Invited paper

A formal dynamic semantics of Java: an essential ingredient of Java security

Mourad Debbabi, Nadia Tawbi, and Hamdi Yahyaoui

Abstract — Security is becoming a major issue in our highly networked and computerized era. Malicious code detection is an essential step towards securing the execution of applications in a highly inter-connected context. In this paper, we present a formal definition of Java dynamic semantics. This semantics has been used as a basis to develop efficient, rigorous and provably correct static analysis tools and a certifying compiler aimed to detect and prevent the presence of malicious code in Java applications. We propose a small step operational semantics of a large subset for Java. The latter includes features that have not been completely addressed in the related work or addressed in another semantics style. We provide a fully-fledged semantic handling of exceptions, reachable statements, modifiers and class initialization.

Keywords — security, static analysis, certifying compilers, Java, dynamic semantics, operational semantics, small step semantics.

1. Motivation

Security vulnerabilities occur at two levels, the communication level and the application level. At the communication level the use of cryptographic protocols aims at protecting from security breaches. This issue is being actively studied in order to ensure the absence of flaws in such protocols [1, 7, 11–13]. At the application level, malicious code could be inserted in applications without the consent nor the knowledge of end users. This malicious code can cause data corruption, data divulgation to non authorized users, an extensive use of system resources leading to denial of service, etc. Existing techniques to detect such a code are ad-hoc techniques based on merely syntactic analysis to detect the so-called virus signatures. They can only be effective in the detection of well known and cataloged malicious code. In our research group, we explore three approaches to address malicious code detection: dynamic analysis of code, static analysis of code and self certifying compilation. These approaches are described in [3, 4, 9, 10]. The main idea underlying these approaches is the use of language technology in order to address security issues. In order to make our analysis reliable we base all our techniques on formal and rigorous foundations. For this purpose we elaborated a static semantics for a large subset of the Java language [8]. We present in this paper a dynamic operational small-step semantics for the same Java language subset.

Lately, a surge of interest has been expressed in the elaboration of semantic foundations for Java. This interest is not only motivated by popular appeal and fashion considerations. Indeed, Java has a very sophisticated and subtle semantics as we will exemplify in the sequel. Moreover, Java is meant to be widely used in safety-critical embedded systems. Furthermore, Java support for mobile code through applets poses severe, and very interesting, challenges to the currently established language technologies in terms of security. All these factors justify the need for robust theoretical foundations for Java.

The Java language is, certainly, innovative, but still immature and unstable. Several modifications were made to its description, and errors are still present in its implementations. This is understandable, since the language combines attractive features, which makes its semantics far from being straightforward and leads to substantial complexity.

The only available official specification of Java [15] is an informal description that is subject to different interpretations. Besides, it is rather ambiguous, incomplete and sometimes not consistent with the behavior of the Java compilers. This is not acceptable mainly for the properties that have a direct impact on the security.

We believe that the theoretical investigations of Java semantics are very useful to clarify, correct and complete its semantics description. It will lead, without any doubts, to a better understanding of the language, to a more efficient, safe and secure execution. We strongly believe that a semantics theory for Java is not a luxury but rather a necessity.

A static semantics description has been elaborated in our research group and presented in [8]. In the present paper, we present a dynamic semantics for the same subset.

We believe that the operational semantics style is easy to understand and to manipulate. Actually, the operational style does not require complex mathematical tools which would increase the difficulty to understand the Java language.

Our ultimate goal is to provide a complete formal and easy to use description of the semantic aspects of Java. This will include static as well as dynamic semantics. Another goal we would like to achieve is to prove the subject reduction which guarantees the correctness of our static and dynamic semantic descriptions. This would also be a guarantee that the language is correctly designed i.e. the program behavior is consistent with the typing specification. On the other hand the two semantics could be used as guidelines to ensure a correct design of the Java Virtual Machine (JVM).

The paper is structured as follows. Related work is depicted in Section 2. An evaluation of the semantic issues related to the Java language specification is given in Section 3. A short overview of the Java dynamic semantics is presented in Section 4. Some concluding remarks are ultimately sketched in Section 5 together with some directions for the future work. The syntax of the language and the whole semantic rules set are given in an Appendix.

2. Related work

Many investigations for studying the Java language yielded very interesting results despite the restrictions that have been adopted.

In a pioneering exploration of Java formal semantics, Drossopoulou *et al.* [14] have studied a subset of Java that includes many features like hiding, overloading and exceptions. They proposed an operational semantics for this subset. In order to formalize the evaluation of exceptions, it is not obvious to define rules that could represent the control flow discontinuity which occurs when an exception is raised. The solution proposed by the authors is based on the notion of context. A context encompasses all the enclosing terms up to the nearest enclosing try-catch or try-catch-finally clause i.e. up to the first possible position at which the exception might be handled. Other important features like modifiers and initialization still need to be formalized. Among the assumptions used in [14], the execution of a return statement always terminates the execution of a method. Actually, sometimes we need to execute the enclosing finally clauses before returning to the caller. This adds significant complexity when exceptions are considered in the semantics formalization.

Syme [22] has studied a similar subset, except that he has included, in addition, the local variables. He used the theorem prover *Declare* in order to validate the elaborated operational semantics. The validation consists in proving the soundness of the dynamic semantics w.r.t. the static semantics which he has elaborated. In this work the try-catch, statement is not considered.

Tobias and Oheimb [18] have designed an operational semantics for a subset of Java called *Bali*. They adopted a big-step natural semantics style for elaborating this semantics. Many features of the Java language are considered such as exceptions, local variables, etc. However, the authors did not describe all the possibilities when handling the finally clause. For instance, they did not consider the case where a return statement occurs before the finally clause is evaluated.

Boerger and Sculte [5] have elaborated a dynamic semantics of Java by providing an ASM (abstract state machine) that interprets arbitrary Java programs. They have considered a subset of Java including initialization, exceptions and threads. They have exhibited some weaknesses in the initialization process as far as the threads are used. They pointed out that deadlocks could occur in such a situation.

One of the related work covering almost all Java language is the work of Alves-Foss *et al.* [2]. Their semantics covers the full range of this language excluding concurrency and the Java APIs. This semantics does not address the modifiers. Indeed, the evaluation of a field access expression does not show the modifiers role. Another interesting work is [16]. In this work the author presents a full treatment of the exception mechanism. He uses coalgebras in order to formalize the exception semantics in Java. In [23] the author extends the work done within the *Bali* project to cover exception handling and class initialization. The extension is elaborated in an axiomatic approach. Cenciarelli *et al.* [6] have presented an operational semantics for a significant subset of Java including threads. The major goal of their work is to deal with shared memory.

The operational small step style [19] we have adopted to formalize the dynamic semantics is easy to manipulate and to understand than the denotational and axiomatic styles. It is mandatory to understand the semantics of a language in order to design reliable applications. A precise, formal and easy to understand semantics specification helps to achieve this goal. In our work, we put to the treatment of the exceptions, the class initialization and the modifiers.

3. Semantic issues

The elaboration of a dynamic semantics for Java is a complex task. This complexity is due to many semantic issues related to the Java language. In the sequel, we highlight some of these issues.

3.1. Specification evaluation

The first task when elaborating a formal semantics for any language is to understand the existent informal specification of this language and to evaluate it in order to check whether it is consistent w.r.t. existing compiler reference implementations. Hence, we have started this work by the evaluation of the Java language as it is officially specified in [15]. We have discovered an inconsistency between the aforementioned specification and the JDK 1.3.0 compiler under Linux Redhat 6.2(build Linux_JDK_1.3.0_FCS). It concerns the class initialization process when triggered by a field access expression. Indeed, in the last clarification published by SUN Microsystems in [17], null is considered as a constant expression so every static and final field that is initialized to null is considered as a constant field [15] and cannot trigger class initialization. Actually, access to such a field causes the initialization of the class in which this field is declared.

This inconsistency pointed out that it is very hard to grasp all the subtleties of Java semantics. This is especially crucial when designing a compiler for the language.

One can draw two conclusions. First, the language designers would gain if they adopt a less complex semantics. Second, it is essential to have a non ambiguous specification of the language which is easily understandable.

Table 1
Exception handling

finally

$$\frac{\Gamma \vdash (\xi, \mathcal{F}, h, Block) \rightarrow (\xi, \mathcal{F}, h') \quad \mathcal{P} = \mathsf{FinalPosition}(Block) \quad \mathcal{F}.ReturnValue = \perp}{\Gamma \vdash (\xi, \mathcal{F}, h, \texttt{finally}\ \ Block) \rightarrow (\xi, \mathcal{F}, h', \texttt{throw}\ \ \xi, \mathcal{P})}$$

$$\frac{\Gamma \vdash (\mathcal{F}, h, Block) \rightarrow (\mathcal{F}, h') \quad \mathcal{P} = \mathsf{FinalPosition}(Block) \quad \mathcal{F}.ReturnValue \neq \perp \quad \neg\mathsf{HandlerInTable}(Any, \mathcal{F}.Method.ExceptionTable, H, \mathcal{P})}{\Gamma \vdash (\mathcal{F}, h, \texttt{finally}\ \ Block) \rightarrow (\mathcal{F}.PreviousFrame, h', \mathcal{F}.ReturnValue)}$$

Table 2
Return statement evaluation

ReturnStatement

$$\frac{\neg\mathsf{HandlerInTable}(Any, \mathcal{F}.Method.ExceptionTable, H, \mathcal{P})}{\Gamma \vdash (\xi, \mathcal{F}, h, \texttt{return}\ v, \mathcal{P}) \rightarrow (\mathcal{F}.PreviousFrame, h, v)}$$

3.2. Java semantics is complex

Java is a real-life language that comes with many convenient features that make its use appealing for the users. The elaboration of a formal semantics for any significant subset of it would be a complex task. The mechanisms underlying some of the interesting features of Java contribute to the complexity of the language semantics. As an example of such mechanisms we can cite the exceptions and the class initialization.

We give, in the sequel, details of the main difficulties or subtleties we encountered while elaborating our dynamic semantics.

The exception mechanism in Java has more complex semantics than other exception mechanisms in other languages due to the finally construct it offers. Actually, a try-catch statement in Java is designed for handling the exceptions that can occur during execution. The try clause contains a block of statements that can raise exceptions. A catch clause can handle an exception and then the execution continues normally. A finally clause may appear in the try statement. This clause is executed whether an exception has occurred or not. A try clause can be enclosed in another one making the semantics more complex especially when a return could occur.

The semantics of jumps and exceptions are usually elaborated using continuation-based techniques [20, 21] in order to preserve the compositionality of the semantic description. Compositionality might be affected by the discontinuity of the control flow caused by jumps or exceptions. Hence a special care is needed.

A continuation models the rest of the code to be executed and is used as a parameter in the semantic functions (denotational framework) or in the semantic rules (operational framework). For the sake of clarity, we use in our semantic rules exception tables that help to compute the needed continuation[1].

Table 1 shows two of the most important rules for the evaluation of the finally clause. A finally clause can be evaluated in the context of an exception ξ. If this clause does not raise any exception and if there is no return statement executed yet then it re-throws the exception ξ. Another interesting rule shows how the execution must return to the caller method after a finally clause is executed. Actually, if there is a return that has been executed and if there is no

[1] The interested reader can refer to the Appendix for more details.

enclosing finally (the predicate HandlerlnTable is false), the execution must return to the calling method after returning the resulting value of the return statement evaluation. This is specified in the second rule of Table 1. The previous discussion shows how complex is the handling of a finally clause. In fact its semantics depends on whether a return occurs or not, whether there is another enclosing try or not and whether there is a current raised exception or not.
The rules corresponding to the try and catch clauses evaluation can be found in the Appendix.

3.3. Staged semantics elaboration and its adequacy

Elaborating a dynamic semantics for a reduced subset of Java cannot provide a full understanding of the specifications. When this subset is extended to include the omitted features, a revision of the established rules is often necessary. The dependence of a construct semantics on other constructs makes a Java staged semantics an inadequate strategy. For example, to formalize the semantics of a return statement, it is unavoidable to take in consideration the finally statement as specified in [15]. The rule of the Table 2 shows how the semantics of a return depends on the existence of an enclosing finally. In fact, if there is no enclosing finally to this return the execution returns to the calling method. The statements that appear after the return statement are not necessarily unreachable.

3.4. A deep understanding of the static semantics

Another complexity that we have encountered is that one cannot elaborate a dynamic semantics without understanding the static one. The class initialization illustrates such dependency. Actually, a field access expression can trigger the class initialization if the field is not constant. A constant field is a final, static and initialized by a constant expression at compile time. This information is defined by the static semantics.
Another example concerns the method invocation semantics. In fact, to determine the actual invoked method, a dynamic search process is needed. This search process is based on the method signature which is determined by the static semantics.

4. Short overview of the Java dynamic semantics

In the sequel, we present some aspects of the Java dynamic semantics that we have elaborated.

4.1. Grammar of the Java subset

The syntax of the Java subset that we have considered is given in Tables 13, 14, 15 and 16 of the Appendix. This syntax has been defined in [8]. Notice that this subset is a large one.

4.2. Environment

A dynamic environment is denoted Γ. It consists of a set of class file representations. Each class or interface representation is composed of a set of fields including methods, fields and the *ConstantPool* of the classfile. The complete description of this environment is given in Tables 3 and 4.

4.3. Annotations

The execution of a Java program requires that the compiler adds some relevant information. For instance, a field access would be annotated with the descriptor of this field and the class where it is defined.
Hereafter, we describe the annotations that have been added in the corresponding cases.

Field access expression. Each field access expression *e.f* is annotated with C, the class where this field f is declared, and D its type (descriptor in the Java terminology).

Method invocation. Each method invocation $e.m()$ is annotated with D the method descriptor[2] and C the class where it is declared.

Constructor invocation. An explicit constructor invocation or new instance creation expression is annotated with its descriptor D.

Annotated syntax. We present in Table 5 the syntax annotations which correspond to what a Java compiler generates. The actual syntax that we adopt while elaborating the semantics rules is actually the previously described syntax in which the annotations in the Table 5 are assumed to be propagated.

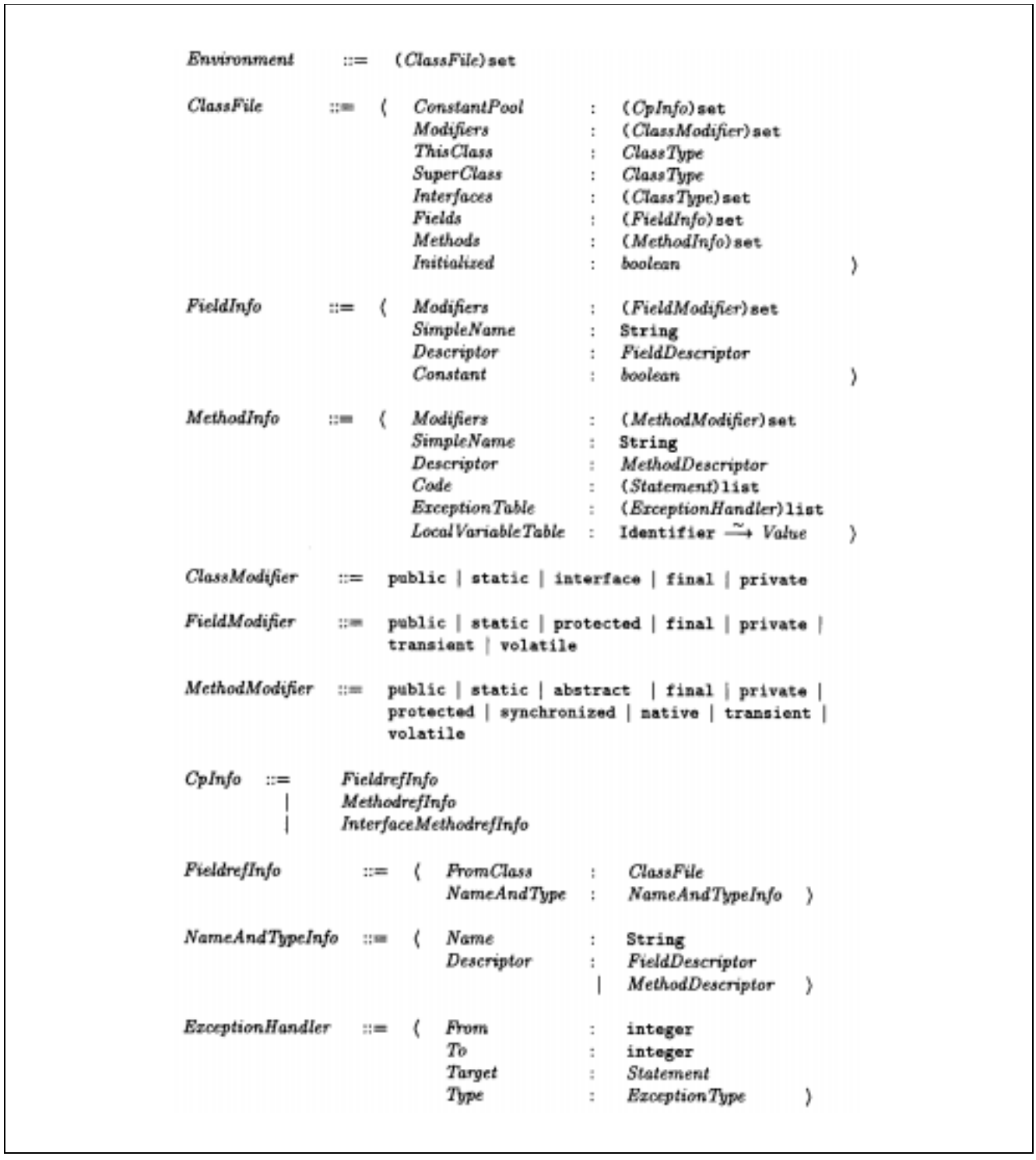
4.4. Notations

Notations:

- Given two sets A and B, $A \xrightarrow{\sim} B$ denotes the set of all maps from A to B. A map $m \in A \xrightarrow{\sim} B$ could be defined by extension as $[a_0 \mapsto b_0 \ \ldots \ a_{n-1} \mapsto b_{n-1}]$ to denote the association of the elements b_i's to a_i's, $a_i \in A$ et $b_i \in B$.
- $dom(m)$ denotes the domain of the map m. Given two maps m and m', we will write $m \dagger m'$ the overwriting of the map m by the associations of the map m' i.e. the domain of $m \dagger m'$ is $dom(m) \cup dom(m')$ and we have $(m \dagger m')(a) = m'(a)$ if $a \in dom(m')$ and $m(a)$ otherwise.
- $s \oplus s'$ denotes the disjoint union of the two sets s and s'.
- $S[f \leftarrow v]$ denotes the assignment of the value v to the field f of the structure S.
- ϵ denotes an empty value, cf Section 4.6.

[2]Descriptor stands for signature in Java terminology.

Table 3
Environment – Part 1



Environment ::= (*ClassFile*) `set`

ClassFile ::= ⟨ *ConstantPool* : (*CpInfo*) `set`
Modifiers : (*ClassModifier*) `set`
ThisClass : *ClassType*
SuperClass : *ClassType*
Interfaces : (*ClassType*) `set`
Fields : (*FieldInfo*) `set`
Methods : (*MethodInfo*) `set`
Initialized : `boolean` ⟩

FieldInfo ::= ⟨ *Modifiers* : (*FieldModifier*) `set`
SimpleName : `String`
Descriptor : *FieldDescriptor*
Constant : `boolean` ⟩

MethodInfo ::= ⟨ *Modifiers* : (*MethodModifier*) `set`
SimpleName : `String`
Descriptor : *MethodDescriptor*
Code : (*Statement*) `list`
ExceptionTable : (*ExceptionHandler*) `list`
LocalVariableTable : `Identifier` $\xrightarrow{\sim}$ *Value* ⟩

ClassModifier ::= `public | static | interface | final | private`

FieldModifier ::= `public | static | protected | final | private | transient | volatile`

MethodModifier ::= `public | static | abstract | final | private | protected | synchronized | native | transient | volatile`

CpInfo ::= *FieldrefInfo*
| *MethodrefInfo*
| *InterfaceMethodrefInfo*

FieldrefInfo ::= ⟨ *FromClass* : *ClassFile*
NameAndType : *NameAndTypeInfo* ⟩

NameAndTypeInfo ::= ⟨ *Name* : `String`
Descriptor : *FieldDescriptor*
| *MethodDescriptor* ⟩

ExceptionHandler ::= ⟨ *From* : `integer`
To : `integer`
Target : *Statement*
Type : *ExceptionType* ⟩

Table 4
Environment – Part 2

```
ExceptionType      ::= ClassFile
                     | Any

Any                ::= ClassType
                     | ArrayType

FieldDescriptor    ::= FieldType

FieldType          ::= PrimitiveType
                     | ClassType
                     | ArrayType

PrimitiveType      ::= byte    | char | double | float
                       integer | long | short  | boolean

ArrayType          ::= SimpleType[]

SimpleType         ::= PrimitiveType
                     | ClassOrInterfaceType

ClassOrInterfaceType ::= ClassType
                       | InterfaceType

ClassType          ::= ClassFile

InterfaceType      ::= ClassFile

MethodDescriptor   ::= (ParameterDescriptor) ReturnDescriptor

ParameterDescriptor ::= FieldType

ReturnDescriptor   ::= FieldType
                     | void
```

Table 5
Annotations

```
ExplicitConsInvocation ::= [D] this ( Argument ) ;
                         | [D] super( Argument ) ;
                         | ε

ClassInstanceCreation  ::= [D] new ClassType ( Argument )

SimpleFieldAccess      ::= Primary . [C, D] Identifier
                         | super . [C, D] Identifier
                         | [C, D]FieldName

MethodInvocation       ::= [C, D]MethodName ( Argument)
                         | Primary .[C, D] Identifier ( Argument )
                         | super .[C, D] Identifier ( Argument )
```

Table 6
Configurations

$$
\begin{array}{lll}
\textit{Configuration} & ::= & (\textit{ExceptionObject option}, \textit{Frame}, \textit{heap}, \textit{IntermediateTerm}, \\
 & & \quad \textit{Position option}) \\
 & & \\
\textit{Frame} & ::= & \langle \quad \texttt{this} : \textit{RefValue} \\
 & & \qquad \textit{Class} : \textit{ClassFile} \\
 & & \qquad \textit{Method} : \textit{MethodInfo} \\
 & & \qquad \textit{PreviousFrame} : \textit{Frame} \\
 & & \qquad \textit{ReturnAddress} : \texttt{int32} \\
 & & \qquad \textit{ReturnValue} : \textit{Value} \ \rangle \\
 & & \\
\textit{heap} & : & \textit{FieldRecord} \xrightarrow{\sim} \textit{Value} \\
 & \cup & \texttt{address} \xrightarrow{\sim} (\textit{ClassFile}, \textit{MF}) \mid (\textit{ClassFile}, \textit{MI}) \\
 & & \\
\textit{MF} & : & \textit{FieldRecord} \xrightarrow{\sim} \textit{Value} \\
 & & \\
\textit{MI} & : & \texttt{int32} \xrightarrow{\sim} \textit{Value} \\
 & & \\
\textit{FieldRecord} & ::= & \langle \quad \textit{Field} : \textit{FieldInfo} \\
 & & \qquad \textit{FromClass} : \textit{ClassFile} \ \rangle \\
 & & \\
\textit{ExceptionObject} & ::= & \texttt{address} \\
 & & \\
\textit{Position} & ::= & \texttt{int32}
\end{array}
$$

Table 7
Computable values

$$
\begin{array}{lll}
\textit{Value} & ::= & \textit{PrimValue} \\
 & \mid & \textit{RefValue} \\
 & \mid & \texttt{null} \\
 & \mid & \bot \\
 & \mid & (\textit{Value})\texttt{set} \\
 & & \\
\textit{PrimValue} & ::= & \texttt{byte8} \oplus \texttt{short16} \oplus \texttt{int32} \oplus \texttt{long64} \\
 & & \oplus \texttt{char16} \oplus \texttt{iee32} \oplus \texttt{iee64} \oplus \textit{boolean} \\
 & & \\
\textit{RefValue} & ::= & \texttt{address} \\
 & & \\
\textit{boolean} & ::= & \texttt{true} \oplus \texttt{false}
\end{array}
$$

Table 8
Semantic categories

Γ	$\in$	*Environment*	..... Environment of class-files
$\mathcal{F}$	$\in$	*Frame*	Frame representing an invoked method
ms	$\in$	*(modifiers)set*	..... Modifiers
ξ	$\in$	*RefValue*	..... Exception object
h	$\in$	*heap*	..... Memory
ρ	$\in$	**address**	..... Some address in memory
v	$\in$	*Value*	..... Computable value
M	$\in$	*MethodInfo*	..... Structure representing a method
F	$\in$	*FieldRecord*	..... Structure representing a field
f	$\in$	*FieldInfo*	..... Field in some class
$\mathcal{P}$	$\in$	**integer**	.... Number associated to a statement

4.5. Intermediate terms

Since we adopted a small step style, we have to represent intermediate results during the evaluation process. These intermediate results are formalized as algebra terms and are denoted intermediate terms. Actually, *intermediate terms* consist of terms that involve syntactic entities as well as computable values. For instance, let $T[e]$ be a Java expression where T is an array and e is a Java expression. The evaluation of this expression yields as an intermediate result $T[v]$ where v stands for a computable value representing the result of the evaluation of e. Clearly, this intermediate result does not belong to the Java syntax because the integer values are different from the integer constants that would appear in a Java source code.

4.6. Configurations

A configuration is a tuple $(\xi, \mathcal{F}, h, t)$ where ξ is the object exception that is thrown and not yet handled, $\mathcal{F}$ is the frame of the current method, h is the global memory and t is an intermediate term. For the sake of convenience, this configuration may appear as $(\mathcal{F}, h, t)$ when no exception is thrown and not handled. Actually, this is an abbreviation of $(\xi, \mathcal{F}, h, t)$ where ξ is empty, denoting the absence of exceptions. The configuration may also appear as $(\mathcal{F}, h)$ where there is no thrown-exception after evaluating the term t and this term is fully evaluated. The Table 6 shows the configurations used in our semantics. In our configurations the notation ξ^+ represent either an exception ξ or the absence of a raised exception denoted by ϵ.

4.7. Computable values

The evaluation of the syntactic constructs of a language using a formal semantics produces values that are commonly denoted *computable values*. We define in Table 7 the computable values manipulated by our dynamic semantics.

A computable value can be a primitive value or a reference (memory address). We introduce the undefined value ($\perp$) which is used as:

- the value of this in a static method,
- the value of the field *ReturnValue* in the current frame if no return statement has been executed yet.

4.8. Memory abstraction

We abstract the memory by a map h which associates a computable value to a *RefValue* (cf Table 7) or to a *FieldRecord* which represents a class field (cf Table 6). In the map h a *FieldRecord* always corresponds to a static field. Notice that a *RefValue* represents a reference to an object or to an array.
The following items describe how objects and arrays are represented in our memory abstraction:

- An object is represented by an ordered pair (*ClassType, MF*) where $MF = [F_0 \mapsto v_0, \ldots, F_{n-i} \mapsto v_{n-i}]$, *ClassType* is the concrete type of the object, F_i is a *FieldRecord* structure corresponding to a non static field and v_i represents the computable value associated to this field.
- An array T is represented by an ordered pair (*ClassType, MI*) where $MI = [0 \mapsto v_0, \ldots, n-1 \mapsto v_{n-1}]$, n is the dimension of the array and v_i is a computable value associated with $T[i]$.

4.9. Semantic categories

We define in Table 8 the semantic categories manipulated by our semantics.

4.10. Semantic rules

The evaluation process is formalized as a transformation of a configuration to a new one. We denote this transforma-

Table 9
Return statement evaluation

ReturnStatement

$$\frac{\mathsf{HandlerInTable}(Any, \mathcal{F}.Method.ExceptionTable, H, \mathcal{P}) \quad Statement = H.Target}{\Gamma \vdash (\xi, \mathcal{F}, h, \texttt{return } v, \mathcal{P}) \rightarrow (\mathcal{F}[ReturnValue \leftarrow v], h, Statement)}$$

$$\frac{\neg\mathsf{HandlerInTable}(Any, \mathcal{F}.Method.ExceptionTable, H, \mathcal{P})}{\Gamma \vdash (\xi, \mathcal{F}, h, \texttt{return } v, \mathcal{P}) \rightarrow (\mathcal{F}.PreviousFrame, h, v)}$$

Table 10
Exception handling

finally

$$\frac{\Gamma \vdash (\xi, \mathcal{F}, h, Block) \rightarrow (\xi, \mathcal{F}, h') \quad \mathcal{P} = \mathsf{FinalPosition}(Block) \quad \mathcal{F}.ReturnValue = \perp}{\Gamma \vdash (\xi, \mathcal{F}, h, \texttt{finally } Block) \rightarrow (\xi, \mathcal{F}, h', \texttt{throw } \xi, \mathcal{P})}$$

$$\frac{\Gamma \vdash (\mathcal{F}, h, Block) \rightarrow (\mathcal{F}, h') \quad \mathcal{P} = \mathsf{FinalPosition}(Block) \quad \mathcal{F}.ReturnValue \neq \perp \quad \neg\mathsf{HandlerInTable}(Any, \mathcal{F}.Method.ExceptionTable, H, \mathcal{P})}{\Gamma \vdash (\mathcal{F}, h, \texttt{finally } Block) \rightarrow (\mathcal{F}.PreviousFrame, h', \mathcal{F}.ReturnValue)}$$

tion by $\Gamma \vdash (\xi, \mathcal{F}, h, t) \rightarrow (\xi', \mathcal{F}', h', t')$ which says that an intermediate term t is evaluated under the exception ξ, in the frame $\mathcal{F}$. The result of this evaluation is the new intermediate term t'. The evaluation may modify the current memory h, may raise a new exception and finally may change the current method being evaluated. ξ' stands for the new exception, $\mathcal{F}'$ stands for the new frame corresponding to the new method and h' stands for the new memory. The operational semantics consists of a set of semantic rules. Each rule states that the evaluation in the conclusion part can be deduced from the evaluations in the premise part.

The complete set of the semantic rules is given in the Appendix. We give in the sequel the explanation of some relevant rules. The remaining rules in the Appendix should be understood in a similar way.

4.10.1. The return statement

The evaluation of a return statement is very subtle. Actually, after the execution of a return statement, every enclosing finally clause must be executed. A predicate HandlerInTable (cf Section A.3.5 in the Appendix) indicates if a finally clause exists in the exceptions table of the method represented by the frame $\mathcal{F}$. By the number associated to this return statement, we can get the first enclosing finally, if it exists, in the exceptions table of the method in $\mathcal{F}$ (cf Tables 26 and 27). *H. Target* represents this finally statement. The value returned by the evaluation is assigned to the field *ReturnValue* of the frame $\mathcal{F}$ representing the current method, the current exception is given up and the execution continues by handling the *Block* of the finally clause. This is specified in the first rule in Table 9.

When there is no enclosing finally clause, the execution continues in the calling method by returning the value of the return evaluation. This is specified in the second rule in Table 9.

4.10.2. Exceptions

Exception handling in Java is a highly designed mechanism that provides to developers the possibility to deal with abnormal situations without causing the execution abortion. Actually, the developer can control bad results and associate a specific code to handle such situations. A try statement in Java is designed for handling exceptions that can occur through execution. The try clause contains a block of

Table 11
Static method call evaluation

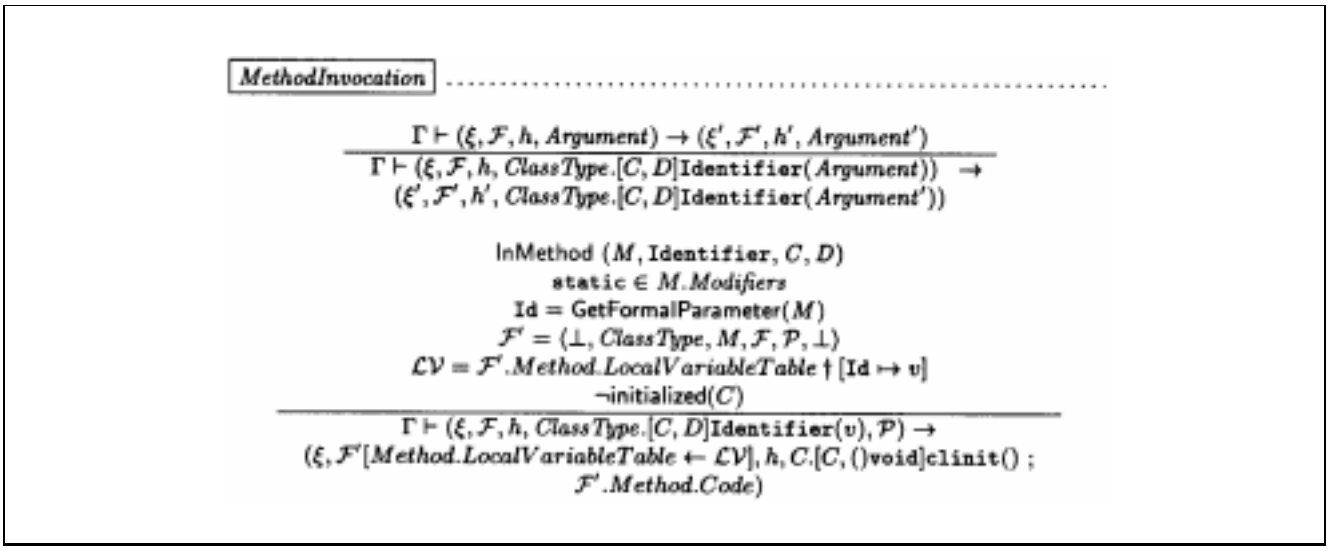
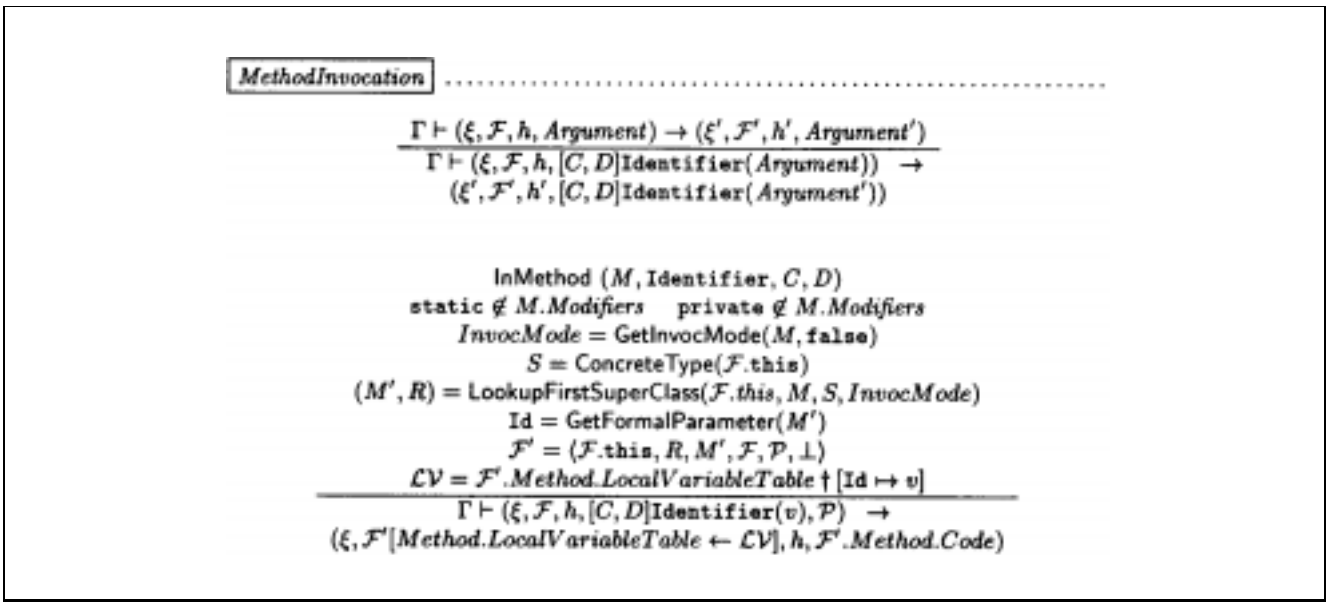


Table 12
Instance method call evaluation



statements that can raise exceptions. A catch clause can handle an exception and then the execution continues normally. A finally clause may appear in the try statement. This clause is executed whether an exception has occurred or not. It is considered as a clean code. We present two rules for the finally evaluation. The rules corresponding to the try and catch clauses evaluation can be found in the Appendix).

Table 10 shows two rules for the evaluation of the finally clause. A finally clause can be evaluated in the context of an exception ξ. If this clause does not raise any if exception and there is no return statement executed yet ($\mathcal{F}.ReturnValue = \perp$) then it re-throws the exception ξ at the position of the last statement of finally. The function finalPosition returns the number of the last statement of a block of statements. This is specified in the first rule in Table 10.

Another interesting rule shows how the execution must return to the caller method after a finally clause. Actually, if there is a return that has been executed ($\mathcal{F}.Returnvalue \neq \perp$) and if there is no enclosing finally (the predicate HandlerInTable evaluates to false), the execution must return to the caller method (represented by $\mathcal{F}.PreviousFrame$) after returning the resulting value of the return evaluation. This is specified in the second rule in Table 10.

4.10.3. Method invocation

Java uses the dynamic dispatch to determine which method is to be executed in each call site when an instance method is invoked. The search of the actual method is performed at run time. Actually, the JVM launches a search procedure to determine the method to be executed. On the other hand, if the invoked method is static or private there is no need to launch this search the actual method would be statically determined. The semantic rules corresponding to the method invocation represent this process as a first step of the evaluation.

After determining the method to be invoked, we proceed to the evaluation of the actual parameters. Let M be the invoked method and C the class where it is declared. If static belongs to the modifiers set of M, the evaluation of the invocation must trigger the initialization of the class C if it is not initialized yet. This is checked using the predicate initialized. If C is not initialized we call its clinit() method before evaluating the code of M. The code of M is evaluated under the substitution of the formal parameter by the value of the argument. A new frame is created and used along the evaluation of the method code. The semantic rules of evaluating a static method as well as an instance method are shown in Tables 11 and 12. If the method is an instance many operations are performed before executing its code. Let M be the instance method, C the class where it is declared as determined at compile time and D its descriptor. The first evaluation step, is the search of the actual method. For this, a predicate InMethod checks the existence of a method M having a descriptor D in the class C. The second evaluation step consists in searching for the actual method to be executed at this site. This depends on the type of the receiving object. The semantic function LookupFirstSuperClass performs this search. Notice that we consider methods with one parameter if any. We made this restriction only to seek more clarity of the rules. The generalization to more than one parameter could be easily performed. For more details cf Section A.3.13 in the Appendix.

5. Conclusion and future work

We discussed in this paper a dynamic semantics of a large subset of Java in which we have handled some subtle problems such as initialization, modifiers, and exceptions. The formalization has been carried out in an operational small step style. This makes it extendable to handle another aspects of the language such as the threads. On the other hand, this style is easily understood and manipulated without any heavy theoretical background. The whole semantics is detailed in the Appendix. We plan to extend this semantics to include packages, inner-classes and threads and ultimately to prove the consistency between this semantics and the static one described in [8].

On the other hand, this paper gives some insights into the task consisting in elaborating a Java dynamic semantics. We can sum up our conclusions as follows. Elaborating a dynamic semantics for Java should treat all features of the language not just a reduced subset of it. Hence, a staged semantics strategy would be inadequate. Any research effort that address the whole language or at least a realistic subset of it would be a worthwhile. We hope that SUN Microsystems simplifies some constructs semantics such as the exceptions.

This work contributes to build a formal foundations for our techniques and tools designed to address verification of Java applications properties related to security issues.

References

[1] K. Adi, M. Debbabi, and M. Mejri, "A logic for specifying security properties of electronic commerce protocols", in *Proceedings 8th International Conference on Algebraic Methodology and Software Technology, AMAST'2000*, *Lecture Notes in Computer Science*. Springer, 2000, vol. 1816, pp. 499–513 (also, accepted for publication in the *Int. J. Theor. Comput. Sci.*, Elsevier).

[2] J. Alves-Foss and F. S. Lam, "Dynamic denotational semantics of Java", in *Formal Syntax and Semantics of Java*, J. Alves-Foss, Ed., *LNCS*. Springer, 1999, pp. 201–240.

[3] J. Bergeron, M. Debbabi, J. Desharnais, M. Erhioui, Y. Lavoie, and N. Tawbi, "Static detection of malicious code in executable programs", in *Symp. Requir. Eng. Inform. Secur.*, May 2000.

[4] J. Bergeron, M. Debbabi, J. Desharnais, M. Erhioui, Y. Lavoie, and N. Tawbi, "Static detection of malicious code in executable programs", in *REJ Special Issue Requir. Eng. Inform. Secur.*, June 2001.

[5] E. Boerger and W. Schulte, "A programmer friendly modular definition of the semantics of Java", in *Formal Syntax and Semantics of Java*, J. Alves-Foss, Ed., *LNCS*. Springer, 1999, pp. 353–404.

[6] P. Cenciarelli, A. Knapp, B. Reus, and M. Wirsing, "An event-based operational semantics of multi-threaded Java", in *Formal Syntax and Semantics of Java*, J. Alves-Foss, Ed., *LNCS*. Springer, 1999, pp. 157–200.

[7] M. Debbabi, N. Durgin, M. Mejri, and J. Mitchell, "Security by typing", in *Int. J. Softw. Tools Technol. Transfer (STTT)*, 2001.

[8] M. Debbabi and M. Fourati, "Sur la sémantique statique de Java", in *LMO*, Montréal, Jan. 2000, pp. 167–182.

[9] M. Debbabi, E. Giasson, B. Ktari, F. Michaud, and N. Tawbi, "Secure self-certified code", in *Proc. IEEE 9th Int. Worksh. Enterpr. Secur. (WETICE'OO)*, National Institute of Standards and Technology (NIST), Maryland, USA, June 2000.

[10] M. Debbabi, M. Girard, L. Poulin, M. Salois, and N. Tawbi, "Dynamic monitoring of malicious activity in software systems", in *Symp. Requir. Eng. Inform. Secur.*, May 2000.

[11] M. Debbabi, M. Mejri, N. Tawbi, and I. Yahmadi, "A new algorithm for the automatic verification of authentication protocols", in *DIMACS Worksh. Des. Form. Verif. Secur. Protoc.*, Sept. 1997.

[12] M. Debbabi, M. Mejri, N. Tawbi, and I. Yahmadi, "Formal automatic verification of authentication cryptographic protocols", in *First IEEE Int. Conf. Form. Eng. Meth. (ICFEM'97)*, Nov. 1997.

[13] M. Debbabi, M. Mejri, N. Tawbi, and I. Yahmadi, "From protocol specifications to flaws and attack scenarios: an automatic and formal algorithm", in *Worksh. Enterpr. Secur. (WETICE'97)*, June 1997.

[14] S. Drossopoulou, T. Valkevych, and S. Einsenbach, "Java type soundness revisited". Tech. Rep., Imperial College of Science, Technology and Medicine, Oct. 1999.

[15] J. Gosling, B. Joy, and G. Steele, *The Java Language Specification*. Addison Wesley, 1996.

[16] B. Jacobs, "A formalization of Java's exception mechanism", in *Programming Languages and Systems*, D. Sands, Ed., *LNCS*. Springer, 2001, pp. 284–301.

[17] "SUN Microsystems. Clarifications and amendments to the Java language specification", 1998. [Online]. Available: http://java.sun.com/docs/books/jls/clarify.html

[18] T. Nipkov and D. Oheimb, "Machine-checking the Java specification: proving type-safety", in *Formal Syntax and Semantics of Java*, J. Alves-Foss, Ed., *LNCS*. Springer, 1999, pp. 119–156.

[19] G. D. Plotkin, "A structural approach to operational semantics". Tech. Rep. FN-19, DAIMI, University of Aarhus, Denmark, Sept. 1981.

[20] J. C. Reynolds, "The discoveries of continuations", in *Lisp and Symbolic Computation*. 1993, pp. 233–247.

[21] C. Strachey and C. P. Wadsworth, "Continuations a mathematical semantics for handling full jumps". Tech. Rep. PRG-11, Oxford University Computing Laboratory, Programming Research Group, Jan. 1974.

[22] D. Syme, "Proving Java type soundness", in *Formal Syntax and Semantics of Java*, J. Alves-Foss, Ed., *LNCS*. Springer, 1999, pp. 83–118.

[23] D. von Oheimb, "Axiomatic semantics for Java light in Isabelle/HOL", in "Formal techniques for Java programs", S. Drossopoulou, S. Eisenbach, B. Jacobs, G. T. Leavens, P. Müller, and A. Poetzsch-Heffter, Eds. Tech. Rep. 269, 5/2000, Fernuniversität Hagen, Fernuniversität Hagen, 2000. [Online]. ECOOP2000 Workshop Proc. available: http://www.informatik.fernuni-hagen.de/pi5/publications.html

Appendix
A full overview of the Java dynamic semantics

We present here a complete overview of the dynamic operational semantics that we have elaborated.

A.1. Grammar of the subset

The syntax of the Java subset that we have considered is given in Tables 13, 14, 15 and 16 (Tables 13–48, see this issue, pp. 97–119). This syntax has been defined in [8].

A.2. Hypothesis

The following hypothesis are assumed in our work. We present them together with the rationale underlying their assumption:

- Our semantics is able to evaluate syntactically correct programs. Furthermore, we assume that all the needed classes have been loaded and checked. We also assume that the reference resolution step has been performed correctly. These assumptions are essential. Actually, we do not formalize the dynamic linking process.
- We assume that the Java program is preprocessed so that each statement is identified by a number and each expression is tagged with some relevant annotations. These annotations are described in the sequel. These annotations correspond actually to what the compiler generates.
- For the sake of clarity, we assume that all the methods have only one argument. We also consider that all arrays are mono-dimensional. The generalization is obvious in both cases, but would unnecessarily make cumbersome the presentation.
- We consider that two methods have been added in each class, namely init() and clinit(). These methods have been added in order to express the initialization process as it is performed by the JVM.
  - init (*Argument*): this method represents the constructor code of the class and the initializers of the instance variables. *Argument* is the parameter of the constructor. It can be void.
  - clinit(): this method contains the class static code and the initializers of the static variables.

  It is clear that these two methods represent what actually is generated by the compiler.

A.3. Semantics rules

The evaluation process is formalized as a transformation of a configuration to a new one. We denote this transformation by $\Gamma \vdash (\xi, \mathcal{F}, h, t) \rightarrow (\xi', \mathcal{F}', h', t')$ which means that an intermediate term t is evaluated under the exception ξ, in the frame $\mathcal{F}$. The result of this evaluation is the new intermediate term t'. The evaluation may modify the current memory h, may raise a new exception and finally may change the current method being evaluated. ξ' stands for the new current exception, $\mathcal{F}'$ stands for the new frame corresponding to the new method and h' stands for the new memory.
The operational semantics consists of a set of semantic rules. Each rule states that the evaluation in the conclusion part can be deduced from the evaluations in the premise part.

A.3.1. Field declaration evaluation

The following remarks help the understanding of the field declaration evaluation rules.

- A *FieldDeclaration* expression is considered as a part of clinit() declaration or init() one depending on the modifiers of this field. If the field is static then its declaration will be included in the clinit() method otherwise it will be in the init() method.
- We represent each static field in the memory by a *FieldRecord* that contains the field itself (*FieldInfo*) and the class in which it is declared (*ClassFrom*).
- Each declaration of a static field adds to the memory the *FieldRecord* with its default value or the value resulting from the evaluation of the expression that initializes it.

- An instance field declaration is included in the init() method and will be executed when a new object is created.
- A field is considered as an instance field when static $\notin$ modifiers of this field.
- A *FieldRecord* is added to an object having this as its address.
- The concrete type of an object having ρ as an address is obtained by applying the function ConcreteType(ρ).

The methods clinit() and init() are defined in Tables 17 and 18.
In the field declaration evaluation rules, InField stands for a predicate which evaluates to true when the field represented by the first argument belongs to the class represented by the third argument and false otherwise. The second argument of the predicate InField stands for the simple name of the field. The fourth argument represents the type of the field.

InField : *FieldInfo* $\times$ Identifier $\times$ *ClassFile* $\times$
$\times$ *FieldDescriptor* $\rightarrow$ bool

$$\begin{aligned} \text{InField} \quad & (f, \text{Identifier}, C, D) \\ & (f \in C.Fields) \wedge \\ & (f.SimpleName = \text{Identifier}) \wedge \\ & (f.Descriptor = D). \end{aligned}$$

The rules of field declaration evaluation are presented in Tables 19 and 20.

A.3.2. Constructor evaluation

A constructor invocation (Table 21) is equivalent to the invocation of the method init() of the class that represents the concrete type of the newly created object. For example, an explicit constructor invocation like this (*Argument*) is annotated with its descriptor D as follows [D] this (*Argument*) is evaluated to this.[C, D|init(υ) where C is the concrete type of this and υ is the value of *Argument*.

A.3.3. ocal variable declaration expression evaluation

A *LocalVarDeclaration* expression (Table 22) adds a new variable to the local environment of the current method. An initialization of a local variable updates its value in the local variable table of the current method.

A.3.4. Statement evaluation

Statement evaluation:

- The statement if-then: first the condition of the if clause is evaluated. When its value is true, the then clause is executed otherwise the configuration is not modified.
- The statement if-then-else: first the condition is evaluated to produce υ as a value. If υ is true then the clause then is evaluated, otherwise the clause else is evaluated.
- The statement while: we evaluate first the condition, when its value is false the configuration does not change, otherwise the evaluation of this statement produces a statement if having the body of the while statement as its then clause.

The evaluations of the previous statements are presented in Tables 23 and 24.
The evaluation of the return statement is presented in Table 25.

A.3.5. Exception handling

We suppose that a preprocessing of each method is performed in order to associate numbers with statements. These numbers respect the textual order. The exceptions table indicates where the control has to flow (continuation) after each potential exception occurrence in a try-catch construct. For example, the exceptions Table 27 is associated with the piece of code in Table 26. The column *Target* represents the statement block of a catch that can handle an exception in the clause try which is thrown between statement 1 and statement i. The block of the finally clause is executed if the exception is thrown between statement 1 and n whether a catch has been executed or not, where n represents the number of the last statement in the last catch.
A throw statement raises an exception from the position where it appears. First *Argument*, the argument of the throw, is evaluated. It produces a reference value υ. If υ is null then a NullPointerException is raised at the same position as the throw statement. Otherwise, we must search a handler for the thrown exception E. This is formalized by the predicate HandlerlnTable($E, H, \mathscr{F}.Method.ExceptionTable, \mathscr{P}$) which evaluates to true if the first enclosing catch or the first enclosing finally exists in the exceptions table given as its third argument. This exceptions table is associated with the current frame $\mathscr{F}$. The predicate evaluates to false otherwise.
A try statement can have one of the following three forms: try-catch or try-finally or try-catch-finally.
A try clause is a guarded section where each abnormal execution will cause a jump to a catch which argument type is a supertype of the raised exception type.
A finally clause is known as a clean code that will be executed whether a previous exception has occurred or not. When a finally clause exists in a try statement, the program must execute this clause whether an exception occurred or not in the associated try and/or catch clauses of the same statement. Accordingly, we should formally state such a semantic constraint. In the exception table, we specify the column *ExceptionType* which contains the type of the thrown exception. We introduce the type *Any* representing every non-primitive type that can exist. Hence, if an

exception is thrown, it will verify the constraint of subtyping with *Any*. The example of Table 26 shows a try statement with a finally clause: When an exception is thrown at some position $\mathscr{P}$, the program execution continues at the first *Target* in the table that corresponds to a type in the column Type, which is a supertype of this exception. We define the predicate HandlerlnTable that checks the existence of the first catch or finally in the exceptions table of the method represented by the frame $\mathscr{F}$ i.e. the first range (*From-To*) that contains the position $\mathscr{P}$ where an exception E has occurred and having an exception type in the third column as a superclass of E (this relation is defined under the environment Γ). It returns the *Block* contained in the *Target* column. This *Block* represents the continuation of the execution after the occurrence of the exception. The predicate HandlerlnTable is defined as follows:

HandlerlnTable : $ExceptionType \times ExceptionTable \times$
$\times\ ExceptionHandler \times \text{Position} \rightarrow \text{bool}$

$$\begin{aligned}
&\text{HandlerlnTable}(E, \mathscr{F}.Method.ExceptionTable, H, \mathscr{P}) = \\
&(\mathscr{F}.Method.ExceptionTable = [] \Rightarrow \text{false}) \vee \\
&((H = \text{hd}(\mathscr{F}.Method.ExceptionTable)) \wedge \\
&(\mathscr{P} \geq H.From) \wedge \\
&(\mathscr{P} \leq H.To) \wedge \\
&((H.Type = Any) \vee ((E \sqsubseteq H.Type) \wedge (E \neq Any)))) \vee \\
&((\text{HandlerlnTable}(E, \text{tl}(\mathscr{F}.Method.ExceptionTable), \\
&H, \mathscr{P})) \wedge \\
&(\mathscr{P} < \text{hd}(\mathscr{F}.Method.ExceptionTable).From) \vee \\
&(\mathscr{P} > \text{hd}(\mathscr{F}.Method.ExceptionTable).To) \vee \\
&((\text{hd}(\mathscr{F}.Method.ExceptionTable).Type) \neq Any) \wedge \\
&((E \not\sqsubseteq (\text{hd}(\mathscr{F}.Method.ExceptionTable).Type)) \vee \\
&(E = Any)))
\end{aligned}$$

Let us explain now some rules from Tables 28, 29 and 30:

- A try clause can be executed without raising any exception. If there is an enclosing finally statement in the exceptions table, the execution will continue at the *Block* of this finally, otherwise it will continue normally. If this clause raises an exception, it will be equivalent to a throw statement at the position where the exception occurred.

- A catch clause is executed when it handles a thrown exception that has happened before (and not from another catch clauses in the same statement). So, no exception will be present in the configurations before executing this clause. The enclosing finally clause is then determined and the execution continues at the *Block* of this finally if it is found. When this catch clause raises an exception it will be equivalent to a throw statement at the position of the statement that has caused it.

- The finally clause is more complicated to formalize. In fact, there are two factors that influence the evaluation: first, a return statement (if it has been executed before this finally or inside it) and second, if there is an exception before executing it or caused by this clause itself. When the evaluation of this clause terminates normally and there is no return statement that has been executed ($\mathscr{F}.ReturnValue = \perp$ where $\mathscr{F}$ is the current frame) and no other enclosing finally (the predicate Handlerlntable evaluates to false), the execution continues normally. When a return has been executed before this finally and there is no thrown exception in the left configuration, we must go to the first enclosing finally if it exists. Otherwise the execution returns to the calling method. The execution of finally can itself raise an exception and it will be equivalent in this case to a throw. When a finally clause is executed under some exception and raises by itself another exception, it gives up the former and raises this new exception. If it does not cause another exception it removes the initial exception.

A.3.6. New array creation expression evaluation

An array creation expression returns a new reference to the created array. Each element of the array is initialized by its default value. A NegativeSizeException is raised when the array size is negative. The Table 31 shows the semantics of such an expression.

A.3.7. iteral, this and parenthesi ed expression evaluation

A literal is evaluated to its primitive value this is evaluated to the field this of the current frame. Evaluation of a parenthesized expression returns the value of the expression that is inside the parentheses. All these rules are formally stated in the Table 32.

A.3.8. New class instance creation expression evaluation

The Table 33 states how to evaluate a new instance class creation which triggers a call of the method init() and the initialization of this class (call of clinit() if it is not yet initialized). To obtain all the fields of some class C, we use the function Fields(C).

Fields : $ClassFile \rightarrow (FieldRecord)\text{set}$

$$\begin{aligned}
&\text{Fields}\ (C) = \\
&\quad \forall \quad F \in C.Fields. \\
&\qquad \text{if static} \notin F.Modifiers \\
&\qquad \text{then } \{\langle F, C\rangle\} \\
&\qquad \cup \\
&\qquad (\text{if } (C.ThisClass \neq \text{Object}) \\
&\qquad \text{then Fields}(C.SuperClass))
\end{aligned}$$

A.3.9. Cast expression evaluation *type Expression*

At run time, the JVM checks if the concrete type of the value υ of *Expression* evaluation (obtained by calling ConcreteType(υ)) is a subclass of *type*. If this constraint is not satisfied then the exception ClassCastException is thrown from the position of the nearest statement where

it exists. If no exception is thrown after the evaluation of such an expression, the value of the expression is returned as a result. This is presented in Table 34.

A.3.10. Field access expression evaluation

We evaluate the *FieldAccess* expression that appear in the righthand side of an assignment expression. So, each expression will return a value. The static fields that are not initialized with constant expressions, at compile time, can trigger the initialization of the class in which they are declared. In this case, before returning the value of the field, the method clinit() is called to initialize this class.
The rules of the field access expression evaluation are presented in the Tables 35, 36, 37 and 38.

A.3.11. Array field access evaluation

An access to an array component of the form *PrimaryNoNewArray*[*Expression*] can cause the NullPointerException if the reference to the array (value of *PrimaryNoNewArray*) is null. When the reference to this array is not null, the value of *Expression* must be a positive integer between 0 and the length of the array. Otherwise, an exception of type IndexOutOfBoundsException will be thrown. The rules of evaluating such an expression are given in the Table 39.

A.3.12. Simple local variable access evaluation

An access to a local variable returns its value from the local variable table of the current frame. The rule is presented in Table 40.

A.3.13. Method call evaluation

For a method invocation, there are many steps that are needed before to jump to the invoked method. First, the value of the receiver is computed. Then the argument is evaluated after which the accessibility to the invoked method is checked (we suppose that the method is accessible). Afterwards, the underlying method code is localized. Finally, a new frame is created to contain the information that is associated with the invoked method.

A.3.14. Computing receiver value

The invocation mode decides what value to give to the receiver. Actually, for a static mode (static $\in$ modifiers of the invoked method) the receiver value is $\perp$ (no receiver) otherwise, it will have some reference value that is the value of this.

A.3.15. Argument evaluation

An argument list is evaluated from the left to the right. In our case, we show how to evaluate just one argument. The same schema could be applied to the case of many arguments.

A.3.16. Method code locali ation

The localization of the invoked method depends on the invocation mode:

- If the invocation mode is static then we know that the invoked method is from the class C (the *ClassFrom* of the method annotation). In this case, the class C can be initialized if it is not already.
- If the invocation mode is private the invoked method is also known but no initialization is triggered.
- Otherwise, a dynamic process is required to retrieve the real method to call. This is achieved in the semantic rules of the method invocation evaluation thanks to the function LookupFirstSuperClass.

We need some functions that allow us to gather the information that is relevant to the invoked method:

- InMethod: a predicate that evaluates to trueif some method exists in some class:

  InMethod:
  $$\textit{MethodInfo} \times \text{Identifier} \times \textit{ClassFile} \times \times \textit{MethodDescriptor} \rightarrow \text{bool}$$
  InMethod:
  $$(M, \text{Identifier}, C, D)$$
  $$(M \in C.\textit{Methods}) \wedge$$
  $$(M.\textit{SimpleName} = \text{Identifier}) \wedge$$
  $$(M.\textit{Descriptor} = D).$$

- GetInvocMode(M, B): a function that returns the invocation mode of a method invocation expression using the modifier information that is in M. The value of the parameter B is true when the method invocation is *super*.

  GetInvocMode : $\textit{MethodInfo} \times \text{bool} \rightarrow \text{String}$

  GetInvocMode$(M, \mathscr{B}) =$

  if (static $\in$ *M.Modifiers*)
  then 'static'
  else if (private $\in$ *M.Modifiers*)
  then 'nonvirtual'
  else if ($\mathscr{B}$)
  then 'super'
  else if (abstract $\in$ *M.Modifiers*)
  then 'interface'
  else 'virtual'

- LookupFirstSuperClass(ρ, M, S, I): represents the dynamic process to search in the class hierarchy (explored by Γ) a method M' having the same name and descriptor as M with respect to the invocation mode I. This search is recursive through the class hierarchy and begins from the class S.

LookupFirstSuperClass:
address × *MethodInfo* × *ClassFile* ×
× String → (*MethodInfo, ClassFile*)

LookupFirstSuperClass($\rho, M, S, InvocMode$)
if (Match(M, M', S))
then if ((*InvocMode* ='super') ∨
(*InvocMode* ='interface'))
then (M', S)
else if (*InvocMode* ='virtual' ∧
overrides((*M*, concreteType(h(ρ))), (M', S)))
then (*M*, concreteType($h(\rho)$))
else if ($S.ThisClass \neq$ Object)
then LookupFirstSuperClass(ρ, M,
$S.SuperClass, InvocMode$)
else $(\perp, \perp)$

- The function overrides verifies if some method overrides another one through the class hierarchy.

overrides : $(MethodInfo, ClassFile) \times$
$\times (MethodInfo, ClassFile) \rightarrow$bool
overrides((*m*,*C*), $(m', C') =$
$(C \sqsubseteq C') \wedge$
(((private $\notin m'.Modifiers) \wedge (C \sqsubseteq C')) \vee$
$(\exists (m'', C'' \wedge m'' \neq m'' \wedge m'' \neq m') \wedge$
(overrides ((*m*,*C*), $(m'', C'')) \wedge$
overrides $((m'', C''), (m', C'))))) \wedge$
$(m.SimpleName = m'.SimpleName \wedge$
$m.Descriptor = m'.Descriptor)$

The underlying rules are presented in Tables 41, 42, 43 and 44.

A.3.17. Assignment expression evaluation

An assignment expression is made of a left-hand side, a right-hand side and the operator . The left-hand side must return a variable, the right-hand side must return a value. We show in the rules of an assignment expression how a field access expression must return a variable. The evaluation result of such an expression is an update of the value of the class or the instance variable with the value of the expression in the righthand side.
Another possible expression in the left hand side is an access to an array component. A runtime check is made between to guarantee the type compliance between the type of the righthand side expression and the type of the left hand expression. If the former is not a subtype of the latter, the arrayStoreException will be thrown at the position of the statement containing this expression. If the left hand side is a *FieldAccess* expression then it returns the variable representing a static or an instance field. An access to a static field can trigger the initialization of the class in which it is declared (if it is not already). The value of the field is updated with the value of the righthand side expression. When the left hand side is an *ArrayAccess* expression, we use a function GetMappeFields(h(ρ)) to return the map *MI* of an array having ρ as address. The rules of the assignment expression evaluation are presented in Tables 45, 46, 47 and 48.

Table 13
Grammar of the subset – Part 1

Program	::=	*ClassDeclaration Program* *InterfaceDeclaration Program* ε
ClassDeclaration	::=	*Modifiers* `class Identifier` *Extends Implements* `{` *ClassBody* `}`
Modifiers	::=	`public` *Modifiers* `private` *Modifiers* `static` *Modifiers* `abstract` *Modifiers* `final` *Modifiers* `native` *Modifiers* `transient` *Modifiers* `volatile` *Modifiers* ε
Extends	::=	`extends` *ClassType* ε
Implements	::=	`implements` *InterfaceTypeList* ε
InterfaceTypeList	::=	*InterfaceType* *InterfaceType* `,` *InterfaceTypeList*
ClassBody	:=	*FieldDeclaration ClassBody* *MethodDeclaration ClassBody* *AbstractMethodDeclaration ClassBody* *ConstructorDeclaration ClassBody* ε
FieldDeclaration	::=	*Modifiers Type* `Identifier` *Modifiers Type* `identifier =` *Expression* | *Modifiers SimpleType*`[] identifier =` *ArrayInitializer*
ArrayInitializer	::=	`{ }` `{` *ExpressionInitializer* `}`
ExpressionInitializer	::=	*Expression* | *Expression* `,` *ExpressionInitializer*
MethodDeclaration	::=	*Modifiers ResultType* `Identifier (` *Parameter* `)` *Throws Block*

Table 14
Grammar of the subset – Part 2

Parameter	::=	*Type Identifier*
	|	ϵ
Throws	::=	`throws` *ClassTypeList*
	|	ϵ
ClassTypeList	::=	*ClassType*
	|	*ClassType* , *ClassTypeList*
ConstructorDeclaration	::=	*Modifiers* `Identifier` (*Parameter*) *Throws* *ConstructorBody*
ConstructorBody	::=	{ *ExplicitConsInvocation* *BlockStatementsOrEmpty* }
ExplicitConsInvocation	::=	`this` (*Argument*) ;
	|	`super` (*Argument*) ;
	|	ϵ
Argument	::=	*Expression*
	|	ϵ
InterfaceDeclaration	::=	*Modifiers* `interface Identifier` *ExtendsInterfaces* { *InterfaceBody* }
ExtendsInterfaces	::=	`extends` *InterfaceTypeList*
	|	ϵ
InterfaceBody	::=	*FieldDeclaration*
	|	*AbstractMethodDeclaration*
AbstractMethodDeclaration	::=	*Modifiers ResultType* `Identifier` (*Parameter*) *Throws* ;
Block	::=	{ *BlockStatementsOrEmpty* }
BlockStatementsOrEmpty	::=	*BlockStatements*
	|	ϵ
BlockStatements	::=	*BlockStatement BlockStatements*
	|	*BlockStatement*
BlockStatement	::=	*LocalVariableDeclaration*
	|	*Statement*

Table 15
Grammar of the subset – Part 3

LocalVariableDeclaration	::=	*Type* `Identifier ;`
	|	*Type* `Identifier =` *Expression*
	|	*SimpleType*`[] Identifier =` *ArrayInitializer* `;`
Statement	::=	`;`
	|	*Block*
	|	*IfStatement*
	|	*WhileStatement*
	|	*ThrowStatement*
	|	*TryStatement*
	|	*ReturnStatement*
	|	*ExpressionStatement*
IfStatement	::=	`if (` *Expression* `)` *Statement* `else` *Statement*
	|	`if (` *Expression* `)` *Statement*
WhileStatement	::=	`while (` *Expression* `)` *Statement*
ThrowStatement	::=	`throw` *Expression* `;`
TryStatement	::=	*try Catches finally*
	|	*try Catches*
	|	*try finally*
ReturnStatement	::=	`return` *Expression* `;`
	|	`return ;`
try	::=	`try` *Block*
finally	::=	`finally` *Block*
Catches	::=	*Catch*
	|	*Catch Catches*
Catch	::=	`catch (` *ClassType* `Identifier )` *Block*
ExpressionStatement	::=	*StatementExpression* `;`
StatementExpression	::=	*AssignmentExpression*
	|	*MethodInvocation*
	|	*ClassInstanceCreation*
Primary	::=	*ArrayCreation*
	|	*PrimaryNoNewArray*
ArrayCreation	::=	`new` *SimpleType* `[` *Expression* `]`

Table 16
Grammar of the subset – Part 4

```
PrimaryNoNewArray   ::=  Literal
                         | this
                         | ( Expression )
                         | ClassInstanceCreation
                         | SimpleFieldAccess
                         | ArrayFieldAccess
                         | MethodInvocation

ClassInstanceCreation ::= new ClassType ( Argument )

SimpleFieldAccess   ::=  Primary . Identifier
                         | super . Identifier
                         | FieldName

FieldName           ::=  Identifier
                         | ClassOrInterfaceType . Identifier
                         | ExpressionName . Identifier

ExpressionName      ::=  FieldName
                         | SimpleLocalVarAccess

SimpleLocalVarAccess ::= Identifier

ArrayFieldAccess    ::=  PrimaryNoNewArray [ Expression ]

MethodInvocation    ::=  MethodName ( Argument )
                         | Primary . Identifier ( Argument )
                         | super . Identifier ( Argument )

MethodName          ::=  Identifier
                         | ClassType . Identifier
                         | ExpressionName . Identifier

Expression          ::=  AssignmentExpression
                         | CastExpression
                         | Primary
                         | SimpleLocalVarAccess
                         | ArrayLocalVarAccess

AssignmentExpression ::= SimpleFieldAccess = Expression
                         | ArrayFieldAccess = Expression
                         | Identifier = Expression
                         | Identifier[ Expression ] = Expression

CastExpression      ::=  ( Type ) Expression

ArrayLocalVarAccess ::=  Identifier [ Expression ]
```

Table 17
Method clinit

```
Method clinit() // supposed to be in a class C

if (not (C.Initialized))
  if ((C != Object) and (interface ∉ C.Modifiers))
   C = C.SuperClass ;
   C.clinit() ;
   C.Initialized = true ;
clinitBody
```

Table 18
Method init

```
Method init(Argument)

Class C = this.getClass() ;
if (not (C.Initialized))
  C.clinit() ;
if (C != Object)
  super() ;
initBody
```

Table 19
Instance variables evaluation

FieldDeclaration

$$\frac{\begin{array}{c}\mathsf{InField}\ (f, \texttt{Identifier}, \mathcal{F}.Class, Type) \\ F = \langle f, \mathcal{F}.Class \rangle \\ \texttt{static} \notin F.Field.Modifiers \quad \rho = \mathcal{F}.\texttt{this} \\ v = \mathsf{DefaultValue}(Type) \\ C' = \mathsf{ConcreteType}(h(\rho)) \\ MF = \mathsf{GetMappeFields}(h(\rho)) \\ h' = h \dagger [\rho \mapsto (C', MF \dagger [F \mapsto v])]\end{array}}{\Gamma \vdash (\xi, \mathcal{F}, h, Modifiers \quad Type \quad \texttt{Identifier}) \rightarrow (\xi, \mathcal{F}, h', F)}$$

$$\frac{\Gamma \vdash (\xi, \mathcal{F}, h, Expression) \rightarrow (\xi', \mathcal{F}', h', Expression')}{\Gamma \vdash (\xi, \mathcal{F}, h, F = Expression) \rightarrow (\xi', \mathcal{F}', h', F = Expression')}$$

$$\frac{\begin{array}{c}\texttt{static} \notin F.Field.Modifiers \quad \rho = \mathcal{F}.\texttt{this} \\ C' = \mathsf{ConcreteType}(h(\rho)) \\ MF = \mathsf{GetMappeFields}(h(\rho)) \\ h' = h \dagger [\rho \mapsto (C', MF \dagger [F \mapsto v])]\end{array}}{\Gamma \vdash (\xi, \mathcal{F}, h, F = v) \rightarrow (\xi, \mathcal{F}, h')}$$

$$\frac{\Gamma \vdash (\xi, \mathcal{F}, h, Expression_0) \rightarrow (\xi', \mathcal{F}', h', Expression_0')}{\begin{array}{l}\Gamma \vdash (\xi, \mathcal{F}, h, \{Expression_0, \ldots, Expression_k\}) \rightarrow \\ (\xi', \mathcal{F}', h', \{Expression_0', \ldots, Expression_k\})\end{array}}$$

$$\frac{\Gamma \vdash (\xi, \mathcal{F}, h, Expression_k) \rightarrow (\xi', \mathcal{F}', h', Expression_k')}{\Gamma \vdash (\xi, \mathcal{F}, h, \{v_0, \ldots, v_{k-1}, Expression_k\}) \rightarrow (\xi', \mathcal{F}', h', \{v_0, \ldots, v_{k-1}, Expression_k'\})}$$

$$\frac{\begin{array}{c}\texttt{static} \notin F.Field.Modifiers \quad \rho = \mathcal{F}.\texttt{this} \\ C' = \mathsf{ConcreteType}(h(\rho)) \\ MF = \mathsf{GetMappeFields}(h(\rho)) \\ \rho' = MF(F) \\ T = \mathsf{ConcreteType}(\rho') \\ h' = h[\rho' \mapsto (T, [0 \mapsto v_0, \ldots, k \mapsto v_k])] \\ h'' = h' \dagger [\rho \mapsto (C', MF \dagger [F \mapsto \rho'])]\end{array}}{\Gamma \vdash (\xi, \mathcal{F}, h, F = \{v_0, \ldots, v_k\}) \rightarrow (\xi, \mathcal{F}, h'')}$$

Table 20
Static variables evaluation

FieldDeclaration

$$\frac{\begin{array}{c}\mathsf{InField}\ (f, \texttt{Identifier}, \mathcal{F}.Class, Type) \\ F = \langle f, \mathcal{F}.Class \rangle \\ \texttt{static} \in F.Field.Modifiers \\ v = \mathsf{DefaultValue}(Type) \\ h' = h \dagger [F \mapsto v]\end{array}}{\Gamma \vdash (\xi, \mathcal{F}, h, Modifiers\ \ Type\ \ \texttt{Identifier}) \rightarrow (\xi, \mathcal{F}, h', F)}$$

$$\frac{\Gamma \vdash (\xi, \mathcal{F}, h, Expression) \rightarrow (\xi', \mathcal{F}', h', Expression')}{\begin{array}{c}\Gamma \vdash (\xi, \mathcal{F}, h, F = Expression) \rightarrow \\ (\xi', \mathcal{F}', h', F = Expression')\end{array}}$$

$$\frac{\begin{array}{c}\texttt{static} \in F.Field.Modifiers \\ h' = h \dagger [F \mapsto v]\end{array}}{\Gamma \vdash (\xi, \mathcal{F}, h, F = v) \rightarrow (\xi, \mathcal{F}, h')}$$

$$\frac{\begin{array}{c}\texttt{static} \in F.Field.Modifiers \\ \rho = h(F) \\ T = \mathsf{ConcreteType}(\rho) \\ h' = h \dagger [\rho \mapsto (T, [0 \mapsto v_0, \ldots, k \mapsto v_k])]\end{array}}{\Gamma \vdash (\xi, \mathcal{F}, h, F = \{v_0, \ldots, v_k\}) \rightarrow (\xi, \mathcal{F}, h')}$$

Table 21
Constructor evaluation

ExplicitConsInvocation

$$\frac{\Gamma \vdash (\xi, \mathcal{F}, h, Argument) \rightarrow (\xi', \mathcal{F}', h', Argument')}{\Gamma \vdash (\xi, \mathcal{F}, h, [D]\texttt{this}(Argument)) \rightarrow (\xi', \mathcal{F}', h', [D]\texttt{this}(Argument'))}$$

$$\frac{C = \mathsf{ConcreteType}(\mathcal{F}.\texttt{this})}{\Gamma \vdash (\xi, \mathcal{F}, h, [D]\texttt{this}(v)) \rightarrow (\xi, \mathcal{F}, h, \texttt{this}.[C, D]\texttt{init}(v))}$$

$$\frac{\Gamma \vdash (\xi, \mathcal{F}, h, Argument) \rightarrow (\xi', \mathcal{F}', h', Argument')}{\Gamma \vdash (\xi, \mathcal{F}, h, [D]\texttt{super}(Argument)) \rightarrow (\xi', \mathcal{F}', h', [D]\texttt{super}(Argument'))}$$

$$\frac{C = \mathcal{F}.Class.SuperClass}{\Gamma \vdash (\xi, \mathcal{F}, h, [D]\texttt{super}(v)) \rightarrow (\xi, \mathcal{F}, h, \texttt{super}.[C, D]\texttt{init}(v))}$$

Table 22
Local variable declaration evaluation

LocalVariableDeclaration

$$\frac{v = \mathsf{DefaultValue}(Type) \quad \mathcal{LV} = \mathcal{F}.Method.LocalVariableTable \dagger [\texttt{Identifier} \mapsto v]}{\Gamma \vdash (\xi, \mathcal{F}, h, Type\ \ \texttt{Identifier}) \rightarrow (\xi, \mathcal{F}[Method.LocalVariableTable \leftarrow \mathcal{LV}], h, \texttt{Identifier})}$$

$$\frac{\Gamma \vdash (\xi, \mathcal{F}, h, Expression) \rightarrow (\xi', \mathcal{F}', h', Expression')}{\Gamma \vdash (\xi, f, h, \texttt{Identifier} = Expression) \rightarrow (\xi', \mathcal{F}', h', \texttt{Identifier} = Expression')}$$

$$\frac{\mathcal{LV} = \mathcal{F}.Method.LocalVariableTable \dagger [\texttt{Identifier} \mapsto v]}{\Gamma \vdash (\xi, \mathcal{F}, h, \texttt{Identifier} = v) \rightarrow (\xi, \mathcal{F}[Method.LocalVariableTable \leftarrow \mathcal{LV}], h)}$$

$$\frac{\begin{array}{c}\rho = \mathsf{fresh}(h) \\ T = \mathsf{ConcreteType}(\mathcal{F}.Method.LocalVariableTable(\texttt{Identifier})) \\ h' = h \dagger [\rho \mapsto (T, [0 \mapsto v_0, \ldots, k \mapsto v_k])] \\ \mathcal{LV} = \mathcal{F}.Method.LocalVariableTable \dagger [\texttt{Identifier} \mapsto \rho]\end{array}}{\begin{array}{l}\Gamma \vdash (\xi, \mathcal{F}, h, \texttt{Identifier} = \{v_0, \ldots, v_k\}) \rightarrow \\ (\xi, \mathcal{F}[Method.LocalVariableTable \leftarrow \mathcal{LV}], h')\end{array}}$$

Table 23
Statement evaluation – Part 1

Block

$$\frac{\square}{\Gamma \vdash (\xi, \mathcal{F}, h, ;) \rightarrow (\xi, \mathcal{F}, h)}$$

$$\frac{\Gamma \vdash (\xi, \mathcal{F}, h, Statement) \rightarrow (\xi', \mathcal{F}', h', Statement')}{\Gamma \vdash (\xi, \mathcal{F}, h, Statement\ ;\ Statements) \rightarrow (\xi', \mathcal{F}', h', Statement'\ ;\ Statements)}$$

$$\frac{\begin{array}{c}\Gamma \vdash (\xi, \mathcal{F}, h, Statement) \rightarrow (\xi^*, \mathcal{F}, h') \\ \mathcal{F}.ReturnValue = \bot\end{array}}{\Gamma \vdash (\xi, \mathcal{F}, h, Statement\ ;\ Statements) \rightarrow (\xi^+, \mathcal{F}, h', Statements)}$$

IfStatement

$$\frac{\Gamma \vdash (\xi, \mathcal{F}, h, Expression) \rightarrow (\xi', \mathcal{F}', h', Expression')}{\begin{array}{l}\Gamma \vdash (\xi, \mathcal{F}, h, \texttt{if}\ (Expression)\ Statement) \rightarrow \\ (\xi', \mathcal{F}', h', \texttt{if}\ (Expression')\ Statement)\end{array}}$$

$$\frac{\square}{\Gamma \vdash (\xi, \mathcal{F}, h, \texttt{if}\ (\texttt{true})\ Statement) \rightarrow (\xi, \mathcal{F}, h, Statement)}$$

$$\frac{\square}{\Gamma \vdash (\xi, \mathcal{F}, h, \texttt{if}\ (\texttt{false})\ Statement) \rightarrow (\xi, \mathcal{F}, h)}$$

$$\frac{\Gamma \vdash (\xi, \mathcal{F}, h, Expression) \rightarrow (\xi', \mathcal{F}', h', Expression')}{\begin{array}{l}\Gamma \vdash (\xi, \mathcal{F}, h, \texttt{if}\ (Expression)\ Statement_1\ \texttt{else}\ Statement_2\ ) \rightarrow \\ (\xi', \mathcal{F}', h', \texttt{if}\ (Expression')\ Statement_1\ \texttt{else}\ Statement_2)\end{array}}$$

$$\frac{\square}{\begin{array}{l}\Gamma \vdash (\xi, \mathcal{F}, h, \texttt{if}\ (\texttt{true})\ Statement_1\ \texttt{else}\ Statement_2\ ) \rightarrow \\ (\xi, \mathcal{F}, h, Statement_1)\end{array}}$$

$$\frac{\square}{\begin{array}{l}\Gamma \vdash (\xi, \mathcal{F}, h, \texttt{if}\ (\texttt{false})\ Statement_1\ \texttt{else}\ Statement_2) \rightarrow \\ (\xi, \mathcal{F}, h, Statement_2)\end{array}}$$

Table 24
Statement evaluation – Part 2

WhileStatement

$$\frac{\Gamma \vdash (\xi, \mathcal{F}, h, Expression) \rightarrow (\xi', \mathcal{F}', h', Expression')}{\Gamma \vdash (\xi, \mathcal{F}, h, \texttt{while}\ (Expression)\ Statement) \rightarrow (\xi', \mathcal{F}', h', (\texttt{if}\ (Expression')\ \texttt{then}\ Statement\ );\ \texttt{while}\ (Expression)\ Statement)}$$

$$\Gamma \vdash (\xi, \mathcal{F}, h, \texttt{while}\ (\texttt{false})\ Statement) \rightarrow (\xi, \mathcal{F}, h) \quad \square$$

Table 25
Return statement evaluation

ReturnStatement

$$\frac{\Gamma \vdash (\xi, \mathcal{F}, h, Expression) \rightarrow (\xi', \mathcal{F}', h', Expression')}{\Gamma \vdash (\xi, \mathcal{F}, h, \texttt{return}\ Expression) \rightarrow (\xi', \mathcal{F}', h', \texttt{return}\ Expression')}$$

$$\frac{\mathsf{HandlerInTable}(Any, \mathcal{F}.Method.ExceptionTable, H, \mathcal{P}) \quad Statement = H.Target}{\Gamma \vdash (\xi, \mathcal{F}, h, \texttt{return}\ v, \mathcal{P}) \rightarrow (\mathcal{F}[ReturnValue \leftarrow v], h, Statement)}$$

$$\frac{\neg\mathsf{HandlerInTable}(Any, \mathcal{F}.Method.ExceptionTable, H, \mathcal{P})}{\Gamma \vdash (\xi, \mathcal{F}, h, \texttt{return}\ v, \mathcal{P}) \rightarrow (\mathcal{F}.PreviousFrame, h, v)}$$

$$\frac{\mathsf{HandlerInTable}(Any, \mathcal{F}.Method.ExceptionTable, H, \mathcal{P}) \quad Statement = H.Target}{\Gamma \vdash (\xi, \mathcal{F}, h, \texttt{return}, \mathcal{P}) \rightarrow (\mathcal{F}, h, Statement)}$$

$$\frac{\neg\mathsf{HandlerInTable}(Any, \mathcal{F}.Method.ExceptionTable, H, \mathcal{P})}{\Gamma \vdash (\xi, \mathcal{F}, h, \texttt{return}, \mathcal{P}) \rightarrow (\mathcal{F}.PreviousFrame, h)}$$

Table 26
Exception constructs

```
try
{
1 X.c = b;
2 if (b) return;
}
catch(E1 X1) Block1
...

catch(En Xn) Blockn
finally Blockn+1
```

Table 27
Exceptions table

From	*To*	*Target*	*Parameter*	*Type*
1	2	$Block_1$	X_1	E_1
1	2	$Block_2$	X_2	E_2
...	...	...	...	...
1	n	$Block_{n+1}$	void	*Any*

Table 28

Exception handling – Part 1

ThrowStatement

$$\frac{\Gamma \vdash (\xi, \mathcal{F}, h, Expression) \rightarrow (\xi', \mathcal{F}', h', Expression')}{\Gamma \vdash (\xi, \mathcal{F}, h, \texttt{throw}\ \ Expression) \rightarrow (\xi', \mathcal{F}', h', \texttt{throw}\ \ Expression')}$$

$$\frac{\begin{array}{c}\rho = \mathsf{fresh}(h) \\ h' = h \dagger [\rho \mapsto (\texttt{NullPointerException}, \ldots)]\end{array}}{\Gamma \vdash (\xi, \mathcal{F}, h, \texttt{throw}\ \ \texttt{null}, \mathcal{P}) \rightarrow (\rho, \mathcal{F}, h', \texttt{throw}\ \ \rho, \mathcal{P})}$$

$$\frac{\begin{array}{c}\mathsf{HandlerInTable}(\mathsf{ConcreteType}(\rho), \mathcal{F}.Method.ExceptionTable, H, \mathcal{P}) \\ Statement = H.Target \quad X = H.Parameter \\ \mathcal{LV} = \mathcal{F}.Method.LocalVariableTable \dagger [X \mapsto \rho]\end{array}}{\Gamma \vdash (\rho, \mathcal{F}, h, \texttt{throw}\ \ \rho, \mathcal{P}) \rightarrow (\mathcal{F}[Method.LocalVariableTable \leftarrow \mathcal{LV}], h, Statement)}$$

$$\frac{\begin{array}{c}\mathsf{HandlerInTable}(\mathsf{ConcreteType}(\rho), \mathcal{F}.Method.ExceptionTable, H, \mathcal{P}) \\ Statement = H.Target \quad H.Type = Any\end{array}}{\Gamma \vdash (\xi, \mathcal{F}, h, \texttt{throw}\ \ \rho, \mathcal{P}) \rightarrow (\rho, \mathcal{F}, h, Statement)}$$

$$\frac{\begin{array}{c}\neg\mathsf{HandlerInTable}(\mathsf{ConcreteType}(\rho), \mathcal{F}.Method.ExceptionTable, H, \mathcal{P}) \\ \mathcal{F}.Method.SimpleName \neq \ \texttt{'Main'}\end{array}}{\Gamma \vdash (\xi, \mathcal{F}, h, \texttt{throw}\ \ \rho, \mathcal{P}) \rightarrow (\rho, \mathcal{F}.PreviousFrame, h, \texttt{throw}\ \ \rho, \mathcal{F}.ReturnAddress)}$$

$$\frac{\begin{array}{c}\neg\mathsf{HandlerInTable}(\mathsf{ConcreteType}(\rho), \mathcal{F}.Method.ExceptionTable, H, \mathcal{P}) \\ \mathcal{F}.Method.SimpleName = \ \texttt{'Main'}\end{array}}{\Gamma \vdash (\xi, \mathcal{F}, h, \texttt{throw}\ \ \rho, \mathcal{P}) \rightarrow (\rho, \mathcal{F}, h, \texttt{System.exit(1)})}$$

TryStatement

try

$$\frac{\Gamma \vdash (\xi, \mathcal{F}, h, Block) \rightarrow (\xi', \mathcal{F}', h', Block')}{\Gamma \vdash (\xi, \mathcal{F}, h, \texttt{try}\ \ Block) \rightarrow (\xi', \mathcal{F}', h', \texttt{try}\ \ Block')}$$

Table 29

Exception handling – Part 2

$$\frac{\Gamma \vdash (\xi, \mathcal{F}, h, Block) \rightarrow (\xi^{+}, \mathcal{F}, h') \quad \mathcal{P} = \mathsf{FinalPosition}(Block) \quad \mathsf{HandlerInTable}(Any, \mathcal{F}.Method.ExceptionTable, H, \mathcal{P}) \quad Statement = H.Target \quad H.Type = Any}{\Gamma \vdash (\xi, \mathcal{F}, h, \texttt{try}\ \ Block) \rightarrow (\xi^{+}, \mathcal{F}, h', Statement)}$$

$$\frac{\rho = \mathsf{fresh}(h) \quad \Gamma \vdash (\xi, \mathcal{F}, h, Block) \rightarrow (\rho, \mathcal{F}, h', \texttt{throw}\ \ \rho, \mathcal{P})}{\Gamma \vdash (\xi, \mathcal{F}, h, \texttt{try}\ \ Block) \rightarrow (\rho, \mathcal{F}, h', \texttt{throw}\ \ \rho, \mathcal{P})}$$

catch

$$\frac{\Gamma \vdash (\mathcal{F}, h, Block) \rightarrow (\mathcal{F}', h', Block')}{\Gamma \vdash (\mathcal{F}, h, \texttt{catch}(ClassType\ \ \texttt{Identifier})\ \ Block) \rightarrow (\mathcal{F}', h', \texttt{catch}\ \ Block')}$$

$$\frac{\Gamma \vdash (\mathcal{F}, h, Block) \rightarrow (\mathcal{F}, h') \quad \mathcal{P} = \mathsf{FinalPosition}(Block) \quad \mathsf{HandlerInTable}(Any, \mathcal{F}.Method.ExceptionTable, H, \mathcal{P}) \quad Statement = H.Target \quad H.Type = Any}{\Gamma \vdash (\mathcal{F}, h, \texttt{catch}(ClassType\ \ \texttt{Identifier})\ \ Block) \rightarrow (\mathcal{F}, h', Statement)}$$

$$\frac{\rho = \mathsf{fresh}(h) \quad \Gamma \vdash (\mathcal{F}, h, Block) \rightarrow (\rho, \mathcal{F}, h', \texttt{throw}\ \ \rho, \mathcal{P})}{\Gamma \vdash (\mathcal{F}, h, \texttt{catch}(ClassType\ \ \texttt{Identifier})\ \ Block) \rightarrow (\rho, \mathcal{F}, h', \texttt{throw}\ \ \rho, \mathcal{P})}$$

finally

$$\frac{\Gamma \vdash (\xi, \mathcal{F}, h, Block) \rightarrow (\xi', \mathcal{F}', h', Block')}{\Gamma \vdash (\xi, \mathcal{F}, h, \texttt{finally}\ \ Block) \rightarrow (\xi', \mathcal{F}', h', \texttt{finally}\ \ Block')}$$

Table 30
Exception handling – Part 3

finally

$$\frac{\Gamma \vdash (\xi, \mathcal{F}, h, Block) \to (\xi, \mathcal{F}, h') \quad \mathcal{P} = \mathsf{FinalPosition}(Block) \quad \mathcal{F}.ReturnValue = \bot}{\Gamma \vdash (\xi, \mathcal{F}, h, \texttt{finally}\ \ Block) \to (\xi, \mathcal{F}, h', \texttt{throw}\ \ \xi, \mathcal{P})}$$

$$\frac{\rho = \mathsf{fresh}(h) \quad \Gamma \vdash (\xi, \mathcal{F}, h, Block) \to (\rho, \mathcal{F}, h', \texttt{throw}\ \ \rho, \mathcal{P})}{\Gamma \vdash (\xi, \mathcal{F}, h, \texttt{finally}\ \ Block) \to (\rho, \mathcal{F}, h', \texttt{throw}\ \ \rho, \mathcal{P})}$$

$$\frac{\Gamma \vdash (\mathcal{F}, h, Block) \to (\mathcal{F}, h') \quad \mathcal{P} = \mathsf{FinalPosition}(Block) \quad \mathcal{F}.ReturnValue \neq \bot \quad \neg\mathsf{HandlerInTable}(Any, \mathcal{F}.Method.ExceptionTable, H, \mathcal{P})}{\Gamma \vdash (\mathcal{F}, h, \texttt{finally}\ \ Block) \to (\mathcal{F}.PreviousFrame, h', \mathcal{F}.ReturnValue)}$$

$$\frac{\Gamma \vdash (\mathcal{F}, h, Block) \to (\mathcal{F}, h') \quad \mathcal{P} = \mathsf{FinalPosition}(Block) \quad \mathsf{HandlerInTable}(Any, \mathcal{F}.Method.ExceptionTable, H, \mathcal{P}) \quad Statement = H.Target \quad H.Type = Any}{\Gamma \vdash (\mathcal{F}, h, \texttt{finally}\ \ Block) \to (\mathcal{F}, h', Statement)}$$

$$\frac{\Gamma \vdash (\mathcal{F}, h, Block) \to (\mathcal{F}, h') \quad \mathcal{P} = \mathsf{FinalPosition}(Block) \quad \mathcal{F}.ReturnValue = \bot \quad \neg\mathsf{HandlerInTable}(Any, \mathcal{F}.Method.ExceptionTable, H, \mathcal{P})}{\Gamma \vdash (\mathcal{F}, h, \texttt{finally}\ \ Block) \to (\mathcal{F}, h')}$$

Table 31
New array expression evaluation

ArrayCreation

$$\frac{\Gamma \vdash (\xi, \mathcal{F}, h, Expression) \to (\xi', \mathcal{F}', h', Expression')}{\Gamma \vdash (\xi, \mathcal{F}, h, \texttt{new}\ \ SimpleType[Expression]) \to (\xi', \mathcal{F}', h', \texttt{new}\ \ SimpleType[Expression'])}$$

$$\frac{\rho = \mathsf{fresh}(h) \quad v < 0 \quad h' = h \dagger [\rho \mapsto (\texttt{NegativeArraySizeException}, \ldots)]}{\Gamma \vdash (\xi, \mathcal{F}, h, \texttt{new}\ \ SimpleType[v], \mathcal{P}) \to (\rho, \mathcal{F}, h', \texttt{throw}\ \ \rho, \mathcal{P})}$$

$$\frac{\rho = \mathsf{fresh}(h) \quad v \geq 0 \quad v_i = \texttt{Defaultvalue}(SimpleType)\ \ i \in \{0, \ldots, v-1\} \quad MI = [0 \mapsto v_0, 1 \mapsto v_1, \ldots, v-1 \mapsto v_{v-1}] \quad h' = h \dagger [\rho \mapsto (SimpleType[], MI)]}{\Gamma \vdash (\xi, \mathcal{F}, h, \texttt{new}\ \ SimpleType[v]) \to (\xi, \mathcal{F}, h', \rho)}$$

Table 32
Literal, this and parenthesized expression evaluation

PrimayNoNewArray

$$\frac{v = \text{eval}(Literal)}{\Gamma \vdash (\xi, \mathcal{F}, h, Literal) \rightarrow (\xi, \mathcal{F}, h, v)}$$

$$\frac{\rho = \mathcal{F}.\texttt{this}}{\Gamma \vdash (\xi, \mathcal{F}, h, \texttt{this}) \rightarrow (\xi, \mathcal{F}, h, \rho)}$$

$$\frac{\Gamma \vdash (\xi, \mathcal{F}, h, Expression) \rightarrow (\xi', \mathcal{F}', h', Expression')}{\Gamma \vdash (\xi, \mathcal{F}, h, (Expression)) \rightarrow (\xi', \mathcal{F}', h', (Expression'))}$$

$$\frac{\Gamma \vdash (\xi, \mathcal{F}, h, Expression) \rightarrow (\xi^{+}, \mathcal{F}, h', v)}{\Gamma \vdash (\xi, \mathcal{F}, h, (Expression)) \rightarrow (\xi^{+}, \mathcal{F}, h', v)}$$

Table 33
New instance class creation evaluation

ClassInstanceCreation

$$\frac{\Gamma \vdash (\xi, \mathcal{F}, h, Argument) \rightarrow (\xi', \mathcal{F}', h', Argument')}{\begin{array}{c}\Gamma \vdash (\xi, \mathcal{F}, h, [D]\texttt{new}\ ClassType(Argument)) \rightarrow \\ (\xi', \mathcal{F}', h', [D]\texttt{new}\ ClassType(Argument'))\end{array}}$$

$$\frac{\begin{array}{c}\rho = \text{fresh}(h) \\ F_i \in \text{Fields}(ClassType)\ \ i \in \{0, \ldots, n-1\} \\ v_i = \texttt{Defaultvalue}(F_i.Field.Descriptor) \\ h' = h \dagger [\rho \mapsto (ClassType, [F_0 \mapsto v_0, \ldots, F_{n-1} \mapsto v_{n-1}])]\end{array}}{\begin{array}{c}\Gamma \vdash (\xi, \mathcal{F}, h, [D]\texttt{new}\ \ ClassType(v)) \rightarrow \\ (\xi, \mathcal{F}[\texttt{this} \leftarrow \rho], h', \texttt{this}.[ClassType, D]\texttt{init}(v)\ ;\ \rho)\end{array}}$$

Table 34
Cast expression evaluation

CastExpression

$$\frac{\Gamma \vdash (\xi, \mathcal{F}, h, Expression) \rightarrow (\xi', \mathcal{F}', h', Expression')}{\Gamma \vdash (\xi, \mathcal{F}, h, (Type)Expression) \rightarrow (\xi', \mathcal{F}', h', (Type)Expression')}$$

$$\frac{\begin{array}{c}\text{isRefType}(Type) \\ \text{ConcreteType}(h(v)) \sqsubseteq Type\end{array}}{\Gamma \vdash (\xi, \mathcal{F}, h, (Type)v) \rightarrow (\xi, \mathcal{F}, h, v)}$$

$$\frac{\begin{array}{c}\text{isRefType}(Type) \\ \text{ConcreteType}(h(v)) \not\sqsubseteq Type \\ h' = h \dagger [\rho \mapsto (\texttt{ClassCastException}, \ldots)]\end{array}}{\Gamma \vdash (\xi, \mathcal{F}, h, (Type)v, \mathcal{P}) \rightarrow (\rho, \mathcal{F}, h', \texttt{throw}\ \ \rho, \mathcal{P})}$$

$$\frac{\neg\text{isRefType}(Type)}{\Gamma \vdash (\xi, \mathcal{F}, h, (Type)v) \rightarrow (\xi, \mathcal{F}, h, v)}$$

Table 35

Field access expression evaluation – Part 1

SimpleFieldAccess

$$\frac{\Gamma \vdash (\xi, \mathcal{F}, h, Primary) \rightarrow (\xi', \mathcal{F}', h', Primary')}{\Gamma \vdash (\xi, \mathcal{F}, h, Primary.[C, D]\texttt{Identifier}) \rightarrow (\xi', \mathcal{F}', h', Primary'.[C, D]\texttt{Identifier})}$$

$$\frac{\begin{array}{c}\mathsf{InField}\ (f, \texttt{Identifier}, C, D) \\ F = \langle f, C \rangle \\ \texttt{static} \in F.Field.Modifiers \quad \texttt{final} \in F.Field.Modifiers \\ \mathsf{constant}(F)\end{array}}{\Gamma \vdash (\xi, \mathcal{F}, h, \rho.[C, D]\texttt{Identifier}) \rightarrow (\xi, \mathcal{F}, h, h(F))}$$

$$\frac{\begin{array}{c}\mathsf{InField}\ (f, \texttt{Identifier}, C, D) \\ F = \langle f, C \rangle \\ \texttt{static} \in F.Field.Modifiers \quad \texttt{final} \in F.Field.Modifiers \\ \neg\mathsf{constant}(F) \quad \mathsf{initialized}(C)\end{array}}{\Gamma \vdash (\xi, \mathcal{F}, h, \rho.[C, D]\texttt{Identifier}) \rightarrow (\xi, \mathcal{F}, h, h(F))}$$

$$\frac{\begin{array}{c}\mathsf{InField}\ (f, \texttt{Identifier}, C, D) \\ F = \langle f, C \rangle \\ \texttt{static} \in F.Field.Modifiers \quad \texttt{final} \in F.Field.Modifiers \\ \neg\mathsf{constant}(F) \quad \neg\mathsf{initialized}(C)\end{array}}{\Gamma \vdash (\xi, \mathcal{F}, h, \rho.[C, D]\texttt{Identifier}) \rightarrow (\xi, \mathcal{F}, h, C.[C, ()\texttt{void}]\texttt{clinit}()\ ;\ h(F))}$$

$$\frac{\begin{array}{c}\mathsf{InField}\ (f, \texttt{Identifier}, C, D) \\ F = \langle f, C \rangle \\ \texttt{static} \in F.Field.Modifiers \quad \texttt{final} \notin F.Field.Modifiers \\ \neg\mathsf{initialized}(C)\end{array}}{\Gamma \vdash (\xi, \mathcal{F}, h, \rho.[C, D]\texttt{Identifier}) \rightarrow (\xi, \mathcal{F}, h, C.[C, ()\texttt{void}]\texttt{clinit}()\ ;\ h(F))}$$

Table 36

Field access expression evaluation – Part 2

SimpleFieldAccess

$$\frac{\begin{array}{c}\mathsf{InField}\ (f, \texttt{Identifier}, C, D) \\ F = \langle f, C\rangle \\ \texttt{static} \in F.Field.Modifiers \quad \texttt{final} \notin F.Field.Modifiers \\ \mathsf{initialized}(C)\end{array}}{\Gamma \vdash (\xi, \mathcal{F}, h, \rho.[C, D]\texttt{Identifier}) \rightarrow (\xi, \mathcal{F}, h, h(F))}$$

$$\frac{\begin{array}{c}\mathsf{InField}\ (f, \texttt{Identifier}, C, D) \\ F = \langle f, C\rangle \\ \rho = \mathsf{fresh}(h) \\ \texttt{static} \notin F.Field.Modifiers \\ h' = h \dagger [\rho \mapsto (\texttt{NullPointerException}, \ldots)]\end{array}}{\Gamma \vdash (\xi, \mathcal{F}, h, \texttt{null}.[C, D]\texttt{Identifier}) \rightarrow (\rho, \mathcal{F}, h, \texttt{throw}\ \ \rho, \mathcal{P})}$$

$$\frac{\begin{array}{c}\mathsf{InField}\ (f, \texttt{Identifier}, C, D) \\ F = \langle f, C\rangle \\ \texttt{static} \notin F.Field.Modifiers \\ MF = \mathsf{GetMappeFields}(h(\rho))\end{array}}{\Gamma \vdash (\xi, \mathcal{F}, h, \rho.[C, D]\texttt{Identifier}) \rightarrow (\xi, \mathcal{F}, h', MF(F))}$$

$$\frac{\varsigma = \mathcal{F}.Class.SuperClass}{\begin{array}{l}\Gamma \vdash (\xi, \mathcal{F}, h, \texttt{super}.[C, D]\texttt{Identifier}) \ \rightarrow \\ (\xi, \mathcal{F}, h, ((\varsigma)\texttt{this}).[C, D]\texttt{Identifier})\end{array}}$$

$$\frac{\begin{array}{c}\mathsf{InField}\ (f, \texttt{Identifier}, C, D) \\ F = \langle f, C\rangle \\ \texttt{static} \in F.Field.Modifiers \quad \texttt{final} \in F.Field.Modifiers \\ \mathsf{constant}(F)\end{array}}{\Gamma \vdash (\xi, \mathcal{F}, h, ClassType.[C, D]\texttt{Identifier}) \rightarrow (\xi, \mathcal{F}, h, h(F))}$$

Table 37

Field access expression evaluation – Part 3

SimpleFieldAccess

$$\frac{\mathsf{InField}\ (f, \texttt{Identifier}, C, D) \quad F = \langle f, C\rangle \quad \texttt{static} \in F.Field.Modifiers \quad \texttt{final} \in F.Field.Modifiers \quad \neg\mathsf{constant}(F) \quad \neg\mathsf{initialized}(C)}{\Gamma \vdash (\xi, \mathcal{F}, h, ClassType.[C, D]\texttt{Identifier}) \rightarrow (\xi, \mathcal{F}, h, C.[C, ()\texttt{void}]\texttt{clinit}()\ ;\ h(F))}$$

$$\frac{\mathsf{InField}\ (f, \texttt{Identifier}, C, D) \quad F = \langle f, C\rangle \quad \texttt{static} \in F.Field.Modifiers \quad \texttt{final} \in F.Field.Modifiers \quad \neg\mathsf{constant}(F) \quad \mathsf{initialized}(C)}{\Gamma \vdash (\xi, \mathcal{F}, h, ClassType.[C, D]\texttt{Identifier}) \rightarrow (\xi, \mathcal{F}, h, h(F))}$$

$$\frac{\mathsf{InField}\ (f, \texttt{Identifier}, C, D) \quad F = \langle f, C\rangle \quad \texttt{static} \in F.Field.Modifiers \quad \texttt{final} \notin F.Field.Modifiers \quad \neg\mathsf{initialized}(C)}{\Gamma \vdash (\xi, \mathcal{F}, h, ClassType.[C, D]\texttt{Identifier}) \rightarrow (\xi, \mathcal{F}, h, C.[C, ()\texttt{void}]\texttt{clinit}()\ ;\ h(F))}$$

$$\frac{\mathsf{InField}\ (f, \texttt{Identifier}, C, D) \quad F = \langle f, C\rangle \quad \texttt{static} \in F.Field.Modifiers \quad \texttt{final} \notin F.Field.Modifiers \quad \mathsf{initialized}(C)}{\Gamma \vdash (\xi, \mathcal{F}, h, ClassType.[C, D]\texttt{Identifier}) \rightarrow (\xi, \mathcal{F}, h, h(F))}$$

$$\frac{\mathsf{InField}\ (f, \texttt{Identifier}, C, D) \quad F = \langle f, C\rangle \quad \neg\mathsf{constant}(F) \quad \neg\mathsf{initialized}(C)}{\Gamma \vdash (\xi, \mathcal{F}, h, InterfaceType.[C, D]\texttt{Identifier}) \rightarrow (\xi, \mathcal{F}, h, C.[C, ()\texttt{void}]\texttt{clinit}()\ ;\ h(F))}$$

$$\frac{\mathsf{InField}\ (f, \texttt{Identifier}, C, D) \quad F = \langle f, C\rangle \quad \neg\mathsf{constant}(F) \quad \mathsf{initialized}(C)}{\Gamma \vdash (\xi, \mathcal{F}, h, InterfaceType.[C, D]\texttt{Identifier}) \rightarrow (\xi, \mathcal{F}, h, h(F))}$$

Table 38
Field access expression evaluation – Part 4

SimpleFieldAccess

$$\frac{\begin{array}{c}\mathsf{InField}\ (f, \texttt{Identifier}, C, D) \\ F = \langle f, C\rangle \\ \mathsf{constant}(F)\end{array}}{\Gamma \vdash (\xi, \mathcal{F}, h, \mathit{InterfaceType}.[C, D[\texttt{Identifier}) \rightarrow (\xi, \mathcal{F}, h, h(F))}$$

$$\frac{\begin{array}{c}\mathsf{InField}\ (f, \texttt{Identifier}, C, D) \\ F = \langle f, C\rangle \\ \texttt{static} \notin F.\mathit{Field.Modifiers} \\ MF = \mathsf{GetMappeFields}(h(\rho))\end{array}}{\Gamma \vdash (\xi, \mathcal{F}, h, [C, D]\texttt{Identifier}) \rightarrow (\xi, \mathcal{F}, h, MF(F))}$$

$$\frac{\begin{array}{c}\mathsf{InField}\ (f, \texttt{Identifier}, C, D) \\ F = \langle f, C\rangle \\ \texttt{static} \in F.\mathit{Field.Modifiers}\end{array}}{\Gamma \vdash (\xi, \mathcal{F}, h, [C, D]\texttt{Identifier}) \rightarrow (\xi, \mathcal{F}, h, h(F))}$$

Table 39
Array field access evaluation

ArrayFieldAccess

$$\frac{\Gamma \vdash (\xi, \mathcal{F}, h, \mathit{PrimaryNoNewArray}) \rightarrow (\xi', \mathcal{F}', h', \mathit{PrimaryNoNewArray}')}{\begin{array}{c}\Gamma \vdash (\mathcal{F}, h, \mathit{PrimaryNoNewArray}[\mathit{Expression}]) \rightarrow \\ (\xi', \mathcal{F}', h', \mathit{PrimaryNoNewArray}'[\mathit{Expression}])\end{array}}$$

$$\frac{\Gamma \vdash (\xi, \mathcal{F}, h, \mathit{Expression}) \rightarrow (\xi', \mathcal{F}', h', \mathit{Expression}')}{\Gamma \vdash (\xi, \mathcal{F}, h, \rho[\mathit{Expression}]) \rightarrow (\xi', \mathcal{F}', h', \rho[\mathit{Expression}'])}$$

$$\frac{\begin{array}{c}\rho = \mathsf{fresh}(h) \\ h' = h \dagger [\rho \mapsto (\texttt{NullPointerException}, \ldots)]\end{array}}{\Gamma \vdash (\xi, \mathcal{F}, h, \texttt{null}[v], \mathcal{P}) \rightarrow (\rho, \mathcal{F}, h', \texttt{throw}\ \ \rho, \mathcal{P})}$$

$$\frac{\begin{array}{c}v < 0 \ \vee \ v \geq h(\rho).\texttt{length} \\ \rho' = \mathsf{fresh}(h) \\ h' = h \dagger [\rho' \mapsto (\texttt{IndexOutOfBoundsException}, \ldots)\end{array}}{\Gamma \vdash (\xi, \mathcal{F}, h, \rho[v], \mathcal{P}) \rightarrow (\rho', \mathcal{F}, h', \texttt{throw}\ \ \rho', \mathcal{P})}$$

$$\frac{\begin{array}{c}v \geq 0 \ \wedge \ v < h(\rho).\texttt{length} \\ MI = \mathsf{GetMappeIndex}(h(\rho))\end{array}}{\Gamma \vdash (\xi, \mathcal{F}, h, \rho[v]) \rightarrow (\xi, \mathcal{F}, h, MI(v))}$$

Table 40
Simple local variable access

SimpleLocalVariable

$$\frac{v = \mathcal{F}.\mathit{Method.LocalVariableTable}(\texttt{Identifier})}{\Gamma \vdash (\xi, \mathcal{F}, h, \texttt{Identifier}) \rightarrow (\xi, \mathcal{F}, h, v)}$$

Table 41
Method call evaluation – Part 1

MethodInvocation

$$\frac{\Gamma \vdash (\xi, \mathcal{F}, h, Argument) \rightarrow (\xi', \mathcal{F}', h', Argument')}{\Gamma \vdash (\xi, \mathcal{F}, h, [C, D]\texttt{Identifier}(Argument)) \rightarrow (\xi', \mathcal{F}', h', [C, D]\texttt{Identifier}(Argument'))}$$

$$\frac{\begin{array}{c}\mathsf{InMethod}\ (M, \texttt{Identifier}, C, D) \\ \texttt{static} \in M.Modifiers \ \vee\ \texttt{private} \in M.Modifiers \\ \texttt{Id} = \mathsf{GetFormalParameter}(M) \\ \mathcal{F}' = \langle \bot, C, M, \mathcal{F}, \mathcal{P}, \bot \rangle \\ \mathcal{LV} = \mathcal{F}'.Method.LocalVariableTable \dagger [\texttt{Id} \mapsto v]\end{array}}{\begin{array}{c}\Gamma \vdash (\xi, \mathcal{F}, h, [C, D]\texttt{Identifier}(v), \mathcal{P}) \rightarrow \\ (\xi, \mathcal{F}'[Method.LocalVariableTable \leftarrow \mathcal{LV}], h, \mathcal{F}'.Method.Code)\end{array}}$$

$$\frac{\begin{array}{c}\mathsf{InMethod}\ (M, \texttt{Identifier}, C, D) \\ \texttt{static} \notin M.Modifiers \quad \texttt{private} \notin M.Modifiers \\ InvocMode = \mathsf{GetInvocMode}(M, \texttt{false}) \\ S = \mathsf{ConcreteType}(\mathcal{F}.\texttt{this}) \\ (M', R) = \mathsf{LookupFirstSuperClass}(\mathcal{F}.this, M, S, InvocMode) \\ \texttt{Id} = \mathsf{GetFormalParameter}(M') \\ \mathcal{F}' = \langle \mathcal{F}.\texttt{this}, R, M', \mathcal{F}, \mathcal{P}, \bot \rangle \\ \mathcal{LV} = \mathcal{F}'.Method.LocalVariableTable \dagger [\texttt{Id} \mapsto v]\end{array}}{\begin{array}{c}\Gamma \vdash (\xi, \mathcal{F}, h, [C, D]\texttt{Identifier}(v), \mathcal{P}) \rightarrow \\ (\xi, \mathcal{F}'[Method.LocalVariableTable \leftarrow \mathcal{LV}], h, \mathcal{F}'.Method.Code)\end{array}}$$

$$\frac{\Gamma \vdash (\xi, \mathcal{F}, h, Argument) \rightarrow (\xi', \mathcal{F}', h', Argument')}{\Gamma \vdash (\xi, \mathcal{F}, h, ClassType.[C, D]\texttt{Identifier}(Argument)) \rightarrow (\xi', \mathcal{F}', h', ClassType.[C, D]\texttt{Identifier}(Argument'))}$$

$$\frac{\begin{array}{c}\mathsf{InMethod}\ (M, \texttt{Identifier}, C, D) \\ \texttt{static} \in M.Modifiers \\ \texttt{Id} = \mathsf{GetFormalParameter}(M) \\ \mathcal{F}' = \langle \bot, ClassType, M, \mathcal{F}, \mathcal{P}, \bot \rangle \\ \mathcal{LV} = \mathcal{F}'.Method.LocalVariableTable \dagger [\texttt{Id} \mapsto v] \\ \neg\mathsf{initialized}(C)\end{array}}{\begin{array}{c}\Gamma \vdash (\xi, \mathcal{F}, h, ClassType.[C, D]\texttt{Identifier}(v), \mathcal{P}) \rightarrow \\ (\xi, \mathcal{F}'[Method.LocalVariableTable \leftarrow \mathcal{LV}], h, C.[C, ()\texttt{void}]\texttt{clinit}() \ ; \\ \mathcal{F}'.Method.Code)\end{array}}$$

Table 42
Method call evaluation – Part 2

MethodInvocation

$$\frac{\begin{array}{c}\mathsf{InMethod}\ (M, \texttt{Identifier}, C, D)\\ \texttt{static} \in M.Modifiers\\ \texttt{Id} = \mathsf{GetFormalParameter}(M)\\ \mathcal{F}' = \langle \bot, ClassType, M, \mathcal{F}, \mathcal{P}, \bot \rangle\\ \mathcal{LV} = \mathcal{F}'.Method.LocalVariableTable \dagger [\texttt{Id} \mapsto v]\\ \mathsf{initialized}(C)\end{array}}{\begin{array}{l}\Gamma \vdash (\xi, \mathcal{F}, h, ClassType.[C, D]\texttt{Identifier}(v), \mathcal{P}) \rightarrow\\ (\xi, \mathcal{F}'[Method.LocalVariableTable \leftarrow \mathcal{LV}], h, \mathcal{F}'.Method.Code)\end{array}}$$

$$\frac{\Gamma \vdash (\xi, \mathcal{F}, h, Primary) \rightarrow (\xi', \mathcal{F}', h', Primary')}{\begin{array}{l}\Gamma \vdash (\xi, \mathcal{F}, h, Primary.[C, D]\texttt{Identifier}(Argument)) \rightarrow\\ (\xi', \mathcal{F}', h', Primary'.[C, D]\texttt{Identifier}(Argument))\end{array}}$$

$$\frac{\Gamma \vdash (\xi, \mathcal{F}, h, Argument) \rightarrow (\xi', \mathcal{F}', h', Argument')}{\begin{array}{l}\Gamma \vdash (\xi, \mathcal{F}, h, \rho.[C, D]\texttt{Identifier}(Argument)) \rightarrow\\ (\xi', \mathcal{F}', h', \rho.[C, D]\texttt{Identifier}(Argument'))\end{array}}$$

$$\frac{\begin{array}{c}\mathsf{InMethod}\ (M, \texttt{Identifier}, C, D)\\ \texttt{static} \in M.Modifiers \quad \texttt{private} \notin M.Modifiers\\ \texttt{Id} = \mathsf{GetFormalParameter}(M)\\ \mathcal{F}' = \langle \bot, C, M, \mathcal{F}, \mathcal{P}, \bot \rangle\\ \mathcal{LV} = \mathcal{F}'.Method.LocalVariableTable \dagger [\texttt{Id} \mapsto v]\\ \neg\mathsf{initialized}(C)\end{array}}{\begin{array}{l}\Gamma \vdash (\xi, \mathcal{F}, h, \rho.[C, D]\texttt{Identifier}(v), \mathcal{P}) \rightarrow\\ (\xi, \mathcal{F}'[Method.LocalVariableTable \leftarrow \mathcal{LV}], h, C.[C, ()\texttt{void}]\texttt{clinit}() ;\\ \mathcal{F}'.Method.Code)\end{array}}$$

$$\frac{\begin{array}{c}\mathsf{InMethod}\ (M, \texttt{Identifier}, C, D)\\ \texttt{static} \in M.Modifiers \quad \texttt{private} \notin M.Modifiers\\ \texttt{Id} = \mathsf{GetFormalParameter}(M)\\ \mathcal{F}' = \langle \bot, C, M, \mathcal{F}, \mathcal{P}, \bot \rangle\\ \mathcal{LV} = \mathcal{F}'.Method.LocalVariableTable \dagger [\texttt{Id} \mapsto v]\\ \mathsf{initialized}(C)\end{array}}{\begin{array}{l}\Gamma \vdash (\xi, \mathcal{F}, h, \rho.[C, D]\texttt{Identifier}(v), \mathcal{P}) \rightarrow\\ (\xi, \mathcal{F}'[Method.LocalVariableTable \leftarrow \mathcal{LV}], h, \mathcal{F}'.Method.Code)\end{array}}$$

Table 43
Method call evaluation – Part 3

MethodInvocation

$$\frac{\begin{array}{c}\mathsf{InMethod}\ (M, \texttt{Identifier}, C, D) \\ \texttt{private} \in M.Modifiers \\ \texttt{Id} = \mathsf{GetFormalParameter}(M) \\ \mathcal{F}' = \langle \bot, C, M, \mathcal{F}, \mathcal{P}, \bot \rangle \\ \mathcal{LV} = \mathcal{F}'.Method.LocalVariableTable \dagger [\texttt{Id} \mapsto v]\end{array}}{\begin{array}{c}\Gamma \vdash (\xi, \mathcal{F}, h, \rho.[C, D]\texttt{Identifier}(v), \mathcal{P}) \ \rightarrow \\ (\xi, \mathcal{F}'[Method.LocalVariableTable \leftarrow \mathcal{LV}], h, \mathcal{F}'.Method.Code)\end{array}}$$

$$\frac{\begin{array}{c}\mathsf{InMethod}\ (M, \texttt{Identifier}, C, D) \\ \texttt{static} \notin M.Modifiers \\ \rho = \mathsf{fresh}(h) \\ h' = h \dagger [\rho \mapsto (\texttt{NullPointerException}, \ldots)]\end{array}}{\Gamma \vdash (\xi, \mathcal{F}, h, \texttt{null}.[C, D]\texttt{Identifier}(v), \mathcal{P}) \rightarrow (\rho, \mathcal{F}, h', \texttt{throw}\ \ \rho, \mathcal{P})}$$

$$\frac{\begin{array}{c}\mathsf{InMethod}\ (M, \texttt{Identifier}, C, D) \\ \texttt{static} \notin M.Modifiers \quad \texttt{private} \notin M.Modifiers \\ S = \mathsf{ConcreteType}(h(\rho)) \\ InvocMode = \mathsf{GetInvocMode}(M, \texttt{false}) \\ (M', R) = \mathsf{LookupFirstSuperClass}(\rho, M, S, InvocMode) \\ \texttt{Id} = \mathsf{GetFormalParameter}(M') \\ \mathcal{F}' = \langle \rho, R, M', \mathcal{F}, \mathcal{P}, \bot \rangle \\ \mathcal{LV} = \mathcal{F}'.Method.LocalVariableTable \dagger [\texttt{Id} \mapsto v]\end{array}}{\begin{array}{c}\Gamma \vdash (\xi, \mathcal{F}, h, \rho.[C, D]\texttt{Identifier}(v), \mathcal{P}) \ \rightarrow \\ (\xi, \mathcal{F}'[Method.LocalVariableTable \leftarrow \mathcal{LV}], h, \mathcal{F}'.Method.Code)\end{array}}$$

$$\frac{\begin{array}{c}\mathsf{InMethod}\ (M, \texttt{Identifier}, C, D) \\ \texttt{static} \notin M.Modifiers \quad \texttt{private} \notin M.Modifiers \\ InvocMode = \mathsf{GetInvocMode}(M, \texttt{false}) \\ (\bot, \bot) = \mathsf{LookupFirstSuperClass}(\rho, M, S, InvocMode)\end{array}}{\begin{array}{c}\Gamma \vdash (\xi, \mathcal{F}, h, \rho.[C, D]\texttt{Identifier}(v), \mathcal{P}) \ \rightarrow \\ (\xi, \mathcal{F}, h, \texttt{throw new AbstractMethodError}(), \mathcal{P})\end{array}}$$

$$\frac{\Gamma \vdash (\xi, \mathcal{F}, h, FieldName) \rightarrow (\xi', \mathcal{F}', h', FieldName')}{\begin{array}{c}\Gamma \vdash (\xi, \mathcal{F}, h, FieldName.[C, D]\texttt{Identifier}(Argument)) \ \rightarrow \\ (\xi', \mathcal{F}', h', FieldName'.[C, D]\texttt{Identifier}(Argument))\end{array}}$$

Table 44
Method call evaluation – Part 4

MethodInvocation

$$\frac{\Gamma \vdash (\xi, \mathcal{F}, h, Argument) \rightarrow (\xi', \mathcal{F}', h', Argument')}{\Gamma \vdash (\xi, \mathcal{F}, h, \rho.[C, D]\texttt{Identifier}(Argument)) \rightarrow (\xi', \mathcal{F}', h', \rho.[C, D]\texttt{Identifier}(Argument'))}$$

$$\frac{\Gamma \vdash (\xi, \mathcal{F}, h, Argument) \rightarrow (\xi', \mathcal{F}', h', Argument')}{\Gamma \vdash (\xi, \mathcal{F}, h, \texttt{super}.[C, D]\texttt{Identifier}(Argument)) \rightarrow (\xi', \mathcal{F}', h', \texttt{super}.[C, D]\texttt{Identifier}(Argument'))}$$

$$\frac{\begin{array}{c}\mathsf{InMethod}\ (M, \texttt{Identifier}, C, D) \\ \texttt{static} \in M.Modifiers \\ \mathsf{Id} = \mathsf{GetFormalParameter}(M) \\ \mathcal{F}' = \langle \bot, C, M, \mathcal{F}, \mathcal{P}, \bot \rangle \\ \mathcal{LV} = \mathcal{F}'.Method.LocalVariableTable \dagger [\mathsf{Id} \mapsto v]\end{array}}{\begin{array}{c}\Gamma \vdash (\xi, \mathcal{F}, h, \texttt{super}.[C, D]\texttt{Identifier}(v)) \rightarrow \\ (\xi, \mathcal{F}'[Method.LocalVariableTable \leftarrow \mathcal{LV}], h, \mathcal{F}'.Method.Code)\end{array}}$$

$$\frac{\begin{array}{c}\mathsf{InMethod}\ (M, \texttt{Identifier}, C, D) \\ \texttt{static} \notin M.Modifiers \\ InvocMode = \mathsf{GetInvocMode}(M, \texttt{true}) \\ (M', R) = \mathsf{LookupFirstSuperClass}(\mathcal{F}.\texttt{this}, M, \mathcal{F}.Class.SuperClass, InvocMode) \\ \mathsf{Id} = \mathsf{GetFormalParameter}(M') \\ \mathcal{F}' = \langle \mathcal{F}.\texttt{this}, R, M', \mathcal{F}, \mathcal{P}, \bot \rangle \\ \mathcal{LV} = \mathcal{F}'.Method.LocalVariableTable \dagger [\mathsf{Id} \mapsto v]\end{array}}{\begin{array}{c}\Gamma \vdash (\xi, \mathcal{F}, h, \texttt{super}.[C, D]\texttt{Identifier}(v)) \rightarrow \\ (\xi, \mathcal{F}'[Method.LocalVariableTable \leftarrow \mathcal{LV}], h, \mathcal{F}'.Method.Code)\end{array}}$$

$$\frac{\begin{array}{c}\mathsf{InMethod}\ (M, \texttt{Identifier}, C, D) \\ \texttt{static} \notin M.Modifiers \\ InvocMode = \mathsf{GetInvocMode}(M, \texttt{true}) \\ (\bot, \bot) = \mathsf{LookupFirstSuperClass}(\mathcal{F}.\texttt{this}, M, \mathcal{F}.Class.SuperClass, InvocMode)\end{array}}{\begin{array}{c}\Gamma \vdash (\xi, \mathcal{F}, h, \texttt{super}.[C, D]\texttt{Identifier}(v), \mathcal{P}) \rightarrow \\ (\xi, \mathcal{F}, h, \texttt{throw new AbstractMethodError}(), \mathcal{P})\end{array}}$$

Table 45
Assignment evaluation – Part 1

AssignmentExpression

$$\frac{\Gamma \vdash (\xi, \mathcal{F}, h, \mathit{SimpleFieldAccess}) \to (\xi', \mathcal{F}', h', \mathit{SimpleFieldAccess}')}{\Gamma \vdash (\xi, \mathcal{F}, h, \mathit{SimpleFieldAccess} = \mathit{Expression}) \to (\xi', \mathcal{F}', h', \mathit{SimpleFieldAccess}' = \mathit{Expression})}$$

$$\frac{(\xi, \mathcal{F}, h, \mathit{Expression}) \to (\xi', \mathcal{F}', h', \mathit{Expression}')}{\Gamma \vdash (\xi, \mathcal{F}, h, \rho.[C, D]\texttt{Identifier} = \mathit{Expression}) \to (\xi', \mathcal{F}', h', \rho.[C, D]\texttt{Identifier} = \mathit{Expression}')}$$

$$\frac{\begin{array}{c}\mathsf{InField}\ (f, \texttt{Identifier}, C, D) \\ F = \langle f, C \rangle \\ \texttt{static} \notin F.\mathit{Field.Modifiers} \quad \texttt{final} \notin F.\mathit{Field.Modifiers} \\ C' = \mathsf{ConcreteType}(h(\rho)) \\ MF = \mathsf{GetMappeFields}(h(\rho)) \\ h' = h \dagger [\rho \mapsto (C', MF \dagger [F \mapsto v])]\end{array}}{\Gamma \vdash (\xi, \mathcal{F}, h, \rho.[C, D]\texttt{Identifier} = v) \to (\xi, \mathcal{F}, h', v)}$$

$$\frac{\begin{array}{c}\mathsf{InField}\ (f, \texttt{Identifier}, C, D) \\ F = \langle f, C \rangle \\ \texttt{static} \in F.\mathit{Field.Modifiers} \quad \texttt{final} \notin F.\mathit{Field.Modifiers} \\ \neg\mathsf{initialized}(C)\end{array}}{\Gamma \vdash (\xi, \mathcal{F}, h, \rho.[C, D]\texttt{Identifier} = v) \to (\xi, \mathcal{F}, h, C.[C, ()\texttt{void}]\texttt{clinit}() \ ; \ \rho.[C, D]\texttt{Identifier} = v)}$$

Table 46
Assignment evaluation – Part 2

AssignmentExpression

$$\frac{\begin{array}{c}\mathsf{InField}\ (f, \texttt{Identifier}, C, D) \\ F = \langle f, C \rangle \\ \texttt{static} \in F.\mathit{Field.Modifiers} \quad \texttt{final} \notin F.\mathit{Field.Modifiers} \\ \mathsf{initialized}(C) \\ h' = h \dagger [F \mapsto v]\end{array}}{\Gamma \vdash (\xi, \mathcal{F}, h, \rho.[C, D]\texttt{Identifier} = v) \to (\xi, \mathcal{F}, h', v)}$$

$$\frac{\begin{array}{c}\mathsf{InField}\ (f, \texttt{Identifier}, C, D) \\ F = \langle f, C \rangle \\ \texttt{static} \in F.\mathit{Field.Modifiers} \quad \texttt{final} \notin F.\mathit{Field.Modifiers} \\ \neg\mathsf{initialized}(C)\end{array}}{\Gamma \vdash (\xi, \mathcal{F}, h, \mathit{ClassType}.[C, D]\texttt{Identifier} = v) \to (\xi, \mathcal{F}, h, C.[C, ()\texttt{void}]\texttt{clinit}() \ ; \ \mathit{ClassType}.[C, D]\texttt{Identifier} = v)}$$

$$\frac{\begin{array}{c}\mathsf{InField}\ (f, \texttt{Identifier}, C, D) \\ F = \langle f, C \rangle \\ \texttt{static} \in F.\mathit{Field.Modifiers} \quad \texttt{final} \notin F.\mathit{Field.Modifiers} \\ \mathsf{initialized}(C) \\ h' = h \dagger [F \mapsto v]\end{array}}{\Gamma \vdash (\xi, \mathcal{F}, h, \mathit{ClassType}.[C, D]\texttt{Identifier} = v) \to (\xi, \mathcal{F}, h', v)}$$

$$\frac{\begin{array}{c}\mathsf{InField}\ (f, \texttt{Identifier}, C, D) \\ F = \langle f, C \rangle \\ \texttt{static} \in F.\mathit{Field.Modifiers} \quad \texttt{final} \notin F.\mathit{Field.Modifiers} \\ h' = h \dagger [F \mapsto v]\end{array}}{\Gamma \vdash (\xi, \mathcal{F}, h, [C, D]\texttt{Identifier} = v) \to (\xi, \mathcal{F}, h', v)}$$

Table 47
Assignment evaluation – Part 3

AssignmentExpression .....

$$\frac{\begin{array}{c}\mathsf{InField}\ (f, \texttt{Identifier}, C, D) \\ F = \langle f, C\rangle \\ \texttt{static} \notin F.Field.Modifiers \\ \rho = \mathcal{F}.\texttt{this} \\ C' = \mathsf{ConcreteType}(h(\rho)) \\ MF = \mathsf{GetMappeFields}(h(\rho)) \\ h' = h \dagger [\rho \mapsto (C', MF \dagger [F \mapsto v])]\end{array}}{\Gamma \vdash (\xi, \mathcal{F}, h, [C, D]\texttt{Identifier} = v) \rightarrow (\xi, \mathcal{F}, h', v)}$$

$$\frac{\begin{array}{c}\mathsf{InField}\ (f, \texttt{Identifier}, C, D) \\ F = \langle f, C\rangle \\ \texttt{static} \in F.Field.Modifiers \\ h' = h \dagger [F \mapsto v]\end{array}}{\Gamma \vdash (\xi, \mathcal{F}, h, \texttt{super}.[C, D]\texttt{Identifier} = v) \rightarrow (\xi, \mathcal{F}, h', v)}$$

$$\frac{\begin{array}{c}\mathsf{InField}\ (f, \texttt{Identifier}, C, D) \\ F = \langle f, C\rangle \\ \texttt{static} \notin F.Field.Modifiers \\ \rho = \mathcal{F}.\texttt{this} \\ C' = \mathsf{ConcreteType}(h(\rho)) \\ MF = \mathsf{GetMappeFields}(h(\rho)) \\ h' = h \dagger [\rho \mapsto (C', MF \dagger [F \mapsto v])]\end{array}}{\Gamma \vdash (\xi, \mathcal{F}, h, \texttt{super}.[C, D]\texttt{Identifier} = v) \rightarrow (\xi, \mathcal{F}, h', v)}$$

$$\frac{(\xi, \mathcal{F}, h, Expression) \rightarrow (\xi', \mathcal{F}', h', Expression')}{\Gamma \vdash (\xi, \mathcal{F}, h, \texttt{Identifier} = Expression) \rightarrow (\xi', \mathcal{F}', h', \texttt{Identifier} = Expression')}$$

Table 48
Assignment evaluation – Part 4

AssignmentExpression

$$\frac{\mathcal{LV} = \mathcal{F}.Method.LocalVariableTable \dagger [\texttt{Identifier} \mapsto v]}{\Gamma \vdash (\xi, \mathcal{F}, h, \texttt{Identifier} = v) \rightarrow (\xi, \mathcal{F}[Method.LocalVariableTable \leftarrow \mathcal{LV}], h, v)}$$

$$\frac{\Gamma \vdash (\xi, \mathcal{F}, h, ArrayFieldAccess) \rightarrow (\xi', \mathcal{F}', h', ArrayFieldAccess')}{\Gamma \vdash (\xi, \mathcal{F}, h, ArrayFieldAccess = Expression) \rightarrow (\xi', \mathcal{F}', h', ArrayFieldAccess' = Expression)}$$

$$\frac{\Gamma \vdash (\xi, \mathcal{F}, h, Expression) \rightarrow (\xi', \mathcal{F}', h', Expression')}{\Gamma \vdash (\xi, \mathcal{F}, h, \rho[v] = Expression) \rightarrow (\xi', \mathcal{F}', h', \rho[v] = Expression')}$$

$$\frac{\begin{array}{c} v_1 \geq 0 \wedge v_1 < h(\rho).\texttt{length} \\ MI = \mathsf{GetMappeIndex}(h(\rho)) \\ \mathsf{isRefType}(\mathsf{ConcreteType}(v_2)) \\ (\mathsf{ConcreteType}(v_2) \sqsubseteq \mathsf{ConcreteType}(h(MI(v_1)))) \\ C' = \mathsf{ConcreteType}(h(\rho)) \\ h' = h \dagger [\rho \mapsto (C', MI \dagger [v_1 \mapsto v_2])] \end{array}}{\Gamma \vdash (\xi, \mathcal{F}, h, \rho[v_1] = v_2) \rightarrow (\xi, \mathcal{F}, h', v_2)}$$

$$\frac{\begin{array}{c} \rho' = \mathsf{fresh}(h) \\ h' = h \dagger [\rho' \mapsto (\texttt{ArrayStoreException}, \ldots)] \\ MI = \mathsf{GetMappeIndex}(h(\rho)) \\ \mathsf{isRefType}(\mathsf{ConcreteType}(v_2)) \\ \neg(\mathsf{ConcreteType}(v_2) \sqsubseteq \mathsf{ConcreteType}(h(MI(v_1)))) \end{array}}{\Gamma \vdash (\xi, \mathcal{F}, h, \rho[v_1] = v_2, \mathcal{P}) \rightarrow (\rho', \mathcal{F}, h', \texttt{throw } \rho', \mathcal{P})}$$

$$\frac{\begin{array}{c} v_1 \geq 0 \wedge v_1 < h(\rho).\texttt{length} \\ MI = \mathsf{GetMappeIndex}(h(\rho)) \\ \neg\mathsf{isRefType}(\mathsf{ConcreteType}(v_2)) \\ C' = \mathsf{ConcreteType}(h(\rho)) \\ h' = h \dagger [\rho \mapsto (C', MI \dagger [v_1 \mapsto v_2])] \end{array}}{\Gamma \vdash (\xi, \mathcal{F}, h, \rho[v_1] = v_2) \rightarrow (\xi, \mathcal{F}, h', v_2)}$$

Mourad ebbabi holds Ph.D. and M.Sc. degrees in computer science from Paris- I Orsay, University, France. He is a Lead Scientist at Panasonic Information and Networking Technologies Laboratory, Princeton, New Jersey, USA. He is pursuing research on middleware for next-generation networks. He is the Specification Lead of four JAIN (Java Intelligent Networks) Java Specification Requests (JSRs) dedicated to the elaboration of standard specifications for presence and instant messaging through the Java Community Process (JCP) program. He is also a member of the JAIN Council and participates at the JCP Executive Committee. He is also a Tenured Associate Professor (on leave) at the Computer Science Department of Laval University, uebec, Canada. In the past, he served as a Senior Scientist at General Electric Research Center, New York, USA Research Associate at the Computer Science Department of Stanford University, California, USA and Permanent Researcher at the Bull Corporate Research Center, Paris, France. He published more than 40 research papers in international journals and conferences on Java technology security and acceleration, crypto-protocol analysis, malicious code detection, programming languages, formal semantics, type theory and specification and verification of safety-critical systems.

e-mail: debbabim research.panasonic.com
Panasonic Information
Networking Technologies Laboratory
Two Research Way
Princeton, New Jersey 08540, USA
e-mail: debbabi ift.ulaval.ca
LSFM Research Group, Computer Science Department
Laval University
uebec, G1K 7P4, Canada

adia Tawbi holds Ph.D. and M.Sc. degrees in computer science from Paris-VI University, France. She is a Tenured Associate Professor at the Computer Science Department of Laval University, uebec, Canada. Before that she served as a Group Leader and a Permanent Researcher at the Bull Corporate Research Center, Paris, France. She published several research papers in international journals and conferences on computer security, crypto-protocol analysis, malicious code detection, programming languages, static analysis, formal semantics, and specification and verification of safety-critical systems.

e-mail: tawbi ift.ulaval.ca
LSFM Research Group, Computer Science Department
Laval University
uebec, G1K 7P4, Canada

amdi ahyaoui is a researcher at the LSFM (Languages Semantics and Formal Methods) Research Group at the Computer Science Department of Laval University, uebec, Canada. He is pursuing actively a Ph.D. thesis on the use of semantic techniques to accelerate and secure Java technologies. He holds an M.Sc. degree from Laval University. In the past years, Hamdi Yahyaoui worked on the formal dynamic semantics of Java and also on the control flow analysis of Java. He participated in the implementation of an LSFM Java optimizing compiler.

e-mail: hamdi ift.ulaval.ca
LSFM Research Group, Computer Science Department
Laval University
uebec, G1K 7P4, Canada