttt{this} \\
&\mid \textit{A-FieldAccess} \mid \textit{ClassInstanceCreationExpression} \\
&\mid \textit{A-MethodInvocation} \mid \textit{ActivationFrame} \\
&\mid \textit{A-Assignment} \mid \textit{UnaryOperator}\ \textit{A-Expression} \\
&\mid \textit{A-Expression}\ \textit{BinaryOperator}\ \textit{A-Expression} \\
\textit{A-FieldAccess} &::= \textit{A-Expression}\,.\,\textit{FieldIdentifier} \\
\textit{FieldIdentifier} &::= (\textit{Identifier}\,,\ \textit{ClassType}) \\
\textit{A-MethodInvocation} &::= \textit{A-Expression}\,.\,\textit{MethodIdentifier}\,(\ \textit{A-Expression}^*\ ) \\
\textit{MethodIdentifier} &::= (\textit{Identifier}\,,\ \textit{ClassType}\,,\ \textit{Type}^*,\ \textit{ResultType}) \\
\textit{ActivationFrame} &::= (\textit{MethodIdentifier}\,,\ \textit{A-Block}) \\
\textit{A-Assignment} &::= \textit{A-LeftHandSide}\ =\ \textit{A-Expression} \\
\textit{A-LeftHandSide} &::= \textit{Identifier} \mid \textit{A-FieldAccess} \\
\textit{A-StatementExpression} &::= \textit{A-Assignment} \mid \textit{ClassInstanceCreationExpression} \\
&\mid \textit{A-MethodInvocation} \mid \textit{ActivationFrame} \\
\textit{A-TryStatement} &::= \texttt{try}\ \textit{A-Block}\ \textit{A-CatchClause}^+ \\
&\mid \texttt{try}\ \textit{A-Block}\ \textit{A-CatchClause}^*\ \texttt{finally}\ \textit{A-Block} \\
\textit{A-CatchClause} &::= \texttt{catch(}\ \textit{Type Identifier}\,\texttt{)}\ \textit{A-Block} \\
\textit{A-IfThenStatement} &::= \texttt{if}\,(\ \textit{A-Expression}\,)\ \textit{A-Statement}
\end{aligned}
$$

Table 1. Abstract syntax

$$
\begin{aligned}
\{\ S\ \}^\circ &= \{\ S^\circ\ \}_{(I,\rho_\emptyset)} \text{ where } I \text{ is the set of local variables declared in } S \\
(\texttt{catch}\ (\tau\ i)\ b)^\circ &= \texttt{catch}\ (\tau\ i)\ \{\ S\ \}_{(I\cup\{i\},\rho_\emptyset)} \text{ where } \{\ S\ \}_{(I,\rho_\emptyset)} = b^\circ \\
((e))^\circ &= e^\circ \\
(e\,.\,i)^\circ &= e^\circ.f \text{ where } e:\tau \text{ and } f = (i,\tau) \\
(e\,.\,i(E))^\circ &= e^\circ.\,m(E^\circ) \text{ where } m = (i, C, \mathcal{T}, \tau) \text{ and the ``compile-time declaration'' of } i \text{ is found in } C \text{ and has signature } \mathcal{T} \to \tau
\end{aligned}
$$

$$
i^\circ = \begin{cases} i & \text{if } i \text{ appears in the scope of a local variable declaration with that name;} \\ \texttt{this}.f & \text{otherwise, where } \texttt{this}:\tau \text{ and } f = (i,\tau). \end{cases}
$$

Table 2. Translation to abstract syntax

a partial map from thread identifiers to pairs (t, σ), where t is an abstract term and σ is a stack. We let the metavariable T range over M-terms:

$$T : \mathit{Thread\_id} \rightharpoonup \mathit{A\text{-}Term} \times \mathit{Stack}.$$

When we assume that θ is not in the domain of T we write $T \mid (\theta, t, \sigma)$ for the M-term T' such that $T'(\theta) = (t, \sigma)$ and $T'(\theta') \simeq T(\theta')$ for $\theta' \neq \theta$, where $\simeq$ is as in Section 3.1.

A *configuration* of the operational semantics is a triple (T, η, μ) consisting of an M-term T, an event space η and a store μ. In writing configurations, we generally drop the parentheses and all parts that are not immediately relevant in the context of discourse; for example, we may write just "t, σ, η" to mean some configuration $(T \mid (\theta, t, \sigma), \eta, \mu)$. Configurations are ranged over by γ.

Operational semantics. The *operational semantics* is the smallest binary relation $\longrightarrow$ on configurations which is closed under the rules of Section 3.4 to 3.10. These are, in fact, rule *schemes*, whose instances are obtained by replacing the metavariables with suitable semantic objects. Rules with no premise are called *axioms*. Related pairs of configurations are written $\gamma_1 \longrightarrow \gamma_2$ and called *operational judgements* or *transitions*.

Rule conventions. In writing an axiom $\gamma_1 \longrightarrow \gamma_2$ we focus only on the relevant parts of the configurations involved, and understand that whatever is omitted from γ_1 remains unchanged in γ_2. For example, we understand that the axiom $; \longrightarrow *$ stands for $T \mid (\theta, ;, \sigma), \eta, \mu \longrightarrow T \mid (\theta, *, \sigma), \eta, \mu$. On the other hand, rules with a premise are read by assuming that whatever changes occur in the omitted parts of the premise (besides thread identifiers) also occur in the conclusion (unless otherwise specified). For example, we understand that:

$$\frac{e_1 \longrightarrow e_2}{e_1\,; \longrightarrow e_2\,;} \quad \text{stands for} \quad \frac{T_1 \mid (\theta, e_1, \sigma_1), \eta_1, \mu_1 \longrightarrow T_2 \mid (\theta, e_2, \sigma_2), \eta_2, \mu_2}{T_1 \mid (\theta, e_1\,;, \sigma_1), \eta_1, \mu_1 \longrightarrow T_2 \mid (\theta, e_2\,;, \sigma_2), \eta_2, \mu_2}\,.$$

Metavariable convention. The metavariables used below (in variously decorated form) in the rule schemes range as follows: $k \in \mathit{Literal}$, $i \in \mathit{Identifier}$, $f \in \mathit{FieldIdentifier}$, $m \in \mathit{MethodIdentifier}$, $o \in \mathit{Obj}$, $l \in \mathit{LVal}$, $v \in \mathit{RVal}$, $V \in \mathit{RVal}^*$, $e \in \mathit{A\text{-}Expression}$, $E \in \mathit{A\text{-}Expression}^*$, $\tau \in \mathit{Type}$, $C \in \mathit{ClassType}$, $d \in \mathit{A\text{-}VariableDeclarator}$, $D \in \mathit{A\text{-}VariableDeclarator}^*$, $s \in \mathit{A\text{-}BlockStatement}$, $S \in \mathit{A\text{-}BlockStatement}^*$, $b \in \mathit{A\text{-}Block}$, $h \in \mathit{A\text{-}CatchClause}$, $H \in \mathit{A\text{-}CatchClause}^*$, $c \in \mathit{Results}$, and $q \in \mathit{AbruptResults}$.

3.4 "Silent" Actions

We call *Load*, *Store*, *Read* and *Write* the "silent" actions because they may spontaneously occur during the execution of a Java program without the intervention of any thread's execution engine (no term evaluation). In some cases such an occurrence is subject to the previous occurrence of other actions. In

the operational semantics, the relevant "historical" information is recorded in a configuration's event space. Note that, given an event space η and an action a, only if $\eta \oplus a$ is defined, and hence the occurrence of a in η complies with the requirements of the language specification, can a rule $\eta \longrightarrow \eta \oplus a$ be fired. This point is crucial for a correct understanding of the rules [read, load, store, write] for silent actions given in Table 3, as well as [assign5, access3] of Table 4 and [syn2, syn4] of Table 8.

The same argument explains how is the [store] rule able to "guess" the right value to be stored: the axioms (6) and (9) of Section 2 guarantee that the apparently arbitrary value v in $\eta \longrightarrow \eta \oplus (\mathit{Store}, \theta, l, v)$ is in fact the latest value assigned by θ to l. In Section 4, changing the event space axioms, we let [store] make a *real* guess on v, by looking "presciently" into the future.

[read][1]	$T, \eta, \mu \longrightarrow T, \eta \oplus (\mathit{Read}, \theta, l, \mu(l)), \mu$
[load][1]	$T, \eta \longrightarrow T, \eta \oplus (\mathit{Load}, \theta, l, v)$
[store][1]	$T, \eta \longrightarrow T, \eta \oplus (\mathit{Store}, \theta, l, v)$
[write][1]	$T, \eta, \mu \longrightarrow T, \eta \oplus (\mathit{Write}, \theta, l, v), \mu[l \mapsto v]$

[1] if $T(\theta)$ is defined

Table 3. "Silent" actions

3.5 Expressions

Table 4 contains the rules for expressions.

To evaluate the assignment to an instance variable successfully, the left hand side is evaluated first by repeatedly applying [assign1], until a left value is produced. Then the right hand side is evaluated by [assign3], and the assignment of the resulting value is recorded in the event space by [assign5]. Note that [assign1] does not apply to an assignment $e_1 = e$ when e_1 is a left value l because, even though l may further evaluate to a right value v by [access3], $v = e$ would not be a legal abstract term. The same argument applies below to rules such as [syn3] and so forth. Note that evaluating $\mathit{null}.f$ to $\mathit{throw}(o)$ in rule [access2] would not allow exceptions thrown to the left hand side of an assignment to propagate outward in the structure of the program (see Section 3.8). To wit, $\mathit{throw}(o)$ is an expression result while $\mathit{throw}(o).f$ can be viewed as an "*A-LeftHandSide* result."

The rules [assign2] and [assign4] deal with assignments to local variables. In the present semantics an attempt to access a field of the *null* object raises a

`NullPointerException` [access2]. A more elaborate treatment is required when *static* fields are considered (see [10, §15.10.1]).

The evaluation of a method invocation $e.m(e_1, \ldots, e_k)$ is done in three steps: First e, $e_1, \ldots, e_k$ are evaluated (in this order). If evaluation is successful, the actual method to be invoked is then determined from m and from the type of the object denoted by e. We deal with non-successful evaluations in Section 3.8. Finally, the actual method call is performed. We assume that the run-time retrieval of methods is performed by a function

$$\mathit{methodBody} : \mathit{ClassType} \times \mathit{MethodIdentifier} \times \mathit{Store} \rightharpoonup \mathit{A\text{-}Block} \times \mathit{Identifier}^*$$

which receives in input the class of the object for which the method is being invoked, a method identifier m and a store (containing the class declarations), and returns, together with the body of m, the list of its formal parameters. This function is partial: $\mathit{methodBody}(C, m, \mu)$, where $m = (i, C', \mathcal{T}, \tau)$, is undefined if no *user-defined* method i with signature $\mathcal{T} \to \tau$ can be found in μ, inspecting the classes which lie between C and C' in the class hierarchy. In that case m could still be a Java *built-in* method, like `start` or `stop`, otherwise a compile time error would have occured. Separate operational rules are provided for built-in methods (see Table 12 for example). Note that all such rules are subject to the condition that *methodBody* is undefined (which it must be for *final* methods), thus implementing method overriding.

Method calls produce *activation frames*, the elements of *ActivationFrame* in Table 1. The block of a frame represents the body of the invoked method. Activation frames are produced at run-time by the function

$$\mathit{frame} : \mathit{Obj} \times \mathit{MethodIdentifier} \times \mathit{RVal}^* \times \mathit{Store} \rightharpoonup \mathit{ActivationFrame}$$

defined as follows: $\mathit{frame}(o, m, V, \mu) = (m, \{S\}_{\rho[\mathit{this} \mapsto o][I \mapsto V]})$, for an object o of type C, if $\mathit{methodBody}(C, m, \mu) = (\{S\}_\rho, I)$; otherwise it is undefined. Note that, since the type of the null object has no name (see [10, §4.1]), *frame* is always undefined when applied to *null*. Since it is the "static" information contained in μ which is used by *frame*, we generally leave this parameter implicit. The operational rules for evaluating activation frames are given in Table 5.

Start configuration. Let C be the only class in a program called P to be *public*, and let the compilation of P produce an initial store $\mu_\emptyset$ recording all relevant type information. Let C have a method `main` with a string parameter (this is a simplifying assumption: Java requires an *array* of strings, but arrays are not treated in this paper). We understand that a command line "`java` P *arg*" given as input to the computer produces a start configuration

$$(\theta, (\texttt{main}, \{\, S \,\}_{\rho[i \mapsto v]}), \sigma_\emptyset), \emptyset, \mu$$

where $\emptyset$ is the empty event space, $(\theta, \mu) = \mathit{new}(\texttt{Thread}, \mu_\emptyset)$, $v = \mathit{value}(\mathit{arg})$, and $\mathit{methodBody}(C, \texttt{main}, \mu_\emptyset) = (\{\, S \,\}_\rho, i)$.

[assign1] $\dfrac{e_1 \longrightarrow e_2}{e_1 = e \longrightarrow e_2 = e}$ [assign2] $\dfrac{e_1 \longrightarrow e_2}{i = e_1 \longrightarrow i = e_2}$

[assign3] $\dfrac{e_1 \longrightarrow e_2}{l = e_1 \longrightarrow l = e_2}$ [assign4] $i = v\,, \sigma \longrightarrow v\,, \sigma[i \mapsto v]$

[assign5] $(\theta, l = v), \eta \longrightarrow (\theta, v), \eta \oplus (\mathit{Assign}, \theta, l, v)$

[access1] $\dfrac{e_1 \longrightarrow e_2}{e_1\,.\,f \longrightarrow e_2\,.\,f}$ [access2][1] $\mathit{null}\,.\,f, \mu \longrightarrow \mathit{throw}(o)\,.\,f, \mu'$

[access3] $(\theta, l), \eta \longrightarrow (\theta, v), \eta \oplus (\mathit{Use}, \theta, l, v)$

[this] $\texttt{this}\,, \sigma \longrightarrow \sigma(\mathit{this})\,, \sigma$ [var] $i\,, \sigma \longrightarrow \sigma(i)\,, \sigma$

[new] $\texttt{new}\; C\;\texttt{()}, \mu \longrightarrow \mathit{new}(C, \mu)$ [lit] $k \longrightarrow \mathit{value}(k)$

[unop1] $\dfrac{e_1 \longrightarrow e_2}{\texttt{op}\; e_1 \longrightarrow \texttt{op}\; e_2}$ [unop2] $\texttt{op}\; v \longrightarrow \mathit{op}(v)$

[binop1] $\dfrac{e_1 \longrightarrow e_2}{e_1\;\texttt{bop}\;e \longrightarrow e_2\;\texttt{bop}\;e}$ [binop2] $\dfrac{e_1 \longrightarrow e_2}{v\;\texttt{bop}\;e_1 \longrightarrow v\;\texttt{bop}\;e_2}$

[binop3] $v_1\;\texttt{bop}\;v_2 \longrightarrow \mathit{bop}(v_1, v_2)$

[parseq1] $\dfrac{e_1 \longrightarrow e_2}{e_1\,E \longrightarrow e_2\,E}$ [parseq2] $\dfrac{E_1 \longrightarrow E_2}{v\,E_1 \longrightarrow v\,E_2}$

[call1] $\dfrac{e_1 \longrightarrow e_2}{e_1.m(E) \longrightarrow e_2.m(E)}$ [call2] $\dfrac{E_1 \longrightarrow E_2}{o.m(E_1) \longrightarrow o.m(E_2)}$

[call3] $o.m(V) \longrightarrow \mathit{frame}(o, m, V)$ [call4][1] $\mathit{null}.m(V), \mu \longrightarrow \mathit{throw}(o), \mu'$

[1] where $(o, \mu') = \mathit{new}(\texttt{NullPointerException}, \mu)$

Table 4. Expressions

[frame] $\dfrac{b_1 \longrightarrow b_2}{(m, b_1) \longrightarrow (m, b_2)}$ [exit1] $(m, \{\,\})\,; \longrightarrow *$

[exit2] $(m, \{\,\mathit{return}\; S\,\})\,; \longrightarrow *$ [exit3] $(m, \{\,\mathit{return}\; v\; S\,\}) \longrightarrow v$

Table 5. Activation frames

$$[\text{decl}] \quad \frac{e_1 \longrightarrow e_2}{\tau\ i = e_1\ D; \longrightarrow \tau\ i = e_2\ D;}$$

$$[\text{locvardecl1}] \quad \tau\ i = v\, d\, D;\ ,\ \sigma \longrightarrow \tau\ d\, D;\ ,\ \sigma[i = v]$$

$$[\text{locvardecl2}] \quad \tau\ i = v\,;\ ,\ \sigma \longrightarrow *\ ,\ \sigma[i = v]$$

Table 6. Local variable declarations

$$[\text{expstat1}] \quad \frac{e_1 \longrightarrow e_2}{e_1\,; \longrightarrow e_2\,;} \qquad [\text{expstat2}] \quad v\,; \longrightarrow *$$

$$[\text{skip}] \quad ; \longrightarrow * \qquad [\text{if1}] \quad \frac{e_1 \longrightarrow e_2}{\texttt{if}(e_1)\ s \longrightarrow \texttt{if}(e_2)\ s}$$

$$[\text{if2}] \quad \texttt{if}(\mathit{true})\ s \longrightarrow s \qquad [\text{if3}] \quad \texttt{if}(\mathit{false})\ s \longrightarrow *$$

Table 7. Expression statements, skip and conditional

$$[\text{statseq}] \quad \frac{s_1 \longrightarrow s_2}{s_1\ S \longrightarrow s_2\ S} \qquad [*] \quad *\ S \longrightarrow S$$

$$[\text{block1}] \quad \{\ \} \longrightarrow *$$

$$[\text{block2}] \quad \frac{S_1, \mathit{push}(\rho_1, \sigma_1) \longrightarrow S_2, \mathit{push}(\rho_2, \sigma_2)}{\{S_1\}_{\rho_1}, \sigma_1 \longrightarrow \{S_2\}_{\rho_2}, \sigma_2}$$

$$[\text{syn1}]^1 \quad \frac{e_1 \longrightarrow e_2}{\texttt{synchronized}\ (e_1)\ b \longrightarrow \texttt{synchronized}\ (e_2)\ b}$$

$$[\text{syn2}] \quad \frac{e, \eta_1 \longrightarrow o, \eta_2}{(\theta, \texttt{synchronized}\ (e)\ b), \eta_1 \longrightarrow \texttt{synchronized}\ (o)\ b, \eta_2 \oplus (\mathit{Lock}, \theta, o)}$$

$$[\text{syn3}] \quad \frac{b_1 \longrightarrow b_2}{\texttt{synchronized}\ (o)\ b_1 \longrightarrow \texttt{synchronized}\ (o)\ b_2}$$

$$[\text{syn4}] \quad \frac{b, \eta_1 \longrightarrow c, \eta_2}{(\theta, \texttt{synchronized}\ (o)\ b), \eta_1 \longrightarrow c, \eta_2 \oplus (\mathit{Unlock}, \theta, o)}$$

[1] if $e_2 \notin \mathit{RVal}$

Table 8. Blocks and synchronization

3.6 Local Variable Declarations

The rules for local variable declarations are given in Table 6.

3.7 Statements

Table 7 contains the rules for expression statements, skip and conditional statements. Table 8 contains the rules for blocks and synchronization. The statements for control manipulation (`return` and exception `try`) are treated in Section 3.8.

Example. Consider the two threads θ_1 and θ_2 of Example 2.3 running in parallel with initially empty working memories, empty event space $\emptyset$, and stacks mapping the local variable `p` to o. We write t_2 the portion of program run by θ_2. In the example θ_1 enters its synchronized block first. Its evaluation is described in Figure 2, where stacks are omitted.

3.8 Control mechanisms

In Java, the evaluation of expressions and statements may have a *normal* or an *abrupt* completion. Abrupt completion may be caused by the occurrence of an exceptional situation during execution, such as an attempt to divide an integer by 0; it can also be forced by the program by means of a `throw` or a `return` statement. For example, the execution of `throw` e;, where the expression e evaluates to some object o, throws an exception "with reason o" to be caught by the nearest dynamically-enclosing `catch` clause of a `try` statement (see [10, §11.3]). Similarly, the execution of `return` e; returns control, together with the value of e, to the nearest dynamically-enclosing activation frame.

The interactions between these two mechanisms are described in [10, §14.15, §14.16, §14.18], to which we refer for more detail. The rules for exception handling are given in Table 9 and Table 11. Uncaught exceptions are not treated in the present paper.

Some of the rules for the `try` statement include a `finally` clause written in square brackets, to be regarded as "optional:" the brackets indicate that the clause should be ignored if the statement at hand has no `finally` block. A similar convention is adopted for the return statements and results, where `return` $[v]$; accounts for both cases where some value v is and is not returned (and similarly for the results).

Table 10 contains a grammar of syntactic contexts which pop control out upon occurrence of an abrupt evaluation result, with no further ado. Contexts of the form $\vartheta\,[\,\_\,]$, called "pop-out" contexts, are used in the rule scheme [pop] to propagate abrupt evaluation results outwards through the structure of a program. All syntactic constructs which are not represented in a pop-out context respond to such results with some computational action described by a separate semantic rule. Examples of such constructs are the `synchronized` and the `try` statements.

[pop][1] $\vartheta\,[q] \longrightarrow q$

[exit4] $(m, \{\ \mathit{throw}(o)\ S\ \}) \longrightarrow \mathit{throw}(o)$

[ret1] $$\frac{e_1 \longrightarrow e_2}{\texttt{return}\ e_1\,; \longrightarrow \texttt{return}\ e_2\,;}$$

[ret2] $\texttt{return}\ [v]\,; \longrightarrow \mathit{return}\,[v]$

[throw1] $$\frac{e_1 \longrightarrow e_2}{\texttt{throw}\ e_1\,; \longrightarrow \texttt{throw}\ e_2\,;}$$

[throw2] $\texttt{throw}\ o\,; \longrightarrow \mathit{throw}(o)$

[try1] $\texttt{try}\ \{\ \}\ H \longrightarrow *$

[try2] $\texttt{try}\ \{\ \mathit{return}[v]\ S\ \}\ H\ [\texttt{finally}\ \{\ \}] \longrightarrow \mathit{return}\,[v]$

[try3][2] $$\frac{b_1 \longrightarrow b_2}{\texttt{try}\ b_1\ H\ [\texttt{finally}\ b] \longrightarrow \texttt{try}\ b_2\ H\ [\texttt{finally}\ b]}$$

[try4][3] $$\frac{b \longrightarrow \{\ \mathit{throw}(o)\ S\ \}}{\begin{array}{l}\texttt{try}\ b\ \texttt{catch}\ (\tau\ i)\ \{\ S'\ \}_\rho\ H\ [\texttt{finally}\ b'] \longrightarrow \\ \texttt{try}\ \{\ \mathit{throw}(o)\ S\ \}\ \texttt{catch}\ (\tau\ i)\ \{\ S'\ \}_{\rho[i \mapsto o]}\ H\ [\texttt{finally}\ b']\end{array}}$$

[try5][3] $$\frac{b_1 \longrightarrow b_2}{\begin{array}{l}\texttt{try}\ \{\ \mathit{throw}(o)\ S\ \}\ \texttt{catch}\ (\tau\ i)\ b_1\ H\ [\texttt{finally}\ b] \longrightarrow \\ \texttt{try}\ \{\ \mathit{throw}(o)\ S\ \}\ \texttt{catch}\ (\tau\ i)\ b_2\ H\ [\texttt{finally}\ b]\end{array}}$$

[try6][3] $$\frac{b \longrightarrow c}{\texttt{try}\ \{\ \mathit{throw}(o)\ S\ \}\ \texttt{catch}\ (\tau\ i)\ b\ H \longrightarrow c}$$

[try7][4] $$\frac{b \longrightarrow \{\ \mathit{throw}(o)\ S\ \}}{\begin{array}{l}\texttt{try}\ b\ \texttt{catch}\ (\tau\ i)\ b' \longrightarrow \\ \texttt{try}\ \{\ \mathit{throw}(o)\ S\ \}\ \texttt{catch}\ (\tau\ i)\ b'\end{array}}$$

[try8][4] $$\frac{\texttt{try}\ b\ H_1\ [\texttt{finally}\ b_1] \longrightarrow \texttt{try}\ \{\ \mathit{throw}(o)\ S\ \}\ H_2\ [\texttt{finally}\ b_2]}{\begin{array}{l}\texttt{try}\ b\ \texttt{catch}\ (\tau\ i)\ b'\ H_1\ [\texttt{finally}\ b_1] \longrightarrow \\ \texttt{try}\ \{\ \mathit{throw}(o)\ S\ \}\ \texttt{catch}\ (\tau\ i)\ b'\ H_2\ [\texttt{finally}\ b_2]\end{array}}$$

[try9][4] $\texttt{try}\ \{\ \mathit{throw}(o)\ S\ \}\ \texttt{catch}\ (\tau\ i)\ b\ [\texttt{finally}\ \{\ \}] \longrightarrow \mathit{throw}(o)$

[try10][4] $$\frac{\texttt{try}\ \{\ \mathit{throw}(o)\ S\ \}\ H\ [\texttt{finally}\ b] \longrightarrow c}{\texttt{try}\ \{\ \mathit{throw}(o)\ S\ \}\ \texttt{catch}\ (\tau\ i)\ b\ H\ [\texttt{finally}\ b] \longrightarrow c}$$

[1] where $\vartheta\,[\_]$ is a "pop-out" context
[2] if $b_2 \neq \{\ \mathit{throw}(o)\ S\ \}$
[3] if $o \in \tau$
[4] if $o \notin \tau$

Table 9. Exceptions and `return`

$$
\begin{array}{lcl}
\vartheta\,[\_] & ::= & [\_].f = e \quad | \quad i = [\_] \quad | \quad l = [\_] \\
& & | \ \texttt{op}\,[\_] \quad | \quad [\_]\ \texttt{bop}\ e \quad | \quad v\ \texttt{bop}\ [\_] \\
& & | \ [\_].f \quad | \quad [\_].m(E) \quad | \quad o.m(\xi\,[\_]) \\
& & | \ \tau\ i = [\_]\,D; \quad | \quad [\_]; \quad | \quad \{\,[\_]\,S\,\} \\
& & | \ \texttt{if}\ ([\_])\ s \quad | \quad \texttt{return}\,[\_]; \\
& & | \ \texttt{throw}\,[\_]; \quad | \quad \texttt{synchronized}([\_])\ b \\
& & | \ \texttt{try}\ \{\ \}\ H\ \texttt{finally}\ \{\,[\_]\,S\,\} \\
& & | \ \texttt{try}\ \{\,\mathit{throw}(o)\,S\,\}\ \texttt{finally}\ \{\,[\_]\,S'\,\} \\
& & | \ \texttt{try}\ \{\,\mathit{throw}(o)\,S\,\}\ \texttt{catch}\ (\tau\ i)\ \{\,[\_]\,S'\,\}\ H\ \texttt{finally}\ \{\,\} \quad \text{if } o \in \tau \\
& & | \ \texttt{try}\ \{\,\mathit{throw}(o)\,S\,\}\ \texttt{catch}\ (\tau\ i)\ \{\,q'S'\,\}\ H\ \texttt{finally}\ \{\,[\_]\ S''\,\} \quad \text{if } o \in \tau \\
& & | \ \texttt{try}\ \{\,\mathit{return}[v]\ S\,\}\ H\ \texttt{finally}\ \{\,[\_]\,S'\,\} \\
\xi\,[\_] & ::= & [\_]\ E \mid v\ \xi\,[\_]
\end{array}
$$

Table 10. "Pop-out" contexts

$$
\begin{array}{ll}
[\text{fin1}] & \dfrac{b_1 \longrightarrow b_2}{\texttt{try}\ \{\ \}\ H\ \texttt{finally}\ b_1 \longrightarrow \texttt{try}\ \{\ \}\ H\ \texttt{finally}\ b_2} \\[2ex]
[\text{fin2}] & \dfrac{b \longrightarrow c}{\texttt{try}\ \{\ \}\ H\ \texttt{finally}\ b \longrightarrow c} \\[2ex]
[\text{fin3}] & \dfrac{b_1 \longrightarrow b_2}{\begin{array}{l}\texttt{try}\ \{\,\mathit{return}\,[v]\ S\,\}\ H\ \texttt{finally}\ b_1 \longrightarrow \\ \texttt{try}\ \{\,\mathit{return}\,[v]\ S\,\}\ H\ \texttt{finally}\ b_2\end{array}} \\[2ex]
[\text{fin4}]^1 & \dfrac{b_1 \longrightarrow b_2}{\begin{array}{l}\texttt{try}\ \{\,\mathit{throw}(o)\,S\,\}\ \texttt{catch}\ (\tau\ i)\ \{\ \}\ H\ \texttt{finally}\ b_1 \longrightarrow \\ \texttt{try}\ \{\,\mathit{throw}(o)\,S\,\}\ \texttt{catch}\ (\tau\ i)\ \{\ \}\ H\ \texttt{finally}\ b_2\end{array}} \\[2ex]
[\text{fin5}]^1 & \dfrac{b \longrightarrow c}{\texttt{try}\ \{\,\mathit{throw}(o)\,S\,\}\ \texttt{catch}\ (\tau\ i)\ \{\ \}\ H\ \texttt{finally}\ b \longrightarrow c} \\[2ex]
[\text{fin6}]^1 & \dfrac{b_1 \longrightarrow b_2}{\begin{array}{l}\texttt{try}\ \{\,\mathit{throw}(o)\,S\,\}\ \texttt{catch}\ (\tau\ i)\ \{\,q\,S'\,\}\ H\ \texttt{finally}\ b_1 \longrightarrow \\ \texttt{try}\ \{\,\mathit{throw}(o)\,S\,\}\ \texttt{catch}\ (\tau\ i)\ \{\,q\,S'\,\}\ H\ \texttt{finally}\ b_2\end{array}} \\[2ex]
[\text{fin7}] & \dfrac{b \longrightarrow \{\,\mathit{throw}(o)\,S\,\}}{\texttt{try}\ b\ \texttt{finally}\ b' \longrightarrow \texttt{try}\ \{\,\mathit{throw}(o)\,S\,\}\ \texttt{finally}\ b'} \\[2ex]
[\text{fin8}] & \dfrac{b_1 \longrightarrow b_2}{\begin{array}{l}\texttt{try}\ \{\,\mathit{throw}(o)\,S\,\}\ \texttt{finally}\ b_1 \longrightarrow \\ \texttt{try}\ \{\,\mathit{throw}(o)\,S\,\}\ \texttt{finally}\ b_2\end{array}}
\end{array}
$$

[1] if $o \in \tau$

Table 11. `finally`

$$(\theta_1,\ \texttt{synchronized}(\texttt{p})\,\{\,\texttt{p.y} = 2;\,\}\ \ldots) \mid (\theta_2,\ t_2),\ \emptyset,\ \mu$$
$$\text{by [statseq, } \downarrow \text{ syn2, var]}$$
$$(\theta_1,\ \texttt{synchronized}(o)\,\{\,\texttt{p.y} = 2;\,\}\ \ldots) \mid (\theta_2,\ t_2),\ \{(\mathit{Lock}, \theta_1, o)\},\ \mu$$
$$\text{by [statseq, syn3, block2, statseq, } \downarrow \text{ expstat1, assign1, access1, var]}$$
$$(\theta_1,\ \texttt{synchronized}(o)\,\{\,o.\texttt{y} = 2;\,\}\ \ldots) \mid (\theta_2,\ t_2),\ \{(\mathit{Lock}, \theta_1, o)\},\ \mu$$
$$\text{by [statseq, syn3, block2, statseq, } \downarrow \text{ expstat1, assign3, lit]}$$
$$(\theta_1,\ \texttt{synchronized}(o)\,\{\,o.\texttt{y} = 2;\,\}\ \ldots) \mid (\theta_2,\ t_2),\ \{(\mathit{Lock}, \theta_1, o)\},\ \mu$$
$$\text{by [statseq, syn3, block2, statseq, } \downarrow \text{ expstat1, assign5]}$$
$$(\theta_1,\ \texttt{synchronized}(o)\,\{\,2;\,\}\ \ldots) \mid (\theta_2,\ t_2),\ \{\cdots \leq (\mathit{Assign}, \theta_1, o.\texttt{y}, 2)\},\ \mu$$
$$\text{by [statseq, syn3, block2, statseq } \downarrow \text{ expstat2]}$$
$$(\theta_1,\ \texttt{synchronized}(o)\,\{\,*\,\}\ \ldots) \mid (\theta_2,\ t_2),\ \{\cdots \leq (\mathit{Assign}, \theta_1, o.\texttt{y}, 2)\},\ \mu$$
$$\text{by [statseq, syn3, block2, } \downarrow *]$$
$$(\theta_1,\ \texttt{synchronized}(o)\,\{\ \}\ \ldots) \mid (\theta_2,\ t_2),\ \{\cdots \leq (\mathit{Assign}, \theta_1, o.\texttt{y}, 2)\},\ \mu$$
$$\downarrow \text{ by [store]}$$
$$(\theta_1,\ \texttt{synchronized}(o)\,\{\ \}\ \ldots) \mid (\theta_2,\ t_2),\ \{\cdots \leq (\mathit{Store}, \theta_1, o.\texttt{y}, 2)\},\ \mu$$
$$\downarrow \text{ by [write]}$$
$$(\theta_1,\ \texttt{synchronized}(o)\,\{\ \}\ \ldots) \mid (\theta_2,\ t_2),\ \{\cdots \leq (\mathit{Write}, \theta_1, o.\texttt{y}, 2)\},\ \mu[o.\texttt{y} \mapsto 2]$$
$$\text{by [statseq, } \downarrow \text{ syn4, block1]}$$
$$(\theta_1,\ *\texttt{a} = \texttt{p.x};\ \ldots) \mid (\theta_2,\ t_2),\ \{\cdots \leq (\mathit{Unlock}, \theta_1, o)\},\ \mu[o.\texttt{y} \mapsto 2]$$
$$\downarrow \text{ by [*]}$$
$$(\theta_1,\ \texttt{a} = \texttt{p.x};\ \ldots) \mid (\theta_2,\ t_2),\ \{\cdots \leq (\mathit{Unlock}, \theta_1, o)\},\ \mu[o.\texttt{y} \mapsto 2]$$
$$(\ldots)$$

Figure 2. Run of Example 2.3

3.9 Starting and Stopping Threads

The notion of configuration introduced in Section 3.3 is extended here to include a set Θ of thread identifiers, whose elements identify threads which are bound to stop. We write $\Theta \mid \theta$ for $\Theta \cup \{\theta\}$ when we assume that θ is not in Θ. A *configuration* is now redefined to be a 4-tuple of the form:

$$(T, \Theta, \eta, \mu).$$

All operational rules introduced so far have no interaction with the mechanism for stopping threads; in view of the conventions introduced in Section 3.3, by which parts of a configuration may be left implicit when not directly involved in the evaluation, the rules of the previous sections can be read with no editing in the new operational setting with Θ.

Table 12 presents the rules for the methods `start()` and `stop()` of the class `Thread`. The interplay of stopping threads and Java's notification system is discussed in Section 3.10.

$$[\text{start1}]^{1,2} \qquad \theta.\texttt{start}(); , \Theta \longrightarrow * \mid (\theta, \mathit{frame}(\theta, \texttt{run}, ()); , \sigma_\emptyset), \Theta$$

$$[\text{start2}]^{1,3} \qquad \theta.\texttt{start}(); , \Theta \longrightarrow * \mid (\theta, *, \sigma_\emptyset), \Theta \setminus \{\theta\}$$

$$[\text{start3}]^{1,4,5} \qquad T \mid (\theta', \theta.\texttt{start}();), \mu \longrightarrow T \mid (\theta', \mathit{throw}(o)), \mu'$$

$$[\text{stop1}] \qquad \theta.\texttt{stop}(); , \Theta \longrightarrow *, \Theta \cup \{\theta\}$$

$$[\text{stop2}]^{6} \qquad (\theta, t), \Theta \mid \theta, \mu \longrightarrow (\theta, \mathit{throw}(o)), \Theta, \mu'$$

[1] if $\mathit{frame}(\theta, \texttt{start}, ())$ is undefined
[2] if $\theta \notin \Theta$
[3] if $\theta \in \Theta$ or $\mathit{frame}(\theta, \texttt{run}, ())$ is undefined
[4] if $T(\theta)$ is defined or $\theta = \theta'$
[5] where $(o, \mu') = \mathit{new}(\texttt{IllegalThreadStateException}, \mu)$
[6] where t is a redex and $(o, \mu') = \mathit{new}(\texttt{ThreadDeath}, \mu)$

Table 12. `start()` and `stop()`

The rules [start1], [start2] and [start3] can only be applied if the method `start()` has not been overloaded, that is if $\mathit{frame}(\theta, \texttt{start}, ())$ is undefined. Since `stop()` is declared as *final* in class `Thread` and thus cannot be redefined, no analogous side condition is required in the rules for `stop()`. The rule [start1] only applies if no thread with the same identifier as the one to be started is currently running; this is implicit in the use of "|". If such a thread identifier exists an `IllegalThreadStateException` is thrown by [start3].

If a thread θ is started and $\mathit{frame}(\theta, \texttt{run}, ())$ is undefined, the built-in `run` method of the class `Thread` is invoked. The latter simply calls the `run` method of θ's *run object*, that is the runnable object given as argument to the expression that created θ [10, §20.20], if such an object exists, and do nothing otherwise [10, §20.20.13]. Since, for simplicity, we only consider class instance creation

expressions with empty parameter list, and hence have no run objects associated with threads, θ does nothing when started if $\mathit{frame}(\theta, \texttt{run}, ())$ is undefined. This explains [start2]. This rule also captures the case of a thread which has been stopped before having ever been started (indeed possible in Java [10, §20.20.15]). If the thread is eventually started, it will immediately terminate and its name removed from Θ.

As a result of the invocation of a `stop` method of class `Thread` an *asynchronous* exception is thrown. Java allows a small but bounded amount of execution to occur between the method call and the actual throw of the exception [10, §11.3.2]. We allow such execution to be arbitrarily long: at any time during execution a thread whose `stop` method has been invoked (by [stop1]) may decide that the time has come to throw a `ThreadDeath` exception. The exception is thrown by [stop2] as deep inside the structure of the program as is necessary to allow a catch by a possibly enclosing `try-catch` statement. This is ensured by the side condition that t is a *redex*. These are the terms of the form:

$$\begin{aligned} \mathit{Redex} ::= \; & i = v \mid l = v \mid \mathit{null}.f \mid \mathit{null}.f = e \mid l \mid \texttt{this} \mid i \mid \texttt{new}\, C\,() \,; \mid k \\ & \mid \texttt{op}\; v \mid v_1\; \texttt{bop}\; v_2 \mid o.m\,(V) \mid \tau\; i = v\; d\, D \,; \mid \tau\; i = v \,; \mid v \,; \mid \,; \\ & \mid \texttt{if}\,(v)\, s \mid \{\,\} \mid (m, \{\,\}) \mid \texttt{throw}\,(o) \,; \mid \texttt{return}\,[v] \,; \mid \texttt{try}\{\,\}\, H \\ & \mid \theta.\texttt{start}\,() \,; \mid \theta.\texttt{stop}\,() \,; \mid o.\texttt{wait}\,() \,; \mid o.\texttt{notify}\,() \,; \end{aligned}$$

As *throw* v and *return* $[v]$ are not contained in this list of redices, a thread cannot stop as long as it is performing a transfer of control, i.e. performing pop-out rules.

A more committed policy for stopping threads may be adopted either by requiring *fairness* on [stop2] or by enforcing such a condition by means of a counter binding the amount of execution steps allowed before this rule is applied.

No rule removes threads from a configuration: when they finish execution, threads keep dwelling in an M-term together with the result that they produced.

3.10 Wait and notification

In Java every object has a "wait set." A thread θ who owns at least one, say n locks on an object o can add itself on that object's wait set by invoking o.`wait`(). This thread would then lose all its locks on o and lie dormant until some other thread wakes it up by invoking o.`notify`(). Before resuming computation, θ must get its n locks back, possibly competing with other threads in the usual manner. When a thread goes to sleep in a wait set it is said to change its *state* from *running* to *waiting*. When notified, such a thread changes its state from *waiting* to *notified*, and finally from *notified* to *running* when it obtains its locks back.

Let the letters R, W and N stand respectively for running, waiting and notified. The notion of M-term introduced in Section 3.3 is extended here by endowing each thread with a *record* of its state. The record of a running thread consists just of the identifier R. The record of a thread which is waiting or notified consists of a triple (X, o, n), where X is the identifier W or N, o is the

object on whose wait set the thread is waiting and n is the number of locks that the thread acquired on that object.

An M-term is now redefined to be a partial function mapping thread identifiers to triples (t, ϵ, σ), where t and σ are as before and ϵ is a state record. The notation $T \mid (\theta, t, \epsilon, \sigma)$ extends that of Section 3.3 in the obvious way. When ϵ is a triple (X, o, n) we write $T \mid (\theta, t, X, o, n, \sigma)$ for $T \mid (\theta, t, (X, o, n), \sigma)$ and omit the parts that are not immediately relevant as usual when no confusion arises.

The operational rules introduced so far apply to M-terms of the new form by agreeing that, unless otherwise specified, evaluation applies to running threads (which can nevertheless change state when evaluated). More precisely: if the state record of a thread is omitted in the left hand side of a judgement, then it is understood to be R. For example, [expstat1] is now read

$$\frac{T_1 \mid (\theta, e_1, R, \sigma_1), \Theta_1, \eta_1, \mu_1 \longrightarrow T_2 \mid (\theta, e_2, \epsilon, \sigma_2), \Theta_2, \eta_2, \mu_2}{T_1 \mid (\theta, e_1\,;, R, \sigma_1), \Theta_1, \eta_1, \mu_1 \longrightarrow T_2 \mid (\theta, e_2\,;, \epsilon, \sigma_2), \Theta_2, \eta_2, \mu_2}$$

while [skip] is now read $T \mid (\theta, ;, R, \sigma), \Theta, \eta, \mu \longrightarrow T \mid (\theta, *, R, \sigma), \Theta, \eta, \mu$. Moreover, silent actions only apply to running threads; more precisely: the side condition in Table 3 changes now to "if $T(\theta) = (t, R)$." Finally, threads run when started, that is: the state of θ in the right hand side of [start1] and [start2] is R.

[wait1][1]	$(\theta, o.\texttt{wait}();), \eta, \mu \longrightarrow (\theta, \mathit{throw}(o')), \eta, \mu'$
[wait2][2]	$(\theta, o.\texttt{wait}();, R), \eta \longrightarrow (\theta, *, W, o, n), \eta \oplus (\mathit{Unlock}, \theta, o)^n$
[notify1][1]	$(\theta, o.\texttt{notify}();), \eta, \mu \longrightarrow (\theta, \mathit{throw}(o')), \eta, \mu'$
[notify2][2]	$(\theta, o.\texttt{notify}();) \mid (t, W, o), \eta \longrightarrow (\theta, *) \mid (t, N, o), \eta$
[notify3][3]	$T \mid o.\texttt{notify}(); \longrightarrow T \mid *$
[ready]	$(\theta, t, N, o, n), \eta \longrightarrow (\theta, t, R), \eta \oplus (\mathit{Lock}, \theta, o)^n$
[stop3]	$(\theta, t, W), \Theta \mid \theta \longrightarrow (\theta, t, N), \Theta \mid \theta$

[1] if $\mathit{locks}(\theta, o, \eta) = 0$ and $(o', \mu') = \mathit{new}(\texttt{IllegalMonitorException}, \mu)$
[2] if $\mathit{locks}(\theta, o, \eta) = n > 0$
[3] if $T(\theta) \neq (W, o)$ for all θ

Table 13. Wait sets and notification

The rules for the notification system are given in Table 13.

By the rules [wait1] and [notify1], an appropriate exception is thrown if a thread attempts to operate on the wait set of an object on which it possesses no locks. The expression $\mathit{locks}(\theta, o, \eta)$ denotes the number of locks that a thread θ possesses on o in an event space η (the number of events $(\mathit{Lock}, \theta, o)$ with no matching Unlock). By the rule [wait2] a thread θ can put itself in the wait set of an object o. This step involves the release by θ of all its locks on o. Rule [notify2] notifies a thread waiting in the wait set of an object o. Such a thread, however, cannot run until all its locks on o are restored. This is done by [ready].

Any notification on an object whose wait set is empty has no effect ([notify3]). A waiting thread which has been stopped is woken up by [stop3].

Example. Figure 3 illustrates the interaction of the rules for wait and notification. Consider the M-term

$$(\theta, \texttt{synchronized(p) \{ if(c) p.wait(); \}}, \sigma) \mid$$
$$(\theta', \texttt{synchronized(p) \{ p.notify(); \}}, \sigma').$$

Let $t = \texttt{synchronized}(o)\ \{ * \}$ and $t' = \texttt{synchronized(p) \{ p.notify(); \}}$, let $\emptyset$ be the empty event space and $\eta = \{(\mathit{Lock}, \theta, o) \leq (\mathit{Unlock}, \theta, o)\}$; let σ and σ' be stacks with $\sigma(\texttt{p}) = \sigma'(\texttt{p}) = o$ and $\sigma(\texttt{c}) = \mathit{true}$. The stacks, which do not change during execution, are omitted in the figure.

4 Prescient Event Spaces

The aim of this section is the formalization of the so-called "prescient stores" of [10, §17.8] in our event space semantics. The specification claims that the "prescient" semantics is conservative for "properly synchronized" programs. We also formalize the intuitive notion of "proper synchronization" and prove this claim.

The *prescient* store actions are introduced in [10, §17.8, p. 408] as follows:

> " ... the *store* action [of variable V by thread T is allowed] to instead occur before the *assign* action, if the following restrictions are obeyed:
>
> - If the *store* action occurs, the *assign* is bound to occur. ...
> - No *lock* action intervenes between the relocated *store* and the *assign*.
> - No *load* of V intervenes between the relocated *store* and the *assign*.
> - No other *store* of V intervenes between the relocated *store* and the *assign*.
> - The *store* action sends to the main memory the value that the *assign* action will put into the working memory of thread T.
>
> The last property inspires us to call such an early *store* action *prescient*: ... "

This section is an improved and corrected version of [17].

4.1 Prescient Event Space Rules

The specification of prescient stores [10, §17.8] seems to assume that it is known which *Store* events are prescient and which prescient *Store* event is matched by which *Assign* event (as if they would be e.g. re-arrangements of *Store* actions in the old sense). We do not assume such knowledge but adopt a more general approach introducing so-called labellings that allow us to use the "old" *Store* and *Assign* events as introduced in Section 2.1 with an additional "labelling" that states whether they are prescient or not. These labellings are not necessarily unique but it is always possible to infer a labelling at run time. It will turn

$$(\theta,\ \texttt{synchronized(p) \{ if(c) p.wait(); \}}, R) \mid (\theta',\ t', R), \emptyset$$

$$\Big\downarrow \text{ by [syn2, var]}$$

$$(\theta,\ \texttt{synchronized}(o)\ \{\ \texttt{if(c) p.wait();}\ \}, R) \mid (\theta',\ t', R), \{(\mathit{Lock}, \theta, o)\}$$

$$\text{by [syn3, block2, statseq,} \quad \Big\downarrow \text{ if1, var]}$$

$$(\theta,\ \texttt{synchronized}(o)\ \{\ \texttt{if}(\mathit{true})\ \texttt{p.wait();}\ \}, R) \mid (\theta',\ t', R), \{(\mathit{Lock}, \theta, o)\}$$

$$\text{by [syn3, block2, statseq,} \quad \Big\downarrow \text{ if2]}$$

$$(\theta,\ \texttt{synchronized}(o)\ \{\ \texttt{p.wait();}\ \}, R) \mid (\theta',\ t', R), \{(\mathit{Lock}, \theta, o)\}$$

$$\text{by [syn3, block2, statseq,} \quad \Big\downarrow \text{ expstat1, call1, var]}$$

$$(\theta,\ \texttt{synchronized}(o)\ \{\ o\texttt{.wait();}\ \}, R) \mid (\theta',\ t', R), \{(\mathit{Lock}, \theta, o)\}$$

$$\text{by [syn3, block2, statseq,} \quad \Big\downarrow \text{ wait2]}$$

$$(\theta'\ t', R) \mid (\theta, \texttt{syn'd}(o)\ \{\ *\ \}, W, o, 1), \{(\mathit{Lock}, \theta, o) \leq (\mathit{Unlock}, \theta, o)\}$$

$$= \Big\downarrow$$

$$(\theta',\ \texttt{synchronized(p) \{ p.notify(); \}}) \mid (\theta, t, W, o, 1), \eta$$

$$\Big\downarrow \text{ by [syn2, var]}$$

$$(\theta',\ \texttt{synchronized}(o)\ \{\ \texttt{p.notify();}\ \}, R) \mid (\theta, t, W, o, 1), \eta \oplus (\mathit{Lock}, \theta', o)$$

$$\text{by [syn3, block2, statseq,} \quad \Big\downarrow \text{ expstat1, call1, var]}$$

$$(\theta',\ \texttt{synchronized}(o)\ \{\ o\texttt{.notify();}\ \}, R) \mid (\theta, t, W, o, 1), \eta \oplus (\mathit{Lock}, \theta', o)$$

$$\text{by [syn3, block2, statseq,} \quad \Big\downarrow \text{ notify2]}$$

$$(\theta',\ \texttt{syn'd}(o)\ \{\ *\ \}, R) \mid (\theta, \texttt{syn'd}(o)\ \{\ *\ \}, N, o, 1), \eta \oplus (\mathit{Lock}, \theta', o)$$

$$\text{by [syn3, block2,} \quad \Big\downarrow *]$$

$$(\theta',\ \texttt{syn'd}(o)\ \{\ \}, R) \mid (\theta, \texttt{syn'd}(o)\ \{\ *\ \}, N, o, 1), \eta \oplus (\mathit{Lock}, \theta', o)$$

$$\Big\downarrow \text{ by [syn4, block1]}$$

$$(\theta',\ *, R) \mid (\theta, \texttt{synchronized}(o)\ \{\ *\ \}, N, o, 1), \eta \oplus \cdots \oplus (\mathit{Unlock}, \theta', o)$$

$$\Big\downarrow \text{ by [ready]}$$

$$(\theta',\ *, R) \mid (\theta,\ \texttt{synchronized}(o)\ \{\ *\ \}, R), \eta \oplus \cdots \oplus (\mathit{Lock}, \theta, o)$$

$$\text{by [syn3, block2,} \quad \Big\downarrow \text{ by } *]$$

$$(\theta',\ *, R) \mid (\theta,\ \texttt{synchronized}(o)\ \{\ \}, R), \eta \oplus \cdots \oplus (\mathit{Lock}, \theta, o)$$

$$\Big\downarrow \text{ by [syn4, block1]}$$

$$(\theta',\ *, R) \mid (\theta,\ *, R), W, \eta \oplus \cdots \oplus (\mathit{Unlock}, \theta, o)$$

Figure 3. Interaction of `wait()` and `notify()`

out, however, that the semantics is independent of the choice of labellings, see Corollary 4.7.

Prescient event spaces are defined, on the one hand, by a relaxation of the event space rules: All rules which forbid prescient stores are cancelled and used instead to define inductively a predicate that tells whether a *Store* event is *necessarily* prescient. But, on the other hand, we have to add some rules to ensure that a prescient *Store* corresponds to a relocated *Store* that obeyes the old event space rules.

First, we define an abbreviation for the maximal event of type (A, θ, l), irrelevant of its fourth component, occurring before some other event a, and thus write $(A, \theta, l) \leq_L a$ if

$$(A, \theta, l) \leq a \wedge \big((A, \theta, l)' \leq a \Rightarrow (A, \theta, l)' \leq (A, \theta, l)\big)$$

If we write, however, $(\mathit{Store}, \theta, l, v) \leq_L (\mathit{Assign}, \theta, l, v)$, i.e. both events are written with their values and those are *identical*, then we mean the maximal $(\mathit{Store}, \theta, l, v)$ event *with value* v before $(\mathit{Assign}, \theta, l, v)$.

We define $\mathit{prescient}_\eta(\mathit{Store}, \theta, l)$ to be valid if one of the rules (P1–P7) below holds. The subscript η is usually omitted if it is clear from the context. Note that $\Phi \not\Rightarrow \Psi$ abbreviates $\neg(\Phi \Rightarrow \Psi)$ where we use the conventions of Section 2.2, i.e. $\neg(\Phi \Rightarrow \Psi)$ is short for $\neg\forall \mathbf{a} \,.\, (\Phi \Rightarrow \exists \mathbf{b} \,.\, \Psi)$ where $\mathbf{a}$ and $\mathbf{b}$ are lists of events and $\mathbf{a}$ contains precisely all events occurring in Φ except the bound $(\mathit{Store}, \theta, l)$ event.

$$(\mathit{Store}, \theta, l)' \leq (\mathit{Store}, \theta, l) \not\Rightarrow (\mathit{Store}, \theta, l)' \leq (\mathit{Assign}, \theta, l) \leq (\mathit{Store}, \theta, l) \tag{P1}$$

$$(\mathit{Store}, \theta, l) \not\Rightarrow (\mathit{Assign}, \theta, l) \leq (\mathit{Store}, \theta, l) \tag{P2}$$

$$\begin{aligned}&(\mathit{Assign}, \theta, l, v') \leq (\mathit{Store}, \theta, l, v) \not\Rightarrow \\ &\quad (\mathit{Assign}, \theta, l, v') \leq (\mathit{Assign}, \theta, l)' \leq (\mathit{Store}, \theta, l, v)\end{aligned} \tag{P3}$$

$$(\mathit{Lock}, \theta) \leq (\mathit{Store}, \theta, l) \not\Rightarrow (\mathit{Lock}, \theta) \leq (\mathit{Assign}, \theta, l) \leq (\mathit{Store}, \theta, l) \tag{P4}$$

$$\begin{aligned}&(\mathit{Store}, \theta, l)' \leq (\mathit{Store}, \theta, l) \wedge \mathit{prescient}((\mathit{Store}, \theta, l)') \not\Rightarrow \\ &\quad (\mathit{Store}, \theta, l)' \leq (\mathit{Assign}, \theta, l) \leq (\mathit{Assign}, \theta, l)' \leq (\mathit{Store}, \theta, l)\end{aligned} \tag{P5}$$

$$\begin{aligned}&(\mathit{Store}, \theta, l, v) \leq_L (\mathit{Assign}, \theta, l, v) \leq_L (\mathit{Load}, \theta, l) \not\Rightarrow \\ &\quad (\mathit{Assign}, \theta, l, v) \leq (\mathit{Store}, \theta, l, v)' \leq (\mathit{Load}, \theta, l)\end{aligned} \tag{P6}$$

$$\begin{aligned}&(\mathit{Store}, \theta, l, v) \leq_L (\mathit{Assign}, \theta, l, v) \leq_L (\mathit{Unlock}, \theta) \not\Rightarrow \\ &\quad (\mathit{Assign}, \theta, l, v) \leq (\mathit{Store}, \theta, l, v)' \leq \mathit{write\_of}((\mathit{Store}, \theta, l, v)') \\ &\qquad \leq (\mathit{Unlock}, \theta) \ \wedge \neg \mathit{prescient}((\mathit{Store}, \theta, l, v)')\end{aligned} \tag{P7}$$

Rules (P1–P4) are the negations of (4), (6), (9), and (17), respectively, that forbid prescient *Store* events. Rule (P5) is sound because if there is only one $(\mathit{Assign}, \theta, l, v)$ between two stores and the first is *prescient*, then by re-arranging the prescient *Store* two *Store* events would follow each other without a triggering *Assign* in between, which contradicts the old semantics. Rules (P6–P7) ensure

that in cases where old event space rules (3) and (15) are violated, still a relocated (i.e. *prescient*) *Store* exists which is responsible for storing the right value. So e.g. (P6) states that if any *Store* between the last *Assign* before a *Load* and the *Load* itself is necessarily prescient, then the last *Store* before the *Assign* must also be prescient and thus responsible for fulfilling old (3) when relocated. Note that it is sufficient to consider the last *Assign* before the *Load* (and the *Unlock*, respectively).

With respect to the other (old) event space laws, we keep rules (1), (2), (5), (7–8), (10–14), and (16).

Rule (3) has to be adapted as follows, allowing prescient *Store*s on the right hand side of an implication:

$$\begin{aligned}&(\mathit{Assign},\theta,l,v) \leq_L (\mathit{Load},\theta,l) \Rightarrow \\ &\quad \big((\mathit{Assign},\theta,l,v) \leq (\mathit{Store},\theta,l,v) \leq (\mathit{Load},\theta,l) \big) \vee \\ &\quad \big((\mathit{Store},\theta,l,v) \leq_L (\mathit{Assign},\theta,l,v) \leq_L (\mathit{Load},\theta,l) \big)\end{aligned} \tag{3'}$$

and rule (15) analogously:

$$\begin{aligned}&(\mathit{Assign},\theta,l,v) \leq_L (\mathit{Unlock},\theta) \Rightarrow \\ &\quad \big((\mathit{Assign},\theta,l,v) \leq (\mathit{Store},\theta,l,v) \leq \mathit{write\_of}(\mathit{Store},\theta,l,v) \\ &\qquad \leq (\mathit{Unlock},\theta) \wedge \neg \mathit{prescient}(\mathit{Store},\theta,l) \big) \quad \vee \\ &\quad \big((\mathit{Store},\theta,l,v) \leq_L (\mathit{Assign},\theta,l,v) \leq_L (\mathit{Unlock},\theta) \ \wedge \\ &\qquad \mathit{write\_of}(\mathit{Store},\theta,l,v) \leq (\mathit{Unlock},\theta) \big)\end{aligned} \tag{15'}$$

Both rules are used in cooperation with (P6–P7). Note that in the left branch of the the disjunction in the conclusion of (3') it is unnecessary to stipulate $\neg \mathit{prescient}(\mathit{Store},\theta,l)$ since this will follow from (NP3) and (18) that will be defined below.

We can also infer which *Store* events are necessarily *not* prescient: We define the predicate $\mathit{non\_prescient}(\mathit{Store},\theta,l)$ on the given event space η to be true if one of the rules (NP1–NP3) is fulfilled.

$$\begin{aligned}&\forall a \in \{(\mathit{Lock}), (\mathit{Load},l), (\mathit{Store},l)\} \,.\, (\mathit{Store},\theta,l,v) < a \not\Rightarrow \\ &\quad (\mathit{Store},\theta,l,v) \leq (\mathit{Assign},\theta,l,v) < a\end{aligned} \tag{NP1}$$

$$\begin{aligned}&(\mathit{Store},\theta,l) \leq (\mathit{Store},\theta,l)' \wedge \mathit{non\_prescient}((\mathit{Store},\theta,l)') \not\Rightarrow \\ &\quad (\mathit{Store},\theta,l) \leq (\mathit{Assign},\theta,l) \leq (\mathit{Assign},\theta,l)' \leq (\mathit{Store},\theta,l)'\end{aligned} \tag{NP2}$$

$$\begin{aligned}&\big((\mathit{Assign},\theta,l,v) \leq_L (\mathit{Unlock},\theta) \wedge \\ &\quad (\mathit{Assign},\theta,l,v) \leq (\mathit{Store},\theta,l,v) \leq (\mathit{Unlock},\theta) \big) \not\Rightarrow \\ &\quad \big((\mathit{Assign},\theta,l,v) \leq (\mathit{Store},\theta,l,v)' \leq \mathit{write\_of}((\mathit{Store},\theta,l,v)') \\ &\qquad \leq (\mathit{Unlock},\theta) \wedge \neg \mathit{prescient}((\mathit{Store},\theta,l,v)') \big) \vee \\ &\quad \big((\mathit{Store},\theta,l,v)'' \leq_L (\mathit{Assign},\theta,l,v) \leq_L (\mathit{Unlock},\theta) \wedge \\ &\qquad \neg \mathit{non\_prescient}((\mathit{Store},\theta,l,v)'') \big)\end{aligned} \tag{NP3}$$

Rule (NP1) corresponds to the second, third, and fourth requirement in [10, §17.8] (see top of Section 4), (NP2) to (P5), and (NP3) to (P7). If $Assign \leq_L Unlock$, such that the *Assign* is the last one before the *Unlock*, then (NP3) says that if all *Store*s in between are prescient but one, then this one is necessarily *non_prescient* if the following holds: There is no matching *Store* before the *Assign* or the last such is *non_prescient*. This is a sound rule, because if the *Store* of discourse were not non-prescient, then one might choose it to be prescient, but then no last matching *Store* would occur before the *Assign* that could be chosen prescient. In such a case the *Assign* would not have been stored before the *Unlock*—not even by a prescient store—and hence the old semantics is not preserved.

Notice that the predicate *prescient* propagates from past to present with the exception of (P6–P7) which in some case needs to look back to the last non-prescient *Store*, whereas *non_prescient* is computed in the opposite direction. Also observe that $\neg non\_prescient(s)$ is *not* equivalent to $prescient(s)$ for a *Store* event s and hence also $prescient(s) \vee non\_prescient(s)$ does not always hold.

Finally, we add the new rule

$$(Store, \theta, l) \Rightarrow \neg\big(\, prescient(Store, \theta, l) \wedge non\_prescient(Store, \theta, l)\,\big) \qquad (18)$$

according to the specification of prescient *Store* events. This rule in cooperation with (NP1–NP3) prohibits that prescient *Store*s occur at places ruled out by the specification.

Summing up, a *prescient event space* is a poset of events every chain of which can be counted monotonically and satisfying conditions (1), (2), (3'), (5), (7–8), (10–14), (15'), (16), and (18).

The non-deterministic operation $\oplus$ of Section 2.4 also works for prescient event spaces (the only difference being that it defines a predicate on event spaces that are prescient).

An event space is called *complete* if for all *Read* and *Store* events corresponding *Load* and *Write* events exist (all *load_of* and *write_of* functions are total; see the discussion at the end of Section 2.2). A prescient event space η is called *complete* if additionally for any necessarily prescient $(Store, \theta, l, v)$ there is a subsequent $(Assign, \theta, l, v)$. Note that it makes sense only for the final event space of a reduction sequence to be complete. During execution, the matching *Assign* for a prescient *Store* might not have happened. A complete prescient event space fulfills the first and last requirement in [10, §17.8] (see top of Section 4). A prescient event space Γ is called *completable* if there is a sequence of events $\mathbf{a}$ such that $\Gamma \oplus \mathbf{a}$ is complete.

4.2 Labellings

According to the definitions above even for complete prescient event spaces there might be a *Store* event s in a given event space for which neither $prescient(s)$ nor $non\_prescient(s)$ is derivable. We define so-called *labellings* which allow to choose to a certain extent which *Store* shall be considered prescient and which not.

For a complete prescient event space η a labelling is a predicate ℓ on *Store*s that obeys rules (L1–L4) below together with a corresponding matching function

$$passign\_of^{\ell}_{\eta,\theta,l} : \{(Store, \theta, l) \in \eta \mid \ell(Store, \theta, l)\} \rightleftharpoons \eta(Assign, \theta, l)$$

that fulfills the axioms (M1–M5). Note that rule (M1) ensures that *passign_of* is total.

$$prescient(s) \Rightarrow \ell(s) \tag{L1}$$

$$non\_prescient(s) \Rightarrow \neg\ell(s) \tag{L2}$$

$$(\text{P5})\,[\ell/prescient] \Rightarrow \ell(Store, \theta, l, v) \tag{L3}$$

$$(\text{NP3})\,[\ell/prescient, \neg\ell/non\_prescient] \Rightarrow \neg\ell(Store, \theta, l, v) \tag{L4}$$

$$\begin{aligned}&(Store, \theta, l, v) \wedge \ell(Store, \theta, l, v) \Rightarrow \\ &\quad (Store, \theta, l, v) \leq passign\_of^{\ell}(Store, \theta, l, v) = (Assign, \theta, l, v)\end{aligned} \tag{M1}$$

$$\begin{aligned}&\forall a \in \{(Lock), (Load, l), (Store, l)\}\,.\,(Store, \theta, l) < a \wedge \ell(Store, \theta, l) \Rightarrow \\ &\quad passign\_of^{\ell}(Store, \theta, l) \leq a\end{aligned} \tag{M2}$$

$$\begin{aligned}&passign\_of^{\ell}(Store, \theta, l) \leq (Store, \theta, l)' \wedge \neg\ell((Store, \theta, l)') \Rightarrow \\ &\quad passign\_of^{\ell}(Store, \theta, l) \leq (Assign, \theta, l)' \leq (Store, \theta, l)'\end{aligned} \tag{M3}$$

$$\begin{aligned}&\big(\ (Store, \theta, l, v) \leq_L (Assign, \theta, l, v) \leq_L (Load, \theta, l) \ \wedge \ell(Store, \theta, l, v) \not\Rightarrow \\ &\quad (Assign, \theta, l, v) \leq (Store, \theta, l, v)' \leq (Load, \theta, l)\ \big) \\ &\qquad \Rightarrow passign\_of^{\ell}(Store, \theta, l, v) = (Assign, \theta, l, v)\end{aligned} \tag{M4}$$

$$\begin{aligned}&\big(\ (Store, \theta, l, v) \leq_L (Assign, \theta, l, v) \leq_L (Unlock, \theta) \ \wedge \ell(Store, \theta, l, v) \not\Rightarrow \\ &\quad (Assign, \theta, l, v) \leq (Store, \theta, l, v)' \leq write\_of((Store, \theta, l, v)') \\ &\qquad \leq (Unlock, \theta) \ \wedge \neg\ell((Store, \theta, l, v)')\ \big) \\ &\qquad\quad \Rightarrow passign\_of^{\ell}(Store, \theta, l, v) = (Assign, \theta, l, v)\end{aligned} \tag{M5}$$

In rule (L3) we use "(P5) $[\ell/prescient]$" to abbreviate the axiom (P5) where *prescient* is syntactically replaced by ℓ and the (bound) event $(Store, \theta, l, v)$ of (P5) coincides with the one in the conclusion of (L3). The analogous convention applies for (L4). Rule (L3) is necessary to propagate ℓ (as *prescient*) according to (P5), and rule (L4) to propagate $\neg\ell$ (as *non_prescient*) according to (NP3). Observe that one does not need similar rules in order to propagate (P7) and (NP2), since those are already covered by (the contraposition of) rules (L4) and (L3), respectively.

By rule (M3) one can never choose an *Assign* event as matching when its rearrangement would lead to a situation forbidden by the old event space rule (4), i.e. where two *Store* events would follow each other. Rules (M4) and (M5) fix the matching for the prescient *Store* in situations where rules (3') and (15') apply but only the right disjunct in their conclusion is fulfilled. Note that for (M4–M5)

the nested implication

$$(\Phi \not\Rightarrow \Psi) \Rightarrow \mathit{passign\_of}^{\ell}(\mathit{Store}, \theta, l, v) = (\mathit{Assign}, \theta, l, v)$$

is read with the usual conventions for $\not\Rightarrow$ but $(\mathit{Store}, \theta, l, v)$ and $(\mathit{Assign}, \theta, l, v)$ are obviously universally quantified outermost.

Lemma 4.1. *For any complete prescient event space one can give a labelling.*

Proof. We choose $\ell := \mathit{prescient}$ and show that it fulfills the labelling rules: (L1) holds by definition of ℓ, (L2) follows from (18), (L3) follows from (P5), and (L4) can be shown by contradiction employing (15'), (P7), and (18).

For any $(\mathit{Store}, \theta, l)$ in the event space with $\ell(\mathit{Store}, \theta, l)$ there is a following matching $(\mathit{Assign}, \theta, l)$ event as the event space of discourse is complete. So for $\mathit{passign\_of}^{\ell}$ we can choose the function which maps any labelled (Store, l) to the last following matching Assign before the first following event $a \in \{ (\mathit{Load}, l), (\mathit{Lock}), (\mathit{Unlock}), (\mathit{Store}, l) \}$, unless $a = (\mathit{Store}, l)$ and $\neg\ell(\mathit{Store}, l)$ when the last but one such Assign is chosen which exists by (NP2). Then $\mathit{passign\_of}^{\ell}$ is a matching function by definition.

4.3 Prescient Operational Semantics

We obtain the prescient operational semantics from the old semantics of Section 3 just by switching from the event spaces of Section 2 to the prescient event spaces of Section 4 keeping the operational rules untouched.

For the prescient operational semantics we write $\longrightarrow\!\!\triangleright$. Moreover, let $\mathit{Conf}_{\triangleright}$ denote the set of configurations with prescient event spaces, and $\mathit{Conf}_{\blacktriangleright}$ those according to the definition of $\longrightarrow$ of Section 3.

Lemma 4.2. *Any event space η (obeying the old rules) is also a prescient event space, thus any old configuration is a new configuration, i.e., $\mathit{Conf}_{\blacktriangleright} \subseteq \mathit{Conf}_{\triangleright}$, and any reduction $\Gamma \longrightarrow \Gamma'$ is also a prescient one, i.e. $\Gamma \longrightarrow\!\!\triangleright \Gamma'$ holds as well.*

Proof. Assume η is an event space satisfying the old rules. By a simple induction, $\mathit{prescient}(s)$ never holds for any Store event s in η. Thus η is a prescient event space because the new rules form a subset of the old rules. Since the configurations only differ in the event space definition and the rules of the semantics are not changed at all, the other claims of the lemma now hold trivially.

Since we use labellings our operational semantics is very liberal. It accepts reductions using Store events even if it is not clear during execution whether this Store event is meant to be prescient or not. In such a case, however, the prescient Store is not done as early as possible. Therefore, in practical cases, any Store which is not recognized by the rules (P1–P7) can be considered $\mathit{non\_prescient}$. This corresponds to choose the labelling to be simply $\mathit{prescient}$ (cf. Lemma 4.1). As a consequence, the labelling can be computed at run time. Due to (P6–P7), however, it is not always possible to detect immediately whether a Store is

prescient, sometimes one has to wait for a *Load*- or *Lock* event to happen. Also the matching can be computed at run-time with a little lookahead, cf. (M4–M5).

By the proof of Lemma 4.1, however, labellings only exist for complete prescient event spaces, hence, in the rest of the paper, any prescient event space Γ is supposed to be completable. Any completion of Γ has a labelling and though its restriction to Γ does not necessarily give a labelling, because (M1) obviously need not be valid, it is easily checked that all the other rules for labellings still hold. Thus for any completable prescient event space there exists a "partial" labelling, which fulfills only (L1–L4) and (M2–M5). Therefore we can assume that any completable prescient event space is endowed with a fixed (partial) labelling ℓ that, for the sake of simplicity, will be exhibited in form of special action names: *pStore* and *pAssign*. If $\ell(Store, \theta, l, v)$ holds then $(Store, \theta, l, v)$ is denoted $(pStore, \theta, l, v)$ and analogously for the corresponding *Assign* we use *pAssign*. This notation contains implicitly all information given by the matching function, since by monotonicity of *passign_of* for every $(pStore, \theta, l, v)$ the first subsequent $(pAssign, \theta, l, v)$ must be the matching one.

4.4 Prescient Semantics is Conservative

The relation between the "normal" and the "prescient" semantics is described in [10, §17.8, p. 408] as follows:

> "The purpose of this relaxation is to allow optimizing Java compilers to perform certain kinds of code rearrangements that preserve the semantics of properly synchronized programs but might be caught in the act of performing memory actions out of order by programs that are not properly synchronized."

This has to be formalized in the sequel. The following notation, exemplified for $\longrightarrow$ only, will be used analogously for all kinds of arrows: $\xrightarrow{r}$ denotes a one-step reduction with rule r; if $e = (r_1, \ldots, r_n)$ is a list of rules then $\xrightarrow{e}$ denotes $\xrightarrow{r_1} \ldots \xrightarrow{r_n}$; if the list is irrelevant we write $\longrightarrow^*$. For rules that change the event space we often decorate arrows with actions instead of rule names as the latter are ambiguous.

First, we observe that $\longrightarrow\!\!\triangleright$ and $\longrightarrow$ can not be bisimilar by definition since $\longrightarrow\!\!\triangleright$ permits (prescient) *Store*-actions where $\longrightarrow$ does not. But $\longrightarrow\!\!\triangleright$ cannot even be bisimilar to the reflexive closure of $\longrightarrow$, since simulating a $(pStore, \theta, l)$ and the following *Write*s by void steps leads to inequivalent configurations (since the main memories will contain different values for l).

As a prerequisite for a simulation relation of type $Conf_{\blacktriangleright} \times Conf_{\triangleright}$, we define an equivalence on prescient configurations $\sim \subseteq Conf_{\triangleright} \times Conf_{\triangleright}$ as follows:

$$(T, \eta, \mu) \sim (T', \eta', \mu') \Longleftrightarrow T = T' \wedge (T, \eta, \sigma, \mu) \downarrow (T', \eta', \sigma', \mu')$$

$$(T, \eta, \mu) \downarrow (T', \eta', \mu') \Longleftrightarrow \forall \mathbf{a} \,.\, \eta \oplus \mathbf{a}\downarrow \Leftrightarrow \eta' \oplus \mathbf{a}\downarrow \;\wedge$$

$$\forall e.\, (T, \eta, \mu) \xrightarrow{e}\!\!\triangleright^{\mathrm{c}} (T_1, \eta_1, \mu_1) \wedge (T', \eta', \mu') \xrightarrow{e}\!\!\triangleright^{\mathrm{c}} (T_2, \eta_2, \mu_2) \Rightarrow \mu_1 = \mu_2$$

where $\mathbf{a}$ is any sequence of actions, e is a sequence of rules and $(T, \eta, \mu) \longrightarrow\!\!\triangleright^{\mathrm{c}} (T', \eta', \mu')$ if $(T, \eta, \mu) \longrightarrow\!\!\triangleright^{*} (T', \eta', \mu')$ such that η' is complete. (For the sake of simplicity we do not consider the extended configurations of Section 3.10.)

This equivalence relation is obviously preserved by the rules of the semantics:

Lemma 4.3. *The relation $\sim$ is an equivalence relation such that if $\Gamma_1 \sim \Gamma_2$ then $\Gamma_1 \xrightarrow{r}\!\!\triangleright\ \Gamma_1'$ iff $\Gamma_2 \xrightarrow{r}\!\!\triangleright\ \Gamma_2'$ for any rule r, and if such a reduction r exists then $\Gamma_1' \sim \Gamma_2'$ holds.*

In order to establish a bisimulation result, we must delay all the operations which are possible due to a $(pStore, \theta, l, v)$ until the matching *pAssign* event.

But that will not work for all kinds of programs. Consider the following example:

$$(\theta,\ \{\ \texttt{synchronized(p)}\ \{\ \texttt{p.x} = \texttt{1;}\ \}\ \},\ \sigma)\ \ |\ \ (\theta',\ \{\ \texttt{p.x} = \texttt{2;}\ \},\ \sigma')$$

with $\sigma(\mathtt{p}) = \sigma'(\mathtt{p}) = o$ and $l = o.\mathtt{x}$. Its execution may give rise to a sequence of computation steps which contains the following complete subsequence of actions:

$$\begin{aligned}&(\mathit{Lock}, \theta, o), (\mathit{Assign}, \theta, l, 1), (\mathit{Store}, \theta, l, 1), (\mathit{pStore}, \theta', l, 2),\\ &\quad (\mathit{Write}, \theta', l, 2), (\mathit{Write}, \theta, l, 1), (\mathit{Unlock}, \theta, o), (\mathit{pAssign}, \theta', l, 2)\end{aligned}$$

In a simulation the $(pStore, \theta', l, 2)$ is illegal w.r.t. to the old event space definition and can only be simulated by a void (i.e. delaying) step as well as the following *Write*. Now the $(\mathit{Write}, \theta, l, 1)$ and the corresponding $(\mathit{Store}, \theta, l, 1)$ are bound to occur before the *Unlock*. Finally, after the *pAssign* we must recover the pending prescient $(\mathit{Store}, \theta', l, 2)$ and its corresponding $(\mathit{Write}, \theta', l, 2)$. According to this simulation, l has value 2 in the global memory but the reduction via $\longrightarrow\!\!\triangleright$ yields 1 for l. Thus, both end-configurations are not equivalent.

Consequently, we have to restrict ourselves to "properly synchronized" programs. A multi-threaded program T is called *properly synchronized* if any (prescient) event space in any possible configuration occurring during reduction fulfills the following axiom:

$$\begin{aligned}&(\mathit{Assign}, \theta, l), (\mathit{Assign}, \theta', l) \Rightarrow\\ &\quad (\mathit{Assign}, \theta, l) \leq (\mathit{Unlock}, \theta, o) \leq (\mathit{Lock}, \theta', o) \leq (\mathit{Assign}, \theta', l)\end{aligned} \tag{19}$$

where the *Assign*s may correspond to prescient *Store* actions. Analogously, an event space is called properly synchronized if it fulfills (19). A sufficient condition for "properly synchronizedness" is obviously the syntactic criterion that in a program *shared variables* may only be assigned in synchronized blocks.

Proper synchronization guarantees that between a prescient *Store* event and its corresponding *pAssign* event no other thread can change the main memory:

Lemma 4.4. *Let Γ be a properly synchronized complete prescient event space. If $\theta \neq \theta'$ the following holds:*

$$(\mathit{pStore}, \theta, l) \leq (\mathit{Write}, \theta', l) \Rightarrow \mathit{passign\_of}(\mathit{pStore}, \theta, l) \leq (\mathit{Write}, \theta', l)$$

Proof. Let $(pStore, \theta, l) \leq (Write, \theta', l)$ with $\theta \neq \theta'$ and let $(pAssign, \theta, l) = passign\_of(pStore, \theta, l)$.

First, assume that $store\_of(Write, \theta', l) = (Store, \theta', l) \leq (Write, \theta', l)$, for a non-prescient $(Store, \theta', l)$ such that we have $(Assign, \theta', l) \leq (Store, \theta', l)$ by the negation of (P2). There is a maximal non-prescient $(Assign, \theta', l) \leq_L (Store, \theta', l)$ such that by (P3) the fourth (value-)components of $(Assign, \theta', l)$ and $(Store, \theta', l)$ are equal. Moreover, by (M3) no $(pAssign, \theta', l)$ whatsoever can occur between those two. If now

$$(pAssign, \theta, l) \leq (Unlock, \theta, o) \leq (Lock, \theta', o) \leq (Assign, \theta', l)$$

we obviously have $(pAssign, \theta, l) \leq (Write, \theta', l)$. Otherwise, from properly synchronization, i.e. (19), it follows

$$(Assign, \theta', l) \leq (Unlock, \theta', o) \leq (Lock, \theta, o) \leq (pAssign, \theta, l) \qquad (*)$$

for a suitable $(Unlock, \theta')$–$(Lock, \theta)$ pair. We show that even

$$(Assign, \theta', l) \leq (Store, \theta', l) \leq write\_of(Store, \theta', l) \leq (Unlock, \theta', o) \qquad (**)$$

which proves the lemma since, by the negation of (NP3), we also have

$$(Lock, \theta, o) \leq (pStore, \theta, l) \leq (pAssign, \theta, l)$$

which together with $(*)$ leads to a contradiction to our assumption that $(pStore, \theta, l) \leq (Write, \theta', l)$.

In order to prove $(**)$, first note that

$$\begin{aligned}(Store, \theta', l) \leq (Unlock, \theta') \Rightarrow \\ (Store, \theta', l) \leq write\_of(Store, \theta', l) \leq (Unlock, \theta')\end{aligned}$$

holds in arbitrary prescient event spaces. To see this, it is sufficient to consider the maximal $(Store, \theta', l) \leq_L (Unlock, \theta')$ by monotonicity of $write\_of$. By (P7) and (M4) it is then impossible that there is also another $(Assign, \theta', l)$ or $(pAssign, \theta', l)$ after $(Store, \theta', l)$. There is a maximal $(Assign, \theta', l) \leq_L (Store, \theta', l)$. Between those two events no $(pAssign, \theta', l)$ can occur due to (M3), hence (15') is applicable and we are done.

For a proof of $(**)$ by contradiction, assume that

$$(Assign, \theta', l) \leq (Unlock, \theta', o) \leq (Store, \theta', l)$$

such that $(Assign, \theta', l) \leq_L (Unlock, \theta', o)$ follows. Then by (15') we have

$$(Assign, \theta', l) \leq (Store, \theta', l)' \leq write\_of((Store, \theta', l)') \leq (Unlock, \theta')$$

since if we only had

$$(pStore, \theta', l, v) \leq_L (Assign, \theta', l, v) \leq (Unlock, \theta', o) \leq (Store, \theta', l)$$

the matching rule (M5) would be violated. By (P1), however, there must exist a $(pAssign, \theta', l)$ event such that

$$(Store, \theta', l)' \leq (pAssign, \theta', l) \leq (Store, \theta', l).$$

which contradicts the assumed maximality of $(Assign, \theta', l)$.

The second case that $store\_of(Write, \theta', l) = (pStore, \theta', l) \leq (Write, \theta', l)$ is treated analogously.

In the rest of this subsection we formalize the already sketched simulation idea. To that end, in the sequel Δ (possibly with annotations) stands for configurations in $Conf_{\blacktriangleright}$ and Γ for new configurations in $Conf_{\triangleright}$. Recall that any old configuration is also a valid one in the new sense by Lemma 4.2. According to the observations above, we define a new reduction relation $\rightarrowtriangle$: $(Conf_{\blacktriangleright} \times E^*) \times (Conf_{\blacktriangleright} \times E^*)$ where $E = \{(pStore), (Write), (Read)\}$ by the rules of (red_s)–(red_d) below. Note that we do not need to treat $(Load)$ events (cf. rule (NP3)). The corresponding $\rightarrowtriangle$-configurations (Δ, e) consist of an old configuration $\Delta \in Conf_{\blacktriangleright}$ plus a list of "pending" events e. Appending an event a at the end of a list e is written $e \circ a$. An additional operation $split_{\theta,l}(e)$ is needed. Given a list of events e it yields a pair of lists (e_l, e') where both are sublists of e; the sublist e_l is obtained from e by extracting all $(pStore, \theta, l)$, $(Write, \theta, l)$ and $(Read, \theta', l)$ events and simultaneously changing a $(pStore, \theta, l)$ into $(Store, \theta, l)$; e' is e_l's complement w.r.t. e.

$$(\Delta, e) \overset{(pStore,\theta,l,v)}{\rightarrowtriangle} (\Delta, e \circ (pStore, \theta, l, v)) \qquad (\text{red}_\text{s})$$

$$(\Delta, e) \overset{(Write,\theta,l)}{\rightarrowtriangle} (\Delta, e \circ (Write, \theta, l, v)) \quad \text{if} \quad (pStore, \theta, l, v) \in e \wedge write\_of(pStore, \theta, l, v) = (Write, \theta, l, v) \qquad (\text{red}_\text{w})$$

$$(\Delta, e) \overset{(Read,\theta',l,v)}{\rightarrowtriangle} (\Delta, e \circ (Read, \theta', l, v)) \quad \text{if} \quad (Write, \theta, l) \in e \qquad (\text{red}_\text{r})$$

$$(\Delta, e) \overset{(pAssign,\theta,l,v)}{\rightarrowtriangle} (\Delta', e') \quad \text{if} \quad split_{\theta,l}(e) = (e_l, e') \wedge \Delta \xrightarrow{(Assign,\theta,l,v)} \Delta_1 \xrightarrow{e_l} \Delta' \qquad (\text{red}_\text{a})$$

$$(\Delta, e) \overset{r}{\rightarrowtriangle} (\Delta', e) \text{ for any other case } r \text{ if } \Delta \xrightarrow{r} \Delta' \qquad (\text{red}_\text{d})$$

To relate configurations of $\longrightarrow\!\!\triangleright$ and $\rightarrowtriangle$ reductions the simulation relation $\approx \subseteq Conf_{\triangleright} \times (Conf_{\blacktriangleright} \times E^*)$ is defined as follows:

$$\Gamma \approx (\Delta, e) \quad \text{if, and only if,} \quad (\Delta, e)\!\downarrow \ \wedge \ \Delta \overset{e}{\longrightarrow\!\!\triangleright} \Gamma_\Delta \wedge \Gamma_\Delta \sim \Gamma$$

where

$$(\Delta, e)\!\downarrow \quad \text{if, and only if,} \quad \exists \Delta'.\ (\Delta', \varepsilon) \rightarrowtriangle^* (\Delta, e)$$

i.e. Γ is equivalent to (Δ, e) if e is obtained correctly by means of $\rightarrowtriangle$ and Γ is equivalent to the completion of Δ, usually called Γ_Δ, by executing the pending

events in e. Note that $\longrightarrow\!\!\triangleright$ is used here for the sequence of events e, as e may contain prescient *Store* events.

Below we use the following notation of a commuting diagram

$$\begin{array}{ccc} \Gamma & \longrightarrow & \Gamma_1 \\ \downarrow & \sim & \downarrow \\ \Gamma_3 & \longrightarrow & \Gamma_2 \end{array}$$

stating that $\Gamma \longrightarrow \Gamma_1 \longrightarrow \Gamma_2$ and $\Gamma \longrightarrow \Gamma_3 \longrightarrow \Gamma_2'$ and $\Gamma_2 \sim \Gamma_2'$. This notation is also used for any other kind of arrows.

Lemma 4.5. *If $\Gamma \approx (\Delta, e)$ and $\Gamma \xrightarrow{r}\!\!\triangleright\, \Gamma'$, where r is as in case* ($\mathrm{red_d}$) *and Γ stems from a properly synchronized program, then $\Delta \xrightarrow{r} \Delta'$ and the diagram*

$$\begin{array}{ccccc} \Delta & \xrightarrow{e}\!\!\triangleright & \Gamma_\Delta & \sim & \Gamma \\ \downarrow r & & r\downarrow\ \triangledown & & \downarrow r\ \triangledown \\ \Delta' & \xrightarrow{e}\!\!\triangleright & \Gamma'_\Delta & \sim & \Gamma' \end{array}$$

commutes; thus $\Gamma \approx (\Delta, e) \overset{r}{\triangleright\!\!\longrightarrow} (\Delta', e) \approx \Gamma'$ holds.

Proof. (sketched) First note that if the left square commutes, then the whole diagram commutes by Lemma 4.3.

Next, observe that r can be executed also before e. For a proof of this check that r does not depend on e by inspecting the relevant laws for event spaces: Rules (5), (16) refer to in-between-events which are not possible in $e \in E^*$. Rules (10) and (3') are impossible since corresponding *Load*s are ruled out by (NP1) and (18). Rule (11) is not relevant as matching *Write*s are treated in ($\mathrm{red_w}$). Thus, we are left with (15'). Suppose $r = (\mathit{Unlock}, \theta)$ and that $(\mathit{pAssign}, \theta, l, v) \leq r$ is ensured via rule (15') by a preceding *Store only* (i.e. the right branch of the disjunction in (15') holds exclusively), then the last of those preceding $(\mathit{Store}, \theta, l, v)$ events is prescient, i.e. $\ell(\mathit{Store}, \theta, l, v)$ holds by (P6). Therefore, $(\mathit{pAssign}, \theta, l, v) = \mathit{passign\_of}^{\ell}(\mathit{Store}, \theta, l, v)$ by (M4) such that e can not contain the *Store* anymore as it is obtained via $\triangleright\!\!\longrightarrow^*$.

To prove that the diagram commutes it suffices, by definition of $\sim$, to show that the same actions are executed, but maybe in different order. We have to ensure that *Write* events of the same variable from different threads are not re-ordered. Consider some $(\mathit{Write}, \theta, l)$ of e. By Lemma 4.4 *Write* events of a different thread θ' can not occur in the completion of Δ, so neither in Γ_Δ and hence neither in e. But e can also never contain two $(\mathit{Write}, \theta, l)$ events, since the first would be the matching *Write* event for the starting *pStore*; the second *Write* event's matching *Store* (maybe prescient) would have to intervene between the starting *pStore* and its corresponding *pAssign* event by the monotonicity of *store_of*, thus contradicting (M2).

Theorem 4.6. *For properly synchronized programs the relation $\approx$ is a simulation relation of $\longrightarrow\!\triangleright$ and $\triangleright\!\!\rightarrow$, i.e. if $\Gamma \xrightarrow{r} \Gamma'$ during the execution of such a program and $\Gamma \approx (\Delta, e)$ then there is a (Δ', e') such that $(\Delta, e) \overset{r}{\triangleright\!\!\rightarrow} (\Delta', e')$ and $\Gamma' \approx (\Delta', e')$.*

Proof. Assume $\Gamma \approx (\Delta, e)$, i.e. $\Delta \xrightarrow{e} \Gamma_\Delta \sim \Gamma$. We do a case analysis for $\Gamma \xrightarrow{r} \Gamma'$:
Case $r = (Write)$: If $(pStore, \theta, l) \in e$ then it holds that $(\Delta, e) \overset{r}{\triangleright\!\!\rightarrow} (\Delta, e \circ r)$ by $(\mathrm{red_w})$. Moreover, by Lemma 4.3, $\Gamma' \approx (\Delta, e \circ r)$.

If $(pStore, \theta, l) \notin e$ then by Lemma 4.5, $(\Delta, e) \overset{r}{\triangleright\!\!\rightarrow} (\Delta', e')$ and $\Gamma' \approx (\Delta', e)$.
Case $r = (pAssign)$: Let $split_{\theta,l}(e) = (e_l, e')$. Since an *Assign* is always possible, assume that $\Delta \xrightarrow{(Assign,\theta,l,v)} \Delta_1$. Now every action in e_l becomes legal for the old semantics, so we can further assume $\Delta_1 \xrightarrow{e_l} \Delta'$, such that $(\Delta, e) \overset{r}{\triangleright\!\!\rightarrow} (\Delta', e')$. One can prove analogously to Lemma 4.5 that the left rectangle in

$$\begin{array}{ccccc}
\Delta & \xrightarrow{e} & \Gamma_\Delta & \sim & \Gamma \\
\downarrow{\scriptstyle (Assign,\theta,l,v)} & & & & \\
\Delta_1 & \sim & \downarrow r & & \downarrow r \\
\downarrow{\scriptstyle e_l} & & & & \\
\Delta' & \xrightarrow{e'} & \Gamma'_\Delta & \sim & \Gamma'
\end{array}$$

commutes; the right rectangle commutes by Lemma 4.3, thus $(\Delta, e) \overset{r}{\triangleright\!\!\rightarrow} (\Delta', e')$ and $\Gamma' \approx (\Delta', e')$.

For *pStore* and *Read* one proceeds as for *Write*, all other cases follow from Lemma 4.5.

The main result of Section 4 is the following corollary which states that the prescient semantics is conservative, i.e. any prescient execution sequence of a properly synchronized program can be simulated by a "normal" execution of Java.

Corollary 4.7. *Given $\Gamma \in Conf_\triangleright$ from a properly synchronized program and $\Delta \in Conf_\blacktriangleright$, if $\Gamma \sim \Delta$ and $\Gamma \longrightarrow\!\triangleright^* \Gamma'$ such that the event space $\eta_{\Gamma'}$ of Γ' is complete, then for any labelling of $\eta_{\Gamma'}$ there is a reduction sequence $\Delta \longrightarrow^* \Delta'$ such that $\Gamma' \sim \Delta'$.*

Moreover, if two different labellings yield two different reduction sequences $\Delta \longrightarrow^* \Delta'_1$ and $\Delta \longrightarrow^* \Delta'_2$, then still $\Delta'_1 \sim \Delta'_2$ holds.

Proof. First, observe that if $\Gamma \sim \Delta$ then $\Gamma \approx (\Delta, \varepsilon)$. By a simple induction on the length of the derivation by Theorem 4.6, we get $(\Delta, \varepsilon) \overset{*}{\triangleright\!\!\rightarrow} (\Delta', e)$ and $\Gamma' \approx (\Delta', e)$. Now $e = \varepsilon$ follows from the fact that Γ' is complete which entails that all prescient stores are matched by an *Assign* such that e must be empty in

the end. From $e = \varepsilon$ we immediately get $\Gamma' \sim \Delta'$. Also from $(\Delta, \varepsilon) \triangleright\!\!\rightarrow^* (\Delta', \varepsilon)$ we can strip off a derivation $\Delta \longrightarrow^* \Delta'$ by definition of $\triangleright\!\!\rightarrow$.

The second claim follows just by transitivity of $\sim$ as $\Delta'_1 \sim \Gamma' \sim \Delta'_2$.

5 Conclusions and Future Work

In this paper we presented a structural operational semantics of concurrent Java and showed its flexibility by proving a non-trivial result relating two memory implementations. Our semantics covers a substantial part of the dynamic behaviour of the language, and we expect it to combine easily with the type system developed in [8]. A further ambitious step is to include in the semantics practical features like input/output, garbage collection, distributed applications via sockets or remote method invocation, and applets.

Event spaces are not necessarily "complete," that is, no matching *Load* must necessarily occur after a *Read* action or *Write* after a *Store*. In fact, there are well-formed event spaces which are not completable, and this complicates the metatheory of the semantics. However, it is conceivable that completability may be axiomatized by means of "local" conditions such as the rules of Section 2.2.

It might also be worthwhile to study stronger notions of "proper synchronization" (for example, by taking into account *Use* actions). This might simplify the simulation of prescient semantics and allow a synchronous treatment of *Read* and *Load*.

The proofs of semantical properties (like Lemma 4.4 or Theorem 4.6) are combinatorial in nature; this is a typical situation where proof checkers or automated theorem provers can be usefully employed.

Finally, we intend to investigate operationally based notions of program equivalence, which may serve as foundations for program logics. Abadi and Leino [2] have provided an axiomatic semantics, in Hoare style, for one of the (sequential) object calculi of [1] and proved that the logic is sound with respect to the operational semantics of the object calculus in use. The development of such a logic for a real concurrent object-oriented language like Java remains a challenge.

Acknowledgements. We thank Doug Lea for useful comments and some inspiration regarding future work.

References

1. Martín Abadi and Luca Cardelli. *A Theory of Objects*. Springer, New York, 1996.
2. Martín Abadi and K. Rustan M. Leino. A Logic of Object-Oriented Programs. In Michel Bidoit and Max Dauchet, editors, *Proc. 7th Int. Conf. Theory and Practice of Software*, volume 1214 of *Lect. Notes Comp. Sci.*, pages 682–696, Berlin, 1997. Springer.
3. Jim Alves-Foss, editor. *Formal Syntax and Semantics of Java*. Lect. Notes Comp. Sci. Springer, Berlin, 1998. This volume.

4. Ken Arnold and James Gosling. *The Java Programming Language*. Addison–Wesley, Reading, Mass., 1996.
5. Egon Börger and Wolfram Schulte. A Programmer Friendly Modular Definition for the Semantics of Java. In Alves-Foss [3]. This volume.
6. Pietro Cenciarelli, Alexander Knapp, Bernhard Reus, and Martin Wirsing. From Sequential to Multi-threaded Java: An Event-Based Operational Semantics. In Michael Johnson, editor, *Proc. 6th Int. Conf. Algebraic Methodology and Software Technology*, volume 1349 of *Lect. Notes Comp. Sci.*, pages 75–90, Berlin, 1997. Springer.
7. Richard Cohen. The Defensive Java Virtual Machine Specification. Technical report, Computational Logic Inc., 1997. Draft at `http://www.cli.com`.
8. Sophia Drossopoulou and Susan Eisenbach. Java is Type Safe — Probably. In Alves-Foss [3]. This volume.
9. Matthew Flatt, Shriram Krishnamurthi, and Matthias Felleisen. A Programmer's Reduction Semantics for Classes and Mixins. In Alves-Foss [3]. This volume.
10. James Gosling, Bill Joy, and Guy Steele. *The Java Language Specification*. Addison–Wesley, Reading, Mass., 1996.
11. Yuri Gurevich. Evolving Algebras 1993: Lipari Guide. In Egon Börger, editor, *Specification and Validation Methods*, pages 9–36. Oxford University Press, 1995.
12. Doug Lea. *Concurrent Programming in Java*. Addison–Wesley, Reading, Mass., 1997.
13. Robin Milner, Mads Tofte, Robert Harper, and David MacQueen. *The Definition of Standard ML (Revised)*. MIT Press, Cambridge, Mass., 1997.
14. Tobias Nipkow and David von Oheimb. Machine-checking the Java Specification: Proving Type-Saftey. In Alves-Foss [3]. This volume.
15. Gordon D. Plotkin. A Structural Approach to Operational Semantics (Lecture notes). Technical Report DAIMI FN–19, Aarhus University, 1981 (repr. 1991).
16. Zhenyu Qian. A Formal Specification of Java Virtual Machine Instructions for Objects, Methods and Subroutines. In Alves-Foss [3]. This volume.
17. Bernhard Reus, Alexander Knapp, Pietro Cenciarelli, and Martin Wirsing. Verifying a Compiler Optimization for Multi-threaded Java. In Francesco Parisi Presicce, editor, *Sel. Papers 12th Int. Wsh. Recent Trends in Algebraic Development Techniques*, volume 1376 of *Lect. Notes Comp. Sci.*, pages 402–417, Berlin, 1998. Springer.
18. Raymie Stata and Martín Abadi. A Type System for Java Bytecode Subroutines. In *Proc. 25th ACM Symp. Principles of Programming Languages*, 1998. To appear.
19. David Syme. Proving Java Type Soundness. In Alves-Foss [3]. This volume.
20. Charles Wallace. The Semantics of the Java Programming Language: Preliminary Version. Technical Report CSE-TR-355-97, University of Michigan, 1997.
21. Glynn Winskel. An Introduction to Event Structures. In Jacobus W. de Bakker, Willem P. de Roever, and Grzegorg Rozenberg, editors, *Linear Time, Branching Time and Partial Order in Logics and Models for Concurrency*, volume 354 of *Lect. Notes Comp. Sci.*, pages 364–397. Springer, Berlin, 1988.

A Syntax

$$
\begin{aligned}
\mathit{Statement} ::= {} & \texttt{;} \mid \mathit{Block} \mid \mathit{StatementExpression}\ \texttt{;} \\
& \mid \texttt{synchronized(}\ \mathit{Expression}\ \texttt{)}\ \mathit{Block} \\
& \mid \texttt{throw}\ \mathit{Expression}\ \texttt{;} \mid \mathit{TryStatement} \\
& \mid \texttt{return}\ \mathit{Expression}_{\mathit{opt}}\ \texttt{;} \\
& \mid \mathit{IfThenStatement} \\
\mathit{Block} ::= {} & \texttt{\{}\ \mathit{BlockStatements}_{\mathit{opt}}\ \texttt{\}} \\
\mathit{BlockStatements} ::= {} & \mathit{BlockStatement} \mid \mathit{BlockStatements}\ \mathit{BlockStatement} \\
\mathit{BlockStatement} ::= {} & \mathit{LocalVariableDeclaration}\ \texttt{;} \mid \mathit{Statement} \\
\mathit{LocalVariableDeclaration} ::= {} & \mathit{Type}\ \mathit{VariableDeclarators} \\
\mathit{ReturnType} ::= {} & \mathit{Type} \mid \texttt{void} \\
\mathit{Type} ::= {} & \mathit{PrimitiveType} \mid \mathit{ClassType} \\
\mathit{PrimitiveType} ::= {} & \texttt{boolean} \mid \texttt{int} \mid \ldots \\
\mathit{ClassType} ::= {} & \mathit{Identifier} \\
\mathit{VariableDeclarators} ::= {} & \mathit{VariableDeclarator} \\
& \mid \mathit{VariableDeclarators}\ \texttt{,}\ \mathit{VariableDeclarator} \\
\mathit{VariableDeclarator} ::= {} & \mathit{Identifier}\ \texttt{=}\ \mathit{Expression} \\
\mathit{Expression} ::= {} & \mathit{AssignmentExpression} \\
\mathit{AssignmentExpression} ::= {} & \mathit{Assignment} \mid \mathit{BinaryExpression} \\
\mathit{Assignment} ::= {} & \mathit{LeftHandSide}\ \texttt{=}\ \mathit{AssignmentExpression} \\
\mathit{LeftHandSide} ::= {} & \mathit{Name} \mid \mathit{FieldAccess} \\
\mathit{Name} ::= {} & \mathit{Identifier} \mid \mathit{Name}\ \texttt{.}\ \mathit{Identifier} \\
\mathit{FieldAccess} ::= {} & \mathit{Primary}\ \texttt{.}\ \mathit{Identifier} \\
\mathit{Primary} ::= {} & \mathit{Literal} \mid \texttt{this} \mid \mathit{FieldAccess} \mid \texttt{(}\ \mathit{Expression}\ \texttt{)} \\
& \mid \mathit{ClassInstanceCreationExpression} \\
& \mid \mathit{MethodInvocation} \\
\mathit{ClassInstanceCreationExpression} ::= {} & \texttt{new}\ \mathit{ClassType}\ \texttt{()} \\
\mathit{MethodInvocation} ::= {} & \mathit{Primary}\ \texttt{.}\ \mathit{Identifier}\texttt{(}\ \mathit{ArgumentList}_{\mathit{opt}}\ \texttt{)} \\
\mathit{ArgumentList} ::= {} & \mathit{Expression} \mid \mathit{ArgumentList}\ \texttt{,}\ \mathit{Expression} \\
\mathit{BinaryExpression} ::= {} & \mathit{UnaryExpression} \\
& \mid \mathit{BinaryExpression}\ \mathit{BinaryOperator} \\
& \quad \mathit{UnaryExpression} \\
\mathit{UnaryExpression} ::= {} & \mathit{UnaryOperator}\ \mathit{UnaryExpression} \\
& \mid \mathit{Primary} \mid \mathit{Name} \\
\mathit{StatementExpression} ::= {} & \mathit{Assignment} \mid \mathit{ClassInstanceCreationExpression} \\
& \mid \mathit{MethodInvocation} \\
\mathit{TryStatement} ::= {} & \texttt{try}\ \mathit{Block}\ \mathit{Catches} \\
& \mid \texttt{try}\ \mathit{Block}\ \mathit{Catches}_{\mathit{opt}}\ \texttt{finally}\ \mathit{Block} \\
\mathit{Catches} ::= {} & \mathit{CatchClause} \mid \mathit{CatchClauses}\ \mathit{CatchClause} \\
\mathit{CatchClause} ::= {} & \texttt{catch(}\ \mathit{Type}\ \mathit{Identifier}\ \texttt{)}\ \mathit{Block} \\
\mathit{IfThenStatement} ::= {} & \texttt{if(}\ \mathit{Expression}\ \texttt{)}\ \mathit{Statement}
\end{aligned}
$$

Dynamic Denotational Semantics of Java

Jim Alves-Foss and Fong Shing Lam

Center for Secure and Dependable Software, Department of Computer Science,
University of Idaho, Moscow ID 83844-1010, USA

Abstract. This chapter presents a dynamic denotational semantics of the Java programming language. This semantics covers almost the full range of the base language, excluding only concurrency and the API's. A discussion of these limitations is provided in the final section of the chapter.

The abstract syntax described in Chapter 1 tells us how to construct a grammatically correct program. Every syntactically correct program describes an *environment* that provides all the information about what to do during program execution. The semantics presented in this chapter, formalizes the definition of Java program behavior as defined in the *Java Language Specification* (JLS) [1]. We describe the Java environment in Section 1. Each executing program is associated with a *store* that is a repository for all instance values during program execution. The Java store is described in Section 2. Executing a Java program begins with executing the command in the static method "main" in the given class definition. Therefore, the result of a program depends on the semantics of commands and the expressions in the commands. We shall introduce a denotational semantics of these commands and expressions in Sections 3 and 4. Throughout these semantics, we concurrently define two sets of semantics, a *dynamic* and a *static* semantics, to respectively represent the execution and definitional denotations of the programs.

1 Environment

An *environment* is the information center for the execution engine and is at the heart of these semantics. Our environment is a semantic domain that has two components, the dynamic and static semantics. The *dynamic* aspect of the environment contains the traditional environmental information related to variables, their types and locations in the store (as in Stoy's classical book on denotational semantics [4]. It also contains control flow information for exceptions and breaks. The *static* aspect of the environment contains information related to all of the classes used by the program. This information includes the class members, types, initialization functions, super class and implemented interfaces. The static part of the environment is determined by evaluating the input files and then is used as an input parameter to the denotation of the main method of the invoked class.

Jim Alves-Foss (Ed.): Formal Syntax and Semantics of Java, LNCS 1523, pp. 201–240, 1999.

In addition to information related to classes, their members and local variables; the environment contains a number of auxiliary variables (all starting with the symbol &. These variables are used to record nesting and scoping information as well as flow control information.

- &*package* - specifies the fully qualified name of the package currently being defined.
- &*currentInt* - specifies the fully qualified name of the interface currently being defined.
- &*currentClass* - specifies the fully qualified name of the class currently being defined.
- &$Mods$ - provides a list of modifiers used in the current declaration.
- &$Type$ - defines the type used in the current declaration.
- &*varType* - defines if this is a "Field" or "Local" variable declaration.
- &*switchExpr* - value of the expression of the current Switch statement.
- &*caseFound* - boolean variable indicating if the case matching the switch expression has been found.
- &$defaultFound$ - boolean variable indicating if the default switch case had been found.
- &*caseCont* - command continuation for execution of the appropriate switch statement. This is needed due to the fact that the default case may be defined anywhere in the switch statement.
- &*break* - continuation information.
- &*return* - specifies the command continuation to execute upon return.
- &$returnVal$, &*returnType* - specify the return type and value for a call.
- &*super* - specifies the name of the current executing classes super class
- &*throw* - specifies the command continuation to be executed upon a throw command.
- &*thrown* - specifies the value, type pair referring the thrown object (exception).
- &*thisObject* - specifies a reference to the current object in which execution is occurring.
- &*thisClass* - specifies a reference to the class of the current object in which execution is occurring.

To simplify the semantic presentation, we include within the environment a collection of methods (or auxiliary functions). For these functions, we use method invocation notation for these functions, where $\gamma.m(p_1 \ldots p_n)$ denotes invocation of auxiliary function m, with parameters $(p_1 \ldots p_n)$, invoked in the context of the current environment, γ. (Note that variable names will be referenced in the usual way with $\gamma[name]$ referring to the current value of *name* in the environment, and $\gamma[name \leftarrow v]$, denoting the new environment with name know returning the value v. The functions related to the dynamic semantics (execution) of a program are:

- *assnCompatible*(τ, τ_1). This boolean function returns true if a value of type τ can be assigned to a variable of type τ_1, according to the rules of the JLS [1].

- *classLoader(name, store).* In Java, whenever a new instance of a class is created, we invoke the class loader. In the runtime system this involves first determining if the class is already loaded, otherwise finding it from some source location, loading the bytecode, and then instantiating the class constant variables and executing any static class initializer. This function represents this complex operation, and may result in modification to both the environment and the store.
- *condTypeOf*(τ_t, v_t, τ_f, v_f). This function returns the type of conditional expressions as defined by the JLS [1]. A full definition of this function appears following the specification of condition expressions in the semantics that follow.
- *getArrayElem(a, ind).* This function returns the location of array element *ind* from the array referenced by *a*.
- *getArrayElemType(a).* This function returns the type of array elements in the array referenced by array *a*.
- *getArrayRef(name).* This function returns the reference for the array *name*.
- *getComCont(term).* This function retrieves the command condition from the environment auxiliary variable denoted by *term*, where this term can be *&break* or *&continue*. Continuations are discussed in detail in section 3.3.
- *getMethod(name, signature).* This function returns the denotation of the named method (of the specified signature). Specifically the value returned is a semantic function that takes a set of arguments as parameters and returns a command function (a function that takes an environment, command continuation and a store and returns an answer).. All appropriate searching of the nested class and interface definitions is conducted, in accordance with the JLS [1].
- *getValue(term).* This function returns the value of the auxiliary variable referred to by *term*.

The auxiliary functions used to build the static (declaration) portion of the environment are:

- *addConstr(mods, defn, throws, body).* This function is used to add the constructor specification to the environment for the current class.
- *addField(name, initExp).* This function is used to add the specified field and initialization expression to the environment given the current type and class scope.
- *addLocal(name, initExp).* This function is used to add the specified local variable and initialization expression to the environment, given the current type and class scope.
- *addMethod(mod, hdrInfo, throws, body).* This function is used to add the method specification to the environment for the current class.
- *addMethodHdr(hdrInfo).* This function is used to add the abstract method header specification to the environment for the current interface.
- *addStaticField(name, valExpr).* This function is used to add the specified static field and initialization expression to the environment given the current type and class scope.

- *addStaticInit(com).* This function is used to add the command code (or denotation) for the specified static initializer to the class specification in the environment.
- *enterClass(mods, id, super, interfaces).* This function is used to denote that we are currently parsing a class specification. It modifies the current scope of the environment and sets the *&currentClass* field of the environment to indicate the current class.
- *enterInterface(mods, id, extends).* This function is used to denote that we are currently parsing an interface specification. It modifies the current scope of the environment and sets the *&currentInt* field of the environment to indicate the current interface.
- *import(name)* and *importOnDemand(name).* These function are used to denote the java `import` command. Specifically they are used to add all the class definitions from the specified files to the environment.
- *instanceOf*(τ_1, τ_2) - returns true if τ_1 is an instance of τ_2 in the current environment.
- *isStatic().* This boolean function returns true if the modifier of the current field declaration include the **static** modifier.

2 Store

The *store* is memory that is dynamically created, expanded, and destroyed by the execution engine. We can view the store as a communication channel between statements. Together with the environment of Section 1, it forms the *state* of the execution environment. Every local variable declaration, loading of a class object or **new** operator applied to a class type or array type creates one or more entries in the store. If the entry is a class object, the content of the entry is filled according to the constructor code of the class and field initializations. An array object is initialized with a field name of "length" denoting the number of elements in the array.

For the semantic presented in this chapter the only auxiliary function for the store is:

- mkException(className). This function creates a new exception object in the store, as defined by the exception class referred to by *className*.

3 Denotational Semantics

This section presents the (almost) full denotational semantics of the Java language – only missing aspects of concurrency.

3.1 Semantics Domains and Data Values

One is often tempted, when developing a formal model of a language, to abstract out the limitations of the concrete representation of the language. For example,

authors of many language models will abstract values of type int to mathematical integers. Unfortunately, this provides an unrealistic definition of the behavior and meaning of the language constructs. For example, in the Java language there are no run-time indications of overflow or underflow of integers operations, but rather an implicit truncation of the resulting two's complement representation of the number to the requisite number of bits. Without an understanding of this functionality of the language and an explicit representation of it in a formal description of the language, correctness proofs of the code may be incorrect. To avoid this difficulty we represent all concrete limitations of the Java language in the following semantics of expressions. This is possible since Java precisely defines these limitations for all primitive types.

3.2 Semantic Domains

The semantic domains representing the values of the numeric data types are defined below. To simplify the semantics, we have added two special values to each of these domains, $\perp$ ("bottom") which represented a value with no information content and $\top$ ("top") which represents a value with full (potentially conflicting information). The purpose of these values is to enable each domain to be a *complete partial order*, which simplifies the mathematics underlying the semantics. These values are used by the semantic functions and do not have an equivalent representation within the Java language. The basic domains are flat domains in that there is no implicit ordering between values of the domains other than between the values and $\perp$ and $\top$.

Let $\mathcal{I}$ represent the set of integers, and $\mathcal{R}$ represent the set of real numbers. In the following **IEEE**(s.m.e) denotes an IEEE 754 floating point number with sign, mantissa and exponent, **NAN** represents not-a-number and $+\infty$ and $-\infty$ represent positive and negative infinity, respectively [2].

$$
\begin{aligned}
\mathit{Byte} &= \{n \in \mathcal{I} \mid -128 \leq n \leq 127\} \cup \\
&\quad \{\perp, \top\} \\
\mathit{Short} &= \{n \in \mathcal{I} \mid -32768 \leq n \leq 32767\} \cup \\
&\quad \{\perp, \top\} \\
\mathit{Int} &= \{n \in \mathcal{I} \mid -2147483648 \leq n \leq 2147483647\} \cup \\
&\quad \{\perp, \top\} \\
\mathit{Long} &= \{n \in \mathcal{I} \mid -9223372036854775808 \leq n \leq 9223372036854775807\} \cup \\
&\quad \{\perp, \top\} \\
\mathit{Char} &= \{n \in \mathcal{I} \mid 0 \leq n \leq 65535\} \cup \\
&\quad \{\perp, \top\} \\
\mathit{Float} &= \{f \in \mathcal{R} \mid f = \mathbf{IEEE}(s.m.e), 0 \leq m \leq 2^{24} - 1 \wedge -149 \leq e \leq 104\} \cup \\
&\quad \{\perp, \top, \mathbf{NAN}, +\infty, -\infty\} \\
\mathit{Double} &= \{f \in \mathcal{R} \mid f = \mathbf{IEEE}(s.m.e), 0 \leq m \leq 2^{24} - 1 \wedge -149 \leq e \leq 104\} \cup \\
&\quad \{\perp, \top, \mathbf{NAN}, +\infty, -\infty\}
\end{aligned}
$$

We define several other semantic domains for use within the denotational semantics of Java presented in this Chapter. Note that we deliberately avoid specifying the detailed semantic domains of literals, but leave them abstract and presume that a parser will interpret them correctly. Note that in this presentation here, we do not present the specifics of the domains, but rather try and define them in a context specific manner. For example we typically have $\sigma \in \Sigma$ be a store, and $\gamma \in \Gamma$ be an environment. The values (such as $r \in \mathcal{V}$) denote the basic values of the java language (shorts, ints, floats, etc.) and their types, $\tau \in \mathcal{T}$. We also refer to locations $l \in \mathcal{L}$ as indecies into the store. These are all flat domains, with a $\perp$ and $\top$ value as discussed above.

3.3 Continuations

Many of the semantic functions defined in the following sections utilize the concept of continuations. While evaluating a syntactic construct, we typically focus on one piece of the code. A continuation defines the semantics of the rest of that code (whether it be the rest of an expression, the rest of a command, all of a method, or the rest of a declaration). The results of continuations are either values of the specified semantic domains, such as environment or an *answer*. Since the core Java language does not interact with the outside world, we have left the concept of modifications to this world as an abstract answer domain. None of the semantics here modify that domain, such modification only occurs in the runtime libraries (Java API). We utilize the following continuations in these semantics:

- $\rho(\gamma)$ - package continuation. This continuation takes the environment parameter, γ, and returns an environment based on the declarations of the rest of the code. Note that this continuation is used in the highest level of package/code declarations.
- $\delta(\gamma)$ - declaration continuation. This continuation takes the environment parameter, γ, and returns an environment based on the declarations of the rest of the code. This continuation is used within specific declaration constructs.
- $\theta(\gamma, \sigma)$ - command continuation. This continuation takes the environment parameter, γ and store parameter, σ, and returns a an answer based on the denotation of the rest of the command. The denotation is dependent on the parameters specified, which are typically a modified store and a potentially modified environment from a command execution.
- $\kappa(r, \tau, \sigma)$ - expression continuation. This continuation takes a value, r, of type τ and a store parameter, σ, and returns a store based on the denotation of the rest of the program. The denotation is dependent on the typed data value specified, which is typically the result of an expression evaluation.
- $\alpha(v, \tau, l, \sigma)$ - location continuation. This continuation takes a value, v, of type, τ, location, l, and a store parameter, σ, and returns an answer based on the denotation of the rest of the program. The denotation is dependent on the typed data value specified and location, which are typically the result of an expression evaluation and the location is the location of the variable referenced in the expression.

3.4 Semantic Functions

Within the denotational semantics, we make use of several semantic functions. These functions define the relationship between the code, as seen by the parser, and the actual operations of the resulting program. Since we are working with a full language specification (excluding multi-threading), we need to use a wider range of semantic functions than those found in simpler examples in the literature. The semantic functions used are divided into two categories, *operational* and *definitional*.

Operational Semantic Functions. In the context of the Java programming language an operational semantic function is one that defines the relationship between the current language construct and the execution time behavior of the program. Specifically, operational semantic functions directly manipulate the store, resulting in a new store. Specifically, in the following semantics, the operational semantic functions are:

- Command functions $\mathcal{C}[\![]\!]$. These functions define the meaning of Java commands. The meaning of any Java command is defined in terms of three parameters, the current environment γ, a command continuation θ, and the current store σ. The command continuation is a function that defines the behavior of the rest of the program in the context of an environment and a store. Therefore, the result of the $\mathcal{C}[\![\mathrm{c}]\!]$ function is typically $\theta(\gamma_1, \sigma_1)$, where γ_1 and σ_1 are the new environment and store obtained from executing the command c, and $\theta(\gamma_1, \sigma_1)$ represents the behavior of the program given these values. This is not true when the command c results in an exception, or abnormal flow of control change (such as a `break` or `return` command.) In these cases, the result of the $\mathcal{C}[\![\mathrm{c}]\!]$ function is based on a related continuation stored within the environment (e.g., see the semantics of the `break` and `return` commands.)
- Expression functions $\mathcal{E}[\![]\!]$ and location functions $\mathcal{L}[\![]\!]$. These functions define the meaning of Java expressions. We separate the location functions to denote those expressions that result in a value (called value expressions in the JLS [1]) and those that result in reference to some memory location (called variable expressions in the JLS[1]). Note that modification of the store must result in the assignment of a value to a location (either directly through an assignment statement or through a pre or post expressions – e.g., i++.)
  - The $\mathcal{E}[\![]\!]$ functions are defined in terms of three parameters, the current environment γ, an expression continuation κ, and the current store σ. The expression continuation is a function that defines the behavior of the rest of the program in the context of a value, type and a store. Therefore, the result of the $\mathcal{E}[\![e]\!]$ function is typically $\kappa(v, \tau, \sigma_1)$, where v is the resulting value of type τ obtained from executing the expression e, and σ_1 is the resulting store. As with commands, exceptions do occur that may result in using a saved expression continuation instead of the current continuation.

- The $\mathcal{L}[\![]\!]$ functions are also defined in terms of three parameters, the current environment γ, a location continuation α, and the current store σ. The location continuation is a function that defines the behavior of the rest of the program in the context of a value, type, location and a store. Therefore, the result of the $\mathcal{L}[\![e]\!]$ function is typically $\alpha(v, \tau, l, \sigma_1)$, where v is the resulting value of type τ obtained from executing the expression e, which refers to a variable at location l, and σ_1 is the resulting store.

Definitional Semantic Functions. A definitional semantic function is ancillary to the actual execution behavior of the program, but rather defines the context in which the execution takes place. The definitional functions used in the following semantics are:

- Goal $\mathcal{G}[\![]\!]$ and Package functions $\mathcal{P}[\![]\!]$. These functions define the high-level meaning of a Java source file, defined in terms of import files and class definitions. The goal function takes no parameters and returns an environment. The environment is subsequently used during execution and provides the full class definitions for the command and expression functions. The package function takes two parameters, the current environment γ and a package continuation function ρ. We use a continuation function here to be consistent with the style of semantics presents in the command and expression functions. The continuation function takes the newly modified environment and returns an environment based on the rest of the file.
- Declaration functions $\mathcal{D}[\![]\!]$. These functions define the declarations of methods, classes, interfaces and other lower-level constructs within the package.
- Modifier functions $\mathcal{M}[\![]\!]$. This function defines the list of modifiers for fields and methods.
- Type functions $\mathcal{T}[\![]\!]$. These functions are used solely to determine specified data types. These types are calculated based on the current environment, provided as a parameter. The result of the type function is a string representation of the data type. Specifically we use the same string notation that Java bytecode uses to specify types [3]. Note that there are some cases where multiple types must be returned (for example the list of interfaces implemented by a class), in this case we just append the string representation of the types as the Java bytecode does for parameter lists.
- Value functions $\mathcal{V}[\![]\!]$. These functions are a catch-all function that returns a value associated with the static input. A value function only takes the current environment as a parameter and returns a pair that consists of the value (as a basic type or string) and the type. Throughout these semantics we may need only the first or second element of this pair. We will select these using the **fst** and **snd** operations on the result or by direct assignment $(r, \tau) = \mathcal{V}[\![E]\!]\gamma$, where r is assigned the first value of the pair and τ is assigned the second.

3.5 Auxiliary Functions

- **mkArrayType**(τ, n) - this function takes the type specified by the first parameter and returns an array type of n dimensions.
- **mkMethodValue**(d, τ) - this function takes the definition specification of a method, d which is a pair consisting of a method name and the formal parameters, and a return type, τ, to create a value to store in the environment for searching and retrieving the methods.
- **binaryPromoteType**(τ_1, τ_2) - - returns the type resulting from a binary number promotion of the types τ_1 and τ_2 according to the rules of the JLS.
- **unaryPromoteType**(τ) - returns the type resulting from a unary numeric promotion of type τ according to the rules of the JLS.
- **promote**$(\tau, (r, \tau_1))$ - this function converts a value r, of type τ_1 to a compatible value of type τ following the numeric promotion rules of the JLS.
- **cast**$(\tau, (r, \tau_1))$ - this function converts a value r, of type τ_1 to a compatible value of type τ following the type casting rules of the JLS.
- **leftShift**$((r_1, \tau_1), (r_2, \tau_2))$ - this function returns the value of r_1, of type τ_1 left shifted r_2 places (where r_2 is of type τ_2). The resulting value is of type τ_1.
- **String**(r, τ) - this function actually invokes the toString function of the java.lang class corresponding to the type τ on te value r to return a string.
- **fst**(p) - this is a function that takes returns the first value of a pair.
- **snd**(p) - this is a function that takes returns the second value of a pair.
- **append**$(l1, l1)$ - this is a simple list append function.
- **isNumeric**(τ) - this function returns true is the type of the parameter can be classified as a numeric value.

4 Java Semantics

The following sections detail the semantics of the Java language. To simplify the presentation, we have taken the full syntax of the Java language (as presented in Chapter 2 of this volume) and reduced it, by removing high-level redundancy. For example, syntactically we define several levels of statements, including those with and without trailing if's. For the presentation here, we only worry about the actual statements, such as the for-statement, while-statement, etc.

4.1 The Meaning of a Java Program

When a user wants to execute a Java program *myclass* they type "`javac` *myclass args*". In the context of the semantics presented here, this is defined as:

$(\gamma.getMethod(\text{“}myclass.main''\text{, “}[Ljava.lang.String;'')\mathcal{V}[\![\text{args}]\!]\gamma)\gamma c_{\text{exit}}\sigma$ where
$\gamma = \mathcal{G}[\![\text{Goal}]\!]$ // where Goal is the myclass.java file and
$\sigma = \gamma.classLoader(myclass, \textbf{newStore}())$ and
c_{exit} = the command continuation function that terminates the program

This semantics evaluates the source file (and all other imported class files) to create a new environment γ. It then invokes a `classLoader` function to load the specified class into the store, σ and executes the method main of the class with respect to the specified command-line arguments, the environment and the new store.

4.2 Names and Literals

The designers of the Java language separated the concept of names from other primary entities in the grammar. The reason for this is to avoid some possible ambiguities in a LALR(1) parser. For the semantic functions, all we need to return is a representation of the name to be used once the full name is defined. Since the value semantic function requires a returned (value,type) pair, we specify the type of names as "name".

$\mathcal{V}[\![$<Name>$]\!]\gamma$::=
  $\mathcal{V}[\![$<SimpleName>$]\!]\gamma$
 | $\mathcal{V}[\![$<QualifiedName>$]\!]\gamma$

$\mathcal{V}[\![$<SimpleName>$]\!]\gamma$ ::=
  $\mathcal{V}[\![$Id$]\!]\gamma$ = (ValueOf(Id), "name")

$\mathcal{V}[\![$<QualifiedName>$]\!]\gamma$::=
  $\mathcal{V}[\![$Name.Id$]\!]\gamma$ =
    (Str + '.' + ValueOf(Id),"name") where
      Str = fst($\mathcal{V}[\![$Name$]\!]\gamma$)

$\mathcal{V}[\![$<Literal>$]\!]\gamma$ ::=
  $\mathcal{V}[\![$IntLit$]\!]\gamma$ = (ValueOf(IntLit), "`I`")
 | $\mathcal{V}[\![$FloatLit$]\!]\gamma$ =(ValueOf(FloatLit), "`F`")
 | $\mathcal{V}[\![$BoolLit$]\!]\gamma$ = (ValueOf(BoolLit), "`Z`")
 | $\mathcal{V}[\![$CharLit$]\!]\gamma$ = (ValueOf(CharLit),"`C`")
 | $\mathcal{V}[\![$StringLit$]\!]\gamma$ = (ValueOf(StringLit), "Ljava.lang.String;")
 | $\mathcal{V}[\![$NullLit$]\!]\gamma$ = (**null**, "L;")

4.3 Packages

Goal and Compilation Unit. The following productions define the semantics of a single Java compilation unit. This is encapsulated within a goal, which has no parameters. The goal semantics specify the creation of a new environment and an identity continuation such that the result of the goal will be an environment to be used during execution. In the <CompUnit> specification we forced the automatic loading of the java.lang package as if there were the statement "`import java.lang.*`" immediately following any package declaration statement.

$\mathcal{G}[\![$<Goal>$]\!]$ ::=
  $\mathcal{G}[\![$<CompUnit>$]\!]$ = $\mathcal{P}[\![$<CompUnit>$]\!]\gamma\rho$ where
    γ = newEnvironment() and

$\forall\gamma_1.\rho(\gamma_1) = \gamma_1$

$\mathcal{P}[\![$<CompUnit>$]\!]\gamma\rho$ ::=
  $\mathcal{P}[\![$<PackageDecl>$^?$ <ImportDeclList>$^?$ <TypeDeclList>$^?]\!]\gamma\rho =$
    $\mathcal{P}[\![$<PackageDecl>$^?]\!]\gamma\rho_1$ where
      $\forall\gamma_1.\rho_1(\gamma_1) = \mathcal{P}[\![$<ImportDeclList>$^?]\!]\gamma_2\rho_2$ where
        $\gamma_2 = \rho(\gamma.importOnDemand($java.lang$))$ and
        $\forall\gamma_2.\rho_2(\gamma_2) = \mathcal{P}[\![$<TypeDeclList>$^?]\!]\gamma_2\rho 1$‘

$\mathcal{P}[\![$<PackageDecl>$]\!]\gamma\delta$ ::=
  $\mathcal{P}[\![$**package** <Name> **;**$]\!]\gamma\rho = \rho(\gamma[\&package \leftarrow(\text{fst}(\mathcal{V}[\![$<Name>$]\!]\gamma))])$

Import commands. The import commands cause some difficulty in the semantics. Specifically, an import command loads into the environment all the relevant information related to an entity specified in a separate compilation unit. For the sake of brevity we include auxiliary functions that modify the environment to reflect the action of the import command.

$\mathcal{P}[\![$<ImportDeclList>$]\!]\gamma\delta$ ::=
  $\mathcal{P}[\![$<ImportDecl>$]\!]\gamma\rho$
| $\mathcal{P}[\![$<ImportDeclList$_1$> <ImportDecl>$]\!]\gamma\rho =$
    $\mathcal{P}[\![$<ImportDeclList$_1$>$]\!]\gamma\rho_1$ where
      $\forall\gamma_1.\rho_1(\gamma_1) = \mathcal{P}[\![$<ImportDecl>$]\!]\gamma_1\rho$

$\mathcal{P}[\![$<ImportDecl>$]\!]\gamma\rho$ ::=
  $\mathcal{P}[\![$<SingleTypeImportDecl>$]\!]\gamma\rho$
| $\mathcal{P}[\![$<TypeImportOnDemandDecl>$]\!]\gamma\rho$

$\mathcal{P}[\![$<SingleTypeImportDecl>$]\!]\gamma\delta$ ::=
  $\mathcal{P}[\![$**import** <Name> **;**$]\!]\gamma\rho = \rho(\gamma.import(\mathbf{fst}(\mathcal{V}[\![$<Name>$]\!]\gamma))$

$\mathcal{P}[\![$<TypeImportOnDemandDecl>$]\!]\gamma\delta$ ::=
  $\mathcal{P}[\![$**import** <Name> **. * ;**$]\!]\gamma\rho = \rho(\gamma.importOnDemand(\mathbf{fst}(\mathcal{V}[\![$<Name>$]\!]\gamma))$

Class and Interface Declarations. The class and interface declaration productions are defined in terms of the declaration semantic function. The following semantics define the relationship between the package and declaration semantic functions.

$\mathcal{P}[\![$<TypeDeclList>$]\!]\gamma\delta$ ::=
  $\mathcal{P}[\![$<TypeDecl>$]\!]\gamma\rho$
| $\mathcal{P}[\![$<TypeDeclList> <TypeDecl>$]\!]\gamma\rho =$
    $\mathcal{P}[\![$<TypeDeclList>$]\!]\gamma\rho_1$ where
      $\forall\gamma_1.\rho_1(\gamma_1) = \mathcal{D}[\![$<TypeDecl>$]\!]\gamma_1\delta$ where
        $\forall\gamma_1.\delta(\gamma_1) = \rho(\gamma_1)$

$\mathcal{D}[\![$<TypeDecl>$]\!]\gamma\delta$ ::=
  $\mathcal{D}[\![$<ClassDecl>$]\!]\gamma\delta$
| $\mathcal{D}[\![$<InterfaceDecl>$]\!]\gamma\delta$
| $\mathcal{D}[\![$**;**$]\!]\gamma\delta = \delta(\gamma)$

4.4 Types

The $\mathcal{T}[\![]\!]$ semantic function returns the type defined by the given syntactic construct. The returned type is a string representation of the specified type using the notation defined in the JVM [3]. Specifically, the returned values are:

Type	Return Value
boolean	"Z"
byte	"B"
short	"S"
char	"C"
int	"I"
long	"J"
float	"F"
double	"D"
void	"V"
array of *Type*	"[*Type*"
class or interface	"L*classname*;"
method*	$(t_1 t_2 \ldots t_n) t_r$

for methods of the form:
return-type meth-name$(parm_1, parm_2, \ldots, parm_n)$
where t_r is the return type, t_i is the type of $parm_i$.

$\mathcal{T}[\![$<Type>$]\!]\gamma$ ::=
  $\mathcal{T}[\![$<PrimitiveType>$]\!]\gamma$
 | $\mathcal{T}[\![$<ReferenceType>$]\!]\gamma$

$\mathcal{T}[\![$<PrimitiveType>$]\!]\gamma$ ::=
  $\mathcal{T}[\![$<NumericType>$]\!]\gamma$
 | $\mathcal{T}[\![$**boolean**$]\!]\gamma$ = "Z"

$\mathcal{T}[\![$<NumericType>$]\!]\gamma$ ::=
  $\mathcal{T}[\![$<IntegralType>$]\!]\gamma$
  $\mathcal{T}[\![$<FloatingPointType>$]\!]\gamma$

$\mathcal{T}[\![$<IntegralType>$]\!]\gamma$ ::=
  $\mathcal{T}[\![$**byte**$]\!]\gamma$ = "B"
 | $\mathcal{T}[\![$**int**$]\!]\gamma$ = "I"
 | $\mathcal{T}[\![$**long**$]\!]\gamma$ = "J"
 | $\mathcal{T}[\![$**short**$]\!]\gamma$ = "S"
 | $\mathcal{T}[\![$**char**$]\!]\gamma$ = "C"

$\mathcal{T}[\![$<FloatingPointType>$]\!]\gamma$ ::=
  $\mathcal{T}[\![$**float**$]\!]\gamma$ = "F"
 | $\mathcal{T}[\![$**double**$]\!]\gamma$ = "D"

$\mathcal{T}[\![$<ReferenceType>$]\!]\gamma$ ::=
  $\mathcal{T}[\![$<ClassOrInterfaceType>$]\!]\gamma$
 | $\mathcal{T}[\![$<ArrayType>$]\!]\gamma$

$\mathcal{T}[\![$<ClassOrInterfaceType>$]\!]\gamma$ ::=
  $\mathcal{T}[\![$<name>$]\!]\gamma$ = "L"+fst($\mathcal{V}[\![$<Name>$]\!]\gamma$)+";"

$\mathcal{T}[\![$<ClassType>$]\!]\gamma$ ::=
  $\mathcal{T}[\![$<ClassOrInterfaceType>$]\!]\gamma$

$\mathcal{T}[\![$<InterfaceType>$]\!]\gamma$ ::=
  $\mathcal{T}[\![$<ClassOrInterfaceType>$]\!]\gamma$

$\mathcal{T}[\![$<ArrayType>$]\!]\gamma$ ::=
  $\mathcal{T}[\![$<PrimitiveType> [ ]$]\!]\gamma$ =**mkArrayType**(τ_1,1) where
    $\tau_1 = \mathcal{T}[\![$<PrimitiveType>$]\!]\gamma$
| $\mathcal{T}[\![$<Name> [ ]$]\!]\gamma$ = **mkArrayType**(τ_1,1) where
    τ_1 = **fst**($\mathcal{V}[\![$<Name>$]\!]\gamma$)
| $\mathcal{T}[\![$<ArrayType> [ ]$]\!]\gamma$ = **mkArrayType**(τ_1,1) where
    $\tau_1 = \mathcal{T}[\![$<ArrayType>$]\!]\gamma$ where

$$\textbf{mkArrayType}(\tau, n) = \begin{cases} \tau & \texttt{when } n = 0 \\ \text{"["}+\textbf{mkArrayType}(\tau, n-1) & \texttt{when } n > 0 \end{cases}$$

4.5 Modifiers

Modifiers specify the access constraints of classes, methods and fields in Java programs. As such, we need to specify the list of modifiers for the declaration semantic functions that use them. The $\mathcal{M}[\![]\!]$ semantic function returns all modifiers as a list of strings.

$\mathcal{M}[\![$<Modifiers>$]\!]$ ::=
  $\mathcal{M}[\![$<Modifier>$]\!]$
| $\mathcal{M}[\![$<Modifiers$_1$> <Modifier>$]\!]$ = cons($\mathcal{M}[\![$Modifiers$_1]\!]$,$\mathcal{M}[\![$Modifier$]\!]$)

$\mathcal{M}[\![$<Modifier>$]\!]$ ::=
  $\mathcal{M}[\![$**public**$]\!]$ = "public"
| $\mathcal{M}[\![$**private**$]\!]$ = "private"
| $\mathcal{M}[\![$**protected**$]\!]$ = "protected"
| $\mathcal{M}[\![$**static**$]\!]$ = "static"
| $\mathcal{M}[\![$**abstract**$]\!]$ = "abstract"
| $\mathcal{M}[\![$**final**$]\!]$ = "final"
| $\mathcal{M}[\![$**native**$]\!]$ = "native"
| $\mathcal{M}[\![$**synchronized**$]\!]$ = "synchronized"
| $\mathcal{M}[\![$**transient**$]\!]$ = "transient"
| $\mathcal{M}[\![$**volatile**$]\!]$ = "volatile"

4.6 Interface Declarations

Interfaces specify a group of classes. Within each interface is a set of nested class and interface declarations (starting in Java 1.1), constant fields and abstract methods. In addition, an interface can extend another interface. All of

this syntax represents a declaration abstraction for a group of classes. When a class is declared to implement an interface, all of the interface body declarations are included in the beginning of the class declaration. The auxiliary function $enterInterface$ modifies the current environment to include the new interface declaration and members. The current interface is defined in terms of the name of the declared interface for the remainder of the declaration, it then reverts to the calling interface name for the continuation.

$\mathcal{D}[\![$<InterfaceDecl>$]\!]\gamma\delta$ ::=
  $\mathcal{D}[\![$<Modifiers>$^?$ **interface** <Id> <Extends>$^?$ <InterfaceBody>$]\!]\gamma\delta =$
    $\mathcal{D}[\![$<InterfaceBody>$]\!]\gamma_1\delta_1$ where
      $\gamma_1 = \gamma.enterInterface(\mathcal{M}[\![$<Modifiers>$]\!]$,
          $\mathcal{V}[\![$<Id>$]\!]\gamma$,
          $\mathcal{T}[\![$<Extends>$]\!]\gamma)$ and
      $\forall\gamma_2.\delta_1(\gamma_2) = \delta(\gamma_2[\&currentInt \leftarrow \gamma(\&currentInt)])$

$\mathcal{T}[\![$<Extends>$]\!]\gamma$ ::=
    $\mathcal{T}[\![$**extends** <InterfaceType>$]\!]\gamma = \mathcal{T}[\![$<InterfaceType>$]\!]\gamma$
  | $\mathcal{T}[\![$<Extends> **,** <InterfaceType>$]\!]\gamma =$
    $\mathcal{T}[\![$<Extends>$]\!]\gamma + \mathcal{T}[\![$<InterfaceType>$]\!]\gamma$

$\mathcal{D}[\![$<InterfaceBody>$]\!]\gamma\delta$ ::=
    $\mathcal{D}[\![$**{** <InterfaceMemberDeclList>$^?$ **}**$]\!]\gamma\delta =$
    $\mathcal{D}[\![$<InterfaceMemberDeclList>$]\!]\gamma\delta$

$\mathcal{D}[\![$<InterfaceMemberDeclList>$]\!]\gamma\delta$ ::=
    $\mathcal{D}[\![$<InterfaceMemberDecl>$]\!]\gamma\delta$
  | $\mathcal{D}[\![$<InterfaceMemberDeclList$_1$> <InterfaceMemberDecl>$]\!]\gamma\delta =$
    $\mathcal{D}[\![$<InterfaceMemberDecl>$]\!]\gamma\delta_1$ where
      $\forall\gamma_1.\delta_1(\gamma_1) = \mathcal{D}[\![$<InterfaceMemberDecl>$]\!]\gamma_1\delta$

$\mathcal{D}[\![$<InterfaceMemberDecl>$]\!]\gamma\delta$ ::=
    $\mathcal{D}[\![$<ClassDecl>$]\!]\gamma\delta$
  | $\mathcal{D}[\![$<InterfaceDecl>$]\!]\gamma\delta$
  | $\mathcal{D}[\![$<AbsMethodDecl>$]\!]\gamma\delta$
  | $\mathcal{D}[\![$<ConstantDecl>$]\!]\gamma\delta$

$\mathcal{D}[\![$<AbsMethodDecl>$]\!]\gamma\delta$ ::=
    $\mathcal{D}[\![$<MethodHdr> **;**$]\!]\ \gamma\delta = \delta(\gamma_1)$ where
      $\gamma_1 = \gamma.addMethodHdr(v)$ and
      $v = \mathcal{V}[\![$<MethodHdr>$]\!]\ \gamma$

$\mathcal{D}[\![$<ConstantDecl>$]\!]\gamma\delta$ ::=
    $\mathcal{D}[\![$<FieldDecl>$]\!]\gamma\delta$

4.7 Class Declarations

The following grammar presents class declarations. As with interfaces, a class declaration enters a new class, thus modifying the environment. The environment

includes the definitions of all members of the class and links to the super class and implemented interfaces.

$\mathcal{D}[\![$<ClassDecl>$]\!]\gamma\delta$ ::=
  $\mathcal{D}[\![$<Modifiers>$^?$ **class** <Id> <Super>$^?$ <Interfaces>$^?$ <ClassBody>$]\!]\gamma\delta$ =
    $\mathcal{D}[\![$<ClassBody>$]\!]\gamma_1\delta_1$ where
      $\gamma_1 = \gamma.enterClass(\mathcal{M}[\![$<Modifiers>$]\!],$
        $\mathcal{V}[\![$<Id>$]\!]\gamma,$
        $\mathcal{T}[\![$<Super>$]\!]\gamma,$
        $\mathcal{T}[\![$<Interfaces>$]\!]\gamma)$ and
      $\forall\gamma_2.\delta_1(\gamma_2) = \delta(\gamma_2[\&currentClass \leftarrow \gamma(\&currentClass)])$

$\mathcal{T}[\![$<Super>$]\!]\gamma$ ::=
  $\mathcal{T}[\![$**extends** <ClassType>$]\!]\gamma = \mathcal{T}[\![$<ClassType>$]\!]\gamma$

$\mathcal{T}[\![$<Interfaces>$]\!]\gamma$ ::=
  $\mathcal{T}[\![$**implements** <InterfaceTypeList>$]\!]\gamma = \mathcal{T}[\![$<InterfaceTypeList>$]\!]\gamma$

$\mathcal{T}[\![$<InterfaceTypeList>$]\!]\gamma$ ::=
  $\mathcal{T}[\![$<InterfaceType>$]\!]\gamma$
| $\mathcal{T}[\![$<InterfaceTypeList$_1$> , <InterfaceType>$]\!]\gamma$ =
    $\mathcal{T}[\![$<InterfaceTypeList$_1$>$]\!]\gamma + \mathcal{T}[\![$<InterfaceType>$]\!]\gamma$

$\mathcal{D}[\![$<ClassBody>$]\!]\gamma\delta$ ::=
  $\mathcal{D}[\![$<ClassBodyDeclList$^?$>$]\!]\gamma\delta$

$\mathcal{D}[\![$<ClassBodyDeclList>$]\!]\gamma\delta$ ::=
  $\mathcal{D}[\![$<ClassBodyDecl>$]\!]\gamma\delta$
| $\mathcal{D}[\![$<ClassBodyDeclList$_1$> <ClassBodyDecl>$]\!]\gamma\delta$

The Class Body consists of class members which include nested classes, nested interfaces, fields and methods; constructors and static initializers (which are class-level constructors invoked the first time a class is activated). It is important to note that when a class is activated (denoted in these semantics by the *classLoader* function): the parent class is loaded, then all static variables are initialized and all static initializers are executed in the order they appear in the class declaration. The *addStaticInit* routine used below, and the *addStaticField* routine used in the Field Declaration section enters the partial semantic functions for these initializers into the environment. The *classLoader* function recovers these partial functions and completes them with the current parameters.

$\mathcal{D}[\![$<ClassBodyDecl>$]\!]\gamma\delta$ ::=
  $\mathcal{D}[\![$<ClassMemberDecl>$]\!]\gamma\delta$
| $\mathcal{D}[\![$<StaticInit>$]\!]\gamma\delta$ =
    $\delta(\gamma.addStaticInit(\mathcal{C}[\![$<StaticInit>$]\!]))$
| $\mathcal{D}[\![$<ConstrDecl>$]\!]\gamma\delta$

$\mathcal{D}[\![$<ClassMemberDecl>$]\!]\gamma\delta$ ::=
  $\mathcal{D}[\![$<ClassDecl>$]\!]\gamma\delta$

| $\mathcal{D}[\![$<InterfaceDecl>$]\!]\gamma\delta$
| $\mathcal{D}[\![$<FieldDecl>$]\!]\gamma\delta$
| $\mathcal{D}[\![$<MethodDecl>$]\!]\gamma\delta$

4.8 Method Declarations

We have slightly modified the method declaration productions from the grammar presented in the JLS [1], including modifiers and throws directly in the method declaration instead of in the header. This was done to simplify the semantic functions. The major action of these productions is to add a method into the environment of the current class. Associated with the type signature and name of the method is a partial semantic function that defines the operational behavior of the method. When the method is invoked, values of the actual parameters are passed to the formal parameters of the method, and then the method body is executed using the new values. Any resultant value is returned as the result of the method semantic function. The **mkMethodValue** function takes the method name and formal parameter type list and returns what we term a method value. This method value specifies the name and type signature of the method along with the names of the formal parameters. The exact details of this notation is not important here, it is sufficient to know that this information is used by the *addMethod* routine.

$\mathcal{D}[\![$<MethodDecl>$]\!]\gamma\delta$ ::=
  $\mathcal{D}[\![$<Modifiers>$^{?}$ <MethodHdr> <Throws>$^{?}$ <MethodBody>$]\!]\gamma\delta$ =
    $\gamma.addMethod(\mathcal{M}[\![$<Modifiers>$]\!],\mathcal{V}[\![$<MethodHdr>$]\!]\gamma,$
      $\mathcal{T}[\![$<Throws>$]\!]\gamma,\mathcal{C}[\![$<MethodBody>$]\!])$

$\mathcal{V}[\![$<MethodHdr>$]\!]\gamma$ ::=
  $\mathcal{V}[\![$<Type> <MethodDef>$]\!]\gamma$ = **mkMethodValue**($\mathcal{V}[\![$<MethodDef>$]\!]\gamma$,<Type>)
| $\mathcal{V}[\![$**void** <MethodDef>$]\!]\gamma$ = **mkMethodValue**($\mathcal{V}[\![$<MethodDef>$]\!]\gamma$,**void**)

$\mathcal{V}[\![$<MethodDef>$]\!]\gamma$ ::=
  $\mathcal{V}[\![$<Id> (<FormalParmList>$^{?}$)$]\!]\gamma$ =
    (fst($\mathcal{V}[\![$<Id>$]\!]env$),$\mathcal{V}[\![$<FormalParmList>$]\!]\gamma$)
| $\mathcal{V}[\![$<MethodDef> [ ]$]\!]\gamma$ = **mkArrayType**($\mathcal{V}[\![$<MethodDef>$]\!]\gamma$,1)

The formal parameter list is returns a pair that consists of a list names of each of the parameters and a list of corresponding types. The following semantic functions return this pair. The throws clause returns a type that corresponds to the types of each of the thrown classes.

$\mathcal{V}[\![$<FormalParmList>$]\!]\gamma$ ::=
  $\mathcal{V}[\![$<FormalParam>$]\!]\gamma$
| $\mathcal{V}[\![$<FormalParmList> , <FormalParam>$]\!]\gamma$ =
    $\mathcal{V}[\![$<FormalParmList>$]\!]\gamma$ + $\mathcal{V}[\![$<FormalParam>$]\!]\gamma$

$\mathcal{V}[\![$<FormalParam>$]\!]\gamma$ ::=
  $\mathcal{T}[\![$<Modifier> <Type> <VarDeclId>$]\!]\gamma$ = (fst($\mathcal{V}[\![$([$\!]$<VarDecId>),$\mathcal{T}[\![$<Type>$]\!]\gamma$)

$\mathcal{T}[\![$<Throws>$]\!]\gamma$ ::=
  $\mathcal{T}[\![$**throws** <ClassTypeList>$]\!]\gamma = \mathcal{T}[\![$<ClassTypeList>$]\!]\gamma$

$\mathcal{T}[\![$<ClassTypeList>$]\!]\gamma$ ::=
  $\mathcal{T}[\![$<ClassType>$]\!]\gamma$
| $\mathcal{T}[\![$<ClassTypeList> , <ClassType>$]\!]\gamma\delta =$
  $\mathcal{T}[\![$<ClassTypeList>$]\!]\gamma + \mathcal{T}[\![$<ClassType>$]\!]\gamma$

$\mathcal{C}[\![$<MethodBody>$]\!]\gamma\theta\sigma$ ::=
  $\mathcal{C}[\![$;$]\!]\gamma\theta\sigma = \theta(\gamma, \sigma)$
| $\mathcal{C}[\![$<Block>$]\!]\gamma\delta\sigma$

4.9 Field and Variable Declarations

The following semantic functions define the meaning of the field and variable declarations. These declarations are used to modify the environment to define static and regular fields and local variables.

$\mathcal{D}[\![$<FieldDecl>$]\!]\gamma\delta$ ::=
  $\mathcal{D}[\![$<Modifiers>$^?$ <Type> <VarDecl> ;$]\!]\gamma\delta = \mathcal{D}[\![$<VarDecl>$]\!]\gamma_1\delta$ where
    $\gamma_1 = \gamma[\&Mods \leftarrow \mathcal{M}[\![$<Modifiers>$]\!]$,
      $\&Type \leftarrow \mathcal{T}[\![$<Type>$]\!]\gamma$,
      $\&varType \leftarrow$ "Field"]

$\mathcal{D}[\![$<VarDeclList>$]\!]\gamma\delta$ ::=
  $\mathcal{D}[\![$<VarDecl>$]\!]\gamma\delta$
| $\mathcal{D}[\![$<VarDeclList> , <VarDecl>$]\!]\gamma\delta = \mathcal{D}[\![$<VarDeclList>$]\!]\gamma\delta_1$ where
    $\forall\gamma_1.\delta_1(\gamma_1) = \mathcal{D}[\![$<VarDecl>$]\!]\gamma_1\delta$

$\mathcal{D}[\![$<VarDecl>$]\!]\gamma\delta$ ::=
  $\mathcal{D}[\![$<VarDeclId>$]\!]\gamma\delta = \delta(\gamma_1)$ where
    let $(name, type) = \mathcal{V}[\![$<VarDeclId>$]\!]\gamma$ in
    $\gamma_1 =$
      if $(\gamma(\&varType) ==$ "Field")
        if $(\gamma.isStatic())$)
          $\gamma.addStaticField(name$,defaultInit$(type))$
        else
          $\gamma.addField(name$,defaultInit$(type))$
        endif
      else //$(\gamma(\&varType) ==$ "Local")
        $\gamma.addLocal(name$,defaultInit$(type))$
      endif
| $\mathcal{D}[\![$<VarDeclId> = <VarInit>$]\!]\gamma\delta = \delta(\gamma_1)$ where
    let $(name, type) = \mathcal{V}[\![$<VarDeclId>$]\!]\gamma$ in
    $\gamma_1 =$
      if $(\gamma(\&varType) ==$ "Field")
        if $(\gamma.isStatic())$)
          $\gamma.addStaticField(name$,$\mathcal{E}[\![$<VarInit>$]\!])$

else
$\gamma.addField(name,\mathcal{E}[\![$<VarInit>$]\!])$
endif
else //(γ(&$varType$) == "Local")
$\gamma.addLocal(name,\mathcal{E}[\![$<VarInit>$]\!])$
endif

$\mathcal{V}[\![$<VarDeclId>$]\!]\gamma$ ::=
$\mathcal{V}[\![$<Id>$]\!]\gamma$ = (ValueOf(<Id>),γ(&$Type$))
| $\mathcal{D}[\![$<VarDeclId> [ ]$]\!]\gamma\delta$ = (ValueOf(<Id>),**mkArrayType**(γ(&$Type$),1))

4.10 Initializers

Initializers consist of both static block initializers for classes and field and local variable initializers. All initializers are simply evaluated upon instantiation of the class, field or variable. For fields and variables the resultant value is a pair consisting of a list of values and a list of types corresponding to these values.

$\mathcal{C}[\![$<StaticInit>$]\!]\gamma\theta\sigma$ ::=
$\mathcal{C}[\![$**static** <Block>$]\!]\gamma\theta\sigma = \mathcal{C}[\![$<Block>$]\!]\gamma\theta\sigma$

$\mathcal{E}[\![$<VarInits>$]\!]\gamma\kappa\sigma$ ::=
$\mathcal{E}[\![$<VarInit>$]\!]\gamma\kappa\sigma$
| $\mathcal{E}[\![$<VarInits$_1$> , <VarInit>$]\!]\gamma\kappa\sigma = \mathcal{E}[\![$<VarInits$_1$>$]\!]\gamma\kappa_1\sigma$ where
$\forall r_1, \tau_1, \sigma_1.\ \kappa_1(r_1, \tau_1, \sigma_1) = \mathcal{E}[\![$<VarInit>$]\!]\gamma\kappa_2\sigma_1$ where
$\forall r_2, \tau_2, \sigma_2.\ \kappa_2(r_2, \tau_2, \sigma_2) = \kappa(q, \tau, \sigma_2)$ where
q = **append**(r_1, r_2) and
$\tau = \tau_1 + \tau_2$

$\mathcal{E}[\![$<VarInit>$]\!]\gamma\kappa\sigma$::=
$\mathcal{E}[\![$<Expression>$]\!]\gamma\kappa\sigma$
$\mathcal{E}[\![$<ArrayInitializer>$]\!]\gamma\kappa\sigma$

$\mathcal{E}[\![$<ArrayInit>$]\!]\gamma\kappa\sigma$ ::=
$\mathcal{E}[\![$\{ <VarInits>$^?$,$^?$ \}$]\!]\gamma\kappa\sigma = \mathcal{E}[\![$<VarInits>$]\!]\gamma\kappa_1\sigma$ where
$\forall r, \tau, \sigma.\kappa_1(r, \tau, \sigma) = \kappa(r, \tau_1, \sigma)$ where
τ_1 = **mkArrayType**(τ,1)

4.11 Constructor Declarations

The constructor semantic functions define the meaning of object constructors. It is important to understand that when a constructor is invoked, it either calls an implicit constructor of the super class or an explicit constructor. This is denoted in the semantic functions below. The *instantiateClass* function of stores return a triple consisting of a value (the reference to the new object), a type (of the object), and a new store that contains the new locations for the fields of the object.

$\mathcal{D}[\![$<ConstrDecl>$]\!]\gamma\delta$ ::=
  $\mathcal{D}[\![$<Modifiers>$^?$ <ConstrDef> <Throws>$^?$ <ConstrBody>$]\!]\gamma\delta = \delta(\gamma_1)$ where
    $\gamma_1 = \gamma.addConstr(\mathcal{M}[\![$<Modifiers>$]\!], \mathcal{V}[\![$<ConstrDef>$]\!]\gamma,$
      $\mathcal{T}[\![$<Throws>$]\!]\gamma, \mathcal{E}[\![$<ConstrBody>$]\!])$

$\mathcal{V}[\![$<ConstrDef>$]\!]\gamma$ ::=
  $\mathcal{V}[\![$<SimpleName> **(** <FormalParmList>$^?$ **)**$]\!]\gamma =$
    $(\mathrm{fst}(\mathcal{V}[\![$<SimpleName>$]\!]\gamma), \mathcal{V}[\![$<FormalParmList>$]\!]\gamma)$

$\mathcal{E}[\![$<ConstrBody>$]\!]\gamma\kappa\sigma$ ::=
  $\mathcal{C}[\![$**{** <ExplConstrInv> <BlockStmtList>$^?$ **}**$]\!]\gamma\kappa\sigma =$
    $\mathcal{E}[\![$<ExplConstrInv>$]\!]\gamma\kappa_1\sigma$ where
      $\forall r, \tau, \sigma_1.\kappa_1(r, \tau, \sigma_1) = \theta(\sigma_1)$ where
        $\forall \sigma_2.\theta(\sigma_2) = \mathcal{C}[\![$<BlockStmtList>$]\!]\gamma, \theta_1, \sigma_3$ where
          let $(r_1, \tau_1, \sigma_3) = \sigma_2.instantiateClass(r)$ in
          $\forall \sigma_4.\theta_1(\sigma_4) = \kappa(r_1, \tau_1, \sigma_4)$
  | $\mathcal{E}[\![$**{** <BlockStmtList>$^?$ **}**$]\!]\gamma\kappa\sigma =$
    $\mathcal{E}[\![$**super ()**$]\!]\gamma\kappa_1\sigma$ where
      $\forall r, \tau, \sigma_1.\kappa_1(r, \tau, \sigma_1) = \theta(\sigma_1)$ where
        $\forall \sigma_2.\theta(\sigma_2) = \mathcal{C}[\![$<BlockStmtList>$]\!]\gamma, \theta_1, \sigma_3$ where
          let $(r_1, \tau_1, \sigma_3) = \sigma_2.instantiateClass(r)$ in
          $\forall \sigma_4.\theta_1(\sigma_4) = \kappa(r_1, \tau_1, \sigma_4)$

$\mathcal{E}[\![$<ExplConstrInv>$]\!]\gamma\kappa\sigma$ ::=
  $\mathcal{E}[\![$**this (** <ArgList>$^?$ **) ;**$]\!]\gamma\kappa\sigma$
  | $\mathcal{E}[\![$**super (** <ArgList>$^?$ **) ;**$]\!]\gamma\kappa\sigma$

4.12 Blocks and Statements

In this section, we present the semantics for blocks and statements in the Java language. We differ from the grammar in the JLS [1], by not presenting any of the statements associated with the *No Short If* constructs, used in the JLS to avoid syntactic ambiguity with dangling else clauses. The semantics for all of these clauses can be easily derived from the semantics presented here.

Blocks. A block consists of an optional sequence of block statements within a pair of braces. Note here, that if there are no block statements, the semantics of the block are equivalent to $\theta(\gamma, \sigma)$.

$\mathcal{C}[\![$<Block>$]\!]\gamma\theta\sigma$ ::=
  $\mathcal{C}[\![$**{** <BlockStmtList>$^?$ **}**$]\!]\gamma\theta\sigma = \mathcal{C}[\![$BlockStmtList$]\!]\gamma\theta\sigma$

$\mathcal{C}[\![$<BlockStmtList>$]\!]\gamma\theta\sigma$ ::=
  $\mathcal{C}[\![$<BlockStmt>$]\!]\gamma\theta\sigma$
  | $\mathcal{C}[\![$<BlockStmtList$_1$> <BlockStmt>$]\!]\gamma\theta\sigma = \mathcal{C}[\![$<BlockStmtList$_1$>$]\!]\gamma\theta_1\sigma$ where
      $\forall \gamma_1, \sigma_1.\theta_1(\gamma_1, \sigma_1) = \mathcal{C}[\![$<BlockStmt>$]\!]\gamma_1\theta\sigma_1$

$\mathcal{C}[\![$<BlockStmt>$]\!]\gamma\theta\sigma$ ::=
  $\mathcal{C}[\![$<LocalVarDeclStmt>$]\!]\gamma\theta\sigma$

$| \mathcal{C}[\![$<Stmt>$]\!]\gamma\theta\sigma$

$\mathcal{C}[\![$<LocalVarDeclStmt>$]\!]\gamma\theta\sigma$ ::=
$\mathcal{C}[\![$<LocalVarDecl> **;**$]\!]\gamma\theta\sigma$

Local Variable Declarations. These modify both the environment and the store (local store), by creating a new semantic entity. As such, a local variable declaration will continue program execution with these new attributes.

$\mathcal{C}[\![$<LocalVarDecl>$]\!]\gamma\theta\sigma$ ::=
$\mathcal{C}[\![$<Type> <VarDeclList>$]\!]\gamma\theta\sigma = \mathcal{D}[\![$<VarDeclList>$]\!]\gamma\delta\sigma$ where
$\forall d, \gamma_1, \sigma_1.\delta(d, \gamma_1, \sigma_1) = \theta(\gamma_2, \sigma_1)$ where
$\gamma_2 = \gamma_1[\&Type \leftarrow \mathcal{T}[\![$<Type>$]\!]\gamma, \&varType \leftarrow$ "Local"]

Empty, Labeled and Expression Statements. These statements are basic primitive statements of the Java language. The empty statement consists solely of a single semicolon and semantically continues operation as if nothing happened. The labeled statement modifies the environment to contain an identifier, *Id* that refers to the current statement. The environment maintains the semantic evaluation of the statement as a function of the current statement, parameterized by possibly new environment and store. Note that the environment of the statement contains a reference to the label, *Id*, but upon completion of execution, that label is removed from the environment. The expression statement evaluates the expression using the semantic function for expressions, discarding any returned value or type, and continues execution using the possibly modified store. Note that we have simplified an expression statement to consist of any expression, although this is not strictly true. In the JLS [1], the grammar restricts expression statements to a list of possible expressions. We take liberty with our assumption of syntactically correct programs to simplify the grammar here.

$\mathcal{C}[\![$<EmptyStmt>$]\!]\gamma\theta\sigma$ ::=
$\mathcal{C}[\![$**;**$]\!]\gamma\theta\sigma = \theta(\gamma, \sigma)$

$\mathcal{C}[\![$<LabeledStmt>$]\!]\gamma\theta\sigma$ ::=
$\mathcal{C}[\![$<Id> **:** <Stmt>$]\!]\gamma\theta\sigma = \mathcal{C}[\![$<Stmt>$]\!]\gamma_1\theta_1\sigma$ where
$\gamma_1 = \gamma[Id \leftarrow \theta_2]$ where
$\forall\gamma_2, \sigma_2.\theta_2(\gamma_2, \sigma_2) = \mathcal{C}[\![$<Stmt>$]\!]\gamma_2\theta_1\sigma_2$
$\forall\gamma_1, \sigma_1.\theta_1(\gamma_1, \sigma_1) = \theta(\gamma, \sigma_1)$

$\mathcal{C}[\![$<ExprStmt> **;**$]\!]\gamma\theta\sigma$ ::=
$\mathcal{C}[\![$<Expr>$]\!]\gamma\theta\sigma = \mathcal{E}[\![$<Expr>$]\!]\gamma\kappa\sigma$ where
$\forall r, \tau, \sigma_1.\kappa(r, \tau, \sigma_1) = \theta(\gamma, \sigma_1)$

If Statements. The if statement has two forms, one with and one without an else clause. The if statement executes the expression first, possibly modifying the store, and then behaves as the first statement if the expression is true. If the expression is false it either continues execution or behaves as the statement following the else clause.

$\mathcal{C}[\![$<IfStmt>$]\!]\gamma\theta\sigma$ ::=
  $\mathcal{C}[\![$**if** (<Expr>) <Stmt>$]\!]\gamma\theta\sigma = \mathcal{E}[\![$Expr$]\!]\gamma\kappa\sigma$ where
    $\forall r, \tau, \sigma_1.\kappa(r, \tau, \sigma_1) =$
      if $(r ==$ **true**$)$
        $\mathcal{C}[\![$<Stmt>$]\!]\gamma\theta\sigma_1$
      else
        $\theta(\gamma, \sigma_1)$
      endif

$\mathcal{C}[\![$<IfElseStmt>$]\!]\gamma\theta\sigma$ ::=
  $\mathcal{C}[\![$**if** (<Expr>) <Stmt$_1$> **else** Stmt$_2$$]\!]\gamma\theta\sigma = \mathcal{E}[\![$<Expr>$]\!]\gamma\kappa\sigma$ where
    $\forall r, \tau, \sigma_1.\kappa(r, \tau, \sigma_1) =$
      if $(r ==$ **true**$)$
        $\mathcal{C}[\![$<Stmt$_1$>$]\!]\gamma\theta\sigma_1$
      else
        $\mathcal{C}[\![$<Stmt$_1$>$]\!]\gamma\theta\sigma_1$
      endif

The Switch Statement. The Java switch statement presents a few problems for the design of a denotational semantics. The problems we found and their resolution are discussed below. The approach we took involves modification of the environment to provide additional information to subsequent semantic functions. This modification is in the form of auxiliary variables. Note that these variables must be restored to their previous values upon completion of the switch statement to permit the correct evaluation of nested switch statements.

- The data value obtained upon execution of the switch statement determines which case to execute. Thus this value must be carried along through the semantic functions until it is utilized. We decided to maintain the value in the environment under the auxiliary variable name $\&switchExpr$.
- Once a case label has been found to match the switch expression, all subsequent switch block statements are to be executed. Thus, the semantic meaning of these statements is dependent on whether or not a case label matched the switch expression. We decided to maintain a boolean flag, $\&caseFound$, in the environment to indicate whether or not a match has been found.
- The default switch case label may occur any place a case label may occur. If no case label matches the switch expression, the meaning of the switch statement is the meaning of all switch block statements that follow the default label. The problem is that not only do we have to inform the semantic evaluation functions that a default label has been found, but the functions also have to allow for the existence of a matching case label occurring after the default label. The first problem is resolved with the boolean flag $\&defaultFound$, which operates the same as the $\&caseFound$ flag. The other problem is resolved using the $\&caseCont$ variable which records the environment, store and continuation parameters for the switch statement.
- Unlabelled break statement may occur within a switch statement. The intent of this statement is to terminate execution of the switch statement. As such, the break information is stored in the environment.

$\mathcal{C}[\![$<SwitchStmt>$]\!]\gamma\theta\sigma$ ::=
  $\mathcal{C}[\![$**switch (** <Expr> **)** <SwitchBlock>$]\!]\gamma\theta\sigma$ =
    $\mathcal{E}[\![$<Expr>$]\!]\gamma\kappa\sigma$ where
      $\forall r, \tau, \sigma_1.\ \kappa(r, \tau, \sigma_1) = \mathcal{C}[\![$<SwitchBlock>$]\!]\gamma_1\theta_1\sigma_1$ where
        $\gamma_1 = \gamma[\&switchExpr \leftarrow r, \&caseFound \leftarrow false,$
          $\&defaultFound \leftarrow false, \&caseCont \leftarrow (\gamma, \theta_1, \sigma_1),$
          $\&break \leftarrow \theta_1)]$ and
        $\forall \gamma_1, \sigma_2.\ \theta_1(\gamma_1, \sigma_2) = \theta(\gamma, \sigma_2)$

$\mathcal{C}[\![$<SwitchBlock>$]\!]\gamma\theta\sigma$ ::=
  $\mathcal{C}[\![$**{** <SwitchBlockStmtList>$^?$ <SwitchLabelList>$^?$ **}**$]\!]\gamma\theta\sigma$ =
    $\mathcal{C}[\![$<SwitchBlockStmtList>$^?]\!]\gamma\theta_1\sigma$ where
      $\forall \gamma_1, \sigma_2.\ \theta_1(\gamma_1, \sigma_2) = \mathcal{C}[\![$<SwitchLabelList>$]\!]\gamma_1\theta\sigma_2$

$\mathcal{C}[\![$<SwitchBlockStmtList>$]\!]\gamma\theta\sigma$ ::=
  $\mathcal{C}[\![$<SwitchBlockStmt>$]\!]\gamma\theta\sigma$
| $\mathcal{C}[\![$<SwitchBlockStmtList$_1$> <SwitchBlockStmt>$]\!]\gamma\theta\sigma$ =
    $\mathcal{C}[\![$<SwitchBlockStmtList>$^?]\!]\gamma\theta_1\sigma$ where
    $\forall \gamma_1, \sigma_1.\ \theta_1(\gamma_1, sto_1) = \mathcal{C}[\![$<SwitchBlockStmt>$^?]\!]\gamma_1\theta\sigma_1$

$\mathcal{C}[\![$<SwitchBlockStmt>$]\!]\gamma\theta\sigma$ ::=
  $\mathcal{C}[\![$<SwitchLabelList> <BlockStmtList>$]\!]\gamma\theta\sigma$ =
    if $(\gamma.getValue(\&caseFound) ==$ **true**)
      $\mathcal{C}[\![$BlockStmtList$]\!]\gamma\theta\sigma$
    else
      $\mathcal{C}[\![$<SwitchLabelList>$]\!]\gamma\theta_1\sigma$ where
        $\forall \gamma_1, \sigma_1.\ \theta_1(\gamma_1\sigma_1) =$
          if $(\gamma_1(\&caseFound) ==$ **true**)
            $\mathcal{C}[\![$BlockStmtList$]\!]\gamma_1(\&caseCont)$
          else if $\gamma_1(\&defaultFound)$
            $\mathcal{C}[\![$BlockStmtList$]\!]\gamma_1\theta\sigma_1$
          else
            $\theta(\gamma_1, \sigma_1)$
          endif
    endif

$\mathcal{C}[\![$<SwitchLabelList>$]\!]\gamma\theta\sigma$ ::=
  $\mathcal{C}[\![$<SwitchLabel>$]\!]\gamma\theta\sigma$
| $\mathcal{C}[\![$<SwitchLabelList$_1$> <SwitchLabel>$]\!]\gamma\theta\sigma$ =
    $\mathcal{C}[\![$<SwitchLabelList$_1$>$]\!]\gamma\theta_1\sigma$ where
      $\forall \gamma_1, \sigma_1.\ \theta_1(\gamma_1\sigma_1) = \mathcal{C}[\![$<SwitchLabel>$]\!]\gamma_1\theta\sigma_1$

$\mathcal{C}[\![$<SwitchLabel>$]\!]\gamma\theta\sigma$ ::=
  $\mathcal{C}[\![$**case** <ConstExpr>$]\!]\gamma\theta\sigma = \mathcal{E}[\![$<ConstExpr>$]\!]\gamma\kappa\sigma$ where
    $\forall r, \tau, \sigma_1.\ \kappa(r, \tau, \sigma_1) =$
      if (fst(r) $== \gamma[\&switchExpr]$)
        $\theta(\gamma[\&caseFound \leftarrow true], \sigma_1)$
      else

$\theta(\gamma, \sigma_1)$
endif
$\mid \mathcal{C}[\![\textbf{default :}]\!]\gamma\theta\sigma = \theta(\gamma[\&defaultFound \leftarrow true], \sigma)$

Looping Statements. The Java language looping constructs, the while, do and for statements, are similar to the looping constructs of other languages. And as such, they cause difficulty for the writing of denotational semantics. Two different approaches to specifying the semantics of loops have been presented in the literature, the fixpoint approach [4] and the recursive definition approach [5]. We have defined the do-statement and the for-statement in terms of the while statement. Note that the expression in the for statement is optional. In the case where it is not present, the semantics need to assume that the result is always true; we have divided this case into two separate syntactic forms for clarity.

$\mathcal{C}[\![$<WhileStmt>$]\!]\gamma\theta\sigma$ ::=
$\mathcal{C}[\![\textbf{while (}$ <Expr> $\textbf{)}$ <Statement>$]\!]\gamma\theta\sigma = \theta_1(\gamma[\&break \leftarrow \theta], \sigma)$ where
rec, $\forall\gamma_1, \sigma_1.\ \theta_1(\gamma_1, \sigma_1) = \mathcal{E}[\![$<Expr>$]\!]\gamma_1\kappa\sigma$ where
$\forall r, \tau, \sigma_1.\ \kappa(r, \tau, \sigma_1) =$
if $(r == true)$
$\mathcal{C}[\![$<Statement>$]\!]\gamma_1\theta_1\sigma_1$
else
$\theta(\gamma, \sigma_1)$
endif

$\mathcal{C}[\![$<DoStmt>$]\!]\gamma\theta\sigma$ ::=
$\mathcal{C}[\![\textbf{do}$ <Statement> $\textbf{while (}$ <Expr> $\textbf{)}]\!]\gamma\theta\sigma =$
$\mathcal{C}[\![$<Statement> **; while (** <Expr> **)** <Statement>$]\!]\gamma\theta\sigma$

$\mathcal{C}[\![$<ForStmt>$]\!]\gamma\theta\sigma$ ::=
$\mathcal{C}[\![\textbf{for (}$ <ForInit>$^?$ **; ;** <ForUpdate>$^?$ **)** <Statement>$]\!]\gamma\theta\sigma =$
$\mathcal{C}[\![$<ForInit> **; while (true)** <Statement> ; <ForUpdate>$]\!]\gamma\theta\sigma$
$\mid \mathcal{C}[\![\textbf{for (}$ <ForInit>$^?$ **;** <Expr> **;** <ForUpdate>$^?$ **)** <Statement>$]\!]\gamma\theta\sigma =$
$\mathcal{C}[\![$<ForInit> **; while (** <Expr> **)** <Statement> ; <ForUpdate>$]\!]\gamma\theta\sigma$

$\mathcal{C}[\![$<ForInit>$]\!]\gamma\theta\sigma$ ::=
$\mathcal{C}[\![$<StmtExprList>$]\!]\gamma\theta\sigma$
$\mid \mathcal{C}[\![$<LocalVarDecl>$]\!]\gamma\theta\sigma$

$\mathcal{C}[\![$<ForUpdate>$]\!]\gamma\theta\sigma$ ::=
$\mathcal{C}[\![$<StmtExprList>$]\!]\gamma\theta\sigma$

$\mathcal{C}[\![$<StmtExprList>$]\!]\gamma\theta\sigma$ ::=
$\mathcal{C}[\![$<ExprStmt>$]\!]\gamma\theta\sigma$
$\mid \mathcal{C}[\![$<StmtExprList$_1$> **,** <ExprStmt>$]\!]\gamma\theta\sigma = \mathcal{C}[\![$<StmtExprList$_1$>$]\!]\gamma\theta_1\sigma$ where
$\forall\gamma_1, \sigma_1.\theta_1(\gamma_1, \sigma_1) = \mathcal{C}[\![$<ExprStmt>$]\!]\gamma_1\theta\sigma_1$

Misc. The following semantic functions define the behavior of the miscellaneous syntactic commands in the Java language. The expression statement list involves evaluation of list of expression, with the result values discarded. The break,

continue, return, and throw statements all evaluate their parameters and then look up the corresponding continuation in the environment. The meaning of the rest of the program is based on this continuation. The synchronized command is ignored in these semantics since we do not specify concurrency.

$\mathcal{C}[\![$<ExprStmtList>$]\!]\gamma\theta\sigma$ ::=
  $\mathcal{C}[\![$<ExprStmtList$_1$> , <ExprStmt>$]\!]\gamma\theta\sigma = \mathcal{C}[\![$<ExprStmtList$_1$>$]\!]\gamma\theta_1\sigma$ where
    $\forall\gamma_1, \sigma_1.\ \theta_1(\gamma_1, \sigma_1) = \mathcal{E}[\![$<ExprStmt>$]\!]\gamma_1\kappa\sigma_1$ where
      $\forall r, \tau, \sigma_2.\ \kappa(r, \tau, \sigma_2) = \theta_1(\gamma_1, \sigma_2)$

$\mathcal{C}[\![$<BreakStmt>$]\!]\gamma\theta\sigma$ ::=
  $\mathcal{C}[\![$**break ;**$]\!]\gamma\theta\sigma = \theta_1(\gamma\sigma)$ where
    $\theta_1 = \gamma.getComCont(\&break)$
  | $\mathcal{C}[\![$**break** <Id> **;**$]\!]\gamma\theta\sigma = \theta_1(\gamma\sigma)$ where
    $\theta_1 = \gamma.getComCont(\&break)$ (fst($\mathcal{V}[\![$Id$]\!]\gamma$))

$\mathcal{C}[\![$<ContStmt>$]\!]\gamma\theta\sigma$ ::=
  $\mathcal{C}[\![$**continue;**$]\!]\gamma\theta\sigma = \theta_1(\gamma\sigma)$ where
    $\theta_1 = \gamma.getComCont(\&continue))$
  | $\mathcal{C}[\![$**continue** <Id> **;**$]\!]\gamma\theta\sigma = \theta_1(\gamma\sigma)$ where
    $\theta_1 = \gamma.getComCont(\&continue)$ fst($\mathcal{V}[\![$Id$]\!]\gamma$)

$\mathcal{C}[\![$<RetStmt>$]\!]\gamma\theta\sigma$ ::=
  $\mathcal{C}[\![$**return ;**$]\!]\gamma\theta\sigma = \gamma.getComCont(\&return)$
  | $\mathcal{C}[\![$**return** <Expr> **;**$]\!]\gamma\theta\sigma = \mathcal{E}[\![$<Expr>$]\!]\gamma\kappa\sigma$ where
    $\forall r, \tau, \sigma.\kappa(r, \tau, \sigma) = \theta_1(\sigma_1)$ where
      $\theta_1 = \gamma.getComCont(\&return)$ and
      $\tau_1 = \gamma[\&returnType]$ and
      $r_1 = \mathbf{promote}(\tau_1, (r, \tau)$ and
      $\sigma_1 = \sigma[\&returnVal \leftarrow r_1]$

$\mathcal{C}[\![$<ThrowStmt>$]\!]\gamma\theta\sigma$ ::=
  $\mathcal{C}[\![$**throw** <Expr> **;**$]\!] = \mathcal{E}[\![$<Expr>$]\!]\gamma\kappa\sigma$ where
    $forallr, \tau, \gamma_1, \sigma_2.\kappa(r, \tau, \gamma_1,]sto_1) = \theta_1(\gamma_2, \sigma_1)$ where
      $\gamma_2 = \gamma_1[\&thrown \leftarrow (r, \tau)]$ and
      $\theta_1 = \gamma.[\&throw]$

$\mathcal{C}[\![$<SynchStmt>$]\!]\gamma\theta\sigma$ ::=
  $\mathcal{C}[\![$**synchronized (** <Expr> **)** <Block>$]\!]\gamma\theta\sigma = \mathcal{E}[\![$<Expr>$]\!]\gamma\kappa\sigma$ where
    $\forall r, \tau, \sigma_1.\ \kappa(r, \tau, \sigma_1) = \mathcal{C}[\![$<Block>$]\!]\gamma\theta\sigma_1$

The try-catch statements of the Java language are an important aspect of the language for error control. The following semantic functions capture the meaning of these statements. A try block is executed until a throw command is executes, at that point the execution continues based on the continuation stored in the environment. This continuation consists o the execution of the finally block followed by evaluation of the catch parameter. If the thrown exception matches the formal parameter, the catch clause is executed and the program continues using the continuation from the commands following the try block. If none of the catch clauses match, then the throw propagates on up.

$\mathcal{C}[\![$<TryStmt>$]\!]\gamma\theta\sigma$ ::=
  $\mathcal{C}[\![$**try** <Block> <Catches>$]\!]\gamma\theta\sigma = \mathcal{C}[\![$<Block>$]\!]\gamma_1\theta_1\sigma$ where
    $\gamma_1 = \gamma[\&throw \leftarrow \theta_2]$ and
    $\forall\gamma_2, \sigma_2.\theta_1(\gamma_2, \sigma_2) = \theta(\gamma, \sigma_2)$ and
    $\forall\gamma_2, \sigma_2.\theta_2(\gamma_2, \sigma_2) = \mathcal{C}[\![$<Catches>$]\!]\gamma\theta_3\sigma_2$ where
      $\forall\gamma_3, \sigma_3.\theta_3(\gamma_3, \sigma_3) =$
        if $(\gamma_3(\&thrown) == (null,$ "V$''$)) then
          $\theta(\gamma_3, \sigma_3)$
        else
          $(\gamma_3.[\&throw])(\gamma_3, \sigma_3)$
        endif
  | $\mathcal{C}[\![$**try** <Block> <Catches>$^?$ <Finally>$]\!]\gamma\theta\sigma = \mathcal{C}[\![$<Block>$]\!]\gamma_1\theta_1\sigma$ where
    $\gamma_1 = \gamma[\&throw \leftarrow \theta_2]$ and
    $\forall\gamma_2, \sigma_2.\theta_1(\gamma_2, \sigma_2) = \mathcal{C}[\![$<finally>$]\!]\gamma\theta\sigma_2$ and
    $\forall\gamma_2, \sigma_2.\theta_2(\gamma_2, \sigma_2) = \mathcal{C}[\![$<finally>$]\!]\gamma\theta_3\sigma_2$ where
      $\forall\gamma_3, sto_3.\theta_3(\gamma_3, \sigma_3) = \mathcal{C}[\![$<Catches>$]\!]\gamma_3\theta_4\sigma_3$ where
      $\forall\gamma_4, \sigma_4.\theta_4(\gamma_4, \sigma_4) =$
        if $(\gamma_4(\&thrown) == (null,$ "V$''$)) then
          $\theta(\gamma_4, \sigma_4)$
        else
          $(\gamma_4.[\&throw])(\gamma_4, \sigma_4)$
        endif

$\mathcal{C}[\![$<Catches>$]\!]\gamma\theta\sigma$ ::=
  $\mathcal{C}[\![$<CatchClause>$]\!]\gamma\theta\sigma$
  | $\mathcal{C}[\![$<Catches$_1$> <CatchClause>$]\!]\gamma\theta\sigma = \mathcal{C}[\![$<Catches$_1$>$]\!]\gamma\theta_1\sigma$ where
    $forall\gamma_1, \sigma_1.\theta_1(\gamma_1, \sigma_1) = \mathcal{C}[\![$<CatchClause>$]\!]\gamma_1\theta\sigma_1$

$\mathcal{C}[\![$<CatchClause>$]\!]\gamma\theta\sigma$ ::=
  $\mathcal{C}[\![$**catch (** <FormalParam> **)** <Block>$]\!]\gamma\theta\sigma =$
    let $(r, \tau) = \gamma(\&thrown)$ and
      $(e, \tau_1 = \mathcal{V}[\![$<FormalParm>$]\!])$ in
    if $(\tau == \tau_1)$ then
      $\mathcal{C}[\![$<Block>$]\!]\gamma_1\theta\sigma_1$ where
        $\gamma_1 = \gamma[\&thrown \leftarrow (null,$ "V$''$)] and
        $\sigma_1 = \sigma[\gamma(e) \leftarrow r]$
    else
      $\theta(\gamma, \sigma)$
    endif

$\mathcal{C}[\![$<Finally>$]\!]\gamma\theta\sigma$ ::=
  $\mathcal{C}[\![$**finally** <Block>$]\!]\gamma\theta\sigma = \mathcal{C}[\![$<Block>$]\!]\gamma\theta\sigma$

4.13 Expressions

Expressions in Java return either values or variables. In these semantics we have broken these into two categories, handled by different semantic function s, $\mathcal{E}[\![]\!]$ for values and $\mathcal{L}[\![]\!]$ for variables. The first two syntactic expressions denote constant expressions (which must return a value) and general expressions (which also

return values). Note that restriction that constant expressions return constant values is a compile-time check and thus is not represented in these semantics.

$\mathcal{E}[\![$<ConstantExpr>$]\!]\gamma\kappa\sigma$ ::=
    $\mathcal{E}[\![$<Expr>$]\!]\gamma\kappa\sigma$

$\mathcal{E}[\![$<Expr>$]\!]\gamma\kappa\sigma$ ::=
    $\mathcal{E}[\![$<AssignExpr>$]\!]\gamma\kappa\sigma$

Assignment Expressions: There are several assignment operators in Java besides the simple assignment. According to the JLS [1] the compound assignment $E_1 op = E_2$ is equivalent to $E_1 = (T)((E_1)op(E_2))$ where T is the type of E_1 and the expression E_1 is evaluated only once. In the semantic model, we evaluate E_1 once to obtain its memory location for assignment in the store, and use that location to determine the value of the expression for the operation. This evaluation requires the expression to return a variable (actually a variable location for use by the store) as opposed to a value. To indicate this return type we use the location $\mathcal{L}[\![]\!]$ semantic functions.

$\mathcal{E}[\![$<AssignExpr>$]\!]\gamma\kappa\sigma$ ::=
    $\mathcal{E}[\![$<CondExpr>$]\!]\gamma\kappa\sigma$
  | $\mathcal{E}[\![$<Assign>$]\!]\gamma\kappa\sigma$

$\mathcal{E}[\![$<Assign>$]\!]\gamma\kappa\sigma$ ::=
    $\mathcal{E}[\![$<LHS> <AssignOp> <AssignExpr>$]\!]\gamma\kappa\sigma$ =
      if (AssignOp == '+ =')
        $\mathcal{L}[\![$<LHS>$]\!]\gamma\alpha\sigma$ where
          $\forall r_1, \tau_1, l, \sigma_1.\ \alpha(r_1, \tau_1, l, \sigma_1) = \mathcal{E}[\![$<AssignExpr>$]\!]\gamma\kappa_2\sigma_1$ where
            $\forall r_2, \tau_2, \sigma_2.\ \kappa_2(r_2, \tau_2, \sigma_2) =$
            let $\tau = \mathbf{binaryPromoteType}(\tau_1, \tau_2)$ and
              $d = \mathbf{cast}(\tau_1,(\mathbf{promote}(\tau, (r_1, \tau_1)) +_\tau \mathbf{promote}(\tau, (r_2, \tau_2)),\tau))$ in
            $\kappa(d, \tau_1, \sigma_2[l \leftarrow d])$

    similar for $- =, * =, \% =, \& =, \ ^\wedge =, | =$
    where the meaning of op_τ is defined in the section on numeric expressions

      else if (AssignOp == '/ =')
        $\mathcal{L}[\![$<LHS>$]\!]\gamma\alpha\sigma$ where
          $\forall r_1, \tau_1, l, \sigma_1.\ \alpha(r_1, \tau_1, l, \sigma_1) = \mathcal{E}[\![$<AssignExpr>$]\!]\gamma\kappa_2\sigma_1$ where
            $\forall r_2, \tau_2, \sigma_2.\ \kappa_2(r_2, \tau_2, \sigma_2) =$
            let $\tau = \mathbf{binaryPromoteType}(\tau_1, \tau_2)$ in
            if $(r_2 = 0 \wedge (\tau =$ "$\mathtt{I}''$ $\vee\ \tau =$ "$L''))$
              $\theta(\gamma_1, \sigma_3)$ where
                $\theta = \gamma.[\&throw]$ and
                $\sigma_3, r_3, \tau_3 = \sigma.mkException(ArithmeticException)$ and
                $\gamma_1 = \gamma[\&thrown \leftarrow (r_3, \tau_3)]$
            else
              let $d = \mathbf{cast}(\tau_1,(\mathbf{promote}(\tau, (r_1, \tau_1)) /_\tau \mathbf{promote}(\tau, (r_2, \tau_2)),\tau)$ in
              $(\kappa(d, \tau_1, \sigma_2[l \leftarrow d])$

endif
else if (AssignOp == '<<=')
$\mathcal{L}[\![$<LHS>$]\!]\gamma\alpha\sigma$ where
$\forall r_1, \tau_1, l, \sigma_1.\ \alpha(r_2, \tau_1, l, \sigma_1) = \mathcal{E}[\![$<AssignExpr>$]\!]\gamma\kappa_2\sigma_1$ where
$\forall r_2, \tau_2, \sigma_2.\ \kappa_2(r_2, \tau_2, \sigma_2) =$
let $\tau_1' = \mathbf{unaryPromoteType}(\tau_1)$ and
$\tau_2' = \mathbf{unaryPromoteType}(\tau_2)$ and
$r_1' = \mathbf{promote}(\tau_1', (r_1, \tau_1))$ and
$r_2' = \mathbf{promote}(\tau_2', (r_2, \tau_2))$ and
$d = (\mathbf{leftShift}((r_1', \tau_1'), (r_2', \tau_2'))$ in
$\kappa(d, \tau_1, \sigma_2[l \leftarrow d])$

similar for >>=, >>>=
where the meaning of op_τ is defined in the section on numeric expressions

else if (AssignOp == '=')
$\mathcal{L}[\![$<LHS>$]\!]\gamma\alpha\sigma$ where
$\forall r_1, \tau_1, l, \sigma_1.\ \alpha(r_1, \tau_1, l, \sigma_1) = \mathcal{E}[\![$<AssignExpr>$]\!]\gamma\kappa_2\sigma_1$ where
$\forall r_2, \tau_2, \sigma_2.\ \kappa_2(r_2, \tau_2, \sigma_2) =$
if (r_1 == null)
$\theta(\gamma_1, \sigma_3)$ where
$\theta = \gamma.[\&throw]$ and
$\sigma_3, r_3, \tau_3 = \sigma.mkException(NullPointerException)$ and
$\gamma_1 = \gamma[\&thrown \leftarrow (r_3, \tau_3)]$
else if (r_1 == OutOfBounds)
$\theta(\gamma_1, \sigma_3)$ where
$\theta = \gamma.[\&throw]$ and
$\sigma_3, r_3, \tau_3 = \sigma.mkException(IndexOutOfBoundsException)$ and
$\gamma_1 = \gamma[\&thrown \leftarrow (r_3, \tau_3)]$
else if **not** ($\tau_2 <_T \tau_1$) then
$\theta(\gamma_1, \sigma_3)$ where
$\theta = \gamma.[\&throw]$ and
$\sigma_3, r_3, \tau_3 = \sigma.mkException(ArrayStoreException)$ and
$\gamma_1 = \gamma[\&thrown \leftarrow (r_3, \tau_3)]$
else
$\kappa(r_2, \tau_1, \sigma_2[l \leftarrow \mathbf{promote}(\tau_1, (r_2, \tau_2))])$
endif
endif

<AssignOp> ::=
=
| * =
| / =
| % =
| + =
| − =
| <<=
| >>=
| >>>=
| & =

$|\ \hat{}\ =$
$|\ |\ =$

$\mathcal{L}[\![$<LHS>$]\!]\gamma\alpha\sigma ::=$
$\quad \mathcal{L}[\![$<Name>$]\!]\gamma\alpha\sigma$
$\quad | \ \mathcal{L}[\![$<FieldAccess>$]\!]\gamma\alpha\sigma$
$\quad | \ \mathcal{L}[\![$<ArrayAccess>$]\!]\gamma\alpha\sigma$

Conditional Expressions. The conditional expressions (operator: `?`) of the Java language are the only expressions that do not guarantee that all subexpressions are evaluated. The regular conditional expression, is a choice operation that executes the second or third subexpression based on the boolean result of the first subexpression. The return type of the expression is based on the type of the two possible resultant subexpressions. The conditional-or expression (operator ||) and the conditional-and expression (operator &&) are *short-circuit* boolean expressions that only evaluate the second subexpression if the result of the first subexpression does not determine the result of the expression (i.e., short circuits on **true** for or and **false** for and).

$\mathcal{E}[\![$<CondExpr>$]\!]\gamma\kappa\sigma ::=$
$\quad \mathcal{E}[\![$<CondOrExpr>$]\!]\gamma\kappa\sigma$
$\quad | \ \mathcal{E}[\![$<CondOrExpr> **?** <Expr> **:** <CondExpr$_1$>$]\!]\gamma\kappa\sigma =$
$\qquad \mathcal{E}[\![$<CondOrExpr>$]\!]\gamma\kappa_1\sigma$ where
$\qquad\quad \forall r_1, \tau_1, \sigma_1.\ \kappa_1(r, \tau, \sigma_1) =$
$\qquad\qquad$ if $(r_1 ==$ **true**$)$
$\qquad\qquad\quad \mathcal{E}[\![$<Expr>$]\!]\gamma\kappa_2\sigma_1$
$\qquad\qquad$ else
$\qquad\qquad\quad \mathcal{E}[\![$<CondOrExpr$_1$>$]\!]\gamma\kappa_2\sigma_1$
$\qquad\qquad$ endif
$\qquad\quad \forall r_2, \tau_2, \sigma_2.\ \kappa_2(r_2, \tau_2, \sigma_2) =$
$\qquad\qquad \kappa($**promote**$(\tau, (r_2, \tau_2)), \tau, \sigma_2)$ where
$\qquad\qquad\quad \tau = env.condTypeOf(\tau_t, v_t, \tau_f, v_f)$ and
$\qquad\qquad\quad v_t =$ compile-time value of <Expr> and
$\qquad\qquad\quad \tau_t =$ type of <Expr> and
$\qquad\qquad\quad \tau_f =$ type of <CondOrExpr$_1$> and
$\qquad\qquad\quad v_f =$ compile-time value of <CondOrExpr$_1$>

we compute the result of $r_1 <_\tau r_2$ using the following:

$\gamma.condTypeOf(\tau_t, v_t, \tau_f, v_f) =$
$\quad$ if $(\tau_t == \tau_f)$
$\qquad \tau_t$
$\quad$ else if (isNumeric(τ_t) and isNumeric(τ_f))
$\qquad$ if $((\tau_t ==$ "`B`"and $\tau_f ==$ "`S`") or
$\qquad\qquad ((\tau_f ==$ "`B`"and $\tau_t ==$ "`S`"))
$\qquad\quad$ "`S`"
$\qquad$ else if $(\tau_t \in$ ["`B`", "`S`","`C`"] and $v_f \in \tau_t)$
$\qquad\quad \tau_t$
$\qquad$ else if $(\tau_f \in$ ["`B`", "`S`", "`C`"] and $v_t \in \tau_f)$

τ_f
else
binaryPromotionType(τ_t, τ_f)
end if
else if $(\gamma.assnCompatible(\tau_t, \tau_f))$
τ_f
else
τ_t
endif

$\mathcal{E}[\![$<CondOrExpr>$]\!]\gamma\kappa\sigma$::=
$\mathcal{E}[\![$<CondAndExpr>$]\!]\gamma\kappa\sigma$
| $\mathcal{E}[\![$<CondOrExpr$_1$> || <CondAndExpr>$]\!]\gamma\kappa\sigma$ =
$\mathcal{E}[\![$<CondOrExpr$_1$>$]\!]\gamma\kappa_1\sigma$ where
$\forall r, \tau, \sigma_1.\ \kappa_1(r, \tau, \sigma_1)$ =
if $(r ==$ **true**$)$
$\kappa(r, \tau, \sigma_1)$
else
$\mathcal{E}[\![$<CondAndExpr>$]\!]\gamma\kappa\sigma_1$
endif

$\mathcal{E}[\![$<CondAndExpr>$]\!]\gamma\kappa\sigma$::=
$\mathcal{E}[\![$<IncOrExpr>$]\!]\gamma\kappa\sigma$
| $\mathcal{E}[\![$<CondAndExpr$_1$> **&&** <IncOrExpr>$]\!]\gamma\kappa\sigma$ =
$\mathcal{E}[\![$<CondAndExpr$_1$>$]\!]\gamma\kappa_1\sigma$ where
$\forall r, \tau, \sigma_1.\ \kappa_1(r, \tau, \sigma_1)$ =
if $(r ==$ **true**$)$
$\kappa(r, \tau, \sigma_1)$
else
$\mathcal{E}[\![$<IncOrExpr>$]\!]\gamma\kappa\sigma_1$
endif

Bitwise and Boolean Expressions . The following expression all return boolean results, with the exception of the first three (and, or and xor) which perform bitwise operations on integral operands and logical operations on boolean operands. The comparison expressions can work with any operands of compatible types and thus require a more extensive definition. The results of these operations are rather complex for floating point values, and have been defined in tables to simplify the presentation. Shift and comparison operations that a similar have been removed from this presentation for space consideration.

$\mathcal{E}[\![$<IncOrExpr>$]\!]\gamma\kappa\sigma$::=
$\mathcal{E}[\![$<XORExpr>$]\!]\gamma\kappa\sigma$
| $\mathcal{E}[\![$<IncOrExpr$_1$> | <XORExpr>$]\!]\gamma\kappa\sigma = \mathcal{E}[\![$<IncOrExpr$_1$>$]\!]\gamma\kappa_1\sigma$ where
$\forall r_1, \tau_1, \sigma_1.\ \kappa_1(r_1, \tau_1, \sigma_1) = \mathcal{E}[\![$<XORExpr>$]\!]\gamma\kappa_2\sigma_1$ where
$\forall r_2, \tau_2, \sigma_2.\ \kappa_2(r_2, \tau_2, \sigma_1) = \kappa(r_1\ \text{or}_\perp\ r_2, \tau, \sigma_2)$ where
if $(\tau_1 ==$ "Z"$)$
$\tau =$ "Z"
else

τ = **binaryPromoteType**(τ_1, τ_2)
endif

$\mathcal{E}[\![$<XORExpr>$]\!]\gamma\kappa\sigma$::=
$\mathcal{E}[\![$<AndExpr>$]\!]\gamma\kappa\sigma$
| $\mathcal{E}[\![$<XORExpr$_1$> ^ <AndExpr>$]\!]\gamma\kappa\sigma = \mathcal{E}[\![$<XORExpr$_1$>$]\!]\gamma\kappa_1\sigma$ where
$\forall r_1, \tau_1, \sigma_1.\ \kappa_1(r_1, \tau_1, \sigma_1) = \mathcal{E}[\![$<AndExpr>$]\!]\gamma\kappa_2\sigma_1$ where
$\forall r_2, \tau_2, \sigma_2.\ \kappa_2(r_2, \tau_2, \sigma_1) = \kappa(r_1\ \text{xor}_\perp\ r_2, \tau, \sigma_2)$ where
if (τ_1 == "Z")
τ = "Z"
else
τ = **binaryPromoteType**(τ_1, τ_2)
endif

$\mathcal{E}[\![$<AndExpr>$]\!]\gamma\kappa\sigma$::=
$\mathcal{E}[\![$<EqualExpr>$]\!]\gamma\kappa\sigma$
| $\mathcal{E}[\![$<AndExpr$_1$> & <EqualExpr>$]\!]\gamma\kappa\sigma = \mathcal{E}[\![$<AndExpr$_1$>$]\!]\gamma\kappa_1\sigma$ where
$\forall r_1, \tau_1, \sigma_1.\ \kappa_1(r_1, \tau_1, \sigma_1) = \mathcal{E}[\![$<EqualExpr>$]\!]\gamma\kappa_2\sigma_1$ where
$\forall r_2, \tau_2, \sigma_2.\ \kappa_2(r_2, \tau_2, \sigma_1) = \kappa(r_1\ \text{and}_\perp\ r_2, \tau, \sigma_2)$ where
if (τ_1 == "Z")
τ = "Z"
else
τ = **binaryPromoteType**(τ_1, τ_2)
endif

$\mathcal{E}[\![$<EqualExpr>$]\!]\gamma\kappa\sigma$ ::=
$\mathcal{E}[\![$<RelatExpr>$]\!]\gamma\kappa\sigma$
| $\mathcal{E}[\![$<EqualExpr$_1$> == <RelatExpr>$]\!]\gamma\kappa\sigma$ =
$\mathcal{E}[\![$<EqualExpr$_1$>$]\!]\gamma\kappa_1\sigma$ where
$\forall r_1, \tau_1, \sigma_1.\ \kappa_1(r_1, \tau_1, \sigma_1) = \mathcal{E}[\![$<RelatExpr>$]\!]\gamma\kappa_2\sigma_1$ where
$\forall r_2, \tau_2, \sigma_2.\ \kappa_2(r_2, \tau_2, \sigma_1) = \kappa(q, \textbf{boolean}, \sigma_2)$ where
if (**isNumeric**(τ_1))
let (τ = **binaryPromoteType**(τ_1, τ_2)) and
r_1' = **promote**($\tau, (r_1, \tau_1)$) and
r_2' = **promote**($\tau, (r_2, \tau_2)$) in
if (τ = "F"or τ = "D")
if (r_1' == **NAN** or r_2' == **NAN**)
t = **false**
else if ($|\ r_1'\ | == 0$ and $|\ r_2'\ | == 0$)
t = **true**
else if (r_1' == r_2')
t = **true**
endif
else if (r_1' == r_2')
t = **true**
endif
else if (τ == "Z")
$t = (r_1 == r_2)$
else // must be ref types
endif

$\quad \mid \mathcal{E}[\![\text{<EqualExpr}_1\text{> != <RelatExpr>}]\!]\gamma\kappa\sigma =$
$\quad\quad \mathcal{E}[\![\textbf{!(}\text{<EqualExpr}_1\text{> == <RelatExpr>}\textbf{)}]\!]\gamma\kappa\sigma$

$\mathcal{E}[\![\text{<RelatExpr>}]\!]\gamma\kappa\sigma ::=$
$\quad \mathcal{E}[\![\text{<ShiftExpr>}]\!]\gamma\kappa\sigma$
$\quad \mid \mathcal{E}[\![\text{<RelatExpr}_1\text{> < <ShiftExpr>}]\!]\gamma\kappa\sigma = \mathcal{E}[\![\text{<RelatExpr}_1\text{>}]\!]\gamma\kappa_1\sigma$ where
$\quad\quad \forall r_1, \tau_1, \sigma_1.\ \kappa_1(r_1, \tau_1, \sigma_1) = \mathcal{E}[\![\text{<ShiftExpr>}]\!]\gamma\kappa_2\sigma_1$ where
$\quad\quad\quad \forall r_2, \tau_2, \sigma_2.\ \kappa_2(r_2, \tau_2, \sigma_1) = \kappa(q, \text{“}\mathtt{Z}''\text{”}, \sigma_2)$ where
$\quad\quad\quad\quad$ let $(\tau = \textbf{binaryPromoteType}(\tau_1, \tau_2))$ and
$\quad\quad\quad\quad\quad r_1' = \textbf{promote}(\tau, (r_1, \tau_1))$ and
$\quad\quad\quad\quad\quad r_2' = \textbf{promote}(\tau, (r_2, \tau_2))$ in
$\quad\quad\quad\quad q = r_1' <_\tau r_2'$
$\quad\quad$ similar for >, <= and >=,
$\quad\quad$ with the understanding that positive and negative 0 are equal.
$\quad \mid \mathcal{E}[\![\text{<RelatExpr}_1\text{> }\textbf{instanceof}\text{ <RefType>}]\!]\gamma\kappa\sigma = \mathcal{E}[\![\text{<RelatExpr}_1\text{>}]\!]\gamma\kappa_1\sigma_1$ where
$\quad\quad \forall r_1, \tau_1, \sigma_1.\ \kappa_1(r_1, \tau_1, \sigma_1) =$
$\quad\quad\quad \mathcal{E}[\![\text{<RefType>}]\!]\gamma\kappa_2\sigma_1$ where
$\quad\quad\quad\quad \forall r_2, \tau_2, \sigma_2.\ \kappa_2(r_2, \tau_2, \sigma_1) = \kappa(\gamma.instanceof(\tau_1, \tau_2), \text{“}\mathtt{Z}''\text{”}, \sigma_2)$

we compute the result of $r_1 <_\tau r_2$ using the following table

Computation of $r_1 <_\tau r_2$

		r_1				
		NAN	∞	$-\infty$	$\mid 0 \mid$	other
r_2	**NAN**	**false**	**false**	**false**	**false**	**false**
	∞	**false**	**false**	**false**	**false**	**false**
	$-\infty$	**false**	**true**	**false**	**true**	**true**
	$\mid 0 \mid$	**false**	**true**	**false**	**false**	$r_1 <_{\tau\perp} r_2$
	other	**false**	**true**	**false**	$r_1 <_{\tau\perp} r_2$	$r_1 <_{\tau\perp} r_2$

where $+_{\tau\perp}$ is normal addition
(using either IEEE 754, or twos complement arithmetic)
32 or 64 bit computation is based on the value of τ
this is a strict extension of normal addition
IEEE underflow or overflow returns an ∞ or a 0 value
twos complement overflow or underflow returns the low order bits of the result

Numeric Expressions. The numeric expressions take numeric operands and produce numeric results. Again, the use of floating point values greatly complicates the specification of operations such as addition and multiplication, and thus are defined in tables to simplify the presentation. We also define subtraction in terms of addition. There is an oversight in the JLS [1] involving multiplication of infinity values. Consistent with the JDK we define $\infty * \infty == \infty$ and $-\infty * \infty == \infty * -\infty == -\infty$.

$\mathcal{E}[\![\text{<ShiftExpr> }]\!]\gamma\kappa\sigma ::=$
$\quad \mathcal{E}[\![\text{<AddExpr>}]\!]\gamma\kappa\sigma$
$\quad \mid \mathcal{E}[\![\text{<ShiftExpr}_1\text{> << <AddExpr>}]\!]\gamma\kappa\sigma = \mathcal{E}[\![\text{<ShiftExpr}_1\text{>}]\!]\gamma\kappa_1\sigma$ where
$\quad\quad \forall r_1, \tau_1, \sigma_1.\ \kappa_1(r_1, \tau_1, \sigma_1) = \mathcal{E}[\![\text{<AddExpr>}]\!]\gamma\kappa_2\sigma_1$ where

$\forall r_2, \tau_2, \sigma_2.\ \kappa_2(r_2, \tau_2, \sigma_1) = \kappa(q, \tau_1', \sigma_2)$ where
let $\tau_1' =$ **unaryPromoteType**(τ_1) and
$\tau_2' =$ **unaryPromoteType**(τ_2) and
$r_1' =$ **promote**$(\tau_1', (r_1, \tau_1))$ and
$r_2' =$ **promote**$(\tau_2', (r_2, \tau_2))$ in
$q =$ **leftShift**$((r_1', \tau_1'), (r_2', \tau_2'))$
similar for >> and >>>

$\mathcal{E}[\![$<AddExpr> $]\!]\gamma\kappa\sigma ::=$
$\mathcal{E}[\![$<MultExpr>$]\!]\gamma\kappa\sigma$
| $\mathcal{E}[\![$<AddExpr$_1$> + <MultExpr>$]\!]\gamma\kappa\sigma = \mathcal{E}[\![$<AddExpr>$]\!]\gamma\kappa_1\sigma$ where
$\forall r_1, \tau_1, \sigma_1.\ \kappa_1(r_1, \tau_1, \sigma_1) = \mathcal{E}[\![$<MultExpr>$]\!]\gamma\kappa_2\sigma_1$ where
$\forall r_2, \tau_2, \sigma_2.\ \kappa_2(r_2, \tau_2, \sigma_1) = \kappa(q, \tau, \sigma_2)$ where
if (τ_1 == "Ljava.lang.String;" or τ_2 == "Ljava.lang.String;")
$\tau =$ "Ljava.lang.String;" and
$q = r_1 +_\tau r_2$
else
$\tau =$ **binaryPromoteType** (τ_1, τ_2) and
let $r_1' =$ **promote**$(\tau, (r_1, \tau_1))$ and
$r_2' =$ **promote**$(\tau, (r_2, \tau_2))$ in
$q = (r_1' +_\tau r_2')$
endif
| $\mathcal{E}[\![$<AddExpr> - <MultExpr>$]\!]\gamma\kappa\sigma = \mathcal{E}[\![$<AddExpr>$]\!]\gamma\kappa_1\sigma$ where
$\forall r_1, \tau_1, \sigma_1.\ \kappa_1(r_1, \tau_1, \sigma_1) = \mathcal{E}[\![$<MultExpr>$]\!]\gamma\kappa_2\sigma_1$ where
$\forall r_2, \tau_2, \sigma_2.\ \kappa_2(r_2, \tau_2, \sigma_1) = \kappa(q, \tau, \sigma_2)$ where
$\tau =$ **binaryPromoteType** (τ_1, τ_2) and
let $r_1' =$ **promote**$(\tau, (r_1, \tau_1))$ and
$r_2' =$ **promote**$(\tau, (r_2, \tau_2))$ in
$q = ((r_1', \tau_1) +_\tau (-r_2', \tau_2))$

where we define $(r_1, \tau_1) +_\tau (r_2, \tau_2) =$
if (τ == "Ljava.lang.String;")
String(r_1, τ_1) + **String**(r_2, τ_2)
else
compute the result using the following table
endif

Computation of $r_1 +_\tau r_2$

		r_1					
		NAN	∞	$-\infty$	0	-0	other
r_2	**NAN**	**NAN**	**NAN**	**NAN**	**NAN**	**NAN**	**NAN**
	∞	**NAN**	∞	**NAN**	∞	∞	∞
	$-\infty$	**NAN**	**NAN**	$-\infty$	$-\infty$	$-\infty$	$-\infty$
	0	**NAN**	∞	$-\infty$	0	0	r_1
	-0	**NAN**	∞	$-\infty$	0	-0	r_1
	other	**NAN**	∞	$-\infty$	r_2	r_2	$r_1 +_{\tau\perp} r_2$

where $+_{\tau\perp}$ is normal addition
(using either IEEE 754, or twos complement arithmetic)

32 or 64 bit computation is based on the value of τ
this is a strict extension of normal addition
IEEE underflow or overflow returns an ∞ or a 0 value
twos complement overflow or underflow returns the low order bits of the result

$\mathcal{E}[\![$<MultExpr>$]\!]\gamma\kappa\sigma$ ::=
  $\mathcal{E}[\![$<UnaryExpr>$]\!]\gamma\kappa\sigma$
 | $\mathcal{E}[\![$<MultExpr$_1$> ***** <UnaryExpr>$]\!]\gamma\kappa\sigma = \mathcal{E}[\![$<MultExpr$_1$>$]\!]\gamma\kappa_1\sigma$ where
    $\forall r_1, \tau_1, \sigma_1.\ \kappa_1(r_1, \tau_1, \sigma_1) = \mathcal{E}[\![$<UnaryExpr>$]\!]\gamma\kappa_2\sigma_1$ where
      $\forall r_2, \tau_2, \sigma_2.\ \kappa_2(r_2, \tau_2, \sigma_1) = \kappa(q, \tau, \sigma_2)$ where
      $\tau =$ **binaryPromoteType** (τ_1, τ_2) and
        let $r_1' =$ **promote**$(\tau, (r_1, \tau_1))$ and
          $r_2' =$ **promote**$(\tau, (r_2, \tau_2))$ in
        $q = (r_1' *_\tau r_2')$
 | $\mathcal{E}[\![$<MultExpr$_1$> / <UnaryExpr>$]\!]\gamma\kappa\sigma = \mathcal{E}[\![$<MultExpr$_1$>$]\!]\gamma\kappa_1\sigma$ where
    $\forall r_1, \tau_1, \sigma_1.\ \kappa_1(r_1, \tau_1, \sigma_1) = \mathcal{E}[\![$<UnaryExpr>$]\!]\gamma\kappa_2\sigma_1$ where
      $\forall r_2, \tau_2, \sigma_2.\ \kappa_2(r_2, \tau_2, \sigma_1) = \kappa(q, \tau, \sigma_2)$ where
      $\tau =$ **binaryPromoteType** (τ_1, τ_2) and
        let $r_1' =$ **promote**$(\tau, (r_1, \tau_1))$ and
          $r_2' =$ **promote**$(\tau, (r_2, \tau_2))$ in
        if $(|\ r_2'\ | == 0)$
          $\theta(\gamma_1, \sigma_3)$ where
            $\theta = \gamma.[\&\mathit{throw}]$ and
            $\sigma_3, r_3, \tau_3 = \sigma.\mathit{mkException}(\mathit{ArithmeticException})$ and
            $\gamma_1 = \gamma[\&\mathit{thrown} \leftarrow (r_3, \tau_3)]$
        else
          $q = (r_1'/_\tau r_2')$
        endif
 | $\mathcal{E}[\![$<MultExpr$_1$> **%** <UnaryExpr>$]\!]\gamma\kappa\sigma = \mathcal{E}[\![$<MultExpr$_1$>$]\!]\gamma\kappa_1\sigma$ where
    $\forall r_1, \tau_1, \sigma_1.\ \kappa_1(r_1, \tau_1, \sigma_1) = \mathcal{E}[\![$<UnaryExpr>$]\!]\gamma\kappa_2\sigma_1$ where
      $\forall r_2, \tau_2, \sigma_2.\ \kappa_2(r_2, \tau_2, \sigma_1) = \kappa(q, \tau, \sigma_2)$ where
      $\tau =$ **binaryPromoteType** (τ_1, τ_2) and
        let $r_1' =$ **promote**$(\tau, (r_1, \tau_1))$ and
          $r_2' =$ **promote**$(\tau, (r_2, \tau_2))$ in
        $q = (r_1' \%_\tau r_2')$

where we compute $r_1 *_\tau r_2$ using the following table

Computation of $r_1 *_\tau r_2$

		r_1					
		NAN	∞	$-\infty$	0	-0	other
r_2	**NAN**	**NAN**	**NAN**	**NAN**	**NAN**	**NAN**	**NAN**
	∞	**NAN**	∞	$-\infty$	**NAN**	**NAN**	$(s)\infty$
	$-\infty$	**NAN**	$-\infty$	∞	**NAN**	**NAN**	$(s)\infty$
	0	**NAN**	**NAN**	**NAN**	0	-0	$(s)0$
	-0	**NAN**	**NAN**	**NAN**	-0	0	$(s)0$
	other	**NAN**	$(s)\infty$	$(s)\infty$	$(s)0$	$(s)0$	$r_1 *_{\tau\perp} r_2$

where $*_{\tau\perp}$ is normal multiplication

(using either IEEE 754, or twos complement arithmetic)
32 or 64 bit computation is based on the value of τ
this is a strict extension of normal addition
IEEE underflow or overflow returns an ∞ or a 0 value
twos complement overflow or underflow returns the low order bits of the result
(s) represents the sign of the result which is positive if both
r_1 and r_2 have the same sign and negative otherwise

where we compute $r_1 /_\tau r_2$ using the following table

Computation of $r_1 /_\tau r_2$

		r_1					
		NAN	∞	$-\infty$	0	-0	other
r_2	**NAN**	**NAN**	**NAN**	**NAN**	**NAN**	**NAN**	**NAN**
	∞	**NAN**	**NAN**	**NAN**	0	-0	$(s)0$
	$-\infty$	**NAN**	**NAN**	**NAN**	-0	0	$(s)0$
	0	**NAN**	∞	$-\infty$	**NAN**	**NAN**	$(s)\infty$
	-0	**NAN**	$-\infty$	∞	**NAN**	**NAN**	$(s)\infty$
	other	**NAN**	$(s)\infty$	$(s)\infty$	$(s)0$	$(s)0$	$r_1 /_{\tau\perp} r_2$

where $/_{\tau\perp}$ is normal division
(using either IEEE 754, or twos complement arithmetic)
32 or 64 bit computation is based on the value of τ
this is a strict extension of normal addition
IEEE underflow or overflow returns an ∞ or a 0 value
twos complement overflow or underflow returns the low order bits of the result
(s) represents the sign of the result which is positive if both
r_1 and r_2 have the same sign and negative otherwise

where we compute $r_1 \%_\tau r_2$ using the following table

Computation of $r_1 \%_\tau r_2$

		r_1					
		NAN	∞	$-\infty$	0	-0	other
r_2	**NAN**	**NAN**	**NAN**	**NAN**	**NAN**	**NAN**	**NAN**
	∞	**NAN**	**NAN**	**NAN**	0	-0	r_1
	$-\infty$	**NAN**	**NAN**	**NAN**	0	-0	r_1
	0	**NAN**	**NAN**	**NAN**	**NAN**	**NAN**	**NAN**
	-0	**NAN**	**NAN**	**NAN**	**NAN**	**NAN**	**NAN**
	other	**NAN**	**NAN**	**NAN**	0	-0	$r_1 \%_{\tau\perp} r_2$

where $\%_{\tau\perp}$ is integer division
(using C/C++ style remainder, or twos complement arithmetic)
Does NOT follow IEEE 754 remainder operation,
rather C/C++ style integer remainder operation
Floating point underflow or overflow returns an ∞ or a 0 value
twos complement overflow or underflow returns the low order bits of the result
(s) represents the sign of the result which is positive if both
r_1 and r_2 have the same sign and negative otherwise

4.14 Location Expressions

All Java expressions return either a value, variable or void (for method invocations that return no value). Unary and primary expressions are the only expressions that can return a variable (location in a store). As such, we use the location semantic function to evaluate unary and primary expressions. However, to maintain consistency in the grammar, we have included regular expression productions and semantics interleaved with the location expressions.

Unary Expressions. Unary expressions involve changing the sign or type of an expression, or incrementing or decrementing a value. In the case or pre or post increment or decrement operations the expression has a definite side-effect on the store, as is indicated in the semantics. Note that the return value of the expression indicates the pre or post nature of the expression. If the unary (or primary) expression does not return a variable, then the value *undef* is returned.

$$\mathcal{E}[\![\text{<UnaryExpr>}]\!]\gamma\kappa\sigma = \mathcal{L}[\![\text{<UnaryExpr>}]\!]\gamma\alpha\sigma \text{ where}$$
$$\quad \forall r, \tau, l, \sigma_1.\ \alpha(r, \tau, l, \sigma_1) = \kappa(r, \tau, \sigma_1)$$

$$\mathcal{L}[\![\text{<UnaryExp>}]\!]\gamma\alpha\sigma ::=$$
$$\quad \mathcal{L}[\![\text{<PreIncExpr>}]\!]\gamma\kappa\sigma$$
$$\mid \mathcal{L}[\![\text{<PreDecExpr>}]\!]\gamma\kappa\sigma$$
$$\mid \mathcal{L}[\![\text{<UnaryExprNotPlusMinus>}]\!]\gamma\kappa\sigma$$
$$\mid \mathcal{L}[\![\text{+ <UnaryExpr}_1\text{>}]\!]\gamma\alpha\sigma = \mathcal{L}[\![\text{<UnaryExpr}_1\text{>}]\!]\gamma\alpha_1\sigma \text{ where}$$
$$\quad \forall r, \tau, l, \sigma_1.\ \alpha_1(r, \tau, l, \sigma_1) = \alpha(r, \tau, \mathit{undef}, \sigma_1)$$
$$\mid \mathcal{L}[\![\text{- <UnaryExpr}_1\text{>}]\!]\gamma\alpha\sigma = \mathcal{L}[\![\text{<UnaryExpr}_1\text{>}]\!]\gamma\alpha_1\sigma \text{ where}$$
$$\quad \forall r, \tau, l, \sigma_1.\ \alpha_1(r, \tau, l, \sigma_1) = \alpha(0 -_{\tau\perp} r, \tau, \mathit{undef}, \sigma_1)$$

$$\mathcal{L}[\![\text{<UnaryExprNotPlusMinus>}]\!]\gamma\kappa\sigma ::=$$
$$\quad \mathcal{L}[\![\text{<PostExpr>}]\!]\gamma\kappa\sigma$$
$$\mid \mathcal{L}[\![\text{<CastExpr>}]\!]\gamma\kappa\sigma$$
$$\mid \mathcal{E}[\![\sim \text{<UnaryExpr>}]\!]\gamma\kappa\sigma = \mathcal{L}[\![\text{<UnaryExpr>}]\!]\gamma\alpha\sigma \text{ where}$$
$$\quad \forall r, \tau, l, \sigma_1.\ \alpha(r, \tau, l, \sigma_1) =$$
$$\qquad \text{let } \tau_1 = \mathbf{unaryPromoteType}(\tau) \text{ and}$$
$$\qquad\quad r_1 = \mathbf{promote}(\tau_1, (r, \tau)) \text{ in}$$
$$\qquad \kappa((-r_1) - 1, \tau, \sigma_1)$$
$$\mid \mathcal{E}[\![\mathbf{!} \text{<UnaryExpr>}]\!]\gamma\kappa\sigma = \mathcal{L}[\![\text{<UnaryExpr>}]\!]\gamma\alpha\sigma \text{ where}$$
$$\quad \forall r, \tau, l, \sigma_1.\ \alpha_1(r, \tau, l, \sigma_1) =$$
$$\qquad \text{if } (r == \mathbf{true})$$
$$\qquad\quad \kappa(\mathbf{false}, \tau, \sigma_2)$$
$$\qquad \text{else}$$
$$\qquad\quad \kappa(\mathbf{false}, \tau, \sigma_2)$$
$$\qquad \text{endif}$$

We have reverted to the non-LALR(1) grammar for cast expressions to simplify the presentation of the semantics. Specifically, the return type of the expression is the type of the cast (given that no error occurs), and the return value is the converted value of the expression.

$\mathcal{E}[\![$<CastExpr>$]\!]\gamma\kappa\sigma$ ::=
  $\mathcal{E}[\![$(<PrimType> <Dims>$^?$) <UnaryExpr>$]\!]\gamma\kappa\sigma = \mathcal{E}[\![$<UnaryExp>$]\!]\gamma\kappa_1\sigma$ where
    let $\tau' = \mathcal{T}[\![$<PrimType>$]\!]\gamma$ and
      $d = \mathbf{fst}(\mathcal{V}[\![$<Dims>$]\!]\gamma)$ and
      $\tau = \mathbf{mkArrayType}(\tau', d)$ in
    $\forall r, \tau_1, \sigma.\kappa_1(r, \tau_1, sto) = \kappa(r_1, \tau, \sigma)$ where
      $r_1 = \mathbf{cast}(\tau, (r, \tau_1))$
| $\mathcal{E}[\![$(<RefType>) <UnaryExprNotPlusMinus>$]\!]\gamma\kappa\sigma$ =
    $\mathcal{E}[\![$<UnaryExprNotPlusMinus>$]\!]\gamma\kappa_1\sigma$ where
    let $\tau = \mathcal{T}[\![$<RefType>$]\!]\gamma$ in
    $\forall r, \tau_1, \sigma.\kappa_1(r, \tau_1, sto) =$
      if (not ($env.assnCompatible(\tau, \tau_1)$ or
          $env.assnCompatible(\tau_1, \tau)$))
        $\theta(\gamma_1, \sigma_2)$ where
          $\theta = \gamma.[\&throw]$ and
          $\sigma_2, r_2, \tau_2 = \sigma.mkException(CastConversionException)$ and
          $\gamma_1 = \gamma[\&thrown \leftarrow (r_2, \tau_2)]$
      else
        $\kappa(r_1, \tau, \sigma)$ where
          $r_1 = \mathbf{cast}(\tau, (r, \tau_1))$
      endif

In the JLS [1], there is discussion that (p)++ is a valid post fix operation ("`(p)++` can make sense only as a postfix increment of `p`"). However, in the JDK, any parenthesized expression returns only a value and not a variable. We follow that convention here.

$\mathcal{L}[\![$<PostIncExpr>$]\!]\gamma\alpha\sigma$ ::=
  $\mathcal{L}[\![$<PostExpr> ++$]\!]\gamma\alpha\sigma = \mathcal{L}[\![$<PostExpr>$]\!]\gamma\alpha_1\sigma$ where
    $\forall r, \tau_1, l, \sigma_1.\ \alpha_1(r, \tau_1, l, \sigma_1) =$
      let $\tau = \mathbf{binaryPromotionType}(\tau, \text{“}\texttt{I}''$) and
        $r_1 = \mathbf{promote}(\tau, (r, \tau_1))$ and
        $r_2 = \mathbf{promote}(\tau, (1, \text{“}\texttt{I}''))$ and
        $q = \mathbf{cast}(\tau_1, (r_1 +_{\tau\perp} r_2), \tau)$ in
      $\alpha(r, \tau_1, undef, \sigma_1[l \leftarrow q])$

$\mathcal{L}[\![$<PostDecExpr>$]\!]\gamma\alpha\sigma$ ::=
  $\mathcal{L}[\![$<PostExpr> - -$]\!]\gamma\alpha\sigma = \mathcal{L}[\![$<PostExpr>$]\!]\gamma\alpha_1\sigma$ where
    $\forall l, r, \tau_1, l, \sigma_1.\ \alpha_1(r, \tau_1, l, \sigma_1) =$
      let $\tau = \mathbf{binaryPromotionType}(\tau, \text{“}\texttt{I}''$) and
        $r_1 = \mathbf{promote}(\tau, (r, \tau_1))$ and
        $r_2 = \mathbf{promote}(\tau, (-1, \text{“}\texttt{I}''))$ and
        $q = \mathbf{cast}(\tau_1, (r_1 +_{\tau\perp} r_2), \tau)$ in
      $\alpha(r, \tau_1, undef, \sigma_1[l \leftarrow q])$

$\mathcal{L}[\![$<PostExpr>$]\!]\gamma\alpha\sigma$ ::=
  $\mathcal{L}[\![$<Primary>$]\!]\gamma\alpha\sigma$
| $\mathcal{L}[\![$<Name>$]\!]\gamma\alpha\sigma$
| $\mathcal{L}[\![$<PostIncExpr>$]\!]\gamma\alpha\sigma$
| $\mathcal{L}[\![$<PostDecExpr>$]\!]\gamma\alpha\sigma$

$\mathcal{L}[\![$<PreIncExpr>$]\!]\gamma\alpha\sigma ::=$
$\quad \mathcal{L}[\![$++ <UnaryExpr>$]\!]\gamma\alpha\sigma = \mathcal{L}[\![$<UnaryExpr>$]\!]\gamma\alpha_1\sigma$ where
$\quad\quad \forall r, \tau_1, l, \sigma_1.\ \alpha_1(r, \tau_1, l, \sigma_1) =$
$\quad\quad\quad$ let $\tau = \mathbf{binaryPromotionType}(\tau, \text{“}\texttt{I}''\text{)}$ and
$\quad\quad\quad\quad r_1 = \mathbf{promote}(\tau, (r, \tau_1))$ and
$\quad\quad\quad\quad r_2 = \mathbf{promote}(\tau, (1, \text{“}\texttt{I}''\text{)}))$ and
$\quad\quad\quad\quad q = \mathbf{cast}(\tau_1, (r_1 +_{\tau\perp} r_2, \tau))$ in
$\quad\quad\quad \alpha(q, \tau_1, \mathit{undef}, \sigma_1[l \leftarrow q])$

$\mathcal{L}[\![$<PreDecExpr>$]\!]\gamma\alpha\sigma ::=$
$\quad \mathcal{L}[\![$- - <UnaryExpr>$]\!]\gamma\alpha\sigma = \mathcal{L}[\![$<UnaryExpr>$]\!]\gamma\alpha_1\sigma$ where
$\quad\quad \forall r, \tau_1, l, \sigma_1.\ \alpha_1(r, \tau_1, l, \sigma_1) =$
$\quad\quad\quad$ let $\tau = \mathbf{binaryPromotionType}(\tau, \text{“}\texttt{I}''\text{)}$ and
$\quad\quad\quad\quad r_1 = \mathbf{promote}(\tau, (r, \tau))$ and
$\quad\quad\quad\quad r_2 = \mathbf{promote}(\tau, (-1, \text{“}\texttt{I}''\text{)}))$ and
$\quad\quad\quad\quad q = \mathbf{cast}(\tau_1, (r_1 +_{\tau\perp} r_2, \tau))$ in
$\quad\quad\quad \alpha(q, \tau_1, \mathit{undef}, \sigma_1[l \leftarrow q])$

Primary Expressions. These expressions are the base expressions of the Java language providing access to variables, fields, methods, arrays and new object instances.

$\mathcal{L}[\![$<Primary>$]\!]\gamma\alpha\sigma ::=$
$\quad \mathcal{L}[\![$<PrimaryNoNewArray>$]\!]\gamma\alpha\sigma$
$\quad | \ \mathcal{L}[\![$<ArrayCreationExpr>$]\!]\gamma\alpha\sigma = \mathcal{E}[\![$<ArrayCreationExpr>$]\!]\gamma\kappa\sigma$ where
$\quad\quad \forall r, \tau, \sigma_1.\kappa(r, \tau, \sigma_1) = \alpha(r, \tau, \mathit{undef}, \sigma_1)$

$\mathcal{L}[\![$<PrimaryNoNewArray>$]\!]\gamma\alpha\sigma ::=$
$\quad \mathcal{L}[\![$<Literal>$]\!]\gamma\alpha\sigma =$
$\quad\quad$ let $(r, \tau) = \mathcal{V}[\![$<Literal>$]\!]\gamma$ in
$\quad\quad \alpha(r, \tau, \mathit{undef}, \sigma)$
$\quad | \ \mathcal{L}[\![\mathbf{this}]\!]\gamma\alpha\sigma = \alpha(\gamma[\&\mathit{thisObject}], \gamma[\&\mathit{thisClass}], \mathit{undef}, \sigma)$
$\quad | \ \mathcal{L}[\![($ <Expr> $)]\!]\gamma\alpha\sigma = \mathcal{E}[\![$<Expr>$]\!]\gamma\kappa\sigma$ where
$\quad\quad \forall r, \tau, \sigma_1.\kappa(r, \tau, \sigma_1) = \alpha(r, \tau, \mathit{undef}, \sigma_1)$
$\quad | \ \mathcal{L}[\![$<ClassInstCreationExpr>$]\!]\gamma\alpha\sigma$
$\quad | \ \mathcal{L}[\![$<FieldAccess>$]\!]\gamma\alpha\sigma$
$\quad | \ \mathcal{L}[\![$<MethodInv>$]\!]\gamma\alpha\sigma = \mathcal{E}[\![$<MethodInv>$]\!]\gamma\kappa\sigma$ where
$\quad\quad \forall r, \tau, \sigma_1.\kappa(r, \tau, \sigma_1) = \alpha(r, \tau, \mathit{undef}, \sigma_1)$
$\quad | \ \mathcal{L}[\![$<ArrayAccess>$]\!]\gamma\alpha\sigma$

Array creation expressions are responsible for the creation of a new array of values. Specifically, they allocate space in the store for the array, and then initialize all of the elements of the array based on the default initializer for the array elements. Note that if these elements are reference types, they are initialized to `null`. This expression only returns a value, the reference to the array, and not a location.

$\mathcal{E}[\![$<ArrayCreationExpr>$]\!]\gamma\kappa\sigma ::=$
$\quad \mathcal{E}[\![\mathbf{new}$ <PrimType> <DimExprList> <Dims>$^?]\!]\gamma\kappa_1\sigma =$
$\quad\quad \mathcal{E}[\![$<DimExprList>$]\!]\gamma\kappa\sigma$ where
$\quad\quad\quad \forall r, \tau, \sigma_1.\kappa_1(r, \tau, \sigma_1) = \kappa(q, \tau_1, \sigma_2)$ where

$v = \text{fst}(\mathcal{V}[\![\text{<Dims>}]\!]\gamma)$ and
$\tau_p = \mathcal{T}[\![\text{<PrimType>}]\!]\gamma$ and
$\tau_1 = \textbf{mkArrayType}(\tau, v)$ and
$(\sigma_2, q) = \sigma_1.allocateArray(\tau_1, \tau_p)$

| $\mathcal{E}[\![\textbf{new}$ <ClassInterfaceType> <DimExprList> <Dims>$^?]\!]\gamma\kappa\sigma =$
$\mathcal{E}[\![\text{<DimExprList>}]\!]\gamma\kappa_1\sigma_1$ where
$sto_1 = \gamma.classLoader(\textbf{fst}(\mathcal{V}[\![\text{<ClassInterfaceType>}]\!]),\sigma)$ and
$\forall r, \tau, \sigma_1.\kappa_1(r, \tau, \sigma_1) = \kappa(q, \tau_1, \sigma_2)$ where
$v = \text{fst}(\mathcal{V}[\![\text{<Dims>}]\!]\gamma)$ and
$\tau_p = \mathcal{T}[\![\text{<ClassInterfaceType>}]\!]\gamma$ and
$\tau_1 = \textbf{mkArrayType}(\tau, v)$ and
$(\sigma_2, q) = \sigma_1.allocateArray(\tau_1, \tau_p)$

The following semantics denote field access. Specifically, these semantics look up the named field in the enviroment and return the fields location and type.

$\mathcal{L}[\![\text{<FieldAccess>}\ ]\!]\gamma\alpha\sigma ::=$
$\mathcal{L}[\![\text{<Primary> . <Id>}]\!]\gamma\alpha\sigma = \alpha(r, \tau, l, \sigma)$ where
$r = \sigma[l]$ and
$l, \tau = \gamma[\mathcal{V}[\![\text{<Primary>. <Id>}]\!]\gamma]$
| $\mathcal{L}[\![\textbf{super}$. <Id>$]\!]\gamma\alpha\sigma = \alpha(r, \tau, l, \sigma)$ where
$r = \sigma[l]$ and
$l, \tau = \gamma[\mathcal{V}[\![\gamma[\&super]$. <Id>$]\!]\gamma]$

The following semantics denote the process of invoking a method call. We have simplified the syntax here from the JLS by just specifying a Name for the method instead of separating the primary and super constructs. The concept for this access is detailed in the auxiliary functions that search the environment. The result of the environment search and retrieval is a function that takes an environment, command continuation and a store and returns an answer. These semantics evaluate the arguments, look up the function for the specified method and execute that function.

$\mathcal{E}[\![\text{<MethodInv>}\ ]\!]\gamma\kappa\sigma ::=$
$\mathcal{E}[\![\text{<Name> (}$ <ArgList>$^?$ $)]\!]\gamma\kappa\sigma =$
$\mathcal{E}[\![\text{<ArgList>}]\!]\ \gamma\kappa_1\sigma$ where
$\forall r, \tau, \sigma_1.\ \kappa_1(r, \tau, \sigma_1) = m(\gamma\theta\sigma_1)$ where
$sig = \textbf{getSigs}(r)$ and
$m = \gamma.getMethod(\textbf{fst}\mathcal{V}[\![\text{<Name>}]\!]\gamma$,sig) and
$\forall\gamma_2, \sigma_2.\ \theta(\gamma_2, \sigma_2) = \kappa(\gamma_2[\&returnVal], \gamma_2[\&returnType], \sigma_2)$

The following semantics are used to specify the creation of an instance of a class through the invocation of a **n**ew operator. Upon invocation of this operator the class needs to be loaded (if it had not been loaded). Loading involves creation of storage space in the store, execution of field initializers for static fields of the class, and execution of static constructors for the class. After the execution of these entities, only then is the explicit constructor invoked (and its arguments evaluated). Note the inclusion of the <ClassBody> construct in the last two productions. These are new as of Java 1.1 and permit the construction of anonymous classes. For the sake of brevity, we do not include their semantics here. These semantics would be the same as the first semantics except that the execution method m would be the evaluation of the class body.

$\mathcal{E}[\![$<ClassInstCreationExpr>$]\!]\gamma\kappa\sigma$ ::=
  $\mathcal{E}[\![$**new** <ClassType> **(** <ArgList>$^?$ **)**$]\!]\gamma\kappa\sigma =$
    $\mathcal{E}[\![$<ArgList>$]\!]\gamma\kappa_1\sigma_1$ where
      $sto_1 = \gamma.classLoader($**fst**$(\mathcal{V}[\![$<ClassType>$]\!]),\sigma)$ and
        $\forall r, \tau, \sigma_2.\kappa_1(r, \tau, \sigma_2) = m(\gamma\theta\sigma_2)$ where
      $sig =$ **getSigs**(r) and
        $m = \gamma.getMethod($**fst**$\mathcal{V}[\![$<ClassType>$]\!]\gamma$,sig) and
        $\forall\gamma_2, \sigma_2.\ \theta(\gamma_2, \sigma_2) = \kappa(\gamma_2[\&returnVal], \gamma_2[\&returnType], \sigma_2)$
  | **new** <ClassType> **(** <ArgList>$^?$ **)** <ClassBody>
  | **new** <InterfaceType> **()** <ClassBody>

The array access expressions allow us to dereference an existing array and return the location of an element of the array. If that element is a reference type, then the location returned is the location that stores the references and not the location of the object itself.

$\mathcal{L}[\![$<ArrayAccess> $]\!]\gamma\alpha\sigma$ ::=
  $\mathcal{L}[\![$<Name> **[** <Expr> **]**$]\!]\gamma\alpha_1\sigma = \mathcal{E}[\![$<Expr>$]\!]\gamma\kappa\sigma$ where
    $\forall r, \tau, \sigma_1.\ \kappa(r, \tau, \sigma_1) = \alpha(q, \tau_1, l, \sigma_1)$ where
      $v =$ **fst**$(\mathcal{V}[\![$Name$]\!]\gamma)$ and
      $a = \gamma.getArrayRef(v)$ and
      $\tau' =$ **unaryPromoteType**(τ) and
      $l = \gamma.getArrayElem(a,$ **promote**$(\tau', (r, \tau)))$ and
      $\tau_1 = \gamma.getArrayElemType(a)$ and
      $q = \sigma(l)$
  | $\mathcal{L}[\![$<PrimaryNoNewArray> **[** <Expr> **]**$]\!]\gamma\alpha\sigma =$
    $\mathcal{L}[\![$<PrimaryNoNewArray>$]\!]\gamma\alpha_1\sigma$ where
      $\forall r_1, \tau_1, l_1, \sigma_1.\ \alpha r, \tau, l, \sigma_1 = \mathcal{E}[\![$<Expr>$]\!]\gamma\kappa\sigma_1$ where
        $\forall r_2, \tau_2, \sigma_2.\ \kappa r_2, \tau_2, \sigma_2 = \alpha(q, \tau_1, l, \sigma_2)$ where
          $\tau' =$ **unaryPromoteType**(τ_2) and
          $l = \gamma.getArrayElem(r_1,$ **promote**$(\tau', (r_2, \tau_2)))$ and
          $\tau_1 = \gamma.getArrayElemType(r_1)$ and
          $q = \sigma(l)$

The <ArgList> construction allows us to specify a list of expressions. The result is a list of pairs of values and types for each of the arguments in the argument list.

$\mathcal{E}[\![$<ArgList> $]\!]\gamma\kappa\sigma$ ::=
  [$\mathcal{E}[\![$<Expr>$]\!]\gamma\kappa\sigma$ **]**
  | $\mathcal{E}[\![$<ArgList$_1$> **,** <Expr>$]\!]\gamma\kappa\sigma = \mathcal{E}[\![$<ArgList$_1$>$]\!]\gamma\kappa_1\sigma$ where
    $\forall r_1, \tau_1, \sigma_1.\ \kappa_1(r_1, \tau_1, \sigma_1) = \mathcal{E}[\![$<Expr>$]\!]\gamma\kappa_2\sigma_1$ where
      $\forall r_2, \tau_2, \sigma_2.\ \kappa_2(r_2, \tau_2, \sigma_2) = \kappa(q, \tau, \sigma_2)$ where
        $q =$ **append**(r_1, r_2) and
        $\tau = \tau_1 + \tau_2$

Dims. The following three productions are used in cast expressions and array creation expressions to specify the dimensions of the created array. The information obtained from these productions is used to provide a count of the number of array indices and any specified dimension sizes.

$\mathcal{E}[\![$<DimExprList>$]\!]\gamma\kappa\sigma$ ::=
  $\mathcal{E}[\![$<DimExpr>$]\!]\gamma\kappa\sigma$
 | $\mathcal{E}[\![$<DimExprList$_1$> <DimExpr>$]\!]\gamma\kappa\sigma = \mathcal{E}[\![$<DimExprList$_1$>$]\!]\gamma\kappa_1\sigma$ where
    $\forall r_1, \tau_1, \sigma_1.\ \kappa_1(r_1, \tau_1, \sigma_1) = \mathcal{E}[\![$<DimExpr>$]\!]\gamma\kappa_2\sigma_1$ where
      $\forall r_2, \tau_2, \sigma_2.\ \kappa_2(r_2, \tau_2, \sigma_2) = \kappa(q, \tau, \sigma_2)$ where
        $q = \mathbf{append}(r_1, r_2)$ and
        $\tau = \mathbf{mkArrayType}(\tau_1,1)$

$\mathcal{E}[\![$<DimExpr>$]\!]\gamma\kappa\sigma$ ::=
  $\mathcal{E}[\![$[<Expr>]$]\!]\gamma\kappa\sigma = \mathcal{E}[\![$<Expr>$]\!]\gamma\kappa_1\sigma$ where
    $\forall r, \tau, \sigma_1.\ \kappa_1(r, \tau, \sigma) = \kappa(q, \mathbf{int}[], sto)$ where
      $\tau' = \mathbf{unaryPromoteType}(\tau)$ and
      $q = [\mathbf{promote}(\tau', (r, \tau))]$

$\mathcal{E}[\![$<Dims>$]\!]\gamma\kappa\sigma$ ::=
  $\mathcal{V}[\![$[ ]$]\!]\gamma = (1,\mathbf{int})$
 | $\mathcal{V}[\![$<Dims$_1$> [ ]$]\!]\gamma = (v + 1, \mathbf{int})$ where
    $(v, \tau) = \mathcal{V}[\![$<Dims$_1$>$]\!]\gamma$

References

1. J. Gosling, B. Joy, and G. Steele. *The Java Language Specification*. Addison-Wesley, 1996.
2. IEEE. IEEE standard for binary floating-point arithmetic, 1985.
3. T. Lindholm and F. Yellin. *The Java Virtual Machine Specification*. Addison-Wesley, 1996.
4. J.E. Stoy. *Denotational Semantics: The Scott-Strachey Approach to Programming Language Theory*. M.I.T. Press, 1977.
5. R.D. Tennent. *Principles of Programming Languages*. Prentice-Hall, 1981.

A Programmer's Reduction Semantics for Classes and Mixins

Matthew Flatt, Shriram Krishnamurthi, and Matthias Felleisen

Rice University

Abstract. While class-based object-oriented programming languages provide a flexible mechanism for re-using and managing related pieces of code, they typically lack linguistic facilities for specifying a uniform extension of many classes with one set of fields and methods. As a result, programmers are unable to express certain abstractions over classes. In this paper we develop a model of class-to-class functions that we refer to as *mixins*. A mixin function maps a class to an extended class by adding or overriding fields and methods. Programming with mixins is similar to programming with single inheritance classes, but mixins more directly encourage programming to interfaces. The paper develops these ideas within the context of Java. The results are

1. an intuitive model of an essential Java subset;
2. an extension that explains and models mixins; and
3. type soundness theorems for these languages.

1 Organizing Programs with Functions and Classes

Object-oriented programming languages offer classes, inheritance, and overriding to parameterize over program pieces for management purposes and re-use. Functional programming languages provide various flavors of functional abstractions for the same purpose. The latter model was developed from a well-known, highly developed mathematical theory. The former grew in response to the need to manage large programs and to re-use as many components as possible.

Each form of parameterization is useful for certain situations. With higher-order functions, a programmer can easily define many functions that share a similar core but differ in a few details. As many language designers and programmers readily acknowledge, however, the functional approach to parameterization is best used in situations with a relatively small number of parameters. When a function must consume a large number of arguments, the approach quickly becomes unwieldy, especially if many of the arguments are the same for most of the function's uses.[1]

Class systems provide a simple and flexible mechanism for managing collections of highly parameterized program pieces. Using class extension (inheritance) and overriding, a programmer derives a new class by specifying only the

[1] Function entry points *à la* Fortran or keyword arguments *à la* Common Lisp are a symptom of this problem, not a remedy.

Jim Alves-Foss (Ed.): Formal Syntax and Semantics of Java, LNCS 1523, pp. 241–269, 1999.

elements that change in the derived class. Nevertheless, a pure class-based approach suffers from a lack of abstractions that specify uniform extensions and modifications of classes. For example, the construction of a programming environment may require many kinds of text editor frames, including frames that can contain multiple text buffers and frames that support searching. In Java, for example, we cannot implement all combinations of multiple-buffer and searchable frames using derived classes. If we choose to define a class for all multiple-buffer frames, there can be no class that includes only searchable frames. Hence, we must repeat the code that connects a frame to the search engine in at least two branches of the class hierarchy: once for single-buffer searchable frames and again for multiple-buffer searchable frames. If we could instead specify a mapping from editor frame classes to searchable editor frame classes, then the code connecting a frame to the search engine could be abstracted and maintained separately.

Some class-based object-oriented programming languages provide multiple inheritance, which permits a programmer to create a class by extending more than one class at once. A programmer who also follows a particular protocol for such extensions can mimic the use of class-to-class functions. Common Lisp programmers refer to this protocol as *mixin programming* [20,21], because it roughly corresponds to mixing in additional ingredients during class creation. Bracha and Cook [6] designed a language of class manipulators that promote mixin thinking in this style and permit programmers to build mixin-like classes. Unfortunately, multiple inheritance and its cousins are semantically complex and difficult to understand for programmers.[2] As a result, implementing a mixin protocol with these approaches is error-prone and typically avoided.

For the design of MzScheme's class and interface system [15], we experimented with a different approach. In MzScheme, classes form a single inheritance hierarchy, but are also first-class values that can be created and extended at run-time. Once this capability was available, the programmers of our team used it extensively for the construction of DrScheme [14], a Scheme programming environment. However, a thorough analysis reveals that the code only contains first-order functions on classes.

In this paper, we present a typed model of such "class functors" for Java [17]. We refer to the functors as *mixins* due to their similarity to Common Lisp's multiple inheritance mechanism and Bracha's class operators. Our proposal is superior in that it isolates the useful aspects of multiple inheritance yet retains the simple, intuitive nature of class-oriented Java programming. In the following section, we develop a calculus of Java classes. In the third section, we motivate mixins as an extension of classes using a small but illuminating example. The fourth section extends the type-theoretic model of Java to mixins. The last section considers implementation strategies for mixins and puts our work in perspective.

[2] Dan Friedman determined in an informal poll in 1996 that almost nobody who teaches C++ teaches multiple inheritance [pers. com.].

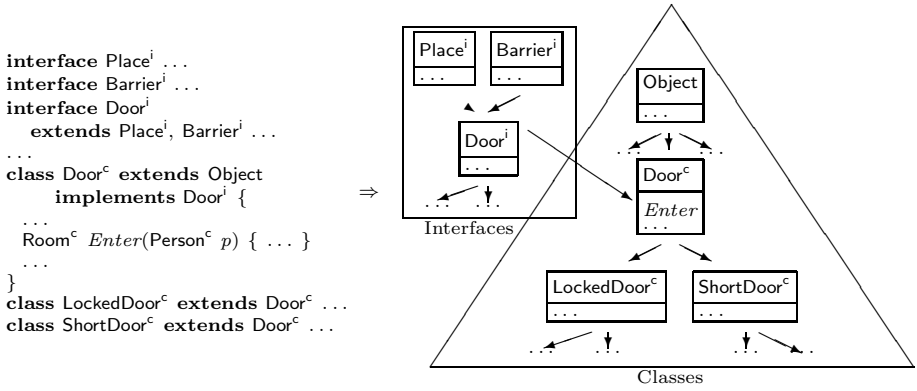


Fig. 1. A program determines a static directed acyclic graph of types

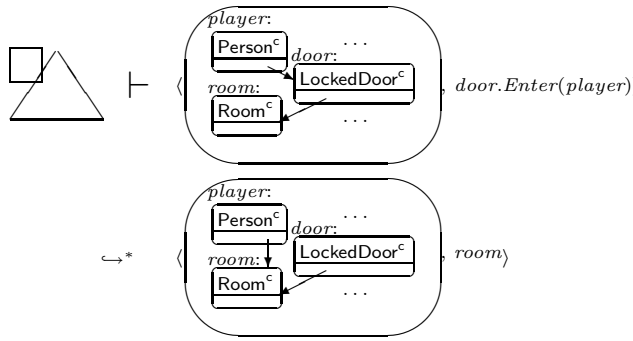


Fig. 2. Given a type graph, reductions map a store-expression pair to a new pair

2 A Model of Classes

ClassicJava is a small but essential subset of sequential Java. To model its type structure and semantics, we use well-known type elaboration and rewriting techniques for Scheme and ML [13,18,29]. Figures 1 and 2 illustrate our strategy. Type elaboration verifies that a program defines a static tree of classes and a directed acyclic graph (DAG) of interfaces. A type is simply a node in the combined graph. Each type is annotated with its collection of fields and methods, including those inherited from its ancestors.

Evaluation is modeled as a reduction on expression-store pairs in the context of a static type graph. Figure 2 demonstrates reduction using a pictorial representation of the store as a graph of objects. Each object in the store is a class-tagged record of field values, where the tag indicates the run-time type

$$
\begin{array}{rcl}
P & = & defn^*\ e \\
defn & = & \textbf{class}\ c\ \textbf{extends}\ c\ \textbf{implements}\ i^*\ \{\ field^*\ meth^*\ \} \\
 & | & \textbf{interface}\ i\ \textbf{extends}\ i^*\ \{\ meth^*\ \} \\
field & = & t\ fd \\
meth & = & t\ md\ (\ arg^*\ )\ \{\ body\ \} \\
arg & = & t\ var \\
body & = & e \mid \textbf{abstract} \\
e & = & \textbf{new}\ c \mid var \mid \mathsf{null} \mid \underline{e : c}.fd \mid \underline{e : c}.fd = e \\
 & | & e.md\ (e^*\ ) \mid \textbf{super}\ \underline{\equiv \textbf{this} : c}.md\ (e^*\ ) \\
 & | & \textbf{view}\ t\ e \mid \textbf{let}\ var = e\ \textbf{in}\ e \\
var & = & \text{a variable name or } \textbf{this} \\
c & = & \text{a class name or } \mathsf{Object} \\
i & = & \text{interface name or } \mathsf{Empty} \\
fd & = & \text{a field name} \\
md & = & \text{a method name} \\
t & = & c \mid i
\end{array}
$$

Fig. 3. ClassicJava syntax; underlined phrases are inserted by elaboration and are not part of the surface syntax

of the object and its field values are references to other objects. A single reduction step may extend the store with a new object, or it may modify a field for an existing object in the store. Dynamic method dispatch is accomplished by matching the class tag of an object in the store with a node in the static class tree; a simple relation on this tree selects an appropriate method for the dispatch.

The class model relies on as few implementation details as possible. For example, the model defines a mathematical relation, rather than a selection algorithm, to associate fields with classes for the purpose of type-checking and evaluation. Similarly, the reduction semantics only assumes that an expression can be partitioned into a proper redex and an (evaluation) context; it does not provide a partitioning algorithm. The model can easily be refined to expose more implementation details [12,18].

2.1 ClassicJava Programs

The syntax for ClassicJava is shown in Figure 3. A program P is a sequence of class and interface definitions followed by an expression. Each class definition consists of a sequence of field declarations and a sequence of method declarations, while an interface consists of methods only. A method body in a class can be **abstract**, indicating that the method must be overridden in a subclass before the class is instantiated. A method body in an interface must be **abstract**. As in Java, classes are instantiated with the **new** operator, but there are no class constructors in ClassicJava; instance variables are always initialized to null. In the evaluation language for ClassicJava, field uses and **super** invocations are annotated by the type-checker with extra information (see the underlined parts of the syntax). Finally, the **view** and **let** forms represent Java's casting expressions and local variable bindings, respectively.

A valid ClassicJava program satisfies a number of simple predicates and relations; these are described in Figure 4. For example, the ClassesOnce(P)

The sets of names for variables, classes, interfaces, fields, and methods are assumed to be mutually distinct. The meta-variable T is used for method signatures of the form $(t \ldots \longrightarrow t)$, V is used for variable lists of the form $(var \ldots)$, and Γ is used for environments mapping variables to types. Ellipses on the baseline $(\ldots)$ indicate a repeated pattern or continued sequence, while centered ellipses $(\cdots)$ indicate arbitrary missing program text (without straddling a class or interface definition).

ClassesOnce(P)	Each class name is declared only once $\forall c,c'$ **class** $c \cdots$ **class** $c' \cdots$ is in $P \implies c \neq c'$
FieldOncePerClass(P)	Field names in each class declaration are unique $\forall fd,fd'$ **class** $\cdots \{ \cdots fd \cdots fd' \cdots \}$ is in $P \implies fd \neq fd'$
MethodOncePerClass(P)	Method names in each class declaration are unique $\forall md,md'$ **class** $\cdots \{ \cdots md (\cdots) \{ \cdots \} \cdots md' (\cdots) \{ \cdots \} \cdots \}$ is in $P \implies md \neq md'$
InterfacesOnce(P)	Each interface name is declared only once $\forall i,i'$ **interface** $i \cdots$ **interface** $i' \cdots$ is in $P \implies i \neq i'$
InterfacesAbstract(P)	Method declarations in an interface are **abstract** $\forall md,e$ **interface** $\cdots \{ \cdots md (\cdots) \{e\} \cdots \}$ is in $P \implies e$ is **abstract**
$\prec^{\mathsf{c}}_P$	Class is declared as an immediate subclass $c \prec^{\mathsf{c}}_P c' \Leftrightarrow$ **class** c **extends** $c' \cdots \{ \cdots \}$ is in P
$\Subset^{\mathsf{c}}_P$	Field is declared in a class $\langle c.fd, t\rangle \Subset^{\mathsf{c}}_P c \Leftrightarrow$ **class** $c \cdots \{ \cdots t\ fd \cdots \}$ is in P
$\Subset^{\mathsf{c}}_P$	Method is declared in class $\langle md, (t_1 \ldots t_n \longrightarrow t), (var_1 \ldots var_n), e\rangle \Subset^{\mathsf{c}}_P c$ $\Leftrightarrow$ **class** $c \cdots \{ \cdots t\ md\ (t_1\ var_1 \ldots t_n\ var_n)\ \{e\} \cdots \}$ is in P
$\prec^{\mathsf{i}}_P$	Interface is declared as an immediate subinterface $i \prec^{\mathsf{i}}_P i' \Leftrightarrow$ **interface** i **extends** $\cdots i' \cdots \{ \cdots \}$ is in P
$\Subset^{\mathsf{i}}_P$	Method is declared in an interface $\langle md, (t_1 \ldots t_n \longrightarrow t), (var_1 \ldots var_n), e\rangle \Subset^{\mathsf{i}}_P i$ $\Leftrightarrow$ **interface** $i \cdots \{ \cdots t\ md\ (t_1\ var_1 \ldots t_n\ var_n)\ \{e\} \cdots \}$ is in P
$\prec\!\!\prec^{\mathsf{c}}_P$	Class declares implementation of an interface $c \prec\!\!\prec^{\mathsf{c}}_P i \Leftrightarrow$ **class** $c \cdots$ **implements** $\cdots i \cdots \{ \cdots \}$ is in P
$\leq^{\mathsf{c}}_P$	Class is a subclass $\leq^{\mathsf{c}}_P \equiv$ the transitive, reflexive closure of $\prec^{\mathsf{c}}_P$
CompleteClasses(P)	Classes that are extended are defined $\mathrm{rng}(\prec^{\mathsf{c}}_P) \subseteq \mathrm{dom}(\prec^{\mathsf{c}}_P) \cup \{\mathsf{Object}\}$
WellFoundedClasses(P)	Class hierarchy is an order $\leq^{\mathsf{c}}_P$ is antisymmetric
ClassMethodsOK(P)	Method overriding preserves the type $\forall c,c',e,e',md,T,T',V,V'$ $(\langle md, T, V, e\rangle \Subset^{\mathsf{c}}_P c$ and $\langle md, T', V', e'\rangle \Subset^{\mathsf{c}}_P c') \implies (T = T'$ or $c \not\leq^{\mathsf{c}}_P c')$
$\in^{\mathsf{c}}_P$	Field is contained in a class $\langle c'.fd, t\rangle \in^{\mathsf{c}}_P c$ $\Leftrightarrow \langle c'.fd, t\rangle \Subset^{\mathsf{c}}_P c'$ and $c' = \min\{c'' \mid c \leq^{\mathsf{c}}_P c''$ and $\exists t'$ s.t. $\langle c''.fd, t'\rangle \Subset^{\mathsf{c}}_P c''\}$
$\in^{\mathsf{c}}_P$	Method is contained in a class $\langle md, T, V, e\rangle \in^{\mathsf{c}}_P c$ $\Leftrightarrow (\langle md, T, V, e\rangle \Subset^{\mathsf{c}}_P c'$ and $c' = \min\{c'' \mid c \leq^{\mathsf{c}}_P c''$ and $\exists e', V'$ s.t. $\langle md, T, V', e'\rangle \Subset^{\mathsf{c}}_P c''\})$

Table continues in Figure 5.

Fig. 4. Predicates and relations in the model of ClassicJava

predicate states that each class name is defined at most once in the program P. The relation $\prec^{\mathsf{c}}_P$ associates each class name in P to the class it extends, and the (overloaded) $\Subset^{\mathsf{c}}_P$ relations capture the field and method declarations of P.

The syntax-summarizing relations induce a second set of relations and predicates that summarize the class structure of a program. The first of these is the subclass relation $\leq^{\mathsf{c}}_P$, which is a partial order if the CompleteClasses(P) and

$\leq^{\mathsf{i}}_P$	Interface is a subinterface $\leq^{\mathsf{i}}_P \equiv$ the transitive, reflexive closure of $\prec^{\mathsf{i}}_P$
CompleteInterfaces(P)	Extended/implemented interfaces are defined $\mathrm{rng}(\prec^{\mathsf{i}}_P) \cup \mathrm{rng}(\prec\!\!\ll^{\mathsf{c}}_P) \subseteq \mathrm{dom}(\prec^{\mathsf{i}}_P) \cup \{\mathsf{Empty}\}$
WellFoundedInterfaces(P)	Interface hierarchy is an order $\leq^{\mathsf{i}}_P$ is antisymmetric
$\ll^{\mathsf{c}}_P$	Class implements an interface $c \ll^{\mathsf{c}}_P i \Leftrightarrow \exists c', i'$ s.t. $c \leq^{\mathsf{c}}_P c'$ and $i' \leq^{\mathsf{i}}_P i$ and $c' \prec\!\!\ll^{\mathsf{c}}_P i'$
InterfaceMethodsOK(P)	Redeclarations of methods are consistent $\forall i, i', md, T, T', V, V'$ $\langle md,\ T,\ V,\ \mathbf{abstract}\rangle \notin^{\mathsf{i}}_P i$ and $\langle md,\ T',\ V',\ \mathbf{abstract}\rangle \notin^{\mathsf{i}}_P i' \implies (T = T'$ or $i \not\leq^{\mathsf{i}}_P i')$
$\in^{\mathsf{i}}_P$	Method is contained in an interface $\langle md,\ T,\ V,\ \mathbf{abstract}\rangle \in^{\mathsf{i}}_P i \Leftrightarrow \exists i'$ s.t. $i \leq^{\mathsf{i}}_P i'$ and $\langle md,\ T,\ V,\ \mathbf{abstract}\rangle \notin^{\mathsf{i}}_P i'$
ClassesImplementAll(P)	Classes supply methods to implement interfaces $\forall i, c\ c \prec\!\!\ll^{\mathsf{c}}_P i \implies (\forall md, T, V\ \langle md,\ T,\ V,\ \mathbf{abstract}\rangle \in^{\mathsf{i}}_P i \implies \exists e, V'$ s.t. $\langle md,\ T,\ V',\ e\rangle \in^{\mathsf{c}}_P c)$
NoAbstractMethods(P, c)	Class has no **abstract** methods (can be instantiated) $\forall md, T, V, e\ \ \langle md,\ T,\ V,\ e\rangle \in^{\mathsf{c}}_P c \implies e \neq \mathbf{abstract}$
$\leq_P$	Type is a subtype $\leq_P \equiv \leq^{\mathsf{c}}_P \cup \leq^{\mathsf{i}}_P \cup \ll^{\mathsf{c}}_P$
$\in_P$	Field or method is in a type $\in_P \equiv \in^{\mathsf{c}}_P \cup \in^{\mathsf{i}}_P$

Fig. 5. Predicates and relations continued from Figure 4

WellFoundedClasses(P) predicates hold. In this case, the classes declared in P form a tree that has Object at its root.

If the program describes a tree of classes, we can "decorate" each class in the tree with the collection of fields and methods that it accumulates from local declarations and inheritance. The source declaration of any field or method in a class can be computed by finding the *minimum* (*i.e.*, farthest from the root) superclass that declares the field or method. This algorithm is described precisely by the $\in^{\mathsf{c}}_P$ relations. The $\in^{\mathsf{c}}_P$ relation retains information about the source class of each field, but it does not retain the source class for a method. This reflects the property of Java classes that fields cannot be overridden (so instances of a subclass always contain the field), while methods can be overridden (and may become inaccessible).

Interfaces have a similar set of relations: the superinterface declaration relation $\prec^{\mathsf{i}}_P$ induces a subinterface relation $\leq^{\mathsf{i}}_P$. Unlike classes, a single interface can have multiple proper superinterfaces, so the subinterface order forms a DAG instead of a tree. The methods of an interface, as described by $\in^{\mathsf{i}}_P$, are the union of the interface's declared methods and the methods of its superinterfaces.

Finally, classes and interfaces are related by **implements** declarations, as captured in the $\prec\!\!\ll^{\mathsf{c}}_P$ relation. This relation is a set of edges joining the class tree and the interface graph, completing the sub*type* picture of a program. A type in the full graph is a subtype of all of its ancestors.

$\vdash_{\mathsf{p}}$

$$\frac{\begin{array}{l}\textsc{ClassesOnce}(P) \quad \textsc{InterfacesOnce}(P) \quad \textsc{MethodOncePerClass}(P) \quad \textsc{FieldOncePerClass}(P) \\ \textsc{CompleteClasses}(P) \quad \textsc{WellFoundedClasses}(P) \quad \textsc{CompleteInterfaces}(P) \quad \textsc{WellFoundedInterfaces}(P) \\ \textsc{ClassFieldsOk}(P) \quad \textsc{ClassMethodsOk}(P) \quad \textsc{InterfaceMethodsOk}(P) \quad \textsc{InterfacesAbstract}(P) \\ \textsc{ClassesImplementAll}(P) \quad P \vdash_{\mathsf{d}} defn_j \Rightarrow defn'_j \text{ for } j \in [1,n] \quad P, [\,] \vdash_{\mathsf{e}} e \Rightarrow e' : t \\ \text{where } P = defn_1 \ \ldots \ defn_n \ e\end{array}}{\vdash_{\mathsf{p}} defn_1 \ \ldots \ defn_n \ e \Rightarrow defn'_1 \ \ldots \ defn'_n \ e' : t}\,[\mathbf{prog^c}]$$

$\vdash_{\mathsf{d}}$

$$\frac{P \vdash_{\mathsf{t}} t_j \text{ for each } j \in [1,n] \qquad P, c \vdash_{\mathsf{m}} meth_k \Rightarrow meth'_k \text{ for each } k \in [1,p]}{\begin{array}{l}P \vdash_{\mathsf{d}} \textbf{class } c \cdots \{ \ t_1 \ fd_1 \ \ldots \ t_n \ fd_n \\ \qquad meth_1 \ \ldots \ meth_p \ \}\end{array} \Rightarrow \begin{array}{l}\textbf{class } c \cdots \{ \ t_1 \ fd_1 \ \ldots \ t_n \ fd_n \\ \qquad meth'_1 \ \ldots \ meth'_p \ \}\end{array}}\,[\mathbf{defn^c}]$$

$$\frac{P, i \vdash_{\mathsf{m}} meth_j \Rightarrow meth'_j \text{ for each } j \in [1,p]}{P \vdash_{\mathsf{d}} \textbf{interface } i \cdots \{ \ meth_1 \ \ldots \ meth_p \ \} \Rightarrow \textbf{interface } i \cdots \{ \ meth'_1 \ \ldots \ meth'_p \ \}}\,[\mathbf{defn^i}]$$

$\vdash_{\mathsf{m}}$

$$\frac{P \vdash_{\mathsf{t}} t \quad P \vdash_{\mathsf{t}} t_j \text{ for } j \in [1,n] \quad P, [\mathbf{this} : t_o, \ var_1 : t_1, \ \ldots \ var_n : t_n] \vdash_{\mathsf{s}} e \Rightarrow e' : t}{P, t_o \vdash_{\mathsf{m}} t \ md \ (t_1 \ var_1 \ \ldots \ t_n \ var_n) \ \{ \ e \ \} \Rightarrow t \ md \ (t_1 \ var_1 \ \ldots \ t_n \ var_n) \ \{ \ e' \ \}}\,[\mathbf{meth}]$$

$\vdash_{\mathsf{e}}$

$$\frac{P \vdash_{\mathsf{t}} c \qquad \textsc{NoAbstractMethods}(P, c)}{P, \Gamma \vdash_{\mathsf{e}} \textbf{new } c \Rightarrow \textbf{new } c : c}\,[\mathbf{new^c}] \qquad \frac{\text{where } var \in \mathrm{dom}(\Gamma)}{P, \Gamma \vdash_{\mathsf{e}} var \Rightarrow var : \Gamma(var)}\,[\mathbf{var}]$$

$$\frac{P \vdash_{\mathsf{t}} t}{P, \Gamma \vdash_{\mathsf{e}} \mathsf{null} \Rightarrow \mathsf{null} : t}\,[\mathbf{null}] \qquad \frac{P, \Gamma \vdash_{\mathsf{e}} e \Rightarrow e' : t' \quad \langle c.fd, \, t\rangle \in_P t'}{P, \Gamma \vdash_{\mathsf{e}} e.fd \Rightarrow e' \underline{: c} .fd : t}\,[\mathbf{get^c}]$$

$$\frac{P, \Gamma \vdash_{\mathsf{e}} e \Rightarrow e' : t' \qquad \langle c.fd, \, t\rangle \in_P t' \qquad P, \Gamma \vdash_{\mathsf{s}} e_v \Rightarrow e'_v : t}{P, \Gamma \vdash_{\mathsf{e}} e.fd = e_v \Rightarrow e' \underline{: c} .fd = e'_v : t}\,[\mathbf{set^c}]$$

Rules continue in Figure 7

Fig. 6. Context-sensitive checks and type elaboration rules for ClassicJava

2.2 ClassicJava Type Elaboration

The type elaboration rules for ClassicJava are defined by the following judgements:

$\vdash_{\mathsf{p}} P \Rightarrow P' : t$	P elaborates to P' with type t
$P \vdash_{\mathsf{d}} defn \Rightarrow defn'$	$defn$ elaborates to $defn'$
$P, t \vdash_{\mathsf{m}} meth \Rightarrow meth'$	$meth$ in t elaborates to $meth'$
$P, \Gamma \vdash_{\mathsf{e}} e \Rightarrow e' : t$	e elaborates to e' with type t
$P, \Gamma \vdash_{\mathsf{s}} e \Rightarrow e' : t$	e has type t using subsumption
$P \vdash_{\mathsf{t}} t$	t exists

The type elaboration rules translate expressions that access a field or call a **super** method into annotated expressions (see the underlined parts of Figure 3). For field uses, the annotation contains the compile-time type of the instance expression, which determines the class containing the declaration of the accessed field. For **super** method invocations, the annotation contains the compile-time

$$\frac{P, \Gamma \vdash_{\mathsf{e}} e \Rightarrow e' : t' \; \langle md,\, (t_1 \ldots t_n \longrightarrow t),\, (var_1 \ \ldots \ var_n),\, e_b \rangle \in_P t' \quad P, \Gamma \vdash_{\mathsf{s}} e_j \Rightarrow e'_j : t_j \text{ for } j \in [1,n]}{P, \Gamma \vdash_{\mathsf{e}} e.md\ (e_1\ \ldots\ e_n) \Rightarrow e'.md\ (e'_1\ \ldots\ e'_n) : t}\,[\mathbf{call}^{\mathsf{c}}]$$

$$\frac{P, \Gamma \vdash_{\mathsf{e}} \mathbf{this} \Rightarrow \mathbf{this} : c' \quad c' \prec^{\varsigma}_P c \quad \langle md,\, (t_1 \ldots t_n \longrightarrow t),\, (var_1 \ \ldots \ var_n),\, e_b \rangle \in_P c \quad P, \Gamma \vdash_{\mathsf{s}} e_j \Rightarrow e'_j : t_j \text{ for } j \in [1,n] \quad e_b \neq \mathbf{abstract}}{P, \Gamma \vdash_{\mathsf{e}} \mathbf{super}.md(e_1\ \ldots\ e_n) \Rightarrow \mathbf{super} \underline{\equiv \mathbf{this} : c}.md(e'_1\ \ldots\ e'_n) : t}\,[\mathbf{super}^{\mathsf{c}}]$$

$$\frac{P, \Gamma \vdash_{\mathsf{s}} e \Rightarrow e' : t}{P, \Gamma \vdash_{\mathsf{e}} \mathbf{view}\ t\ e \Rightarrow e' : t}\,[\mathbf{wcast}^{\mathsf{c}}] \qquad \frac{P \vdash_{\mathsf{t}} t}{P, \Gamma \vdash_{\mathsf{e}} \mathbf{abstract} \Rightarrow \mathbf{abstract} : t}\,[\mathbf{abs}]$$

$$\frac{P, \Gamma \vdash_{\mathsf{e}} e \Rightarrow e' : t' \quad t \leq_P t' \text{ or } t \in \mathrm{dom}(\prec^{\mathsf{i}}_P) \text{ or } t' \in \mathrm{dom}(\prec^{\mathsf{i}}_P)}{P, \Gamma \vdash_{\mathsf{e}} \mathbf{view}\ t\ e \Rightarrow \mathbf{view}\ t\ e' : t}\,[\mathbf{ncast}^{\mathsf{c}}]$$

$$\frac{P, \Gamma \vdash_{\mathsf{e}} e_1 \Rightarrow e'_1 : t_1 \quad P, \Gamma[var : t_1] \vdash_{\mathsf{e}} e_2 \Rightarrow e'_2 : t}{P, \Gamma \vdash_{\mathsf{e}} \mathbf{let}\ var = e_1\ \mathbf{in}\ e_2 \Rightarrow \mathbf{let}\ var = e'_1\ \mathbf{in}\ e'_2 : t}\,[\mathbf{let}]$$

$\vdash_{\mathsf{s}}, \vdash_{\mathsf{t}}$

$$\frac{P, \Gamma \vdash_{\mathsf{e}} e \Rightarrow e' : t' \quad t' \leq_P t}{P, \Gamma \vdash_{\mathsf{s}} e \Rightarrow e' : t}\,[\mathbf{sub}^{\mathsf{c}}] \qquad \frac{t \in \mathrm{dom}(\prec^{\varsigma}_P) \cup \mathrm{dom}(\prec^{\mathsf{i}}_P) \cup \{\mathsf{Object},\, \mathsf{Empty}\}}{P \vdash_{\mathsf{t}} t}\,[\mathbf{type}^{\mathsf{c}}]$$

Fig. 7. Rules continued from Figure 6

type of **this**, which determines the class that contains the declaration of the method to be invoked.

The complete typing rules are shown in Figure 6. A program is well-typed if its class definitions and final expression are well-typed. A definition, in turn, is well-typed when its field and method declarations use legal types and the method body expressions are well-typed. Finally, expressions are typed and elaborated in the context of an environment that binds free variables to types. For example, the $\mathbf{get}^{\mathsf{c}}$ and $\mathbf{set}^{\mathsf{c}}$ rules for fields first determine the type of the instance expression, and then calculate a class-tagged field name using $\in_P$; this yields both the type of the field and the class for the installed annotation. In the $\mathbf{set}^{\mathsf{c}}$ rule, the right-hand side of the assignment must match the type of the field, but this match may exploit subsumption to coerce the type of the value to a supertype. The other expression typing rules are similarly intuitive.

2.3 ClassicJava Evaluation

The operational semantics for ClassicJava is defined as a contextual rewriting system on pairs of expressions and stores. A store $\mathcal{S}$ is a mapping from *objects* to class-tagged field records. A field record $\mathcal{F}$ is a mapping from elaborated field names to values. The evaluation rules are a straightforward modification of those for imperative Scheme [13].

The complete evaluation rules are in Figure 8. For example, the *call* rule invokes a method by rewriting the method call expression to the body of the invoked method, syntactically replacing argument variables in this expression

$$
\begin{array}{ll}
e = \ldots \mid \mathit{object} & \mathsf{E} = [\,] \mid \mathsf{E} \underline{:\ c}.\mathit{fd} \mid \mathsf{E} \underline{:\ c}.\mathit{fd} = e \mid v \underline{:\ c}.\mathit{fd} = \mathsf{E} \\
v = \mathit{object} \mid \mathsf{null} & \quad \mid \mathsf{E}.\mathit{md}(e \ldots) \mid v.\mathit{md}(v \ldots \mathsf{E}\ e \ldots) \\
& \quad \mid \mathbf{super} \equiv v \underline{:\ c}.\mathit{md}(v \ldots \mathsf{E}\ e \ldots) \\
& \quad \mid \mathbf{view}\ t\ \mathsf{E} \mid \mathbf{let}\ \mathit{var} = \mathsf{E}\ \mathbf{in}\ e
\end{array}
$$

$$
\begin{array}{ll}
P \vdash \langle \mathsf{E}[\mathbf{new}\ c], \mathcal{S}\rangle \hookrightarrow \langle \mathsf{E}[\mathit{object}], \mathcal{S}[\mathit{object} \mapsto \langle c, \mathcal{F}\rangle]\rangle & [\mathit{new}] \\
\quad \text{where } \mathit{object} \notin \mathrm{dom}(\mathcal{S}) \text{ and } \mathcal{F} = \{c'.\mathit{fd} \mapsto \mathsf{null} \mid c \leq^{\mathsf{c}}_P c' \text{ and } \exists t \text{ s.t. } \langle c'.\mathit{fd}, t\rangle \notin^{\mathsf{c}}_P c'\} & \\
P \vdash \langle \mathsf{E}[\mathit{object} \underline{:\ c'}.\mathit{fd}], \mathcal{S}\rangle \hookrightarrow \langle \mathsf{E}[v], \mathcal{S}\rangle & [\mathit{get}] \\
\quad \text{where } \mathcal{S}(\mathit{object}) = \langle c, \mathcal{F}\rangle \text{ and } \mathcal{F}(c'.\mathit{fd}) = v & \\
P \vdash \langle \mathsf{E}[\mathit{object} \underline{:\ c'}.\mathit{fd} = v], \mathcal{S}\rangle \hookrightarrow \langle \mathsf{E}[v], \mathcal{S}[\mathit{object} \mapsto \langle c, \mathcal{F}[c'.\mathit{fd} \mapsto v]\rangle]\rangle & [\mathit{set}] \\
\quad \text{where } \mathcal{S}(\mathit{object}) = \langle c, \mathcal{F}\rangle & \\
P \vdash \langle \mathsf{E}[\mathit{object}.\mathit{md}(v_1, \ldots\ v_n)], \mathcal{S}\rangle \hookrightarrow \langle \mathsf{E}[e[\mathit{object}/\mathbf{this}, v_1/\mathit{var}_1, \ldots\ v_n/\mathit{var}_n]], \mathcal{S}\rangle & [\mathit{call}] \\
\quad \text{where } \mathcal{S}(\mathit{object}) = \langle c, \mathcal{F}\rangle \text{ and } \langle \mathit{md}, (t_1 \ldots t_n \longrightarrow t), (\mathit{var}_1 \ldots \mathit{var}_n), e\rangle \in^{\mathsf{c}}_P c & \\
P \vdash \langle \mathsf{E}[\mathbf{super} \equiv \mathit{object} : c' \overline{.\mathit{md}(v_1, \ldots\ v_n)}], \mathcal{S}\rangle & [\mathit{super}] \\
\quad \hookrightarrow \langle \mathsf{E}[e[\overline{\mathit{object}/\mathbf{this}, v_1/\mathit{var}_1, \ldots\ v_n/\mathit{var}_n}]], \mathcal{S}\rangle & \\
\quad \text{where } \langle \mathit{md}, (t_1 \ldots t_n \longrightarrow t), (\mathit{var}_1 \ldots \mathit{var}_n), e\rangle \in^{\mathsf{c}}_P c' & \\
P \vdash \langle \mathsf{E}[\mathbf{view}\ t'\ \mathit{object}], \mathcal{S}\rangle \hookrightarrow \langle \mathsf{E}[\mathit{object}], \mathcal{S}\rangle & [\mathit{cast}] \\
\quad \text{where } \mathcal{S}(\mathit{object}) = \langle c, \mathcal{F}\rangle \text{ and } c \leq_P t' & \\
P \vdash \langle \mathsf{E}[\mathbf{let}\ \mathit{var} = v\ \mathbf{in}\ e], \mathcal{S}\rangle \hookrightarrow \langle \mathsf{E}[e[v/\mathit{var}]], \mathcal{S}\rangle & [\mathit{let}] \\
\\
P \vdash \langle \mathsf{E}[\mathbf{view}\ t'\ \mathit{object}], \mathcal{S}\rangle \hookrightarrow \langle \mathsf{error: bad\ cast}, \mathcal{S}\rangle & [\mathit{xcast}] \\
\quad \text{where } \mathcal{S}(\mathit{object}) = \langle c, \mathcal{F}\rangle \text{ and } c \not\leq_P t' & \\
P \vdash \langle \mathsf{E}[\mathsf{null} \underline{:\ c}.\mathit{fd}], \mathcal{S}\rangle \hookrightarrow \langle \mathsf{error: dereferenced\ null}, \mathcal{S}\rangle & [\mathit{nget}] \\
P \vdash \langle \mathsf{E}[\mathsf{null} \underline{:\ c}.\mathit{fd} = v], \mathcal{S}\rangle \hookrightarrow \langle \mathsf{error: dereferenced\ null}, \mathcal{S}\rangle & [\mathit{nset}] \\
P \vdash \langle \mathsf{E}[\mathsf{null}.\mathit{md}(v_1, \ldots\ v_n)], \mathcal{S}\rangle \hookrightarrow \langle \mathsf{error: dereferenced\ null}, \mathcal{S}\rangle & [\mathit{ncall}]
\end{array}
$$

Fig. 8. Operational semantics for ClassicJava

with the supplied argument values. The dynamic aspect of method calls is implemented by selecting the method based on the run-time type of the object (in the store). In contrast, the *super* reduction performs **super** method selection using the class annotation that is statically determined by the type-checker.

2.4 ClassicJava Soundness

For a program of type t, the evaluation rules for ClassicJava produce either a value that has a subtype of t, or one of two errors. Put differently, an evaluation cannot get stuck. This property can be formulated as a type soundness theorem.

Theorem 1 (Type Soundness). *If* $\vdash_{\mathsf{p}} P \Rightarrow P' : t$ *and* $P' = \mathit{defn}_1 \ldots \mathit{defn}_n\ e$, *then either*

- $P' \vdash \langle e, \emptyset\rangle \hookrightarrow^* \langle \mathrm{object}, \mathcal{S}\rangle$ *and* $\mathcal{S}(\mathrm{object}) = \langle t', \mathcal{F}\rangle$ *and* $t' \leq_P t$; *or*
- $P' \vdash \langle e, \emptyset\rangle \hookrightarrow^* \langle \mathsf{null}, \mathcal{S}\rangle$; *or*
- $P' \vdash \langle e, \emptyset\rangle \hookrightarrow^* \langle \mathsf{error: bad\ cast}, \mathcal{S}\rangle$; *or*
- $P' \vdash \langle e, \emptyset\rangle \hookrightarrow^* \langle \mathsf{error: dereferenced\ null}, \mathcal{S}\rangle$.

The main lemma in support of this theorem states that each step taken in the evaluation preserves the type correctness of the expression-store pair (relative to the program) [29]. Specifically, for a configuration on the left-hand side of an

evaluation step, there exists a type environment that establishes the expression's type as some t. This environment must be consistent with the store.

Definition 2 (Environment-Store Consistency).

$$
\begin{array}{ll}
 & P,\Gamma \vdash_\sigma \mathcal{S} \\
 & \quad \Leftrightarrow (\mathcal{S}(\mathit{object}) = \langle c,\mathcal{F}\rangle \\
\Sigma_1: & \qquad \Rightarrow \Gamma(\mathit{object}) = c \\
\Sigma_2: & \qquad\quad \text{and } \mathrm{dom}(\mathcal{F}) = \{c_1.\mathit{fd} \mid \langle c_1.\mathit{fd},c_2\rangle \in^{\mathsf{c}}_P c_1\} \\
\Sigma_3: & \qquad\quad \text{and } \mathrm{rng}(\mathcal{F}) \subseteq \mathrm{dom}(\mathcal{S}) \cup \{\mathsf{null}\} \\
\Sigma_4: & \qquad\quad \text{and } (\mathcal{F}(c_1.\mathit{fd}) = \mathit{object}' \text{ and } \langle c_1.\mathit{fd},c_2\rangle \in^{\mathsf{c}}_P c_1) \\
 & \qquad\qquad \Rightarrow ((\mathcal{S}(\mathit{object}') = \langle c',\mathcal{F}'\rangle) \Rightarrow c' \leq_P c_2)) \\
\Sigma_5: & \quad \text{and } \mathit{object} \in \mathrm{dom}(\Gamma) \Rightarrow \mathit{object} \in \mathrm{dom}(\mathcal{S}) \\
\Sigma_6: & \quad \text{and } \mathrm{dom}(\mathcal{S}) \subseteq \mathrm{dom}(\Gamma).
\end{array}
$$

Note that the environment may contain bindings for lexical variables, which are not store objects.

Since the rewriting rules reduce *annotated* terms, we derive new type judgements that relate annotated terms. Each of the new rules performs exactly the same checks as the rule it is derived from, but does not add any annotation. Thus $\vdash_{\underline{\mathsf{s}}}$ is derived from $\vdash_{\mathsf{s}}$, and so forth. Only the judgement on expressions ($\vdash_{\underline{\mathsf{e}}}$) is altered slightly: we retain the **view** operation in all cases and ignore the [$\mathbf{wcast}^{\mathsf{c}}$] relation, which is only an optimization that removes an unnecessary check. This relaxation obviously does not change the type-checking or extensional behavior of any programs.

The following lemmata are used to prove the main lemma.

Lemma 3 (Free). *If* $P,\Gamma \vdash_{\underline{\mathsf{e}}} e : t$ *and* $a \notin \mathrm{dom}(\Gamma)$*, then* $P,\Gamma\ [a : t'] \vdash_{\underline{\mathsf{e}}} e : t$.

Proof. This follows by reasoning about the shape of the derivation. □

Lemma 4 (Replacement). *If* $P,\Gamma \vdash_{\underline{\mathsf{e}}} \mathsf{E}[e] : t$*,* $P,\Gamma \vdash_{\underline{\mathsf{e}}} e : t'$*, and* $P,\Gamma \vdash_{\underline{\mathsf{e}}} e' : t'$*, then* $P,\Gamma \vdash_{\underline{\mathsf{e}}} \mathsf{E}[e'] : t$.

Proof. This follows by a replacement argument in the derivation tree. □

Lemma 5 (Substitution). *If* $P,\Gamma\ [\mathit{var}_1 : t_1, \ldots\ \mathit{var}_n : t_n] \vdash_{\underline{\mathsf{e}}} e : t$ *and* $\{\mathit{var}_1, \ldots\ \mathit{var}_n\} \cap \mathrm{dom}(\Gamma) = \emptyset$ *and* $P,\Gamma \vdash_{\underline{\mathsf{s}}} v_i : t_i$ *for* $i \in [1,n]$*, then* $P,\Gamma \vdash_{\underline{\mathsf{s}}} e\ [v_1/\mathit{var}_1, \ldots\ v_n/\mathit{var}_n] : t$.

Proof. Let σ denote the substitution $[v_1/\mathit{var}_1, \ldots\ v_n/\mathit{var}_n]$, and $e' = \sigma(e)$. The proof proceeds by induction over the shape of the derivation showing that $P,\Gamma\sigma \vdash_{\underline{\mathsf{e}}} e : t$. We perform a case analysis on the last step.

Case $e = \mathbf{new}\ c$. $P,\Gamma\sigma \vdash_{\underline{\mathsf{e}}} e : c$ and $P,\Gamma \vdash_{\underline{\mathsf{e}}} e' : c$.

Case $e = var$. If $var \notin \mathrm{dom}(\sigma)$, then var must be in $\mathrm{dom}(\Gamma)$. Thus $P,\Gamma\sigma \vdash_{\underline{e}} var : t$ iff $P,\Gamma \vdash_{\underline{e}} var : t$. Otherwise, $var = var_i$ for some $i \in [1,n]$, and $P,\Gamma\sigma \vdash_{\underline{e}} var : t_i$. But $P,\Gamma \vdash_{\underline{s}} v_i : t_i$ and $e' = \sigma(e) = \sigma(var_i) = v_i$, so $P,\Gamma \vdash_{\underline{s}} e' : t_i$.

Case $e = \mathsf{null}$. By [**null**], any type is derivable.

Case $e = e_1 \underline{:c}.fd$. $P,\Gamma\sigma \vdash_{\underline{e}} e_1 : t'$ and $\langle c.fd,t\rangle \in_P t'$ follow from the antecedents. By induction, $P,\Gamma \vdash_{\underline{s}} \sigma(e_1) : t'$. Therefore $P,\Gamma \vdash_{\underline{e}} \sigma(e_1) : t''$, where t'' is a sub-type of t'. Hence, $\langle c.fd,t\rangle \in_P t''$. Thus $P,\Gamma \vdash_{\underline{e}} \sigma(e_1)\underline{:c}.fd : t$.

Case $e = e_1 \underline{:c}.fd = e_2$. This case is similar to the one above.

Case $e =$ **view** $t\ e_1$. $P,\Gamma\sigma \vdash_{\underline{e}} e_1 : t'$ and $t \leq_P t'$ follow from the antecedent. Inductively, $P,\Gamma \vdash_{\underline{e}} \sigma(e_1) : t''$ for $t'' \leq_P t'$. If $t \leq_P t''$ or $t'' \leq_P t$, P, $\Gamma \vdash_{\underline{e}}$ **view** $t\ \sigma(e_1) : t$ (by our relaxed [$\mathbf{ncast}^{\mathsf{c}}$] rule).

Case $e =$ **let** $var = e_1$ **in** e_2. Let $\sigma_1 = \sigma$. From [**let**], we get $P,\Gamma\sigma_1 \vdash_{\underline{e}} e_1 : t_1$. Let σ_2 be the substitution $[var : t_1]$. Then $P,\Gamma\sigma_2\sigma_1 \vdash_{\underline{e}} e_2 : t$. By induction, $P,\Gamma \vdash_{\underline{s}} \sigma_1(e_1) : t_1$ and $P,\Gamma\sigma_2 \vdash_{\underline{s}} \sigma_1(e_2) : t$. By using Lemma 6 for each term, $P,\Gamma \vdash_{\underline{e}} \sigma_1($**let** $var = e_1$ **in** $e) : t$.

Case $e = e_0.md\ (e_1, \ldots\ e_n)$. Typability of the expression implies $P,\Gamma\sigma \vdash_{\underline{s}} e_i : t_i$ for $i \in [1,n]$ and $P,\Gamma\sigma \vdash_{\underline{e}} e_0 : t_0$ where $\langle md, (t_1 \ldots\ t_n \to t), (var_1, ..., var_n), e\rangle \in^{\mathsf{c}}_P t_0$. By induction, $P,\Gamma \vdash_{\underline{s}} \sigma(e_i) : t_i$ for each e_i, and $P,\Gamma \vdash_{\underline{e}} \sigma(e_0) : t_0{}'$ where $t_0{}' \leq_P t_0$, which implies that $\langle md, (t_1 \ldots\ t_n \to t), (var_1, ..., var_n), e\rangle \in^{\mathsf{c}}_P t_0{}'$. Thus $P,\Gamma \vdash_{\underline{e}} \sigma(e_0.md\ (e_1, \ldots\ e_n)) : t$.

Case $e =$ **super** $\underline{\equiv$ **this** $: c}\ .md\ (e_1, \ldots\ e_n)$. This follows in a similar fashion to the rule above. Since the class c is embedded in the expression, and the induction yields a subtype of the original type, **this** can be subsumed appropriately to instantiate the method in the superclass. □

Lemma 6. *If* $P,\Gamma \vdash_{\underline{e}} \mathsf{E}[e] : t$, $P,\Gamma \vdash_{\underline{e}} e : t'$, *and* $P,\Gamma \vdash_{\underline{e}} e' : t''$ *where* $t'' \leq_P t'$, *then* $P,\Gamma \vdash_{\underline{s}} \mathsf{E}[e'] : t$.

Proof. The proof is by induction on the depth of the evaluation context E. If E is the empty context $[\,]$ we are done. Otherwise, partition $\mathsf{E}[e] = \mathsf{E}_1[\mathsf{E}_2[e]]$ where E_2 is a singular evaluation context, *i.e.*, a context whose depth is one. Consider the shape of $\mathsf{E}_2[\bullet]$, which must be one of:

Case $\bullet \underline{:c}.fd$. Since c is fixed, $\bullet$'s type does not matter: the result is the type of the field.

Case $\bullet \underline{:c}.fd = e$. Compare to the previous case.

Case $v \underline{:c}.fd = \bullet$. Since $t'' \leq_P t'$, the type of $\bullet$ is t' by subsumption and the type of the expression is unchanged.

Case $\bullet.md(e\ \ldots)$. Since $t'' \leq_P t'$ and methods in an inheritance chain must preserve the type, the result of method application is the same type.

Case $v.md(v\ \ldots\ \bullet\ e\ \ldots)$. By subsumption, arguments have the declared type by [**meth**]; t'' can be t' by subsumption.

Case super $\underline{\equiv v : c}.md\,(v \ldots \bullet\ e \ldots)$. Analogous to the previous case.

Case view $t\ \bullet$. The type of this expression is the same regardless of $\bullet$. Since $\vdash_{\underline{\mathsf{e}}}$ has the less restrictive condition for [**ncast**$^{\mathsf{c}}$] that t and t'' be comparable by $\leq_P$, the typing proceeds even if t'' is a subtype of t.

Case let $var = \bullet$ **in** e_2. We are given $P,\Gamma \vdash_{\underline{\mathsf{e}}} e : t'$, so from [**let**], $P,\Gamma\sigma_1 \vdash_{\underline{\mathsf{e}}} e_2 : t_1$ for some type t_1 where σ_1 is $[var : t']$. We must show that $P,\Gamma\sigma_2 \vdash_{\underline{\mathsf{s}}} e_2 : t_1$ where $\sigma_2 = [var : t'']$. This follows from Lemma 8. □

Definition 7. $\Gamma \leq_\Gamma \Gamma'$ if $\mathrm{dom}(\Gamma) = \mathrm{dom}(\Gamma')$ and $\forall\ v \in \mathrm{dom}(\Gamma)$, $\Gamma'(v) \leq_P \Gamma(v)$.

Lemma 8. *If* $P,\Gamma \vdash_{\underline{\mathsf{e}}} e : t$ *and* $\Gamma \leq_\Gamma \Gamma'$, *then* $P,\Gamma' \vdash_{\underline{\mathsf{s}}} e : t$.

Proof. The proof is a simple adaptation of that of Lemma 5. □

We can now prove the subject reduction lemma. Since ClassicJava does not include any primitives, its type soundness follows by induction over this result.

Definition 9 (Error Configuration). An *error configuration* is any one of [*xcast*], [*nget*], [*nset*] and [*ncall*].

Lemma 10 (Subject Reduction). *If* $P,\Gamma \vdash_{\underline{\mathsf{e}}} e : t$, $P,\Gamma \vdash_\sigma \mathcal{S}$, $\mathrm{fv}(e) \subseteq \mathrm{dom}(\Gamma)$, *and* $\langle e,\mathcal{S}\rangle \hookrightarrow \langle e',\mathcal{S}'\rangle$, *then* e' *is an answer,* e' *is an error configuration, or* $\exists\ \Gamma'$ *such that*

1. $P,\Gamma' \vdash_{\underline{\mathsf{s}}} e' : t$,
2. $P,\Gamma' \vdash_\sigma \mathcal{S}'$.

Proof. The proof examines the structure of the reduction step. For each case, we construct the new environment Γ' and show that, if execution has not halted with an answer or in an error configuration, the two consequents of the theorem are satisfied relative to the new expression, store, and environment.

Case [*new*]. Set $\Gamma' = \Gamma\ [object : c]$.

1. We have $P,\Gamma \vdash_{\underline{\mathsf{e}}} \mathsf{E}[\mathbf{new}\ c] : t$. From Σ_5, $object \notin \mathrm{dom}(\mathcal{S}) \Rightarrow object \notin \mathrm{dom}(\Gamma)$. Thus $P,\Gamma' \vdash_{\underline{\mathsf{e}}} \mathsf{E}[\mathbf{new}\ c] : t$ by Lemma 3. Since $P,\Gamma' \vdash_{\underline{\mathsf{e}}} \mathbf{new}\ c : c$ and $P,\Gamma' \vdash_{\underline{\mathsf{e}}} object : c$ we use Lemma 4 to get $P,\Gamma' \vdash_{\underline{\mathsf{e}}} \mathsf{E}[object] : t$.
2. Let $\mathcal{S}'(object) = \langle c,\mathcal{F}\rangle$. *object* is the only new element in $\mathrm{dom}(\mathcal{S}')$.
   Σ_1**:** $\Gamma'(object) = c$.
   Σ_2**:** $\mathrm{dom}(\mathcal{F})$ is correct by construction.
   Σ_3**:** $\mathrm{rng}(\mathcal{F}) = \{\mathsf{null}\}$.
   Σ_4**:** Since $\mathrm{rng}(\mathcal{F}) = \{\mathsf{null}\}$, this property is unaffected.
   Σ_5 **and** Σ_6**:** The only change to Γ and $\mathcal{S}$ is *object*.

Case [*get*]. Set $\Gamma' = \Gamma$. Let t' be the type such that $P,\Gamma \vdash_{\underline{e}} object\ \underline{:\ c'}\ .fd : t'$. $P,\Gamma \vdash_{\underline{e}} \mathsf{E}[object\ \underline{:\ c'}\ .fd] : t$ implies that $\Gamma(object) \leq_P c'$. Thus $\mathcal{S}(object) = \langle c,\mathcal{F}\rangle$ with $c'.fd \in \mathrm{dom}(\mathcal{F})$.

1. If v is null, it can be cast to t', so $P,\Gamma' \vdash_{\underline{e}} \mathsf{E}[v] : t$ by Lemma 4. If v is not null, by Σ_4, $\mathcal{S}(v) = \langle c'',\_\rangle$ where $c'' \leq_P t'$. By Lemma 6, $P,\Gamma' \vdash_{\underline{s}} \mathsf{E}[v] : t$.
2. $\mathcal{S}$ and Γ are unchanged.

Case [*set*].

1. The proof is by a straight-forward extension of the proof for [*get*].
2. The only change to the store is a field update; thus only Σ_3 and Σ_4 are affected. Let v be the assigned value. Assume v is non-null.

   Σ_3**:** Since v is typable, it must be in $\mathrm{dom}(\Gamma)$. By Σ_5, it is therefore in $\mathrm{dom}(\mathcal{S})$.

   Σ_4**:** The typing of the active expression indicates that the type of v can be treated as the type of the field *fd* by subsumption. Combining this with Σ_1 indicates that the type tag of v will preserve Σ_4.

Case [*call*]. From $P,\Gamma \vdash_{\underline{e}} object.md(v_1, \ldots\ v_n) : t$ we know $P,\Gamma \vdash_{\underline{e}} object : t'$, $P,\Gamma \vdash_{\underline{s}} v_i : t_i$ for i in $[1,n]$, and $\langle md, (t_1 \ldots\ t_n \to t), (var_1, ..., var_n), e\rangle \in^{\mathsf{c}}_P t'$. The type-checking of P proves that $P,t_0 \vdash_{\underline{\mathsf{m}}} t\ md\ (t_1\ var_1, \ldots\ t_n\ var_n)\ \{e\}$, which implies that $P,[\mathbf{this} : t_0, var_1 : t_1, \ldots\ var_n : t_n] \vdash_{\underline{s}} e : t$ where t_0 is the defining class of *md*. Further, we know that $t' \leq_P t_0$ from $\in^{\mathsf{c}}_P$ for methods and ClassMethodsOk(P).

1. Lemma 5 shows that $P,\Gamma \vdash_{\underline{s}} e[object/\mathbf{this}, v_1/var_1, \ldots\ v_n/var_n] : t$.
2. $\mathcal{S}' = \mathcal{S}$ and Γ' does not bind new addresses, so $\vdash_\sigma$ is preserved.

Case [*super*]. The proof is essentially the same as that for [*call*].

Case [*let*]. $P,\Gamma \vdash_{\underline{e}} \mathbf{let}\ var = v\ \mathbf{in}\ e : t$ implies $P,\Gamma \vdash_{\underline{e}} v : t'$ for some type t'. Set $\Gamma' = \Gamma\ [var : t']$. From [**let**], $P,\Gamma' \vdash_{\underline{e}} e : t$.

1. By Lemma 5, $P,\Gamma \vdash_{\underline{s}} e\ [v/var] : t$.
2. The store is unchanged and the only addition to the environment is not an *object*, so the store relation holds. □

2.5 Related Work on Classes

Our model for class-based object-oriented languages is similar to two recently published semantics for Java [9,28], but entirely motivated by prior work on Scheme and ML models [13,18,29]. The approach is fundamentally different from most of the previous work on the semantics of objects. Much of that work has focused on interpreting object systems and the underlying mechanisms via record extensions of lambda calculi [11,19,24,22,25] or as "native" object calculi (with a record flavor) [1,2,3]. In our semantics, types are simply the names of entities declared in the program; the collection of types forms a DAG, which is specified

by the programmer. The collection of types is static during evaluation[3] and is only used for field and method lookups and casts. The evaluation rules describe how to transform statements, formed over the given type context, into plain values. The rules work on plain program text such that each intermediate stage of the evaluation is a complete program. In short, the model is as simple and intuitive as that of first-order functional programming enriched with a language for expressing hierarchical relationships among data types.

3 From Classes to Mixins: An Example

Implementing a maze adventure game [16, page 81] illustrates the need for adding mixins to a class-based language. A player in the adventure game wanders through rooms and doors in a virtual world. All locations in the virtual world share some common behavior, but also differ in a wide variety of properties that make the game interesting. For example, there are many kinds of doors, including locked doors, magic doors, doors of varying heights, and doors that combine several varieties into one. The natural class-based approach for implementing different kinds of doors is to implement each variation with a new subclass of a basic door class, $\mathsf{Door}^{\mathsf{c}}$. The left side of Figure 9 shows the Java definition for two simple $\mathsf{Door}^{\mathsf{c}}$ subclasses, $\mathsf{LockedDoor}^{\mathsf{c}}$ and $\mathsf{ShortDoor}^{\mathsf{c}}$. An instance of $\mathsf{LockedDoor}^{\mathsf{c}}$ requires a key to open the door, while an instance of $\mathsf{ShortDoor}^{\mathsf{c}}$ requires the player to duck before walking through the door.

A subclassing approach to the implementation of doors seems natural at first because the programmer declares only what is different in a particular door variation as compared to some other door variation. Unfortunately, since the superclass of each variation is fixed, door variations cannot be composed into more complex, and thus more interesting, variations. For example, the $\mathsf{LockedDoor}^{\mathsf{c}}$ and $\mathsf{ShortDoor}^{\mathsf{c}}$ classes cannot be combined to create a new $\mathsf{LockedShortDoor}^{\mathsf{c}}$ class for doors that are both locked and short.

A mixin approach solves this problem. Using mixins, the programmer declares how a particular door variation differs from an *arbitrary* door variation. This creates a function from door classes to door classes, using an interface as the input type. Each basic door variation is defined as a separate mixin. These mixins are then functionally composed to create many different kinds of doors.

A programmer implements mixins in exactly the same way as a derived class, except that the programmer cannot rely on the *implementation* of the mixin's superclass, only on its *interface*. We consider this an advantage of mixins because it enforces the maxim "program to an interface, not an implementation" [16, page 11].

The right side of Figure 9 shows how to define mixins for locked and short doors. The mixin $\mathsf{Locked}^{\mathsf{m}}$ is nearly identical to the original $\mathsf{LockedDoor}^{\mathsf{c}}$ class definition, except that the superclass is specified via the interface $\mathsf{Door}^{\mathsf{i}}$. The new $\mathsf{LockedDoor}^{\mathsf{c}}$ and $\mathsf{ShortDoor}^{\mathsf{c}}$ classes are created by applying $\mathsf{Locked}^{\mathsf{m}}$ and $\mathsf{Short}^{\mathsf{m}}$

[3] Dynamic class loading could be expressed in this framework as an addition to the static context. Still, the context remains the same for most of the evaluation.

```
class LockedDoor^c extends Door^c {
  boolean canOpen(Person^c p) {
    if (!p.hasItem(theKey)) {
      System.out.println("You don't have the Key");
      return false;
    }
    System.out.println("Using key...");
    return super.canOpen(p);
  }
}
class ShortDoor^c extends Door^c {
  boolean canPass(Person^c p) {
    if (p.height() > 1) {
      System.out.println("You are too tall");
      return false;
    }
    System.out.println("Ducking into door...");
    return super.canPass(p);
  }
}

/* Cannot merge for LockedShortDoor^c */
```

```
interface Door^i {
  boolean canOpen(Person^c p);
  boolean canPass(Person^c p);
}
mixin Locked^m extends Door^i {
  boolean canOpen(Person^c p) {
    if (!p.hasItem(theKey)) {
      System.out.println("You don't have the Key");
      return false;
    }
    System.out.println("Using key...");
    return super.canOpen(p);
  }
}
mixin Short^m extends Door^i {
  boolean canPass(Person^c p) {
    if (p.height() > 1) {
      System.out.println("You are too tall");
      return false;
    }
    System.out.println("Ducking into door...");
    return super.canPass(p);
  }
}
class LockedDoor^c = Locked^m(Door^c);
class ShortDoor^c = Short^m(Door^c);
class LockedShortDoor^c = Locked^m(Short^m(Door^c));
```

Fig. 9. Some class definitions and their translation to composable mixins

to the class Door^c, respectively. Similarly, applying $\mathsf{Locked}^\mathsf{m}$ to $\mathsf{ShortDoor}^\mathsf{c}$ yields a class for locked, short doors.

Consider another door variation: $\mathsf{MagicDoor}^\mathsf{c}$, which is similar to $\mathsf{LockedDoor}^\mathsf{c}$ except the player needs a book of spells instead of a key. We can extract the common parts of the implementation of $\mathsf{MagicDoor}^\mathsf{c}$ and $\mathsf{LockedDoor}^\mathsf{c}$ into a new mixin, $\mathsf{Secure}^\mathsf{m}$. Then, key- or book-specific information is composed with $\mathsf{Secure}^\mathsf{m}$ to produce $\mathsf{Locked}^\mathsf{m}$ and $\mathsf{Magic}^\mathsf{m}$, as shown in Figure 10. Each of the new mixins extends Door^i since the right hand mixin in the composition, $\mathsf{Secure}^\mathsf{m}$, extends Door^i.

The $\mathsf{Locked}^\mathsf{m}$ and $\mathsf{Magic}^\mathsf{m}$ mixins can also be composed to form $\mathsf{LockedMagic}^\mathsf{m}$. This mixin has the expected behavior: to open an instance of $\mathsf{LockedMagic}^\mathsf{m}$, the player must have both the key and the book of spells. This combinational effect is achieved by a chain of **super**.*canOpen*() calls that use distinct, non-interfering versions of *neededItem*. The *neededItem* declarations of $\mathsf{Locked}^\mathsf{m}$ and $\mathsf{Magic}^\mathsf{m}$ do not interfere with each other because the interface extended by $\mathsf{Locked}^\mathsf{m}$ is Door^i, which does not contain *neededItem*. In contrast, Door^i does contain *canOpen*, so the *canOpen* method in $\mathsf{Locked}^\mathsf{m}$ overrides and chains to the *canOpen* in $\mathsf{Magic}^\mathsf{m}$.

4 Mixins for Java

MixedJava is an extension of ClassicJava with mixins. In ClassicJava, a class is assembled as a chain of **class** expressions. Specifically, the content of a

```
interface Secure^i extends Door^i {
  Object neededItem();
}
mixin Secure^m extends Door^i implements Secure^i {
  Object neededItem() { return null; }
  boolean canOpen(Person^c p) {
    Object item = neededItem();
    if (!p.hasItem(item)) {
      System.out.println("You don't have the " + item);
      return false;
    }
    System.out.println("Using " + item + "...");
    return super.canOpen(p);
  }
}
mixin NeedsKey^m extends Secure^i {
  Object neededItem() {
    return theKey;
  }
}
mixin NeedsSpell^m extends Secure^i {
  Object neededItem() {
    return theSpellBook;
  }
}
mixin Locked^m = NeedsKey^m compose Secure^m;
mixin Magic^m = NeedsSpell^m compose Secure^m;
mixin LockedMagic^m = Locked^m compose Magic^m;
mixin LockedMagicDoor^m = LockedMagic^m compose Door^m;
class LockedDoor^c = Locked^m(Door^c); ...
```

Fig. 10. Composing mixins for localized parameterization

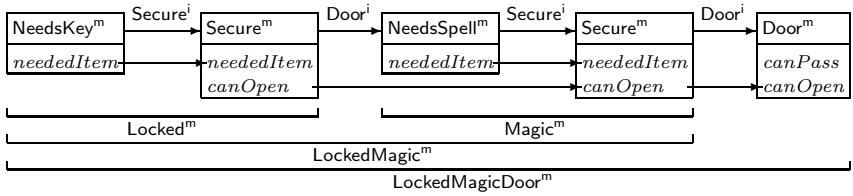


Fig. 11. The LockedMagicDoor^m mixin corresponds to a sequence of atomic mixins

class is defined by its immediate field and method declarations *and* by the declarations of its superclasses, up to Object.[4] In MixedJava, a "class" is assembled by composing a chain of mixins. The content of the class is defined by the field and method declarations in the entire chain.

[4] We use boldfaced **class** to refer to the content of a single **class** expression, as opposed to an actual class.

MixedJava provides two kinds of mixins:

- An *atomic* mixin declaration is similar to a **class** declaration. An atomic mixin declares a set of fields and methods that are extensions to some inherited set of fields and methods. In contrast to a class, an atomic mixin specifies its inheritance with an *inheritance interface*, not a static connection to an existing class. By abuse of terminology, we say that a mixin *extends* its inheritance interface.

  A mixin's inheritance interface determines how method declarations within the mixin are combined with inherited methods. If a mixin declares a method x that is not contained in its inheritance interface, then that declaration never overrides another x.

  An atomic mixin *implements* one or more interfaces as specified in the mixin's definition. In addition, a mixin always implements its inheritance interface.
- A *composite* mixin does not declare any new fields or methods. Instead, it composes two existing mixins to create a new mixin. The new composite mixin has all of the fields and methods of its two constituent mixins. Method declarations in the left-hand mixin override declarations in the right-hand mixin according to the left-hand mixin's inheritance interface. Composition is allowed only when the right-hand mixin implements the left-hand mixin's inheritance interface.

  A composite mixin extends the inheritance interface of its right-hand constituent, and it implements all of the interfaces that are implemented by its constituents. Composite mixins can be composed with other mixins, producing arbitrarily long chains of atomic mixin compositions.[5]

Figure 11 illustrates how the mixin $\mathsf{LockedMagicDoor}^{\mathsf{m}}$ from the previous section corresponds to a chain of atomic mixins. The arrows connecting the tops of the boxes represent mixin compositions; in each composition, the inheritance interface for the left-hand side is noted above the arrow. The other arrows show how method declarations in each mixin override declarations in other mixins according to the composition interfaces. For example, there is no arrow from the first $\mathsf{Secure}^{\mathsf{m}}$'s *neededItem* to $\mathsf{Magic}^{\mathsf{m}}$'s method because *neededItem* is not included in the $\mathsf{Door}^{\mathsf{i}}$ interface. The *canOpen* method is in both $\mathsf{Door}^{\mathsf{i}}$ and $\mathsf{Secure}^{\mathsf{i}}$, so that corresponding arrows connect all declarations of *canOpen*.

Mixins completely subsume the role of classes. A mixin can be instantiated with **new** when the mixin does not inherit any services. In MixedJava, this is indicated by declaring that the mixin extends the special interface Empty.

[5] Our composition operator is associative semantically, but not type-theoretically. The type system could be strengthened to make composition associative—giving MixedJava a categorical flavor—by letting each mixin declare a set of interfaces for inheritance, rather than a single interface. Each required interface must then either be satisfied or propagated by a composition. We have not encountered a practical use for the extended type system.

$$
\begin{array}{rcl}
defn & = & \textbf{mixin}\ m\ \textbf{extends}\ i\ \textbf{implements}\ i^*\ \{\ field^*\ meth^*\ \} \\
 & | & \textbf{mixin}\ m = m\ \textbf{compose}\ m \\
 & | & \textbf{interface}\ i\ \textbf{extends}\ i^*\ \{\ meth^*\ \} \\
e & = & \textbf{new}\ m \mid var \mid \mathsf{null} \mid e\underline{:m}.fd \mid e\underline{:m}.fd = e \\
 & | & e.md\ (e^*) \mid \textbf{super}\underline{\equiv \textbf{this}}.md\ (e^*) \\
 & | & \textbf{view}\underline{t\ \textbf{as}\ t}\ e \mid \textbf{let}\ var = e\ \textbf{in}\ e \\
m & = & \text{mixin name} \\
t & = & m \mid i
\end{array}
$$

Fig. 12. Syntax extensions for MixedJava

Consequently, we omit classes from our model of mixins, even though a realistic language would include both mixins and classes.

The following subsections present a precise description of MixedJava. Section 4.1 describes the syntax and type structure of MixedJava programs, followed by the type elaboration rules in Section 4.2. Section 4.3 explains the operational semantics of MixedJava, which is significantly different from that of ClassicJava. Section 4.4 presents a type soundness theorem, Section 4.5 briefly considers implementation issues, and Section 4.6 discusses related work.

4.1 MixedJava Programs

Figure 12 contains the syntax for MixedJava; the missing productions are inherited from the grammar of ClassicJava in Figure 3. The primary change to the syntax is the replacement of **class** declarations with **mixin** declarations. Another change is in the annotations added by type elaboration. First, **view** expressions are annotated with the source type of the expression. Second, a type is no longer included in the **super** annotation. Type elaboration also inserts extra **view** expressions into a program to implement subsumption.

The predicates and relations in Figure 13 (along with the interface-specific parts of Figure 4) summarize the syntactic content of a MixedJava program. A well-formed program induces a subtype relation $\leq_P^{\mathsf{m}}$ on its mixins such that a composite mixin is a subtype of each of its constituent mixins.

Since each composite mixin has two supertypes, the type graph for mixins is a DAG, rather than a tree as for classes. This DAG can lead to ambiguities if subsumption is based on subtypes. For example, $\mathsf{LockedMagic}^{\mathsf{m}}$ is a subtype of $\mathsf{Secure}^{\mathsf{m}}$, but it contains two copies of $\mathsf{Secure}^{\mathsf{m}}$ (see Figure 11), so an instance of $\mathsf{LockedMagic}^{\mathsf{m}}$ is ambiguous as an instance of $\mathsf{Secure}^{\mathsf{m}}$. More concretely, the fragment

$$
\begin{array}{l}
\mathsf{LockedMagicDoor}^{\mathsf{m}}\ door = \textbf{new}\ \mathsf{LockedMagicDoor}^{\mathsf{m}}; \\
(\textbf{view}\ \mathsf{Secure}^{\mathsf{m}}\ door).neededItem();
\end{array}
$$

is ill-formed because $\mathsf{LockedMagic}^{\mathsf{m}}$ is not *viewable as* $\mathsf{Secure}^{\mathsf{m}}$. The "viewable as" relation $\trianglelefteq_P$ is a restriction on the subtype relation that eliminates ambiguities. Subsumption is thus based on $\trianglelefteq_P$ rather than $\leq_P$. The relations $\in_P^{\mathsf{m}}$, which collect the fields and methods contained in each mixin, similarly eliminate ambiguities.

$\text{MixinsOnce}(P)$	Each mixin name is declared only once $\forall m,m'$ **mixin** $m \cdots$ **mixin** $m' \cdots$ is in $P \implies m \neq m'$
$\text{FieldOncePerMixin}(P)$	Field names in each mixin declaration are unique $\forall fd,fd'$ **mixin** $\cdots \{ \cdots fd \cdots fd' \cdots \}$ is in $P \implies fd \neq fd'$
$\text{MethodOncePerMixin}(P)$	Method names in each mixin declaration are unique $\forall md,md'$ **mixin** $\cdots \{ \cdots md\ (\cdots)\ \{\cdots\} \cdots md'\ (\cdots)\ \{\cdots\} \cdots \}$ is in $P \implies md \neq md'$
$\text{NoAbstractMixins}(P)$	Methods in a mixin are not **abstract** $\forall md,e$ **mixin** $\cdots \{ \cdots md\ (\cdots)\ \{\ e\ \} \cdots \}$ is in $P \implies e \neq$ **abstract**
$\prec^{\mathsf{m}}_P$	Mixin declares an inheritance interface $m \prec^{\mathsf{m}}_P i \Leftrightarrow$ **mixin** m **extends** $i \cdots \{\cdots\}$ is in P
$\prec\!\!\!\prec^{\mathsf{m}}_P$	Mixin declares implementation of an interface $m \prec\!\!\!\prec^{\mathsf{m}}_P i \Leftrightarrow$ **mixin** $m \cdots$ **implements** $\cdots i \cdots \{\cdots\}$ is in P
$\bullet \doteq^{\mathsf{m}}_P \bullet \circ \bullet$	Mixin is declared as a composition $m \doteq^{\mathsf{m}}_P m' \circ m'' \Leftrightarrow$ **mixin** $m =$ m' **compose** m'' is in P
$\Subset^{\mathsf{m}}_P$	Method is declared in a mixin $\langle md,\ (t_1 \ldots t_n \longrightarrow t),\ (var_1\ \ldots\ var_n),\ e\rangle \Subset^{\mathsf{m}}_P m$ $\Leftrightarrow$ **mixin** $m \cdots \{ \cdots t\ md\ (t_1\ var_1\ \ldots\ t_n\ var_n)\ \{\ e\ \} \cdots \}$ is in P
$\Subset^{\mathsf{m}}_P$	Field is declared in a mixin $\langle m.fd,\ t\rangle \Subset^{\mathsf{m}}_P m \Leftrightarrow$ **mixin** $m \cdots \{ \cdots t\ fd \cdots \}$ is in P
$\leq^{\mathsf{m}}_P$	Mixin is a submixin $m \leq^{\mathsf{m}}_P m' \Leftrightarrow m = m'$ or $(\exists m'', m'''$ s.t. $m \doteq^{\mathsf{m}}_P m'' \circ m'''$ and $(m'' \leq^{\mathsf{m}}_P m'$ or $m''' \leq^{\mathsf{m}}_P m'))$
$\trianglelefteq^{\mathsf{m}}_P$	Mixin is viewable as a mixin $m \trianglelefteq^{\mathsf{m}}_P m' \Leftrightarrow m = m'$ or $(\exists m'', m'''$ s.t. $m \doteq^{\mathsf{m}}_P m'' \circ m'''$ and $(m'' \trianglelefteq^{\mathsf{m}}_P m'$ xor $m''' \trianglelefteq^{\mathsf{m}}_P m'))$
$\text{CompleteMixins}(P)$	Mixins that are composed are defined $\text{rng}(\doteq^{\mathsf{m}}_P) \subseteq \{m \circ m' \mid m,m' \in \text{dom}(\prec^{\mathsf{m}}_P) \cup \text{dom}(\doteq^{\mathsf{m}}_P)\}$
$\text{WellFoundedMixins}(P)$	Mixin hierarchy is an order $\leq^{\mathsf{m}}_P$ is antisymmetric
$\text{CompleteInterfaces}(P)$	Extended/implemented interfaces are defined $\text{rng}(\prec^{\mathsf{i}}_P) \cup \text{rng}(\prec^{\mathsf{m}}_P) \cup \text{rng}(\prec\!\!\!\prec^{\mathsf{m}}_P) \subseteq \text{dom}(\prec^{\mathsf{i}}_P) \cup \{\mathsf{Empty}\}$
$\lhd^{\mathsf{m}}_P$	Mixin extends an interface $m \lhd^{\mathsf{m}}_P i \Leftrightarrow m \prec^{\mathsf{m}}_P i$ or $(\exists m', m''$ s.t. $m \doteq^{\mathsf{m}}_P m' \circ m''$ and $m'' \lhd^{\mathsf{m}}_P i)$
$\ll^{\mathsf{m}}_P$	Mixin implements an interface $m \ll^{\mathsf{m}}_P i \Leftrightarrow \exists m', i'$ s.t. $m \leq^{\mathsf{m}}_P m'$ and $i' \leq^{\mathsf{i}}_P i$ and $(m' \prec^{\mathsf{m}}_P i'$ or $m' \prec\!\!\!\prec^{\mathsf{m}}_P i')$
$\ll\!\!\lhd^{\mathsf{m}}_P$	Mixin is viewable as an interface $m \ll\!\!\lhd^{\mathsf{m}}_P i \Leftrightarrow (\exists i'$ s.t. $i \leq^{\mathsf{i}}_P i'$ and $(m \prec^{\mathsf{m}}_P i'$ or $m \prec\!\!\!\prec^{\mathsf{m}}_P i'))$ or $(\exists m', m''$ s.t. $m \doteq^{\mathsf{m}}_P m' \circ m''$ and $(m' \ll\!\!\lhd^{\mathsf{m}}_P i$ xor $m'' \ll\!\!\lhd^{\mathsf{m}}_P i))$
$\text{MixinCompositionsOK}(P)$	Mixins are composed safely $\forall m,m',m''\ m \doteq^{\mathsf{m}}_P m' \circ m'' \implies \exists i\ s.t.\ m' \lhd^{\mathsf{m}}_P i$ and $m'' \ll\!\!\lhd^{\mathsf{m}}_P i$
:: and @	Sequence constructors :: adds an element to the beginning of a sequence; @ appends two sequences
$\longrightarrow_P$	Mixin corresponds to a chain of atomic mixins $m \longrightarrow_P M$ $\Leftrightarrow (\exists i$ s.t. $m \prec^{\mathsf{m}}_P i$ and $M = [m])$ or $(\exists m', m'', M', M''$ s.t. $m \doteq^{\mathsf{m}}_P m' \circ m''$ and $m' \longrightarrow_P M'$ and $m'' \longrightarrow_P M''$ and $M = M'@M'')$
$\leq^{\mathsf{M}}$	Views have an inverted subsequence order $M \leq^{\mathsf{M}} M' \Leftrightarrow \exists M''$ s.t. $M = M''@M'$

Table continues in Figure 14.

Fig. 13. Predicates and relations in the model of MixedJava

4.2 MixedJava Type Elaboration

Despite replacing the subtype relation with the "viewable as" relation for subsumption, ClassicJava's type elaboration strategy applies equally well for typing MixedJava. The typing rules in Figure 15 are combined with the **defn**$^{\mathsf{i}}$, **meth**, **let**, **var**, **null**, and **abs** rules from Figure 6.

$\text{MixinMethodsOK}(P)$ Method definitions match inheritance interface
$\forall m,i,e,md,T,T',V,V'$
$(\langle md,\ T,\ V,\ e\rangle \notin^{\mathsf{m}}_P m$ and $\langle md,\ T',\ V',$ **abstract**$\rangle \in^{\mathsf{m}}_P i) \Longrightarrow (T = T'$ or $m \not\prec^{\mathsf{m}}_P i)$

$\in^{\mathsf{m}}_P$ Field is contained in a mixin
$\langle m'.fd,\ t\rangle \in^{\mathsf{m}}_P m$
$\Leftrightarrow \exists M, M'$ s.t. $m \longrightarrow_P M$ and $\langle m'.fd,\ t\rangle \notin^{\mathsf{m}}_P m'$
and $\{m'::M'\} = \{m'::M' \mid M \leq^{\mathsf{M}} m'::M'$ and $\exists t'$ s.t. $\langle m'.fd,\ t'\rangle \notin^{\mathsf{m}}_P m'\}$

$\in^{\mathsf{m}}_P$ Method is contained in a mixin
$\langle md,\ T,\ V,\ e\rangle \in^{\mathsf{m}}_P m$
$\Leftrightarrow \exists M, m', M'$ s.t. $m \longrightarrow_P M$ and $\langle md,\ T,\ V,\ e\rangle \notin^{\mathsf{m}}_P m'$
and $\{m'::M'\} = \{m'::M' \mid M \leq^{\mathsf{M}} m'::M'$ and $\exists V', e'$ s.t. $\langle md,\ T,\ V',\ e'\rangle \notin^{\mathsf{m}}_P m'\}$

$\text{MixinsImplementAll}(P)$ Mixins supply methods to implement interfaces
$\forall m,i\ m \not\prec^{\mathsf{m}}_P i \Longrightarrow (\forall md,T\ \langle md,\ T,\ V,$ **abstract**$\rangle \in^{\mathsf{i}}_P i$
$\Longrightarrow (\exists\ e$ s.t. $\langle md,\ T,\ V,\ e\rangle \notin^{\mathsf{m}}_P m$
or $\exists\ i'$ *s.t.* $(m \not\prec^{\mathsf{m}}_P i'$
and $\langle md,\ T,\ V,$ **abstract**$\rangle \in^{\mathsf{i}}_P i')))$

$\leq_P$ Type is a subtype $\leq_P \equiv \leq^{\mathsf{m}}_P \cup \leq^{\mathsf{i}}_P \cup \ll^{\mathsf{m}}_P$

$\trianglelefteq_P$ Type is viewable as another type $\trianglelefteq_P \equiv \trianglelefteq^{\mathsf{m}}_P \cup \leq^{\mathsf{i}}_P \cup \lll^{\mathsf{m}}_P$

$\in_P$ Field or method is in a type $\in_P \equiv \in^{\mathsf{m}}_P \cup \in^{\mathsf{i}}_P$

$\bullet/\bullet \triangleright \bullet$ Mixin selects a view in a chain
$M/m \triangleright M' \Leftrightarrow \{M'\} = \{M''@M''' \mid m \longrightarrow_P M''$ and $M \leq^{\mathsf{M}} M''@M'''\}$

$\bullet/\bullet \triangleright \bullet$ Interface selects a view in a chain
$M/i \triangleright M' \Leftrightarrow M' = \min\{m::M'' \mid m \prec^{\mathsf{m}}_P i$ and $M \leq^{\mathsf{M}} m::M''\}$

$\bullet/\bullet \propto \bullet$ Method in a sequence is the same as in a subsequence
$m::M/md \propto M' \Leftrightarrow\ m::M = M'$ or $(\exists i,T,V,M''$ s.t. $m \prec^{\mathsf{m}}_P i$ and $\langle md,\ T,\ V,$ **abstract**$\rangle \in^{\mathsf{i}}_P i$
and $M'/i \triangleright M''$ and $M''/md \propto M')$

$\bullet \in^{\mathsf{rt}}_P \bullet$ in $\bullet$ Method and view is selected by a view in a chain
$\langle md,\ T,\ V,\ e,\ m::M\rangle \in^{\mathsf{rt}}_P M_v$ in M_o
$\Leftrightarrow \langle md,\ T,\ V,\ e\rangle \notin^{\mathsf{m}}_P m$ and $M_b = \max\{M' \mid M_v/md \propto M'\}$
and $\{m::M\} = \{m::M \mid m::M \leq^{\mathsf{M}} M_o$ and $m::M/md \propto M_b$
and $\exists V', e'$ s.t. $\langle md,\ T,\ V',\ e'\rangle \notin^{\mathsf{m}}_P m\}$

Fig. 14. Predicates and relations continued from Figure 13

Three of the new rules deserve special attention. First, the **super**$^{\mathsf{m}}$ rule allows a **super** call only when the method is declared in the current mixin's inheritance interface, where the current mixin is determined by looking at the type of **this**. Second, the **wcast**$^{\mathsf{m}}$ rule strips out the **view** part of the expression and delegates all work to the subsumption rules. Third, the **sub**$^{\mathsf{m}}$ rule for subsumption inserts a **view** operator to make subsumption coercions explicit.

4.3 MixedJava Evaluation

The operational semantics for MixedJava differs substantially from that of ClassicJava. The rewriting semantics of the latter relies on the uniqueness of each method name in the chain of **class**es associated with an object. This uniqueness is not guaranteed for chains of mixins. Specifically, a composition m_1 **compose** m_2 contains two methods named x if both m_1 and m_2 declare x and m_1's inheritance interface does not contain x. Both x methods are accessible in an instance of the composite mixin since the object can be viewed specifically as an instance of m_1 or m_2.

$\vdash_{\mathsf{p}}$

$$\frac{\begin{array}{l}\textsc{MixinsOnce}(P) \quad \textsc{MethodOncePerMixin}(P) \quad \textsc{InterfacesOnce}(P) \quad \textsc{CompleteMixins}(P) \\ \textsc{WellFoundedMixins}(P) \quad \textsc{CompleteInterfaces}(P) \quad \textsc{WellFoundedInterfaces}(P) \quad \textsc{MixinFieldsOK}(P) \\ \textsc{MixinMethodsOK}(P) \quad \textsc{InterfaceMethodsOK}(P) \quad \textsc{InterfacesAbstract}(P) \quad \textsc{NoAbstractMixins}(P) \\ \textsc{MixinsImplementAll}(P) \quad P \vdash_{\mathsf{d}} defn_j \Rightarrow defn'_j \text{ for } j \in [1,n] \quad P, [\,] \vdash_{\mathsf{e}} e \Rightarrow e' : t \\ \text{where } P = defn_1 \ \ldots \ defn_n \ e\end{array}}{\vdash_{\mathsf{p}} defn_1 \ \ldots \ defn_n \ e \Rightarrow defn'_1 \ \ldots \ defn'_n \ e' : t}[\mathbf{prog}^{\mathsf{m}}]$$

$\vdash_{\mathsf{d}}$

$$\frac{P \vdash_{\mathsf{t}} t_j \text{ for each } j \in [1,n] \qquad P, m \vdash_{\mathsf{m}} meth_k \Rightarrow meth'_k \text{ for each } k \in [1,p]}{\begin{array}{l}P \vdash_{\mathsf{d}} \mathbf{mixin}\ m \cdots \{\ t_1\ fd_1\ \ldots\ t_n\ fd_n \\ \qquad meth_1\ \ldots\ meth_p\ \}\end{array} \Rightarrow \begin{array}{l}\mathbf{mixin}\ m \cdots \{\ t_1\ fd_1\ \ldots\ t_n\ fd_n \\ \qquad meth'_1\ \ldots\ meth'_p\ \}\end{array}}[\mathbf{defn}^{\mathsf{m}}]$$

$\vdash_{\mathsf{e}}$

$$\frac{P \vdash_{\mathsf{t}} m \ m \prec^{\mathsf{m}}_P \mathsf{Empty}}{P, \Gamma \vdash_{\mathsf{e}} \mathbf{new}\ m \Rightarrow \mathbf{new}\ m : m}[\mathbf{new}^{\mathsf{m}}] \qquad \frac{P, \Gamma \vdash_{\mathsf{e}} e \Rightarrow e' : m \qquad \langle m'.fd,\, t\rangle \in_P m}{P, \Gamma \vdash_{\mathsf{e}} e.fd \Rightarrow e' \underline{:\, m'}.fd : t}[\mathbf{get}^{\mathsf{m}}]$$

$$\frac{P, \Gamma \vdash_{\mathsf{e}} e \Rightarrow e' : m \qquad \langle m'.fd,\, t\rangle \in_P m \qquad P, \Gamma \vdash_{\mathsf{s}} e_v \Rightarrow e'_v : t}{P, \Gamma \vdash_{\mathsf{e}} e.fd = e_v \Rightarrow e' \underline{:\, m'}.fd = e'_v : t}[\mathbf{set}^{\mathsf{m}}]$$

$$\frac{\begin{array}{l}P, \Gamma \vdash_{\mathsf{e}} e \Rightarrow e' : t' \ \langle md,\, (t_1 \ldots t_n \longrightarrow t),\, (var_1\ \ldots\ var_n),\, e_b\rangle \in_P t' \\ P, \Gamma \vdash_{\mathsf{s}} e_j \Rightarrow e'_j : t_j \text{ for } j \in [1,n]\end{array}}{P, \Gamma \vdash_{\mathsf{e}} e.md\ (e_1\ \ldots\ e_n) \Rightarrow e'.md\ (e'_1\ \ldots\ e'_n) : t}[\mathbf{call}^{\mathsf{m}}]$$

$$\frac{\begin{array}{l}P, \Gamma \vdash_{\mathsf{e}} \mathbf{this} \Rightarrow \mathbf{this} : m \ m \prec^{\mathsf{m}}_P i \ \langle md,\, (t_1 \ldots t_n \longrightarrow t),\, (var_1\ \ldots\ var_n),\, \mathbf{abstract}\rangle \in_P i \\ P, \Gamma \vdash_{\mathsf{s}} e_j \Rightarrow e'_j : t_j \text{ for } j \in [1,n]\end{array}}{P, \Gamma \vdash_{\mathsf{e}} \mathbf{super}.md(e_1\ \ldots\ e_n) \Rightarrow \mathbf{super}\ \underline{\equiv\ \mathbf{this}}\ .md(e'_1\ \ldots\ e'_n) : t}[\mathbf{super}^{\mathsf{m}}]$$

$$\frac{P, \Gamma \vdash_{\mathsf{s}} e \Rightarrow e' : t}{P, \Gamma \vdash_{\mathsf{e}} \mathbf{view}\ t\ e \Rightarrow e' : t}[\mathbf{wcast}^{\mathsf{m}}] \qquad \frac{P, \Gamma \vdash_{\mathsf{e}} e \Rightarrow e' : t' \qquad t' \not\leq_P t}{P, \Gamma \vdash_{\mathsf{e}} \mathbf{view}\ t\ e \Rightarrow \mathbf{view}\ \underline{t'\ \mathbf{as}}\ t\ e' : t}[\mathbf{ncast}^{\mathsf{m}}]$$

$\vdash_{\mathsf{s}}$

$$\frac{P, \Gamma \vdash_{\mathsf{e}} e \Rightarrow e' : t' \qquad t' \leq_P t}{P, \Gamma \vdash_{\mathsf{s}} e \Rightarrow \mathbf{view}\ \underline{t'\ \mathbf{as}}\ t\ e' : t}[\mathbf{sub}^{\mathsf{m}}]$$

$\vdash_{\mathsf{t}}$

$$\frac{t \in \mathrm{dom}(\prec^{\mathsf{m}}_P) \cup \mathrm{dom}(\doteq^{\mathsf{m}}_P) \cup \mathrm{dom}(\prec^{\mathsf{i}}_P) \cup \{\mathsf{Empty}\}}{P \vdash_{\mathsf{t}} t}[\mathbf{type}^{\mathsf{m}}]$$

Fig. 15. Context-sensitive checks and type elaboration rules for MixedJava

One strategy to avoid the duplication of x is to rename it in m_1 and m_2. At best, this is a global transformation on the program, since x is visible to the entire program as a public method. At worst, renaming triggers an exponential explosion in the size of the program, which occurs when m_1 and m_2 are actually the same mixin m. Since the mixin m represents a type, renaming x in each use of m splits it into two different types, which requires type-splitting at every expression in the program involving m.

Our MixedJava semantics handles the duplication of method names with run-time context information: the current *view* of an object.[6] During evaluation, each reference to an object is bundled with its view of the object, so that values are of the form $\langle object \| view \rangle$. A reference's view can be changed by subsumption, method calls, or explicit casts.

$$
\begin{array}{ll}
e = \ldots \mid \langle object \| M \rangle & \mathsf{E} = [\,] \mid \mathsf{E}\ \underline{:\ m}\ .fd \mid \mathsf{E}\ \underline{:\ m}\ .fd = e \mid v\ \underline{:\ m}\ .fd = \mathsf{E} \\
v = \langle object \| M \rangle \mid \mathsf{null} & \quad \mid \mathsf{E}.md(e \ldots) \mid v.md(v \ldots \mathsf{E}\ e \ldots) \\
& \quad \mid \mathbf{super}\ \underline{\equiv\ v}\ .md(v \ldots \mathsf{E}\ e \ldots) \\
& \quad \mid \mathbf{view}\ \underline{t\ \mathbf{as}\ t}\ \mathsf{E} \mid \mathbf{let}\ var = \mathsf{E}\ \mathbf{in}\ e
\end{array}
$$

$$
\begin{array}{ll}
P \vdash \langle \mathsf{E}[\mathbf{new}\ m], \mathcal{S} \rangle \hookrightarrow \langle \mathsf{E}[\langle object \| M \rangle], \mathcal{S}[object \mapsto \langle m, [M_1.fd_1 \mapsto \mathsf{null}, \ldots M_n.fd_n \mapsto \mathsf{null}] \rangle] \rangle & [new] \\
\quad \text{where } object \notin \mathrm{dom}(\mathcal{S}) \text{ and } m \longrightarrow_P M & \\
\qquad \{M_1.fd_1, \ldots M_n.fd_n\} = \{m'{::}M'.fd \mid M \leq^{\mathsf{M}} m'{::}M' \text{ and } \exists t \text{ s.t. } \langle m'.fd, t \rangle \notin^{\mathsf{m}}_P m'\} & \\
P \vdash \langle \mathsf{E}[\langle object \| M \rangle\ \underline{:\ m'}\ .fd], \mathcal{S} \rangle \hookrightarrow \langle \mathsf{E}[v], \mathcal{S} \rangle & [get] \\
\quad \text{where } \mathcal{S}(object) = \langle m, \mathcal{F} \rangle \text{ and } M/m' \triangleright M' \text{ and } \mathcal{F}(M'.fd) = v & \\
P \vdash \langle \mathsf{E}[\langle object \| M \rangle\ \underline{:\ m'}\ .fd = v], \mathcal{S} \rangle \hookrightarrow \langle \mathsf{E}[v], S'[object \mapsto \langle m, \mathcal{F}[M'.fd \mapsto v] \rangle] \rangle & [set] \\
\quad \text{where } \mathcal{S}(object) = \langle m, \mathcal{F} \rangle \text{ and } M/m' \triangleright M' & \\
P \vdash \langle \mathsf{E}[\langle object \| M \rangle.md(v_1, \ldots v_n)], \mathcal{S} \rangle & [call] \\
\qquad \hookrightarrow \langle \mathsf{E}[e[\langle object \| M' \rangle / \mathbf{this}, v_1/var_1, \ldots v_n/var_n]], \mathcal{S} \rangle & \\
\quad \text{where } \mathcal{S}(object) = \langle m, \mathcal{F} \rangle \text{ and } m \longrightarrow_P M_o & \\
\qquad \text{and } \langle md, T, (var_1 \ldots var_n), e, M' \rangle \in^{\mathsf{rt}}_P M \text{ in } M_o & \\
P \vdash \langle \mathsf{E}[\mathbf{super} \equiv \langle object \| m{::}M \rangle\ .md(v_1, \ldots v_n)], \mathcal{S} \rangle & [super] \\
\qquad \hookrightarrow \langle \mathsf{E}[e[\langle object \| M' \rangle / \mathbf{this}, v_1/var_1, \ldots v_n/var_n]], \mathcal{S} \rangle & \\
\quad \text{where } m \dashv^{\mathsf{m}}_P i \text{ and } M/i \triangleright M'' \text{ and } \langle md, T, (var_1 \ldots var_n), e, M' \rangle \in^{\mathsf{rt}}_P M'' \text{ in } M'' & \\
P \vdash \langle \mathsf{E}[\mathbf{view}\ \underline{t'\ \mathbf{as}\ t}\ \langle object \| M \rangle], \mathcal{S} \rangle \hookrightarrow \langle \mathsf{E}[\langle object \| M' \rangle], \mathcal{S} \rangle & [view] \\
\quad \text{where } t' \trianglelefteq_P t \text{ and } M/t \triangleright M' & \\
P \vdash \langle \mathsf{E}[\mathbf{view}\ \underline{t'\ \mathbf{as}\ t}\ \langle object \| M \rangle], \mathcal{S} \rangle \hookrightarrow \langle \mathsf{E}[\langle object \| M'' \rangle], \mathcal{S} \rangle & [cast] \\
\quad \text{where } t' \not\trianglelefteq_P t \text{ and } \mathcal{S}(object) = \langle m, \mathcal{F} \rangle \text{ and } m \trianglelefteq_P t \text{ and } m \longrightarrow_P M' \text{ and } M'/t \triangleright M'' & \\
P \vdash \langle \mathsf{E}[\mathbf{let}\ var = v\ \mathbf{in}\ e], \mathcal{S} \rangle \hookrightarrow \langle \mathsf{E}[e[v/var]], \mathcal{S} \rangle & [let] \\
 & \\
P \vdash \langle \mathsf{E}[\mathbf{view}\ \underline{t'\ \mathbf{as}\ t}\ \langle object \| M \rangle], \mathcal{S} \rangle \hookrightarrow \langle \text{error: bad cast}, \mathcal{S} \rangle & [xcast] \\
\quad \text{where } t' \not\trianglelefteq_P t \text{ and } \mathcal{S}(object) = \langle m, \mathcal{F} \rangle \text{ and } m \not\trianglelefteq_P t & \\
P \vdash \langle \mathsf{E}[\mathsf{null}\ \underline{:\ m}\ .fd], \mathcal{S} \rangle \hookrightarrow \langle \text{error: dereferenced null}, \mathcal{S} \rangle & [nget] \\
P \vdash \langle \mathsf{E}[\mathsf{null}\ \underline{:\ m}\ .fd = v], \mathcal{S} \rangle \hookrightarrow \langle \text{error: dereferenced null}, \mathcal{S} \rangle & [nset] \\
P \vdash \langle \mathsf{E}[\mathsf{null}.md(v_1, \ldots v_n)], \mathcal{S} \rangle \hookrightarrow \langle \text{error: dereferenced null}, \mathcal{S} \rangle & [ncall]
\end{array}
$$

Fig. 16. Operational semantics for MixedJava

A view is represented as a chain of mixins. This chain is always a tail of the object's full chain of mixins, *i.e.*, the chain of mixins for the object's instantiation type. The tail designates a specific point in the full mixin chain for selecting methods during dynamic dispatch. For example, when an instance of $\mathsf{LockedMagicDoor}^{\mathsf{m}}$ is used as a $\mathsf{Magic}^{\mathsf{m}}$ instance, the view of the object is

$$[\mathsf{NeedsSpell}^{\mathsf{m}}\ \mathsf{Secure}^{\mathsf{m}}\ \mathsf{Door}^{\mathsf{m}}].$$

With this view, a search for the *neededItem* method of the object begins in the $\mathsf{NeedsSpell}^{\mathsf{m}}$ element of the chain.

[6] A view is analogous to a "subobject" in languages with multiple inheritance, but without the complexity of shared superclasses [26].

The first phase of a search for some method x locates the *base declaration* of x, which is the unique non-overriding declaration of x that is visible in the current view. This declaration is found by traversing the view from left to right, using the inheritance interface at each step as a guide for the next step (via the $\propto$ and $\triangleright$ relations). When the search reaches a mixin whose inheritance interface does not include x, the base declaration of x has been found. But the base declaration is not the destination of the dispatch; the destination is an overriding declaration of x for this base that is contained in the object's instantiated mixin. Among the declarations that override this base, the leftmost declaration is selected as the destination. The location of that overriding declaration determines both the method definition that is invoked and the view of the object (*i.e.*, the representation of **this**) within the destination method body. This dispatching algorithm is encoded in the $\in_P^{\mathsf{rt}}$ relation.

The dispatching algorithm explains how $\mathsf{Secure}^{\mathsf{m}}$'s *canOpen* method calls the appropriate *neededItem* method in an instance of $\mathsf{LockedMagicDoor}^{\mathsf{m}}$, sometimes dispatching to the method in $\mathsf{NeedsKey}^{\mathsf{m}}$ and sometimes to the one in $\mathsf{NeedsSpell}^{\mathsf{m}}$. The following example illustrates the essence of dispatching from $\mathsf{Secure}^{\mathsf{m}}$'s *canOpen*:

$$\mathsf{Object}\ \mathit{canOpen}(\mathsf{Secure}^{\mathsf{m}}\ o)\ \{\ \ldots\ o.\mathit{neededItem}()\ \ldots\ \}$$

$$\begin{array}{l}\textbf{let}\ \mathit{door} = \textbf{new}\ \mathsf{LockedMagicDoor}^{\mathsf{m}}\\ \ \textbf{in}\ \mathit{canOpen}(\underline{\textbf{view}\ \mathsf{Secure}^{\mathsf{m}}}\ \textbf{view}\ \mathsf{Locked}^{\mathsf{m}}\ \mathit{door})\ \ldots\\ \qquad \mathit{canOpen}(\underline{\textbf{view}\ \mathsf{Secure}^{\mathsf{m}}}\ \textbf{view}\ \mathsf{Magic}^{\mathsf{m}}\ \mathit{door})\end{array}$$

The **new** $\mathsf{LockedMagicDoor}^{\mathsf{m}}$ expression produces *door* as an $\langle \mathit{object} \| \mathit{view} \rangle$ pair, where *object* is a new object in the store and *view* is (recall Figure 11)

$$[\mathsf{NeedsKey}^{\mathsf{m}}\ \mathsf{Secure}^{\mathsf{m}}\ \mathsf{NeedsSpell}^{\mathsf{m}}\ \mathsf{Secure}^{\mathsf{m}}\ \mathsf{Door}^{\mathsf{m}}].$$

The **view** expressions shift the view part of *door*. Thus, for the first call to *canOpen*, o is replaced by a reference with the view

$$[\mathsf{Secure}^{\mathsf{m}}\ \mathsf{NeedsSpell}^{\mathsf{m}}\ \mathsf{Secure}^{\mathsf{m}}\ \mathsf{Door}^{\mathsf{m}}].$$

In this view, the base declaration of *neededItem* is in the leftmost $\mathsf{Secure}^{\mathsf{m}}$ since *neededItem* is not in the interface extended by $\mathsf{Secure}^{\mathsf{m}}$. The overriding declaration is in $\mathsf{NeedsKey}^{\mathsf{m}}$, which appears to the left of $\mathsf{Secure}^{\mathsf{m}}$ in the instantiated chain and extends an interface that contains *neededItem*.

In contrast, the second call to *canOpen* receives a reference with the view

$$[\mathsf{Secure}^{\mathsf{m}}\ \mathsf{Door}^{\mathsf{m}}].$$

In this view, the base definition of *neededItem* is in the rightmost $\mathsf{Secure}^{\mathsf{m}}$ of the full chain, and it is overridden in $\mathsf{NeedsSpell}^{\mathsf{m}}$. Neither the definition of *neededItem* in $\mathsf{NeedsKey}^{\mathsf{m}}$ nor the one in the leftmost occurrence of $\mathsf{Secure}^{\mathsf{m}}$ is a candidate relative to the given view, because $\mathsf{Secure}^{\mathsf{m}}$ extends an interface that hides *neededItem*.

MixedJava not only differs from ClassicJava with respect to method dispatching, but also in its treatment of **super**. In MixedJava, **super** dispatches are dynamic, since the "supermixin" for a **super** expression is not statically known. The **super** dispatch for mixins is implemented like regular dispatches with the $\in^{\sharp}_{P}$ relation, but using a tail of the current view in place of both the instantiation and view chains; this ensures that a method is selected from the leftmost mixin that follows the current view.

Figure 16 contains the complete operational semantics for MixedJava as a rewriting system on expression-store pairs, like the class semantics described in Section 2.3. In this semantics, an *object* in the store is tagged with a mixin instead of a class, and the values are null and $\langle object \| view \rangle$ pairs.

4.4 MixedJava Soundness

The type soundness theorem for MixedJava is *mutatis mutandis* the same as the soundness theorem for ClassicJava as described in Section 2.4. To prove the soundness theorem, we introduce a conservative extension, MixedJava′, which is defined by revising some of the MixedJava relations (see Figure 17).

In the extended language, the subtype relation is used directly for the "viewable as" relation without eliminating ambiguities. Thus, MixedJava′ allows coercions and method calls that are rejected as ambiguous in MixedJava. This makes MixedJava′ less suitable as a programming language, but simplifies the proof of a type soundness theorem. The soundness theorem for MixedJava′ applies to MixedJava by the following two lemmas:

1. Every MixedJava program is a MixedJava′ program.
2. $P \vdash \langle e, \mathcal{S} \rangle \hookrightarrow \langle e', \mathcal{S}' \rangle$ in MixedJava
   $\Rightarrow P \vdash \langle e, \mathcal{S} \rangle \hookrightarrow \langle e', \mathcal{S}' \rangle$ in MixedJava′.

The proof of the soundness theorem is divided into two parts: we first sketch the soundness of MixedJava′, then show why this result applies to MixedJava.

Type Soundness of MixedJava′. To prove the soundness of MixedJava′, we must first update the type of the environment and the environment-store consistency relation ($\vdash_{\sigma}$) to reflect the differences between ClassicJava and MixedJava′. In MixedJava′, the environment Γ maps $\langle object || M \rangle$ pairs to the mixin type M. The updated consistency relation is defined as follows:

Definition 11 (Environment-Store Consistency).

$$
\begin{array}{ll}
 & P,\Gamma \vdash_{\sigma} \mathcal{S} \\
 & \quad \Leftrightarrow (\mathcal{S}(object) = \langle m, \mathcal{F} \rangle \\
\Sigma_1\text{:} & \qquad \Rightarrow m \leq_P \Gamma(object) \\
\Sigma_2\text{:} & \qquad\quad \text{and } \mathrm{dom}(\mathcal{F}) = \{m'::M'.fd \mid |m| \leq^{\mathsf{M}} m'::M' \text{ and} \\
 & \qquad\qquad\qquad\qquad \exists\ t\ \langle m'.fd, t \rangle \Subset^{\mathsf{m}}_{P} m'\} \\
\Sigma_3\text{:} & \qquad\quad \text{and } \mathrm{rng}(\mathcal{F}) \subseteq \mathrm{dom}(\mathcal{S}) \cup \{\mathsf{null}\} \\
\Sigma_4\text{:} & \qquad\quad \text{and } (\mathcal{F}(m'::M'.fd) = object' \text{ and}
\end{array}
$$

$\leq_P$ Type is a subtype
Extended for views: $M \leq_P m \Leftrightarrow M$ contains m's sequence;
$M \leq_P i \Leftrightarrow M$ contains an m s.t. $m \prec_P^{\mathsf{m}} i$
$\trianglelefteq_P$ Type is viewable as another type $\trianglelefteq_P \equiv \leq_P$
$\in_P$ Field or method is contained in a type
Choose the leftmost field/method instance
$\bullet/\bullet \triangleright \bullet$ Mixin selects a view in a chain
Choose the leftmost instance in the chain
$\bullet \in_P^{\mathsf{rt}} \bullet$ in $\bullet$ Method and view is selected by a view in a chain
Choose the minimum view with a method

Fig. 17. Revised relations for MIXEDJAVA′

$$|m| \leq^{\mathsf{M}} m'::M' \text{ and } \langle m'.fd,t\rangle \Subset_P^{\mathsf{m}} m') \Rightarrow ((\mathcal{S}(object') = \langle m'',\mathcal{F}'\rangle) \Rightarrow m'' \leq_P t))$$

Σ_5: and $\langle object||\_\_\rangle \in \mathrm{dom}(\Gamma) \Rightarrow object \in \mathrm{dom}(\mathcal{S})$

Σ_6: and $object \in \mathrm{dom}(\mathcal{S}) \Rightarrow \langle object||\_\_\rangle \in \mathrm{dom}(\Gamma)$.

The statements of the theorems and lemmata remain unchanged, but the proofs must be adjusted for differences between the two languages. We show how the subject reduction lemma is updated; the remaining proofs change along similar lines.

To prove the type soundness of MIXEDJAVA′, we must establish that field accesses and method invocations that have passed the type-checker will not fail at run-time. The salient differences in the proof of the subject reduction lemma are:

Case [*get*]. The typing rules show that $P,\Gamma \vdash_{\underline{\mathsf{e}}} \langle object||M\rangle \underline{: m'}.fd : t_1$ where $\langle m'.fd,t_1\rangle \in_P \Gamma(\langle object||M\rangle)$. By Σ_2, *object* has the field $m'.fd$. The rest of the proof follows as for CLASSICJAVA.

Case [*call*]. $\Gamma(\langle object||M\rangle) = M$ combined with [**call**$^{\mathsf{m}}$] shows that the method is in M. The search algorithm seeks out the base class of the method definition, and then the leftmost definition of the method in the instantiated mixin. Since the search algorithm ($\bullet \in_P^{\mathsf{rt}} \bullet$ in $\bullet$) follows interfaces in both directions, we know that the method must exist. Further, both the "downward" and "upward" searches are type-preserving, since method overriding must preserve type (by MIXINMETHODSOK). Thus, the invoked method must exist and must have the same type. The rest of the proof is similar to that for CLASSICJAVA.

The proof for the remaining language features is similar to the corresponding proofs for CLASSICJAVA.

Relationship Between MIXEDJAVA **and** MIXEDJAVA′**.** Since the revised relations for MIXEDJAVA′ are conservative extensions of those for MIXEDJAVA, it is easy to see that every MIXEDJAVA program is also a MIXEDJAVA′ program.

What remains to be shown is that for programs common to both languages, their evaluators produce analogous configurations for each reduction step.

The crucial difference between the languages is, for a given expression, which field or method is chosen by the run-time system of each language. Whereas in MixedJava the choice is unique (this is ensured by the "viewable as" relation, $\trianglelefteq_P$), MixedJava$'$ allows implicit and explicit views that can result in ambiguity, and then chooses the leftmost entity (in the linearization) from the set of options. These differences are captured in the $\in_P$, $\bullet/\bullet \propto \bullet$ and $\bullet \in_P^{\mathsf{rt}} \bullet$ in $\bullet$ relations.

Since we are only concerned with programs common to the two languages, we can ignore programs that select views that result in ambiguity. In the remaining programs there is only one field or method to be picked at each stage, which is also the leftmost choice. Hence the two evaluators coincide by making the same choices. As a result, they compute the same answers, and can be used interchangeably for programs common to the two languages. This establishes that the type soundness of MixedJava$'$ applies to MixedJava.

4.5 Implementation Considerations

The MixedJava semantics is formulated at a high level, leaving open the question of how to implement mixins efficiently. Common techniques for implementing classes can be applied to mixins, but two properties of mixins require new implementation strategies. First, each object reference must carry a view of the object. This can be implemented using double-wide references, one half for the object pointer and the other half for the current view. Second, method invocation depends on the current view as well as the instantiation mixin of an object, as reflected in the $\in\!\!\!\!\!\sharp$ relation. Nevertheless, this relation determines a static, per-mixin method table that is analogous to the virtual method tables typically generated for classes.

The overall cost of using mixins instead of classes is equivalent to the cost of using interface-typed references instead of class-typed references. The justification for this cost is that mixins are used to implement parts of a program that cannot be easily expressed using classes. In a language that provides both classes and mixins, portions of the program that do not use mixins do not incur any extra overhead.

4.6 Related Work on Mixins

Mixins first appeared as a CLOS programming pattern [20,21]. Unfortunately, the original linearization algorithm for CLOS's multiple inheritance breaks the encapsulation of class definitions [10], which makes it difficult to use CLOS for proper mixin programming. The CommonObjects [27] dialect of CLOS supports multiple inheritance without breaking encapsulation, but the language does not provide simple composition operators for mixins.

Bracha has investigated the use of "mixin modules" as a general language for expressing inheritance and overriding in objects [5,6,7]. His system is based on earlier work by Cook [8]; its underlying semantics was recently reformulated in

categorical terms by Ancona and Zucca [4]. Bracha's system gives the programmer a mechanism for defining *modules* (**class**es, in our sense) as a collection of *attributes* (methods). Modules can be combined into new modules through various merging operators. Roughly speaking, these operators provide an assembly language for expressing class-to-class functions and, as such, permit programmers to construct mixins. However, this language forces the programmer to resolve attribute name conflicts manually and to specify attribute overriding explicitly at a mixin merge site. As a result, the programmer is faced with the same problem as in Common Lisp, *i.e.*, the low-level management of details. In contrast, our system provides a language to specify both the content of a mixin *and* its interaction with other mixins for mixin compositions. The latter gives each mixin an explicit role in the construction of programs so that only sensible mixin compositions are allowed. It distinguishes method overriding from accidental name collisions and thus permits the system to resolve name collisions automatically in a natural manner.

5 Conclusion

We have presented a programming language of mixins that relies on the same intuition as single inheritance classes. Indeed, a mixin declaration in our language hardly differs from a class declaration since, from the programmer's local perspective, there is little difference between knowing the properties of a superclass as described by an interface and knowing the exact implementation of a superclass. However, from the programmer's global perspective, mixins free each collection of field and method extensions from the tyranny of a single superclass, enabling new abstractions and increasing the re-use potential of code.

While using mixins is inherently more expensive than using classes (because mixins enforce the distinction between implementation inheritance and subtyping), the cost is reasonable and offset by gains in code re-use. Future work on mixins must focus on exploring compilation strategies that lower the cost of mixins, and on studying how designers can exploit mixins to construct better design patterns.

Acknowledgements: Thanks to Corky Cartwright, Robby Findler, Cormac Flanagan, and Dan Friedman for their comments on early drafts of this paper.

References

1. Abadi, M., and Cardelli, L. A theory of primitive objects — untyped and first-order systems. In *Theoretical Aspects of Computer Software*, M. Hagiya and J. C. Mitchell, Eds., vol. 789 of *LNCS*. Springer-Verlag, Apr. 1994, pp. 296–320.
2. Abadi, M., and Cardelli, L. A theory of primitive objects: second-order systems. In *Proc. European Symposium on Programming* (New York, N.Y., 1994), D. Sannella, Ed., Lecture Notes in Computer Science 788, Springer Verlag, pp. 1–25.

3. Abadi, M., and Cardelli, L. An imperative object calculus. In *TAPSOFT'95: Theory and Practice of Software Development* (May 1995), P. D. Mosses, M. Nielsen, and M. I. Schwartzbach, Eds., Lecture Notes in Computer Science 915, Springer-Verlag, pp. 471–485.
4. Ancona, D., and Zucca, E. An algebraic approach to mixins and modularity. In *Proc. Conference on Algebraic and Logic Programming* (Berlin, 1996), M. Hanus and M. Rodríguez-Artalejo, Eds., Lecture Notes in Computer Science 1139, Springer Verlag, pp. 179–193.
5. Bracha, G. *The Programming Language Jigsaw: Mixins, Modularity and Multiple Inheritance*. Ph.D. thesis, Dept. of Computer Science, University of Utah, Mar. 1992.
6. Bracha, G., and Cook, W. Mixin-based inheritance. In *Proc. Joint ACM Conf. on Object-Oriented Programming, Systems, Languages and Applications and the European Conference on Object-Oriented Programming* (Oct. 1990).
7. Bracha, G., and Lindstrom, G. Modularity meets inheritance. In *Proc. IEEE Computer Society International Conference on Computer Languages* (Washington, DC, Apr. 1992), IEEE Computer Society, pp. 282–290.
8. Cook, W. R. *A Denotational Semantics of Inheritance*. Ph.D. thesis, Department of Computer Science, Brown University, Providence, RI, May 1989.
9. Drossopolou, S., and Eisenbach, S. Java is typesafe – probably. In *Proc. European Conference on Object Oriented Programming* (June 1997).
10. Ducournau, R., Habib, M., Huchard, M., and Mugnier, M. L. Monotonic conflict resolution mechanisms for inheritance. In *Proc. Conference on Object-Oriented Programming Systems, Languages, and Applications* (Oct. 1992), pp. 16–24.
11. Eifrig, J., Smith, S., Trifonov, V., and Zwarico, A. Application of OOP type theory: State, decidability, integration. In *Proc. Conference on Object-Oriented Programming Systems, Languages, and Applications* (Oct. 1994), pp. 16–30.
12. Felleisen, M. Programming languages and lambda calculi. URL: `www.cs.rice.edu/~matthias/411web/mono.ps`.
13. Felleisen, M., and Hieb, R. The revised report on the syntactic theories of sequential control and state. Tech. Rep. 100, Rice University, June 1989. *Theoretical Computer Science*, volume 102, 1992, pp. 235–271.
14. Findler, R. B., Flanagan, C., Flatt, M., Krishnamurthi, S., and Felleisen, M. DrScheme: A pedagogic programming environment for Scheme. In *Proc. International Symposium on Programming Languages: Implementations, Logics, and Programs* (1997), pp. 369–388.
15. Flatt, M. PLT MzScheme: Language manual. Tech. Rep. TR97-280, Rice University, 1997.
16. Gamma, E., Helm, R., Johnson, R., and Vlissides, J. *Design Patterns: Elements of Reusable Object-Oriented Software*. Addison Wesley, Massachusetts, 1994.
17. Gosling, J., Joy, B., and Steele, G. *The Java Language Specification*. The Java Series. Addison-Wesley, Reading, MA, USA, June 1996.
18. Harper, R., and Stone, C. A type-theoretic semantics for Standard ML 1996. Submitted for publication, 1997.
19. Kamin, S. Inheritance in smalltalk-80: a denotational definition. In *Proc. ACM Symposium on Principles of Programming Languages* (Jan. 1988).
20. Kessler, R. R. *LISP, Objects, and Symbolic Programming*. Scott, Foresman and Company, Glenview, IL, USA, 1988.
21. Koschmann, T. *The Common LISP Companion*. John Wiley and Sons, New York, N.Y., 1990.

22. Mason, I. A., and Talcott, C. L. Reasoning about object systems in VTLoE. *International Journal of Foundations of Computer Science 6*, 3 (Sept. 1995), 265–298.
23. Palsberg, J., and Schwartzbach, M. I. *Object-oriented Type Systems.* John Wiley & Sons, 1994.
24. Reddy, U. S. Objects as closures: Abstract semantics of object oriented languages. In *Proc. Conference on Lisp and Functional Programming* (July 1988), pp. 289–297.
25. Rémy, D. Programming objects with ML-ART: An extension to ML with abstract and record types. In *Theoretical Aspects of Computer Software* (New York, N.Y., Apr. 1994), M. Hagiya and J. C. Mitchell, Eds., Lecture Notes in Computer Science 789, Springer-Verlag, pp. 321–346.
26. Rossie, J. G., Friedman, D. P., and Wand, M. Modeling subobject-based inheritance. In *Proc. European Conference on Object-Oriented Programming* (Berlin, Heidelberg, and New York, July 1996), P. Cointe, Ed., Lecture Notes in Computer Science 1098, Springer-Verlag, pp. 248–274.
27. Snyder, A. Inheritance and the development of encapsulated software components. In *Research Directions in Object-Oriented Programming.* MIT Press, 1987, pp. 165–188.
28. Syme, D. Proving Java type soundness. Tech. Rep. 427, University of Cambridge, July 1997.
29. Wright, A., and Felleisen, M. A syntactic approach to type soundness. Tech. Rep. 160, Rice University, 1991. *Information and Computation*, volume 115(1), 1994, pp. 38–94.

A Formal Specification of Java™ Virtual Machine Instructions for Objects, Methods and Subroutines

Zhenyu Qian

Bremen Institute for Safe Systems (BISS), FB3 Informatik
Universität Bremen, D-28334 Bremen, Germany

Abstract. In this chapter we formally specify a subset of Java Virtual Machine (JVM) instructions for objects, methods and subroutines based on the official JVM Specification, the official Java Language Specification and Sun's JDK 1.1.4 implementation of the JVM. Our formal specification describes the runtime behaviors of the instructions in relevant memory areas as state transitions and most structural and linking constraints on the instructions as a static typing system. The typing system includes a core of the Bytecode Verifier and resembles data-flow analysis. We state some properties based on our formal specification and sketch the proofs. One of these properties is that if a JVM program is statically well-typed with respect to the typing system, then the runtime data of the program will be type-correct. Our formal specification clarifies some ambiguities and incompleteness and removes some (in our view) unnecessary restrictions in the description of the official JVM Specification.

1 Introduction

The Java Virtual Machine (JVM) is a platform-independent abstract computing machine containing an instruction set and running on various memory areas. The JVM is typically used as an intermediate machine in the implementation of the programming language `Java`. The official JVM Specification by Lindholm and Yellin [10] (OJVMS) defines the syntax of the instructions and describes the semantics of the instructions in related memory areas.

This chapter specifies a subset of the instructions for objects, methods and subroutines by giving a formal semantics to them. The formal specification is based on the OJVMS, Sun's JDK 1.1.4 implementation of the JVM, in particular, the Bytecode Verifier, and the official Java Language Specification by Gosling, Joy and Steele [8] (OJLS). The formal specification provides a foundation for exposing the behaviors of the subset of the JVM. Since programs of the instructions in the JVM can be used directly over the Web, our formal specification defines parts of the security of internet programming in `Java`.

The formal specification considers the following essential instructions: the load and store instructions for objects and integers, the object creation instruction, one operand stack management instruction, several control transfer instruc-

Jim Alves-Foss (Ed.): Formal Syntax and Semantics of Java, LNCS 1523, pp. 271–311, 1999.

tions, all method invocation instructions, several return instructions, and the `jsr` and `ret` instructions for implementing `finally`-clauses.

Many features in the JVM are not considered in this chapter. They are multi-threads, arrays, primitive types other than type *int*, two-word wide data, class initialization method `<clinit>`, access control modifiers, exception handlings, native methods, `lookupswitch`, `tableswitch`, `wide`, runtime exceptions, memory organization, the overflow and underflow of the operand stack, the legality of accesses of local variables, the `class` file format in details, constant pool resolution in details and the difference between "static" and "link time". We assume that all classes have been loaded by a single class loader. Due to space constraints we only very briefly sketch all proofs in this chapter.

The paper [12] considers a larger subset of JVM instructions, in particular, those for exception handling. In addition, it contains the proofs.

The main ideas of our approach are as follows:

- We formalize an operational semantics of the instructions by defining each instruction as a state transition.
- At the same time we formulate a static typing system. Based on the typing rules in the system, one may try to derive a static type for each memory location such that the static type covers the types of all runtime data possibly held by the memory location. The typing system characterizes aspects of the data-flow analysis (see e.g. [1]).
- Our formal specification consists of the state transition machine and the typing system. The state transition machine is defined only for programs, where static types for all memory locations are derivable with respect to the typing system. Practically, the typing system includes a core of the Bytecode Verifier.
- We finally state some properties of the formal specification. In particular, we state that if the type inference system can be successful, then the runtime data are guaranteed to be type-correct.

To a large extent, our formal specification follows the OJVMS. However, some extensions and changes of the semantics are necessary and desirable. Four of them are as follows:

- The OJVMS (page 130) requires that the static type of an operand stack entry or a local variable should be the least upper bound of the types of all possible runtime data in it, and the least upper bound should be one JVM type. The problem is, however, that the subtyping relation on interfaces allows multiple inheritance and thus two interfaces need not have one least common superinterface. Our solution is to allow a set of interfaces (and classes) to be a static type of an operand stack entry or a local variable.
- The OJVMS (page 132) uses a special type indicating that an object is *new*, i.e. it has been created by the instruction `new` but not yet initialized by an instance initialization method. We introduce two kinds of special types indicating two different stages of object initialization in the specification: one indicates that the object is uninitialized; the other indicates that the

object is being initialized by an instance initialization method, but has not yet encountered the invocation of another instance initialization method. We distinguish between these two stages because the objects at different stages should be dealt with differently.
- The OJVMS introduces a concept of *subroutines*: a `jsr` instruction jumps to or calls a subroutine and a `ret` instruction *returns* from a subroutine. The mechanism of subroutines is based on the correct use of return addresses. The OJVMS defines a new primitive type `returnAddress` indicating that a value is a return address. For the formal specification we refine type `returnAddress` into a family of special types, called *subroutine types*, where a value of a subroutine type is the address of a `jsr` instruction calling a subroutine and thus can be used to compute the return address of the subroutine. As we will see, subroutine types are crucial in our specification of constraints on `jsr` and `ret` instructions.
- The OJVMS does not clearly distinguish between types for memory locations and types for runtime data. Our formal specification clearly distinguish between static types for memory locations and types (or tags) of runtime data. Therefore, we can formally discuss the type safety property of runtime data in the execution.

In this chapter we use the following notations.

We use the notation $\overline{\alpha_n}$ to denote n syntactical objects $\alpha_1, \cdots, \alpha_n$, the notation $\{\cdots\}$ a set and define $size(\{\overline{\alpha_n}\}) := n$.

We use $\{\overline{\alpha_n \mapsto \alpha'_n}\}$, where $\alpha_i \neq \alpha_j$ hold for all i, j with $0 \leq i, j \leq n$ and $i \neq j$, to denote a mapping, where the mapping of each α_i is α'_i, and the mapping of every other element will be defined in each concrete case. In fact, in each concrete case, the mapping of every other element will always be either the element itself, or a special value *failure*, or not explicitly defined because it is never used. We define $\mathcal{D}om(\{\overline{\alpha_n \mapsto \alpha'_n}\}) := \{\overline{\alpha_n}\}$. For a mapping θ, we use $\theta(\alpha)$ to denote the result of the mapping for α, and write $\theta[\alpha \mapsto \alpha']$ for the mapping that is equal to θ except it maps α to α'. For a set D, we define

$$\theta_{|D} := \{\alpha \mapsto \theta(\alpha) \mid \alpha \in \mathcal{D}om(\theta) \cap D\}$$
$$\theta_{|-D} := \{\alpha \mapsto \theta(\alpha) \mid \alpha \in \mathcal{D}om(\theta) - D\}$$

A *list* $[\alpha_0 \cdots, \alpha_n]$ with $n \geq -1$ is a special mapping $\{i \mapsto \alpha_i \mid 0 \leq i \leq n\}$. For any list lis, we define $lis + \alpha := lis[size(lis) \mapsto \alpha]$.

2 Related Work

Stata and Abadi proposed a type system for a set of instructions focusing on subroutines and proved the soundness [15]. Since they considered only a few instructions, they could provide lengthy proofs and clarify several key semantic issues about subroutines. Freund and Mitchell made a significant extension of Stata and Abadi's type system by considering object initialization [5,6], and in

doing this, discovered a bug in Sun's implementation of the bytecode verifier, which allows a program to use an object before it has been initialized. To fix the bug, they wrote a typing rule that ensures that at no time during program execution, there may be more than one uninitialized object that is created by the same `new` instruction and is usable.

After realizing the bug discovered Freund and Mitchell, we detected that an early version of the current paper contains the same bug. Except for this point, the results of the current paper are independent of those by Stata, Abadi, Freund and Mitchell. There are several differences between our approach and theirs. First, we follow the constraint-solving framework and use typing rules to generate constraints that define all legal types. Second, we consider more JVM instructions and more details. Two examples are that our approach allows an inner subroutine to return directly to an outer `jsr` instruction in nested subroutines, whereas their approach does not, and that upon a subroutine return, our approach assigns a type to a local variable using the information on whether the local variable is modified in the subroutine, whereas their approach does not consider the case.

Cohen described a formal model of a subset of the JVM, called *defensive JVM* (dJVM), where runtime checks are used to assure type-safe execution [2]. Our approach is different in that we design a static type inference system, which assures that statically well-typed programs do not have runtime type errors. In addition, the current dJVM does not consider subroutines, whereas our specification does.

Goldberg gave a formal specification of bytecode verification [7]. Compared with our work, he considered array types, but not subroutines. In addition, his formal specification is a dataflow analysis and thus closer to the implementation.

Hagiya presented another type system for subroutines [9]. One of the interesting points in his approach is to introduce a mechanism to distinguish the so-called "used" from the "unused" data in a subroutine. His idea is to use a kind of special types indicating that a certain memory location in a subroutine always has the same content as a memory location at a call to the subroutine.

The Kimera project is quite successful in testing some running bytecode verifiers and detecting some flaws [14]. In general, testing is often based on a precise specification. Thus a formal specification may be useful for testing.

Dean [3] studied a formal model relating static typing and dynamic linking and proved the safety of dynamic linking with respect to static typing. As mentioned before, our formal specification does not consider the issue between static typing and dynamic linking.

Our formal specification considers only one single class loader. Saraswat studied static type-(un)safety in `Java` in the presence of more than one class loader [13] .

Although the JVM uses some structures of the `Java` language, our type system for the JVM resembles data-flow analysis and thus is quite different from a formal specification of a type system for `Java` in e.g. [4,11,16].

3 JVM Programs, Methods, Data Areas, and Frames

According to the OJVMS, a byte is 8 bits, and a word is an abstract size that is larger than, among others, a byte. One-byte-wide data build instructions, whereas one-word wide data represent runtime data. We use *byt* to range over all one-byte data and *wrd* over all one-word wide data.

The OJVMS still allows two-word wide integers. But, as mentioned before, we do not consider two-word wide data in this chapter for simplicity.

A *(JVM) program* in this chapter is defined to contain a set of *methods*. We assume that each method has a unique *method code reference*. We use *cod* to range over all method code references. An *address* is a pair (cod, off), where *off* is a one-word wide datum, called a *byte offset*. For any address (cod, off) and another byte offset off', we define $(cod, off) + off' := (cod, off + off)$. An instruction may be longer than one byte. A *program point*, denoted by *pp*, is the starting address of an instruction. Since we do not consider multi-threads, we assume that there is just one *program count register*, which contains the current program point.

As mentioned in the introduction to this chapter, the program point of a `jsr` instruction may be used in computing the returning program point for a subroutine. In fact, it is the byte offset of the program point, not the program point itself, that may be used, since, as we will see later, a `ret` instruction, which uses the program point of a `jsr` instruction, is always in the same method as the `jsr` instruction. Thus we may talk about *the byte offset* of a `jsr` in the rest of this chapter.

We consider an arbitrary but fixed program *Prg*. Note that the methods in a program may stem from different `class` files. A method in *Prg* consists of all instructions in *Prg* whose program points contain the same given method code reference. We use *Mth* to denote an arbitrary but fixed method in *Prg*.

We use $allPP(Prg)$ and $allPP(Mth)$ to denote the sets of all program points in *Prg* and *Mth*, respectively. We assume that $allPP(Mth)$ always contains one unique element of the form $(\_, 0)$. Intuitively, it is the starting program point of the method.

We define that the function $offset(byt1, byt2)$ yields $(byt1 * (2^8)) + byt2$ if it is a one-word wide value, a failure otherwise.

In our specification an *object reference* is formally a one-word wide datum. We use *obj* to range over all object references. Furthermore, we use *null* to denote a special object reference.

Following the OJVMS, we formally specify *int* as the primitive type of all one-word wide integers and use *val* to range over these integers.

We use *cnam*, *inam*, *mnam* and *fnam* to range over names of classes, interfaces, methods and fields, respectively. For our formal specification we require that *fnam* is always a qualified name.

A *record* is formally a mapping of the form $\{\overline{fnam_n \mapsto wrd_n}\}$, which maps all elements other than $\overline{fnam_n}$ to a special value *failure*. We use *rec* to range over all records.

The JVM has a *heap*, from which memory for all objects is allocated. Formally, a *heap state* is defined as a mapping of the form $\{\overline{obj_n \mapsto rec_n}\}$, which maps all elements other than $\overline{obj}_n$ to a special value *failure*. We use *hp* to range over all heap states.

A *frame* is created each time a method is invoked, which contains a *local variable table* and an *operand stack* for the method. A frame is destroyed when the method completes.

A *local variable table state* is a list of the form $[wrd_0, \cdots, wrd_n]$ with $n \geq -1$. We use *lvs* to range over all local variable table states. Each method has a fixed number of local variables.

An *operand stack state* is a list $[wrd_0, \cdots, wrd_n]$ with $n \geq -1$. We use *stk* to range over all operand stack states. Each method has a fixed maximal length of operand stacks.

Note that we need not define formally what a frame is, since no frames are explicitly used in our specification.

Each JVM thread has a *Java stack* to store at least the old current frame and a return address upon a method invocation. When the method invocation completes normally, the old current frame becomes the current frame and the return address becomes the current program point. In this chapter the Java stack contains tuples (lvs, stk, pp), where *lvs* is the old current local variable table state, *stk* the old current operand stack state and *pp* a return address. Since we do not consider multi-threads, we need only to consider one Java stack. We use *jstk* to range over all Java stack states.

A *program state* is a tuple of the form $(pp, jstk, lvs, stk, hp)$. We use *stat* to range over all program states.

4 Static Types

Figure 1 defines all *static types*. In the static analysis, a memory location at a program point may obtain a static type, indicating the types of the runtime data that the memory location may hold at that program point in all executions. For simplicity, we may omit the phrases "at a program point" and "in all executions" in the rest of this chapter.

Reference type set	$\{\overline{ref_n}\}$ $(n > 0)$ where each ref_i is either type *null*, or a class or interface name as in `Java`.
Primitive type	*int*
Subroutine type	$sbr(pp)$ | *invldsbr*
Raw object type	$unin(pp, cnam)$ | $init(cnam)$
Unusable value type	*unusable*

Figure 1: Static types

We introduce the static type *null*. If a memory location may hold nothing more than the special object reference *null*, then the memory location may be given the static type *null*.

The static type *null* and class or interface names in `Java` are called *reference types*. Note that `java.lang.Object` (short: *Object*) is a class name in `Java`.

A nonempty *reference type set* is a static type. Intuitively, if a memory location may hold nothing more than *null* and objects that are of the reference types ref_i for $i = 1, \cdots, n$ but not raw objects (see below), then it may obtain the static type $\{\overline{ref_n}\}$.

It is worth mentioning that the Sun's implementation does not implement the concept of reference type sets in the bytecode verifier.

In our specification, a single reference type is always regarded as identical to the singleton set containing the reference type.

If a memory location may hold nothing more than elements of the primitive type *int*, then it may obtain the static type *int*.

As mentioned before, the byte offset of a `jsr` instruction can be regarded as an element of the subroutine type corresponding to the called subroutine. If a memory location may hold nothing more than some valid byte offsets of `jsr` instructions that call one common subroutine starting at *pp*, then the memory location may obtain a *subroutine type* $sbr(pp)$ as its static type. Note that $sbr(pp) \neq sbr(pp')$ if and only if $pp \neq pp'$.

If a memory location may hold some valid and invalid byte offsets of `jsr` instructions, then the memory location may obtain the static type *invldsbr*.

The forms $unin(pp, cnam)$ and $init(cnam)$ are static types for memory locations holding *raw objects*. More concretely, if a memory location may hold nothing more than objects of the class *cnam* created by one common `new` instruction at a program point *pp*, then the memory location may obtain the static type $unin(pp, cnam)$. If the memory location may hold nothing more than an object that is being currently initialized by an instance initialization method for the class *cnam* and has not encountered another instance initialization method within the current instance initialization method, then the memory location may obtain the static type $init(cnam)$. Note that $unin(pp, cnam) \neq unin(pp', cnam')$ if and only if $pp \neq pp'$ or $cnam \neq cnam'$, $init(cnam) \neq init(cnam')$ if and only if $cnam \neq cnam'$.

Any memory location may obtain the static type *unusable*. In particular, if a memory location may hold runtime data of incompatible types, then it should obtain the static type *unusable*, indicating that the content of the memory location is unusable in practice. For example, if a local variable may hold an object and an element of the type *int*, then our specification will enforce the local variable to obtain the static type *unusable*.

To represent the above intuitive semantics more precisely, we define a partial order $\sqsupseteq$ on static types as the smallest reflexive and transitive relation satisfying

that

$\{\overline{ref_n}\} \sqsupseteq \{\overline{ref_m}\}$ for all n and m with $n \leq m$ and all $ref_i, i = 1, \ldots, m$
$invldsbr \sqsupseteq sbr(pp)$ for all pp
$unusable \sqsupseteq any$ for all static types any

The relation $any \sqsupseteq any'$ is read as "*any covers any′*".

Intuitively, if *any* covers *any′*, then any instruction applicable to a memory location with *any* is also applicable to a memory location with *any′*. Note that the relation implies that, for example, if *any* covers both *int* and a reference type *ref*, then *any* must be *unusable*.

5 Short Notations for Zero or One of Several Static Types

The syntax in Figure 2 means that an identifier on the left of ::= denotes an arbitrary static type or the identifier *void* that either explicitly occurs or is denoted by an identifier on the right of ::=.

Conceptually, the identifier *void* is not a static type. It is just an auxiliary identifier denoting the situation that no static type is present.

For example, *ref* denotes an arbitrary class or interface name or *null*, *tys* denotes an arbitrary reference type set or a primitive type, *notnull_void* denotes an arbitrary class or interface name or *void*, and *any* denotes an arbitrary static type.

Class name	$cnam$	::=	an arbitrary class name
Interface name	$inam$	::=	an arbitrary interface name
Reference type	ref	::=	$cnam \mid inam \mid null$
Primitive type	$prim$	::=	int
Void type	$void$	::=	
Type that is not *null*	$notnull$	::=	$cnam \mid inam \mid prim$
Type	ty	::=	$ref \mid prim$
Reference type set	$refs$	::=	$\{\overline{ref_n}\}$ $(n > 0)$
Type set	tys	::=	$refs \mid prim$
Subroutine type	sbr	::=	$sbr(pp) \mid invldsbr$
Raw object type	raw	::=	$unin(pp, cnam) \mid init(cnam)$
Type or void	$notnull\_void$	::=	$notnull \mid void$
Reference type set or raw object type	$refs\_raw$	::=	$refs \mid raw$
Reference type set, raw object type or subroutine type	$refs\_raw\_sbr$	::=	$refs \mid raw \mid sbr$
Anything	any	::=	$tys \mid raw \mid sbr \mid unusable$

Figure 2: Auxiliary symbols denoting zero or one of several static types

6 Program Point Types and Program Types

In general, there is no guarantee that any `class` file that is asked to be loaded is properly formed. Thus according to the OJVMS, the bytecode verifier should ensure that the `class` file satisfies some constraints. In particular, the bytecode verifier should be able to statically derive a static type for each local variable and operand stack entry at each program point, and ensure that the derived static types satisfy some constraints.

For this purpose, we define a *local variable table type* as a list of the form $[any_0, \cdots, any_n]$ with $n \geq -1$. We use *lvsty* to range over all local variable table types. For $lvsty = [any_0, \cdots, any_n]$ and $lvsty' = [any_0, \cdots, any_m]$, we define that $lvsty \sqsupseteq lvsty'$ holds if and only if $n = m$ and $any_i \sqsupseteq any_i'$ hold for all $i = 0, \ldots, n$.

We define an *operand stack type* as a list of the form $[any_0, \cdots, any_n]$ with $n \geq -1$. We use *stkty* to range over all operand stack types. For $stkty = [any_0, \cdots, any_n]$ and $stkty' = [any_0, \cdots, any_m]$, we define that $stkty \sqsupseteq stkty'$ holds if and only if $n = m$ and $any_i \sqsupseteq any_i'$ hold for all $i = 0, \ldots, n$.

The above definitions that a local variable or an operand stack entry can hold values of arbitrary static types.

To record whether an instance initialization method has been called inside another instance initialization method, we use three *initialization tags*, namely *notInitd*, *Initd* and *unknown*. We use *intag* to range over all initialization tags. A $\sqsupseteq$-relation is defined on these tags as follows:

$$intag \sqsupseteq intag' \text{ if and only if } intag = unknown \text{ or } intag = intag'$$

We define a *program point type* as a tuple $(lvsty, stkty, intag, mod)$ where *mod* will be defined in Section 10.6. We use *ptty* to range over all program point types.

Let $ptty = (lvsty, stkty, intag, mod)$ and $ptty' = (lvsty', stkty', intag', mod')$. The relation $ptty \sqsupseteq ptty'$ holds if and only if $lvsty \sqsupseteq lvsty'$, $stkty \sqsupseteq stkty'$, $intag \sqsupseteq intag'$ and $mod \sqsupseteq mod'$ hold, where the last relation will be defined in Section 10.6.

Intuitively, the relation $ptty \sqsupseteq ptty'$ is used to ensure that any instruction that is applicable to all program states of the program point type *ptty* must be applicable to all program states of the program point type *ptty'*.

For the program *Prg*, a *program type* is a mapping $\{pp \mapsto ptty_{pp} \mid pp \in allPP(Prg)\}$. We use *prgty* to range over all program types. Let *prgty* and *prgty'* be two program types. Then we define that $prgty \sqsupseteq prgty'$ holds if and only if $prgty(pp) \sqsupseteq prgty'(pp)$ holds for all $pp \in allPP(Prg)$. These concepts can also be defined for the fixed method *Mth*.

7 The Reference Type Hierarchy

A reference type hierarchy in the JVM is as in `Java`. Following the OJVMS (§ 2.6.4), we formally define a subtyping relation *widRefConvert* as the smallest

reflexive transitive relation on all reference type sets *refs* satisfying:

$$\begin{aligned}
widRefConvert(cnam, cnam') &\text{ if } cnam \texttt{ extends } cnam' \\
widRefConvert(cnam, inam) &\text{ if } cnam \texttt{ implements } inam \\
widRefConvert(inam, inam') &\text{ if } inam \texttt{ extends } inam' \\
widRefConvert(inam, Object) & \\
widRefConvert(null, ref) & \\
widRefConvert(\{\overline{ref_n}\}, \{\overline{ref'_m}\}) &\text{ if } \forall(1 \le i \le n).\exists(1 \le j \le m). \\
& \quad widRefConvert(ref_i, ref'_j)
\end{aligned}$$

Note that we do not consider array types. We use the relation *diSubcls* to denote the direct subclass relation on classes.

To constrain the types of the actual and formal parameters in a method invocation we define the relation *invoConvert* on all reference type sets and the primitive type *int* as

$$invoConvert := widRefConvert \cup \{(int, int)\}$$

Note that $\{(int, int)\}$ is a degenerate case of the widening primitive conversion in the OJVMS (§ 2.6.2). It suffices for us to have the degenerate case, since we consider only one primitive type *int*.

To constrain the types of the variable and the value in an assignment, we define the relation *assConvert* on all reference type sets and the primitive type *int* as

$$assConvert := invoConvert.$$

The OJVMS requires that *assConvert* extends *invoConvert* by some narrowing primitive conversions for integer constants. We do not consider this difference for simplicity.

Intuitively, if a reference type set contains both a super- and a subtype, then the subtype is redundant. Practically, a Bytecode Verifier could implement elimination of redundant reference types from a reference type set with respect to a subtyping relation as an optimization step.

8 Constant Pool Resolution

According to the OJVMS, each class (or interface) should have a *constant pool* whose entries name entities like classes, interfaces, methods and fields referenced from the code of the class (or interface, respectively) or from other constant pool entries. An individual instruction in the class (or interface, respectively) may carry an index of an entry in the constant pool, and during the execution of this instruction, the JVM is responsible for resolving the entry, i.e. determining a concrete entity from the entry. This process of resolving an entry is called *constant pool resolution*.

For our formal specification we introduce some defined functions, called *resolution functions*, which hide the details of resolution. In fact, except that the

resolution processes should take correct sorts of data as argument and yield correct sorts of data or a failure as result, other details are not important at all for the formal specification and proofs in this chapter. Nevertheless, we still explain the definitions of the resolution functions, in order to give a feeling why the resolution functions here are proper abstractions of the real resolution processes.

A resolution function in this section often takes as parameter two one-byte-wide integers $ind1$ and $ind2$, which build the index $offset(ind1, ind2)$ in a constant pool. In this sense, a resolution function has always a constant pool as an implicit parameter.

The resolution function $cResol(index1, index2)$ yields a class name $cnam$.

For any $cnam$, we define a function

$$allFields(cnam) := \{(fnam, notnull) \mid fnam \text{ and } notnull \text{ are the name and type of a field in the class } cnam\}$$

Note that a field in the class $cnam$ is either directly defined in the class or in a superclass of the class. Since a field name $fnam$ is a qualified name, we need not consider the problem with hiding of fields.

We define a resolution function for a field as

$$fResol(ind1, ind2) := (fnam, cnam, notnull)$$

where $fnam$ is the name of the field, $cnam$ the class containing the field declaration, and $notnull$ the type of the field.

We define a resolution function for a special method as

$$mResolSp(ind1, ind2) := (mnam, cnam, (\overline{ty_n})notnull\_void, cod, nlv)$$

where $mnam$ is the name of the method, $cnam$ the class containing the declaration of the method, $(\overline{ty_n})notnull\_void$ the descriptor of the method, cod the method code and nlv the length of the local variable table in the method.

We define a resolution function for a `static` method as

$$mResolSt(ind1, ind2) := (mnam, cnam, (\overline{ty_n})notnull\_void, cod, nlv).$$

We define a resolution function for an instance method[1] as

$$mResolV(ind1, ind2) := (mnam, cnam, (\overline{ty_n})notnull\_void).$$

But the function $mResolV(ind1, ind2)$ does not yield a method code. For doing this, we need to define another function

$$mSelV(obj, mnam, (\overline{ty_n})notnull\_void) := (cod, nlv)$$

which takes an object obj, and yields the method code cod for the object obj and the length nlv of the local variable table in the method.

[1] Thanks to Gilad Bracha for clarifying comments on the semantics of method dispatch at this point.

We define a resolution function for an interface method as

$$mResolI(ind1, ind2) := (mnam, inam, (\overline{ty_n})notnull\_void)$$

where *inam* is the name of the interface, instead of the class, that contains the declaration of the method. Furthermore, we define

$$mSelI(obj, mnam, (\overline{ty_n})notnull\_void) := (cod, nlv).$$

For convenience we define the auxiliary function

$$mInfo(pp) := (mnam, cnam, (\overline{ty_n})notnull\_void, nlv)$$

where *mnam* is the method containing the *pp*, $(\overline{ty_n})notnull\_void$ the descriptor of the method, *cnam* the class containing the declaration of the method, and *nlv* the number of the local variables in the method.

In order to find out whether a method is an instance or a `static` method, we define the following relations:

- $instMeth(mnam, cnam, (\overline{ty_n})notnull\_void)$ holds if and only the method *mnam* with the signature $(\overline{ty_n})notnull\_void$ is an instance method.
- $statMeth(mnam, cnam, (\overline{ty_n})notnull\_void)$ holds if and only the method *mnam* with the signature $(\overline{ty_n})notnull\_void$ is a static method.

9 Constraint Domain and Constraints

The previous sections have in fact introduced (part of) a constraint domain for our formal specification. Although there are no problems to completely formally define all concepts in a constraint domain, we can only discuss (part of) them informally in this chapter due to the space limit.

First of all, all data, data structures (e.g. local variable table states, operand stack states, program states), static types and type structures (e.g. local variable table types, operand stack types, program point types, pogram types) defined in the previous sections are elements of the constraint domain. These elements are all *sorted*. Informally, every time when we introduce an identifier to range over a kind of data, data structures, static types or type structures, we introduce a sort. We use the introduced identifiers also as names of these sorts. So it is possible for one sort to have several names. For example, the sort *byt* consists of all one-byte wide data, the sort *wrd* all one-word wide data, the sort *pp* all program points, the sort *lvs* all local variable table states, the sort *stk* all operand stack states, the sort *stat* all program states, the sorts *ref*, *refs*, *tys* and $refs\_raw$ corresponding static types, respectively, as defined in Figure 2, the sort *lvsty* all local variable table types, the sort *stkty* all operand stack types and the sort *prgty* all program point types. Standard data or type structures, e.g. sets or lists of some data or types, also build sorts, but not necessarily have a sort name.

There is a subsort relation among the sorts, which corresponds to the subset relation. In particular, Figure 2 defines that if a sort occurs as an alternative on

the right of ::=, then the sort on the left of ::= is a supersort of it. For example, the sort *ref* is a supersort of the sorts *cnam* and *inam* and contains *null*, the sort *prim* contains *int*, the sort *notnull* is a supersort of the sorts *cnam*, *inam* and *prim*, *refs* contains all $\{\overline{ref_n}\}$, where each ref_i is an element of the sort *ref*, etc. Since a singleton reference type set $\{ref\}$ and the reference type *ref* are regarded as the same static type, we define that the sort *refs* is a supersort of the sort *ref*.

For each sort, there are a countable set of variables. In general, the completely capitalized version of a sort name denotes a variable of the sort. For example, *BYT* is a variable of *byt* and *WRD* a variable of *wrd*. For notional simplicity we also introduce the variable *P* for the sort *pp*, *L* for the sort *lvs*, *S* for the sort *stk*, *J* for the sort *jstk*, *H* for the sort *hp*, *LG* for the sort *lvstag*, *SG* for the sort *stktag*, Ξ for the sort *stat*, *LT* for the sort *lvsty*, *ST* for the sort *stkty*, *IT* for the sort *intag*, *M* for *mod*, Π for the sort *ptty* and Φ for the sort *prgty*. We use _ to denote a wildcard variable.

In general, terms are built using variables, constants and functions in the constraint domain. Terms are sorted as usual. A sort of a subsort is always a term of a supersort. Every term has a least sort. We will use the partially capitalized version of a sort name, where only the first letter is changed into a capital letter, to range over all terms of the sort. For example, *Pp*, *Stat* and *Ptty* range over the terms in the sorts *pp*, *stat* and *ptty*, respectively.

A term containing no variables is called *closed*. In fact, each element in the constraint domain is a closed term.

Logical formulas are built as in First-Order Predicate Logic, where predicates take only sorted arguments in the constraint domain. We use q and r to range over all logical formulas.

We use the form $q[\overline{s_n}]$ to denote a logical formula containing the (occurrences of) terms $\overline{s_n}$. If the forms $q[\overline{s_n}]$ and $q[\overline{t_n}]$ occur in the same context (e.g. the same rule), then s_i and t_i are of the same sort for $i = 1, \ldots, n$, and $q[\overline{t_n}]$ is the logical formula obtained from $q[\overline{s_n}]$ by replacing each s_i by t_i for $i = 1, \ldots, n$.

A *substitution* is a finite mapping of the form $\{\overline{X_n \mapsto s_n}\}$, where the sort of each term s_i must be a subsort of the sort of X_i for all $i = 1, \cdots, n$. We consider only *closed* substitutions in this chapter, i.e. where s_i is a closed term for $i = 1, \ldots, n$. We use σ to range over all closed substitutions.

A *constraint* is a logical formula. A set of constraints $\{q_1, \cdots, q_m\}$ represents the logical formula $q_1 \wedge \cdots \wedge q_m \wedge true$.

A constraint q is *satisfied under* a substitution σ if and only if $\sigma(q)$ is closed and holds in the constraint domain. A constraint q is *satisfiable* if there is a substitution, under which the constraint q is satisfied.

In our formal specification, we may define a function f that yields results in a sort a for some arguments and the special value *failure* not in a for all other arguments, and use a term $f(\overline{s_n})$ in a constraint, say $q[f(\overline{s_n})]$, where a term of a is required. Intuitively, this usage always implicitly requires that $f(\overline{s_n})$ should not yield *failure*. Formally, we may always define a new sort a', which is a supersort of a and contains the *failure* as a constant, define the f to have

the result sort a', and replace the constraint $q[f(\overline{s_n})]$ by the constraints $q[X]$ and $X = f(\overline{s_n})$, where X is a new variable of the sort a. The reason why the constraint $X = f(\overline{s_n})$ assures that "$f(\overline{s_n})$ is not equal to *failure*" is that *failure* is not in the sort a and thus $X = failure$ is never satisfiable. (Note that if there are two functions yielding *failure*, then we need to assume that they yield different *failure*'s; otherwise the least sort of the term *failure* may not exist.)

Our formal specification consists of two parts. The first part defines a state transition relation on program states $stat \implies stat'$, read as "*stat* changes into *stat*'". The relation is defined by state transition rules of the following form:

$$\frac{Premises}{\Xi[\overline{s_n}] \implies \Xi[\overline{t_n}]}$$

where *Premises* is a set of constraints. Let

$$Q := \mathcal{FV}(Premises) \cup \mathcal{FV}(\Xi[\overline{s_n}] \implies \Xi[\overline{t_n}]).$$

Then the rule means that if all constraints in *Premises* are satisfied under a substitution σ, then $\sigma(\Xi)[\overline{\sigma(s_n)}] \implies \sigma(\Xi)[\overline{\sigma(t_n)}]$ holds. In the sequel, we may also say that $\Xi[\overline{s_n}]$ changes into $\Xi[\overline{t_n}]$ in the informal discussion for simplicity.

To specify all program types of a program, the following two forms of constraints are particularly important:

$$Prgty(Pp) = Ptty \quad \text{and} \quad Prgty(Pp) \sqsupseteq Ptty$$

The former says that the program point type at Pp in *Prgty* is *Ptty*. The latter says that the program point type at Pp in *Prgty* covers *Ptty*. If a program point Pp can be reached by more than one preceding program point, then it is quite convenient to write a constraint of the latter form to constrain the program point type at Pp.

The type system in our formal specification should introduce constraints on one program type for the method *Mth*. Therefore, we require that all typing rules contain one common program type variable Φ.

In general, a typing rule is in the form:

$$\frac{\boxed{\mathcal{AC}}}{\mathcal{CC}}$$
$$\mathcal{SC}$$

The $\mathcal{AC}$ is a set of logical formulas, called *applicability conditions*, and contains a distinguished constraint $Mth(P) = Instr$. The term *Instr* gives the form of an instruction. Intuitively, $\mathcal{AC}$ is used to determine a program point P, where the rule can be applied. The identifier $\mathcal{CC}$ stands for a set of logical formulas. It contains no logical formulas of the form $\Phi(P) \sqsupseteq Ptty$. Intuitively, $\mathcal{CC}$ constrains $\Phi(P)$. The identifier $\mathcal{SC}$ stands for a set of logical formulas of the form $\Phi(Pp') \sqsupseteq Ptty$, where in most cases Pp' stands for a successor program point.

The reason for us to write a typing rule in the form as above is that a typing rule also suggests an intuitive data-flow analysis step. Roughly speaking, if the

data-flow analysis arrives at a program point Pp satisfying $\mathcal{AC}$, in particular the constraint $Mth(Pp) = Instr$ in $\mathcal{AC}$, and if the program type at Pp satisfies $\mathcal{CC}$, then the program type at each successor program point Pp' should satisfy the corresponding constraint in $\mathcal{SC}$.

Let

$$Q := \mathcal{FV}(\mathcal{AC}) - \{\Phi\}$$
$$Q' := \mathcal{FV}(\mathcal{CC} \cup \mathcal{SC}) - (\{\Phi\} \cup Q)$$

then a typing rule as above formally introduces the constraint

$$\forall Q.(\mathcal{AC} \Rightarrow \exists Q'.(\mathcal{CC} \cup \mathcal{SC}))$$

It is easy to see that the constraint holds if and only if, if $\mathcal{AC}$ is satisfied under a substitution σ with $\mathcal{D}om(\sigma) = Q \cup \{\Phi\}$, then there is a substitution σ' with $\mathcal{D}om(\sigma) = Q \cup Q' \cup \{\Phi\}$ such that $\sigma'_{|Q \cup \{\Phi\}} = \sigma$ and $\sigma'(\mathcal{CC} \cup \mathcal{SC})$ hold.

Let $Constrs_{Mth}$ denote the set of the constraints introduced as above from all typing rules. Then we say that the method *Mth has a program type prgty*, or that a program type *prgty is a program type* of the method *Mth*, if and only if all constraints in $Constrs_{Mth}$ are satisfied under $\{\Phi \mapsto prgty\}$. Note that a program may have more than one program type. For example, a local variable that is not used in a method may be given an arbitrary static type in a program type. A program is said to be *statically well-typed* if and only if it has a program type.

10 The Rules in the Formal Specification

There are constraints that should occur in many rules. We omit the explicit presentation of the following constraints for notational simplicity.

- The $\mathcal{CC}$ in a typing rule always implicitly contains a constraint $Pp \in allPP(Mth)$ for each $\Phi(Pp) \sqsupseteq Ptty$ in the $\mathcal{SC}$. This assures that Pp is always a program point, i.e. a starting address of an instruction.
- In the specification we only consider the instructions for one-word wide data. Thus the rules are all based on the assumption that all data in local variables and the operand stack are one-word wide.

10.1 Load and Store Instructions

The state transitions for loading and storing objects and integers of type *int* are defined by the rules in Figure 3. The `aload` and `iload` instructions load a local variable onto the operand stack. The `astore` and `istore` instructions store a value from the operand stack in a local variable.

The typing rules for load and store instructions are given in Figure 4. We explain rule (T-1) to show some of the tricky points in the formulation of constraints. First, $REFS\_RAW = LT(IND)$ expresses a membership constraint, i.e. that the static type $LT(IND)$ should be in the sort *refs_raw*, since $REFS\_RAW$ can only be instantiated by an element in the sort *refs_raw*. It implies that an

$$\frac{Prg(P) = \texttt{aload}\ IND \text{ or } \texttt{iload}\ IND}{\Xi[P, L, S] \implies \Xi[P + 2, L, S + L(IND)]} \quad \text{(S-1),(S-2)}$$

$$\frac{Prg(P) = \texttt{astore}\ IND \text{ or } \texttt{istore}\ IND}{\Xi[P, L, S + WRD] \implies \Xi[P + 2, L[IND \mapsto WRD], S]} \quad \text{(S-3),(S-4)}$$

Figure 3: The state transitions for load and store instructions

`aload` instruction can load both initialized and uninitialized objects. In addition, rule (T-1) says that the local variable table type at $P + 2$ (i.e. after the instruction) should componentwise cover that at P (before the instruction). The same should also hold for the operand stack type, except that the operand stack type at $P+2$ should be extended by the static type of the IND-th local variable. A similar constraint should also hold on the components M at P and Mod' at $P + 2$. The precise definitions of M and Mod' will be given in Sections 10.6 and 10.7. Note that the variables Φ and LT in the terms $\Phi(P)$ and $LT(IND)$ are not higher-order (i.e. function) variables, since the terms of this form in this chapter can always be regarded as applications of an implicit function *app* on two first-order arguments.

Similar explanations can be given for the other three typing rules. One point that is worth noticing in rule (T-3) is that the variable $REFS\_RAW\_SBR$ can be initiantiated into an element of the sort *sbr*, whereas the variable $REFS\_RAW$ in rule (T-1) cannot. This means that, as required in the OJVMS and implemented in the Sun's implementation, an `astore` instruction can store a (valid or invalid) byte offset, whereas an `aload` instruction cannot load it.

An `aconst_null` instruction loads the reference *null*. Its state transition rule and typing rule are defined in Figure 5.

The state transitions for `getfield` and `putfield` are defined in Figure 6. A `getfield` instruction replaces an object reference at the top of the operand stack by the content of a field of the referenced object.

A `putfield` instruction stores the content at the top of the operand stack into a field of the object referenced by the second top of the operand stack.

The typing rules for `getfield` and `putfield` are given in Figure 7. The sort of the variable $REFS$ in $\Pi[ST + REFS]$ and $\Pi[ST + REFS + TYS]$ assures that the OBJ in Figure 6 really references an object. The constraint $widRefConvert(REFS, CNAM)$ assures that in Figure 6 if $OBJ \in \mathcal{D}om(H)$, then $FNAM \in \mathcal{D}om(H(OBJ))$ holds, i.e. both $H(OBJ)(FNAM)$ and $H(OBJ)[FNAM \mapsto WRD]$ are defined and make sense. But the typing rules do not ensure that the condition $OBJ \in \mathcal{D}om(H)$ in Figure 6 holds, since the OBJ may hold *null* at run time. If $OBJ \notin \mathcal{D}om(H)$ holds, then $H(OBJ)$ yields a *failure*. Thus the *Premises* in both rules in Figure 6 are not satisfiable. In fact, in this case we would need another state transition rule to describe which kind of runtime exception can be thrown. However, as mentioned before, our formal specification does not consider this.

$$\frac{\boxed{Mth(P) = \texttt{aload}\ IND} \quad \Phi(P) = \Pi[LT, ST, M] \quad REFS\_RAW = LT(IND)}{\Phi(P+2) \sqsupseteq \Pi[LT, ST + REFS\_RAW, Mod']} \tag{T-1}$$

$$\frac{\boxed{Mth(P) = \texttt{iload}\ IND} \quad \Phi(P) = \Pi[LT, ST, M] \quad int = LT(IND)}{\Phi(P+2) \sqsupseteq \Pi[LT, ST + int, Mod']} \tag{T-2}$$

$$\frac{\boxed{Mth(P) = \texttt{astore}\ IND} \quad \Phi(P) = \Pi[LT, ST + REFS\_RAW\_SBR, M]}{\Phi(P+2) \sqsupseteq \Pi[LT[IND \mapsto REFS\_RAW\_SBR], ST, Mod']} \tag{T-3}$$

$$\frac{\boxed{Mth(P) = \texttt{istore}\ IND} \quad \Phi(P) = \Pi[LT, ST + int, M]}{\Phi(P+2) \sqsupseteq \Pi[LT[IND \mapsto int], ST, Mod']} \tag{T-4}$$

Figure 4: The typing rules for load and store instructions

$$\frac{Prg(P) = \texttt{aconst\_null}}{\Xi[P, S] \implies \Xi[P+1, S + null]} \tag{S-5}$$

$$\frac{\boxed{Mth(P) = \texttt{aconst\_null}} \quad \Phi(P) = \Pi[ST]}{\Phi(P+1) \sqsupseteq \Pi[ST + null]} \tag{T-5}$$

Figure 5: The state transitions for `aconst_null` and `bipush`

$$\frac{Prg(P) = \texttt{getfield}\ IND1\ IND2 \quad (FNAM, \_, NOTNULL) = fResol(IND1, IND2) \quad WRD = H(OBJ)(FNAM)}{\Xi[P, S + OBJ, H] \implies \Xi[P+3, S + WRD, H]} \tag{S-6}$$

$$\frac{Prg(P) = \texttt{putfield}\ IND1\ IND2 \quad (FNAM, \_, \_) = fResol(IND1, IND2) \quad REC = H(OBJ)[FNAM \mapsto WRD]}{\Xi[P, S + OBJ + WRD, H] \implies \Xi[P+3, S, H[OBJ \mapsto REC]]} \tag{S-7}$$

Figure 6: The state transitions for `getfield` and `putfield`

$$\frac{\boxed{Mth(P) = \texttt{getfield}\ IND1\ IND2} \quad \Phi(P) = \Pi[ST + REFS] \quad (\_, CNAM, NOTNULL) = fResol(IND1, IND2) \quad widRefConvert(REFS, CNAM)}{\Phi(P+3) \sqsupseteq \Pi[ST + NOTNULL]} \tag{T-6}$$

$$\frac{\boxed{Mth(P) = \texttt{putfield}\ IND1\ IND2} \quad \Phi(P) = \Pi[ST + REFS + TYS] \quad (\_, CNAM, NOTNULL) = fResol(IND1, IND2) \quad widRefConvert(REFS, CNAM) \quad assConvert(TYS, NOTNULL)}{\Phi(P+3) \sqsupseteq \Pi[ST]} \tag{T-7}$$

Figure 7: The typing rules for `getfield` and `putfield`

10.2 Object Creation

A `new` instruction creates an object. The state transition and typing rules for the instruction are defined in Figure 8.

$$\frac{Prg(P) = \texttt{new}\ IND1\ IND2 \quad CNAM = cResol(IND1, IND2) \quad OBJ \notin \mathcal{D}om(H) \quad \mathcal{D}om(REC) = allFields(CNAM)}{\Xi[P, S, H] \implies \Xi[P+3, S + OBJ, H[OBJ \mapsto REC]]} \tag{S-8}$$

$$\frac{\boxed{Mth(P) = \texttt{new}\ IND1\ IND2} \quad \Phi(P) = \Pi[LT, ST] \quad CNAM = cResol(IND1, IND2) \quad unin(P, CNAM) \notin LT \quad unin(P, CNAM) \notin ST}{\Phi(P+3) \sqsupseteq \Pi[LT, ST + unin(P, CNAM)]} \tag{T-8}$$

Figure 8: The state transition and the typing rule for `new`

The condition $OBJ \notin \mathcal{D}om(H)$ in rule (S-8) assures that the object reference OBJ is new. Rule (T-8) says that the operand stack type after the instruction covers one with $unin(P, CNAM)$[2] at the top, which indicates that the operand stack may hold an object that has not been initialized by an instance initialization method `<init>`, i.e. an uninitialized object. Indeed, a typing rule that forbids the use of a memory location with a static type of the form $unin(\_, \_)$ forbids the use of an uninitialized object.

[2] The OJVMS mentions such a type but gives no details on how it can be used in the specification.

The constraints $unin(P, CNAM) \notin LT$ and $unin(P, CNAM) \notin ST$ assure that at the program point P, no new object created by the same `new` instruction at P can be used as a new object. This is strictly weaker than to say that there is no new object created by the `new` instruction at P, since a memory location at P is still allowed to hold a new object created by the `new` instruction at P if the memory location has the type *unusable*. For an example, see Section 11.

10.3 Operand Stack Management Instructions

We only give the rules for `dup` in Figure 9. The rules for other instructions are similar.

$$\frac{Prg(P) = \texttt{dup}}{\Xi[P, S + WRD] \implies \Xi[P + 1, S + WRD + WRD]} \quad \text{(S-9)}$$

$$\frac{\boxed{Mth(P) = \texttt{dup}} \qquad \Phi(P) = \Pi[ST + ANY]}{\Phi(P + 1) \sqsupseteq \Pi[ST + ANY + ANY]} \quad \text{(T-9)}$$

Figure 9: The state transition and typing rules for `dup`

10.4 Control Transfer Instructions

$$\frac{\begin{array}{l} Prg(P) = \texttt{if\_acmpeq}\ BYT1\ BYT2 \\ OBJ1 = OBJ2 \Rightarrow P' = P + \mathit{offset}(BYT1, BYT2) \\ OBJ1 \neq OBJ2 \Rightarrow P' = P + 3 \end{array}}{\Xi[P, S + OBJ1 + OBJ2] \implies \Xi[P', S]} \quad \text{(S-10)}$$

$$\frac{\begin{array}{l} Prg(P) = \texttt{if\_icmpeq}\ BYT1\ BYT2 \\ VAL1 = VAL2 \Rightarrow P' = P + \mathit{offset}(BYT1, BYT2) \\ VAL1 \neq VAL2 \Rightarrow P' = P + 3 \end{array}}{\Xi[P, S + VAL1 + VAL2] \implies \Xi[, S]} \quad \text{(S-11)}$$

$$\frac{Prg(P) = \texttt{goto}\ BYT1\ BYT2}{\Xi[P] \implies \Xi[\mathit{offset}(BYT1, BYT2)]} \quad \text{(S-12)}$$

Figure 10: The state transitions for control transfer instructions

All control transfer instructions can be dealt with in a very similar way. We consider only a few control transfer instructions. The state transitions for these instructions are given in Figure 10. They are quite straightforward.

$$\frac{\boxed{Mth(P) = \texttt{if\_acmpeq}\ BYT1\ BYT2} \quad \Phi(P) = \Pi[ST + REFS + REFS']}{\Phi(P + \mathit{offset}(BYT1, BYT2)) \sqsupseteq \Pi[ST] \quad \Phi(P+3) \sqsupseteq \Pi[ST]} \tag{T-10}$$

$$\frac{\boxed{Mth(P) = \texttt{if\_icmpeq}\ BYT1\ BYT2} \quad \Phi(P) = \Pi[ST + int + int]}{\Phi(P + \mathit{offset}(BYT1, BYT2)) \sqsupseteq \Pi[ST] \quad \Phi(P+3) \sqsupseteq \Pi[ST]} \tag{T-11}$$

$$\frac{\boxed{Mth(P) = \texttt{goto}\ BYT1\ BYT2} \quad \Phi(P) = \Pi}{\Phi(P + \mathit{offset}(BYT1, BYT2)) \sqsupseteq \Pi} \tag{T-12}$$

Figure 11: The typing rules for control transfer instructions

The OJVMS requires (page 133) that no uninitialized objects may exist on the operand stack or in a local variable when a control transfer instruction causes a backwards branch. In our specification this requirement is unnecessary, thanks to rule (T-8).

10.5 Method Invocation and Return Instructions

The state transitions for method invocation instructions are defined in Figure 12. We first consider the state transition rule (S-13) for `invokespecial`. Since the instruction is only used to invoke instance instantiation methods `<init>` and private methods, and to perform method invocations via `super`, we use the function *mResolSp*. The state transition says that the execution of the invoked method starts with a program state, in which the operand stack is empty and the local variables hold the object, on which the method is invoked, and all actual arguments. We use the notation lvs^n to denote an arbitrary local variable table state with the length n.

The state transition for `invokevirtual` (or `invokeinterface`) is similar to that for `invokespecial`. The difference is only that the former uses the functions *mResolV* and *mSelV* (or *mResolI* and *mSelI*, respectively) to compute the method code associated with *OBJ*, whereas the latter uses the function *mResolSp* to do the same thing, independent of *OBJ*. Note that the bytes *BYT* and 0 in a `invokeinterface` instruction are useless. They are contained in the instruction for historical reasons.

Invocation of a method leads to the execution of a method code. The typing rule in Figure 13 constrains the program point type at the beginning of a method code. The rule is totally independent of method invocation instructions. The rule says that the method must be a `<init>`, an instance or a `static` method. The static types given for the local variables depend on what kind method it is. In

$$\frac{\begin{array}{l} Prg(P) = \texttt{invokespecial}\ IND1\ IND2 \\ (\_,\_,(\overline{TY_n})NOTNULL\_VOID, COD, NLV) = mResolSp(IND1, IND2) \end{array}}{\begin{array}{l} \Xi[P,J,L,S+OBJ\overline{+WRD_n}] \\ \Longrightarrow\ \Xi[COD, J+(L,S,P+3), lvs^{NLV}[0 \mapsto OBJ, \overline{n \mapsto WRD_n}], []] \end{array}} \quad \text{(S-13)}$$

$$\frac{\begin{array}{l} Prg(P) = \texttt{invokevirtual}\ IND1\ IND2 \\ (MNAM,\_,(\overline{TY_n})NOTNULL\_VOID) = mResolV(IND1, IND2) \\ MNAM \neq <\texttt{init}> \\ (COD, NLV) = mSelV(OBJ, MNAM, (\overline{TY_n})NOTNULL\_VOID) \end{array}}{\begin{array}{l} \Xi[P,J,L,S+OBJ\overline{+WRD_n}] \\ \Longrightarrow\ \Xi[COD, J+(L,S,P+3), lvs^{NLV}[0 \mapsto OBJ, \overline{n \mapsto WRD_n}], []] \end{array}} \quad \text{(S-14)}$$

$$\frac{\begin{array}{l} Prg(P) = \texttt{invokeinterface}\ IND1\ IND2\ BYT\ 0 \\ (MNAM,\_,(\overline{TY_n})NOTNULL\_VOID) = mResolI(IND1, IND2) \\ n = BYT - 1 \\ (COD, NLV) = mSelI(OBJ, MNAM, (\overline{TY_n})NOTNULL\_VOID) \end{array}}{\begin{array}{l} \Xi[P,J,L,S+OBJ\overline{+WRD_n}] \\ \Longrightarrow\ \Xi[COD, J+(L,S,P+5), lvs^{NLV}[0 \mapsto OBJ, \overline{n \mapsto WRD_n}], []] \end{array}} \quad \text{(S-15)}$$

$$\frac{\begin{array}{l} Prg(P) = \texttt{invokestatic}\ IND1\ IND2 \\ (\_,\_,(\overline{TY_n})NOTNULL\_VOID, COD, NLV) = mResolSt(IND1, IND2) \end{array}}{\begin{array}{l} \Xi[P,J,L,S\overline{+WRD_n}] \\ \Longrightarrow\ \Xi[COD, J+(L,S,P+3), lvs^{NLV}[i \mapsto WRD_{i+1} \mid 0 \leq i < n], []] \end{array}} \quad \text{(S-16)}$$

Figure 12: The state transitions for method invocation instructions

$$\frac{\begin{array}{l} \boxed{\begin{array}{l} Mth(P) = \_ \\ P = (\_, 0) \end{array}} \\ (MNAM, CNAM, (\overline{TY_n})NOTNULL\_VOID, NLV) = mInfo(P) \\ MNAM = <\texttt{init}> \Rightarrow \\ \quad (\ NOTNULL\_VOID = void \wedge \\ \quad\quad (\ CNAM \neq Object \Rightarrow (\ LT = unusable^{NLV}[0 \mapsto init(CNAM), \overline{n \mapsto TY_n}] \wedge \\ \quad\quad\quad\quad IT = notInitd\ )\ ) \wedge \\ \quad\quad (\ CNAM = Object \Rightarrow (\ LT = unusable^{NLV}[0 \mapsto CNAM, \overline{n \mapsto TY_n}] \wedge \\ \quad\quad\quad\quad IT = Initd\ )\ ) \\ instMeth(MNAM, CNAM, (\overline{TY_n})NOTNULL\_VOID) \Rightarrow \\ \quad LT = unusable^{NLV}[0 \mapsto CNAM, \overline{n \mapsto TY_n}] \\ statMeth(MNAM, CNAM, (\overline{TY_n})NOTNULL\_VOID) \Rightarrow \\ \quad LT = unusable^{NLV}[i \mapsto TY_{i+1} \mid 0 \leq i < n] \end{array}}{\Phi(P) \sqsupseteq (LT, [], IT, mod_0)} \quad \text{(T-13)}$$

Figure 13: The typing rule for the starting program point of a method code

general, however, each local variable that does not store the object, on which the method is invoked, nor an actual parameter, is always given the type *unusable*. This means that the content of such a local variable cannot be used before the program explicitly assigns something to the local variable. This means that the content of such a local variable cannot be used before the program explicitly assigns something to the local variable. We use $unusable^n$ to denote the list $[unusable, \cdots, unusable]$ consisting of n *unusable*.

In the case of an `<init>` method, the local variable 0 stores the object being initialized. The static type for the local variable 0 and the initialization tag depend on whether the class *CNAM* containing the method code is *Object* or not. If *CNAM* is not *Object*, then the initialization tag is *notInitd* and the static type for the local variable 0 is $init(CNAM)$; The initialization tag *notInitd* means that an instance initialization method needs to be called exactly once within the current method code in any case, since, as we will see, rule (T-14) will change the initialization tag into *Initd* and rule (T-20) checks whether the initialization tag is really *Initd*. Note that if *CNAM* is *Object*, the object being initialized cannot, and need not, be initialized by another `<init>` within the current `<init>`.

Another point here is that the class *CNAM* is chosen to be the one containing the `<init>` method. In fact, rule (T-14) will assure that *CNAM* is either the original class of the object being initialized, or a superclass of it. Thus it is safe to use *CNAM* at the place of the original class

The rule contains the component mod_0, which will be defined in Section 10.6.

The cases for an instance method and a `static` method are straightforward. Not much explanation for these rules is necessary.

The typing rules for method invocation instructions are given in Figure 14 and 15. Although these method invocation instructions are based on quite different mechanisms, they all require that the operand stack at the program point of the instruction contain the correct number of arguments with certain types. In order to express this, each of the typing rules contains constraints of the following forms:

$$
\begin{array}{l}
(\cdots, (\overline{TY_n})void, \cdots) = a\_resolution\_function(IND1, IND2) \\
\Phi(P) = \Pi[\cdots, ST + \cdots \overline{+TYS_n}, \cdots] \\
invoConvert(TYS_i, TY_i) \quad (i = 1, \ldots, n)
\end{array}
$$

We consider rule (T-14) for `invokespecial` in Figure 14 in detail. The rule looks quite complicated, since the $\mathcal{CC}$-part of the rule basically gives three cases. The program point type at the program point of a `invokespecial` instruction must satisfy one of these cases.

The first case is when an `<init>` method is invoked on an object, on which no `<init>` method has been invoked before. In this case, the operand stack entry containing the object to be initialized has the static type $unin(P', CNAM)$. Following the OJVMS, the rule requires that the class containing the `<init>` method must be *CNAM*, and that after the instruction, all occurrences of $unin(P', CNAM)$ are changed into *CNAM*, indicating that the object has been initialized.

$$
\frac{
\begin{array}{l}
\boxed{Mth(P) = \texttt{invokespecial}\ IND1\ IND2} \\
(MNAM, CNAM, (\overline{TY_n})NOTNULL\_VOID, \_, \_) = mResolSp(IND1, IND2) \\
\Phi(P) = \Pi[LT, ST + REFS\_RAW\overline{+TYS_n}, IT, M] \\
invoConvert(TYS_i, TY_i) \quad (i = 1, \ldots, n) \\
MNAM = <\texttt{init}> \Rightarrow \\
\quad (\ (\ (\ REFS\_RAW = unin(P', CNAM) \wedge \\
\qquad LT' = LT[CNAM/REFS\_RAW] \wedge \\
\qquad ST' = ST[CNAM/REFS\_RAW] \wedge \\
\qquad IT' = IT \wedge \\
\qquad M' = Mod_1\ ) \vee \\
\qquad (\ REFS\_RAW = init(CNAM') \wedge \\
\qquad IT = notInitd \wedge \\
\qquad (CNAM' = CNAM \vee diSubcls(CNAM', CNAM)) \wedge \\
\qquad LT' = LT[CNAM/REFS\_RAW] \wedge \\
\qquad ST' = ST[CNAM/REFS\_RAW] \wedge \\
\qquad IT' = Initd \wedge \\
\qquad M' = Mod_2\ )\ ) \wedge \\
\quad NOTNULL\_VOID = void\ ) \\
MNAM \neq <\texttt{init}> \Rightarrow \\
\quad (\ widRefConvert(REFS\_RAW, CNAM) \wedge \\
\quad LT' = LT \wedge \\
\quad (\ NOTNULL\_VOID = NOTNULL \Rightarrow ST' = ST + NOTNULL\ ) \wedge \\
\quad (\ NOTNULL\_VOID = void \Rightarrow ST' = ST\ ) \wedge \\
\quad IT' = IT \wedge \\
\quad M' = M\ ) \\
\hline
\Phi(P+3) \sqsupseteq \Pi[LT', ST', IT', M']
\end{array}
} \quad \text{(T-14)}
$$

Figure 14: The typing rule for `invokespecial`

Note that the rule changes the component M into Mod_1 in the above case. The definition of Mod_1 will be given in Section 10.7.

The second case is when the instruction invokes an `<init>` method on an object that is being initialized within the enclosing `<init>` method, i.e. when the initialization tag IT is *notInitd* and the operand stack entry for the object has the static type $init(CNAM')$. In this case $init(CNAM')$ must be introduced by rule (T-13). As mentioned in the discussion for that rule, the enclosing method must be in the class $CNAM'$. The constraint

$$(CNAM' = CNAM) \vee diSubcls(CNAM', CNAM)$$

means that the invoked `<init>` method is either in the same class as the enclosing method or in the immediate superclass of it. Analogous to the first case above, the instruction changes all occurrences of $init(CNAM')$ into $CNAM$, indicating that after the instruction (but still inside the enclosing `<init>` method) the object being initialized is regarded as having been initialized. In addition, the constraint $IT' = Initd$ in the rule expresses the change of the initialization tag

into *Initd*. Rule (T-20) for `return` will use the tag to determine whether an `<init>` method really invokes another `<init>` method or not.

The constraint $NOTNULL\_VOID = void$ assures that the `<init>` method has no return type.

Note that the rule changes the component M into Mod_2 in the second case. The definition of Mod_2 will be given in Section 10.7.

The third case concerns the invocation of a usual instance method (e.g. via `super`). In this case, the constraint $widRefConvert(REFS\_RAW, CNAM)$ assures that the class $CNAM$ is a superclass of all possible classes of the object, on which the method is invoked. In addition, the constraint implicitly implies that $REFS\_RAW = REFS$ holds. Now the method may have a return type or not. the operand stack type ST' after the instruction is either $ST + NOTNULL$ or ST.

$$\frac{\begin{array}{l}\boxed{Mth(P) = \texttt{invokevirtual}\ IND1\ IND2} \\ (MNAM, CNAM, (\overline{TY_n})NOTNULL\_VOID, \_, \_) = mResolV(IND1, IND2) \\ \Phi(P) = \Pi[ST + REFS\overline{+TYS_n}] \\ invoConvert(TYS_i, TY_i)\ \ (i = 1, \ldots, n) \\ widRefConvert(REFS, CNAM) \\ MNAM \neq <\texttt{init}> \\ NOTNULL\_VOID = void \Rightarrow ST' = ST \\ NOTNULL\_VOID = NOTNULL \Rightarrow ST' = ST + NOTNULL\end{array}}{\Phi(P+3) \sqsupseteq \Pi[ST']} \quad \text{(T-15)}$$

$$\frac{\begin{array}{l}\boxed{Mth(P) = \texttt{invokeinterface}\ IND1\ IND2\ BYT1\ BYT2} \\ BYT1 > 0 \\ BYT2 = 0 \\ (MNAM, INAM, (\overline{TY_{BYT1-1}})NOTNULL\_VOID) = mResolI(IND1, IND2) \\ \Phi(P) = \Pi[ST + REFS\overline{+TYS_{BYT1-1}}] \\ invoConvert(TYS_i, TY_i)\ \ (i = 1, \ldots, BYT1 - 1) \\ widRefConvert(REFS, INAM) \\ MNAM \neq <\texttt{init}> \\ NOTNULL\_VOID = void \Rightarrow ST' = ST \\ NOTNULL\_VOID = NOTNULL \Rightarrow ST' = ST + NOTNULL\end{array}}{\Phi(P+5) \sqsupseteq \Pi[ST']} \quad \text{(T-16)}$$

$$\frac{\begin{array}{l}\boxed{Mth(P) = \texttt{invokestatic}\ IND1\ IND2} \\ (MNAM, \_, (\overline{TY_n})NOTNULL\_VOID, \_, \_) = mResolSt(IND1, IND2) \\ \Phi(P) = \Pi[ST\overline{+TYS_n}] \\ invoConvert(TYS_i, TY_i)\ \ (i = 1, \ldots, n) \\ MNAM \neq <\texttt{init}> \\ NOTNULL\_VOID = void \Rightarrow ST' = ST \\ NOTNULL\_VOID = NOTNULL \Rightarrow ST' = ST + NOTNULL\end{array}}{\Phi(P+3) \sqsupseteq \Pi[ST']} \quad \text{(T-17)}$$

Figure 15: The typing rules for other method invocation instructions

Rules (T-15), (T-16) and (T-17) are for `invokevirtual`, `invokeinterface` and `invokestatic`. They are very similar to the third case of rule (T-14). One difference is that they use the resolution functions $mResolV$, $mResolI$ and $mResolSt$, respectively, instead of $mResolSp$. In addition, rule (T-16) needs to deal with the number $BYT1$ and $BYT2$ explicitly occurring in the `invokeinterface` instruction. The `invokestatic` does not need an object, on which the method is invoked.

$$\frac{Prg(P) = \texttt{areturn} \text{ or } \texttt{ireturn}}{\Xi[P, J + [(L', S', P')], L, S + WRD] \implies \Xi[P', J, L', S' + WRD]} \quad \text{(S-17),(S-18)}$$

$$\frac{Prg(P) = \texttt{return}}{\Xi[P, J + [(L', S', P')], L, S] \implies \Xi[P', J, L', S']} \quad \text{(S-19)}$$

Figure 16: The state transitions for return instructions

The state transition rules for return instructions are given in Figure 16. The state transition uses the return address P' stored in the current Java stack.

$$\frac{\boxed{Mth(P) = \texttt{areturn}} \quad \Phi(P) = \Pi[ST + REFS] \quad (\_, \_, (\overline{TY_n})REF, \_) = mInfo(P) \quad widRefConvert(REFS, REF)}{} \quad \text{(T-18)}$$

$$\frac{\boxed{Mth(P) = \texttt{ireturn}} \quad \Phi(P) = \Pi[ST + int] \quad (\_, \_, (\overline{TY_n})int, \_) = mInfo(P)}{} \quad \text{(T-19)}$$

$$\frac{\boxed{Mth(P) = \texttt{return}} \quad (\_, MNAM, (\overline{TY_n})void, \_) = mInfo(P) \quad MNAM = <\texttt{init}> \Rightarrow \Phi(P) = \Pi[Initd]}{} \quad \text{(T-20)}$$

Figure 17: The typing rules for return instructions

The typing rules for return instructions are given in Figure 17. The rules need no additional explanations. The only thing that is worth mentioning is that a `return` instruction may be used to terminate an `<init>` method. In this case, the rule checks whether the initialization tag is *Initd* to assure that the `<init>` method has indeed invoked another `<init>` method. Note that if the `<init>` method is in *Object*, then the tag has been set into *Initd* at the beginning of the method.

Note that in general, there may exist some uninitialized objects in the operand stack or local variables when a method terminates. However, there is no possibility to pass an uninitialized object to the invoking method (see Theorem 2).

10.6 Implementing `Finally`-Clauses

According to the OJVMS, `jsr` and `ret` instructions are control transfer instructions typically used to implement `finally` clauses in `Java`. Following the OJVMS, we call the program point, to which a `jsr` instruction jumps, a `jsr` *target*, and the code starting from a `jsr` target a *subroutine*. If no ambiguity is possible, we also call a `jsr` target a *subroutine*. Roughly speaking, a `jsr` instruction calls a subroutine and a `ret` instruction returns from a subroutine. But, formally a subroutine need not have a `ret` instruction. We use *sb* to range over all `jsr` targets (i.e. subroutines) and write SB as a variable for them.

$$\frac{\begin{array}{l} Prg(P) = \texttt{jsr}\ BYT1\ BYT2 \\ P = (\_, OFF) \end{array}}{\Xi[P, S] \implies \Xi[P + \mathit{offset}(BYT1, BYT2), S + OFF]} \tag{S-20}$$

$$\frac{\begin{array}{l} Prg(P) = \texttt{ret}\ IND \\ P = (COD, \_) \end{array}}{\Xi[P, L] \implies \Xi[(COD, L(IND) + 3), L]} \tag{S-21}$$

Figure 18: The state transitions for `jsr` and `ret`

The state transitions for `jsr` and `ret` are given in Figure 18. Rule (S-20) says that a `jsr` instruction pushes the byte offset OFF of the current program point onto the operand stack and transfers control to the `jsr` target $P + \mathit{offset}(BYT1, BYT2)$.

Rule (S-21) is for `ret`. It uses a byte offset in a local variable to compute the program point following the `jsr` as the returning program point.

Typing `jsr` and `ret` is complex, since the OJVMS requires the following features:

- Not every path in a subroutine needs to reach a `ret` instruction. A subroutine implicitly terminates whenever the current method terminates.
- Subroutines may be nested: a subroutine can call another subroutine. (This feature is useful in implementing nested `finally` clauses.)
- In nested subroutines, an inner subroutine may contain a `ret` instruction that directly returns to an arbitrary outer subroutine.
- During the execution, a returning program point can never be used more than once by a `ret` instruction. Furthermore, at the outer program point, to which a `ret` instruction in an inner subroutine directly returns, no returning program point for a subroutine between the inner and the outer subroutine should still be able to be used as a returning program point.

Technically, the mechanism should be more complex, since the OJVMS still takes three additional situations into account. First, the implementation of a `finally` clause often needs to be reachable from different execution paths. Second, different execution paths often have to use a common local variable to hold their own contents that are incompatible to each other. Third, the content stored in a local variable in an execution path before the execution of a `finally` clause may need to be used after the execution of the `finally` clause. As an example for all these three situations, we can consider the implementation of a `try-catch-finally` clause. More concretely, the `finally` clause needs to be reachable from the end of the `try` clause and from the beginning of the `catch` clause, the `try` clause needs to store a return integer value in a local variable for use after the execution of the `finally` clause, but the `catch` clause stores an exception in the same local variable for use after the execution of the `finally` clause as well.

The problem is that since a common local variable may hold incompatible contents, as described in the second situation above, the usual typing rules in our formal specification would force the local variable in and, in particular, after the `finally` clause to have the type *unusable*. Therefore a use of the local variable in an individual execution path after the `finally` clause, as described in the third situation, would be impossible.

To solve the problem, the OJVMS suggests to change the usual typing process such that in an execution path, if a local variable is not modified or accessed in a `finally` clause, then its type after the execution of the `finally` clause should be the same as before the execution of the `finally` clause. Thus we need a mechanism to record the local variables that are modified or accessed within a `finally` clause. The component *mod* in a program point type has been reserved for this purpose. Now we formally define what a component *mod* is:

- First, we build a set *grf* of pairs of `jsr` targets, representing a directed acyclic graph.
- Then we build a set *csb* of `jsr` targets.
- Finally, a component *mod* is a mapping such that $\mathcal{D}om(mod) = grf \cup csb$, $mod(sb, sb')$ for $(sb, sb') \in grf$ and $mod(sb)$ for $sb \in csb$ are sets of indices of local variables.

Intuitively, a pair (sb, sb') in a *grf* should denote a call of the subroutine sb' inside the subroutine sb, and *grf* should contain nested non-recursive subroutine calls that may reach the current program point. A set *grf* need not be a tree, since more than one subroutine may contain a call of the same subroutine and one subroutine may contain calls of more than one subroutine. A set *csb* should contain current subroutines, i.e. those subroutines that contain the current program point. The set $mod(sb, sb')$ for $(sb, sb') \in grf$ should contain the indices of all local variables that may be modified or accessed in an execution path from sb to sb', and $mod(sb)$ for $sb \in csb$ those from sb to the current address.

We define the following notations:

$$\begin{aligned} &nod(mod) := \{sb \mid (sb, \_) \text{ or } (\_, sb) \text{ or } sb \in \mathcal{D}om(mod)\} \\ &grf(mod) := \{(sb, sb') \mid (sb, sb') \in \mathcal{D}om(mod)\} \\ &csb(mod) := \{sb \mid sb \in \mathcal{D}om(mod)\} \end{aligned}$$

We define that $mod \sqsupseteq mod'$ holds if and only if $grf(mod) \supseteq grf(mod')$ and $csb(mod) \supseteq csb(mod')$ hold, $mod(sb, sb') \supseteq mod'(sb, sb')$ holds for each $(sb, sb') \in grf(mod')$ and $mod(sb) \supseteq mod'(sb)$ holds for each $sb \in csb(mod')$.

$$\frac{\begin{array}{l} \boxed{Mth(P) = \texttt{jsr}\ BYT1\ BYT2} \\ \Phi(P) = \Pi[ST, M] \\ SB = P + \mathit{offset}(BYT1, BYT2) \\ SB \notin nod(M) \end{array}}{\begin{array}{l} \Phi(SB) \sqsupseteq \Pi[ST + sbr(SB), \\ \qquad M_{|grf(M)} \cup \{(sb, SB) \mapsto M(sb) \mid sb \in csb(M)\} \cup \{SB \mapsto \emptyset\}] \end{array}} \quad \text{(T-21)}$$

$$\frac{\begin{array}{l} \boxed{Mth(P) = \texttt{ret}\ IND} \\ \Phi(P) = \Pi[LT] \\ LT(IND) = sbr(\_) \\ \forall P' \forall IND' \forall \Pi' \forall LT'.(\ (Mth(P') = \texttt{ret}\ IND' \ \wedge\ P' \neq P \ \wedge\ \Phi(P') = \Pi'[LT']) \\ \qquad\qquad \Rightarrow LT(IND) \neq LT'(IND')\ ) \end{array}}{} \quad \text{(T-22)}$$

$$\frac{\begin{array}{l} \boxed{\begin{array}{l} Mth(P) = \texttt{ret}\ IND \\ \Phi(P) = (LT, ST, IT, M) \\ Mth(P') = \texttt{jsr}\ BYT1\ BYT2 \\ SB = P' + \mathit{offset}(BYT1, BYT2) \\ LT(IND) = sbr(SB) \\ \Phi(P') = \Pi'[LT', ST', M'] \end{array}} \end{array}}{\begin{array}{l} \Phi(P' + 3) \sqsupseteq (LT'[j \mapsto invld(LT(j), subrs(SB, M)) \mid\ j \in mlvs(SB, M)], \\ \qquad invld(ST, subrs(SB, M)), IT, \\ \qquad reachMod(M, \{sb \mid (sb, SB) \in \mathcal{D}om(M)\}) \cup \\ \qquad \{sb \mapsto M'(sb, SB) \cup mlvs(SB, M) \mid (sb, SB) \in \mathcal{D}om(M)\}) \end{array}} \quad \text{(T-23)}$$

Figure 19: The typing rules for `ret` and `jsr`

The typing rules for `jsr` and `ret` are given in Figure 19. We first consider rule (T-21). The rule defines only one constraint at the program point P of a `jsr` instruction, namely $SB \notin nod(M)$, which assures that the called subroutine SB is not called recursively. At the beginning of the subroutine SB, the new M records the addition of the edges (sb, SB) representing the calls of SB inside all old current subroutines in $csb(M)$, and elimination of the old current subroutines in $csb(M)$ and addition of the new current subroutine SB, where $SB \mapsto \emptyset$ denotes that no local variables have been modified or accessed since the beginning of the new current subroutine SB.

Rule (T-22) is for `ret`. The constraint $LT(IND) = sbr(\_)$ assures that the local variable IND holds a byte offset. The constraint $\forall P' \forall IND' \forall \Pi' \forall LT' . \cdots$ assures that the method Mth has at most one `ret` instruction for the same subroutine. This is not a serious restriction, since whenever two `ret` instructions are needed, one can always write the first `ret` and at the place of the second `ret` a `goto` instruction to the first `ret`.

Rule (T-23) introduces constraints for the program type at the returning program point $P' + 3$, to which a `ret` at P returns, where the calling `jsr` of the subroutine is at P'.

The formulation of rule (T-23) uses several new auxiliary functions.

The first auxiliary function computes the set of the indices of all local variables that may be modified or accessed in an execution path from a given program point to the current program point. For the component mod in a program point type and a subroutine $sb \in nod(mod)$, we define

$$mlvs(sb, mod) := \begin{cases} \bigcup_{(sb,sb') \in grf(mod)} (mod(sb, sb') \cup mlvs(sb', mod)) & \text{if } sb \notin csb(mod) \\ mod(sb) & \text{if } sb \in csb(mod) \end{cases}$$

The term $mlvs(SB, M)$ in rule (T-23) is a set containing the indices of all local variable that may be modified or accessed from SB to P.

The second auxiliary function computes all subroutines called from the call of a given outer subroutine to the current subroutine. For the component mod in a program point type and a subroutine $sb \in nod(mod)$, we define

$$subrs(sb, mod) := \begin{cases} \bigcup_{(sb,sb') \in grf(mod)} \{sb\} \cup subrs(sb', mod) & \text{if } sb \notin csb(mod) \\ \{sb\} & \text{if } sb \in csb(mod) \end{cases}$$

In order to change all subroutine types of those subroutines in a given set of subroutines E into invalid subroutine types, we define the following function $invld(any, E)$:

$$invld(any, E) := \begin{cases} invldsbr & \text{if } any = sbr(sb) \text{ and } sb \in E \\ any & \text{otherwise} \end{cases}$$

Note that any in the second line can be an arbitrary static type.

For convenience, we lift the function $invld$ to operand stack types:

$$invld([any_0, \cdots, any_m], E) := [invld(any_0, E), \cdots, invld(any_m, E)]$$

In order to compute the part of a mod, which is reachable to a subroutine in a given set of subroutines E, we define the following function $reachMod(mod, E)$:

$$reachMod(mod, E) := \begin{cases} \{(sb, sb') \mapsto mod(sb, sb') \mid (sb, sb') \in \mathcal{D}om(mod), sb' \in E\} \\ \quad \cup\, reachMod(mod, \{sb \mid (sb, sb') \in \mathcal{D}om(mod), sb' \in E\}) & \text{if } E \neq \emptyset \\ \emptyset & \text{if } E = \emptyset \end{cases}$$

In rule (T-23), the applicability conditions $\Phi(P) = (LT, \cdots)$, $Mth(P') =$ `jsr` $BYT1\ BYT2$, $SB = P' + offset(BYT1, BYT2)$ and $LT(IND) = sbr(SB)$ assure that the `ret` at P causes the subroutine SB to return to the next program point $P' + 3$ of the calling `jsr` at P'. Note that the constraint $\forall P' \forall IND' \forall \Pi' \forall LT' \cdots$ in rule (T-22) enforces that there exists at most one P for a `jsr` at P' in rule (T-23).

Rule (T-23) expresses the following relationship between the program types at P, P' and $P' + 3$, where the `jsr` at P' calls a subroutine SB, the `ret` at P returns from the subroutine SB to $P' + 3$:

- If a local variable is definitely not modified or accessed from SB to P, then its static type at $P' + 3$ covers that at P'; otherwise, i.e. if the local variable may be modified or accessed from SB to P, then its static type at $P' + 3$ covers that at P, except that if its static type at P is a subroutine type for a subroutine possibly called from SB to P, then its static type at $P' + 3$ covers $invldsbr$.
- The operand stack type at $P' + 3$ covers that at P, except that if an operand stack entry at P has a subroutine type for a subroutine called from SB to P, then the static type of the entry at $P' + 3$ covers $invldsbr$.
- The initialization tag at $P' + 3$ covers that at P.
- The subroutines called until $P' + 3$ include all those called until P but not from SB to P. The local variables possibly modified or accessed from the call of a current subroutine to $P' + 3$ include those from the call of the same current subroutine to SB plus all those possibly modified or accessed from SB to P.

A final tricky point is that although the `ret` instruction in rule (T-23) accesses all those local variables that have a subroutine type for a subroutine called from P' to P, the typing rule need not treat this explicitly. The reason is that the indices of these variables are all contained in the set $mlvs(SB, M)$ in rule (T-23). In fact, if a local variable holds a program point of a `jsr` instruction between SB and P, then the program point must be stored in the local variable by an `astore` instruction between SB and P. By the typing rule for `astore` (see the discussion below) and the definition of $mlvs$, the index of the local variable must be included in the set $mlvs(SB, M)$.

10.7 On the Instructions that Modify or Access Local Variables

Now it is time to give the precise definitions of the term Mod' in Figure 4, the term mod_0 in Figure 13 and the terms Mod_1 and Mod_2 in 14.

We first consider the typing rules in Figure 4. Given the notations in Figure 4, we formally define

$$Mod' := M_{|grf(M)} \cup \{sb \mapsto M(sb) \cup \{IND\} \mid sb \in csb(M)\}$$

The typing rule in Figure 13 introduces the program point type for the starting program point of a method. We define $mod_0 := \emptyset$.

Now we consider rule (T-14) for `invokespecial` in Figure 14. Since the first two cases in rule (T-14) consider the initialization of a raw object, we regard all those local variables whose contents reference the raw object as being modified. Given the notations in Figure 14, we formally define

$$\begin{aligned} Mod_1 &:= M_{|grf(M)} \cup \\ &\quad \{sb \mapsto M(sb) \cup \{i \mid unin(P', CNAM) = LT(i)\} \mid sb \in csb(M)\} \\ Mod_2 &:= M_{|grf(M)} \cup \{sb \mapsto M(sb) \cup \{i \mid init(CNAM) = LT(i)\} \mid sb \in csb(M)\} \end{aligned}$$

Note that the third case in rule (T-14) does not deal with initialization of a raw object, thus does not cause the extension of the M.

11 Examples

In this section we use real methods to illustrate how to check whether a given method has a given program type.

For notational simplicity, some instructions are abbreviated as follows:

- Each instruction
$$opcode\ byt1\ byt2$$
at the program point pp with $opcode \in \{\texttt{if\_acmeq}, \texttt{if\_icmeq}, \texttt{goto}, \texttt{jsr}\}$ and $pp = (\_, off)$ is abbreviated as
$$opcode\ n$$
with $n = off + (byt1 * (2^8)) + byt2$.
- Each instruction
$$opcode\ ind1\ ind2$$
with $opcode \in \{\texttt{getfield}, \texttt{putfield}, \texttt{new}, \texttt{invokespecial}, \texttt{invokevirtual}, \texttt{invokeinterface}, \texttt{invokestatic}\}$ is abbreviated as
$$opcode\ \#n$$
with $n = (ind1 * (2^8)) + ind2$. The instruction
$$\texttt{invokeinterface}\ ind1\ ind2\ byt\ 0$$
is abbreviated as $\texttt{invokeinterface}\ \#n$ with $n = (ind1 * (2^8)) + ind2$.

Figure 20 gives the type checking for the first method. A row in Figure 20 contains a program point, i.e. an instruction, in the given method, the program point type in the given program type at the program point, the typing rule applied at the program point and all possible successor program points with respect to the rule. Since the method does not deal with any subroutines or instance initializations, we consider only the local variable table type and the operand stack type in a program point type.

We assume that the declaration of the method `void m(J1,J2)` in Figure 20 is contained in a class C. Furthermore, we assume that `J1` and `J2` are two interfaces, and the entry at the index `#17` in the constant pool references a method in a superinterface of `J1` and `J2`, which takes no parameters and yields no result. At the program point `13`, the static type of the top entry in the operand stack needs to be represented as a set, since the top entry may be the first or second actual parameter and the interfaces `J1` and `J2` need not have one smallest common superinterface. Rule (T-16) is applied at the program point `13`, where the constraint $widRefConvert(REFS, INAM)$ in the rule assures that the invoked method must exist in a superinterface of `J1` and `J2`.

The method	LT	ST	Rule	Successors
`Method void m(J1,J2)`			(T-13)	`0`
`0 aconst_null`	$[C, J1, J2]$	$[]$	(T-5)	`1`
`1 aload 1`	$[C, J1, J2]$	$[null]$	(T-1)	`3`
`3 if_acmpeq 11`	$[C, J1, J2]$	$[null, J1]$	(T-10)	`6`, `11`
`6 aload 1`	$[C, J1, J2]$	$[]$	(T-1)	`8`
`8 goto 13`	$[C, J1, J2]$	$[J1]$	(T-12)	`13`
`11 aload 2`	$[C, J1, J2]$	$[]$	(T-1)	`13`
`13 invokeinterface #17`	$[C, J1, J2]$	$[\{J1, J2\}]$	(T-16)	`18`
`18 return`	$[C, J1, J2]$	$[]$	(T-20)	

Figure 20: A method containing an interface method invocation

The second example in Figure 21 shows the use of subroutine types. The method contains two `jsr` instructions at `0` and `9` calling the subroutine `13`. The subroutine `13` contains a `jsr` at `15` calling the (inner) subroutine `18`, and the subroutine `18` directly returns to the corresponding calling `jsr` of the (outer) subroutine `13`, i.e. to `3` or `12`. After the return, i.e. at `3` and `12`, the subroutine types $sbr(13)$ and $sbr(18)$ are changed into $invldsbr$. The local variable 1 has different static types at the two calling `jsr`, i.e. at `0` and `9`. Since the local variable 1 is not modified or accessed in the subroutine `13`, after the return of the subroutine, i.e. at `3` and `12`, the static type of the local variable 1 is the same as that at the calling `jsr`, i.e. at `0` and `9`.

12 Static Well-Typedness vs. Runtime Properties

The OJVMS requires that the type-correctness of nearly all runtime uses of data is checked statically. In our formal specification, which considers a subset of the JVM, we can formally prove that if a program is statically well-typed, then all runtime data to be used will definitely have correct types. For doing this, we first need to define precisely what are the types of runtime data.

The method	*LT*	*ST*	Rule	Successors
`Method void m()`			(T-13)	`0`
`0 jsr 13`	$[C, unusable, unusable]$	$[]$	(T-21)	`13`
`3 astore 2`	$[C, unusable, invldsbr]$	$[invldsbr]$	(T-3)	`5`
`5 aload 0`	$[C, unusable, invldsbr]$	$[]$	(T-1)	`7`
`7 astore 1`	$[C, unusable, invldsbr]$	$[C]$	(T-3)	`9`
`9 jsr 13`	$[C, C, invldsbr]$	$[]$	(T-21)	`13`
`12 return`	$[C, C, invldsbr]$	$[invldsbr]$	(T-20)	
`13 astore 2`	$[C, unusable, unusable]$	$[sbr(13)]$	(T-3)	`15`
`15 jsr 18`	$[C, unusable, sbr(13)]$	$[]$	(T-21)	`18`
`18 ret 2`	$[C, unusable, sbr(13)]$	$[sbr(18)]$	(T-22)	
			(T-23)	`3`
			(T-23)	`6`

Figure 21: A method containing subroutines

12.1 Tags of Runtime Data

In the previous sections we have often informally mentioned the types of runtime data. Two examples are as follows:

- In Section 10.2 we mentioned that a `new` instruction creates an object. This informally implies that the created datum is a reference of an object.
- In Section 10.6 we mentioned that a `jsr` instruction pushes a byte offset onto the operand stack.

However, the problem is that both an object reference and a byte offset are one-word wide data in our constraint domain. In other words, the type of a datum cannot be determined by the datum itself. Thus we need an additional mechanism to explicitly determine the type of a datum.

The mechanism can be built in two steps: first, we define the possible types of runtime data; second, we extend the state transition relation to define the types of the contents in the local variables and the operand stack.

A relatively simple set of possible types of runtime data, called *tags*, is defined as follows:

$$tag ::= cnam \mid null \mid int \mid addr \mid undefined$$

Intuitively, the tag *cnam* should be the tag of the reference of each object of the class *cnam*, *null* that of the special reference *null*, and *int* that of each element of the primitive type *int*. As mentioned before, we need to deal with the byte offset of a `jsr`. Thus we introduce the tag *addr* for all byte offsets. The tag *undefined* indicates that the content of a local variable or an operand stack entry has not been explicitly defined by an instruction in the execution so far. We use *tag* to range over all tags.

Note that the above set of tags is a relatively simple one, since they do not contain anything to express that an object is a raw object or an offset is of a special subroutine type. In fact, there are no problems to do that, except that

the definition of the types of the contents in the local variables and the operand stack would become more complicated. We consider the above simple set due to space limits in this chapter.

To record the type of the content of each local variable and each operand entry, lists of tags of the form $[tag_0, \cdots, tag_n]$ with $n \geq -1$ are introduced. We define $[tag_0, \cdots, tag_n](k) := tag_k$ if $0 \leq k \leq n$, $[tag_0, \cdots, tag_n](k) := failure$ otherwise. A list of the above form is called a *local variable state tag* if it consists of the types of the contents in all local variables; it is called an *operand stack state tag* if it consists of the types of the contents of an operand stack.

For readability, we use *lvstag* to range over all local variable state tags, and *stktag* over all operand stack state tags. For notational simplicity, we write LG as a variable for the sort *lvstag*, and SG for the sort *stktag*.

A local variable state tag and an operand stack state tag do not record the type of an object that is held by a field of another object but not directly by a local variable or an operand stack entry. Thus we still need to introduce a *class record* as a mapping $\{\overline{obj_n \mapsto cnam_n}\}$. A class record as above maps all elements other than $\overline{obj_n}$ to a special value *failure*. We use *classof* to range over all concrete class records and C as a variable for the sort *classof*.

In order to record the local variable state tags and the operand stack state tags for the methods stored in a Java stack, we define a *Java stack tag* as a list consisting of entries of the form $(lvstag, stktag)$. We use *jstktag* to range over all Java stack tags and use JG as a variable of the sort *jstktag*.

We define a *program state tag* as a tuple $(jstktag, classof, lvstag, stktag)$ and in the rest of the chapter still use *statag* to range over all program state tags.

Finally, we define that an *extended program state* is a pair $(stat, statag)$, where $stat = (pp, jstk, lvs, stk, hp)$, $statag = (jstktag, classof, lvstag, stktag)$, $size(jstk) = size(jstktag)$, $size(lvs) = size(lvstag)$ and $size(stk) = size(stktag)$ hold.

Now we extend the state transition rules in Section 10. Let us call an extended state transition rule an *extended rule* and an original state transition rule an *extended rule* in this section.

In order to ensure that the extended rule relation does not affect the original state transition relation, we require that if an original rule in Section 10 is of the form

$$\frac{Premises}{Stat \implies Stat'}$$

then the extended rule obtained from it is always of the form

$$\frac{Premises}{Stat, Statag \implies Stat', Statag'}$$

satisfying that

- $\mathcal{FV}(Statag') \subseteq \mathcal{FV}(Statag)$
- for every two program states $stat$ and $stat'$ and every extended program state $(stat, statag)$, if there is a substitution σ such that $\mathcal{D}om(\sigma) = \mathcal{FV}(Premises)$ $cup\mathcal{FV}(Stat \implies Stat')$, $stat = \sigma(Stat)$, $stat' = \sigma(Stat')$ and $\sigma(Premises)$

hold, then there is a substitution σ' such that $\mathcal{D}om(\sigma') = \mathcal{D}om(\sigma) \cup \mathcal{FV}(Statag)$ and $\sigma'(Statag) = statag$ hold, and $(stat', \sigma'(Statag'))$ is an extended program state.

For notational simplicity, we always omit the *Premises*-, *Stat*- and *Stat*′-parts in the definition of an extended rule in this section. Note that the *Statag*- and *Statag*′-parts may contain variables occurring in *Stat*- and *Stat*′-parts.

In many extended rules, the Java stack tags are not changed and the local variable state tags (or the operand stack state tags) are changed in a completely analogous way as the local variable states (or the operand stack states, respectively). The extended rules for `aload` and `new` are two such extended rules. We give their definitions in Figure 22 and omit the explicit presentation of other such extended rules due to space constraints.

$$\frac{}{(JG, C, LG, SG) \implies (JG, C, LG, SG + LG(IND))} \quad \text{(S'-1)}$$

$$\frac{}{(JG, C, LG, SG) \implies (JG, C[OBJ \mapsto CNAM], LG, SG + CNAM)} \quad \text{(S'-8)}$$

Figure 22: The extended rules for `aload` and `new`

Figure 23 contains the extended rule for `getfield`. The rule is slightly tricky, since the way to get the tag of the loaded content depends on whether the loaded content is an object or not. If it is an object, then the tag should be obtained from the class record *classof* in the program state. If it is a value of a primitive type, then the tag should the primitive type. (In this chapter the only primitive type is the type *int*.) To model this, we define the following auxiliary function, which yields the tag of the content held by the field *fnam* of the type *notnull* in the object *obj*.

$$seltag(fnam, notnull, obj, hp, classof) := \begin{cases} classof(hp(obj)(fnam)) & \text{if } notnull \text{ is a } cnam \\ int & \text{if } notnull \text{ is } int \\ failure & \text{otherwise} \end{cases}$$

$$\frac{}{\begin{array}{l}(JG, C, LG, SG) \\ \implies (JG, C, LG, SG + seltag(FNAM, NOTNULL, OBJ, H, C))\end{array}} \quad \text{(S'-6)}$$

Figure 23: The extended rule for `getfield`

Rules (S-13), (S-14) and (S-15) for method invocations change the Java stack states. Thus their extended rules change the Java stack tags. Since these extensions are very similar, we present only one of them. The situation is similar

for rules (S-17) and (S-19). Thus we present only one of the two extended rules. Figure 24 these two extended rules, where $undefined^k$ stands for a list $[undefined, \cdots, undefined]$ consisting of k times $undefined$.

$$\frac{}{(JG, C, LG, SG + TAG_0 \overline{+ TAG_n}) \implies (JG + (LG, SG), C, undefined^{NLV}[i \mapsto TAG_i \mid 0 \le i \le n], [])} \quad \text{(S'-13)}$$

$$\frac{}{(JG + (LG', SG'), C, LG, SG + TAG \implies (JG + (LG', SG'), C, LG', SG' + TAG]} \quad \text{(S'-17)}$$

Figure 24: The extended rules for `invokespecial` and `areturn`

$$\frac{}{(JG, C, LG, SG) \implies (JG, C, LG, SG + addr)} \quad \text{(S'-20)}$$

$$\frac{}{(JG, C, LG, SG) \implies (JG, C, LG, SG)} \quad \text{(S'-21)}$$

Figure 25: The extended rules for `jsr` and `ret`

Figure 25 contains the extended rules for `jsr` and `ret`. Note that in rule (S'-21), the program state tag does not change at all. The intuition is that a `ret` may change some the validity of some byte offsets. However, since we consider only a simple tag *addr* for byte offsets, this intuition cannot be reflected. (As mentioned, the simple tag *addr* could be replaced by a family of tags indexed by all subroutines. But we do not consider them here.)

12.2 The Concepts for Runtime Type Safety

To model the correctness of a tag *tag* with respect to a static type *any*, we formally define a relation *correct* by:

$correct(null, refs)$	
$correct(cnam, refs)$	if $widRefConvert(cnam, refs)$
$correct(cnam, unin(\_, cnam))$	
$correct(cnam, init(cnam'))$	if $widRefConvert(cnam, cnam')$
$correct(int, int)$	
$correct(addr, sbr(\_))$	
$correct(undefined, unusable)$	
$correct(tag, any)$	if $correct(tag, any')$ and $any \sqsupseteq any'$

We also define that $correct(lvstag, lvsty)$ holds if and only if $size(lvstag) = size(lvsty)$ and $correct(lvstag(i), lvsty(i))$ for all $i = 0, \ldots, size(lvstag)$, and that

$correct(stktag, stkty)$ holds if and only if $size(stktag) = size(stkty)$ and $correct(stktag(i), stkty(i))$ for all $i = 0, \ldots, size(stktag)$.

For a heap hp and a class record $classof$, we define that $correct(hp, classof)$ holds if and only if the following conditions are true:

1. $\mathcal{D}om(hp) \subseteq \mathcal{D}om(classof)$.
2. For each $obj \mapsto rec \in hp$, if $(fnam, notnull) \in allFields(classof(obj))$, then $fnam \in \mathcal{D}om(rec)$.
3. For each $obj \mapsto rec \in hp$ and $(fnam, notnull) \in allFields(classof(obj))$, if $notnull = ref$, then $rec(fnam) \in \mathcal{D}om(classof)$.
4. For each $obj \mapsto rec \in hp$ and $(fnam, notnull) \in allFields(classof(obj))$, if $notnull = ref$, then $widRefConvert(classof(rec(fnam)), notnull)$.

Intuitively, condition 1 says that $classof$ can determine the class of each object in hp. Condition 2 assures that an object in hp always contains all fields required by its class. Condition 3 assures that if an object in hp contains a field whose type is a class or an interface, then the field holds an object, whose class can be determined by $classof$. Condition 4 says that the class of the object held by the field in condition 3 is a subtype of the class or interface of the field.

Note that if $notnull \neq ref$, i.e. if $notnull = int$, then conditions 3 and 4 have no effects. Thus one might wonder why we do not define a condition constraining $rec(fnam)$. The intuition is that if the runtime type of a datum is a primitive type, then the runtime type is always the static type. Thus for $(fnam, int) \in allFields(classof(obj))$ and $obj \mapsto rec \in hp$, the content $rec(fnam))$ is always an integer of the type int. Hence such a condition is useless.

12.3 Runtime Properties

From now on, we assume that the program Prg has a program type $prgty$.

Formally we define an arbitrary *execution* of Prg as

$$(stat_1, statag_1) \implies (stat_2, statag_2) \implies \cdots$$

where each $(stat_i, statag_i)$ for $i = 1, 2, 3, \cdots$, are extended program states, $stat_1$ is of the form $(pp_1, [], \cdots)$ and $Prg(pp_1)$ is of the form `invokestatic` $\cdots$.

We use $(stat_1, statag_1) \implies^* (stat_h, statag_h)$ with $h \geq 1$ to denote a zero or more step execution

$$(stat_1, statag_1) \implies \cdots \implies (stat_h, statag_h)$$

For the rest of the chapter, we assume that

- $stat_i = (pp_i, jstk_i, lvs_i, stk_i, hp_i)$ for all $i = 1, 2, 3, \ldots$,
- $statag_i = (jstktag_i, classof_i, lvstag_i, stktag_i)$ for all $i = 1, 2, 3, \ldots$, and
- $prgty(pp_i) = ptty_{pp_i} = (lvsty_{pp_i}, stkty_{pp_i}, intag_{pp_i}, mod_{pp_i})$ for all $i = 1, 2, 3, \ldots$. Note that $i \neq j$ does not imply that $pp_i \neq pp_j$.

Now we give some lemmas and theorems. Proofs are omitted due to space limits.

The first theorem states the runtime type safety.

Theorem 1. *In the execution* $(stat_1, statag_1) \Longrightarrow (stat_2, statag_2) \Longrightarrow \cdots$*, if* $correct(lvstag_1, lvsty_{pp_1})$*,* $correct(stktag_1, stkty_{pp_1})$ *and* $correct(hp_1)$ *hold, then* $correct(lvstag_i, lvsty_{pp_i})$*,* $correct(stktag_i, stkty_{pp_i})$ *and* $correct(hp_i)$ *hold for all* $i = 1, 2, \cdots$.

The proof follows from an induction on the length of the execution using the extended rules and typing rules.

A practical consequence of Theorem 1 is as follows:

Corollary 1. *An offset cannot be manipulated by an instruction described in our formal specification except:*

1. *It can be created and stored onto the operand stack by a* `jsr`.
2. *It can be manipulated in the operand stack by the stack manipulation instruction* `dup`.
3. *It can be stored from the operand stack in a local variable by an* `astore`.
4. *In a local variable, it can be used to compute the return address by a* `ret`.

Now let us consider raw objects and instance initialization methods. The following theorems can either be proved using a set of tags for runtime data that is more refined then the current one, or by a careful analysis of all possible executions. Note that Theorems 2 and 3 are not completely trivial, since a method may pass values via the heap.

Theorem 2. *Assume that a method invokes another method. Then the invoked method can never pass a raw object back to the invoking method.*

Theorem 3. *Assume that a method invokes another method that is not an* `<init>`*. Then the invoking method can never pass a raw object to the invoked method.*

It is very easy to show how an instance initialization method invokes another instance initialization method.

Theorem 4. *If an instance initialization method is not in class Object, then a fragment of an execution path from the starting address to a* `return` *instruction of the method always includes exactly one invocation of an instance initialization method of the same class or the immediate superclass on the object being initialized. If the instance initialization method is in class Object, then the fragment includes no invocations of an instance initialization method on the object being initialized.*

Now we can state when the static type of a local variable or an operand stack entry ensures that it contains a raw object.

Theorem 5. *Assume that* $(stat_1, statag_1) \Longrightarrow (stat_2, statag_2) \Longrightarrow \cdots$ *is an execution and* $\mathcal{X} \in \{lvs_h, stk_h\}$ *and* $\mathcal{XT} \in \{lvsty_h, stkty_h\}$ *with* $h \geq 1$ *are such that* $\mathcal{X}$ *is* lvs_h *if and only if* $\mathcal{XT}$ *is* $lvsty_h$ *(and thus* $\mathcal{X}$ *is* stk_h *if and only if* $\mathcal{XT}$ *is* $stkty_h$*).*

- *If* $\mathcal{XT}(k) = unin(pp, cnam)$ *holds for some* k*,* pp *and* $cnam$*, then* $\mathcal{X}(k)$ *contains a reference to an uninitialized object of the class* $cnam$ *created by a* `new` *at* pp*.*
- *If* $\mathcal{XT}(k) = init(cnam)$ *holds for some* k *and* $cnam$*, then* $\mathcal{X}(k)$ *contains a reference to an object of* $cnam$ *that is being initialized inside an* `<init>` *and has not been initialized by another* `<init>`*.*

The following lemma shows that it is impossible for two different local variables/operand stack entries at a program point to have the same static type $unin(pp, cnam)$ for some pp and $cnam$ but hold references to different uninitialized objects. In fact, the lemma states the correctness of the typing rule for `invokespecial` on an instance initialization method, i.e., that if an object in a local variable/operand stack entry with the static type $unin(pp, cnam)$ is initialized, then all occurrences of $unin(pc, cnam)$ can be replaced by $cnam$.

Lemma 1. *Assume that* $(stat_1, statag_1) \Longrightarrow (stat_2, statag_2) \Longrightarrow \cdots$ *is an execution and* $\mathcal{X}, \mathcal{Y} \in \{lvs_h, stk_h\}$ *and* $\mathcal{XT}, \mathcal{YT} \in \{lvsty_h, stkty_h\}$ *with* $h \geq 1$ *such that* $\mathcal{X}$ *is* lvs_h *if and only if* $\mathcal{XT}$ *is* $lvsty_h$*, and* $\mathcal{Y}$ *is* lvs_h *if and only if* $\mathcal{YT}$ *is* $lvsty_h$*. Then the following conditions cannot hold at the same time for the indices* k *and* k'*:*

- $\mathcal{XT}(k) = \mathcal{YT}(k') = unin(pp, cnam)$ *holds for certain* pp *and* $cnam$*.*
- $\mathcal{X}(k)$ *and* $\mathcal{Y}(k')$ *contain references to different uninitialized objects created by the same* `new` *at* pp*.*

Now we know that if a memory location has a class as a static type, then it always holds initialized object or *null*.

Theorem 6. *Assume that* $(stat_1, statag_1) \Longrightarrow (stat_2, statag_2) \Longrightarrow \cdots$ *is an execution and* $\mathcal{X} \in \{lvs_h, stk_h\}$ *and* $\mathcal{XT} \in \{lvsty_h, stkty_h\}$ *with* $h \geq 1$ *such that* $\mathcal{X}$ *is* lvs_h *if and only if* $\mathcal{XT}$ *is* $lvsty_h$*. If* $\mathcal{XT}(k) = cnam$ *holds for some* k *and* $cnam$*, then* $\mathcal{X}(k)$ *contains a reference to an initialized object of* $cnam$ *or* *null*.

The typing rules for an instruction specify precisely how the instruction behaves on an uninitialized object. The following theorem summarizes some of the results:

Theorem 7. *1. A reference to an uninitialized object cannot be used in an instruction described in our formal specification except it is* `dup` `aload`*,* `astore` *or* `invokespecial`*. In the case of* `invokespecial`*, the method must be* `<init>`*, the object is the one being initialized and must be of the same class as the* `<init>`*.*

2. *Inside a method* `<init>` *that is not declared in the class Object, there must be a call to another* `<init>` *on the object being initialized via an* `invokespecial`*, where the called* `<init>` *is declared in the same class as or in an immediate superclass of that of the calling* `<init>`*. Before this call, the object being initialized cannot be used in an instruction described in our formal specification except it is* `dup`*,* `aload` *or* `astore`*.*

13 Conclusion

We have shown a formal specification of a substantial subset of JVM instructions. The formal specification clarifies some ambiguities and incompleteness and removes some (in our view) unnecessary restrictions in the description of the official Java Virtual Machine Specification [10].

Finally, it is worth mentioning that our study on the semantics of the JVM in this chapter led to the discovery of a possibility of writing a constructor that invoked no other constructor in the JDK 1.1.4 implementation of the JVM, which is clearly an implementation bug with respect to the official Java Virtual Machine Specification (page 122).

Acknowledgement

Thanks to Gilad Bracha and David von Oheimb for clarifying comments and useful feedback, and Masami Hagiya for pointing out an error in the description.

References

1. A. Aho, R. Sethi, and J. Ullman. *Compilers — Principles, Techniques, and Tools.* Addison-Wesley Publishing Company, 1986.
2. R. Cohen. The Defensive Java Virtual Machine specification. Technical report, Computational Logic inc., 1997.
3. D. Dean. The security of static typing with dynamic linking. In *Proc. 4th ACM Conf. on Computer and Communications Security.* ACM, 1996.
4. S. Dossopoulou and S. Eisenbach. Java is type safe — probably. In *Proc. 11th European Conf. on Object-Oriented Programming*, pages 389–418. Springer-Verlag LNCS 1241, 1997.
5. S. Freund and J. Mitchell. A type system for object initialization in the java bytecode language. Presneted at Int. Workshop on Security and Languages, Oct. 1997.
6. S. Freund and J. Mitchell. A type system for object initialization in the java bytecode language (summary). *Electronic Notes in Theoretical Computer Science*, 10, 1998. `http://www.elsevier.nl/locate/entcs/volume10.html`.
7. A. Goldberg. A specification of Java loading and bytecode verification. 1997.
8. J. Gosling, B. Joy, and G. Steele. *The Java™ Language Specification.* Addison-Wesley, 1996.
9. M. Hagiya. On a new method fot dataflow analysis of Java Virtual Machine subroutines. 1998.

10. T. Lindholm and F. Yellin. *The Java™ Virtual Machine Specification*. Addison-Wesley, 1996.
11. T. Nipkow and D. von Oheimb. Java$_{\ell ight}$ is type-safe — definitely. In *Proc. 25st ACM Symp. Principles of Programming Languages*, 1998.
12. Z. Qian. A formal specification of Java™ Virtual Machine instructions. Technical report, FB Informatik, Universität Bremen, September 1997. Revised version to appear June 1998.
13. V. Saraswat. Java is not type-safe. Technical report, AT&T Research, 1997.
14. E. Sirer, S. McDirmid, and B. Bershad. A Java system security architecture. `http://kimera.cs.washington.edu/`, 1997.
15. R. Stata and M. Abadi. A type system for Java bytecode subroutines. In *Proc. 25st ACM Symp. Principles of Programming Languages*, 1998.
16. D. Syme. Proving Java type soundness. Technical report, University of Cambridge Computer Laboratory, 1997.

The Operational Semantics of a Java Secure Processor

Pieter H. Hartel[1], Michael J. Butler[1], and Moshe Levy[2]

[1] Dept. of Electronics and Computer Science, Univ. of Southampton, UK
{phh,mjb}@ecs.soton.ac.uk
[2] JavaSoft, Inc. A Sun Microsystems, Inc. Business, Palo Alto, CA 94043 USA
Moshe.Levy@Sun.COM

Abstract. A formal specification of a Java Secure Processor is presented, which is mechanically checked for type consistency, well formedness and operational conservativity. The specification is executable and it is used to animate and study the behaviour of sample Java programs. The purpose of the semantics is to document the behaviour of the complete JSP for the benefit of implementors.

1 Introduction

A smart card is a complete 'embedded' computer housed in a piece of plastic the same size as a credit card [12]. The computer has to be small to reduce the risk of mechanical problems. Because of these mechanical constraints, as well as aspects of cost, the current generation of smart cards typically contains only a small 8-bit micro processor, a few hundred bytes of RAM, a few Kbytes of ROM and a few Kbytes of EEPROM. This small size constrains the freedom in the design of the software that has to be run on a smart card processor.

Java [4] was originally designed for writing embedded software. Because of this pedigree it is attractive as a smart card programming language. Some facilities provided by the Java language are too expensive to be implemented on a smart card. Threads, and dynamic class file loading fall in this category. Further study is needed to find ways of incorporating the Java exception mechanism and a garbage collector on smart cards. Smart cards do not use floating point arithmetic so this feature of Java is not needed. Using the subset of Java as described above for smart cards is attractive. It is also feasible to implement this Java subset on computers with limited resources.

The standard Java class libraries are not suitable for smart cards because many of the facilities provided are meaningless on smart cards. Examples include the interface to GUI libraries. Instead a smart card would host a specially designed set of class libraries dedicated to the application domain of card applications. The set of class libraries would be small enough to fit in the card and would be versatile enough to provide standard smart card facilities, such as the ISO 7816-4 command set [1], or down loadable applications for multi-application smart cards [9].

Jim Alves-Foss (Ed.): Formal Syntax and Semantics of Java, LNCS 1523, pp. 313–351, 1999.

A Java Secure Processor (JSP) is a virtual machine that is designed to fit on a smart card. A JSP does not implement the full Java Virtual Machine (JVM) [7]. Instead a JSP is accompanied by a JVM to JSP translator, which compiles standard JVM byte codes into byte codes for the JSP. Java Soft has written a sophisticated translator, which performs extensive program analysis to allow a large class of Java programs to be run on the JSP. To support our work on the formal definition of the operational semantics of the JSP we have written a simple translator, which accepts a smaller class of Java programs. The simple translator is used to validate the operational semantics.

A standard Java development environment can be used to write Java programs for smart cards. Instead of relying on the standard class libraries the programmer uses the smart card class libraries. A simulator can be used to test the code. The process of loading Java programs into a card is quite different from loading and running programs on a workstation, as it may involve manufacturing ROM masks. We will not discuss this aspect further, the interested reader is referred to the literature [12].

A smart card is a secure token that may control commodities of real value. Secure here means that the card should be hardware and software tamper resistant, and that it should not leak information. The considerations that apply to the security of Java in general [8] also apply to Java for smart cards. In addition, Java for smart cards should provide facilities such as ownership control and cryptographically protected modes of use.

The resource limitation of a smart card makes it more difficult to ensure that security is maintained. For example currently a complete byte code verifier is too large to be implemented on a smart card. The JSP approach assumes that JVM byte codes are verified when translated into JSP byte codes. The results are then digitally signed so that tampering can be detected when code is being loaded.

A clear, concise and complete specification of the semantics is a prerequisite for a successful and secure implementation of a JSP. The present document provides such a specification. The document is based on an informal description of the JSP from Java Soft, who are currently building a tool suite for a JSP [6]. The formal specification is self contained but does not document the motivation for many of the design decisions made for the JSP. The interested reader is referred to the informal specification.

The present formal specification is a `latos` [5] literate script. `Latos` is a tool for developing operational semantics. `Latos` supports publication quality rendering using LaTeX, execution and animation using a functional programming language, and derivation tree browsing using Netscape. `Latos` helps to check that a specification is operationally conservative. The `latos` meta language is basically Miranda[1] [11] augmented with a notation for rules of inference and sets. Developing a semantics as a literate script avoids clerical errors and confusion, as syntax and type errors are detected by the tool.

[1] Miranda is a trademark of Research Software Ltd.

The formal specification does not support the capabilities of JSP development environment. Instead the `latos` tool provides a tracing facility allowing for a detailed study and analysis of executing application programs.

Related work on the semantics of the JVM includes the executable specification of the 'defensive' JVM made by Computational Logic Inc [3], work by Bertelsen on another subset of the JVM [2], and also other chapters of this book.

The next section describes the restrictions imposed on the kind of Java programs supported by a JSP on a smart card. Section 3 presents the execution model of a JSP and Section 4 defines the instructions of the virtual machine. The relationship between the JVM and the JSP is explored in Section 5. A brief example of how the semantics of the JSP may be used to validate the behaviour of a sample Java program is given in Section 6. The last section presents our conclusions and suggestions for future work.

2 Java Language Restrictions

The JSP design imposes a number of restrictions to allow a Java program to be run in the constrained runtime environment of a smart card. The most important restrictions are:

- The JSP provides no support for threads, multi-dimensional arrays, floating point numbers, and Just-in-time byte code translation.
- Exceptions may be raised by application programs, but they can only be handled by the system.
- There is no garbage collection. Objects can be allocated dynamically but the majority are expected to be allocated statically using compile time garbage collection techniques. The formal specification allows objects to be allocated any time. It would be possible to state and prove a property about programs that are guaranteed not to allocate objects after a certain point in their execution. This constitutes a desirable safety property of those programs.
- Class files cannot be loaded dynamically. Instead the software to be present in the card is loaded when the card is manufactured or personalised.
- Recursive methods are discouraged and recursive class constructors and exceptions are disallowed.
- Integers and shorts are identified. The JVM to JSP byte code translator should ensure that the results obtained from a computation on the JSP are identical to the results that would have been obtained on a JVM.
- The number of arguments, local variables, methods, and object instances are limited.

Java programmers have to be aware of these restrictions when writing code that is intended for a JSP. Some of the restrictions can be circumvented by the use of appropriate class libraries. Others will be taken care of by program analysis techniques in the JVM to JSP translator.

3 Execution Model

The JSP is a byte oriented stack machine. It also has a read-only memory area for storing methods and constants, an area of memory and some registers to maintain the book-keeping of the machine, and a heap.

The data manipulated directly by Java programs is faithfully modelled by the semantics. In particular the operand stack, the fields of objects and the elements of arrays contain bytes only. A short or a reference is always treated as a pair of bytes. The structures that support the machine itself, such as the byte codes, stack frames and heap objects are modelled as higher level entities rather than as collections of bytes. The ensuing specification is of a low level, which makes it eminently suitable to serve as a guideline for implementors of a JSP.

The formal specification defines all structured data (not scalars) of the virtual machine either as (partial) mappings or as algebraic data types (i.e., a sum of product types). Each of these is of a different type, that is incompatible with any other useful type. The `latos` system performs strong type checking to ensure that all the type constraints in the operational semantics are indeed satisfied.

3.1 Basic Data

The basic data in the formal specification are derived from the natural numbers. Similarly, the raw data in the JSP implementation are derived from a sequence of bytes. The type bit (below) permits any numeric value, but sensible values are in the range $0 \ldots 1$. (The equivalence symbol is used to bind a name to a type, the equals symbol binds a name to a value). In a JSP implementation, a boolean is stored in a byte, which permits sensible values as well as non-sensible values. We would have preferred to identify bit and $\mathsf{bit}_{\mathsf{range}}$ but unfortunately the type system used by `latos` (i.e., the Hindley-Milner type system of Miranda) is not strong enough to support sub types.

$\mathsf{bit} \equiv \mathsf{num};$

$\mathsf{bit}_{\mathsf{range}} = 0 \ldots 1;$

Other raw data and ranges defined in a similar way include the signed 8-bit byte, the signed 16-bit short and the unsigned 16-bit reference. The nullreference is a special reference value, which is represented as zero. Regular references should not have this particular value.

3.2 Store Areas

The JSP virtual machine uses a number of areas of store for data, code and book-keeping. Each of these areas is represented in the formal specification as a mapping of numerical indices onto values of the appropriate type, thus providing a uniform, albeit low level approach to information handling in the JSP.

– A JSP uses a stack of activation frames, where each frame contains an operand stack and some book-keeping. The activation frames are gathered in the machine-wide frameArea. The frame area is represented as a partial

mapping from the domain framePointer to the range frame. The representation as a partial function makes it possible to represent common operations on structures in a clear and succinct way. The type frame itself is defined in Section 3.3.

framePointer $\equiv$ num;
framePointer$_{\text{range}} = 0 \ldots 255$;
frameArea $\equiv$ framePointer $\rightharpoonup$ frame;

- Heap objects are instances of classes or arrays. The objects are gathered in the machine-wide heapArea. The type object is defined in Section 3.5.
  heapPointer $\equiv$ num;
  heapPointer$_{\text{range}} = 0 \ldots 65535$;
  heapArea $\equiv$ heapPointer $\rightharpoonup$ object;
- Static program data are represented by bytes. This data is gathered in the machine-wide staticArea.
  staticPointer $\equiv$ num;
  staticPointer$_{\text{range}} = 0 \ldots 65535$;
  staticArea $\equiv$ staticPointer $\rightharpoonup$ byte;
- The machine-wide codeArea gathers the byte codes and the method headers for the methods of all application programs in the system. The type byteCode is defined in Section 4.
  programCounter $\equiv$ num;
  programCounter$_{\text{range}} = 0 \ldots 65535$;
  codeArea $\equiv$ programCounter $\rightharpoonup$ byteCode;
- The application program table progTable records the class table of each loaded application program.
  progId $\equiv$ num;
  progId$_{\text{range}} = 0 \ldots 63$;
  progTable $\equiv$ progId $\rightharpoonup$ classTable;
- There is one instance of class Class for each class in the system. The class table gathers such instances. The type classObject is defined in Section 3.5.
  classId $\equiv$ num;
  classId$_{\text{range}} = 0 \ldots 127$;
  classTable $\equiv$ classId $\rightharpoonup$ classObject;
- Each class in the system is accompanied by a method table, which maps a method id onto the program counter value at which the method header is located. The methodTable is defined as an algebraic data type with two components and with constructor **MethodTable**.
  methodId $\equiv$ num;
  methodId$_{\text{range}} = 0 \ldots 255$;
  entryTable $\equiv$ methodId $\rightharpoonup$ programCounter;
  methodTable $\equiv$ **MethodTable** classId entryTable;

3.3 Stack Frames

The operand stack within the topmost frame plays a special role in that it can be accessed by the JSP instructions. To acknowledge this special role, the

formal specification shadows the operand stack, and manipulates it as a separate component of the virtual machine configuration.

A method invocation creates a stack frame (shown below as an instance of the data type frame). The frame has the following four components:

- the programCounter representing the return address to the caller of the method.
- the framePointer to the previous frame. This information is redundant in the specification, as frames are numbered sequentially starting from 0. In an implementation frames would be referred to by their address, in which case the frame pointer is needed.
- the stackPointer within the operand stack; local and temporary variables of the current method.

In the specifications that follow, a stack is always accompanied by a stack pointer (which points at the last used element). All stack operations can be modelled by a combination of adding (or subtracting) a constant to (from) the stack pointer and/or updating the mapping. For example, pushing an element onto the stack means incrementing the pointer and updating the mapping with a new association.

$$
\begin{aligned}
&\mathsf{stackPointer} && \equiv \mathsf{num};\\
&\mathsf{stackPointer}_{\mathsf{range}} && = 0 \ldots 255;\\
&\mathsf{operandStack} && \equiv \mathsf{stackPointer} \rightharpoonup \mathsf{byte};\\
&\mathsf{frame} && \equiv \textbf{Frame}\ \mathsf{programCounter}\ \mathsf{framePointer}\\
& && \quad\ \mathsf{stackPointer}\ \mathsf{operandStack};
\end{aligned}
$$

A JSP uses a slightly different stack frame configuration than the JVM, a difference that is taken into account by the JVM to JSP byte code translator.

3.4 Headers

Objects and methods have headers, which record book-keeping information. This section describes all possible headers in the system.

- An objectHeader records the identity of the application program progId, the size of the object in bytes instanceSize and a table listing all the methods for the object. The classId for the object is available from the methodTable.

$$
\begin{aligned}
&\mathsf{instanceSize} && \equiv \mathsf{num};\\
&\mathsf{instanceSize}_{\mathsf{range}} && = 0 \ldots 127;\\
&\mathsf{objectHeader} && \equiv \textbf{ObjectHeader}\ \mathsf{progId}\ \mathsf{instanceSize}\ \mathsf{methodTable};
\end{aligned}
$$

- An array may contain scalars (bits, bytes or shorts) or references to objects. An array has a header, which records the application program id, the class id of the element type, the method table for the element type, an indication of the element type and the length of the array.

$$
\begin{aligned}
&\mathsf{dataType} && \equiv \textbf{bit} \mid \textbf{byte} \mid \textbf{short} \mid \textbf{ref};\\
&\mathsf{arrayLength} && \equiv \mathsf{num};\\
&\mathsf{arrayLength}_{\mathsf{range}} && = 1 \ldots 4096;\\
&\mathsf{arrayHeader} && \equiv \textbf{ArrayHeader}\ \mathsf{progId}\ \mathsf{classId}\ \mathsf{methodTable}\\
& && \quad\ \mathsf{dataType}\ \mathsf{arrayLength};
\end{aligned}
$$

– A method has a header, which records two flags and three sizes. The flags record whether the method is native and whether it is public. The stack size is currently unused, but the paramsSize and localsSize are used to create appropriate frames. Stack frames are limited in size due to the limitations on available RAM space in smart cards.

$$
\begin{aligned}
&\mathsf{isNative} && \equiv \mathsf{bool};\\
&\mathsf{isPublic} && \equiv \mathsf{bool};\\
&\mathsf{stackSize} && \equiv \mathsf{num};\\
&\mathsf{stackSize}_{\mathsf{range}} && = 0 \ldots 15;\\
&\mathsf{paramsSize} && \equiv \mathsf{num};\\
&\mathsf{paramsSize}_{\mathsf{range}} && = 0 \ldots 15;\\
&\mathsf{localsSize} && \equiv \mathsf{num};\\
&\mathsf{localsSize}_{\mathsf{range}} && = 0 \ldots 15;\\
&\mathsf{methodHeader} && \equiv \textbf{MethodHeader}\ \mathsf{isNative}\ \mathsf{isPublic}\\
& && \quad \mathsf{stackSize}\ \mathsf{paramsSize}\ \mathsf{localsSize};
\end{aligned}
$$

3.5 Objects

The JSP works with three different kinds of objects:

– A regular object has a header and a number of fields represented by the fieldTable. The fields are represented as bytes and the methods are available from the header.

$$
\begin{aligned}
&\mathsf{fieldId} && \equiv \mathsf{num};\\
&\mathsf{fieldId}_{\mathsf{range}} && = 0 \ldots 255;\\
&\mathsf{fieldTable} && \equiv \mathsf{fieldId} \rightharpoonup \mathsf{byte};\\
&\mathsf{regularObject} && \equiv \textbf{RegularObject}\ \mathsf{objectHeader}\ \mathsf{fieldTable};
\end{aligned}
$$

– An array object records an array header as well as the array elements. The elements are represented as bytes.

$$
\begin{aligned}
&\mathsf{arrayIndex} && \equiv \mathsf{num};\\
&\mathsf{arrayIndex}_{\mathsf{range}} && = 0 \ldots 4095;\\
&\mathsf{arrayTable} && \equiv \mathsf{arrayIndex} \rightharpoonup \mathsf{byte};\\
&\mathsf{arrayObject} && \equiv \textbf{ArrayObject}\ \mathsf{arrayHeader}\ \mathsf{arrayTable};
\end{aligned}
$$

– There is one classObject for every object in the system. The classObject itself is an instance of class Class. The classObject records the normal object header as well as the size of an instance of the class, the method table for the class, the depth in the class hierarchy, the classId of the super classes and the interface classes implemented by the class. The instance size and the method table are redundant as the object header also contains this information.

classDepth $\equiv$ num;
classDepth$_{\text{range}}$ $= 0 \ldots 255$;
superId $\equiv$ num;
superId$_{\text{range}}$ $= 0 \ldots 255$;
superTable $\equiv$ superId $\rightharpoonup$ classId;
interfaceId $\equiv$ num;
interfaceId$_{\text{range}}$ $= 0 \ldots 255$;
implementTable $\equiv$ methodId $\rightharpoonup$ methodId;
interfaceTable $\equiv$ interfaceId $\rightharpoonup$ implementTable;
classObject $\equiv$ **ClassObject** objectHeader instanceSize methodTable classDepth superTable interfaceTable;

The JSP heap is used to store regular and array objects only. A classObject is allocated statically in a area separate from the heap. The union type object therefore does not cover class objects.

object $\equiv$ regularObject | arrayObject;

The two auxiliary predicates below are used to determine whether we are dealing with a regular object or an array object.

isRegularObject regularObject = True;
isRegularObject arrayObject = False;
isArrayObject arrayObject = True;
isArrayObject regularObject = False;

4 Instruction Set

There are 25 different categories of JSP byteCode (below), all with their own type. The methodHeader is treated as a pseudo instruction. This models the practice of preceding the code for each method by its header.

byteCode $\equiv$ methodHeader |
constInst | loadInst | storeInst | incInst | stackInst |
newarrayInst | arrayLoadInst | arrayStoreInst |
arithInst | logicalInst | convertInst | compareInst |
controlInst | switchInst | exceptionInst |
invokeinterfaceInst | invokevirtualInst |
invokeInst | returnInst |
objectInst | instanceInst |
getfieldInst | putfieldInst | getstaticInst | putstaticInst |
breakpointInst;

The following categories of byte codes have been defined:

– Load, store and increment instructions.

constInst ≡ **nop** | **bpush** byte | **spush** byte byte | **apush** byte byte |
aconst$_{\mathsf{null}}$ | **bconst**$_{\mathsf{m1}}$ |
bconst$_0$ | **bconst**$_1$ | **bconst**$_2$ | **bconst**$_3$ | **bconst**$_4$ | **bconst**$_5$;
loadInst ≡ **bload** stackPointer | **bload**$_0$ | **bload**$_1$ | **bload**$_2$ | **bload**$_3$ |
sload stackPointer | **sload**$_0$ | **sload**$_1$ | **sload**$_2$ | **sload**$_3$ |
aload stackPointer | **aload**$_0$ | **aload**$_1$ | **aload**$_2$ | **aload**$_3$;
storeInst ≡ **bstore** stackPointer | **bstore**$_0$ | **bstore**$_1$ | **bstore**$_2$ | **bstore**$_3$ |
sstore stackPointer | **sstore**$_0$ | **sstore**$_1$ | **sstore**$_2$ | **sstore**$_3$ |
astore stackPointer | **astore**$_0$ | **astore**$_1$ | **astore**$_2$ | **astore**$_3$;
incInst ≡ **binc** stackPointer byte | **sinc** stackPointer byte;

– Stack instructions.

stackInst ≡ **pop** | **pop2** | **dup** | **dup2** | **dup_x** byte | **swap** | **swap2**;

– Array creation, load and store instructions.

newarrayInst ≡ **newarray** dataType | **anewarray** classId;
arrayLoadInst ≡ **arraylength** | **baload** | **saload** | **aaload**;
arrayStoreInst ≡ **bastore** | **sastore** | **aastore**;

– Instructions for arithmetical, logical, conversion and comparison operations.

arithInst ≡ **bneg** | **sneg** | **badd** | **sadd** | **bsub** | **ssub** |
bmul | **smul** | **bdiv** | **sdiv** | **brem** | **srem**;
logicalInst ≡ **bshl** | **bshr** | **bushr** | **sshl** | **sshr** | **sushr** |
band | **sand** | **bor** | **sor** | **bxor** | **sxor**;
convertInst ≡ **s2b** | **b2s**;
compareInst ≡ **bcmp** | **scmp** | **acmp**;

– Instructions for the transfer of control.

offset ≡ (byte, byte);
controlInst ≡ **ifeq** offset | **iflt** offset | **ifgt** offset |
ifne offset | **ifge** offset | **ifle** offset | **goto** offset;

– Instructions to support switch statements.

tableswitchIndex ≡ num;
tableswitchIndex$_{\mathsf{range}}$ = 0 ... 127;
tableswitchTable ≡ tableswitchIndex ⇀ offset;
lookupswitchIndex ≡ num;
lookupswitchIndex$_{\mathsf{range}}$ = 0 ... 126;
lookupswitchTable ≡ lookupswitchIndex ⇀ (byte, offset);
switchInst ≡ **tableswitch** offset byte byte tableswitchTable |
lookupswitch offset byte lookupswitchTable;

– Instructions to support exceptions.

exceptionInst ≡ **athrow** | **jsr** offset | **ret** stackPointer;

– Instructions for method invokation.

invokeinterfaceInst ≡ **invokeinterface** paramsSize interfaceId methodId;
invokevirtualInst ≡ **invokevirtual** paramsSize methodId;
invokeInst ≡ **invoke** offset;
returnInst ≡ **breturn** | **sreturn** | **areturn** | **return**;

– Instructions for object creation and manipulation.

objectInst ≡ **new** classId;
instanceInst ≡ **instanceof** classId | **checkcast** classId |
ainstanceof dataType | **acheckcast** dataType |
aainstanceof classId | **aacheckcast** classId;
getfieldInst ≡ **bgetfield** stackPointer | **sgetfield** stackPointer;
putfieldInst ≡ **bputfield** stackPointer | **sputfield** stackPointer;
getstaticInst ≡ **bgetstatic** byte byte | **sgetstatic** byte byte;
putstaticInst ≡ **sputstatic** byte byte | **bputstatic** byte byte;

– Miscellaneous instructions.
breakpointInst ≡ **breakpoint**;

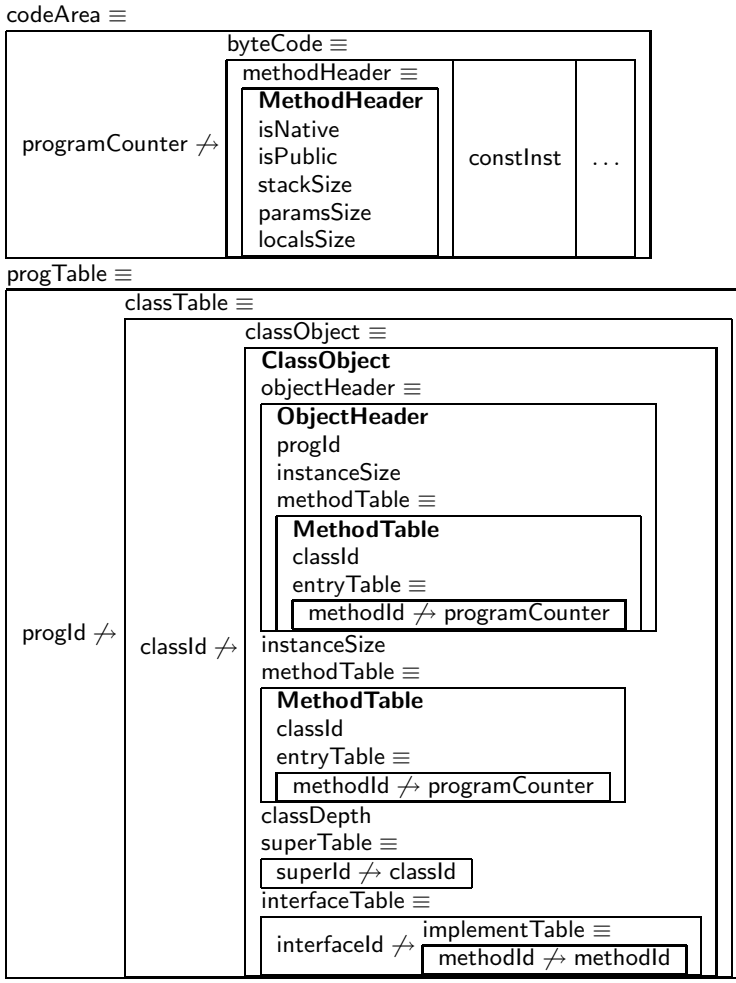


Fig. 1. Read only structures.

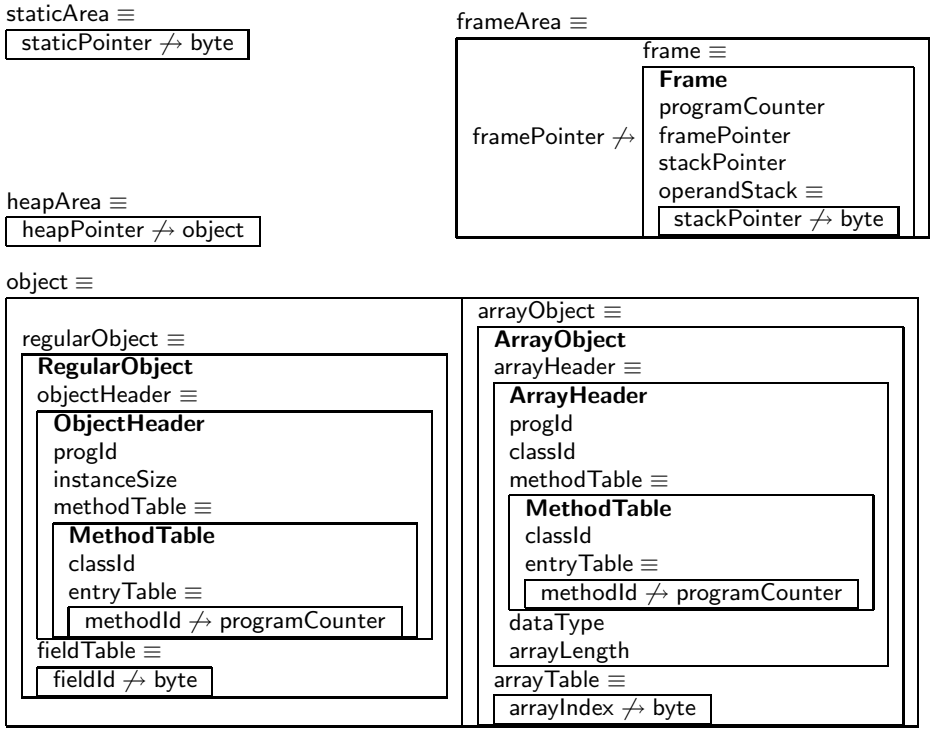


Fig. 2. Structures that can be written to.

We have now completed the definition of the JSP machine structures. To assist the reader retrieving a particular definition, Figures 1 and 2 summarise the read only structures and the structures that are written to during execution of a JSP program respectively. For each of the three different kinds of structures that we have used, the name is given (followed by an $\equiv$ symbol) and a suggestive graphical representation. The partial maps are shown in a single box, with the domain to the left of the $\rightharpoonup$ symbol and the range to the right. A product data type is shown as a sequence of vertically stacked boxes, one for each component. A sum data type is shown as a horizontally arranged sequence of boxes.

The following sections present the semantic rules for a representative selection of the JSP byte codes. Since there are many groups of similar byte codes, we consider it justified to give the rule for just one member of each group without sacrificing the rigour of the specification.

4.1 Pushing Constants onto the Stack

The stack is controlled by the stack pointer, which points at the last used location. A short occupies two consecutive locations in the stack, with the high byte at the lowest stack pointer index (bigendian).

Table 1. Labelled equality relations. The type given is that of the two operands.

$\stackrel{ah}{\Rightarrow}$	arrayHeader
$\stackrel{at}{\Rightarrow}$	arrayTable
$\stackrel{b}{\Rightarrow}$	byte
$\stackrel{ct}{\Rightarrow}$	classTable
$\stackrel{fa}{\Rightarrow}$	frameArea
$\stackrel{ft}{\Rightarrow}$	fieldTable

$\stackrel{ha}{\Rightarrow}$	heapArea
$\stackrel{it}{\Rightarrow}$	implementTable
$\stackrel{ob}{\Rightarrow}$	object
$\stackrel{oh}{\Rightarrow}$	objectHeader
$\stackrel{os}{\Rightarrow}$	operandStack
$\stackrel{p}{\Rightarrow}$	(byte, byte)
$\stackrel{ps}{\Rightarrow}$	[(byte, byte)]

$\stackrel{s}{\Rightarrow}$	short
$\stackrel{hp}{\Rightarrow}$	heapPointer
$\stackrel{pc}{\Rightarrow}$	programCounter
$\stackrel{sa}{\Rightarrow}$	staticArea
$\stackrel{f}{\Rightarrow}$	frame
$\stackrel{bc}{\Rightarrow}$	byteCode

The relation $\stackrel{const}{\Rightarrow}$ below describes the effects of each of the instructions dealing with constants on the stack. The type of the relation shows that in addition to the instruction itself, only the stack pointer and the operand stack are relevant here. The left operand of the relation specifies the machine components that are accessed, the right operand mentions those that may be changed by the instruction. Specifying the types of the relations thus provides an aid in the documentation of the system. The types of all relations of the JSP transition system are summarised in Table 2. We will not give the explicit types of the remaining relations.

$\mathsf{lhs}_{\mathsf{const}} \equiv \langle \mathsf{constInst},\ \mathsf{stackPointer},\ \mathsf{operandStack} \rangle$;
$\mathsf{rhs}_{\mathsf{const}} \equiv \langle \mathsf{stackPointer},\ \mathsf{operandStack} \rangle$;
$\stackrel{const}{\Rightarrow}$ :: $(\mathsf{lhs}_{\mathsf{const}} \leftrightarrow \mathsf{rhs}_{\mathsf{const}})$;

The rules for **nop**, **bpush** and **spush** below reveal most aspects of the notation that we are using. The semantics of an instruction is defined by an axiom or a rule of inference. The text in square brackets to the left of the axiom/rule is a label to identify the rule. A rule has a number of premises (above the horizontal line) and a conclusion. An axiom has a conclusion but no premises. Rules and axioms may have side conditions. The two axioms and the rule below together define the relation $\stackrel{const}{\Rightarrow}$ over components of the JSP virtual machine configurations.

[nop] $\quad \vdash \langle \mathbf{nop},\ \mathsf{sp},\ \mathsf{os} \rangle \stackrel{const}{\Rightarrow} \langle \mathsf{sp},\ \mathsf{os} \rangle$;

[bpush] $\vdash \langle \mathbf{bpush}\ \mathsf{v},\ \mathsf{sp},\ \mathsf{os} \rangle \stackrel{const}{\Rightarrow} \langle \mathsf{sp}+1,\ \mathsf{os} \oplus \{\mathsf{sp}+1 \mapsto \mathsf{v}\} \rangle$,
if $(\mathsf{sp}+1) \in\ \mathsf{stackPointer}_{\mathsf{range}}$;

[spush] $\dfrac{\vdash \mathsf{os} \oplus \{\mathsf{sp}+1 \mapsto \mathsf{hi}\} \oplus \{\mathsf{sp}+2 \mapsto \mathsf{lo}\} \stackrel{os}{\Rightarrow} \mathsf{os}'}{\vdash \langle \mathbf{spush}\ \mathsf{hi}\ \mathsf{lo},\ \mathsf{sp},\ \mathsf{os} \rangle \stackrel{const}{\Rightarrow} \langle \mathsf{sp}+2,\ \mathsf{os}' \rangle}$,
if $(\mathsf{sp}+1 \ldots \mathsf{sp}+2) \subseteq\ \mathsf{stackPointer}_{\mathsf{range}}$;

The configuration on the left hand side of the arrow consists of an instruction and its operands (eg. **spush** hi lo), the current stack pointer (sp), and the operand stack (os). Other components of the JSP machine, such as the heap are not used by the three rules above.

The configuration on the right hand side consists of the next value of the stack pointer (eg. $\mathsf{sp}+2$) and the new operand stack (os'). Some of the components

mentioned on the left hand side are not present on the right hand side, because they are not changed by the instruction. We have been careful in exposing only the information required, so as to improve the clarity and succinctness of the specification.

The premise of the **spush** rule asserts a relationship between components of the old and the new configuration. The relation $\stackrel{os}{\Rightarrow}$ is an equality relation, which holds when the operands are both of type $\mathsf{operandStack}$. Labelling equalities with the type of the operands helps the mechanical type checker spot clerical errors. Many other labelled equalities are used throughout. The labels and the types of the operands are summarised in Table 1. The actual definition of the relations is omitted.

The notation $\mathsf{os} \oplus \{\mathsf{sp}+1 \mapsto \mathsf{v}\}$ extends the mapping os with a new domain/range pair. Any previous association for the new domain value $\mathsf{sp}+1$ is lost. It follows that it is sufficient to decrement the stack pointer to 'forget' mappings for particular values in the domain. Furthermore, we do not in general have the invariant $\mathsf{domain}(\mathsf{os}) = 0 \ldots \mathsf{sp}$.

The side condition for the **bpush** and **spush** operations determines when it is safe to extend the stack. If it is not safe, then the relation $\stackrel{const}{\Rightarrow}$ does not hold.

The rule for the **apush** operation is not shown here because it is identical to that of the **spush** operation: an address is a numeric value and therefore indistinguishable from a short. In a typed version of the JSP the instructions would not be the same.

4.2 Pushing Immediate Constants

Some constants are needed so often that special instructions have been defined to push them onto the stack. The semantics of the specialised instructions such as $\mathbf{bconst}_0$ (below) is defined in terms of the general operation **bpush**. The rules for the remaining instructions $\mathbf{aconst}_{\mathsf{null}}$, $\mathbf{bconst}_{\mathsf{m1}}$, $\mathbf{bconst}_1 \ldots \mathbf{bconst}_5$ (not shown) are defined in a similar way.

$$[\mathsf{bconst}_0]\ \frac{\vdash \langle \mathbf{bpush}\ 0,\ \mathsf{sp},\ \mathsf{os}\rangle \stackrel{const}{\Rightarrow} \langle \mathsf{sp}',\ \mathsf{os}'\rangle}{\vdash \langle \mathbf{bconst}_0,\ \mathsf{sp},\ \mathsf{os}\rangle \stackrel{const}{\Rightarrow} \langle \mathsf{sp}',\ \mathsf{os}'\rangle};$$

4.3 Loading Local Variables onto the Stack

The load instructions transfer values from the parameter and local variable area of the stack frame to the top of the operand stack. Local variables and parameters are accessed via a fixed index from the bottom of the operand stack. The reader is reminded that the operand stack is just a portion of the current frame, but we view the operand stack separately from the frame for convenience.

The side conditions on the rules below check for stack overflow. There is no explicit check on the value of the index i because it is assumed that the static semantics of the byte codes, as enforced by the byte code verifier, will deal with illegal offsets.

Table 2. A summary of the types of all relations defining the transition system of the Java secure processor.

		programCounter	codeArea	byteCode	stackPointer	operandStack	framePointer	frameArea	heapPointer	heapArea	progId	progTable	staticArea	outputStream
$\overset{const}{\Rightarrow}$	constInst				rw	rw								
$\overset{load}{\Rightarrow}$	loadInst				rw	rw								
$\overset{store}{\Rightarrow}$	storeInst				rw	rw								
$\overset{inc}{\Rightarrow}$	incInst				r	rw								
$\overset{stack}{\Rightarrow}$	stackInst				rw	rw								
$\overset{newarray}{\Rightarrow}$	newarrayInst				r	rw			rw	rw	r	r		
$\overset{arrayload}{\Rightarrow}$	arrayloadInst				rw	r				r				
$\overset{arraystore}{\Rightarrow}$	arraystoreInst				rw	r				rw				
$\overset{arith}{\Rightarrow}$	arithInst				rw	rw								
$\overset{logical}{\Rightarrow}$	logicalInst				rw	rw								
$\overset{conv}{\Rightarrow}$	convInst				rw	rw								
$\overset{compare}{\Rightarrow}$	compareInst				rw	rw								
$\overset{control}{\Rightarrow}$	controlInst	rw			rw	rw								
$\overset{switch}{\Rightarrow}$	switchInst	rw			rw	rw								
$\overset{exception}{\Rightarrow}$	exceptionInst	rw			rw	rw								
$\overset{invokeinterface}{\Rightarrow}$	invokeinterfaceInst	rw	r		rw	rw	rw	rw		r	r	r		
$\overset{invokevirtual}{\Rightarrow}$	invokevirtualInst	rw	r		rw	rw	rw	rw		r				
$\overset{invoke}{\Rightarrow}$	invokeInst	rw	r		rw	rw	rw	rw						
$\overset{return}{\Rightarrow}$	returnInst	w			rw	rw	rw	r						
$\overset{object}{\Rightarrow}$	objectInst				rw	rw			rw	rw	r	r		
$\overset{instance}{\Rightarrow}$	instanceInst				rw	rw			r	r	r	r		
$\overset{getfield}{\Rightarrow}$	getfieldInst				rw	r				r				
$\overset{putfield}{\Rightarrow}$	putfieldInst				rw	r				rw				
$\overset{getstatic}{\Rightarrow}$	getstaticInst				rw	r							r	
$\overset{putstatic}{\Rightarrow}$	putstaticInst				rw	r							rw	
$\overset{breakpoint}{\Rightarrow}$	breakpointInst				rw	r								rw
$\overset{exec}{\Rightarrow}$	execInst	rw	r	r	rw	rw	rw	rw	rw	rw	r	r	rw	rw

$$\frac{\vdash \mathsf{os}(\mathsf{i}) \overset{b}{\Rightarrow} \mathsf{v}}{[\mathsf{bload}]\ \vdash \langle \textbf{bload}\ \mathsf{i},\ \mathsf{sp},\ \mathsf{os}\rangle \overset{load}{\Rightarrow} \langle \mathsf{sp}+1,\ \mathsf{os} \oplus \{\mathsf{sp}+1 \mapsto \mathsf{v}\}\rangle,\ \textbf{if}\ (\mathsf{sp}+1) \in \mathsf{stackPointer}_{\mathsf{range}};}$$

$$\frac{\vdash (\mathsf{os}(\mathsf{i}),\ \mathsf{os}(\mathsf{i}+1)) \overset{p}{\Rightarrow} (\mathsf{hi},\ \mathsf{lo}),\quad \vdash \mathsf{os} \oplus \{\mathsf{sp}+1 \mapsto \mathsf{hi}\} \oplus \{\mathsf{sp}+2 \mapsto \mathsf{lo}\} \overset{os}{\Rightarrow} \mathsf{os}'}{[\mathsf{sload}]\ \vdash \langle \textbf{sload}\ \mathsf{i},\ \mathsf{sp},\ \mathsf{os}\rangle \overset{load}{\Rightarrow} \langle \mathsf{sp}+2,\ \mathsf{os}'\rangle,\ \textbf{if}\ (\mathsf{sp}+1 \ldots \mathsf{sp}+2) \subseteq \mathsf{stackPointer}_{\mathsf{range}};}$$

The rule for **aload** and those for the specialised versions $\textbf{bload}_0 \ldots \textbf{bload}_3$, $\textbf{sload}_0 \ldots \textbf{sload}_3$ and $\textbf{aload}_0 \ldots \textbf{aload}_3$ are not shown here.

4.4 Storing Stack Values into Local Variables

The store instructions transfer values from the operand stack into parameter and local variable area of the stack frame. This time the side conditions check for stack underflow.

$$\frac{\vdash \mathsf{os}(\mathsf{sp}) \overset{b}{\Rightarrow} \mathsf{v}}{[\mathsf{bstore}]\ \vdash \langle \textbf{bstore}\ \mathsf{i},\ \mathsf{sp},\ \mathsf{os}\rangle \overset{store}{\Rightarrow} \langle \mathsf{sp}-1,\ \mathsf{os} \oplus \{\mathsf{i} \mapsto \mathsf{v}\}\rangle,\ \textbf{if}\ \mathsf{sp} \in \mathsf{stackPointer}_{\mathsf{range}};}$$

$$\frac{\vdash (\mathsf{os}(\mathsf{sp}-1),\ \mathsf{os}(\mathsf{sp})) \overset{p}{\Rightarrow} (\mathsf{hi},\ \mathsf{lo}),\quad \vdash \mathsf{os} \oplus \{\mathsf{i} \mapsto \mathsf{hi}\} \oplus \{\mathsf{i}+1 \mapsto \mathsf{lo}\} \overset{os}{\Rightarrow} \mathsf{os}'}{[\mathsf{sstore}]\ \vdash \langle \textbf{sstore}\ \mathsf{i},\ \mathsf{sp},\ \mathsf{os}\rangle \overset{store}{\Rightarrow} \langle \mathsf{sp}-2,\ \mathsf{os}'\rangle,\ \textbf{if}\ (\mathsf{sp}-1 \ldots \mathsf{sp}) \subseteq \mathsf{stackPointer}_{\mathsf{range}};}$$

The **astore** instruction is identical to the **sstore** instruction. The specialised instructions $\textbf{bstore}_0 \ldots \textbf{bstore}_3$, $\textbf{sstore}_0 \ldots \textbf{sstore}_3$ and $\textbf{astore}_0 \ldots \textbf{astore}_3$ are not shown here.

Table 3. Explicit conversions between arbitrary integers and shorts (n2s), arbitrary integers and bytes (n2b), between shorts and pairs of bytes (s2p, p2s), between booleans and bytes (b2b) and a range comparison operator ==.

n2s	:: num→short;	n2b	:: num→byte;
n2s(n)	= n *mod* 32768;	n2b(n)	= n *mod* 128;
s2p	:: short→(byte, byte);	p2s	:: (byte, byte)→short;
s2p(s)	= (s *div* 256, s *mod* 256);	p2s(hi, lo)	= 256∗hi + lo;
b2b	:: bool→byte;	==	:: num→num→num;
b2b True	= 1;	x == y	= 1, **if** x > y;
b2b False	= 0;		= 0, **if** x = y;
			= −1, **otherwise**;

4.5 Increment Instructions

The increment instructions load the value of a local, increment the value with a signed, 8-bit constant, and store the result. There is no scope for stack underflow or stack overflow, but it is possible for the data to under or overflow. This particular error condition is ignored by the JSP. The specification models this behaviour by using a conversion function n2s, which maps out of bounds values into the range of a short. The functions of Table 3 define explicit conversions between arbitrary integers and shorts (n2s), arbitrary integers and bytes (n2b), between shorts and pairs of bytes (s2p, p2s), and between booleans and bytes (b2b). These conversions are used consistently throughout the document, so that is would be easy to change the byte order of shorts. This approach makes it easier to implement the JSP on platforms with different views on number representations.

$$[\text{binc}]\ \frac{\vdash \mathsf{n2b}(\mathsf{os}(\mathsf{i}) + \mathsf{c}) \overset{b}{\Rightarrow} \mathsf{v}}{\vdash \langle \textbf{binc}\ \mathsf{i}\ \mathsf{c},\ \mathsf{sp},\ \mathsf{os}\rangle \overset{inc}{\Rightarrow} \langle \mathsf{os} \oplus \{\mathsf{i} \mapsto \mathsf{v}\}\rangle;}$$

$$[\text{sinc}]\ \frac{\begin{array}{l}\vdash \mathsf{s2p}(\mathsf{n2s}(\mathsf{p2s}(\mathsf{os}(\mathsf{i}),\ \mathsf{os}(\mathsf{i}+1)) + \mathsf{c})) \overset{p}{\Rightarrow} (\mathsf{hi},\ \mathsf{lo}),\\ \vdash \mathsf{os} \oplus \{\mathsf{i} \mapsto \mathsf{hi}\} \oplus \{\mathsf{i}+1 \mapsto \mathsf{lo}\} \overset{os}{\Rightarrow} \mathsf{os}'\end{array}}{\vdash \langle \textbf{sinc}\ \mathsf{i}\ \mathsf{c},\ \mathsf{sp},\ \mathsf{os}\rangle \overset{inc}{\Rightarrow} \langle \mathsf{os}'\rangle;}$$

4.6 Stack Instructions

The stack manipulation instructions are intended to rearrange information on the operand stack. The side conditions check for stack underflow and/or overflow.

- The **pop** and **pop2** instructions remove one and two bytes respectively from the stack. There are no separate pop instructions for shorts and references, to save opcodes.

$$[\text{pop}^1]\ \vdash \langle \textbf{pop},\ \mathsf{sp},\ \mathsf{os}\rangle \overset{stack}{\Rightarrow} \langle \mathsf{sp}-1,\ \mathsf{os}\rangle;$$

$$[\text{pop}^2]\ \vdash \langle \textbf{pop2},\ \mathsf{sp},\ \mathsf{os}\rangle \overset{stack}{\Rightarrow} \langle \mathsf{sp}-2,\ \mathsf{os}\rangle;$$

- The **dup** and **dup2** instructions duplicate one and two bytes respectively on top of the stack.

$$[\text{dup}]\ \frac{\vdash \mathsf{os}(\mathsf{sp}) \overset{b}{\Rightarrow} \mathsf{v}}{\begin{array}{l}\vdash \langle \textbf{dup},\ \mathsf{sp},\ \mathsf{os}\rangle \overset{stack}{\Rightarrow} \langle \mathsf{sp}+1,\ \mathsf{os} \oplus \{\mathsf{sp}+1 \mapsto \mathsf{v}\}\rangle,\\ \textbf{if}\ (\mathsf{sp} \ldots \mathsf{sp}+1) \subseteq\ \mathsf{stackPointer}_{\mathsf{range}};\end{array}}$$

$$[\text{dup2}]\ \frac{\begin{array}{l}\vdash (\mathsf{os}(\mathsf{sp}-1),\ \mathsf{os}(\mathsf{sp})) \overset{p}{\Rightarrow} (\mathsf{v}_2,\ \mathsf{v}_1),\\ \vdash \mathsf{os} \oplus \{\mathsf{sp}+1 \mapsto \mathsf{v}_2\} \oplus \{\mathsf{sp}+2 \mapsto \mathsf{v}_1\} \overset{os}{\Rightarrow} \mathsf{os}'\end{array}}{\begin{array}{l}\vdash \langle \textbf{dup2},\ \mathsf{sp},\ \mathsf{os}\rangle \overset{stack}{\Rightarrow} \langle \mathsf{sp}+2,\ \mathsf{os}'\rangle,\\ \textbf{if}\ (\mathsf{sp}-1 \ldots \mathsf{sp}+2) \subseteq\ \mathsf{stackPointer}_{\mathsf{range}};\end{array}}$$

– The **dup_x** instruction duplicates the top k elements of the operand stack n elements down the stack. The symbol $\uplus$ is the function overriding operator and the notation $\{x_i \mid i \leftarrow [a..b]\}$ generates a set of x_i where i ranges from a to b.

$$\frac{\begin{array}{l}\vdash \mathsf{kn}\ mod\ 16 \stackrel{b}{\Rightarrow} \mathsf{n},\\ \vdash \mathsf{kn}\ div\ 16 \stackrel{b}{\Rightarrow} \mathsf{k},\\ \vdash \mathsf{sp}' + \mathsf{k} \stackrel{s}{\Rightarrow} \mathsf{sp}',\\ \vdash \mathsf{os} \uplus \{\mathsf{sp}' - \mathsf{i} + 1 \mapsto \mathsf{os}(\mathsf{sp} - \mathsf{i} + 1) \mid \mathsf{i} \leftarrow [\mathsf{n}..1]\} \stackrel{os}{\Rightarrow} \mathsf{os}',\\ \vdash \mathsf{os}' \uplus \{\mathsf{sp}' - \mathsf{n} - \mathsf{i} + 1 \mapsto \mathsf{os}(\mathsf{sp}' - \mathsf{i} + 1) \mid \mathsf{i} \leftarrow [\mathsf{k}..1]\} \stackrel{os}{\Rightarrow} \mathsf{os}''\end{array}}{\begin{array}{l}\text{[dupx]}\ \vdash \langle \textbf{dup\_x}\ \mathsf{kn},\ \mathsf{sp},\ \mathsf{os}\rangle \stackrel{stack}{\Rightarrow} \langle \mathsf{sp} + \mathsf{k},\ \mathsf{os}''\rangle,\\ \quad \textbf{if}\ (\mathsf{sp} - \mathsf{n} \ldots \mathsf{sp} + \mathsf{k}) \subseteq \mathsf{stackPointer}_{\mathsf{range}} \wedge\\ \quad \mathsf{k} \in (1 \ldots 4) \wedge \mathsf{n} \in (0 \ldots 8) \wedge \mathsf{k} < \mathsf{n};\end{array}}$$

– The **swap** and **swap2** instructions swap the top two bytes and the top two pairs of bytes respectively on top of the operand stack.

$$\frac{\begin{array}{l}\vdash (\mathsf{os}(\mathsf{sp} - 1),\ \mathsf{os}(\mathsf{sp})) \stackrel{p}{\Rightarrow} (\mathsf{v}_2,\ \mathsf{v}_1),\\ \vdash \mathsf{os} \oplus \{\mathsf{sp} - 1 \mapsto \mathsf{v}_1\} \oplus \{\mathsf{sp} \mapsto \mathsf{v}_2\} \stackrel{os}{\Rightarrow} \mathsf{os}'\end{array}}{\begin{array}{l}\text{[swap]}\ \vdash \langle \textbf{swap},\ \mathsf{sp},\ \mathsf{os}\rangle \stackrel{stack}{\Rightarrow} \langle \mathsf{sp},\ \mathsf{os}'\rangle,\\ \quad \textbf{if}\ (\mathsf{sp} - 1 \ldots \mathsf{sp}) \subseteq \mathsf{stackPointer}_{\mathsf{range}};\end{array}}$$

$$\frac{\begin{array}{l}\vdash (\mathsf{os}(\mathsf{sp} - 3),\ \mathsf{os}(\mathsf{sp} - 2)) \stackrel{p}{\Rightarrow} (\mathsf{hi}_2,\ \mathsf{lo}_2),\\ \vdash (\mathsf{os}(\mathsf{sp} - 1),\ \mathsf{os}(\mathsf{sp})) \stackrel{p}{\Rightarrow} (\mathsf{hi}_1,\ \mathsf{lo}_1),\\ \vdash \mathsf{os} \oplus \{\mathsf{sp} - 3 \mapsto \mathsf{hi}_1\} \oplus \{\mathsf{sp} - 2 \mapsto \mathsf{lo}_1\} \stackrel{os}{\Rightarrow} \mathsf{os}',\\ \vdash \mathsf{os}' \oplus \{\mathsf{sp} - 1 \mapsto \mathsf{hi}_2\} \oplus \{\mathsf{sp} \mapsto \mathsf{lo}_2\} \stackrel{os}{\Rightarrow} \mathsf{os}''\end{array}}{\begin{array}{l}\text{[swap2]}\ \vdash \langle \textbf{swap2},\ \mathsf{sp},\ \mathsf{os}\rangle \stackrel{stack}{\Rightarrow} \langle \mathsf{sp},\ \mathsf{os}''\rangle,\\ \quad \textbf{if}\ (\mathsf{sp} - 3 \ldots \mathsf{sp}) \subseteq \mathsf{stackPointer}_{\mathsf{range}};\end{array}}$$

4.7 Creating Array Objects

Arrays are stored in the heap. Therefore, the transition relation $\stackrel{newarray}{\Rightarrow}$ specifies read/write access to the heap, as well as the operand stack. In addition, object creating instructions need to know which is the current application program id (pi). This information is used to classify objects according to who created them. The type of the relation reflects the fact that the program id is used but not changed. (The reader is reminded that Table 2 summarises the types of all transition relations.)

The array operation **newarray** expects the length of the array on the top of the operand stack. It accesses the length as al. **newarray** creates an appropriate array header ah and a mapping with a domain of $0 \ldots \mathsf{al} - 1$ to serve as the initial value of the array. The method table used is that of class java.lang.Object. The heap is extended with a new object which is to receive the created array header and contents. The reference to the new object is pushed onto the stack. The side condition ensures that stack underflow, heap overflow, or an invalid array length is detected.

$$
\frac{\begin{array}{l}
\vdash \mathsf{p2s}(\mathsf{os}(\mathsf{sp}-1),\ \mathsf{os}(\mathsf{sp})) \stackrel{s}{\Rightarrow} \mathsf{al},\\
\vdash \textbf{ArrayHeader}\ \mathsf{pi}\ 0\ \mathsf{java.lang.Object}_{\mathsf{mt}}\ \textbf{byte}\ \mathsf{al} \stackrel{ah}{\Rightarrow} \mathsf{ah},\\
\vdash \{\mathsf{i} \mapsto 0 \mid \mathsf{i} \leftarrow [0..\mathsf{al}-1]\} \stackrel{at}{\Rightarrow} \mathsf{at},\\
\vdash \mathsf{hp}+1 \stackrel{hp}{\Rightarrow} \mathsf{hp}',\\
\vdash \mathsf{s2p}(\mathsf{hp}') \stackrel{p}{\Rightarrow} (\mathsf{hi},\ \mathsf{lo}),\\
\vdash \mathsf{os} \oplus \{\mathsf{sp}-1 \mapsto \mathsf{hi}\} \oplus \{\mathsf{sp} \mapsto \mathsf{lo}\} \stackrel{os}{\Rightarrow} \mathsf{os}',\\
\vdash \mathsf{ha} \oplus \{\mathsf{hp}' \mapsto \textbf{ArrayObject}\ \mathsf{ah}\ \mathsf{at}\} \stackrel{ha}{\Rightarrow} \mathsf{ha}'
\end{array}}{\begin{array}{l}
[\mathsf{newarray}^1]\ \vdash \langle \textbf{newarray byte}, \mathsf{sp}, \mathsf{os}, \mathsf{hp}, \mathsf{ha}, \mathsf{pi}, \mathsf{pt}\rangle\\
\stackrel{newarray}{\Rightarrow} \langle \mathsf{os}', \mathsf{hp}', \mathsf{ha}'\rangle,\\
\textbf{if}\ (\mathsf{sp}-1 \ldots \mathsf{sp}) \subseteq \mathsf{stackPointer}_{\mathsf{range}} \wedge\\
\mathsf{al} \in \mathsf{arrayLength}_{\mathsf{range}} \wedge \mathsf{hp}' \in \mathsf{heapPointer}_{\mathsf{range}};
\end{array}}
$$

The two other versions of **newarray** are not shown here: the **bit** version of **newarray** is identical to the **byte** version above, because each bit is stored in a byte field. The **short** version uses two bytes for storing each short.

The **anewarray** instruction allocates an array of references to objects of the class associated with the given class id (ci). The application program id (pi) is used to access the class table of the current application program. This class table provides the method table for the array elements. The array is initialised to null references.

$$
\frac{\begin{array}{l}
\vdash \mathsf{p2s}(\mathsf{os}(\mathsf{sp}-1),\ \mathsf{os}(\mathsf{sp})) \stackrel{s}{\Rightarrow} \mathsf{al},\\
\vdash \mathsf{pt}(\mathsf{pi}) \stackrel{ct}{\Rightarrow} \mathsf{ct},\\
\vdash \mathsf{ct}(\mathsf{ci}) \stackrel{ob}{\Rightarrow} \textbf{ClassObject}\ \_\ \_\ \mathsf{mt}\ \_\ \_\ \_,\\
\vdash \textbf{ArrayHeader}\ \mathsf{pi}\ \mathsf{ci}\ \mathsf{mt}\ \textbf{ref}\ \mathsf{al} \stackrel{ah}{\Rightarrow} \mathsf{ah},\\
\vdash \{\mathsf{i} \mapsto 0 \mid \mathsf{i} \leftarrow [0..2{*}\mathsf{al}-1]\} \stackrel{at}{\Rightarrow} \mathsf{at},\\
\vdash \mathsf{hp}+1 \stackrel{hp}{\Rightarrow} \mathsf{hp}',\\
\vdash \mathsf{s2p}(\mathsf{hp}') \stackrel{p}{\Rightarrow} (\mathsf{hi},\ \mathsf{lo}),\\
\vdash \mathsf{os} \oplus \{\mathsf{sp}-1 \mapsto \mathsf{hi}\} \oplus \{\mathsf{sp} \mapsto \mathsf{lo}\} \stackrel{os}{\Rightarrow} \mathsf{os}',\\
\vdash \mathsf{ha} \oplus \{\mathsf{hp}' \mapsto \textbf{ArrayObject}\ \mathsf{ah}\ \mathsf{at}\} \stackrel{ha}{\Rightarrow} \mathsf{ha}'
\end{array}}{\begin{array}{l}
[\mathsf{anewarray}]\ \vdash \langle \textbf{anewarray}\ \mathsf{ci}, \mathsf{sp}, \mathsf{os}, \mathsf{hp}, \mathsf{ha}, \mathsf{pi}, \mathsf{pt}\rangle\\
\stackrel{newarray}{\Rightarrow} \langle \mathsf{os}', \mathsf{hp}', \mathsf{ha}'\rangle,\\
\textbf{if}\ (\mathsf{sp}-1 \ldots \mathsf{sp}) \subseteq \mathsf{stackPointer}_{\mathsf{range}} \wedge\\
\mathsf{al} \in \mathsf{arrayLength}_{\mathsf{range}} \wedge \mathsf{hp}' \in \mathsf{heapPointer}_{\mathsf{range}};
\end{array}}
$$

4.8 Loading Values from Arrays

The operation **arraylength** expects an array reference r on the stack and returns the length of the array. The side condition checks for stack underflow, and that a valid heap pointer to an array object is presented.

$$
\begin{array}{ll}
 & \vdash \mathsf{p2s}(\mathsf{os}(\mathsf{sp}-1),\ \mathsf{os}(\mathsf{sp})) \stackrel{s}{\Rightarrow} \mathsf{r}, \\
 & \vdash \mathsf{ha}(\mathsf{r}) \stackrel{ob}{\Rightarrow} \textbf{ArrayObject}(\textbf{ArrayHeader}\ \_\ \_\ \_\ \_\ \mathsf{al})\_, \\
 & \vdash \mathsf{s2p}(\mathsf{al}) \stackrel{p}{\Rightarrow} (\mathsf{hi},\ \mathsf{lo}), \\
 & \vdash \mathsf{os} \oplus \{\mathsf{sp}-1 \mapsto \mathsf{hi}\} \oplus \{\mathsf{sp} \mapsto \mathsf{lo}\} \stackrel{os}{\Rightarrow} \mathsf{os}' \\
\hline
[\mathsf{arraylength}] & \vdash \langle \textbf{arraylength},\ \mathsf{sp},\ \mathsf{os},\ \mathsf{ha} \rangle \stackrel{arrayload}{\Rightarrow} \langle \mathsf{sp},\ \mathsf{os}' \rangle, \\
 & \textbf{if}\ (\mathsf{sp}-1 \ldots \mathsf{sp}) \subseteq\ \mathsf{stackPointer}_{\mathsf{range}} \wedge \\
 & \quad \mathsf{r}\ \in\ \mathsf{heapPointer}_{\mathsf{range}} \wedge \mathsf{isArrayObject}(\mathsf{ha}(\mathsf{r}));
\end{array}
$$

Array load instructions access an array and deliver a value at the given index position. The side conditions check for stack underflow, a null reference, an improper object and illegal values of the array index.

$$
\begin{array}{ll}
 & \vdash \mathsf{p2s}(\mathsf{os}(\mathsf{sp}-3),\ \mathsf{os}(\mathsf{sp}-2)) \stackrel{s}{\Rightarrow} \mathsf{r}, \\
 & \vdash \mathsf{p2s}(\mathsf{os}(\mathsf{sp}-1),\ \mathsf{os}(\mathsf{sp})) \stackrel{s}{\Rightarrow} \mathsf{i}, \\
 & \vdash \mathsf{ha}(\mathsf{r}) \stackrel{ob}{\Rightarrow} \textbf{ArrayObject}\ \_\ \mathsf{at}, \\
 & \vdash \mathsf{at}(\mathsf{i}) \stackrel{b}{\Rightarrow} \mathsf{v} \\
\hline
[\mathsf{baload}] & \vdash \langle \textbf{baload},\ \mathsf{sp},\ \mathsf{os},\ \mathsf{ha} \rangle \stackrel{arrayload}{\Rightarrow} \langle \mathsf{sp}-3,\ \mathsf{os} \oplus \{\mathsf{sp}-3 \mapsto \mathsf{v}\} \rangle, \\
 & \textbf{if}\ (\mathsf{sp}-3 \ldots \mathsf{sp}) \subseteq\ \mathsf{stackPointer}_{\mathsf{range}} \wedge \\
 & \quad \mathsf{r}\ \in\ \mathsf{heapPointer}_{\mathsf{range}} \wedge \mathsf{isArrayObject}(\mathsf{ha}(\mathsf{r})) \wedge \mathsf{i}\ \in\ \mathsf{domain}(\mathsf{at});
\end{array}
$$

$$
\begin{array}{ll}
 & \vdash \mathsf{p2s}(\mathsf{os}(\mathsf{sp}-3),\ \mathsf{os}(\mathsf{sp}-2)) \stackrel{s}{\Rightarrow} \mathsf{r}, \\
 & \vdash \mathsf{p2s}(\mathsf{os}(\mathsf{sp}-1),\ \mathsf{os}(\mathsf{sp})) \stackrel{s}{\Rightarrow} \mathsf{i}, \\
 & \vdash \mathsf{ha}(\mathsf{r}) \stackrel{ob}{\Rightarrow} \textbf{ArrayObject}\ \_\ \mathsf{at}, \\
 & \vdash (\mathsf{at}(\mathsf{i}{*}2),\ \mathsf{at}(\mathsf{i}{*}2+1)) \stackrel{p}{\Rightarrow} (\mathsf{hi},\ \mathsf{lo}), \\
 & \vdash \mathsf{os} \oplus \{\mathsf{sp}-3 \mapsto \mathsf{hi}\} \oplus \{\mathsf{sp}-2 \mapsto \mathsf{lo}\} \stackrel{os}{\Rightarrow} \mathsf{os}' \\
\hline
[\mathsf{saload}] & \vdash \langle \textbf{saload},\ \mathsf{sp},\ \mathsf{os},\ \mathsf{ha} \rangle \stackrel{arrayload}{\Rightarrow} \langle \mathsf{sp}-2,\ \mathsf{os}' \rangle, \\
 & \textbf{if}\ (\mathsf{sp}-3 \ldots \mathsf{sp}) \subseteq\ \mathsf{stackPointer}_{\mathsf{range}} \wedge \\
 & \quad \mathsf{r}\ \in\ \mathsf{heapPointer}_{\mathsf{range}} \wedge \mathsf{isArrayObject}(\mathsf{ha}(\mathsf{r})) \wedge \\
 & (\mathsf{i}{*}2 \ldots \mathsf{i}{*}2+1) \subseteq\ \mathsf{domain}(\mathsf{at});
\end{array}
$$

The operation **aaload** is identical to **saload** and not shown here.

4.9 Storing Values into Arrays

The array store instructions need read/write access to the stack and read access to the heap. The side conditions check for stack underflow, null references, non-array objects, and illegal array indices. The **aastore** instruction is identical to **sastore**.

$$\frac{\begin{array}{l}\vdash \mathsf{p2s}(\mathsf{os}(\mathsf{sp}-4),\ \mathsf{os}(\mathsf{sp}-3)) \overset{s}{\Rightarrow} \mathsf{r},\\ \vdash \mathsf{p2s}(\mathsf{os}(\mathsf{sp}-2),\ \mathsf{os}(\mathsf{sp}-1)) \overset{s}{\Rightarrow} \mathsf{i},\\ \vdash \mathsf{os}(\mathsf{sp}) \overset{b}{\Rightarrow} \mathsf{v},\\ \vdash \mathsf{ha}(\mathsf{r}) \overset{ob}{\Rightarrow} \textbf{ArrayObject}\ \mathsf{ah}\ \mathsf{at},\\ \vdash \mathsf{at} \oplus \{\mathsf{i} \mapsto \mathsf{v}\} \overset{at}{\Rightarrow} \mathsf{at}',\\ \vdash \mathsf{ha} \oplus \{\mathsf{r} \mapsto \textbf{ArrayObject}\ \mathsf{ah}\ \mathsf{at}'\} \overset{ha}{\Rightarrow} \mathsf{ha}'\end{array}}{\begin{array}{ll}[\mathsf{bastore}] & \vdash \langle \textbf{bastore},\ \mathsf{sp},\ \mathsf{os},\ \mathsf{ha}\rangle \overset{arraystore}{\Rightarrow} \langle \mathsf{sp}-5,\ \mathsf{ha}'\rangle,\\ & \textbf{if}\ (\mathsf{sp}-4 \ldots \mathsf{sp}) \subseteq \mathsf{stackPointer}_{\mathsf{range}} \wedge\\ & \quad \mathsf{r} \in \mathsf{heapPointer}_{\mathsf{range}} \wedge \mathsf{isArrayObject}(\mathsf{ha}(\mathsf{r})) \wedge \mathsf{i} \in \mathsf{domain}(\mathsf{at});\end{array}}$$

$$\frac{\begin{array}{l}\vdash \mathsf{p2s}(\mathsf{os}(\mathsf{sp}-5),\ \mathsf{os}(\mathsf{sp}-4)) \overset{s}{\Rightarrow} \mathsf{r},\\ \vdash \mathsf{p2s}(\mathsf{os}(\mathsf{sp}-3),\ \mathsf{os}(\mathsf{sp}-2)) \overset{s}{\Rightarrow} \mathsf{i},\\ \vdash (\mathsf{os}(\mathsf{sp}-1),\ \mathsf{os}(\mathsf{sp})) \overset{p}{\Rightarrow} (\mathsf{hi},\ \mathsf{lo}),\\ \vdash \mathsf{ha}(\mathsf{r}) \overset{ob}{\Rightarrow} \textbf{ArrayObject}\ \mathsf{ah}\ \mathsf{at},\\ \vdash \mathsf{at} \oplus \{\mathsf{i}{*}2 \mapsto \mathsf{hi}\} \oplus \{\mathsf{i}{*}2+1 \mapsto \mathsf{lo}\} \overset{at}{\Rightarrow} \mathsf{at}',\\ \vdash \mathsf{ha} \oplus \{\mathsf{r} \mapsto \textbf{ArrayObject}\ \mathsf{ah}\ \mathsf{at}'\} \overset{ha}{\Rightarrow} \mathsf{ha}'\end{array}}{\begin{array}{ll}[\mathsf{sastore}] & \vdash \langle \textbf{sastore},\ \mathsf{sp},\ \mathsf{os},\ \mathsf{ha}\rangle \overset{arraystore}{\Rightarrow} \langle \mathsf{sp}-6,\ \mathsf{ha}'\rangle,\\ & \textbf{if}\ (\mathsf{sp}-5 \ldots \mathsf{sp}) \subseteq \mathsf{stackPointer}_{\mathsf{range}} \wedge\\ & \quad \mathsf{r} \in \mathsf{heapPointer}_{\mathsf{range}} \wedge \mathsf{isArrayObject}(\mathsf{ha}(\mathsf{r})) \wedge\\ & (\mathsf{i}{*}2 \ldots \mathsf{i}{*}2+1) \subseteq \mathsf{domain}(\mathsf{at});\end{array}}$$

4.10 Arithmetic

The unary (arithmetic) negation operator is defined below for bytes and shorts. It ignores under/overflow of values, but checks for stack underflow.

$$\frac{\vdash \mathsf{os}(\mathsf{sp}) \overset{b}{\Rightarrow} \mathsf{v}}{\begin{array}{ll}[\mathsf{bneg}] & \vdash \langle \textbf{bneg},\ \mathsf{sp},\ \mathsf{os}\rangle \overset{arith}{\Rightarrow} \langle \mathsf{sp},\ \mathsf{os} \oplus \{\mathsf{sp} \mapsto \mathsf{n2b}(-\mathsf{v})\}\rangle,\\ & \textbf{if}\ \mathsf{sp} \in \mathsf{stackPointer}_{\mathsf{range}};\end{array}}$$

$$\frac{\begin{array}{l}\vdash \mathsf{s2p}(\mathsf{n2s}(-(\mathsf{p2s}(\mathsf{os}(\mathsf{sp}-1),\ \mathsf{os}(\mathsf{sp}))))) \overset{p}{\Rightarrow} (\mathsf{hi},\ \mathsf{lo}),\\ \vdash \mathsf{os} \oplus \{\mathsf{sp}-1 \mapsto \mathsf{hi}\} \oplus \{\mathsf{sp} \mapsto \mathsf{lo}\} \overset{os}{\Rightarrow} \mathsf{os}'\end{array}}{\begin{array}{ll}[\mathsf{sneg}] & \vdash \langle \textbf{sneg},\ \mathsf{sp},\ \mathsf{os}\rangle \overset{arith}{\Rightarrow} \langle \mathsf{sp},\ \mathsf{os}'\rangle,\\ & \textbf{if}\ (\mathsf{sp}-1 \ldots \mathsf{sp}) \subseteq \mathsf{stackPointer}_{\mathsf{range}};\end{array}}$$

Binary addition for bytes and shorts is defined below. The other binary arithmetic instructions (for subtraction, multiplication, division and remainder) are defined in the same way, and are not shown. The side condition of the division and remainder operations check that the divisor is non-zero.

$$\frac{\vdash (\mathsf{os}(\mathsf{sp}-1),\ \mathsf{os}(\mathsf{sp})) \overset{p}{\Rightarrow} (\mathsf{v}_2,\ \mathsf{v}_1)}{\begin{array}{ll}[\mathsf{badd}] & \vdash \langle \textbf{badd},\ \mathsf{sp},\ \mathsf{os}\rangle \overset{arith}{\Rightarrow} \langle \mathsf{sp}-1,\ \mathsf{os} \oplus \{\mathsf{sp}-1 \mapsto \mathsf{n2b}(\mathsf{v}_2+\mathsf{v}_1)\}\rangle,\\ & \textbf{if}\ (\mathsf{sp}-1 \ldots \mathsf{sp}) \subseteq \mathsf{stackPointer}_{\mathsf{range}};\end{array}}$$

$$\frac{\begin{array}{l}\vdash \mathsf{p2s}(\mathsf{os}(\mathsf{sp}-3),\ \mathsf{os}(\mathsf{sp}-2)) \overset{s}{\Rightarrow} \mathsf{v}_2,\\ \vdash \mathsf{p2s}(\mathsf{os}(\mathsf{sp}-1),\ \mathsf{os}(\mathsf{sp})) \overset{s}{\Rightarrow} \mathsf{v}_1,\\ \vdash \mathsf{s2p}(\mathsf{n2s}(\mathsf{v}_2+\mathsf{v}_1)) \overset{p}{\Rightarrow} (\mathsf{hi},\ \mathsf{lo}),\\ \vdash \mathsf{os} \oplus \{\mathsf{sp}-3 \mapsto \mathsf{hi}\} \oplus \{\mathsf{sp}-2 \mapsto \mathsf{lo}\} \overset{os}{\Rightarrow} \mathsf{os}'\end{array}}{\begin{array}{l}[\mathsf{sadd}]\ \vdash \langle \textbf{sadd},\ \mathsf{sp},\ \mathsf{os}\rangle \overset{arith}{\Rightarrow} \langle \mathsf{sp}-2,\ \mathsf{os}'\rangle,\\ \quad \textbf{if}\ (\mathsf{sp}-3 \ldots \mathsf{sp}) \subseteq\ \mathsf{stackPointer}_{\mathsf{range}};\end{array}}$$

4.11 Logical Instructions

The logical shift left as defined below shifts the element next to the top of the stack. The shift count is the top of the stack. The remaining binary logical instructions (for arithmetic shift right with sign extension, unsigned shift right, bit-wise and, bit-wise or and bit-wise exclusive or) are defined in the same way and are not shown.

$$\frac{\vdash (\mathsf{os}(\mathsf{sp}-1),\ \mathsf{os}(\mathsf{sp})) \overset{p}{\Rightarrow} (\mathsf{v}_2,\ \mathsf{v}_1)}{\begin{array}{l}[\mathsf{bshl}]\ \vdash \langle \textbf{bshl},\ \mathsf{sp},\ \mathsf{os}\rangle \overset{logical}{\Rightarrow} \langle \mathsf{sp}-1,\ \mathsf{os} \oplus \{\mathsf{sp}-1 \mapsto \mathsf{n2b}(\mathsf{v}_2\ \ll\ \mathsf{v}_1)\}\rangle,\\ \quad \textbf{if}\ (\mathsf{sp}-1 \ldots \mathsf{sp}) \subseteq\ \mathsf{stackPointer}_{\mathsf{range}} \wedge \mathsf{v}_1\ \in (0 \ldots 7);\end{array}}$$

$$\frac{\begin{array}{l}\vdash \mathsf{p2s}(\mathsf{os}(\mathsf{sp}-3),\ \mathsf{os}(\mathsf{sp}-2)) \overset{s}{\Rightarrow} \mathsf{v}_2,\\ \vdash \mathsf{p2s}(\mathsf{os}(\mathsf{sp}-1),\ \mathsf{os}(\mathsf{sp})) \overset{s}{\Rightarrow} \mathsf{v}_1,\\ \vdash \mathsf{s2p}(\mathsf{n2s}(\mathsf{v}_2\ \ll\ \mathsf{v}_1)) \overset{p}{\Rightarrow} (\mathsf{hi},\ \mathsf{lo}),\\ \vdash \mathsf{os} \oplus \{\mathsf{sp}-3 \mapsto \mathsf{hi}\} \oplus \{\mathsf{sp}-2 \mapsto \mathsf{lo}\} \overset{os}{\Rightarrow} \mathsf{os}'\end{array}}{\begin{array}{l}[\mathsf{sshl}]\ \vdash \langle \textbf{sshl},\ \mathsf{sp},\ \mathsf{os}\rangle \overset{logical}{\Rightarrow} \langle \mathsf{sp}-2,\ \mathsf{os}'\rangle,\\ \quad \textbf{if}\ (\mathsf{sp}-3 \ldots \mathsf{sp}) \subseteq\ \mathsf{stackPointer}_{\mathsf{range}} \wedge \mathsf{v}_1\ \in (0 \ldots 15);\end{array}}$$

4.12 Conversions

The conversion operations explicitly truncate a short to a byte or zero fill a byte to a short. Stack underflow and overflow are detected.

$$\frac{\vdash \mathsf{n2b}(\mathsf{p2s}(\mathsf{os}(\mathsf{sp}-1),\ \mathsf{os}(\mathsf{sp}))) \overset{s}{\Rightarrow} \mathsf{v}}{\begin{array}{l}[\mathsf{s2b}]\ \vdash \langle \textbf{s2b},\ \mathsf{sp},\ \mathsf{os}\rangle \overset{conv}{\Rightarrow} \langle \mathsf{sp}-1,\ \mathsf{os} \oplus \{\mathsf{sp}-1 \mapsto \mathsf{v}\}\rangle,\\ \quad \textbf{if}\ (\mathsf{sp}-1 \ldots \mathsf{sp}) \subseteq\ \mathsf{stackPointer}_{\mathsf{range}};\end{array}}$$

$$\frac{\begin{array}{l}\vdash \mathsf{s2p}(\mathsf{os}(\mathsf{sp})) \overset{p}{\Rightarrow} (\mathsf{hi},\ \mathsf{lo}),\\ \vdash \mathsf{os} \oplus \{\mathsf{sp} \mapsto \mathsf{hi}\} \oplus \{\mathsf{sp}+1 \mapsto \mathsf{lo}\} \overset{os}{\Rightarrow} \mathsf{os}'\end{array}}{\begin{array}{l}[\mathsf{b2s}]\ \vdash \langle \textbf{b2s},\ \mathsf{sp},\ \mathsf{os}\rangle \overset{conv}{\Rightarrow} \langle \mathsf{sp}+1,\ \mathsf{os}'\rangle,\\ \quad \textbf{if}\ (\mathsf{sp} \ldots \mathsf{sp}+1) \subseteq\ \mathsf{stackPointer}_{\mathsf{range}};\end{array}}$$

4.13 Comparisons

The compare instruction **bcmp** returns -1 if the top element of the stack is greater than the one below it. It returns 0 if the top two elements are equal and

1 otherwise. The **scmp** instruction compares the shorts on top of the stack. The definition of the range comparison operator == is given in Table 3.

$$\frac{\vdash(\mathsf{os}(\mathsf{sp}-1),\ \mathsf{os}(\mathsf{sp})) \overset{p}{\Rightarrow} (\mathsf{v}_2,\ \mathsf{v}_1)}{[\mathsf{bcmp}]\ \vdash\langle \textbf{bcmp},\ \mathsf{sp},\ \mathsf{os}\rangle \overset{compare}{\Rightarrow} \langle \mathsf{sp}-1,\ \mathsf{os}\oplus\{\mathsf{sp}-1\mapsto(\mathsf{v}_2\ ==\ \mathsf{v}_1)\}\rangle,\ \textbf{if}\ (\mathsf{sp}-1\ldots\mathsf{sp})\subseteq\mathsf{stackPointer}_{\mathsf{range}};}$$

$$\frac{\begin{array}{l}\vdash\mathsf{p2s}(\mathsf{os}(\mathsf{sp}-3),\ \mathsf{os}(\mathsf{sp}-2)) \overset{s}{\Rightarrow} \mathsf{v}_2,\\ \vdash\mathsf{p2s}(\mathsf{os}(\mathsf{sp}-1),\ \mathsf{os}(\mathsf{sp})) \overset{s}{\Rightarrow} \mathsf{v}_1,\\ \vdash\mathsf{os}\oplus\{\mathsf{sp}-3\mapsto(\mathsf{v}_2\ ==\ \mathsf{v}_1)\} \overset{os}{\Rightarrow} \mathsf{os}'\end{array}}{[\mathsf{scmp}]\ \vdash\langle \textbf{scmp},\ \mathsf{sp},\ \mathsf{os}\rangle \overset{compare}{\Rightarrow} \langle \mathsf{sp}-3,\ \mathsf{os}'\rangle,\ \textbf{if}\ (\mathsf{sp}-3\ldots\mathsf{sp})\subseteq\mathsf{stackPointer}_{\mathsf{range}};}$$

The **acmp** instruction compares object references and returns 0 if the references are equal, 1 otherwise.

$$\frac{\begin{array}{l}\vdash\mathsf{p2s}(\mathsf{os}(\mathsf{sp}-3),\ \mathsf{os}(\mathsf{sp}-2)) \overset{s}{\Rightarrow} \mathsf{v}_2,\\ \vdash\mathsf{p2s}(\mathsf{os}(\mathsf{sp}-1),\ \mathsf{os}(\mathsf{sp})) \overset{s}{\Rightarrow} \mathsf{v}_1\end{array}}{[\mathsf{acmp}]\ \vdash\langle \textbf{acmp},\ \mathsf{sp},\ \mathsf{os}\rangle \overset{compare}{\Rightarrow} \langle \mathsf{sp}-3,\ \mathsf{os}\oplus\{\mathsf{sp}-3\mapsto(\mathsf{v}_2\ ==\ \mathsf{v}_1) mod\ 2\}\rangle,\ \textbf{if}\ (\mathsf{sp}-3\ldots\mathsf{sp})\subseteq\mathsf{stackPointer}_{\mathsf{range}};}$$

4.14 Transferring Control

The **ifeq** instruction adds its immediate operand to the value of the program counter (pc) if the top of the stack contains 0. Otherwise the program counter is incremented to point at the next instruction. Stack underflow is detected. The static semantics is assumed to detect illegal values for the program counter.

$$\frac{\begin{array}{l}\vdash\mathsf{os}(\mathsf{sp}) \overset{b}{\Rightarrow} \mathsf{v},\\ \vdash\mathsf{pc}+\mathsf{p2s\ offset} \overset{pc}{\Rightarrow} \mathsf{pc}'\end{array}}{[\mathsf{ifeq}^0]\ \vdash\langle \mathsf{pc},\ \textbf{ifeq}\ \mathsf{offset},\ \mathsf{sp},\ \mathsf{os}\rangle \overset{control}{\Rightarrow} \langle \mathsf{pc}',\ \mathsf{sp}-1,\ \mathsf{os}\rangle,\ \textbf{if}\ \mathsf{sp}\ \in\ \mathsf{stackPointer}_{\mathsf{range}}\wedge\mathsf{v}=0;}$$

$$\frac{\vdash\mathsf{os}(\mathsf{sp}) \overset{b}{\Rightarrow} \mathsf{v}}{[\mathsf{ifeq}^1]\ \vdash\langle \mathsf{pc},\ \textbf{ifeq}\ \mathsf{offset},\ \mathsf{sp},\ \mathsf{os}\rangle \overset{control}{\Rightarrow} \langle \mathsf{pc}+1,\ \mathsf{sp}-1,\ \mathsf{os}\rangle,\ \textbf{if}\ \mathsf{sp}\ \in\ \mathsf{stackPointer}_{\mathsf{range}}\wedge\mathsf{v}\neq0;}$$

The remaining operations **iflt**, **ifgt**, **ifne**, **ifge**, **ifle** are similar and not shown.

The static semantics is assumed to check that the unconditional jump instruction **goto** carries a valid offset.

$$\frac{\vdash\mathsf{pc}+\mathsf{p2s\ offset} \overset{s}{\Rightarrow} \mathsf{pc}'}{[\mathsf{goto}]\ \vdash\langle \mathsf{pc},\ \textbf{goto}\ \mathsf{offset},\ \mathsf{sp},\ \mathsf{os}\rangle \overset{control}{\Rightarrow} \langle \mathsf{pc}',\ \mathsf{sp},\ \mathsf{os}\rangle;}$$

4.15 Support for Switch Statements

The **tableswitch** and **lookupswitch** instructions provide support for the Java switch statements. The **tableswitch** instruction allows for a selection of jump

targets from an indexed table, with the choice index coming from the stack. The **lookupswitch** instruction is similar, except that a keyed table is used rather than an indexed one.

Both instructions have a number of immediate operands, the first of which is the default offset. The **tableswitch** instruction has further immediate operands to specify the lower and upperbounds of a jump table and the jump table itself. The instruction expects a byte index on the stack, which is used to select the appropriate offset from the jump table. The offset is then added to the current value of the program counter. If the index lies outside the range defined by the lower and upperbound, the default offset is added to the program counter.

The side condition checks that the stack pointer is valid, but does not need to check that the old or new values of the program counter are valid. This is the task of the static semantics.

$$[\mathsf{tableswitch}^1]\ \frac{\begin{array}{l}\vdash \mathsf{os}(\mathsf{sp}) \stackrel{b}{\Rightarrow} \mathsf{index},\\ \vdash \mathsf{cases}(\mathsf{index}) \stackrel{p}{\Rightarrow} \mathsf{offset},\\ \vdash \mathsf{pc} + \mathsf{p2s}(\mathsf{offset}) \stackrel{s}{\Rightarrow} \mathsf{pc}'\end{array}}{\begin{array}{l}\vdash \langle \mathsf{pc},\ \textbf{tableswitch}\ \mathsf{default\ low\ high\ cases},\ \mathsf{sp},\ \mathsf{os}\rangle\\ \stackrel{switch}{\Rightarrow} \langle \mathsf{pc}',\ \mathsf{sp}-1,\ \mathsf{os}\rangle,\\ \textbf{if}\ \mathsf{sp}\ \in\ \mathsf{stackPointer}_{\mathsf{range}} \wedge \mathsf{index}\ \in (\mathsf{low}\ldots\mathsf{high});\end{array}}$$

$$[\mathsf{tableswitch}^2]\ \frac{\begin{array}{l}\vdash \mathsf{os}(\mathsf{sp}) \stackrel{b}{\Rightarrow} \mathsf{index},\\ \vdash \mathsf{pc} + \mathsf{p2s}(\mathsf{default}) \stackrel{s}{\Rightarrow} \mathsf{pc}'\end{array}}{\begin{array}{l}\vdash \langle \mathsf{pc},\ \textbf{tableswitch}\ \mathsf{default\ low\ high\ cases},\ \mathsf{sp},\ \mathsf{os}\rangle\\ \stackrel{switch}{\Rightarrow} \langle \mathsf{pc}',\ \mathsf{sp}-1,\ \mathsf{os}\rangle,\\ \textbf{if}\ \mathsf{sp}\ \in\ \mathsf{stackPointer}_{\mathsf{range}} \wedge \mathsf{index}\ \notin (\mathsf{low}\ldots\mathsf{high});\end{array}}$$

The **lookupswitch** has a default offset and further immediate operands to specify the number of entries in the jump table and the jump table itself. The **lookupswitch** instruction expects a key on the stack, which when it occurs in the table is used to select the appropriate offset from the jump table. The offset is then added to the current value of the program counter. If the key does not occur in the jump table, the default offset is added to the program counter.

$$[\mathsf{lookupswitch}^1]\ \frac{\begin{array}{l}\vdash \mathsf{os}(\mathsf{sp}) \stackrel{b}{\Rightarrow} \mathsf{key},\\ \vdash \{\mathsf{o} \mid (\mathsf{k},\ \mathsf{o}) \leftarrow \mathsf{range}(\mathsf{cases}) \wedge\ \mathsf{key} = \mathsf{k}\} \stackrel{ps}{\Rightarrow} \mathsf{offsets},\\ \vdash \mathsf{pc} + \mathsf{p2s}(\mathsf{hd}(\mathsf{offsets})) \stackrel{s}{\Rightarrow} \mathsf{pc}'\end{array}}{\begin{array}{l}\vdash \langle \mathsf{pc},\ \textbf{lookupswitch}\ \mathsf{default\ entries\ cases},\ \mathsf{sp},\ \mathsf{os}\rangle\\ \stackrel{switch}{\Rightarrow} \langle \mathsf{pc}',\ \mathsf{sp}-1,\ \mathsf{os}\rangle,\\ \textbf{if}\ \mathsf{sp}\ \in\ \mathsf{stackPointer}_{\mathsf{range}} \wedge \mathsf{offsets} \neq \{\};\end{array}}$$

$$[\mathsf{lookupswitch}^2]\ \frac{\begin{array}{l}\vdash \mathsf{os}(\mathsf{sp}) \stackrel{b}{\Rightarrow} \mathsf{key},\\ \vdash \{\mathsf{o} \mid (\mathsf{k},\ \mathsf{o}) \leftarrow \mathsf{range}(\mathsf{cases}) \wedge\ \mathsf{key} = \mathsf{k}\} \stackrel{ps}{\Rightarrow} \mathsf{offsets},\\ \vdash \mathsf{pc} + \mathsf{p2s}(\mathsf{default}) \stackrel{s}{\Rightarrow} \mathsf{pc}'\end{array}}{\begin{array}{l}\vdash \langle \mathsf{pc},\ \textbf{lookupswitch}\ \mathsf{default}\ \mathsf{entries}\ \mathsf{cases},\ \mathsf{sp},\ \mathsf{os}\rangle\\ \stackrel{switch}{\Rightarrow} \langle \mathsf{pc}',\ \mathsf{sp}-1,\ \mathsf{os}\rangle,\\ \textbf{if}\ \mathsf{sp}\ \in\ \mathsf{stackPointer}_{\mathsf{range}} \wedge \mathsf{offsets} = \{\};\end{array}}$$

4.16 Exception Handling

The **athrow** instruction terminates the execution of the JSP program, for there is no pc, sp and os for which the relation below holds. The present treatment of exceptions is somewhat crude, but consistent with ISO 7816-4 requirements.

$$[\mathsf{athrow}]\ \vdash \langle \mathsf{pc},\ \textbf{athrow},\ \mathsf{sp},\ \mathsf{os}\rangle \stackrel{exception}{\Rightarrow} \langle \mathsf{pc},\ \mathsf{sp},\ \mathsf{os}\rangle,\ \textbf{if}\ \mathsf{False};$$

The **jsr** and **ret** instructions are used by the JVM to support exception handling. Even though the JSP provides only rudimentary support for exceptions, the semantics of these two instructions is well defined. Stack overflow and illegal return addresses are detected.

$$[\mathsf{ret}]\ \frac{\vdash \mathsf{p2s}(\mathsf{os}(\mathsf{i}),\ \mathsf{os}(\mathsf{i}+1)) \stackrel{s}{\Rightarrow} \mathsf{pc}'}{\vdash \langle \mathsf{pc},\ \textbf{ret}\ \mathsf{i},\ \mathsf{sp},\ \mathsf{os}\rangle \stackrel{exception}{\Rightarrow} \langle \mathsf{pc}',\ \mathsf{sp},\ \mathsf{os}\rangle,\ \textbf{if}\ \mathsf{pc}'\ \in\ \mathsf{programCounter}_{\mathsf{range}};}$$

$$[\mathsf{jsr}]\ \frac{\begin{array}{l}\vdash \mathsf{pc} + \mathsf{p2s}(\mathsf{hi}_{\mathsf{v}},\ \mathsf{lo}_{\mathsf{v}}) \stackrel{s}{\Rightarrow} \mathsf{pc}',\\ \vdash \mathsf{s2p}(\mathsf{pc}) \stackrel{p}{\Rightarrow} (\mathsf{hi}_{\mathsf{p}},\ \mathsf{lo}_{\mathsf{p}}),\\ \vdash \mathsf{os} \oplus \{\mathsf{sp}+1 \mapsto \mathsf{hi}_{\mathsf{p}}\} \oplus \{\mathsf{sp}+2 \mapsto \mathsf{lo}_{\mathsf{p}}\} \stackrel{os}{\Rightarrow} \mathsf{os}'\end{array}}{\begin{array}{l}\vdash \langle \mathsf{pc},\ \textbf{jsr}(\mathsf{hi}_{\mathsf{v}},\ \mathsf{lo}_{\mathsf{v}}),\ \mathsf{sp},\ \mathsf{os}\rangle \stackrel{exception}{\Rightarrow} \langle \mathsf{pc}',\ \mathsf{sp}+2,\ \mathsf{os}'\rangle,\\ \textbf{if}\ (\mathsf{sp}+1 \ldots \mathsf{sp}+2) \subseteq\ \mathsf{stackPointer}_{\mathsf{range}} \wedge\\ \mathsf{pc}'\ \in\ \mathsf{programCounter}_{\mathsf{range}};\end{array}}$$

4.17 Method Invocation

The JSP has three different instructions to invoke methods. The **invokevirtual** is the normal dynamic method dispatch instruction. The **invoke** instruction is used when the Java compiler or JSP to JVM byte code translator are able to determine statically which method to invoke. The **invokeinterface** instruction supports Java's approach to multiple inheritance by searching for a method that implements an abstract method from an interface.

The **invokeinterface** instruction has three operands. The first, params, specifies the number of arguments to be expected on the operand stack. The second immediate operand, ii, indicates the index of an interface. The third mi determines which (abstract) method within the interface is required.

$$
\frac{\begin{array}{l}
\vdash \mathsf{pt(pi)} \stackrel{ct}{\Rightarrow} \mathsf{ct},\\
\vdash \mathsf{p2s(os(sp-params+1),\ os(sp-params+2))} \stackrel{s}{\Rightarrow} \mathsf{r},\\
\vdash \mathsf{ha(r)} \stackrel{ob}{\Rightarrow} \textbf{RegularObject}\ \mathsf{oh}\ \_,\\
\vdash \mathsf{oh} \stackrel{oh}{\Rightarrow} \textbf{ObjectHeader}\ \_\ \_(\textbf{MethodTable}\ \mathsf{ci\ et}),\\
\vdash \mathsf{ct(ci)} \stackrel{ob}{\Rightarrow} \textbf{ClassObject}\ \_\ \_\ \_\ \_\ \_\ \mathsf{cit},\\
\vdash \mathsf{cit(ii)} \stackrel{it}{\Rightarrow} \mathsf{it},\\
\vdash \mathsf{it(mi)} \stackrel{s}{\Rightarrow} \mathsf{mi'},\\
\vdash \mathsf{et(mi')} \stackrel{s}{\Rightarrow} \mathsf{pc'},\\
\vdash \{\mathsf{params-i} \mapsto \mathsf{os(sp-i+1)} \mid \mathsf{i} \leftarrow [1..\mathsf{params}]\} \stackrel{os}{\Rightarrow} \mathsf{os'},\\
\vdash \mathsf{fa} \oplus \{\mathsf{fp}+1 \mapsto \textbf{Frame}(\mathsf{pc}+1)\mathsf{fp(sp-params)os}\} \stackrel{fa}{\Rightarrow} \mathsf{fa'}
\end{array}}{\begin{array}{l}
[\mathsf{invoke}^1]\ \vdash \langle \mathsf{pc,\ ca,\ } \textbf{invokeinterface}\ \mathsf{params\ ii\ mi,\ sp,\ os,\ fp,\ fa,\ ha,\ pi,\ pt} \rangle\\
\qquad \stackrel{invokeinterface}{\Rightarrow} \langle \mathsf{pc',\ params-1,\ os',\ fp+1,\ fa'} \rangle,\\
\qquad \textbf{if}\ (\mathsf{sp-params+1} \ldots \mathsf{sp}) \subseteq \mathsf{stackPointer_{range}} \wedge\\
\qquad\ \mathsf{r} \in \mathsf{heapPointer_{range}} \wedge \mathsf{isRegularObject(ha(r))} \wedge\\
\qquad\ \mathsf{ci} \in \mathsf{classId_{range}} \wedge \mathsf{ii} \in \mathsf{interfaceId_{range}} \wedge\\
\qquad\ \mathsf{mi} \in \mathsf{methodId_{range}} \wedge \mathsf{mi'} \in \mathsf{methodId_{range}} \wedge\\
\qquad\ \mathsf{fp}+1 \in \mathsf{framePointer_{range}};
\end{array}}
$$

The top of the operand stack must contain a reference to an object, which should be an instance of a regular class that implements the interface method. The header of the object is accessed to yield the interface table (cit) associated with the object. The table it maps the method index of the abstract method (mi) onto the method index of the implementation (mi′). The latter is then used to locate the appropriate program counter in the method table of the object pointed at by r. The value of the program counter will be made to point at the first proper instruction of the method. A new frame is created, linking to the previous frame for the benefit of the return instruction. Execution continues at the first instruction of the callee.

The **invokevirtual** instruction expects a reference to an object on top of the operand stack. The object header of the object is accessed to yield the method table associated with the object. The method index mi determines which method is to be activated. The value of the program counter pc will be made to point at the first proper instruction of the method. A new frame is created, linking to the previous frame for the benefit of the return instruction. Execution continues at the first instruction of the callee.

$$
\begin{array}{ll}
 & \vdash \mathsf{p2s}(\mathsf{os}(\mathsf{sp}-\mathsf{params}+1),\ \mathsf{os}(\mathsf{sp}-\mathsf{params}+2)) \stackrel{s}{\Rightarrow} \mathsf{r}, \\
 & \vdash \mathsf{ha}(\mathsf{r}) \stackrel{ob}{\Rightarrow} \textbf{RegularObject}\ \mathsf{oh}\ \_, \\
 & \vdash \mathsf{oh} \stackrel{ob}{\Rightarrow} \textbf{ObjectHeader}\ \_\ \_(\textbf{MethodTable}\ \_\ \mathsf{et}), \\
 & \vdash \mathsf{et}(\mathsf{mi}) \stackrel{s}{\Rightarrow} \mathsf{pc}', \\
 & \vdash \{\mathsf{params}-\mathsf{i} \mapsto \mathsf{os}(\mathsf{sp}-\mathsf{i}+1) \mid \mathsf{i} \leftarrow [1..\mathsf{params}]\} \stackrel{os}{\Rightarrow} \mathsf{os}', \\
 & \vdash \mathsf{fa} \oplus \{\mathsf{fp}+1 \mapsto \textbf{Frame}(\mathsf{pc}+1)\mathsf{fp}(\mathsf{sp}-\mathsf{params})\mathsf{os}\} \stackrel{fa}{\Rightarrow} \mathsf{fa}' \\
\hline
[\mathsf{invoke}^2] & \vdash \langle \mathsf{pc},\ \mathsf{ca},\ \textbf{invokevirtual}\ \mathsf{params}\ \mathsf{mi},\ \mathsf{sp},\ \mathsf{os},\ \mathsf{fp},\ \mathsf{fa},\ \mathsf{ha} \rangle \\
 & \stackrel{invokevirtual}{\Rightarrow} \langle \mathsf{pc}',\ \mathsf{params}-1,\ \mathsf{os}',\ \mathsf{fp}+1,\ \mathsf{fa}' \rangle, \\
 & \textbf{if}\ (\mathsf{sp}-\mathsf{params}+1 \ldots \mathsf{sp}) \subseteq\ \mathsf{stackPointer}_{\mathsf{range}} \wedge \\
 & \mathsf{r}\ \in\ \mathsf{heapPointer}_{\mathsf{range}} \wedge \mathsf{isRegularObject}(\mathsf{ha}(\mathsf{r})) \wedge \\
 & \mathsf{mi}\ \in\ \mathsf{methodId}_{\mathsf{range}} \wedge \mathsf{fp}+1\ \in\ \mathsf{framePointer}_{\mathsf{range}};
\end{array}
$$

The immediate operands of the **invoke** instruction specify the two bytes that determine the index of the method in the codeArea. The number of parameters is retrieved from the method header (which is stored in the pseudo instruction preceding the first proper instruction of the method).

$$
\begin{array}{ll}
 & \vdash \mathsf{p2s}\ \mathsf{offset} \stackrel{s}{\Rightarrow} \mathsf{pc}', \\
 & \vdash \mathsf{ca}(\mathsf{pc}'-1) \stackrel{bc}{\Rightarrow} (\textbf{MethodHeader}\ \_\ \_\ \_\ \mathsf{params}\ \mathsf{locals}), \\
 & \vdash \{\mathsf{params}-\mathsf{i} \mapsto \mathsf{os}(\mathsf{sp}+1-\mathsf{i}) \mid \mathsf{i} \leftarrow [1..\mathsf{params}]\} \stackrel{os}{\Rightarrow} \mathsf{os}', \\
 & \vdash \mathsf{fa} \oplus \{\mathsf{fp}+1 \mapsto \textbf{Frame}(\mathsf{pc}+1)\mathsf{fp}(\mathsf{sp}-\mathsf{params})\mathsf{os}\} \stackrel{fa}{\Rightarrow} \mathsf{fa}' \\
\hline
[\mathsf{invoke}^3] & \vdash \langle \mathsf{pc},\ \mathsf{ca},\ \textbf{invoke}\ \mathsf{offset},\ \mathsf{sp},\ \mathsf{os},\ \mathsf{fp},\ \mathsf{fa} \rangle \\
 & \stackrel{invoke}{\Rightarrow} \langle \mathsf{pc}',\ \mathsf{locals}+\mathsf{params}-1,\ \mathsf{os}',\ \mathsf{fp}+1,\ \mathsf{fa}' \rangle, \\
 & \textbf{if}\ (\mathsf{sp}-\mathsf{params}+1 \ldots \mathsf{sp}) \subseteq\ \mathsf{stackPointer}_{\mathsf{range}} \wedge \\
 & \mathsf{fp}+1\ \in\ \mathsf{framePointer}_{\mathsf{range}};
\end{array}
$$

4.18 Method Return

The return instructions below return from a (non-static) method. The four instructions differ only in the return value produced. Each return instruction abandons the frame pointed at by the frame pointer and returns to the previous frame pointer. The appropriate return value is deposited onto the operand stack of the caller (except in the last case below, which is intended for a void returning method). The side conditions check for stack under/overflow and frame underflow.

$$
\begin{array}{ll}
 & \vdash \mathsf{fa}(\mathsf{fp}) \stackrel{f}{\Rightarrow} \textbf{Frame}\ \mathsf{pc}'\ \mathsf{fp}'\ \mathsf{sp}'\ \mathsf{os}', \\
 & \vdash \mathsf{os}(\mathsf{sp}) \stackrel{b}{\Rightarrow} \mathsf{v}, \\
 & \vdash \mathsf{os}' \oplus \{\mathsf{sp}'+1 \mapsto \mathsf{v}\} \stackrel{os}{\Rightarrow} \mathsf{os}'' \\
\hline
[\mathsf{breturn}] & \vdash \langle \textbf{breturn},\ \mathsf{sp},\ \mathsf{os},\ \mathsf{fp},\ \mathsf{fa} \rangle \stackrel{return}{\Rightarrow} \langle \mathsf{pc}',\ \mathsf{sp}'+1,\ \mathsf{os}'',\ \mathsf{fp}' \rangle, \\
 & \textbf{if}\ \mathsf{fp}\ \in\ \mathsf{framePointer}_{\mathsf{range}} \wedge \mathsf{sp}\ \in\ \mathsf{stackPointer}_{\mathsf{range}} \wedge \\
 & (\mathsf{sp}'+1) \in\ \mathsf{stackPointer}_{\mathsf{range}};
\end{array}
$$

$$
\begin{array}{ll}
 & \vdash \mathsf{fa}(\mathsf{fp}) \stackrel{f}{\Rightarrow} \textbf{Frame}\ \mathsf{pc}'\ \mathsf{fp}'\ \mathsf{sp}'\ \mathsf{os}', \\
 & \vdash (\mathsf{os}(\mathsf{sp}-1),\ \mathsf{os}(\mathsf{sp})) \stackrel{p}{\Rightarrow} (\mathsf{hi},\ \mathsf{lo}), \\
 & \vdash \mathsf{os}' \oplus \{\mathsf{sp}'+1 \mapsto \mathsf{hi}\} \oplus \{\mathsf{sp}'+2 \mapsto \mathsf{lo}\} \stackrel{os}{\Rightarrow} \mathsf{os}'' \\
\hline
[\mathsf{sreturn}] & \vdash \langle \textbf{sreturn},\ \mathsf{sp},\ \mathsf{os},\ \mathsf{fp},\ \mathsf{fa}\rangle \stackrel{return}{\Rightarrow} \langle \mathsf{pc}',\ \mathsf{sp}'+2,\ \mathsf{os}'',\ \mathsf{fp}'\rangle, \\
 & \textbf{if}\ \mathsf{fp} \in \mathsf{framePointer}_{\mathsf{range}} \wedge \\
 & (\mathsf{sp}-1 \ldots \mathsf{sp}) \subseteq \mathsf{stackPointer}_{\mathsf{range}} \wedge \\
 & (\mathsf{sp}'+1 \ldots \mathsf{sp}'+2) \subseteq \mathsf{stackPointer}_{\mathsf{range}};
\end{array}
$$

$$
\begin{array}{ll}
 & \vdash \mathsf{fa}(\mathsf{fp}) \stackrel{f}{\Rightarrow} \textbf{Frame}\ \mathsf{pc}'\ \mathsf{fp}'\ \mathsf{sp}'\ \mathsf{os}' \\
\hline
[\mathsf{return}] & \vdash \langle \textbf{return},\ \mathsf{sp},\ \mathsf{os},\ \mathsf{fp},\ \mathsf{fa}\rangle \stackrel{return}{\Rightarrow} \langle \mathsf{pc}',\ \mathsf{sp}',\ \mathsf{os}',\ \mathsf{fp}'\rangle, \\
 & \textbf{if}\ \mathsf{fp} \in \mathsf{framePointer}_{\mathsf{range}};
\end{array}
$$

The instruction **areturn** is identical to **sreturn** and thus not shown here.

4.19 Object Operations

The new operation creates an instance of the class identified by the given class index ci. The class index is used to lookup the class in the class table pertaining to the current application program, which itself is found by using the current application program id pi as an index in the application program table. The fields are initialised to zeroes.

$$
\begin{array}{ll}
 & \vdash \mathsf{pt}(\mathsf{pi}) \stackrel{ct}{\Rightarrow} \mathsf{ct}, \\
 & \vdash \mathsf{ct}(\mathsf{ci}) \stackrel{ob}{\Rightarrow} \textbf{ClassObject}\ \mathsf{oh}\ \mathsf{is}\ \_\ \_\ \_\ \_, \\
 & \vdash \{\mathsf{i} \mapsto 0 \mid \mathsf{i} \leftarrow [0..\mathsf{is}-1]\} \stackrel{ft}{\Rightarrow} \mathsf{ft}, \\
 & \vdash \mathsf{hp}+1 \stackrel{hp}{\Rightarrow} \mathsf{hp}', \\
 & \vdash \mathsf{s2p}(\mathsf{hp}') \stackrel{p}{\Rightarrow} (\mathsf{hi}_{\mathsf{r}},\ \mathsf{lo}_{\mathsf{r}}), \\
 & \vdash \mathsf{os} \oplus \{\mathsf{sp}+1 \mapsto \mathsf{hi}_{\mathsf{r}}\} \oplus \{\mathsf{sp}+2 \mapsto \mathsf{lo}_{\mathsf{r}}\} \stackrel{os}{\Rightarrow} \mathsf{os}', \\
 & \vdash \mathsf{ha} \oplus \{\mathsf{hp}' \mapsto \textbf{RegularObject}\ \mathsf{oh}\ \mathsf{ft}\} \stackrel{ha}{\Rightarrow} \mathsf{ha}' \\
\hline
[\mathsf{new}] & \vdash \langle \textbf{new}\ \mathsf{ci},\ \mathsf{sp},\ \mathsf{os},\ \mathsf{hp},\ \mathsf{ha},\ \mathsf{pi},\ \mathsf{pt}\rangle \stackrel{object}{\Rightarrow} \langle \mathsf{sp}+2,\ \mathsf{os}',\ \mathsf{hp}',\ \mathsf{ha}'\rangle, \\
 & \textbf{if}\ (\mathsf{sp}+1 \ldots \mathsf{sp}+2) \subseteq \mathsf{stackPointer}_{\mathsf{range}} \wedge \\
 & \mathsf{pi} \in \mathsf{progId}_{\mathsf{range}} \wedge \mathsf{ci} \in \mathsf{classId}_{\mathsf{range}} \wedge \\
 & \mathsf{hp}' \in \mathsf{heapPointer}_{\mathsf{range}};
\end{array}
$$

There are three instructions to determine whether an object is an instance of a particular class. The **instanceof** instruction is for regular objects. The two other instructions **ainstanceof** and **aainstanceof** handle array objects of primitive and non-primitive types respectively.

The immediate operand ci_{t} of the instruction **instanceof** must be the index into the class table of some regular class, t say. In addition, the top of the stack must contain a reference r to a regular object of some class, s say. If t and s are the same, or if t is a super class of s, the instruction pushes 1 on the operand stack; 0 otherwise. (See Table 3 for the definition of b2b).

$$\frac{\begin{array}{l}\vdash \mathsf{p2s}(\mathsf{os}(\mathsf{sp}-1),\ \mathsf{os}(\mathsf{sp})) \overset{s}{\Rightarrow} \mathsf{r},\\ \vdash \mathsf{ha}(\mathsf{r}) \overset{ob}{\Rightarrow} \textbf{RegularObject}(\textbf{ObjectHeader}\ \_\ \_(\textbf{MethodTable}\ \mathsf{ci_s}\ \_))\_,\\ \vdash \mathsf{pt}(\mathsf{pi}) \overset{ct}{\Rightarrow} \mathsf{ct},\\ \vdash \mathsf{ct}(\mathsf{ci_s}) \overset{ob}{\Rightarrow} \textbf{ClassObject}\ \_\ \_\ \_\ \_\ \mathsf{super}\ \_,\\ \vdash \mathsf{b2b}(\mathsf{ci_t} = \mathsf{ci_s} \vee \mathsf{ci_t} \in \mathsf{range}(\mathsf{super})) \overset{b}{\Rightarrow} \mathsf{v},\\ \vdash \mathsf{os} \oplus \{\mathsf{sp}-1 \mapsto \mathsf{v}\} \overset{os}{\Rightarrow} \mathsf{os}'\end{array}}{\begin{array}{l}[\mathsf{instanceof}]\ \vdash \langle \textbf{instanceof}\ \mathsf{ci_t},\ \mathsf{sp},\ \mathsf{os},\ \mathsf{hp},\ \mathsf{ha},\ \mathsf{pi},\ \mathsf{pt}\rangle \overset{instance}{\Rightarrow} \langle \mathsf{sp}-1,\ \mathsf{os}'\rangle,\\ \textbf{if}\ (\mathsf{sp}-1 \ldots \mathsf{sp}) \subseteq\ \mathsf{stackPointer_{range}} \wedge\\ \quad \mathsf{r}\ \in\ \mathsf{heapPointer_{range}} \wedge \mathsf{isRegularObject}(\mathsf{ha}(\mathsf{r})) \wedge\\ \quad \mathsf{pi}\ \in\ \mathsf{progId_{range}} \wedge \mathsf{ci_s}\ \in\ \mathsf{classId_{range}};\end{array}}$$

The immediate operand $\mathsf{dt_t}$ of the instruction **ainstanceof** must specify one of the three primitive array types, t say. The top of the stack r must point at an array of primitive types, s say. If t and s are the same, 1 is pushed on the operand stack; 0 otherwise.

$$\frac{\begin{array}{l}\vdash \mathsf{p2s}(\mathsf{os}(\mathsf{sp}-1),\ \mathsf{os}(\mathsf{sp})) \overset{s}{\Rightarrow} \mathsf{r},\\ \vdash \mathsf{ha}(\mathsf{r}) \overset{ob}{\Rightarrow} \textbf{ArrayObject}(\textbf{ArrayHeader}\ \_\ \_\ \_\ \mathsf{dt_s}\ \_)\_,\\ \vdash \mathsf{b2b}(\mathsf{dt_s} \in \{\textbf{bit},\ \textbf{byte},\ \textbf{short}\} \wedge \mathsf{dt_t} = \mathsf{dt_s}) \overset{b}{\Rightarrow} \mathsf{v},\\ \vdash \mathsf{os} \oplus \{\mathsf{sp}-1 \mapsto \mathsf{v}\} \overset{os}{\Rightarrow} \mathsf{os}'\end{array}}{\begin{array}{l}[\mathsf{ainstanceof}]\ \vdash \langle \textbf{ainstanceof}\ \mathsf{dt_t},\ \mathsf{sp},\ \mathsf{os},\ \mathsf{hp},\ \mathsf{ha},\ \mathsf{pi},\ \mathsf{pt}\rangle \overset{instance}{\Rightarrow} \langle \mathsf{sp}-1,\ \mathsf{os}'\rangle,\\ \textbf{if}\ (\mathsf{sp}-1 \ldots \mathsf{sp}) \subseteq\ \mathsf{stackPointer_{range}} \wedge\\ \quad \mathsf{r}\ \in\ \mathsf{heapPointer_{range}} \wedge \mathsf{isArrayObject}(\mathsf{ha}(\mathsf{r}));\end{array}}$$

The immediate operand $\mathsf{ci_t}$ of the instruction **ainstanceof** must be the index into the class table of some regular class, t say. The top of the stack must contain a reference r to an array object, whose elements are instances of some class, s say. If t and s are the same, or if t is a super class of s, the instruction pushes 1 on the operand stack; 0 otherwise.

$$\frac{\begin{array}{l}\vdash \mathsf{p2s}(\mathsf{os}(\mathsf{sp}-1),\ \mathsf{os}(\mathsf{sp})) \overset{s}{\Rightarrow} \mathsf{r},\\ \vdash \mathsf{ha}(\mathsf{r}) \overset{ob}{\Rightarrow} \textbf{ArrayObject}(\textbf{ArrayHeader}\ \_\ \mathsf{ci_s}\ \_\ \textbf{ref}\ \_)\_,\\ \vdash \mathsf{pt}(\mathsf{pi}) \overset{ct}{\Rightarrow} \mathsf{ct},\\ \vdash \mathsf{ct}(\mathsf{ci_s}) \overset{ob}{\Rightarrow} \textbf{ClassObject}\ \_\ \_\ \_\ \_\ \mathsf{super}\ \_,\\ \vdash \mathsf{b2b}(\mathsf{ci_t} = \mathsf{ci_s} \vee \mathsf{ci_t} \in \mathsf{range}(\mathsf{super})) \overset{b}{\Rightarrow} \mathsf{v},\\ \vdash \mathsf{os} \oplus \{\mathsf{sp}-1 \mapsto \mathsf{v}\} \overset{os}{\Rightarrow} \mathsf{os}'\end{array}}{\begin{array}{l}[\mathsf{aainstanceof}]\ \vdash \langle \textbf{aainstanceof}\ \mathsf{ci_t},\ \mathsf{sp},\ \mathsf{os},\ \mathsf{hp},\ \mathsf{ha},\ \mathsf{pi},\ \mathsf{pt}\rangle \overset{instance}{\Rightarrow} \langle \mathsf{sp}-1,\ \mathsf{os}'\rangle,\\ \textbf{if}\ (\mathsf{sp}-1 \ldots \mathsf{sp}) \subseteq\ \mathsf{stackPointer_{range}} \wedge\\ \quad \mathsf{r}\ \in\ \mathsf{heapPointer_{range}} \wedge \mathsf{isArrayObject}(\mathsf{ha}(\mathsf{r})) \wedge\\ \quad \mathsf{pi}\ \in\ \mathsf{progId_{range}} \wedge \mathsf{ci_s}\ \in\ \mathsf{classId_{range}};\end{array}}$$

The three instructions **checkcast**, **acheckcast** and **aacheckcast** below handle, regular objects, array objects of primitive and non-primitive types respectively in the same way as the three 'instance of' instructions above.

The **checkcast** instruction permits a null reference to be cast to any other reference. Otherwise **instanceof** is used to determine whether the cast is acceptable. The operand stack is unaffected.

$$[\mathsf{checkcast}^0]\ \frac{\vdash \mathsf{p2s}(\mathsf{os}(\mathsf{sp}-1),\ \mathsf{os}(\mathsf{sp})) \overset{s}{\Rightarrow} \mathsf{r}}{\begin{array}{l}\vdash \langle \textbf{checkcast}\ \mathsf{ci},\ \mathsf{sp},\ \mathsf{os},\ \mathsf{hp},\ \mathsf{ha},\ \mathsf{pi},\ \mathsf{pt}\rangle \overset{instance}{\Rightarrow} \langle \mathsf{sp},\ \mathsf{os}\rangle,\\ \textbf{if}\ (\mathsf{sp}-1\ \ldots\ \mathsf{sp}) \subseteq\ \mathsf{stackPointer}_{\mathsf{range}} \wedge \mathsf{r} = \mathsf{nullreference};\end{array}}$$

$$[\mathsf{checkcast}^1]\ \frac{\begin{array}{l}\vdash \langle \textbf{instanceof}\ \mathsf{ci},\ \mathsf{sp},\ \mathsf{os},\ \mathsf{hp},\ \mathsf{ha},\ \mathsf{pi},\ \mathsf{pt}\rangle \overset{instance}{\Rightarrow} \langle \mathsf{sp}',\ \mathsf{os}'\rangle,\\ \vdash \mathsf{os}'(\mathsf{sp}') \overset{b}{\Rightarrow} \mathsf{v}\end{array}}{\begin{array}{l}\vdash \langle \textbf{checkcast}\ \mathsf{ci},\ \mathsf{sp},\ \mathsf{os},\ \mathsf{hp},\ \mathsf{ha},\ \mathsf{pi},\ \mathsf{pt}\rangle \overset{instance}{\Rightarrow} \langle \mathsf{sp},\ \mathsf{os}\rangle,\\ \textbf{if}\ \mathsf{v} = 1;\end{array}}$$

The two instructions **acheckcast** and **aacheckcast** rely on the appropriate 'instance of' instructions in a similar way. They are not shown here.

4.20 Loading and Storing Object Fields

The two 'get' instructions below load a value from an object field onto the operand stack. The two 'put' instructions serve to store a field with a byte or a short. There are no agetfield or aputfield instructions. The side conditions check for stack underflow, null references, or a reference to an object of the wrong type. Illegal field indices should be detected by the static semantics.

$$[\mathsf{bgetfield}]\ \frac{\begin{array}{l}\vdash \mathsf{p2s}(\mathsf{os}(\mathsf{sp}-1),\ \mathsf{os}(\mathsf{sp})) \overset{s}{\Rightarrow} \mathsf{r},\\ \vdash \mathsf{ha}(\mathsf{r}) \overset{ob}{\Rightarrow} \textbf{RegularObject}\ \mathsf{oh}\ \mathsf{ft},\\ \vdash \mathsf{ft}(\mathsf{i}) \overset{b}{\Rightarrow} \mathsf{v}\end{array}}{\begin{array}{l}\vdash \langle \textbf{bgetfield}\ \mathsf{i},\ \mathsf{sp},\ \mathsf{os},\ \mathsf{ha}\rangle \overset{getfield}{\Rightarrow} \langle \mathsf{sp}-1,\ \mathsf{os} \oplus \{\mathsf{sp}-1 \mapsto \mathsf{v}\}\rangle,\\ \textbf{if}\ (\mathsf{sp}-1\ \ldots\ \mathsf{sp}) \subseteq\ \mathsf{stackPointer}_{\mathsf{range}} \wedge\\ \quad \mathsf{r}\ \in\ \mathsf{heapPointer}_{\mathsf{range}} \wedge \mathsf{isRegularObject}(\mathsf{ha}(\mathsf{r}));\end{array}}$$

$$[\mathsf{sgetfield}]\ \frac{\begin{array}{l}\vdash \mathsf{p2s}(\mathsf{os}(\mathsf{sp}-1),\ \mathsf{os}(\mathsf{sp})) \overset{s}{\Rightarrow} \mathsf{r},\\ \vdash \mathsf{ha}(\mathsf{r}) \overset{ob}{\Rightarrow} \textbf{RegularObject}\ \mathsf{oh}\ \mathsf{ft},\\ \vdash (\mathsf{ft}(\mathsf{i}),\ \mathsf{ft}(\mathsf{i}+1)) \overset{p}{\Rightarrow} (\mathsf{hi},\ \mathsf{lo}),\\ \vdash \mathsf{os} \oplus \{\mathsf{sp}-1 \mapsto \mathsf{hi}\} \oplus \{\mathsf{sp} \mapsto \mathsf{lo}\} \overset{os}{\Rightarrow} \mathsf{os}'\end{array}}{\begin{array}{l}\vdash \langle \textbf{sgetfield}\ \mathsf{i},\ \mathsf{sp},\ \mathsf{os},\ \mathsf{ha}\rangle \overset{getfield}{\Rightarrow} \langle \mathsf{sp},\ \mathsf{os}'\rangle,\\ \textbf{if}\ (\mathsf{sp}-1\ \ldots\ \mathsf{sp}) \subseteq\ \mathsf{stackPointer}_{\mathsf{range}} \wedge\\ \quad \mathsf{r}\ \in\ \mathsf{heapPointer}_{\mathsf{range}} \wedge \mathsf{isRegularObject}(\mathsf{ha}(\mathsf{r}));\end{array}}$$

$$\frac{\begin{array}{l}\vdash \mathsf{p2s}(\mathsf{os}(\mathsf{sp}-2),\ \mathsf{os}(\mathsf{sp}-1)) \overset{s}{\Rightarrow} \mathsf{r},\\ \vdash \mathsf{os}(\mathsf{sp}) \overset{b}{\Rightarrow} \mathsf{v},\\ \vdash \mathsf{ha}(\mathsf{r}) \overset{ob}{\Rightarrow} \textbf{RegularObject}\ \mathsf{oh}\ \mathsf{ft},\\ \vdash \mathsf{ft} \oplus \{\mathsf{i} \mapsto \mathsf{v}\} \overset{ft}{\Rightarrow} \mathsf{ft}',\\ \vdash \mathsf{ha} \oplus \{\mathsf{r} \mapsto \textbf{RegularObject}\ \mathsf{oh}\ \mathsf{ft}'\} \overset{ha}{\Rightarrow} \mathsf{ha}'\end{array}}{\begin{array}{l}[\mathsf{bputfield}]\ \vdash \langle \textbf{bputfield}\ \mathsf{i},\ \mathsf{sp},\ \mathsf{os},\ \mathsf{ha}\rangle \overset{putfield}{\Rightarrow} \langle \mathsf{sp}-3,\ \mathsf{ha}'\rangle,\\ \textbf{if}\ (\mathsf{sp}-2 \ldots \mathsf{sp}) \subseteq \mathsf{stackPointer}_{\mathsf{range}} \wedge\\ \quad \mathsf{r} \in \mathsf{heapPointer}_{\mathsf{range}} \wedge \mathsf{isRegularObject}(\mathsf{ha}(\mathsf{r}));\end{array}}$$

$$\frac{\begin{array}{l}\vdash \mathsf{p2s}(\mathsf{os}(\mathsf{sp}-3),\ \mathsf{os}(\mathsf{sp}-2)) \overset{s}{\Rightarrow} \mathsf{r},\\ \vdash (\mathsf{os}(\mathsf{sp}-1),\ \mathsf{os}(\mathsf{sp})) \overset{p}{\Rightarrow} (\mathsf{hi},\ \mathsf{lo}),\\ \vdash \mathsf{ha}(\mathsf{r}) \overset{ob}{\Rightarrow} \textbf{RegularObject}\ \mathsf{oh}\ \mathsf{ft},\\ \vdash \mathsf{ft} \oplus \{\mathsf{i} \mapsto \mathsf{hi}\} \oplus \{\mathsf{i}+1 \mapsto \mathsf{lo}\} \overset{ft}{\Rightarrow} \mathsf{ft}',\\ \vdash \mathsf{ha} \oplus \{\mathsf{r} \mapsto \textbf{RegularObject}\ \mathsf{oh}\ \mathsf{ft}'\} \overset{ha}{\Rightarrow} \mathsf{ha}'\end{array}}{\begin{array}{l}[\mathsf{sputfield}]\ \vdash \langle \textbf{sputfield}\ \mathsf{i},\ \mathsf{sp},\ \mathsf{os},\ \mathsf{ha}\rangle \overset{putfield}{\Rightarrow} \langle \mathsf{sp}-4,\ \mathsf{ha}'\rangle,\\ \textbf{if}\ (\mathsf{sp}-3 \ldots \mathsf{sp}) \subseteq \mathsf{stackPointer}_{\mathsf{range}} \wedge\\ \quad \mathsf{r} \in \mathsf{heapPointer}_{\mathsf{range}} \wedge \mathsf{isRegularObject}(\mathsf{ha}(\mathsf{r}));\end{array}}$$

4.21 Loading and Storing Static Objects

Static objects are kept in the static area. The instructions **bgetstatic**, **sgetstatic**, **bputstatic**, and **sputstatic** are used to manipulate static objects.

$$\frac{\begin{array}{l}\vdash \mathsf{p2s}(\mathsf{hi}_{\mathsf{r}},\ \mathsf{lo}_{\mathsf{r}}) \overset{s}{\Rightarrow} \mathsf{i},\\ \vdash \mathsf{sa}(\mathsf{i}) \overset{b}{\Rightarrow} \mathsf{v},\\ \vdash \mathsf{os} \oplus \{\mathsf{sp}+1 \mapsto \mathsf{v}\} \overset{os}{\Rightarrow} \mathsf{os}'\end{array}}{\begin{array}{l}[\mathsf{bgetstatic}]\ \vdash \langle \textbf{bgetstatic}\ \mathsf{hi}_{\mathsf{r}}\ \mathsf{lo}_{\mathsf{r}},\ \mathsf{sp},\ \mathsf{os},\ \mathsf{sa}\rangle \overset{getstatic}{\Rightarrow} \langle \mathsf{sp}+1,\ \mathsf{os}'\rangle,\\ \textbf{if}\ (\mathsf{sp}+1) \in \mathsf{stackPointer}_{\mathsf{range}};\end{array}}$$

$$\frac{\begin{array}{l}\vdash \mathsf{p2s}(\mathsf{hi}_{\mathsf{r}},\ \mathsf{lo}_{\mathsf{r}}) \overset{s}{\Rightarrow} \mathsf{i},\\ \vdash (\mathsf{sa}(\mathsf{i}),\ \mathsf{sa}(\mathsf{i}+1)) \overset{p}{\Rightarrow} (\mathsf{hi}_{\mathsf{v}},\ \mathsf{lo}_{\mathsf{v}}),\\ \vdash \mathsf{os} \oplus \{\mathsf{sp}+1 \mapsto \mathsf{hi}_{\mathsf{v}}\} \oplus \{\mathsf{sp}+2 \mapsto \mathsf{lo}_{\mathsf{v}}\} \overset{os}{\Rightarrow} \mathsf{os}'\end{array}}{\begin{array}{l}[\mathsf{sgetstatic}]\ \vdash \langle \textbf{sgetstatic}\ \mathsf{hi}_{\mathsf{r}}\ \mathsf{lo}_{\mathsf{r}},\ \mathsf{sp},\ \mathsf{os},\ \mathsf{sa}\rangle \overset{getstatic}{\Rightarrow} \langle \mathsf{sp}+2,\ \mathsf{os}'\rangle,\\ \textbf{if}\ (\mathsf{sp}+1 \ldots \mathsf{sp}+2) \subseteq \mathsf{stackPointer}_{\mathsf{range}};\end{array}}$$

$$\frac{\begin{array}{l}\vdash \mathsf{p2s}(\mathsf{hi}_{\mathsf{r}},\ \mathsf{lo}_{\mathsf{r}}) \overset{s}{\Rightarrow} \mathsf{i},\\ \vdash \mathsf{os}(\mathsf{sp}) \overset{b}{\Rightarrow} \mathsf{v}\end{array}}{\begin{array}{l}[\mathsf{bputstatic}]\ \vdash \langle \textbf{bputstatic}\ \mathsf{hi}_{\mathsf{r}}\ \mathsf{lo}_{\mathsf{r}},\ \mathsf{sp},\ \mathsf{os},\ \mathsf{sa}\rangle \overset{putstatic}{\Rightarrow} \langle \mathsf{sp}-1,\ \mathsf{sa} \oplus \{\mathsf{i} \mapsto \mathsf{v}\}\rangle,\\ \textbf{if}\ \mathsf{sp} \in \mathsf{stackPointer}_{\mathsf{range}};\end{array}}$$

$$\text{[sputstatic]}\ \frac{\begin{array}{l}\vdash \mathsf{p2s}(\mathsf{hi_r},\ \mathsf{lo_r}) \stackrel{s}{\Rightarrow} \mathsf{i},\\ \vdash (\mathsf{os}(\mathsf{sp}-1),\ \mathsf{os}(\mathsf{sp})) \stackrel{p}{\Rightarrow} (\mathsf{hi_v},\ \mathsf{lo_v}),\\ \vdash \mathsf{sa} \oplus \{\mathsf{i} \mapsto \mathsf{hi_v}\} \oplus \{\mathsf{i}+1 \mapsto \mathsf{lo_v}\} \stackrel{sa}{\Rightarrow} \mathsf{sa}'\end{array}}{\begin{array}{l}\vdash \langle \textbf{sputstatic}\ \mathsf{hi_r}\ \mathsf{lo_r},\ \mathsf{sp},\ \mathsf{os},\ \mathsf{sa}\rangle \stackrel{putstatic}{\Rightarrow} \langle \mathsf{sp}-2,\ \mathsf{sa}'\rangle,\\ \textbf{if}\ (\mathsf{sp}-1\ \ldots\ \mathsf{sp}) \subseteq\ \mathsf{stackPointer_{range}};\end{array}}$$

4.22 Miscellaneous Instructions

The **breakpoint** instruction pops the top two elements of the operand stack, interprets them as the high and low byte of a short and appends the short to the output stream.

$\mathsf{outputStream} \equiv [\mathsf{short}];$

$$\text{[breakpoint]}\ \frac{\vdash \mathsf{p2s}(\mathsf{os}(\mathsf{sp}-1),\ \mathsf{os}(\mathsf{sp})) \stackrel{s}{\Rightarrow} \mathsf{v}}{\begin{array}{l}\vdash \langle \textbf{breakpoint},\ \mathsf{sp},\ \mathsf{os},\ \mathsf{output}\rangle \stackrel{breakpoint}{\Rightarrow} \langle \mathsf{sp}-2,\ \mathsf{output} +\!\!+ [\mathsf{v}]\rangle,\\ \textbf{if}\ (\mathsf{sp}-1\ \ldots\ \mathsf{sp}) \subseteq\ \mathsf{stackPointer_{range}};\end{array}}$$

4.23 Combining the Rules

The semantics of the 25 subsets of the instruction set are specified by as many different relations, such as $\stackrel{const}{\Rightarrow}$. These different relations are embedded in the relation $\stackrel{exec}{\Rightarrow}$ by the rules below. The $\stackrel{exec}{\Rightarrow}$ relation also automatically increments the program counter by one upon completing the execution of an instruction, with a few exceptions detailed below.

The separation of the different categories of instructions shows that the specification is modular: The configuration of the virtual machine has 12 components, which is quite large. However, the relation for many of the subsets uses only a small number of components, thus hiding the remaining components.

$$[\mathsf{exec}^{\mathsf{const}}]\ \frac{\vdash \langle \mathsf{constInst},\ \mathsf{sp},\ \mathsf{os}\rangle \stackrel{const}{\Rightarrow} \langle \mathsf{sp}',\ \mathsf{os}'\rangle}{\begin{array}{l}\vdash \langle \mathsf{pc},\ \mathsf{ca},\ \mathsf{constInst},\ \mathsf{sp},\ \mathsf{os},\ \mathsf{fp},\ \mathsf{fa},\ \mathsf{hp},\ \mathsf{ha},\ \mathsf{pi},\ \mathsf{pt},\ \mathsf{sa},\ \mathsf{output}\rangle\\ \stackrel{exec}{\Rightarrow} \langle \mathsf{pc}+1,\ \mathsf{sp}',\ \mathsf{os}',\ \mathsf{fp},\ \mathsf{fa},\ \mathsf{hp},\ \mathsf{ha},\ \mathsf{sa},\ \mathsf{output}\rangle;\end{array}}$$

Most other relations defining subsets of the instruction set are embedded in the relation $\stackrel{execs}{\Rightarrow}$ in the same way as shown above. The exception to this rule is formed by the relations $\stackrel{return}{\Rightarrow}$, $\stackrel{control}{\Rightarrow}$, $\stackrel{switch}{\Rightarrow}$, and $\stackrel{invoke...}{\Rightarrow}$, which calculate the new value of the program counter pc'. The automatic increment of the program counter is thus suppressed.

$$[\mathsf{exec}^{\mathsf{return}}]\ \frac{\vdash \langle \mathsf{returnInst},\ \mathsf{sp},\ \mathsf{os},\ \mathsf{fp},\ \mathsf{fa}\rangle \stackrel{return}{\Rightarrow} \langle \mathsf{pc}',\ \mathsf{sp}',\ \mathsf{os}',\ \mathsf{fp}'\rangle}{\begin{array}{l}\vdash \langle \mathsf{pc},\ \mathsf{ca},\ \mathsf{returnInst},\ \mathsf{sp},\ \mathsf{os},\ \mathsf{fp},\ \mathsf{fa},\ \mathsf{hp},\ \mathsf{ha},\ \mathsf{pi},\ \mathsf{pt},\ \mathsf{sa},\ \mathsf{output}\rangle\\ \stackrel{exec}{\Rightarrow} \langle \mathsf{pc}',\ \mathsf{sp}',\ \mathsf{os}',\ \mathsf{fp}',\ \mathsf{fa},\ \mathsf{hp},\ \mathsf{ha},\ \mathsf{sa},\ \mathsf{output}\rangle;\end{array}}$$

4.24 Main Semantic Function

The function jsp defines the semantics of a JSP programs the transitive closure of the relation $\overset{decode}{\Rightarrow}$ (below). When given an initial JSP machine configuration, jsp computes a list of successive configurations that can be inspected.

$$\begin{aligned}
&\mathsf{configuration} \equiv \langle \mathsf{programCounter},\ \mathsf{codeArea},\ \mathsf{stackPointer},\ \mathsf{operandStack},\\
&\qquad \mathsf{framePointer},\ \mathsf{frameArea},\ \mathsf{heapPointer},\ \mathsf{heapArea},\\
&\qquad \mathsf{progId},\ \mathsf{progTable},\ \mathsf{staticArea},\ \mathsf{outputStream}\rangle;\\
&\mathsf{jsp} \quad :: \mathsf{configuration} \rightarrow [\mathsf{configuration}];\\
&\mathsf{jsp\ s0} \quad = (\mathsf{s0} \overset{decode}{\Rightarrow} *);
\end{aligned}$$

The relation $\overset{decode}{\Rightarrow}$ accesses the instruction at the current program counter. The case analysis by the $\overset{exec}{\Rightarrow}$ relation decides to which category the current instruction belongs and delegates the actual processing of the instruction to the appropriate embedded relation.

$$\begin{aligned}
&\overset{decode}{\Rightarrow} \quad :: (\mathsf{configuration} \leftrightarrow \mathsf{configuration});\\
&\qquad \vdash \langle \mathsf{pc},\ \mathsf{ca},\ \mathsf{ca(pc)},\ \mathsf{sp},\ \mathsf{os},\ \mathsf{fp},\ \mathsf{fa},\ \mathsf{hp},\ \mathsf{ha},\ \mathsf{pi},\ \mathsf{pt},\ \mathsf{sa},\ \mathsf{output}\rangle\\
&\qquad \overset{exec}{\Rightarrow} \langle \mathsf{pc}',\ \mathsf{sp}',\ \mathsf{os}',\ \mathsf{fp}',\ \mathsf{fa}',\ \mathsf{hp}',\ \mathsf{ha}',\ \mathsf{sa}',\ \mathsf{output}'\rangle\\
&[\mathsf{decode}] \quad \overline{\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad}\\
&\qquad \vdash \langle \mathsf{pc},\ \mathsf{ca},\ \mathsf{sp},\ \mathsf{os},\ \mathsf{fp},\ \mathsf{fa},\ \mathsf{hp},\ \mathsf{ha},\ \mathsf{pi},\ \mathsf{pt},\ \mathsf{sa},\ \mathsf{output}\rangle\\
&\qquad \overset{decode}{\Rightarrow} \langle \mathsf{pc}',\ \mathsf{ca},\ \mathsf{sp}',\ \mathsf{os}',\ \mathsf{fp}',\ \mathsf{fa}',\ \mathsf{hp}',\ \mathsf{ha}',\ \mathsf{pi},\ \mathsf{pt},\ \mathsf{sa}',\ \mathsf{output}'\rangle;
\end{aligned}$$

A sample machine configuration such as test (see Section 6) can be supplied as an argument to jsp.

5 On the Relationship Between the JVM and the JSP

The JSP is essentially a scaled down version of the JVM. However, the JSP byte codes are not a strict subset of the JVM and translating JVM byte codes into JSP byte codes presents some interesting problems. This section comments on the relationship between the two virtual machines and sketches a simplified process of translating Java class files into the tables required to run JSP code.

The main problem of translating JVM byte codes into JSP bytecodes is the pervasive use of 32-bit data in Java programs. The translator built by Java Soft performs a sophisticated analysis to ensure that the computations performed by the JSP have the same semantics as those carried out by the JVM. The results of the analysis enable the translator to map certain integers and associated operations on bytes, and some on shorts. The translator also inserts instructions to support multiple precision arithmetic when genuine 32-bit integers are needed.

The simplified translation to be described here assumes that all integers can be represented as shorts. We make no attempt to either identify opportunities for using bytes or to warn if shorts are too limited.

The translation of Java class files into the tables required by the JSP consists of the following steps:

- To allocate all statics in the staticArea, to create an index of all application programs in the progTable, and to gather the code sections of all methods in the codeArea.

- For each application program to allocate a classTable.
- For each class to allocate a classObject with its objectHeader, a methodTable, a superTable, and an interfaceTable, and to decide on the layout of the fields in the instance of the class.
- For each method to allocate a methodHeader, to gather the byte codes of the method and to decide on a start address of the method.
- For each word offset, address or integer to convert it into a short. Depending on the sophistication of the translation process this may simply truncate all values, or restructure the byte code to deal with values that cannot be fit into 16 bits.
- For each instruction to convert it as indicated below.

To present the translation of individual JVM byte codes into JSP byte codes in a reasonably succinct manner we use the following abbreviations:

- byte, short, index, params and address stand for numeric values in the appropriate range.
- class, field, method, and static stand for the appropriate name.
- $[a|b|c]$ stands for exactly one of the words a, b or c.

We list all JVM instructions [7] (on the left), and describe the equivalent JSP instruction or sequence of instructions (on the right).

- Constant instructions.
  - nop = **nop**;
  - bipush byte = **spush** 0 byte;
  - sipush short = **spush**(short div 256)(short mod 256);
  - aconst$_{\text{null}}$ = **aconst**$_{\text{null}}$;
  - iconst$_{\text{m1}}$ = **bconst**$_0$, **bconst**$_{\text{m1}}$;
  - iconst$_{[0|1|2|3|4|5]}$ = **bconst**$_0$, **bconst**$_{[0|1|2|3|4|5]}$;
  - iconst short = **bpush**(short div 256), **bpush**(short mod 256);
  - iconst byte = **bpush** byte;
- The load, store and increment instructions.
  - [a|i]load$_{[0|1]}$ = [**A**|**S**]**load**$_{[0|2]}$;
  - [a|i]load$_{[2|3]}$ = [**A**|**S**]**load** [4|6];
  - [a|i]load index = [**A**|**S**]**load**(2*index);
  - [a|i]store$_{[0|1]}$ = [**A**|**S**]**store**$_{[0|2]}$;
  - [a|i]store$_{[2|3]}$ = [**A**|**S**]**store** [4|6];
  - [a|i]store index = [**A**|**S**]**store**(2*index);
  - iinc index byte = **sinc**(2*index)byte;
- Stack instructions.
  - dup = **dup2**;
  - dup_x [1|2] = **dup_x**(2*16 + [2|4]);
  - dup2 = **dup_x**(4*16 + 4);
  - dup2_x [1|2] = **dup_x**(4*16 + [6|8]);
  - pop = **pop2**;
  - pop2 = **pop2**, **pop2**;
  - swap = **swap2**;

– Array creation, load and store instructions.
  anewarray class = **anewarray** class;
  newarray [boolean|byte|short|int] = **newarray** [**bit**|**byte**|**short**|**short**];
  arraylength = **arraylength**;
  [a|b|i|s]load = [**A**|**B**|**S**|**S**]**load**;
  [a|b|i|s]store = [**A**|**B**|**S**|**S**]**store**;
– Instructions for arithmetical, logical and conversion operations.
  i[add|sub|mul|div|rem] = **S**[**add**|**sub**|**mul**|**div**|**rem**];
  i[shl|shr|ushr] = **S**[**shl**|**shr**|**ushr**];
  i[and|or|xor] = **S**[**and**|**or**|**xor**];
  i2b = **s2b**;
  i2s = **nop**;
– The JVM Conditional branches translate into a number of JSP instructions.
  ifnonnull address = **aconst**$_{\mathsf{null}}$, **acmp**, **ifne** address;
  ifnull address = **aconst**$_{\mathsf{null}}$, **acmp**, **ifeq** address;
  if[a|i]cmp[eq|lt|gt|ne|ge|le] address = [**A**|**S**]**cmp**, **If**[**eq**|**lt**|**gt**|**ne**|**ge**|**le**] address;
  if[eq|lt|gt|ne|ge|le] address = **s2b**, **If**[**eq**|**lt**|**gt**|**ne**|**ge**|**le**] address;
  goto address = **goto** address;
– The JVM instructions tableswitch and lookupswitch are variable length instructions. The tables may contain an arbitrary number of index/target or key/target pairs.
  tableswitch from to default{index $\mapsto$ address} =
  tableswitch default from to{index $\mapsto$ address};
  lookupswitch size default{index $\mapsto$ (key, address)} =
  lookupswitch default size{index $\mapsto$ (key, address)};
– Exception handling.
  athrow = **athrow**;
  jsr address = **jsr** address;
  ret index = **ret**(2*index);
– Instructions for method invokation.
  invokeinterface params class method = **invokeinterface** params class method;
  invokespecial address = **invoke** address;
  invokestatic address = **invoke** address;
  invokevirtual params method = **invokevirtual** params method;
  [a|i]return = [**A**|**S**]**return**;
  return = **return**;
– Instructions for object creation and manipulation.
  new class = **new** class;
  instanceof class = **instanceof** class, **b2s**;
  checkcast class = **checkcast** class;
  getfield field = **sgetfield** field;
  putfield field = **sputfield** field;
  getstatic static = **sgetstatic** static;
  putstatic static = **sputstatic** static;
– Miscellaneous instructions.
  breakpoint = **breakpoint**;

– All other JVM instructions are unsupported. These are jsr_w, goto_w, wide, monitorenter, monitorexit, multianewarray, and all instructions involving character, long, float, and double data types.

We use SUN's Java compiler from the Java Development Kit version 1.1 to generate class files from sample Java programs. The translations sketched above have been implemented as a simple sed/awk script, such that the results of the translation can be used as sample input for the main semantic function jsp. This will be explored briefly in the next section.

6 A Sample Program

We have written a suite of simple Java programs, varying from quick sort to specific tests for the object system, to validate aspects of the semantics. The workings of the JSP semantics is best illustrated by exposing some details of a representative program from our suite. The program below is a slightly modified version of [4, Page 48]. The two calls to `println` have been added to show that the program is working. Furthermore we have added the call to `setColor` to demonstrate the workings of multiple inheritance.

```
public class Point{ int x, y; } ;

public interface Colorable {
  void setColor( byte r, byte g, byte b) ;
}

public class ColoredPoint extends Point implements Colorable {
  byte r,g,b;
  public void setColor( byte rv, byte gv, byte bv ) {
    r = rv ; g = gv; b = bv ;
  }
}

public class test {
  public static void main( String [] args ) {
    Point p = new Point() ;
    ColoredPoint cp = new ColoredPoint() ;
    p = cp ;
    System.out.println( p.x ) ;
    Colorable c = cp ;
    c.setColor( (byte) 0, (byte) 1, (byte) 2 ) ;
    System.out.println( cp.b ) ;
  }
}
```

The 12 components of the JSP virtual machine configuration necessary to execute test.main are initialised as follows:

program counter The program counter is initialised to 0.

code area The code for all methods to be executed by the current application program (which includes the initialiser for java.lang.Object) is gathered in the code area. An extra instruction at address zero is added to the code area whose task it is to invoke the main method. This is represented as $0 \mapsto (\textbf{invoke}\ \mathsf{s2p}(\mathsf{test.main_{pc}}))$

stack pointer The initial value of the stack pointer is argc.

$\mathsf{argc} :: \mathsf{stackPointer};$
$\mathsf{argc} = 1;$

operand stack Initially the operand stack is the same as argv.

$\mathsf{argv} :: \mathsf{operandStack};$
$\mathsf{argv} = \{0 \mapsto 0,\ 1 \mapsto 0\};$

frame pointer The initial value of the frame pointer is -1, to indicate that the frame area is initially empty.

frame area The initial frame area is empty.

heap pointer The initial heap pointer is -1, indicating an empty heap.

heap The heap is initially empty.

application program index $\mathsf{test_{pi}}$ is the index in the application program table of the current application program. The formal specification presently does not specify a mechanism for switching application programs.

application program table $\mathsf{machine_{pt}}$ is the machine wide mapping from application program ids to a class tables, providing one class table per application program.

Static area $\mathsf{machine_{sa}}$ is the machine wide area used to store static values. The sample program does not have any static values.

Initial output The initial output stream is empty.

$$\begin{aligned}&\mathsf{test} :: \mathsf{configuration};\\ &\mathsf{test} = \langle 0,\ \{0 \mapsto \textbf{invoke}(\mathsf{s2p}(\mathsf{test.main_{pc}}))\} \cup\ \mathsf{machine_{ca}},\\ &\qquad \mathsf{argc},\ \mathsf{argv},\ -1,\ \{\},\ -1,\ \{\},\ \mathsf{test_{pi}},\ \mathsf{machine_{pt}},\ \mathsf{machine_{sa}},\ []\rangle;\end{aligned}$$

The JSP byte codes for the main method of class test are shown below. Instead of calling the println method of the library class System, we use the **breakpoint** instruction to inspect the configuration of the machine.

$\mathsf{test.main_{ca}} :: \mathsf{codeArea};$
$\mathsf{test.main_{ca}} = \{\mathsf{test.main_{pc}} - 1 \mapsto \textbf{MethodHeader}\ \mathsf{False}\ \mathsf{False}\ 8\ 2\ 6,$
$\mathsf{test.main_{pc}} + 0 \mapsto \textbf{new}\ \mathsf{Point_{ci}},$
$\mathsf{test.main_{pc}} + 1 \mapsto \textbf{dup2},$
$\mathsf{test.main_{pc}} + 2 \mapsto \textbf{invoke}(\mathsf{s2p}\ \mathsf{Point.init_{pc}}),$
$\mathsf{test.main_{pc}} + 3 \mapsto \textbf{astore}_2,$
$\mathsf{test.main_{pc}} + 4 \mapsto \textbf{new}\ \mathsf{ColoredPoint_{ci}},$
$\mathsf{test.main_{pc}} + 5 \mapsto \textbf{dup2},$
$\mathsf{test.main_{pc}} + 6 \mapsto \textbf{invoke}(\mathsf{s2p}\ \mathsf{ColoredPoint.init_{pc}}),$
$\mathsf{test.main_{pc}} + 7 \mapsto \textbf{astore}\ 4,$
$\mathsf{test.main_{pc}} + 8 \mapsto \textbf{aload}\ 4,$
$\mathsf{test.main_{pc}} + 9 \mapsto \textbf{astore}_2,$
$\mathsf{test.main_{pc}} + 10 \mapsto \textbf{nop},$
$\mathsf{test.main_{pc}} + 11 \mapsto \textbf{aload}_2,$
$\mathsf{test.main_{pc}} + 12 \mapsto \textbf{sgetfield}\ \mathsf{Point.x_{fi}},$
$\mathsf{test.main_{pc}} + 13 \mapsto \textbf{breakpoint},$
$\mathsf{test.main_{pc}} + 14 \mapsto \textbf{aload}\ 4,$
$\mathsf{test.main_{pc}} + 15 \mapsto \textbf{astore}\ 6,$
$\mathsf{test.main_{pc}} + 16 \mapsto \textbf{aload}\ 6,$
$\mathsf{test.main_{pc}} + 17 \mapsto \textbf{bconst}_0,\ \mathsf{test.main_{pc}} + 18 \mapsto \textbf{bconst}_0,$
$\mathsf{test.main_{pc}} + 19 \mapsto \textbf{bconst}_0,\ \mathsf{test.main_{pc}} + 20 \mapsto \textbf{bconst}_1,$
$\mathsf{test.main_{pc}} + 21 \mapsto \textbf{bconst}_0,\ \mathsf{test.main_{pc}} + 22 \mapsto \textbf{bconst}_2,$
$\mathsf{test.main_{pc}} + 23 \mapsto \textbf{invokeinterface}\ 8\ \mathsf{Colorable_{ii}}\ \mathsf{Colorable.setColor_{mi}},$
$\mathsf{test.main_{pc}} + 24 \mapsto \textbf{nop},$
$\mathsf{test.main_{pc}} + 25 \mapsto \textbf{aload}\ 4,$
$\mathsf{test.main_{pc}} + 26 \mapsto \textbf{sgetfield}\ \mathsf{ColoredPoint.b_{fi}},$
$\mathsf{test.main_{pc}} + 27 \mapsto \textbf{breakpoint},$
$\mathsf{test.main_{pc}} + 28 \mapsto \textbf{return}\};$

The execution of the program can be expressed simply as jsp(test). The `latos` tool makes it possible to trace the execution of the program, and to experiment with different initial configurations.

The program starts by creating two heap objects, one representing a Point and the second representing a ColoredPoint. The objects are properly initialised by a chain of calls to the initialisers of the super classes. The most interesting instruction is the **invokeinterface**, which has to discover that the instance of ColoredPoint indeed implements the setColor method.

The program causes two values to be appended to the output stream (via the **breakpoint** instruction). The values are 0 (because the coordinates of the class Point are initialised to 0) and 2 (because ColoredPoint.setColor assigns this value to the field cp.b).

7 Conclusions and Future Work

The result of formalising the operational semantics of the JSP is a specification that is:

- succinct, because it is shorter and more detailed than the natural language documents.
- clear, because the rules are not open to more than one interpretation.
- executable, because a program can be generated automatically from the specification, which can subsequently be executed to validate and explore the behaviour of sample Java programs.
- consistent, because the tools available for the notation used check well formedness, types and source dependency.
- modular, because sub sets of rules can be considered in isolation.
- large, because it has to cope with 25 groups of 124 different JSP instructions.
- not difficult to read, because the rules describing the semantics of many instructions are similar.

The fact that our specification is executable allows implementors to experiment with Java programs and byte codes, inspect the configuration of the JSP and generally sharpen their understanding of the mechanisms. Without tool support it would be impossible to construct a derivation tree for anything but the most trivial Java programs. With the help of our `latos` tool, our specification could be used to automatically construct derivation trees for small to medium sized programs.

We hope to be able to make our complete specification available on the Web, so that others may down load the specification and the `latos` tool and use these resources whilst implementing a JSP.

In future we hope to gain access to a complete operational semantics of the JVM, formally specify the JVM to JSP translator and attempt to give a correctness proof of the translator with respect to the semantics of the JVM byte codes and that of the JSP byte codes.

We have not considered the static semantics of a JSP, that is a specification of properties of JSP programs that can be be checked statically, for example by the JVM to JSP byte code translator, or the byte code verifier. An important goal would be to investigate which static properties of the JVM that are preserved by the JVM to JSP translator. The work of Stata and Abadi [10] offers a promising basis for this.

References

1. ISO/IEC 7816-4:1995. *Information technology–Identification cards–Integrated circuit(s) cards with contacts part4: Inter-Industry commands for interchange.* International Standards Organization, 1995.
2. P. Bertelsen. Semantics of Java byte code. Technical report, Technical Univ. of Denmark, Mar 1997. `www.dina.kvl.dk/~pmb/`.

3. R. M. Cohen. The defensive java virtual machine specification version 0.5. Technical report, Computational Logic Inc, Austin, Texas, May 1997. `www.cli.com/`.
4. J. Gosling, B. Joy, and G. Steele. *The Java Language Specification*. Addison Wesley, Reading, Massachusetts, 1996.
5. P. H. Hartel. LATOS – a lightweight animation tool for operational semantics. Technical report DSSE-TR-97-1, Dept. of Electr. and Comp. Sci, Univ. of Southampton, England, Oct 1997. `www.ecs.soton.ac.uk/~phh/latos.html`.
6. M. Levy. *Java Secure processor language specification version 0.99*. Integrity Arts Inc., San Mateo, California, May 1997.
7. T. Lindholm and F. Yellin. *The Java Virtual Machine Specification*. Addison Wesley, Reading, Massachusetts, 1996.
8. G. McGraw and E. W. Felten. *Java security: Hostile applets, holes and antidotes*. John Wiley & Sons, Chichester, England, 1997.
9. P. Peyret. Application-enabling card systems with plug-and-play applets. In *Smart Card 1996 convention proceedings – Technology and markets conference*, pages 51–72. Quality marketing services Ltd, Peterborough, UK, Feb 1996.
10. R. Stata and M. Abadi. A type system for Java bytecode subroutines. In *25th Principles of programming languages (POPL)*, pages 149–160, San Diego, California, Jan 1998. ACM, New York.
11. D. A. Turner. Miranda: A non-strict functional language with polymorphic types. In J.-P. Jouannaud, editor, *2nd Functional programming languages and computer architecture, LNCS 201*, pages 1–16, Nancy, France, Sep 1985. Springer-Verlag, Berlin.
12. J. L. Zoreda and J. M. Otón. *Smart Cards*. Artech House Inc, Norwood, Massachusetts, 1994.

A Programmer Friendly Modular Definition of the Semantics of Java*

Egon Börger[1] and Wolfram Schulte[2]

[1] Università di Pisa, Dipartimento di Informatica, I-56125 Pisa, Italy
boerger@di.unipi.it
[2] Universität Ulm, Fakultät für Informatik, D-89069 Ulm, Germany
wolfram@informatik.uni-ulm.de

Abstract. We propose in this paper a definition of the semantics of Java programs which can be used as a basis for the standardization of the language and of its implementation on the Java Virtual Machine. The definition provides a machine and system independent view of the language as it is seen by the Java programmer. It takes care to *directly* reflect the description in the Java language reference manual so that the basic design decisions can be checked by standardizers and implementors against a mathematical model.
Our definition is the basis for a related definition we give in a sequel to this paper for the implementation of Java on the Java Virtual Machine as described in the language and in the Virtual Machine reference manuals.

1 Introduction

In this chapter we formalize the semantics of Java by a system independent, purely mathematical yet easily manageable model, which reflects directly the intuitions and design decisions underlying the language as described in Java's language reference manual (LRM) [19]. Our goal is to contribute to a rigorous yet readable definition of the entire language, which supports the programmer's understanding of Java programs. At the same time the definition should provide a basis for the standardization and clarification of critical language features, for the specification and evaluation of variations or extensions of the language and for the mathematical analysis and comparison of Java implementations. In particular we aim for a model that is amenable to both mathematical and computer assisted proofs and to experimental validation of the correctness of compilation schemes to Java Virtual Machine (JVM) code and of safety properties of Java programs when executed on the JVM.

These goals oblige us to abstract the central ideas of Java's LRM into a transparent but rigorous form, whose adequacy can be recognized (or falsified in the sense of Popper[29]) by inspection, i.e., by a direct comparison of the mathematical definitions with the verbal descriptions in the manual (see the discussion on

* A preliminary version of this paper has been presented to the IFIP WG 2.2 Meeting in Graz (22.–26.9.1997) and to the Workshop on Programming Languages in Avendorf/Fehmarn (24.–26.9.1997) [12].

Jim Alves-Foss (Ed.): Formal Syntax and Semantics of Java, LNCS 1523, pp. 353–404, 1999.

ground models in [2]). To be able to establish the required simplicity and faithfulness of such abstractions, one needs a modeling technique which provides the following two possibilities:

- to express the basic language concepts (its objects and operations) directly, without encoding, i.e., as abstract entities as they appear in the LRM,
- to model basic actions on the level of abstraction of the LRM, i.e., as local modifications with clear preconditions and effects and avoiding any a priori imposed static representations of actions-in-time.

Gurevich's *Abstract State Machines* (ASMs), previously called Evolving Algebras, see [20], provide the fundamental concept for such a modeling technique. ASMs have been successfully used to model the semantics and implementation of programming languages as different as Prolog [10,11], Occam[6,5], VHDL [7], C++[36] and others. Furthermore, ASMs are effective in modeling architectures [4,8], protocols [21,1], control software [3,9], and by being amenable to execution (see for an ASM interpreter [17]) they can be used for high-level validation. See [2] for a survey. ASMs have a simple mathematical foundation [20], which justifies their intuitive understanding as "pseudo-code over abstract data" so that the practitioner can use them correctly and successfully without having to go through any special formal training. Therefore we invite the reader to go ahead with reading our specification and to consult the formal definition of ASMs in [20] only should the necessity arise.

We formally define the *semantics* of Java by providing an ASM which interprets arbitrary Java programs. A Java program consists of a set of classes. In the use of a class there are three phases: parsing, elaboration, and execution. Parsing determines the grammatical form yielding an abstract syntax tree. Elaboration, the static phase, determines whether the class is well-typed and well-formed and records such information as annotations in the abstract syntax tree. Execution, the dynamic phase, loads, links, and executes the code of the class by evaluating expressions and modifying the memory. Corresponding to these phases, a full mathematical definition of Java needs a grammar, a static and a dynamic semantics. The grammar is well defined in the LRM [19]. Numerous authors have formalized the static semantics of sequential Java, in particular its type soundness [18,26,31,35]. The dynamic semantics given in these papers cover only a small structured sublanguage of sequential Java and do not consider the interaction of jump statements (like `break`), exception handling and concurrency, which we treat in full. We therefore concentrate in this paper on a complete but nevertheless transparent mathematical definition of the *dynamic* semantics of Java.

Two features characterize our modeling of the dynamics of Java programs: it is run-time instead of syntax oriented and it comes with a systematic separation of static and dynamic concerns.

To let the dynamic aspects stand out as clearly as possible, we relegate compile-time matters to static functions as much as we can without making the specification unreadable for the Java practitioners with no training in formal methods. As is well known such a separation of statics and dynamics also lays

the ground for efficient implementations of the static features for program interpretation and for the generation of program debuggers, animation tools etc. (see for example [28]). In addition it has led to an interesting and novel integration, into ASM specifications, of various useful methods from functional programming and algebraic specifications for the definition of static (compile-time) functions.

After some experimentation we decided to strictly stick to a run-time and not syntax-directed modeling. Structural methods (like SOS [27], natural semantics [22], action semantics [25], etc.) are known to work well for the definition of languages where the control flow essentially follows the syntax (tree) structure with only little involvement of environment information (as is the case for example for purely functional languages or for strongly syntax supported languages). Structural methods offer however no advantage in cases of languages like Java where the participation of the run-time environment in determining the program control flow becomes more complex and when concurrency features—which are not syntax driven—enter the scene. The decision for a strictly process-oriented modeling throughout the entire Java language provides the programmer with a uniform view of the intricate interaction of the different language features like jumps, returns, exceptions, concurrency and synchronization. The use of ASMs makes this view particularly transparent (and thus easily comparable with the verbal explanations in the LRM): ASM specifications concentrate on local changes which avoids having to carry, for a given action, global contexts which remain constant for this action.

Before we proceed to the technical overview of the paper we want to make clear that our paper is not an introduction to (programming in) Java; the intended reader is familiar with Java: a programmer, a standardizer, implementor or teacher who looks for a rigorous but easy to understand language definition.

1.1 Overview

To make the complete dynamic semantics of Java manageable, we factor it into *five sublanguages*, by isolating orthogonal language features, namely imperative, procedural, object-oriented, exception handling and concurrency features. This can be made in such a way that each corresponding ASM model is a conservative extension of its predecessor. We found it interesting to discover at a later stage of our work on the Java language that an analogous modular decomposition can be given also for models of the JVM [14].

Section 2 defines the basic ASM $\text{Java}_{\mathcal{I}}$ for Java's imperative core, which is essentially a `while` language. It contains statements and expressions over Java's primitive types. This section provides an introduction to our approach and notation.

In Sect. 3, we upgrade $\text{Java}_{\mathcal{I}}$ to $\text{Java}_{\mathcal{C}}$ by including Java's classes; $\text{Java}_{\mathcal{C}}$ supports class fields, class methods, and class initializers. Thus, $\text{Java}_{\mathcal{C}}$ defines an object-based sublanguage of Java, which supports procedural abstraction and (module-) global variables.

In Sect. 4, we extend Java$_{\mathcal{C}}$ to Java$_{\mathcal{O}}$ by including Java's (real) object-oriented concepts, namely instances, instance creation, instance field access, instance method calls with late binding, casts, and null pointers.

Section 5 extends Java$_{\mathcal{O}}$ with exceptions, resulting in the model Java$_{\mathcal{E}}$. We specify which exception will be thrown when semantic constraints are violated. We introduce Java's `throw` and `try/catch/finally` statement, and we exhibit the interaction of exception handling with other language constructs.

In Sect. 6, we move from sequential Java to concurrent Java, the corresponding ASM model Java$_{\mathcal{T}}$ introduces Java's lightweight processes, called threads, their synchronization mechanism using locks, and their stopping, waiting and notification mechanism. We study two complementary memory models: the first one uses only the main memory for storing objects, the second model uses the local working memory as much as possible. For 'best practice programs' both agree.

In order not to lengthen the definition of our models by tedious and routine repetitions, we skip those language constructs (in particular in Java$_{\mathcal{I}}$), which can easily be reduced to the core constructs dealt with explicitly in our models; examples are alternative control structures (like `for`, `do`, `switch`), pre- and postfix operators (`++`,`--`), conditional operators (`&&`, `||`), assignments combined with operations (`+=`, `-=`, etc.), variable initialization and similar expressive sugar. And since most of the object-oriented concepts of Java$_{\mathcal{O}}$ apply equally well to arrays and strings, we do not treat them either.

We do *not* consider Java packages, compilation units, and the visibility of names. We abstract from these aspects because they do not influence the dynamic semantics. We do not deal with input/output questions. We also do not consider the loading and linking of classes nor garbage collection. This is in accordance with the usual understanding of the dynamic semantics of programming languages. Yet, in Java these aspects are semantically visible: Dynamic loading and linking might raise exceptions, and in the presence of `finalize` methods also garbage collection is semantically visible. We plan to include these interesting aspects in later stages of the project.

2 The Imperative Core of Java

In this section we define the basic model Java$_{\mathcal{I}}$, which defines the semantics of the sequential imperative core of Java with statements (appearing in Java's method bodies) and expressions (appearing in statements) over Java's primitive types.

2.1 Signature

For each of our models we start with an arbitrary but fixed Java program. We separate standard compile-time matters from run-time issues by assuming that the program is given in a form in which it appears after parsing and elaboration, namely as an annotated abstract syntax tree. In this way we can abstract from

Fig. 1 Abstract Java Syntax for Java$_{\mathcal{I}}$

$$
\begin{aligned}
\mathit{Exp} &::= \mathit{Lit} \mid \mathit{Uop}\ \mathit{Exp} \mid \mathit{Exp}\ \mathit{Bop}\ \mathit{Exp} \mid \mathit{Var} \mid \mathit{Var} = \mathit{Exp} \mid \mathit{Exp}?\ \mathit{Exp}:\ \mathit{Exp}:\\
\mathit{Stm} &::= ; \mid \mathit{Exp}; \mid \mathit{Lab} : \mathit{Stm} \mid \texttt{break}\ \mathit{Lab}; \mid \texttt{continue}\ \mathit{Lab};\\
&\quad \mid \texttt{if}\ (\mathit{Exp})\ \mathit{Stm}\ \texttt{else}\ \mathit{Stm} \mid \texttt{while}\ (\mathit{Exp})\ \mathit{Stm} \mid \mathit{Block}\\
\mathit{Block} &::= \{\mathit{Type}\ \ \mathit{Var}; \ldots \mathit{Type}\ \ \mathit{Var}; \mathit{Stm} \ldots \mathit{Stm}\}\\
\mathit{Phrase} &::= \mathit{Exp} \mid \mathit{Stm} \mid \texttt{finished}
\end{aligned}
$$

the peculiarities of Java's concrete syntax and rely upon a series of useful syntactical simplifications which will be mentioned as we proceed in building our models.

The abstract syntax of Java's imperative core is defined in Fig. 1. It can also be viewed as defining corresponding domains (also called universes) of Java$_{\mathcal{I}}$. Although in our ASM's we will extend some of these domains by a small number of auxiliary constructs which do not appear in Java's syntax, we use the names of Java's grammatical constructs also as names for the corresponding (extended) ASM universes. We are sure the reader will be thankful for the simplified notation provided by this naming convention. Usually we denote domains by words beginning with a capital letter and write *dom* for elements of *Dom*, i.e. assuming without further mentioning that $dom \in Dom$. Figure 1 uses some additional universes, which represent basic syntactic constructs of Java, namely:

$$\mathit{Lit}, \mathit{Bop}, \mathit{Uop}, \mathit{Type}, \mathit{Var}, \mathit{Lab}$$

for Java's literals (except strings), Java's binary operators (except assignment and not including conditional operators), Java's unary operators (except primitive cast and pre- and postfix operators, but including all primitive widening and narrowing conversions), Java's primitive types, local variables, and labels, respectively.

As a result of the parsing and elaboration of the given Java program, no variable is declared twice, i.e., there are no hidden variables; all conversions are made explicit by applying the corresponding unary conversion operator; local variable initializers are syntactically reduced to ordinary assignments, following the variable declarations, which are all shifted to the beginning of blocks.

To separate as much as possible the dynamic (run-time) aspects from the static aspects the compiler (parsing and elaboration) can take care of, we use the idea (which is taken from the work on Occam [6]) to view program execution by a thread as a walk of the thread through the program's annotated abstract syntax tree: at each node the corresponding task is executed and then the control flow proceeds to the next task. The reader should keep in mind that the nodes represent *occurrences* of program constructs (phrases) and that all the functions we are going to define are defined on such occurrences of program constructs.

The following dynamic function *task*, an abstract program counter, always points to the current phrase to execute.

$$task : Phrase$$

The abstract program counter *task* must be updated according to Java's control flow. For sequential Java the *control flow* is fixed. As a matter of fact, for any statement and expression the LRM defines which substatement or subexpression to evaluate first and which expression or statement—depending on the context—to evaluate next (if any). This is captured by the following two static functions *fst* and *nxt*, which yield `finished`, if there is no first or next phrase to execute. (The definition of these functions, belonging to the compiler, will be given below by a recursion on abstract syntax trees.)

$$fst, nxt : Phrase \to Phrase$$

Proceeding from one task to the next task in accordance to Java's (unconditional) control flow is thus reduced to the following macro:

$$\text{proceed} \equiv task := nxt(task)$$

Statement execution and side-effects of expression evaluation typically update the local environment, formalized (for $\text{Java}_{\mathcal{I}}$) using a dynamic function

$$loc : Var \to Value$$

which captures the association between local variables and their values (bound to the given method activation). The universe *Value*, defined by

$$Value ::= Bool \mid Integers \mid Floats$$

contains Java's primitive values: booleans, integers in specific ranges, and floating point numbers according to IEEE 754. For simplicity, we identify Java's booleans with the corresponding ASM values *true* and *false*, and abbreviate in our formulae often $bool = true$ to $bool$.

The storage of *intermediate values* of expressions, which are computed to be used as arguments or operands in larger expressions or to affect the conditional control flow among statements, is formalized using a dynamic function on the subset *Exp* of *Phrase*:

$$val : Exp \to Value$$

This concludes the definition of the signature of $\text{Java}_{\mathcal{I}}$. Minor additions pertaining only to some special constructs will be presented in the corresponding sections.

2.2 Transition Rules

Transition rules describe how the states of Java$_\mathcal{I}$, here its dynamic functions *task*, *loc*, and *val*, change over time by evaluation of expressions and execution of statements.

The initial state of Java$_\mathcal{I}$ is defined by the given *phrase*, which defines also the static functions *fst* and *nxt*. In particular, we assume that $nxt(phrase) =$ `finished`. *task* points to the first phrase to be executed, formally $task = fst(phrase)$. The functions *loc* and *val* are everywhere undefined.

The *run terminates*, if no rule can be executed, because the preconditions of all rules evaluate to false. If the execution completes normally, i.e., without any run-time violation, the ASM reaches: $task =$ `finished`.

Expressions

The expressions of Java's imperative core—except the conditional operators—are evaluated from left to right and from innermost to outermost. This is described in Chap. 15.6 of Java's LRM [19]. (In the remainder of this chapter, we will abbreviate citations like this one using the '§' sign, writing (§ 15.6) to cite the corresponding chapter of the LRM. This should help the reader to check the correctness of our ASM formalization by comparing it with the LRM.)

We capture this 'postfix' evaluation order as follows: When the expression *exp* is going to be executed, we start with *task* pointing to $fst(exp)$, then we repeat applying *nxt* on *task* until *task* points to *exp*. During this process, we evaluate any expression to which *task* points and assign the computed expression value to the task using *val*.

This evaluation order is reflected in the recursive definition of the functions *fst* and *nxt*: If the expression *exp* does not have any subexpression, we set $fst(exp) = exp$. Otherwise let *exp* have the form $f(exp_1, \ldots, exp_n)$, where f denotes any n-ary expression constructor, but not the conditional expression. We set $fst(exp) = fst(exp_1)$, $nxt(exp_i) = fst(exp_{i+1})$, $0 < i < n$, and $nxt(exp_n) = exp$. For the special case of conditional expressions, see below.

This establishes the required control flow. It remains to specify the normal evaluation of expressions (in Java$_\mathcal{I}$ constants and operator terms, variables and conditional expressions) by transition rules.

Evaluating constants and operators. The value $\widetilde{lit}$ of every occurrence *lit* of a *literal* is as defined in § 15.7.1. Its dynamic semantics is defined by the following transition rule:

if *task* is *lit* **then** (Literal)
  $val(task) := \widetilde{lit}$
  proceed

The macro 'is', defined below, tests whether *task* points to *lit*. The macro will be refined for Java$_\mathcal{T}$.

$$task \text{ is } phrase \equiv task = phrase$$

The reader should keep in mind that by our typing convention for elements of universes, *task* is *lit* stands for $task = lit \wedge lit \in Lit$. *lit* stands for a variable, which has to be instantiated by an element of *Lit*. Similarly for all rules below.

The value of a *unary expression* with operation symbol $\odot$ is defined by applying the corresponding semantic operation $\widetilde{\odot}$—a static function for $\text{Java}_{\mathcal{I}}$, which is defined in the LRM—on the result of the operand (§ 15.13, 15.14)

if $task$ is $(\odot exp)$ **then** (UnaryExp)
  $val(task) := \widetilde{\odot}(val(exp))$
  proceed

The value of a *binary expression* with operation symbol $\otimes$ is defined by applying the corresponding semantic operation $\widetilde{\otimes}$ on the results of both operands. The rule cannot be executed, if the binary operator is an integer division or remainder operation (denoted by / and %), and the value of the second operand is 0 (§ 15.13, 15.14). (See Sect. 5 BinaryExp for the exception case.) The transition rule is defined by:

if $task$ is $(exp_1 \otimes exp_2) \wedge (\otimes \in \{/, \%\} \Rightarrow val(exp_2) \neq 0)$ **then** (BinaryExp)
  $val(task) := val(exp_1) \widetilde{\otimes} val(exp_2)$
  proceed

Using variables. The value of a *variable* is the value bound under the name of the variable in the local environment (§ 15.13.1).

if $task$ is var **then** (VarAcc)
  $val(task) := loc(var)$
  proceed

The value of a *simple assignment expression* is the value of its right hand side. The execution of the assignment operator replaces the existing value bound under the name of the variable of the left hand side in the local environment by the result of the right hand side (§ 15.25.1).

if $task$ is $(var = exp)$ **then** (VarAss)
  $loc(var) := val(exp)$
  $val(task) := val(exp)$
  proceed

Evaluating conditional expressions. To determine the value of a *conditional expression* requires two steps. The condition is evaluated first: if its value is *true*, the value of the conditional expression is the value of the second expression, otherwise it is the value of the third expression (§ 15.22–24).

Processing a phrase in several steps means that we have to associate several rules with the same phrase, which are executed at different times. In order to distinguish the rules syntactically, we associate the rules not only with the single

Fig. 2 Normal control flow of a conditional expression / block

$$\begin{array}{ll}
\textbf{let } exp = exp_1 \,?\, exp_2 : exp_3 : \textbf{in} & \textbf{let } stm = \{type\ var; \ldots ; stm_1 \ldots stm_n\} \textbf{ in} \\
\quad fst(exp) \;\; = fst(exp_1) & \quad fst(stm) \;\; = fst(stm_1) \\
\quad nxt(exp_1) = exp & \quad nxt(stm_i) \;\; = fst(stm_{i+1}), 0 < i < n \\
\quad nxt(exp_i) = exp_i :, i \in \{2, 3\} & \quad nxt(stm_n) = nxt(stm)
\end{array}$$

phrase but with (some of) its subphrases, too. Here, we associate a rule with the conditional expression and one with the auxiliary subphrases of form $(exp:)$. The static control flow is defined in Fig. 2.

The rule for the condition triggers the evaluation of the second or third expression.

$$\begin{array}{ll}
\textbf{if } task \text{ is } exp_1 ? exp_2 : exp_3 : \textbf{then} & \text{(IfExp)} \\
\quad \textbf{if } val(exp_1) \textbf{ then } task := fst(exp_2) & \\
\qquad\qquad\quad \textbf{else } task := fst(exp_3) &
\end{array}$$

The rule for the second and third expression assigns the value of the evaluated subexpression to the immediately enclosing conditional expression and proceeds.

$$\begin{array}{ll}
\textbf{if } task \text{ is } exp : \textbf{then} & \text{(ThenElseExp)} \\
\quad val(if(task)) := val(exp) & \\
\quad task \qquad\quad := nxt(if(task)) &
\end{array}$$

The auxiliary static function *if* always points from a then or else (sub-) expression to its father, namely the conditional expression.

Statements

The sequence of execution of a Java program is controlled by statements. We distinguish in Java$_{\mathcal{I}}$ three statement kinds: those which transfer control unconditionally, those which transfer control conditionally, and those which transfer control abruptly. The latter are described in Sect. 2.3.

Unconditional transfer of control. Statements whose *only* effect is to transfer control unconditionally do not have transition rules – their effect is already precompiled into the functions *fst* and *nxt*.

A *block* is executed by executing each of the statements in order from first to last (§ 14.2), see Fig. 2. We abstract from executing variable declarations because in our ASM the assignment of a value to a variable implicitly enlarges the domain of the environment, provided the variable is not already in the environment's domain. Variable access is always defined, since the elaboration phase assures—due to the rules of definite assignment (§ 16)—that every local variable is assigned before it is used. We can also abstract from deleting variables from the environment, because we know that in the annotated syntax tree no variable is declared twice.

Fig. 3 Normal control flow of an empty statement / an expression statement / a labeled statement

let $stm = ;$ **in**	**let** $stm = exp;$ **in**	**let** $stm = lab : stm_1$ **in**
$fst(stm) = nxt(stm)$	$fst(stm) = fst(exp)$	$fst(stm) = fst(stm_1)$
	$nxt(exp) = nxt(stm)$	$nxt(stm_1) = nxt(stm)$

Fig. 4 Normal control flow of `if`/ `while`

let $stm =$ `if` (exp) stm_1 `else` stm_2 **in**	**let** $stm =$ `while` $(exp) stm_1$ **in**
$fst(stm) = fst(exp)$	$fst(stm) = fst(exp)$
$nxt(exp) = stm$	$nxt(exp) = stm$
$nxt(stm_i) = nxt(stm), i \in \{1, 2\}$	$nxt(stm_1) = fst(exp)$

An *empty statement* does nothing (§ 14.5). The control flow simply skips the empty statement. Figure 3 shows its control flow.

An *expression statement* is executed by evaluating the expression (§ 14.7). Figure 3 shows its control flow. (In the JVM an additional action is taken to discard the value because it is not needed furthermore.)

A *labeled statement* is executed by executing the immediately contained statement (§ 14.6), see Fig. 3.

Conditional transfer of control. An `if-else` *statement* is executed by first evaluating the expression. Execution continues by making a choice based on the resulting value. If the value is *true*, the first contained statement is executed, otherwise the second contained statement is executed (§ 14.8). Figure 4 captures the static aspect of the control flow. The dynamic aspect is described by the following transition rule:

if $task$ is `if` (exp) stm_1 `else` stm_2 **then** (IfStm)
  if $val(exp)$ **then** $task := fst(stm_1)$
  else $task := fst(stm_2)$

A `while` *statement* is executed by first evaluating the expression. Execution continues by making a choice on the resulting value. If the value is *true*, then the contained statement is executed, otherwise no further action is taken (§ 14.10), see Fig. 4 for the definition of *fst* and *nxt*.

if $task$ is `while` (exp) stm **then** (While)
  if $val(exp)$ **then** $task := fst(stm)$
  else $task := nxt(task)$

2.3 Abrupt Transfer of Control

The preceding subsection describes the normal control flow, in which certain steps of computations are carried out. Normal control flow can be abrupted in Java$_{\mathcal{I}}$ by the restricted jump statements `break` and `continue`. Upon switch from normal to abrupted mode, the execution of one or more phrases may be terminated before all steps of their normal mode of execution have completed, e.g., only part of the statements of a block are executed because a `break` or `continue` was encountered which transfers control out of the block. Phrases which do not terminate normally are said to *complete abruptly*. In the later extensions *abrupt completion* can also be due to return from procedure execution (in Java$_{\mathcal{C}}$) or to raising and handling of exceptions (in Java$_{\mathcal{E}}$).

Signature. Any abrupt completion always has an associated reason, which in Java$_{\mathcal{I}}$ is a `break` or `continue` with a given label. For a uniform formulation of these interrupts (which supports the process oriented view of interrupt description in the LRM) we introduce a universe

$$Reason ::= Break(Lab) \mid Continue(Lab)$$

(to be extended in Java$_{\mathcal{C}}$ and Java$_{\mathcal{E}}$) together with a dynamic function

$$mode : Reason$$

which records whether the current mode is normal (*mode* is undefined) or abrupted, due to a *Break* or a *Continue* with a specific label.

When execution is abrupted control transfers up the grammatical nesting level. To formalize this we use an auxiliary function

$$up : Phrase \to Phrase$$

which applied to a phrase returns the next *enclosing* phrase for a given phrase, which might handle the reason for abruption. (We say that phrase A is enclosed by phrase B, if B contains A.)

In Java$_{\mathcal{I}}$ *up* points only to labeled statements. Formally, let c be any n-ary phrase constructor but not a labeled statement, i.e. $phrase = c(phrase_1, \ldots, phrase_n)$, then we set $up(phrase_i) = up(phrase), 1 \leq i \leq n$. If *phrase* denotes a labeled statement, i.e. $phrase = lab : stm_1$, we set $up(stm_1) = phrase$. And for the given program *up* returns `finished`. (We shall extend this definition for `static`, `try-catch`, `try-finally` and `synchronized` clauses, see Sects. 3, 5 and 6).

The function *up* is used to update the dynamic function *task*, formalized by the macro:

$$\text{abrupt} \equiv task := up(task).$$

The function *up* supports the uniform treatment of different forms of interrupts: in each of them the control flow is transfered up the grammatical nesting level and then up the method invocation stack. Whereas for `break` and `continue`

the control remains within the given method, for `return` the control leaves the current method (see Sect. 3) and for exceptions the control can climb up the whole method invocation stack (see Sect. 5).

Transition rules. In Java$_{\mathcal{I}}$ a `break` *statement* or a `continue` *statement* (which can be assumed to appear in the annotated syntax tree with label *lab*) transfers control to the (innermost) enclosing labeled statement having *lab* as its label—Java's context conditions guarantee the existence of such a labeled statement. This labeled statement, which is called the target, and all statements which are passed during this transfer complete abruptly, the reason being a *Break* or *Continue* with label *lab*.

if *task* is `jump` *lab*; **then** (Jump)
  mode := *Jump*(*lab*)
  abrupt
for (`jump`, *Jump*) ∈ {(`break`, *Break*), (`continue`, *Continue*)}

If *task* points to a *labeled statement* (due to the definitions of *fst* and *nxt* this can happen only in abrupted mode) it is checked whether the label of the reason agrees with the label of the statement. In case it does, and the reason is a *Break*, execution proceeds normally at the next phrase of the target, in case of a *Continue* the next iteration of the embedded statement is executed, which for `while` statements is the first phrase of the target (§ 14.13,14.14,14.10). (If we would include `do` and `for` statements the following rule has to be refined.)

if *task* is *lab* : *stm* **then** (LabStm)
  if *mode* = *Jump*(*lab*) **then**
    mode := *undef*
    task := *jump*
  else abrupt
for (*Jump*, *jump*) ∈ {(*Break*, *nxt*(*task*)), (*Continue*, *fst*(*stm*))}

Jump transfers control immediately to the nearest enclosing matching labeled statement. We could have included this into the definition of *nxt*, avoiding Jump and LabStm. We prefer to formulate a rule here in order to smoothen the refinement in the context of exception handling and multithreading where control may not directly proceed at the first or next phrase of the target but has to execute certain clauses first, see Sects. 5 and 6.

3 Adding Classes

Java$_{\mathcal{C}}$ enhances Java$_{\mathcal{I}}$ for an object-based language. Java$_{\mathcal{C}}$ includes class fields, class methods and class initializers. (These entities are also known as static fields, static methods and static initializers, respectively.) Java$_{\mathcal{C}}$ also supports a limited form of interfaces, namely its static fields. Conceptually Java$_{\mathcal{C}}$ describes the semantics of an imperative language supporting modules. Modula2 is a typical representative of such a language: Class fields are the module's global variables, class methods are the module's procedures and class initializers correspond to module initializers.

Fig. 5 Abstract Java Syntax for Java$_{\mathcal{C}}$

$$
\begin{aligned}
\mathit{Exp} &::= \ldots \mid \mathit{FieldSpec} \mid \mathit{FieldSpec} = \mathit{Exp} \mid \mathit{MethodSpec}(\mathit{Exp}, \ldots, \mathit{Exp}) \\
\mathit{Stm} &::= \ldots \mid \texttt{return}\ \mathit{Exp}; \mid \texttt{return}; \\
\mathit{Phrase} &::= \ldots \mid \mathit{Init} \\
\mathit{Init} &::= \texttt{static}\ \mathit{Block}\ \texttt{endstatic}
\end{aligned}
$$

3.1 Signature

Figure 5 shows Java$_{\mathcal{C}}$'s abstract syntax, where '...' stands for the constructs of Java$_{\mathcal{I}}$. The abstract syntax uses the universes *FieldSpec* and *MethodSpec*, which denote class fields and overloaded class methods, respectively.

$$
\begin{aligned}
\mathit{FieldSpec} &= \mathit{Class} \times \mathit{Field} \\
\mathit{MethodSpec} &= \mathit{Class} \times \mathit{Method} \times \mathit{Functionality} \\
\mathit{Functionality} &= \mathit{Type}^* \times (\mathit{Type} \mid \{\texttt{void}\})
\end{aligned}
$$

These type abbreviations use Java$_{\mathcal{C}}$'s new abstract universes

$$\mathit{Class}, \mathit{Field}, \mathit{Method}$$

denoting the given program's fully qualified classes *and* interfaces, as well as field and method identifiers, respectively. Furthermore, we extend the universe *Var* of Java$_{\mathcal{I}}$ to denote local variables and method parameters.

Unless otherwise stated we will not distinguish classes and interfaces. So whenever we speak of classes, class fields or class initializers, this normally includes interfaces, their fields and initializers, respectively.

For the sake of simplicity, but without loss of generality, we assume that due to the elaboration phase the annotated syntax trees in Java$_{\mathcal{C}}$ have the following properties: Method specifications denote the most specific method chosen at compile-time (§ 15.11.2.2). Every execution path of any method body ends with a `return` statement. Any class has a class initializer—its body (whose function is to initialize the class fields at the first active use of the class, see below) may be empty. Non constant class field initializations are syntactically reduced to assignments and are placed at the beginning of a class initializer. Java$_{\mathcal{C}}$ abstracts from initializations of constant fields; the latter are `final` class fields, whose values are compile-time constants (§ 15.27). The value of constant fields is precomputed (as part of the elaboration phase) and stored in the program's class and interface environment.

Java$_{\mathcal{C}}$ programs are executed w.r.t. a static *class environment*, which is set up during parsing and elaboration. Each class (not interface) declaration consists of the superclass of the class, of its implemented interfaces, its class fields, class methods, and its initializer. Due to the fact that interfaces do not contain code,

an interface declaration consists only of its superinterfaces, its static fields and its static initializer.

The following static functions look up information in the environment and either access the environment directly or traverse the inheritance hierarchy from bottom to top (subtype to supertype):

$$\begin{aligned} \mathit{super} &: \mathit{Class} \to \mathit{Class} \\ \mathit{supers} &: \mathit{Class} \to \mathcal{P}\,\mathit{Class} \\ \mathit{interfaces} &: \mathit{Class} \to \mathcal{P}\,\mathit{Class} \\ \mathit{classInit} &: \mathit{Class} \to \mathit{Init} \\ \mathit{classFields} &: \mathit{Class} \to \mathcal{P}\,\mathit{Field} \\ \mathit{classFieldValue} &: \mathit{FieldSpec} \to \mathit{Value} \\ \mathit{classMethod} &: \mathit{MethodSpec} \to \mathit{Var}^* \times \mathit{Block} \end{aligned}$$

The function *super* returns the direct superclass of the specified class, provided the class has a superclass. The function *supers* calculates the transitive closure of the direct superclass relationship. The function *interfaces* returns the direct interfaces of a class. The function *classInit* returns the body of a class initializer of the given class. The function *classFields* returns the set of all fields declared by the class (which have to be initialized exactly once for the whole class, just before their first use, as specified in the LRM). The function *classFieldValue* maps non-constant class fields to their default values, and constant class fields to the values of their respective compile-time constant expressions. The function *classMethod* looks up the method in the specified class.

In Java$_\mathcal{C}$ we distinguish three initialization states for a class: either the initialization of the class has not yet started, it is *InProgress* or it is already *Done*, so that we introduce a universe

$$\mathit{InitState} ::= \mathit{InProgress} \mid \mathit{Done}$$

together with a dynamic function

$$\mathit{init} : \mathit{Class} \to \mathit{InitState}$$

which records the current initialization status of a class. A class is 'initialized', if initialization for the class is *InProgress* or *Done*. (If the initialization of *class* has not yet started, $\mathit{init}(\mathit{class})$ is undefined.)

$$\text{initialized}(\mathit{class}) \equiv \mathit{init}(\mathit{class}) \in \{\mathit{InProgress}, \mathit{Done}\}$$

To model the dynamic state of class fields, we have to reserve storage for all these variables. The dynamic function *glo* returns the value stored under a field specification.

$$\mathit{glo} : \mathit{FieldSpec} \to \mathit{Value}$$

The introduction of methods has a fundamental effect on the definition of Java$_\mathcal{I}$'s dynamic objects. During execution of a Java$_\mathcal{I}$ (method) body the local

environment and the value of an expression is computed and both may then be used in future computation steps (of that method). However, if in Java$_\mathcal{C}$ a method invokes itself recursively, the same statements and expressions are executed several times, thus 'overwriting' the local environment as well as values of expressions. To handle this properly, we refine the two functions *loc* and *val* by indexing the value of any expression or (local) variable in the environment with the calling depth of the code in execution. This corresponds to the usual technique of introducing stacks, to handle recursive method activation. Likewise, we introduce a task stack, whose topmost element always points to the code in execution. When a method is invoked, we push the first phrase of the invoked method onto the stack to resume the task of the invoker after the invoked method has finished and its task is popped from the stack. The signatures of the modified dynamic functions of Java$_\mathcal{I}$ are extended to finite sequences:

$$\begin{aligned} task_\mathcal{O} &: Phrase^* \\ val_\mathcal{O} &: (Exp \to Value)^* \\ loc_\mathcal{O} &: (Var \to Value)^* \end{aligned}$$

During execution of a single method body in Java$_\mathcal{I}$ the calling depth does not change. Hence, *task*, *val* and *loc* of Java$_\mathcal{I}$ are guaranteed to be the topmost elements of the corresponding dynamic functions in Java$_\mathcal{C}$.

$$\begin{aligned} task &\equiv top(task_\mathcal{O}) \\ val &\equiv top(val_\mathcal{O}) \\ loc &\equiv top(loc_\mathcal{O}) \end{aligned}$$

Via this refinement Java$_\mathcal{C}$ can be shown to be a conservative extension of Java$_\mathcal{I}$ (see the conclusion) so that all propositions which are valid for Java$_\mathcal{I}$ carry over mutatis mutandis to Java$_\mathcal{C}$.

To simplify notation in connection with method invocation and return we introduce frames, denoting triples consisting of the stacks $task_\mathcal{O}$, $val_\mathcal{O}$ and $loc_\mathcal{O}$:

$$\text{frames} \equiv (task_\mathcal{O}, val_\mathcal{O}, loc_\mathcal{O})$$

Finally, we extend the universe *Reason*, since `return` statements always complete the method's body abruptly. We introduce the new reasons *Return* and *Result*, the latter carrying a specific *Value* which (eventually, see Sect. 5) becomes the value of the method invocation.

$$Reason ::= \ldots \mid Return \mid Result(Value)$$

3.2 Transition Rules

Via the refinement of *task*, *loc* and *val* in Java$_\mathcal{C}$, each Java$_\mathcal{I}$-rule becomes a rule of Java$_\mathcal{C}$. Therefore it only remains to give here the rules for the evaluation of the new Java$_\mathcal{C}$-expressions and for the execution of the new Java$_\mathcal{C}$-statements.

The initial state of Java$_\mathcal{C}$ is as follows: The environment, modeled by the respective lookup functions and predicates, is defined by the given program, which consists of a list of classes and interfaces. For any method body and class

initializer, the functions *fst*, *nxt* and *up* are defined as given in Sect. 2—only `return` and `static` need new definitions. All class fields of all classes are set to their default or constant values and no class is initialized.

The run of Java$_{\mathcal{C}}$ starts by invoking the class method `main` : *MethodSpec* (with an empty parameter list) being part of the environment. *task* denotes the first phrase of `main`'s *body*, *loc* and *val* are undefined. Formally: frames = start(*fst*(*body*)), where the macro 'start' is used to initialize the task, temporary and locals stacks, respectively. (The expression $\langle \ldots \rangle$ denotes finite sequences.)

$$\text{start}(\mathit{phrase}) \equiv (\langle \mathit{phrase} \rangle, \langle \emptyset \rangle, \langle \emptyset \rangle)$$

For the sake of exposition let us first assume that all classes have been initialized. When and how classes are initialized is explained in Sect. 3.3.

The run terminates, if no rule of Java$_{\mathcal{C}}$ can be executed any more. If the execution completes without any run-time violation, the ASM reaches: $\mathit{task}_{\mathcal{O}} = \langle \texttt{finished} \rangle$; this means that `main`'s body has been executed successfully.

Fields. The value of a *class field access* is the value bound under the name of the class and of the field in the global environment (§ 15.10).

if *task* is (*class*, *field*) ∧ initialized(*class*) **then** (CFieldAcc)
  val(*task*) := *glo*(*class*, *field*)
  proceed

The value of a *class field assignment* is the value of its right-hand side (§ 15.25). The execution of the assignment replaces the existing value bound under the class and field's name in the global environment by the result of the right hand side.

if *task* is ((*class*, *field*) = *exp*) ∧ initialized(*class*) **then** (CFieldAss)
  glo(*class*, *field*) := *val*(*exp*)
  val(*task*) := *val*(*exp*)
  proceed

Methods. A *class method invocation* is used to call a class method (§ 15.11). The value of a method invocation is the return value of the invoked method—this is specified in Result and Return. Through method invocation new bindings are created in the environment, containing the bindings of the actual argument values to the methods parameters. The execution begins at the first phrase of the invoked method's body. Formal arguments and the method's body are looked up in the environment.

if *task* is ((*class*, *method*, *fcty*)(*exp*$_1$, ..., *exp*$_n$))∧ (CMethod)
  initialized(*class*) **then**
  frames := invoke($\langle$*val*(*exp*$_1$), ..., *val*(*exp*$_n$)$\rangle$, *args*, *fst*(*body*), frames)
  where (*args*, *body*) = *classMethod*(*class*, *method*, *fcty*)

The macro 'invoke', defined below, pushes the new phrase to be executed on the method call stack, an everywhere undefined function on the temporary stack

Fig. 6 Normal control flow of `return` / `throw`

and the new bindings on the environment stack, respectively. (Concatenation of sequences is denoted by juxtaposition, for example $\langle 1\rangle\ \langle 2\rangle = \langle 1, 2\rangle$).

$$\begin{aligned}&\text{invoke}(\langle val_1, \ldots, val_n\rangle, \langle var_1, \ldots, var_n\rangle, phrase, (tasks, vals, locs)) \equiv\\ &\quad(\langle phrase\rangle\ tasks, \langle\emptyset\rangle\ vals, \langle\{(var_1, val_1), \ldots, (var_n, val_n)\}\rangle\ locs)\end{aligned}$$

Methods return when a `return` statement in their respective bodies is encountered.

A `return` *expression statement* is executed by first evaluating its subexpression. (The control flow aspect of this execution is shown in Fig. 6.) If the expression evaluation completes normally, the `return` statement abrupts processing the method's body, and attempts to transfer control and the value of the expression to the invoker of the method. This is achieved in several steps: First, the `return` statement completes abruptly, the reason being a *Result* with the value of the subexpression.

if *task* is `return` *exp*; **then** (Result)
  mode := *Result*(*val*(*exp*))
  abrupt

If there are still enclosing statements, they too have to be abrupted. For Java$_\mathcal{C}$ the only enclosing statement kind pointed to by *up* is a labeled statement and it is easy to see that (by execution of LabStm) the labeled statement abrupts, since the reason for abruption is a *Result*.

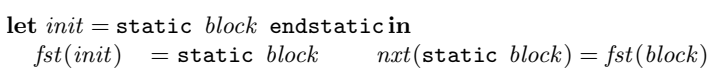
If there is no enclosing statement any more, i.e. when *task* becomes `finished`, execution of the `return` statement transfers the value of the expression to the invoker (provided there is still one), deletes the topmost bindings, and continues processing normally at the invoker's next phrase (§ 14.15). In case there is no invoker anymore—`main` has terminated—the ASM run finishes.

if *task* is `finished` $\wedge$ *mode* = *Result*(*res*) $\wedge$ *length*(*task*$_\mathcal{O}$) > 1 **then** (Result')
  mode := *undef*
  frames := result(*res*, frames)

The modification of the task, temporary and local environment stack is summarized in the macro 'result', see below.

A `return` *statement* without an expression has the same semantics as the `return` expression statement, except that no expression needs to be evaluated and consequently no value needs to be transfered from the invoked method to the invoker (§ 14.15).

Fig. 7 Normal control flow of a class initializer



if *task* is `return;` **then** (Return)
mode := *Return*
abrupt

if *task* is `finished` $\wedge$ *mode* = *Return* $\wedge$ *length*($task_{\mathcal{O}}$) > 1 **then** (Return')
mode := *undef*
frames := return(frames)

Result and Return use macros to determine the next phrase to execute, to transfer the expression's value to the invoker (only 'result'), and to delete the topmost bindings.

$$\begin{aligned}&\text{result}(res, (\langle \_, inv\rangle\ tasks, \langle \_, val\rangle\ vals, \langle \_\rangle\ locs)) \equiv\\ &\qquad (\langle nxt(inv)\rangle\ tasks, \langle val \oplus \{(inv, res)\}\rangle\ vals, locs)\\ &\text{return}(\langle \_, inv\rangle\ tasks, \langle \_\rangle\ vals, \langle \_\rangle\ locs) \equiv\\ &\qquad (\langle nxt(inv)\rangle\ tasks, vals, locs)\end{aligned}$$

The anonymous variable '_' stands for values that don't care. 'result' uses the operator $\oplus$ to override the invoker's *val* function at the phrase of the invocation. (An expression $f \oplus \{(k, v)\}$ is still a function, which returns the same values as f everywhere except at the argument k, where it returns the value v.)

3.3 Initialization

Execution starts in a state in which no class is initialized. A class or interface will be initialized at its *first active use* by executing its static initializer. Before a class is initialized its superclasses must be initialized. The superinterfaces of an interface *need not* be initialized before the interface is initialized (see below). This leads to three rules for class initialization: FirstActiveUse invokes the class initializer at the first active use of a class, Static starts the execution of the class initializer code or invokes the initialization of the direct superclass, Endstatic terminates a static initialization block. (The control flow is shown in Fig. 7.)

The first active use of a class or interface T can occur if a class method declared in T is invoked, or if a (non-constant) static field declared in T is used or assigned. (Sect. 4 defines the case if an instance for T is created.) The rule

if (*task* is (*class*, *field*) $\vee$ *task* is (*class*, *field*) = *exp* $\vee$ (FirstActiveUse)
task is ((*class*, *method*, *fcty*)($exp_1, \ldots, exp_n$))) $\wedge$
$\neg$initialized(*class*) **then**
frames := initialize(*class*, frames)

uses the macro 'initialize', which abbreviates (method) invocation:

$$\text{initialize}(\mathit{class}, \mathit{frames}) \equiv \text{invoke}(\langle\rangle, \langle\rangle, \mathit{fst}(\mathit{classInit}(\mathit{class})), \mathit{frames})$$

Note that by definition calling `main`, where `main` = ($\mathit{startClass}$, _, _), is a first active use of *startClass*. Thus we have to trigger the initialization of *startClass* before we process the first phrase of `main`'s *body*. We set frames = initialize($\mathit{startClass}$, start($\mathit{fst}(\mathit{body})$)).

The execution of a class initializer records that starting from now, initialization of this class is *InProgress*.

if *task* is `static` *block* **then** (Static)
  init(currClass) := *InProgress*
  enter

If the class represents (really) a class rather than an interface, the macro 'enter' invokes the class initialization of its direct superclass (if any), provided it is not initialized; otherwise execution enters the computation of the `static` block.

$$\begin{aligned}\text{enter} \equiv\ &\textbf{if } \mathit{supers}(\text{currClass}) \neq \emptyset \wedge \neg\text{initialized}(\mathit{super}(\text{currClass})) \textbf{ then}\\ &\quad \text{frames} := \text{initialize}(\mathit{super}(\text{currClass}), \text{frames})\\ &\textbf{else } \mathit{task} := \mathit{fst}(\mathit{block})\end{aligned}$$

The macro 'currClass' always returns the class which contains the given phrase using the static function $\mathit{classScope} : \mathit{Phrase} \rightarrow \mathit{Class}$.

$$\text{currClass} \equiv \mathit{classScope}(\mathit{task})$$

But what happens if the current class is actually an interface rather than a class? The LRM says that the superinterfaces may be initialized. However, if the initialization of any of the superinterfaces has a side-effect or fails (see Sect.5) this leads (at least on different machines) to *nondeterministic* behaviour *contradicting* Java's design goals. Therefore we restrained from modelling this nondeterminism and rather present the solution of Sun's Java Development Kit (JDK), which does not initialize superinterfaces.

After having executed the `static` phrase, its block is executed; except that `return` statements cannot be part of the block, there are no restrictions on the kinds of possible statements or expressions. Note however, that when a `static` block of class T is executed, accesses to T's fields and invocations of T's methods must be possible without triggering a new first use of T. This is the reason why we included *InProgress* in the predicate initialized.

When the phrase `endstatic` is executed, it records that the initialization is *Done*, and returns to the invoker.

if *task* is `endstatic` **then** (Endstatic)
  init(currClass) := *Done*
  exit

The used macro 'exit' distinguishes whether this initializer was called implicitly (by a first use) or explicitly (during processing of a `static` phrase). In case the execution of the class initializer was not triggered by a first use, processing has to resume at the next task of the invoker, otherwise it has to go back to the last task of the invoker.

$$\begin{aligned}\text{exit} \equiv \textbf{if } \mathit{invoker} = \texttt{static } \mathit{block} \textbf{ then}\\ \text{frames} := \text{return}(\text{frames})\\ \textbf{else } \text{frames} := \text{goBack}(\text{frames})\\ \textbf{where } \langle \_, \mathit{invoker} \rangle_{\_} = \mathit{task}_{\mathcal{O}}\end{aligned}$$

To pop the task, temporaries and local variables from their respective stacks, we use the macro

$$\text{goBack}(\langle \_ \rangle\ \mathit{tasks}, \langle \_ \rangle\ \mathit{vals}, \langle \_ \rangle\ \mathit{locs}) \equiv (\mathit{tasks}, \mathit{vals}, \mathit{locs})$$

On return from an implicitly invoked static initializer, *task* still points to the same phrase, but this time the class is initialized and processing can continue normally.

4 Adding Objects

Java$_{\mathcal{O}}$, the ASM for the object-oriented sublanguage of Java, extends the object-based Java$_{\mathcal{C}}$ for an object-oriented language. Java$_{\mathcal{O}}$ supports instance fields and instance methods, instance creation and method overriding, type casts and null pointers. In contrast to Java$_{\mathcal{C}}$ one may say that Java$_{\mathcal{O}}$ is object-oriented, since its defining equation "*module* = *type*"[24] holds.

4.1 Signature

The *abstract syntax* of the object-oriented sublanguage of Java is given in Fig. 8, where [`new` *Class*] denotes the optional appearence of the subphrase `new` *Class*. The abstract syntax makes use of constructor specifications, which denote class constructors (whose function is to initialize the instance fields during instance creation):

$$\mathit{ConstrSpec} = \mathit{Class} \times \mathit{Type}^*$$

Instance method invocations have an additional invocation *Kind*, which is used for method lookup.

$$\mathit{Kind} ::= \mathit{Virtual} \mid \mathit{Nonvirtual} \mid \mathit{Super}$$

Additionally, we extend the universe *Type* of Java$_{\mathcal{I}}$ in Java$_{\mathcal{O}}$ to include classes and interfaces and the type `null`.

As a result of the parsing and elaboration phase the following holds: Field access using `super` is reduced to "ordinary" field access of the superclass' field. Method invocations (including their invocation kinds) are attributed as specified in § 15.11.1–3. (The invocation kind *Static* is not needed here, since it is already handled by class methods. We do not introduce the invocation mode *Interface*, because it is semantically equivalent to *Virtual*.) Any constructor body—except the body of the constructor of the class `Object`—either begins with an explicit constructor invocation of another constructor in the same class, or with an explicit invocation of a superclass constructor (§ 12.5). Every execution path of

Fig. 8 Abstract Java Syntax for Java$_{\mathcal{O}}$

$$
\begin{aligned}
Exp ::= \ & \ldots \\
& \mid [\texttt{new}\ Class]ConstrSpec(Exp, \ldots, Exp) \\
& \mid \texttt{this} \mid Exp.FieldSpec \mid Exp.FieldSpec = Exp \\
& \mid Exp.MethodSpec\{Kind\}(Exp, \ldots, Exp) \\
& \mid Exp\ \texttt{instanceof}\ Class \mid (Class)Exp
\end{aligned}
$$

any constructor or instance method body ends with a `return` statement. Instance field initializers of class T are syntactically reduced to assignments. They are replicated in all constructors for class T, which call a superclass constructor. The assignments immediately follow the call of the superclass constructor, which guarantees that instance field initializers are evaluated only once per instance creation.

The following static functions look up compile-time information in the environment:

$$
\begin{aligned}
instFields &: Class \to \mathcal{P}\,FieldSpec \\
instFieldValue &: FieldSpec \to Value \\
instConstr &: ConstrSpec \to Var^* \times Block \\
instMethod &: MethodSpec \times Class \times Kind \to Var^* \times Block \\
compatible &: Class \times Class \to Bool
\end{aligned}
$$

The function *instFields* calculates the set of instance fields declared by the specified class and all of its superclasses (if any). The function *instFieldValue* maps fields to their default values (as specified in the LRM § 4.5.4). The function *instConstr* looks up the required constructor. The function *instMethod* starting at a particular class, returns the (first) method declaration for the given method specification. If the invocation mode is *Nonvirtual*, overriding is not allowed; the specified method in the explicitly given class is the one to be invoked. Otherwise the invocation mode is *Virtual* or *Super* and overriding may occur. If a method with the given method specification is not implemented in the given class, the superclass of that class is then recursively searched; whatever it comes up with is the result of the search (§ 15.11.4). The predicate $compatible(src, tar)$ returns true iff the reference type *src* is assignment compatible—defined according to LRM § 5.2—with reference type *tar*. For instance, if both *src* and *tar* are classes, *src* must be equal to *tar* or *src* must be a subclass of *tar*.

The values of reference types are references to instances of classes. References belong to the abstract dynamic universe

$$Reference.$$

We extend the universe *Value* in Java$_{\mathcal{O}}$ to include the values of Java's primitive types, references and the value *null*.

$$Value ::= \ldots \mid Reference \mid \{null\}$$

In Java$_{\mathcal{O}}$ any class has its `Class` object. Since we ignore linking and loading in this paper, the function *classRef*, which maps any class to its class object, is not dynamic but static.

$$classRef : Class \to Reference$$

To model the dynamic state of instances, we have to reserve storage for all instance variables and have to store to which class an object belongs. The function *classOf* returns the class of the object that is refered to by the reference. The dynamic function *dyn* returns the value stored under a field specification of an object.

$$classOf : Reference \to Class$$
$$dyn : Reference \times FieldSpec \to Value$$

4.2 Transition Rules

The initial state and the termination conditions of Java$_{\mathcal{O}}$ are taken from Java$_{\mathcal{C}}$. Additionally, we require that in Java$_{\mathcal{O}}$'s initial state *classOf* and *classRef* are inverses of each other. Java$_{\mathcal{O}}$ has all the rules of Java$_{\mathcal{C}}$ and in addition the rules below for the new object-oriented features.

Instance creation. In Java$_{\mathcal{O}}$ new class instances are explicitly created by evaluating a *class instance creation expression*. The value of this expression is a reference to the newly created object of the specified class type. The new object contains new instances of all the fields declared in the specified class and its superclasses (§ 15.8). The creation of a new instance proceeds as follows.

First, the term `new` *class* (which is the first subphrase of the instance creation expression) is evaluated. If it is the first active use of the class, the class initializer is invoked.

if *task* is `new` *class* $\wedge$ $\neg$initialized(*class*) **then** (IFirstActiveUse)
  frames := initialize(*class*, frames)

If the class is initialized, we generate a new instance:

if *task* is `new` *class* $\wedge$ initialized(*class*) **then** (NewInstance)
  newInstance(*class*,
    val(*task*) := *ref*)
  proceed

The macro 'newInstance' allocates a new reference, keeps track of its origin, sets each new field to its default value, and executes the updates.

newInstance(*class*, *updates*) ≡ **extend** *Reference* **by** *ref*
classOf(*ref*) := *class*
vary *f* **over** *instFields*(*class*)
dyn(*ref*, *f*) := *instFieldValue*(*f*)
updates

Then the argument expressions are evaluated (if any). When the arguments are evaluated the constructor is called. New bindings are created in the local environment: the formal arguments var_i are bound to the actual ones $val(exp_i)$; if the constructor is part of a new instance creation expression, `this` is bound to the newly generated reference, which is already assigned to *val*(*new*); otherwise—the constructor is called explicitly—the value of `this` can be looked up in the local environment.

if *task* is *new constrSpec*($exp_1, \ldots, exp_n$) **then** (Constr)
frames := invoke(⟨*this*⟩ *vals*, ⟨`this`⟩ *args*, *fst*(*body*), frames)
where (*args*, *body*) = *instConstr*(*constrSpec*)
this = **if** *new* = `new` *class* **then** *val*(*new*) **else** *loc*(`this`)
vals = ⟨*val*(exp_1), ..., *val*(exp_n)⟩

Fields and `this`. The value of a *field access expression* is the value of the field in the object pointed to by the target reference (§ 15.10), provided it is not *null*.

if *task* is (*exp*.*fieldSpec*) ∧ *val*(*exp*) ≠ *null* **then** (IFieldAcc)
val(*task*) := *dyn*(*val*(*exp*), *fieldSpec*)
proceed

The value of a *field assignment expression* is the value of its right-hand side (§ 15.25) – provided the target reference is defined. Execution of the assignment redefines the value of the field in the object pointed to by the target reference with the value of the right-hand side.

if *task* is (exp_1.*fieldSpec* = exp_2) ∧ *val*(exp_1) ≠ *null* **then** (IFieldAss)
dyn(*val*(exp_1), *fieldSpec*) := *val*(exp_2)
val(*task*) := *val*(exp_2)
proceed

The value of the keyword `this` is a reference of the object for which the instance method was invoked (§ 15.7.2). This reference is bound at each method invocation and can therefore be used at the current calling depth.

if *task* is `this` **then** (This)
val(*task*) := *loc*(`this`)
proceed

Methods. An *instance method invocation expression* is used to invoke an instance method (§ 15.11). The value of a method invocation expression is the return value of the invoked method—this is specified in Result. Provided the target reference is defined, we have to distinguish three cases to locate the invoked method: If the invocation mode is

- *Nonvirtual* it denotes a private method. The instance method is looked up statically, searching its declaration at the current class. (Context restrictions guarantee that the method specification's class and the explicitly given class agree.)
- *Virtual*, then the instance method is looked up dynamically, searching its current definition starting at the class of the object.
- *Super*, then the instance method is looked up dynamically, too. However, here the search for the method's definition starts at the immediate superclass of the current class.

Through method invocation new bindings are created in the environment, containing the bindings of the actual argument values to the methods parameters, and the target reference available as `this`. The execution begins at the first phrase of the invoked method's body.

if $task$ is $(exp.methodSpec\{kind\}(exp_1, \ldots, exp_n))\wedge$ (IMethod)
$val(exp) \neq null$ **then**
frames := invoke($\langle val(exp)\rangle$ $vals$, $\langle$`this`$\rangle$ $args$, $fst(body)$, frames)
where $(args,\ body) = instMethod(methodSpec, class, kind)$
$vals = \langle val(exp_1), \ldots, val(exp_n)\rangle$
$class$ = **case** $kind$ **of** $Nonvirtual$: currClass
$Virtual$: $classOf(val(exp))$
$Super$: $super$(currClass)

Dynamic typing. The value of an `instanceof` *expression* is *true*, if the value of its operand is not *null* and the reference is compatible with the required type. Otherwise the result is *false* (§ 15.19.2).

if $task$ is $(exp$ `instanceof` $class)$ **then** (Instanceof)
$val(task) := val(exp) \neq null\wedge$
$compatible(classOf(val(exp)), class)$
proceed

The value of a reference type *cast expression* is the value of its operand – provided it is compatible with the required class or interface type or it is *null* (§ 15.15).

if $task$ is $(class)exp\wedge$ (Cast)
$val(exp) = null \vee compatible(classOf(val(exp)), class)$ **then**
$val(task) := val(exp)$
proceed

4.3 Arrays and Strings

Most of the object-oriented concepts introduced in the previous subsection apply equally well to arrays and strings; so we do not extend Java$_\mathcal{O}$ but rather sketch necessary extensions.

Java's arrays are objects. Therefore, we can use the previously introduced function *classOf* to store the array's type and use the previously introduced *dyn*

function to also model the dynamic state of array components. However, since components are accessed by natural numbers, we have to refine *FieldSpec* to also include natural numbers. Every array also has an associated `Class` object, shared with all other arrays with the same component type. So we also have to refine the function *classRef* and the initialization of *classOf*. In the context of class initialization an *ambiguity* arises. Whereas the LRM specifies that in an array creation expression the array's element type—provided it is a class or interface—is initialized, this is left out in Java's Virtual Machine specification [23], although this is required as part of the resolution process. In fact, Sun's JDK triggers the initialization of the array's element type in array creation expressions.

Java strings are unusual, in that the language treats them almost as if they were primitive types supporting literals; instead they are instances of the Java `String` class. Thus, strings are objects and we could model them accordingly, namely like arrays. However, string literals always refer to the same instance of class `String`—these string literals are *interned*, so as to share unique instances. This is in contrast to strings which are concatenated at run-time. To distinguish both kinds of strings we need a dynamic function *interned* which always holds the set of references of interned strings. Furthermore, we have to modify the initialization of Java$_{\mathcal{O}}$, because strings can be assigned to constant fields. We refine the static function *classFieldValue*, so as to return not only primitive values but also (constant) strings. The initialization of *dyn* must be refined accordingly, i.e. if *classFieldValue* maps a field to a string, the string must be stored.

5 Adding Exceptions

Java$_{\mathcal{E}}$ extends Java$_{\mathcal{O}}$ with exceptions. We take particular care that our refinement of Java$_{\mathcal{O}}$ by exceptions makes it transparent how `break` and `continue` statements (of Java$_{\mathcal{I}}$), `return` statements and the initialization of classes and interfaces (in Java$_{\mathcal{O}}$) interact with catching and handling exceptions. Exception handling is a means of recovering from abnormal situations. Java's exceptions are represented by instances of class `Throwable`. Java distinguishes between *run-time exceptions* (which correspond to invalid operations violating the semantic constraints of Java), *errors* (which are failures detected by the executing machine) and *user-defined exceptions*. We consider here only run-time and user-defined exceptions, because, errors are considered as belonging to the JVM and are therefore ignored in the dynamic semantics.

5.1 Signature

The *abstract syntax* as presented in Fig. 9 defines the extension of Java$_{\mathcal{O}}$ by Java's exception handling constructs.

When an exception is thrown processing completes abruptly. To model this we extend the universe *Reason* with the reason *Throw* embedding the particular exception of type *Reference*.

Fig. 9 Abstract Java Syntax for Java$_{\mathcal{E}}$

Stm ::= ...
| `throw` *Exp*
| `try` *Block* `catch` (*Class Var*) *Block* ... `catch` (*Class Var*) *Block*
| `try` *Block* `finally` *Block* `endfinally`

Reason ::= ... | *Throw*(*Reference*)

Exceptions propagate through the grammatical block structure of a Java method and then up the method call stack to the nearest dynamically enclosing `catch` clause of a `try-catch` statement that handles the exception. A `catch` clause handles an exception if the exception object is compatible with the declared type.

In situations where it is desirable to ensure that after one block of code another one is always executed, Java provides the `try-finally` statement. The `finally` clause is generally used to clean-up after the `try` clause. It is executed if any portion of the `try` block—regardless how it completes—is executed. In the normal case control reaches the end of the `try` block and then proceeds to the `finally` block. If control leaves the `try` block abruptly, the code of the `finally` block is executed before control transfers to the 'intended' interrupt destination.

5.2 Transition Rules

We assume that Java$_{\mathcal{E}}$ is initialized like Java$_{\mathcal{O}}$ and execution starts normally. If all iterations and recursions of the given program terminate, Java$_{\mathcal{E}}$'s final state is $task_{\mathcal{O}} = \langle \texttt{finished} \rangle$. Then, if *mode* is *Return* execution has completed normally, otherwise *mode* denotes the thrown exception, which is not caught by the program.

Throwing exceptions. User-defined exceptions are thrown explicitly, using `throw` statements. Run-time exceptions are thrown, if certain semantic constraints for binary operations, target expressions and reference type cast expressions do not hold.

A `throw` *statement* is executed by first evaluating the expression. (Figure 6 captures the definition of the normal control flow.) If this evaluation completes abruptly, the `throw` statement completes abruptly, the reason being the same as the abrupt completion of the expression. Otherwise, if the value of the expression is *null*, a `NullPointerException` is thrown. In case the exception is not *null*, the intended exception is thrown: the control flow is abrupted the reason being a *Throw* with the value of the subexpression (§ 14.16).

Fig. 10 Normal control flow of `try-catch`

$$
\begin{array}{l}
\textbf{let } stm = \texttt{try } block \texttt{ catch } (\ldots) block_0 \ldots \texttt{ catch } (\ldots) block_n \textbf{ in} \\
\quad fst(stm) \;\; = fst(block) \\
\quad nxt(block) = nxt(block_i) = nxt(stm), 0 \le i \le n \\
\quad up(block) \;\; = \texttt{catch } (\ldots) block_0 \ldots \texttt{ catch } (\ldots) block_n \\
\quad up(block_i) \; = up(stm), 0 \le i \le n
\end{array}
$$

if *task* is `throw` *exp*; **then** (Throw)
  if $val(exp) \neq null$ **then**
    $mode := Throw(val(exp))$
    abrupt
  else fail(`NullPointerException`)

The macro 'fail' allocates an exception object, throws the exception and abrupts (which starts the execution of the corresponding `finally` code, if there is some, and the search for the appropriate exception handler; see the following subsection on propagating and handling of exceptions).

$$\text{fail}(class) \equiv \texttt{newInstance}(class,\ mode := Throw(ref)\ \text{abrupt})$$

This definition left out to call a *class* constructor. This is correct, as long as constructors only call superclass constructors. Otherwise 'fail' must be defined to be equivalent to execute the following code: `throw new` $class(class, \epsilon)()$;

A *binary expression* throws an `ArithmeticException`, if the operator is an integer division or remainder operator and the right operand is 0 (§ 15.13, 15.14). The following rule refines BinaryExp of Sect. 2.

if *task* is $(exp_1 \otimes exp_2) \wedge \otimes \in \{/, \%\} \wedge val(exp_2) = 0$ **then** (BinaryExp)
  fail(`ArithmeticException`)

An *instance target expression* throws a `NullPointerException`, if the operand is *null*. The following rule refines IFieldAcc, IFieldAss and IMethod of Sect. 4.

if (*task* is $exp.fieldspec \vee$ *task* is $exp.fieldspec = exp_2 \vee$ (ITarget)
  task is $exp.methodspec\{kind\}(exp_1, \ldots, exp_n)) \wedge$
  $val(exp) = null$ **then**
  fail(`NullPointerException`)

A reference type *cast expression* throws a `ClassCastException`, if the value of the immediately contained expression is neither *null* nor *compatible* with the required type (§ 15.15). The following rule refines Cast of Sect. 4.

if *task* is $(type)exp \wedge$ (Cast)
  $val(exp) \neq null \wedge \neg compatible(classOf(val(exp)), class)$ **then**
    fail(`ClassCastException`)

Propagating and handling exceptions. A `try-catch` *statement* is executed by executing first the `try` clause. If execution of the `try` clause completes normally, then this completes the `try-catch` statement normally. If in the `try` clause an exception is thrown, it is checked whether there is a `catch` clause, which can handle this exception. If the exception object is not compatible with any of the declared types, the exception propagates to the next higher enclosing block of code. Otherwise (the exception object is compatible with at least one declared type) the first (leftmost) compatible `catch` clause is selected, the exception is bound to the exception handler parameter and execution continues normally by executing the block of the selected `catch` clause. If processing of the latter abrupts, this abrupts the `try-catch` statement.

Figure 10 shows the control flow in normal mode. In addition it defines the function *up*, which is used when processing completes abruptly.

We need no rule for the `try` block, because the control flow abstracts from this syntactic construct. For the selection of the compatible `catch` clause we use the following transition rule, which uses the descriptive operator ι, to determine *that* `catch` clause which fulfills the given predicate.

if *task* is (`catch` $(c_0\ v_0)\ b_0 \ldots$ `catch` $(c_n\ v_n)\ b_n$) **then** (Catch)
  if $mode = Throw(exc) \wedge \exists\, i : 0 \leq i \leq n : catches(c_i)$ **then**
    $loc(v_k) := exc$
    $mode := undef$
    $task := fst(b_k)$
    where $k = \iota i : 0 \leq i \leq n : catches(c_i) \wedge \forall j : 0 \leq j < i : \neg catches(c_j)$
  else abrupt
  where $catches(class) = compatible(classOf(exc), class)$

If the thrown exception is compatible with any of the clauses, Catch assigns the exception object to the exception handler parameter, resets the processing mode, and proceeds normally.

If an exception is not caught by the block of code that throws it, it propagates to the next outer enclosing block of code. If an exception is not caught anywhere in the method, (i.e. *task* becomes `finished`) it cleans the bindings and the temporaries of this method call, and returns to the invoking method, where it again propagates through the block structure.

if *task* is `finished` $\wedge$ $mode = Throw(exc) \wedge length(task_{\bigcirc}) > 1$ **then** (Throw')
  frames := throw(frames)

To pop the task, temporaries and local variables from their respective stacks and to restart the search for an exception handler in the invoking method the rule uses the macro

$$\text{throw}(\langle \_, inv \rangle\ tasks, \langle \_ \rangle\ vals, \langle \_ \rangle\ locs) \equiv (\langle up(inv) \rangle\ tasks, vals, locs)$$

If an exception is never caught, it propagates all the way up to `main`'s body, where in Java$_{\mathcal{E}}$ processing gets stuck. This will be refined by exceptional thread termination in the next section.

Fig. 11 Normal control flow of `try-finally`/ `synchronized`

let $stm =$ `try` $block_0$ `finally` $block_1$ `endfinally` **in**

$fst(stm)$	$= fst(block_0)$
$nxt(block_0)$	$= up(block_0)$
	$=$ `finally` $block_1$
$nxt(block_1)$	$= up(block_1)$
	$=$ `endfinally`
$nxt($`endfinally`$)$	$= nxt(stm)$
$up($`endfinally`$)$	$= up(stm)$

let $stm =$ `synchronized` (exp) $block$ `endsynchronized` **in**

$fst(stm)$	$= fst(exp)$
$nxt(exp)$	$=$ `synchronized` (exp) $block$
$nxt(block)$	$= up(block)$
	$=$ `endsynchronized`
$nxt($`endsynchronized`$)$	$= nxt(stm)$
$up(exp)$	$= up($`endsynchronized`$)$
	$= up(stm)$

Handling clean-up code. A `try-finally` *statement* is executed by executing the `try` clause. Independently of whether the `try` clause completes normally or abruptly, the `finally` clause is executed, see Fig. 11 for the definition of the control flow functions *fst*, *nxt* and *up*. To restore thrown exceptions—when the `finally` clause is left—we introduce an initially empty dynamic function

$$\mathit{finally} : \mathit{Mode}^*$$

to store the current execution mode when the `finally` clause is entered.

if *task* is `finally` *block* **then** (Finally)
  $\mathit{finally} := \langle \mathit{mode} \rangle \mathit{finally}$
  $\mathit{mode} := \mathit{undef}$
  $\mathit{task} := \mathit{fst}(\mathit{block})$

If this `finally` clause completes—*task* points to `endfinally`—there is a choice: If the `finally` clause was entered in normal mode, the execution of the `try-finally` statement proceeds normally. If the `finally` clause was entered in abrupted mode, the execution abrupts again with the same reason given by that abrupted mode. However, if a `finally` clause itself completes abruptly, the pending control transfer is abandoned and this new transfer is processed (§ 14.18). This is expressed by the transition rule for `endfinally`.

if *task* is `endfinally` **then** (Endfinally)
  $\mathit{finally} := \mathit{finally}'$
  if $\mathit{mode} = \mathit{undef} \wedge \mathit{mode}' = \mathit{undef}$ **then** proceed
  elseif $\mathit{mode} = \mathit{undef} \wedge \mathit{mode}' \neq \mathit{undef}$ **then** $\mathit{mode} := \mathit{mode}'$
    abrupt
  else abrupt
  where $\langle \mathit{mode}' \rangle \mathit{finally}' = \mathit{finally}$

We invite the reader to check that these rules correctly formalize the description of the LRM for handling exceptions and clean-up (§ 14.18).

Initialization of classes and interfaces. Java is a robust language. So we also have to care about uncaught exceptions in static initializers. For them Java

specifies the following strategy: If the current class is in an erroneous state, then initialization is not possible; throw a `NoClassDefFoundError`. If during execution of the body of the `static` initializer an exception is thrown and this is not an `Error` or one of its subclasses, then throw `ExceptionInInitializerError`. (With respect to the subclassing of `ExceptionInInitializerError`, the LRM is *ambiguous*. Whereas § 11.5.2 defines it to be a subclass of `Error`, § 20.23 defines it to be a subclass of `RuntimeException`, which is the correct definition, since it signals an uncaught exception during execution and not during loading, linking or preparing.)

We extend the universe *InitState* by a new element, signalling that a class is in an erroneous state.

$$InitState ::= \ldots \mid Error$$

The following transition rule extends Static of Sect. 3.3, to whose guard we add the condition $init(\text{currClass}) = undef$. Furthermore, we assume the correct extension of the function *up*, namely by $up(\texttt{static}\ block) = \texttt{finished}$.

if *task* is `static` *block* $\wedge$ $init(\text{currClass}) = Error$ **then** (Static)
  fail(`NoClassDefFoundError`)

We have no extra transition rules for the `static` block. Instead we assume that the `static` block acts like the `try` block of a `try-catch` statement, where the `endstatic` phrase plays the rôle of the `catch` clause. This induces the correct extension of the *up* function for this case, formally let $init = \texttt{static}\ block\ \texttt{endstatic}$, then $up(block) = \texttt{endstatic}$ and $up(\texttt{endstatic}) = \texttt{finished}$.

According to the LRM, the rule for `endstatic` simply (re)throws the exception.

if *task* is `endstatic` $\wedge mode = Throw(ref)$ **then** (Endstatic)
  $init(\text{currClass}) := Error$
  if $compatible(classOf(ref), \texttt{Error})$ **then**
    frames := throw(frames)
  else fail(`ExceptionInInitializerError`)

This rule extends Endstatic of Sect. 3.3 to whose guard we add the condition $mode = undef$.

6 Adding Threads

The preceding models are concerned with the behavior of Java executing a single phrase at a time, that is by a single thread. In this section we extend $\text{Java}_{\mathcal{E}}$ to $\text{Java}_{\mathcal{T}}$, the model for *multithreaded* Java, which provides support for execution of many different tasks working on shared main and local working memory. We consider Java's thread creation and destruction, its mechanisms for synchronizing the concurrent activity of threads using locks, and Java's waiting and notification mechanism for efficient transfer of control between threads.

Fig. 12 Abstract Java Syntax for Java$_{\mathcal{T}}$

$$\begin{aligned} \mathit{Stm} &::= \ldots \mid \texttt{synchronized}\,(\mathit{Exp})\ \mathit{Block}\ \texttt{endsynchronized} \\ \mathit{Exp} &::= \ldots \mid \mathit{Exp}.\texttt{start}() \mid \mathit{Exp}.\texttt{stop}() \mid \mathit{Exp}.\,\texttt{wait}\,() \mid \mathit{Exp}.\,\texttt{notify}\,() \end{aligned}$$

The reference manual specifies a memory model for shared memory multiprocessors that support high performance implementations. It allows objects to reside in main and local working memory and presents rules (formalized as a particular event-structure by Cenciarelli et al. [15]) specifying when a thread is permitted or required to transfer the contents of its working copy of an instance variable into the master copy in main memory or vice versa. In order to separate this memory model—which "details the low-level actions that may be used to explain the interaction of Java Virtual Machine threads with a shared memory"[23, page 371]—from the semantics of the mechanisms defined by the language for thread creation, destruction, synchronization and for waiting and notification, we will first build a model for these mechanisms which uses only the main memory for storing objects and which agrees for best practice programs with the LRM memory model.

> "Best practice is that if a variable is ever to be assigned by one thread and used or assigned by another, then all accesses to that variable should be enclosed in `synchronized` statements." § 17.13.

In Sect. 6.3 we define another model, which supports local working memories and uses them as much as possible. The meaning of programs running on these two extreme memory models—both of which are in accordance with the semantics described in the LRM—agrees for best practice programs.

6.1 Signature

The abstract syntax of Java$_{\mathcal{T}}$ is given in Fig. 12, where—for ease of reading—invocation kinds are suppressed.

Threads are concurrent independent processes running within a single program so that they correspond to code executing *agents* in distributed ASMs [20]. For the modeling of threads we therefore use a universe

$$\mathit{Thread}$$

to formalize the objects belonging to the class `Thread` through which threads in Java are represented and controlled. Since threads are objects, the universe *Thread* is a subset of *Reference*. Every thread has its own state, consisting of its $\mathit{task}_{\mathcal{O}}$, $\mathit{loc}_{\mathcal{O}}$, $\mathit{val}_{\mathcal{O}}$ stacks and its execution mode, represented by the variables *mode* and *finally*; it is impossible for one thread to access parameters or local

variables or the execution mode of another thread. Correspondingly we parameterize each of these dynamic functions with its *agent*, i.e., we obtain the following signatures of these functions in Java$_{\mathcal{T}}$:

$$\begin{aligned} task_{\mathcal{T}} &: Thread \to Phrase^* \\ val_{\mathcal{T}} &: Thread \to (Exp \to Value)^* \\ loc_{\mathcal{T}} &: Thread \to (Var \to Value)^* \\ mode_{\mathcal{T}} &: Thread \to Reason \\ finally_{\mathcal{T}} &: Thread \to Reason^* \end{aligned}$$

During execution of a single thread the agent, denoted in Java$_{\mathcal{E}}$ by the logical function *self* (which supports the self-identification of agents), does not change. A consequence is that we can define our former functions as abbreviations of the refined ones. We have:

$$\begin{aligned} task_{\mathcal{O}} &\equiv task_{\mathcal{T}}(self) \\ val_{\mathcal{O}} &\equiv val_{\mathcal{T}}(self) \\ loc_{\mathcal{O}} &\equiv loc_{\mathcal{T}}(self) \\ mode &\equiv mode_{\mathcal{T}}(self) \\ finally &\equiv finally_{\mathcal{T}}(self) \end{aligned}$$

Through these abbreviations the rules, macros and propositions—where indicated with some further refinement—carry over from Java$_{\mathcal{E}}$ to Java$_{\mathcal{T}}$.

Threads exchange information among each other by operating on objects residing in shared *main memory*, which is modeled by the functions *glo*, *dyn* and *classOf*.

To synchronize threads Java uses *monitors*, a mechanism for allowing only one thread at a time to execute a region of code protected by the monitor. The behavior of monitors is explained in terms of locks uniquely associated with objects. When a `synchronized` statement is processed, the executing thread must grab the lock associated with the target reference to become the owner of the lock before the thread can continue. Upon completion of the block the mechanism releases that very same lock. We use a dynamic function *owns* to keep track of the dynamic nestings of synchronized statements; *owns*(*thread*) denotes the stack of all references grabbed by *thread*. Since a single thread can hold a lock more than once we have to define dynamic lock counters.

$$\begin{aligned} owns &: Thread \to Reference^* \\ locks &: Reference \to Nat \end{aligned}$$

To assist communication between threads, each object also has an associated *wait set*, a set of threads. Wait sets are used by the statements `wait` and `notify`. The `wait` method allows a thread to wait for a notification condition. Executing `wait` adds the current thread to the wait set for this object and releases the lock—which is reacquired to continue processing after the thread has been notified by another thread. Wait sets are modeled by the dynamic function

$$waitSet : Reference \to \mathcal{P}\, Thread.$$

Every thread can be in one of five *states*. This is modeled using the dynamic function:

$$exec : Thread \to ThreadState$$

where the universe *ThreadState* is defined by

$$ThreadState ::= Runnable \mid Blocked \mid Notified \mid Exiting.$$

A thread T is in the *initial* state ($exec(T) = undef$), from the period when it is created until the `start` method of the `Thread` object is called whereby it becomes *Runnable*. A thread that is in the *Blocked* state is one that cannot be run because a `wait` method has been called. A thread is in the *Notified* state once the `notify` method has been called for it. A thread is in the *Exiting* state once its `run` method has terminated or its `stop` method has been called.

The `stop` method is an asynchronous method. It may be invoked by one thread, to affect another thread in its current point of execution. We use a dynamic function

$$stopped : Thread \to \{Yes\}$$

to signal to a thread that it has been stopped. We strengthen the macro *task* is *phrase* for Java$_{\mathcal{T}}$ as follows:

$$\begin{aligned} task \text{ is } phrase \equiv\ & top(task_{\mathcal{T}}(self)) = phrase \wedge \\ & stopped(self) = undef \wedge \\ & exec(self) \in \{Runnable, Notified, Exiting\} \end{aligned}$$

so that a thread is only allowed to execute a phrase if the thread is neither stopped nor blocked.

The language reference manual leaves the *scheduling* strategy open. Although the language designers had a pre-emptive priority-based scheduler in mind, they explicitly say that there is no guarantee that threads with highest priority will always be running. Therefore, we abstract from priority based scheduling and use a 'loose' scheduling strategy. This means that we make the executability of a Java$_{\mathcal{T}}$-rule, in addition to its being guarded by *task* is *phrase*, depend only on the partial order conditions for distributed ASM runs, leaving the further specification for any particular scheduling open.

6.2 Transition Rules

The initial state of Java$_{\mathcal{T}}$ is like the one for Java$_{\mathcal{E}}$. The run starts with a single thread, called *main*, i.e., $\{main\} = Thread$. The thread *main* starts in normal mode, the state of *main* is *Runnable*, and *main*'s task, temporary and local stacks are initialized as discussed in Sect. 3.3. The remaining newly introduced dynamic functions are undefined.

Execution continues until all threads are blocked or have exited, i.e. the run of the distributed Java$_{\mathcal{T}}$ ASM terminates, if no agent can execute any rule any more.

Thread creation. There are two ways to create a new thread. One way is to declare a class to be a subclass of `Thread`; this subclass should override the `run` method of class `Thread`—the `run` method is the body of the thread. Creating an instance of this class by NewInstance creates a new thread T, which is (by default) in its initial state (so that in particular $exec(T) = undef$, $stopped(T) = undef$, $task_{\mathcal{T}}(T) = undef$, etc.). Since threads are modeled as agents of the distributed ASM $\text{Java}_{\mathcal{T}}$, by definition each newly created thread T gets the rules of $\text{Java}_{\mathcal{T}}$ assigned for execution with $T = self$.

Executing a thread. The `start` *method* of class `Thread` is used to cause a thread (provided it is not *null* and it is started for the first time) to begin execution by calling the `run` method of its `Thread` object (§ 20.20.14). If the thread to be started has already been started, an `IllegalThreadStateException` is thrown.

if *task* is exp.`start`() $\wedge$ $val(exp) \neq null$ **then** (Start)
  if $exec(val(exp)) = undef$ **then**
    newFrames := start($fst(body)$)
    $exec(val(exp)) := Runnable$
    proceed
  else fail(`IllegalThreadStateException`)
  where $((), body) = instMethod(\texttt{run}, classOf(val(exp)), Virtual)$
    newFrames $\equiv (task_{\mathcal{T}}(val(exp)), val_{\mathcal{T}}(val(exp)), loc_{\mathcal{T}}(val(exp)))$

A thread runs until the `run` method has nothing else to do or its `stop` method is called (see below). A thread terminates also if it could not handle an exception which had occurred (and after having executed all relevant `finally` clauses); in this case the uncaught exception method for the parent thread group is invoked. (We have not specified the latter). The transition rule for the normal termination is as follows:

if *task* is `finished` $\wedge$ $length(task_{\mathcal{O}}) = 1 \wedge exec(self) = Runnable$ **then** (Terminate)
  $exec(self) := Exiting$

Thread synchronization. A `synchronized` statement is executed by first grabbing the lock of the object denoted by the target reference, provided it is not *null*. (If it is *null* a `NullPointerException` is thrown.) If the current thread already holds the lock, or if the lock is free and the current thread is the one chosen for execution, it grabs the lock, increments the lock counter and executes the block; this is summarized in the macro 'lock'.

if *task* is `synchronized` (*exp*) *block* **then** (Synchronize)
  if $val(exp) \neq null$ **then**
    lock($val(exp), task := fst(block)$)
  else fail(`NullPointerException`)

Upon completion of the block—either normally or abruptly—the `endsynchronized` phrase is executed. (Figure 11 defines the control flow.) Executing `endsynchronized` releases the very same lock, namely the last in the sequence of locks grabbed by the thread due to the correct nesting of `synchronized`

and `endsynchronized` phrases, and decrements the lock counter, see the macro 'unlock'. If execution has completed normally, processing continues normally. If execution was abrupted, the reason of abruption is propagated up the nesting level.

if *task* is `endsynchronized` **then** (Endsynchronize)
  unlock(*top*(*owns*(*self*)),
    if *mode* = *undef* **then** proceed
      else abrupt)

The macro 'lock' tests whether the thread already holds a lock on the given reference. Whether the agent himself is the one chosen for locking the particular object, is captured by the macro 'competing', see below. If the thread already has or gets the lock the thread can enter the `synchronized` block; the lock counter is incremented and the grabbed reference is pushed onto the *owns* stack. The macro 'unlock' is an inverse of the macro 'lock'. ($elem \in list$ tests whether *elem* is in the *list*).

lock(*ref*, *updates*) ≡
  if *ref* ∈ *owns*(*self*) ∨ (*locks*(*ref*) ∈ {0, *undef*} ∧ *competing*(*ref*) = *self*) **then**
    owns(*self*) := ⟨*ref*⟩ *owns*(*self*)
    locks(*ref*) := *locks*(*ref*) + 1
    updates
unlock(*ref*, *updates*) ≡
  owns(*self*) := *pop*(*owns*(*self*))
  locks(*ref*) := *locks*(*ref*) − 1
  updates

The macro 'competing' uses a not further specified scheduling function $choose_{sync}$, to return an arbitrary thread out of a set of threads competing to get the lock for the given reference. A thread competes for a lock, if it executes a `synchronized` statement, a `wait` expression, or the phrase `static` or `endstatic`, provided the thread is in the appropriate state. We specify the macro '*competing*' in tabular form (where classObj(t) abbreviates $classRef(classScope(top(task_{\mathcal{T}}(t))))$):

competing(*ref*) ≡ $choose_{sync}\{t \in Thread \mid$

$top(task_{\mathcal{T}}(t)) = \cdot$	∧	$ref = \cdot$	∧	$state(t) \in \cdot$
`synchronized` (*exp*) *block*		$top(val(t))(exp)$		$\{Runnable, Exiting\}$
exp. `wait` ()		$top(val(t))(exp)$		$\{Notified\}$
`static` *block*		classObj(t)		$\{Runnable, Exiting\}$
`endstatic`		classObj(t)		$\{Runnable, Exiting\}$

}

It is easy to show that upon leaving a `synchronized` statement, the previous state of the lock counters and the stack of grabbed references become exactly as they were when the statement was entered.

Thread notification. The `wait` *method* of class `Object` causes the current *Runnable* thread to wait until some other thread invokes the `notify` method for this object. This method can be called only when the current thread is already synchronized on this object, otherwise an `IllegalMonitorStateException` is thrown. Executing `wait` *enable*s the thread from executing: it changes the state of the process from *Runnable* to *Blocked*, adds the current thread to the wait set for this object and releases the lock. The LRM does *not* specify what happens if already *Exiting* threads should wait. We decided that the thread should behave as if it were *Runnable*. (Other choices are conceivable and easily formalized changing our rules.) To remember that a thread is already exiting we introduce a new dynamic function

$$backup : Thread \rightarrow ThreadState$$

which is set when the current thread enters the `wait` expression and is used when it proceeds from there.

if $task$ is $(exp.$ `wait` $()) \wedge exec(self) \in \{Exiting, Runnable\} \wedge$ (Wait)
$val(exp) \neq null$ **then**
if $val(exp) \in owns(self)$ **then**
$backup(self) := exec(self)$
enable$(val(exp), self, Blocked)$
else fail(`IllegalMonitorStateException`)

To continue processing the thread first has to be notified by another thread.

The `notify` *method* of class `Object` chooses one thread among those waiting on this object. The choice is left unspecified by the Java LRM; we reflect this by introducing yet another not furthermore specified choice function, say $choose_{notify}$. The chosen thread is then removed from the wait set and its state is changed from *Blocked* to *Notified*. We say the thread is *awake*d. The `notify` method may be called only when the current thread is already synchronized on this object, otherwise an `IllegalMonitorStateException` is thrown.

if $task$ is $(exp.$ `notify` $()) \wedge val(exp) \neq null$ **then** (Notify)
if $val(exp) \in owns(self)$ **then**
if $waitSet(val(exp)) \neq \emptyset$ **then**
awake$(val(exp), choose_{notify}(waitSet(val(exp))), Notified)$
proceed
else fail(`IllegalMonitorStateException`)

The notified thread whose *task* still points to the `wait` method invocation then competes in the usual manner with other threads for the right to synchronize on the object. Once the *reenable*d thread has gained control of the object, all its synchronization claims are reacquired, its state switches to the state in which `wait` was entered (which is either *Runnable* or *Exiting*) and its execution continues normally.

if $task$ is $(exp.$ `wait` $()) \wedge exec(self) = Notified \wedge val(exp) \neq null$ **then** (Wait)
reenable$(val(exp), self, backup(self),$ proceed$)$

The macros 'enable', 'awake' and 'reenable' show the relation between these three notification phases. ($occs(list, elem)$ yields the number of occurrences of $elem$ in the $list$).

$$\begin{aligned}
\text{enable}(ref, thread, state) \equiv\ & waitSet(ref) := waitSet(ref) \cup \{thread\} \\
& locks(ref) \quad := 0 \\
& exec(thread) := state \\
\text{awake}(ref, thread, state) \equiv\ & waitSet(ref) := waitSet(ref) - \{thread\} \\
& exec(thread) := state \\
\text{reenable}(ref, thread,\ & \\
state, updates) \equiv\ & \textbf{if } locks(ref) \in \{0, undef\} \wedge competing(ref) = thread \textbf{ then} \\
& \quad locks(ref) \quad := occs(owns(thread), ref) \\
& \quad exec(thread) := state \\
& \quad updates
\end{aligned}$$

It is easy to show that upon return from the `wait` method, the synchronization state of the object of this thread returns exactly as it was when the `wait` method was invoked.

Stopping a thread. The `stop` *method* of class `Thread` is an asynchronous method. It may be be invoked by one thread to throw the error `ThreadDeath` for another thread; the thread to be stopped is notified if it is waiting. It is permitted to stop threads in *Initial* as well as in *Exiting* mode. In the former case, if the thread is eventually started, it will immediately terminate. In the latter case, nothing happens. In the usual case that the exception is not caught, it propagates up to the `run` (or for the *main* thread up to the `main`) method of this thread (§ 20.20.15).

We model this asynchronous behavior by two rules. During invocation of the `stop` method, we set the dynamic function *stopped* to *Yes* for the stopped thread, provided this thread is not already exiting.

if *task* is *exp*.`stop`() $\wedge$ $val(exp) \neq null$ **then** (Stop)
  if $exec(val(exp)) \neq Exiting$ **then**
    $stopped(val(exp)) := Yes$
  proceed

If a *Runnable* thread receives the stop signal, it changes its execution state to *Exiting*, resets the dynamic function *stopped* and throws the error `ThreadDeath`. *Blocked* threads first have to be awaked, and *Notified* threads first have to be reenabled, so that they can change their state and throw the exception.

if $stopped(self)$ **then** (Stopped)
  case $exec(self)$ **of**
  Runnable : stop
  Blocked : awake($val(exp), self, Notified$)
  Notified : reenable($val(exp), self, Exiting$, stop)
  where *exp*. `wait` () = *task*

The used macro 'stop' changes the thread's state and raises the exception.

$$\begin{aligned} \text{stop} \equiv\ & \mathit{stopped}(\mathit{self}) := \mathit{undef} \\ & \mathit{exec}(\mathit{self}) \quad := \mathit{Exiting} \\ & \text{fail}(\texttt{ThreadDeath}) \end{aligned}$$

Stopped threads can continue to work—the `ThreadDeath` exception may be caught and execution can proceed as if nothing had happened, except that (as is easy to show for Java$_{\mathcal{T}}$) the stopped thread cannot be stopped once more.

The thread that receives the stop signal immediately reacts. For implementations this might be rather expensive. So Java permits a small but bounded amount of execution to occur before an asynchronous exception is received. Java's LRM (§ 11.3.2) gives *only hints* how big the 'bounded amount' might be. In our model it is matter of refining Stopped (and the macro 'is') by strengthening (or weakening) the conditions to specify precisely when a stop signal is received.

Initialization of classes and interfaces. Initialization of a class or interface in Java$_{\mathcal{T}}$ requires synchronization, since several threads may be trying to initialize the class at the same time. If initialization by one thread is *InProgress*, other threads have to wait until the initialization is *Done* or an *Error* occurs. To distinguish the thread that actually initializes a class from those that have to wait, we introduce a dynamic function

$$\mathit{initThread} : \mathit{Class} \to \mathit{Thread}$$

and we refine the predicate 'initialized', so that it is true, if either initialization is *Done* or, if initialization is *InProgress* the initializing thread must be the current thread.

$$\begin{aligned} \text{initialized}(\mathit{class}) \equiv\ & \mathit{init}(\mathit{class}) = \mathit{Done} \vee \\ & (\mathit{initState}(\mathit{class}) = \mathit{InProgress} \wedge \mathit{initThread}(\mathit{class}) = \mathit{self}) \end{aligned}$$

The procedure of Java$_{\mathcal{E}}$ for initializing a class or interface must then be refined as follows. During processing the `static` phrase, first synchronize on the `Class` object. (This is captured by a strengthened 'is' predicate.) If initialization has not yet started, record that the current thread initializes the class, set the initialization state to *InProgress* and either invoke the initialization for the superclass (if any) or enter the computation of the `static` block.

if *task* is `static` *block* $\wedge$ *init*(currClass) = *undef* **then** (Static)
  initThread(*class*) := *self*
  init(currClass) := *InProgress*
  enter

If initialization is *InProgress* by some other thread, then wait. (According to the refined 'initialized' predicate, we do not start the class initialization, if it is already *InProgress* by the current thread).

if *task* is `static` *block* $\wedge$ *init*(currClass) = *InProgress* **then** (Static)
  backup(*self*) := *exec*(*self*)
  enable(*classRef*(currClass), *self*, *Blocked*)

If initialization is *Done*, then no further action is required; return immediately.

if *task* is `static` *block* $\wedge$ *init*(currClass) = *Done* **then** (Static)
  exit

The case that the initialization is erroneous (see Sect. 5.2), remains the same modulo the refinement defined below for *task* is *phrase*.

To express the synchronization of the `Class` object, we strengthen the macro 'is' by an additional condition, which guarantees (just for entering or exiting the class initializer) that the current thread is chosen to execute the monitor.

$$\begin{aligned} \mathit{task}\ \underline{\text{is}}\ \mathit{phrase} \equiv\ & \mathit{task}\ \text{is}\ \mathit{phrase} \wedge \\ & (\mathit{classRef}(\text{currClass}) \in \mathit{owns}(\mathit{self}) \vee \\ & (\mathit{locks}(\mathit{classRef}(\text{currClass})) \in \{0, \mathit{undef}\} \wedge \\ & \mathit{competing}(\mathit{classRef}(\text{currClass})) = \mathit{self})) \end{aligned}$$

When executing `endstatic`, we have again to synchronize on the `Class` object. In every case (i.e., whether *mode* = *undef* or *mode* = *Throw*(*ref*)) we have to wake up all waiting threads and to reset their state. We add to the previous Endstatic rules of Sect. 3.3 and 5.2 the following one:

if *task* is `endstatic` **then** (Endstatic)
  vary *thread* **over** *waitset*(*classRef*(currClass))
    awake(*classRef*(currClass), *thread*, *backup*(*thread*))

By closer inspection of the rules, one can observe that concurrent class initialization may *deadlock*: Assume that two threads T and S start the initialization of two different classes A and B; if during their respective initializations T encounters a first active use of B, and similarily S encounters a first active use of A, both threads become *Blocked* and there is no way that either one of the threads becomes *Runnable* or *Exiting* again. This is in accordance with the LRM.

6.3 Adding Local Working Memory

In Java$_{\mathcal{T}}$ class and instance fields, as well as dynamic type information are kept in main memory, shared by all threads. The main memory is represented by the *glo*, *dyn* and *classOf* functions. This model is not appropriate when Java runs on a shared-memory multiprocessor computer supporting *local working memories*.

In the sequel we will discuss the necessary modifications to support local memories for instance fields. The adoption of the following strategy to support local working memories for class fields and the *classOf* function is described at the end of this section.

Local working memories of threads, as described in the LRM, can be modeled by the dynamic function

$$\mathit{cache} : \mathit{Thread} \to \mathit{Reference} \times \mathit{FieldSpec} \to \mathit{CacheValue}$$

storing values, which are tagged as either *Used* or *Assigned* by the thread; so the universe *CacheValue* is defined by:

$$CacheValue ::= Used(Value) \mid Assigned(Value)$$

The LRM defines rules when a thread is permitted or required to transfer data between main memory, where the master copies of variables are hold, and the local working memory, where working copies reside. The following strategy which guides the definition of the Java model $\text{Java}_{\mathcal{T}}(\text{local})$ in this section is consistent with those rules: Every thread works as much as possible on its own working copy of a variable. When a thread assigns a variable, it uses *its* cache for storing it (in the sequel we abbreviate $cache(self)$ by 'cache') so that we have to refine IFieldAss as follows:

if $task$ is $(exp_1.fieldSpec = exp_2) \wedge val(exp_1) \neq null$ **then** (IFieldAss)
  $\text{cache}(val(exp_1), fieldSpec) := Assigned(val(exp_2))$
  $val(task) := val(exp_2)$
  proceed

Variable access is slightly more complicated. The LRM requires that new variables are always allocated in main memory. Therefore, instance field access has to distinguish whether the variable is already cached or whether it must be transferred between main memory and the local working memory.

if $task$ is $(exp.fieldSpec) \wedge val(exp) \neq null$ **then** (IFieldAcc)
  if $\text{cache}(val(exp), fieldSpec) = undef$ **then**
    $\text{cache}(val(exp), fieldSpec) := Used(dyn(val(exp), fieldSpec))$
    $val(task) := dyn(val(exp), fieldSpec)$
  else $val(task) := \text{get}(\text{cache}(val(exp), fieldSpec))$
  proceed
  where $\text{get}(c) =$**if** $c \in \{Assigned(v), Used(v)\}$ **then** v

The `synchronized` statement allows reliable transmission of values from one thread to another through shared main memory. When we enter a `synchronized` block, we flush all variables (of the target reference of the `synchronized` statement) from the thread's working memory. We model this effect by extending the application of the macro 'lock' in Synchronize as follows:

lock($val(exp)$,
  vary f **over** $instFields(classOf(val(exp)))$
    $\text{cache}(val(exp), f) := undef$
  proceed)

IFieldAcc guarantees that before using a variable the variable is either assigned or loaded from main memory.

When we leave a `synchronized` block, the thread must copy all assigned values in its working memory back to main memory. To this purpose, we extend the application of the macro 'unlock' in Endsynchronize as follows:

unlock($top(owns(self))$,
  vary f **over** $instFields(classOf(top(owns(self))))$
    if $\text{cache}(top(owns(self)), f) = Assigned(res)$ **then**
      $dyn(top(owns(self)), f) := res$
  if $mode = undef$ **then** proceed
    else abrupt)

In contrast to our model, the LRM (§ 17.6) requires to flush all variables from the thread's working memory—this is *overspecified*. For best practice programs our formulation is sufficient to guarantee reliable transmission of values between different threads. Furthermore Java's LRM requirement would reduce the expected performance gain when using shared memory multiprocessors.

Supporting local memories for class fields requires analogous modifications. It is even possible to support local versions of the *classOf* function. However, as long as garbage collection is not considered, local copies of the *classOf* function need not be retransmitted into main memory, since the *classOf* function is only assigned once.

The LRM also formulates rules about the time delay between the transfer of variables from main memory into local working memory and vice versa. However for best practice programs neither these time delays nor the different memory models produce any semantical difference.

7 Conclusion and Outlook

In this work we have defined a rigorous abstract operational model which captures faithfully the programmer's view of Java as described in the reference manual [19]. The model can be used for standardization purposes along the lines the ASM model for Prolog [10] has been used for defining the ISO standard for the semantics of Prolog. For such an endeavor it is important that our mathematical definition of the semantics of Java yields a complete model which is falsifiable by mental or machine experiments, in the sense of Popper [29], and thus complements and enhances purely experimental studies of Java and its implementations (see for example the Kimera project [33]).

Our definition provides a basis for a machine and system independent mathematical analysis of the behavior of Java programs. As illustration we cite here some examples of theorems we can formulate and prove in rigorous mathematical terms for our models of Java; we hope to publish these and related results at another occasion.

Theorem 1. *In the sequence* $\text{Java}_{\mathcal{I}}$, $\text{Java}_{\mathcal{C}}$, $\text{Java}_{\mathcal{O}}$, $\text{Java}_{\mathcal{E}}$, $\text{Java}_{\mathcal{T}}$ *each model is a conservative extension of its predecessor.*

This theorem strongly supports the semantics (not only syntax) based modular approach we propose for the study of Java and its implementations. In particular it allows us to decompose the justification for the correctness of our Java model w.r.t. the LRM into the (routine) justification of the correctness of $\text{Java}_{\mathcal{I}}$ followed by the separate justification of the orthogonal procedural, object-oriented, exception handling and concurrency features and their interaction.

Theorem 2. *Upon correct initialization the exception handling in* $\text{Java}_{\mathcal{E}}$ *and* $\text{Java}_{\mathcal{T}}$ *is precise. Each exception is either caught or propagates through the sequence of method calls to the first statement with which the given program was started.*

Theorem 3. *The semantics of best practice programs in the same in the main memory model* $Java_{\mathcal{T}}$ *and in its refinement by local working memories* $Java_{\mathcal{T}}$*(local).*

Theorem 4. *The static initialization in* $Java_{\mathcal{C}}$*,* $Java_{\mathcal{E}}$ *and* $Java_{\mathcal{T}}$ *runs is correct, i.e. it is done for each class (by exactly one thread) exactly at the first use of the relevant field modification, constructor or method invocation. Once a class is initialized, all its superclasses are initialized.*

The modular structure, and the relegation of standard compile-time matters to static functions, which support the comprehension of the model by humans and its use for proving interesting properties for Java programs, are two main features which distinguish our work from the approach of Wallace [37], which is geared towards executability of the ASM specification. A comparison of these two models illustrates the high degree of freedom ASMs offer to tune a mathematical model to its intended use.

We are working on refinements of our Java model to the level of abstraction of the Java Virtual machine [13]. These refinements take advantage of the modular specification of orthogonal Java features as they appear in our models. The ASMs we are developing for the JVM provide the basis for a rigorous mathematical analysis of general compilation schemes of Java programs into JVM code including correctness proofs as developed for the implementation for Prolog on the WAM [11] (see also [32]) and of Occam on the Transputer [5].

We are also working on applying our JVM models for safety analysis of Java byte code along the research approaches of Stati and Abadi [34], Qian [30] and Cohen [16].

Acknowledgement. We thank the following persons for having read and commented upon previous versions of our Java models: Martin Abadi, Klaus Achatz, Matthias Anlauff, Giuseppe Del Castillo, Dag Diesen, Igor Durdanovic, Vincenzo Gervasi, Alexander Knapp, Bernd Koblinger, Philipp Kutter, Arnd Poetzsch-Heffter, Peter Päppinghaus, Karl Stroetmann, Giovanni Ricci, Kirsten Winter, Wolf Zimmermann, Ton Vullinghs, and in particular Esben Krag Hanson, for pointing out an omision in connection with the try-finally rules.

References

1. G. Bella and E. Riccobene. Formal analysis of the Kerberos authentification system. *Journal of Universal Computer Science (J.UCS)*, 1997.
2. E. Börger. Why use evolving algebras for hardware and software engineering. In *SOFSEM'95 22nd Seminar on Current Trends in Theory and Practice of Informatics*, Springer LNCS 1012, 1995.
3. E. Börger, C. Beierle, I. Durdanovic, U. Glässer, and E. Riccobene. Refining abstract machine specifications of the steam boiler control to well documented executable code. In *Formal Methods for Industrial Applications. Specifying and Programming the Steam-Boiler Control*, Springer LNCS State–of–the–Art Survey 1165, 1996.

4. E. Börger and G. Del Castillo. A formal method for provably correct composition of a real–life processor out of basic components (the APE100 reverse engineering project). In *Proc. First IEEE International Conference on* Engineering of Complex Computer Systems *(ICECCS'95)*. IEEE Computer Society Press, 1995.
5. E. Börger and I. Durdanovic. Correctness of compiling Occam to transputer code. *The Computer Journal*, 39, 1996.
6. E. Börger, I. Durdanovic, and D.Rosenzweig. Occam: Specification and compiler correctness. Part I: Simple mathematical interpreters. In E.-R. Olderog, editor, *Proc. PROCOMET'94 (IFIP Working Conference on Programming Concepts, Methods and Calculi)*, 1994.
7. E. Börger, U. Glässer, and W. Müller. Formal definition of an abstract VHDL'93 simulator by EA-machines. In C. Delgado Kloos and P.T. Breuer, editors, *Formal Semantics for VHDL*. Kluwer Academic Publishers, 1995.
8. E. Börger and S. Mazzanti. A practical method for rigorously controllable hardware design. In *ZUM'97: The Z Formal Specification Notation*, Springer LNCS 1212, 1997.
9. E. Börger and L. Mearelli. Integrating ASMs into the software development lifecycle. *Journal of Universal Computer Science*, Special ASM Issue, 3.5, 1997.
10. E. Börger and D. Rosenzweig. A mathematical definition of full Prolog. *Science of Computer Programming*, 24, 1995.
11. E. Börger and D. Rosenzweig. The WAM - definition and compiler correctness. In L.Plümer C.Beierle, editor, *Logic Programming: Formal Methods and Practical Applications*. Elsevier Science B.V./North-Holland, 1995.
12. E. Börger and W. Schulte. A programmer friendly modular definition of the semantics of Java. In R. Berghammer and F. Simon, editors, Programming Languages and Fundamentals of Programming. Technical report, Christian Albrechts Universität Kiel, Institut für Informatik und Praktische Mathematik, 1997.
13. E. Börger and W. Schulte. Defining the Java Virtual Machine as platform for provably correct Java compilation. In J. Gruska, editor, *23rd International Symposium on Mathematical Foundations of Computer Science, Brno, Czech Republic, August 24–28, 1998*, Springer LNCS, to appear, 1998.
14. E. Börger and W. Schulte. A modular design for the Java VM architecture. In E. Börger, editor, *Architecture Design and Validation Methods*. Springer Verlag, to appear, 1998.
15. P. Cenciarelli, A. Knapp, B. Reus, and M. Wirsing. An event-based structural operational semantics of multi-threaded Java. 1998. This volume.
16. R. M. Cohen. Defensive Java virtual machine version 0.5 alpha release. manuscript, Computer Logic International.
17. G. Del Castillo, I. Durdanović, and U. Glässer. An Evolving Algebra abstract machine. In H. Kleine Büning, editor, *Computer Science Logic (CSL'95)*, Springer LNCS 1092, 1996.
18. S. Drossopoulou and S. Eisenbach. Java is type safe – probably. 1998. This volume.
19. J. Gosling, B. Joy, and G. Steele. *The Java(tm) Language Specification*. Addison Wesley, 1996.
20. Y. Gurevich. Evolving algebras 1993: Lipari guide. In E. Börger, editor, *Specification and Validation Methods*. Oxford University Press, 1995.
21. J. Huggins. Kermit: Specification and verification. In E. Börger, editor, *Specification and Validation Methods*. Oxford University Press, 1995.
22. G. Kahn. Natural semantics. Technical report, INRIA Rapport de Recherche No. 601, Février 1987.

23. T. Lindholm and F. Yellin. *The Java(tm) Virtual Machine Specification*. Addison Wesley, 1996.
24. B. Meyer. *Object-oriented Software Construction*. Prentice Hall, 1988.
25. P. D. Mosses. *Action Semantics*. Cambridge University Press, 1992.
26. Th. Nipkow and D. von Oheimb. Java_{light} is type-safe – definitely. 1998. This volume.
27. G. Plotkin. A structural approach to operational semantics. Technical report, Internal Report, CS Department, Aarhus University, DAIMI FN-19, 1997.
28. A. Poetzsch-Heffter. Prototyping realistic programming languages based on formal specifications. *Acta Informatica*, 34, 1997.
29. K. Popper. *Logik der Forschung*. 1935.
30. Z. Qian. A formal specification of Java(tm) Virtual Machine for objects, methods and subroutines. This volume, 1998.
31. V. Saraswat. Java is not type-safe. manuscript, AT&T Research, New York, 1997.
32. G. Schellhorn and W. Ahrendt. Reasoning about Abstract State Machines: The WAM Case Study. *Journal of Universal Computer Science (J.UCS)*, 1997.
33. E.G. Sirer, S. McDirmid, and B. Bershad. Kimera: A Java system security architecture. Web pages at: `http://www.kimera.cs.washington.edu/`, 1997.
34. R. Stata and M. Abadi. A type system for Java bytecode subroutines. In *Proceedings of the 25th Annual SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, 1998.
35. D. Syme. Proving Java type soundness. 1998. This volume.
36. Ch. Wallace. The semantics of the C++ programming language. In E. Börger, editor, *Specification and Validation Methods*. Oxford University Press, 1995.
37. Ch. Wallace. The semantics of the Java programming language: Preliminary version. Technical report, University of Michigan, Electrical Engineering and Computer Science, Ann Arbor, 1997.

A ASM Rules for Java

A.1 Semantic Domains

Domains for method execution

$$
\begin{aligned}
\mathit{Value} &::= \mathit{Bool} \mid \mathit{Integers} \mid \mathit{Floats} \mid \mathit{Reference} \mid \{\texttt{null}\} \\
\mathit{Reason} &::= \mathit{Break}(\mathit{Lab}) \mid \mathit{Continue}(\mathit{Lab}) \\
&\quad \mid \mathit{Return} \mid \mathit{Result}(\mathit{Value}) \mid \mathit{Throw}(\mathit{Reference})
\end{aligned}
$$

$$
\begin{aligned}
\mathit{task}_{\mathcal{T}} &: \mathit{Thread} \to \mathit{Phrase}^* \\
\mathit{val}_{\mathcal{T}} &: \mathit{Thread} \to (\mathit{Exp} \to \mathit{Value})^* \\
\mathit{loc}_{\mathcal{T}} &: \mathit{Thread} \to (\mathit{Var} \to \mathit{Value})^* \\
\mathit{mode}_{\mathcal{T}} &: \mathit{Thread} \to \mathit{Reason} \\
\mathit{finally}_{\mathcal{T}} &: \mathit{Thread} \to \mathit{Reason}^*
\end{aligned}
$$

$$
\begin{aligned}
\mathit{task}_{\mathcal{O}} &\equiv \mathit{task}_{\mathcal{T}}(\mathit{self}) & \mathit{task} &\equiv \mathit{top}(\mathit{task}_{\mathcal{O}}) \\
\mathit{val}_{\mathcal{O}} &\equiv \mathit{val}_{\mathcal{T}}(\mathit{self}) & \mathit{val} &\equiv \mathit{top}(\mathit{val}_{\mathcal{O}}) \\
\mathit{loc}_{\mathcal{O}} &\equiv \mathit{loc}_{\mathcal{T}}(\mathit{self}) & \mathit{loc} &\equiv \mathit{top}(\mathit{loc}_{\mathcal{O}}) \\
\mathit{mode} &\equiv \mathit{mode}_{\mathcal{T}}(\mathit{self}) & \mathrm{frames} &\equiv (\mathit{task}_{\mathcal{O}}, \mathit{val}_{\mathcal{O}}, \mathit{loc}_{\mathcal{O}}) \\
\mathit{finally} &\equiv \mathit{finally}_{\mathcal{T}}(\mathit{self})
\end{aligned}
$$

Domains for modeling class and instance variables

$$\begin{aligned} glo &: FieldSpec \to Value \\ dyn &: Reference \times FieldSpec \to Value \\ classOf &: Reference \to Class \end{aligned}$$

Domains for modeling concurrency

$$ThreadState ::= Runnable \mid Blocked \mid Notified \mid Exiting$$

$$\begin{aligned} exec, backup &: Thread \to ThreadState \\ stopped &: Thread \to \{Yes\} \\ owns &: Thread \to Reference^* \\ locks &: Reference \to Nat \\ waitSet &: Reference \to \mathcal{P}\, Thread \end{aligned}$$

Domains for class initialization

$$InitState ::= InProgress \mid Done \mid Error$$

$$\begin{aligned} init &: Class \to InitState \\ initThread &: Class \to Thread \end{aligned}$$

General Macros

$$\begin{aligned} \text{proceed} &\equiv task := nxt(task) \\ \text{abrupt} &\equiv task := up(task) \end{aligned}$$

$$task \text{ is } phrase \equiv top(task_{\mathcal{T}}(self)) = phrase \wedge stopped(self) = undef \wedge exec(self) \in \{Runnable, Notified, Exiting\}$$

$$\text{initialized}(class) \equiv init(class) = Done \vee (initState(class) = InProgress \wedge initThread(class) = self)$$

A.2 Java$_{\mathcal{I}}$: The Imperative Core

if $task$ is lit **then** (Literal)
$val(task) := \widetilde{lit}$
proceed

if $task$ is $(\odot exp)$ **then** (UnaryExp)
$val(task) := \widetilde{\odot}(val(exp))$
proceed

if $task$ is $(exp_1 \otimes exp_2) \wedge (\otimes \in \{/, \%\} \Rightarrow val(exp_2) \neq 0)$ **then** (BinaryExp)
$val(task) := val(exp_1) \widetilde{\otimes} val(exp_2)$
proceed

if *task* is *var* **then** (VarAcc)
  $val(task) := loc(var)$
  proceed

if *task* is $(var = exp)$ **then** (VarAss)
  $loc(var) := val(exp)$
  $val(task) := val(exp)$
  proceed

if *task* is $exp_1?exp_2 : exp_3$ **: then** (IfExp)
  if $val(exp_1)$ **then** $task := fst(exp_2)$
    else $task := fst(exp_3)$

if *task* is exp**: then** (ThenElseExp)
  $val(if(task)) := val(exp)$
  $task := nxt(if(task))$

if *task* is `if` (exp) stm_1 `else` stm_2 **then** (IfStm)
  if $val(exp)$ **then** $task := fst(stm_1)$
    else $task := fst(stm_2)$

if *task* is `while` (exp) *stm* **then** (While)
  if $val(exp)$ **then** $task := fst(stm)$
    else $task := nxt(task)$

if *task* is `jump` *lab*; **then** (Jump)
  $mode := Jump(lab)$
  abrupt
for $(\texttt{jump}, Jump) \in \{(\texttt{break}, Break), (\texttt{continue}, Continue)\}$

if *task* is $lab : stm$ **then** (LabStm)
  if $mode = Jump(lab)$ **then**
    $mode := undef$
    $task := jump$
  else abrupt
for $(Jump, jump) \in \{(Break, nxt(task)), (Continue, fst(stm))\}$

A.3 Java$_{\mathcal{C}}$: Adding Classes

if *task* is $(class, field) \wedge$ initialized$(class)$ **then** (CFieldAcc)
  $val(task) := glo(class, field)$
  proceed

if *task* is $((class, field) = exp) \wedge$ initialized$(class)$ **then** (CFieldAss)
  $glo(class, field) := val(exp)$
  $val(task) := val(exp)$
  proceed

if $task$ is $((class, method, fcty)(exp_1, \ldots, exp_n)) \wedge$ (CMethod)
initialized($class$) **then**
frames := invoke($\langle val(exp_1), \ldots, val(exp_n)\rangle, args, fst(body)$,frames)
where $(args, body) = classMethod(class, method, fcty)$

$$\text{invoke}(\langle val_1, \ldots, val_n\rangle, \langle var_1, \ldots, var_n\rangle, phrase, (tasks, vals, locs)) \equiv (\langle phrase\rangle\ tasks, \langle\emptyset\rangle\ vals, \langle\{(var_1, val_1), \ldots, (var_n, val_n)\}\rangle\ locs)$$

if $task$ is `return` exp; **then** (Result)
$mode := Result(val(exp))$
abrupt

if $task$ is `finished` $\wedge\ mode = Result(res) \wedge length(task_{\mathcal{O}}) > 1$ **then** (Result')
$mode := undef$
frames := result(res, frames)

if $task$ is `return`; **then** (Return)
$mode := Return$
abrupt

if $task$ is `finished` $\wedge\ mode = Return \wedge length(task_{\mathcal{O}}) > 1$ **then** (Return')
$mode := undef$
frames := return(frames)

$$\text{result}(res, (\langle \_, inv\rangle\ tasks, \langle \_, val\rangle\ vals, \langle \_\rangle\ locs)) \equiv (\langle nxt(inv)\rangle\ tasks, \langle val \oplus \{(inv, res)\}\rangle\ vals, locs)$$

$$\text{return}(\langle \_, inv\rangle\ tasks, \langle \_\rangle\ vals, \langle \_\rangle\ locs) \equiv (\langle nxt(inv)\rangle\ tasks, vals, locs)$$

A.4 Java$_{\mathcal{O}}$: Adding Objects

if $task$ is `new` $class \wedge$ initialized($class$) **then** (NewInstance)
newInstance($class$,
$val(task) := ref$)
proceed

newInstance($class, updates$) $\equiv$ **extend** $Reference$ **by** ref
$classOf(ref) := class$
vary f **over** $instFields(class)$
$dyn(ref, f) := instFieldValue(f)$
$updates$

if $task$ is $new\ constrSpec(exp_1, \ldots, exp_n)$ **then** (Constr)
frames := invoke($\langle this\rangle\ vals, \langle$`this`$\rangle\ args, fst(body)$,frames)
where $(args,\ body) = instConstr(constrSpec)$
$this$ = **if** $new =$ `new` $class$ **then** $val(new)$ **else** $loc($`this`$)$
$vals$ = $\langle val(exp_1), \ldots, val(exp_n)\rangle$

if *task* is `this` **then** (This)
  $val(task) := loc(\texttt{this})$
  proceed

if *task* is $(exp.fieldSpec) \wedge val(exp) \neq null$ **then** (IFieldAcc)
  $val(task) := dyn(val(exp), fieldSpec)$
  proceed

if *task* is $(exp_1.fieldSpec = exp_2) \wedge val(exp_1) \neq null$ **then** (IFieldAss)
  $dyn(val(exp_1), fieldSpec) := val(exp_2)$
  $val(task) := val(exp_2)$
  proceed

if *task* is $(exp.methodSpec\{kind\}(exp_1, \ldots, exp_n)) \wedge$ (IMethod)
  $val(exp) \neq null$ **then**
  frames := invoke($\langle val(exp)\rangle$ *vals*, $\langle$`this`$\rangle$ *args*, $fst(body)$, frames)
  where $(args,\ body) = instMethod(methodSpec, class, kind)$
    $vals = \langle val(exp_1), \ldots, val(exp_n)\rangle$
    $class =$ **case** *kind* **of** *Nonvirtual* : currClass
      Virtual : $classOf(val(exp))$
      Super : $super$(currClass)

if *task* is (*exp* `instanceof` *class*) **then** (Instanceof)
  $val(task) := val(exp) \neq null \wedge$
    $compatible(classOf(val(exp)), class)$
  proceed

if *task* is $(class)exp \wedge$ (Cast)
  $val(exp) = null \vee compatible(classOf(val(exp)), class)$ **then**
  $val(task) := val(exp)$
  proceed

A.5 Java$_{\mathcal{E}}$: Adding Exceptions

if *task* is `throw` *exp*; **then** (Throw)
  if $val(exp) \neq null$ **then**
    $mode := Throw(val(exp))$
    abrupt
  else fail(`NullPointerException`)

fail(*class*) $\equiv$ `newInstance`(*class*, $mode := Throw(ref)$
  abrupt)

if $task$ is (`catch` $(c_0\ v_0)\ b_0 \ldots$ `catch` $(c_n\ v_n)\ b_n$) **then** (Catch)
  if $mode = Throw(exc) \wedge \exists\, i : 0 \leq i \leq n : catches(c_i)$ **then**
    $loc(v_k) := exc$
    $mode := undef$
    $task := fst(b_k)$
    where $k = \iota i : 0 \leq i \leq n : catches(c_i) \wedge \forall j : 0 \leq j < i : \neg catches(c_j)$
  else abrupt
  where $catches(class) = compatible(classOf(exc), class)$

if $task$ is `finished` $\wedge\ mode = Throw(exc) \wedge length(task_{\mathcal{O}}) > 1$ **then** (Throw')
  frames := throw(frames)
  throw$(\langle _, inv\rangle\ tasks, \langle _\rangle\ vals, \langle _\rangle\ locs) \equiv (\langle up(inv)\rangle\ tasks, vals, locs)$

if $task$ is `finally` $block$ **then** (Finally)
  $finally := \langle mode\rangle finally$
  $mode := undef$
  $task := fst(block)$

if $task$ is `endfinally` **then** (Endfinally)
  $finally := finally'$
  if $mode = undef \wedge mode' = undef$ **then** proceed
  elseif $mode = undef \wedge mode' \neq undef$ **then** $mode := mode'$
    abrupt
  else abrupt
  where $\langle mode'\rangle finally' = finally$

if $task$ is $(exp_1 \otimes exp_2) \wedge \otimes \in \{/, \%\} \wedge val(exp_2) = 0$ **then** (BinaryExp)
  fail(`ArithmeticException`)

if ($task$ is $exp.fieldspec \vee task$ is $exp.fieldspec = exp_2 \vee$ (ITarget)
  $task$ is $exp.methodspec\{kind\}(exp_1, \ldots, exp_n)) \wedge$
  $val(exp) = null$ **then**
  fail(`NullPointerException`)

if $task$ is $(type)exp \wedge$ (Cast)
  $val(exp) \neq null \wedge \neg compatible(classOf(val(exp)), class)$ **then**
    fail(`ClassCastException`)

A.6 Java$_{\mathcal{T}}$: Adding Threads

if $task$ is exp.`start`() $\wedge\ val(exp) \neq null$ **then** (Start)
  if $exec(val(exp)) = undef$ **then**
    newFrames := start$(fst(body))$
    $exec(val(exp)) := Runnable$
    proceed
  else fail(`IllegalThreadStateException`)
  where $((), body) = instMethod(\texttt{run}, classOf(val(exp)), Virtual)$
    newFrames $\equiv (task_{\mathcal{T}}(val(exp)), val_{\mathcal{T}}(val(exp)), loc_{\mathcal{T}}(val(exp)))$

$$\text{start}(phrase) \equiv (\langle phrase\rangle, \langle\emptyset\rangle, \langle\emptyset\rangle)$$

if $task$ is `finished` $\wedge$ $length(task_{\mathcal{O}}) = 1 \wedge exec(self) = Runnable$ **then** (Terminate)
  $exec(self) := Exiting$

if $task$ is `synchronized` (exp) $block$ **then** (Synchronize)
  if $val(exp) \neq null$ **then**
    lock$(val(exp), task := fst(block))$
  else fail(`NullPointerException`)

if $task$ is `endsynchronized` **then** (Endsynchronize)
  unlock$(\, top(owns(self)),$
    if $mode = undef$ **then** proceed
    else abrupt$)$

lock$(ref, updates) \equiv$
  if $ref \in owns(self) \vee (locks(ref) \in \{0, undef\} \wedge competing(ref) = self)$ **then**
    $owns(self) := \langle ref\rangle \ owns(self)$
    $locks(ref) := locks(ref) + 1$
    $updates$
unlock$(ref, updates) \equiv$
  $owns(self) := pop(owns(self))$
  $locks(ref) := locks(ref) - 1$
  $updates$

if $task$ is $(exp.$ `wait` $())$ $\wedge$ $val(exp) \neq null$ **then** (Wait)
  case $exec(self)$ **of**
    $Runnable$,
    $Exiting$: **if** $val(exp) \in owns(self)$ **then**
      $backup(self) := exec(self)$
      enable$(val(exp), self, Blocked)$
    else fail(`IllegalMonitorStateException`)
    $Notified$: reenable$(val(exp), self, backup(self),$ proceed$)$

if $task$ is $(exp.$ `notify` $())$ $\wedge$ $val(exp) \neq null$ **then** (Notify)
  if $val(exp) \in owns(self)$ **then**
    if $waitSet(val(exp)) \neq \emptyset$ **then**
      awake$(val(exp), choose_{notify}(waitSet(val(exp))), Notified)$
    proceed
  else fail(`IllegalMonitorStateException`)

enable$(ref, thread, state) \equiv$ $waitSet(ref) := waitSet(ref) \cup \{thread\}$
  $locks(ref) := 0$
  $exec(thread) := state$

awake$(ref, thread, state) \equiv$ $waitSet(ref) := waitSet(ref) - \{thread\}$
  $exec(thread) := state$

reenable$(ref, thread,$
  $state, updates) \equiv$ **if** $locks(ref) \in \{0, undef\} \wedge competing(ref) = thread$ **then**
    $locks(ref) := occs(owns(thread), ref)$
    $exec(thread) := state$
    $updates$

if *task* is *exp*.`stop`() ∧ *val*(*exp*) ≠ *null* **then** (Stop)
  if *exec*(*val*(*exp*)) ≠ *Exiting* **then**
    stopped(*val*(*exp*)) := *Yes*
  proceed

if *stopped*(*self*) **then** (Stopped)
  case *exec*(*self*) **of**
  Runnable : stop
  Blocked : awake(*val*(*exp*), *self*, *Notified*)
  Notified : reenable(*val*(*exp*), *self*, *Exiting*, stop)
  where *exp*. `wait` () = *task*

stop ≡ *stopped*(*self*) := *undef*
  exec(*self*) := *Exiting*
  fail(`ThreadDeath`)

A.7 Initialization

if (*task* is (*class*, *field*) ∨ *task* is (*class*, *field*) = *exp* ∨ (FirstActiveUse)
  task is ((*class*, *method*, *fcty*)(*exp*$_1$, . . . , *exp*$_n$)) ∨
  task is `new` *class*)) ∧ ¬initialized(*class*) **then**
  frames := initialize(*class*, frames)

initialize(*class*, *frames*) ≡ invoke(⟨⟩, ⟨⟩, *fst*(*classInit*(*class*)), *frames*)

if *task* is `static` *block* **then** (Static)
  case *init*(currClass) **of**
  undef : *initThread*(*class*) := *self*
    init(currClass) := *InProgress*
    enter
  InProgress : *backup*(*self*) := *exec*(*self*)
    enable(*classRef*(currClass), *self*, *Blocked*)
  Done : exit
  Error : fail(`NoClassDefFoundError`)

enter ≡ **if** *supers*(currClass) ≠ ∅ ∧ ¬initialized(*super*(currClass)) **then**
    frames := initialize(*super*(currClass), frames)
  else *task* := *fst*(*block*)
exit ≡ **if** *invoker* = `static` *block* **then**
    frames := return(frames)
  else frames := goBack(frames)
  where ⟨_, *invoker*⟩_ = *task*$_\mathcal{O}$
goBack(⟨_⟩ *tasks*, ⟨_⟩ *vals*, ⟨_⟩ *locs*) ≡ (*tasks*, *vals*, *locs*)
currClass ≡ *classScope*(*task*)

$$\begin{aligned} task \underline{\text{ is }} phrase \equiv\ & task \text{ is } phrase \wedge \\ & (classRef(\text{currClass}) \in owns(self) \vee \\ & (locks(classRef(\text{currClass})) \in \{0, undef\} \wedge \\ & competing(classRef(\text{currClass})) = self)) \end{aligned}$$

if *task* is `endstatic` **then** (Endstatic)
  vary *thread* **over** *waitset*(*classRef*(currClass))
    awake(*classRef*(currClass), *thread*, *backup*(*thread*))
  if *mode* = *undef* **then**
    init(currClass) := *Done*
    exit
  else
    init(currClass) := *Error*
    if *compatible*(*classOf*(*ref*), `Error`) **then**
      frames := throw(frames)
    else fail(`ExceptionInInitializerError`)
    where *Throw*(*ref*) = *mode*