JOURNAL OF OBJECT TECHNOLOGY

Online at www.jot.fm. Published by ETH Zurich, Chair of Software Engineering. ©JOT, 2002

Vol. 6, No. 8, September-October 2007

A Formal Type System for Java

Mourad Debbabi and Myriam Fourati
Computer Security Laboratory
Concordia University

The primary objective of this paper is threefold: First, we present an evaluation of the state of the art on Java static semantics. Accordingly, we discuss the completeness and the soundness of the most prominent proposals recently advanced in the literature. Moreover, we discuss their compliance with respect to the Java language specification. Second, we report a brief evaluation of the official Java language specification. Third, we show how the definition of a realistic static semantics for full Java could be addressed. We exhibit the technical difficulties and discuss the semantic traits that could be used to address them.

1 INTRODUCTION

Nowadays, Java is a very popular programming language. It is multi-paradigmatic since it reconciles imperative, object-oriented, concurrent and distributed programming styles. Since its launching in 1995 by Sun Microsystems, Java is perceived as a revolution into the programming language community. This is mainly due to the attractive features that have been incorporated into the language. One of the most major innovations of Java is the support of mobile code through the concept of applet. The latter is a small compiled unit of code that can migrate from one site to another and be executed on any platform that is Java-enabled i.e. any platform endowed with a Java virtual machine. Furthermore, these applets could be embedded in world wide pages to achieve general-purpose tasks. Java is now widely licensed and firmly established in the industry.

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 it 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 [8, 18]. All these factors justify the need for robust theoretical foundations for Java.

Several corrections and modifications were made to the Java language specification [10, 13]. Also, several errors were found in its implementations. This is understandable since the language combines attractive but complex features, which makes its semantics far from being straightforward and leads to subtleties and complex behaviors. The only available specification of Java [9, 12] 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 really acceptable especially when it comes to security and safety/security critical deployement of the Java technology.

We believe that a formal study of Java is very useful to clarify, correct and complete its semantics. It can only be advantageous to consolidate and establish Java security, and make it possible to reason about effective compilation. In our view, a semantics theory for Java is not a luxury, it is a necessity.

The interest in the Java language lead to an emergence of several formalizations at the language level as well as at the virtual machine level. All of them have a merit : Each one contributes to better understanding of Java. Nevertheless, several important features of the language have not been formalized yet, and some constructions have been simplified. It is well known that these works have proved the type soundness for different Java subsets, but none of these type safe subsets is, in fact, Java.

In this paper, we propose a formal static semantics covering almost all the aspects of the Java language, except for packages and inner classes, which we will treat in a later phase of our work. This includes primitive types, classes with inheritance, interfaces, instance and `static` methods and fields, methods hiding, overriding and overloading, shadowing of fields, constructors, the initialization of fields and variables, local variables as well as their scope and initialization, arrays, `this`, `super`, assignments, object creation, methods call, fields and variables access, modifiers and the processing of all the exceptions including their declaration, throwing and handling. Our formalization is a contribution in the static semantic field of Java since no study hitherto. We believe that our work will help to reason about the semantics of Java and to clarify its informal specification.

The rest of the paper is organized as follows. Section 2 is a brief compilation of the state of the art work on Java semantics, with a discussion of their completeness, soundness and compliance. Section 3 is devoted to an evaluation of the published Java language specification and it exposes subtleties of the Java semantics. Our approach is presented in Section 4. In Section 5, we introduce the syntax of the Java language formalized. The static semantics is detailed in Section 6. Finally, some concluding remarks together with a discussion of ongoing and future research are ultimately reported in Section 7.

2 RELATED WORK

There exist several works on the formalization of the Java language, in particular on the proof of soundness of the Java type system. This section presents some of them and discusses their completeness and compliance with the official specification of Java.

At the language level, Sophia Drossopoulou and Susan Eisenbach [5, 6, 7, 19] have proposed a static semantics, an operational semantics and also a proof of a type soundness of a subset of Java. The latter includes primitive types, classes and inheritance, instance variables and instance methods, interface methods, shadowing of instance variables, dynamic method biding, object creation, the `null` value, arrays, and some exceptions. The latter have been extended [20] and its formalization improved by the addition of exception throwing and the associated `Throws` clauses. Nevertheless, some restrictions and simplifications have been considered in the formalization of this subset.

Heavily based on the work above-named, Don Syme [21] has mechanically specified and verified, using a higher-order declarative proof system DECLARE, the formalization of the subset of Java proposed by Drossopoulou and Eisenbach. Syme corrected some mistakes and clarified many details on the work of Drossopoulou and Eisenbach. Furthermore, he brought less simplifications to the Java language. For example, he did consider the predefined class `Object`.

As part of the BALI project, and simultaneously with Syme's works [15, 16, 14], Tobias Nipkow and David von Oheimb proved the type safety of a subset of Java very similar to Drossopoulou's one. In this work, all definitions and proofs have been done in the theorem prover Isabelle/HOL.

Isabelle Attali, Denis Caromel and Majorie Russo [2, 11] proposed a formal dynamic semantics of a subset of Java, including inheritance, dynamic linking and multi-threading. They used the natural semantics within the Centaur system and the Typol formalism.

All formalizations of Java proposed in the literature contributed to remove some ambiguities from the language. However, several important features have been left out, even though these features have contributed to its expansion. In particular, threads, that are an essential keystone of Java, and modifiers, that strengthen security in the language, have been ignored. Among the other subtle features that have not been considered yet we cite:

- Initialization,
- Constructors,
- Scoping of variables,
- Packages.

Besides, all the formalizations proposed for Java, are based on simplifications. For instance, they assume that there is exactly one `return` statement in each method body, and that it is the last one. In addition, the formalization of some constructs of the Java language is incomplete. As an example, the exceptions handling exclude their declaration by the `throws` clauses. The processing of the latter is not straightforward, and requires several verifications by the type system. We think that the development of a formal static semantics covering these features raises several technical difficulties which will be shown through this work.

Below we describe some errors that we discovered in a few of the works above-named:

- In [16], the `this` expression is modeled as a special local variable, but the `super` construct has not been formalized. Nonetheless, the authors said that the latter can be simulated by a `this` expression that is cast to the superclass of the current class. This is not possible, since a cast to a superclass type is not effective in attempting to access an overridden method [9].
- The type system must check that a Java program includes handlers for *checked exceptions* that result from the execution of a method or a constructor. However, in [16], the authors did not envisage any particular processing when the checked exception classes `Throwable` and `Exception`, respectively, are raised.
- In [16], the authors introduced the type NullT as the null type, and they considered it as a Bali type. This created some errors in their type system. For example, in Bali, it is possible to create an array with the component type of NullT, nevertheless the latter is not a type in Java. We think that this problem is due to the non distinction between semantics and syntactic types.
- In [16] and also in [7], arrays don't have members, nevertheless arrays inherit members from the class `Object`, and declare a field named `length` as well as a method named `clone`.
- Also in [16], starting with the typing rules, it is possible to type an array creation expression as follows: $E \vdash$ `new` $int[][i] :: int[][]$, which does not make sense and is illegal in Java.

Lately, several research initiatives targeted the study of the static semantics of small subsets of Java for static analysis [1, 4, 17] and security [3]. However, it remains that there is no complete work that grasps the underpinning of almost the whole Java language semantics and exhibits the issues related to the elaboration of a fully-fledged static semantics of it.

3 EVALUATION OF THE JAVA LANGUAGE AND ITS SPECIFICATION

Java is a powerful programming language that has considerable potential especially in the field of mobile code. However, the language involves subtleties and exceptions, which make its semantics far from straightforward. On that account, the Java Language Specification [9, 12] turned out to be very useful to help reasoning about the language and to make its formalization easier. We think that without this specification, it is impossible to develop a formal semantics for Java. Nevertheless, the specification is a textual and an informal description that may be erroneous and ambiguous. Moreover, the Java language has been evolving since the publication of the first edition of the official Java Specification in 1996 [12], and new features had to be covered by the Specification. Furthermore, the first edition has been shown to have many errors and omissions. Recently, a second edition of the Java Language Specification was published [9], but many problems rest unresolved [10, 13]. Besides that, this new edition contains yet unknown errors and subtleties. Below, we present some of them.

- The chapter 5 of [9] describes the cast conversion between types, and it stipulates that casting between two interface types that declare `abstract` methods with the same signature and different return types is type-incorrect, since no class could implement both of them. However, if two interfaces contain methods with the same signature and incompatible clause `throws`, they cannot be implemented by a single class, but in this case the casting between such interface types is not prohibited.

- The Java Language Specification [9, chapter 14] states that the expression in the `throw` statement must denote a variable or a value of a reference type that is assignable to the `Throwable` type. However, although the type of the `null` literal is assignable to any reference type, and then to the reference type `Throwable`, it is incorrect to throw the null reference.

- Also in [9, chapter 14], it is reported that the expression in the `synchronized` statement must be of a reference type. Since the type of `null` is a reference type, we can deduce that the following program portion is type-correct:

  ```
  class C {
     void f ( )  { synchronized ( null ) { } }
  }
  ```

  This is the case if we use the JDK 1.1.7 version of the Java compiler or one of its former versions; on the other hand, the compilation of this same example using JDK 1.2 does not pass without errors. We think that by this restriction, it is intended to detect the `NullPointerException` error as soon as possible during the compile-time step, rather than wait for the execution to report it. However, what is confusing is that even though `synchronized` argument evaluates to the `null` reference, the following program portion is accepted by JDK 1.2:

  ```
  class C {
     void f ( )  { synchronized ( (T)null ) { } }
  ```

  These examples prove that it is difficult to figure out the semantics of the Java language, since its specification is erroneous and the behaviors of the Java compilers are sometimes rather incomprehensible.

- Also in [9, chapter 14], a `catch` block `C` is considered to be reachable if and only if both of the following conditions are valid:

  - Some expression or `throw` statement in the `try` block is reachable and can throw an expression that is of a type assignable to the `catch` parameter type.
  - There is no earlier `catch` block `A` in the `try` statement such that the type of `C`'s parameter is the same as or a subclass of `A`'s one.

  However, this is contradictory with the behavior of the Java compiler. Indeed, when the block `try` is empty, and then does not throw any exception, the `catch` block is reachable. This is the fact for all the JDK versions.

- During the course of our work, we noticed that some errors have slipped into the examples given in the Java Language Specification. Although most of them are not serious, they could lead to some confusion. As an example, the following program excerpts from [9, chapter 8] are incorrect:

```
interface Fish { int getNumberOfScales( ); }
interface Piano { int getNumberOfScales( ); }
class Tuna implements Fish, Piano {
   int getNumberOfScales( ) { return 91 ; }
}
```

  Indeed, the class `Tuna` implements the methods named `getNumberOfScales` inherited from its superinterfaces with a method that provides less access than them, since the methods of interfaces are implicitly `public`, while those of classes are `package`.

- Several phrases in the Java Specification are ambiguous. To figure out their meaning, we have done some tests on Java programs. To illustrate this, we present bellow an example of an ambiguous paragraph [9, chapter 16]:

  "V is definitely assigned after a local variable declaration statement that contains at least one variable initializer if and only if either it is definitely assigned after the last initializer expression in the local variable statement or the last initializer expression in the declaration is in the declarator that declares V".

In the rest of this section, we give some examples that illustrate the Java semantics subtleties and problems with the JDK compiler:

- When a class or an interface inherits two `abstract` methods with the same signature that have different `throws` clauses, a Java compiler must verify that these methods could be implemented by a single class. The version 1.2 of

JDK compiler requires that the declared exception types in different methods must be related. Thus, the implementing method must declare the narrower of these class types. We found this restriction rather strict, since inherited methods could be implemented by a method that throws no exceptions. We think that this problem must emerge later, when a method, which implements the inherited methods with the same signature, is defined.

- In Java, the statement `while ( false )` *Statement* is not type-correct, since *Statement* is unreachable. However, when we declare a constant variable `b` that has the `false` value, the statement `while ( b )` *Statement* is compiled without errors.
- The statement `int v = 5 ;` below is not considered unreachable:

  ```
  while ( false ) ;
  int v = 5 ;
  ```

  we can deduce that this is due to the fact that the `while`-statement is empty. However, it is surprising to note that the statement `int v = 5 ;` would be considered unreachable if the `while`-statement were `{;}`. Therefore, the specification does not specify a need for a particular treatment when the `while`-statement is empty.

4 OUR APPROACH

In this paper and unlike previous works, we propose a static semantics for almost the whole of the Java language rather than just a subset of it. We have considered all details even the flow analysis carried out by the Java compiler to treat the variable initialization and unreachable statements. We believe that all Java features are important in modeling, since all of them play a part in the Java security. The only aspects of the language that we have not considered yet are inner classes and packages. The semantics of packages is very subtle and their specification contains errors and ambiguities. Therefore, we have decided to integrate them, as well as the inner classes, in a further stage of our work.

The specification that we adopt is mainly based on the Java Language Specification [9, 12] enriched by amendments and clarifications found in [10, 13]. To solve the ambiguities and incompleteness of the Sun's specification, we used reasoning and ran a number of tests on different versions of Sun's JDK.

During our work, we were faced with many problems. The biggest were the specification's ambiguity, colossal size and incorrectness. Besides, we were forced to deal with non-compositionality and non-uniformity of the Java semantics. A part of the complexity comes from Java's subtlety.

Below, we briefly present some problems that we faced during the formalization of the language, along with the proposed solutions. Significant difficulties are raised by the semantics of syntactical constructions, which are context-dependent. In other words, semantics may depend on the previously evaluated expressions or statements. This makes a naive semantics treatment affect the compositionality of the semantics. For instance, when an expression is used as an argument in an explicit construction invocation statement, it cannot refer to an instance variable or method, or use `this` or `super`. Therefore, when we type an expression, we have to know where it occurs in the program being typed. Typing of exceptions presents another problem. There exist, in the Java language, two kind of exceptions: *checked* exceptions and *unchecked* ones. The unchecked exceptions classes are the classes `RuntimeException`, `Error` and their subclasses. All other exceptions classes are checked exceptions classes. Unchecked exceptions are exempted from type-checking [9, chapter 11]. On the other hand, all checked exceptions that can result from the execution of a method or a constructor have to be propagated until they are either handled by a `catch` block in their bodies or declared in their `throws` clauses. Moreover, the treatment of variable initialization is not straightforward. For example, we have to check that no variable is used before its initialization. To do so, we considered only the assignment expressions for that variable prior to its initialization, and we verify that there is at least one assignment expression for that variable in all possible execution paths. Yet another difficulty comes from having to distinguish if a construction to be typed is a class or instance member. It is important, in order to verify that `this` or `super` are not invoked in the body of a static member. Among the other problems are the formalization of unreachable statements and the need to make sure that no field is used before its declaration.

In order to solve the above-mentioned problems, we introduced additional information in our typing rules. This information is updated as and when required as we move forward through the program, this will be shown in the sequel. Besides, we had no choice but to religiously follow the tedious and non-uniform requirements of the official Java specification.

Along this document, we use a neat formatting conventions: the reserved words (terminals) of the Java language are written in typewriter font (e.g., "`throw`" and "`[`"), nonterminals are written in italic font with initial capital letters and other uppercase letters (e.g., "*ClassDeclaration*", "*Modifiers*"), the meta-language is described using the sans serif font (e.g., wellFormedEnv) and the meta-variables appear in italics font (e.g.,*rt*).

5 SYNTAX

The syntax definition of the Java language considered in this paper is based mainly on the syntax given in the Java Language Specification [9, 12]. It covers all the aspects of the Java language, except for packages and inner classes. This includes primitive types, classes and inheritance, interfaces, instance methods and variables, class methods and variables, object creation, arrays, exception declaration throwing and handling, constructors, modifiers, initialization of fields and variables, assignment, `this` and `super`, proceeding of the unreachable statements and scoping rules for variables. However, in order to lighten the formalization and increase the readability of the typing rules, we did not consider statements that can be coded in terms of some syntactic constructions of our language. In particular, the syntactic variants of loops like the `loop` and `for` statements, as well as the jumps statements like `break` can be replaced by a combination of conditionals and recursion statements. As for expressions, our syntax excludes the standard unary and binary operators as their typing is simple. Furthermore, in the Java language, the `[]`, in an array declaration, may appear as part of the array name or as part of its type. We only considered the latter. Our syntax includes only uni-dimensional arrays, since an extension to a multi-dimensional ones is quite simple. As in [16], and to lighten the formalization, we assume that each method has exactly one parameter. Finally, and to lighten the semantics, our syntax omits some variations introduced by the new version of the Java Specification [9], such as declaration of local variables with modifiers, or new way of method invocation ($Name.super.Identifier(Arguments)$).

Keywords

The set of keywords consists of reserved words of the Java language formalized in this paper. It is introduced in the Figure 1.

```
abstract    boolean    byte    class    double    else    extends
final   finally   float   if   implements   int   interface   long
native   new   private   public   return   short   static   super
synchronized    this    throw    throws    transient    try    void
volatile   while   ( )   [ ]   { }   ;   ,   .   =
```

Figure 1: Java keywords.

Grammar

This section introduces the grammatical rules for the Java language used in this work. Each rule consists of nonterminal followed by the symbol "::=" and a sequence of nonterminals, terminals and operators. The symbol "ε" represents the empty word. More formally, the BNF notation of our language is presented in Figures 2, 3, 4 and 5. We use as far as possible the same nonterminal notations as in [9, 12]. This will be useful when the reader wants to refer to the Java Language Specification [9, 12]. For the sake of the formalization, we have changed some grammatical rules. As an example, we decide to split the rule of a local variable access into to categories: a simple access to a variable and an access to a variable that appears in a left-hand side of an assignment operator. This is necessary to verify that a local variable is initialized before its use. We also distinguish between a non-array local variable, a non-array field and an array access and assignment, since their treatment differs.

Program	::=	*ClassDeclaration Program*
	|	*InterfaceDeclaration Program*
	|	ε
ClassDeclaration	::=	*Modifiers* `class` `Identifier` *Extends Implements*
	|	`{`*ClassBody*`}`
Modifiers	::=	`public` *Modifiers*
	|	`private` *Modifiers*
	|	`static` *Modifiers*
	|	`abstract` *Modifiers*
	|	`final` *Modifiers*
	|	`synchronized` *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*

Figure 2: Grammar of Java: part 1.

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*
LocalVariableDeclaration	::=	*Type* `Identifier` `;`

Figure 3: Grammar of Java: part 2.

```
                              |   Type Identifier = Expression
                              |   SimpleType[] Identifier = ArrayInitializer ;

Statement                    ::=  ;
                              |   Block
                              |   ExpressionStatement
                              |   IfStatement
                              |   WhileStatement
                              |   ThrowStatement
                              |   SynchronizedStatement
                              |   TryStatement
                              |   ReturnStatement

ExpressionStatement          ::=  StatementExpression ;

StatementExpression          ::=  AssignmentExpression
                              |   MethodInvocation
                              |   ClassInstanceCreation

IfStatement                  ::=  if ( Expression ) Statement else Statement
                              |   if ( Expression ) Statement

WhileStatement               ::=  while ( Expression ) Statement

ThrowStatement               ::=  throw Expression ;

SynchronizedStatement        ::=  synchronized ( Expression ) Block

TryStatement                 ::=  try Block Catches finally Block
                              |   try Block Catches
                              |   try Block finally Block

ReturnStatement              ::=  return Expression ;
                              |   return ;

Catches                      ::=  Catch
                              |   Catch Catches

Catch                        ::=  catch ( ClassType Identifier ) Block

Primary                      ::=  ArrayCreation
                              |   PrimaryNoNewArray

ArrayCreationExpression      ::=  new SimpleType [ Expression ]

PrimaryNoNewArray            ::=  Literal
```

Figure 4: Grammar of Java: part 3.

```
                         |    this
                         |    ( Expression )
                         |    ClassInstanceCreation
                         |    SimpleFieldAccess
                         |    ArrayFieldAccess
                         |    MethodInvoction

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 ]
```

Figure 5: Grammar of Java: part 4.

$$
\begin{array}{llll}
\tau_r & \in & \mathit{ResultType} & ::= \mathit{Type} \mid \texttt{void} \\
\tau & \in & \mathit{Type} & ::= \mathit{PrimitiveType} \mid \mathit{ReferenceType} \\
\rho & \in & \mathit{ReferenceType} & ::= \mathit{ClassOrInterfaceType} \mid \mathit{ArrayType} \\
\pi & \in & \mathit{PrimitiveType} & ::= \texttt{boolean} \mid \texttt{byte} \mid \texttt{short} \mid \texttt{int} \mid \texttt{long} \mid \\
 & & & \quad\ \texttt{char} \mid \texttt{float} \mid \texttt{double} \\
\mu & \in & \mathit{ClassOrInterfaceType} & ::= \mathit{ClassType} \mid \mathit{InterfaceType} \\
\alpha & \in & \mathit{ArrayType} & ::= \mathit{SimpleType}\ \texttt{[]} \\
\sigma* & \in & \mathit{SimpleType} & ::= \mathit{PrimitiveType} \mid \mathit{ClassOrInterfaceType} \\
\varsigma & \in & \mathit{ClassType} & ::= \mathit{Identifier} \\
\iota & \in & \mathit{InterfaceType} & ::= \mathit{Identifier}
\end{array}
$$

Figure 6: Syntactic types of Java.

$$
\begin{array}{llll}
\tau_a & \in & \mathit{ArgumentType} & ::= \mathit{ParameterType} \mid \mathsf{Null} \\
\tau_e & \in & \mathit{ExpressionType} & ::= \mathit{ResultType} \mid \mathsf{Null} \\
\tau_p & \in & \mathit{ParameterType} & ::= \mathit{Type} \mid \mathsf{Unit} \\
\sigma & \in & \mathit{NullOrSimpleType} & ::= \mathit{SimpleType} \mid \mathsf{Null}
\end{array}
$$

Figure 7: Semantic types of Java.

6 STATIC SEMANTICS

Below, we propose a static semantics for a very large subset of Java, since there is no such study so far. We dwelt on all the subtleties that we encountered in the Java language. Even the details were not neglected. Note that all reported errors [10, 13] along with the ones that we discovered have been considered in our type system.

Type Algebra

Types in Java are primitives or references. Reference types are class types, interface types and array types. The domain of syntactic types of Java is defined in Figure 6. We extend the domain of types to insert Null and Unit types. The former is the type of the `null` literal and the latter is the type of nullary methods and constructors. The extended domain of types is defined in Figure 7.

Notations

Along this paper, given two sets A and B, we will write $A \xrightarrow[m]{} B$ to denote the set of all mappings from A to B (partial functions from A to B with finite domain). The mapping (map for short) $m \in A \xrightarrow[m]{} B$ could be defined by extension as $[a_1 \mapsto b_1, \ldots, a_n \mapsto b_n]$ to denote the association of the elements b_i's to a_i's. We will write $\mathsf{dom}(m)$ to denote the domain of the map m and $\mathsf{rang}(m)$ to denote its co-domain. An empty map will be written as $[]$. Given two maps m_1 and m_2 from A to B, we will write $m_1 \dagger m_2$ the overriding of the map m_2 by the associations of the map m_1, as following: $(m_1 \dagger m_2)(a) = m_1(a)$ if $a \in \mathsf{dom}(m_1)$ and $m_2(a)$ otherwise. In the same way, we will write $m_1 \backslash m_2$ the restriction of the map m_1 with the associations of the map m_2 as following: $(m_1 \backslash m_2)(a) = m_1(a)$ if $a \in \mathsf{dom}(m_1)$ and $a \notin \mathsf{dom}(m_2)$. The expression m_1/m_2 indicates the map m_1 restricted to the associations of the map m_2, as following: $(m_1/m_2)(a) = m_1(a)$ if $a \in m_1$ and $a \in m_2$. We will write (Type)-set to denote the set of elements type Type, and (Type)-$\mathsf{multiset}$ to denote the multiset of elements type of Type. Multisets have the same forms as sets, except that we use delimiters "$\{\!|$" and "$|\!\}$" instead of "$\{$" and "$\}$". Finally, we will use two projections fst and snd for couples.

Type Environments

In this section, we will define type environments that are needed to deal with some complex aspects of the Java language. An environment, defined in Figure 8, is a record that contains maps of class and interface declarations of the current program. The latter include class and interface modifiers, type definitions of their components and names of their direct super-class, if any, and of their direct super-interfaces, if any. For the sake of modularity, we choose to represent the super-class of a class as a set of class types rather than as single class type. By doing so, we can deal with the non-existent super-class of the class `Object` by assigning it the $\emptyset$ value. Class components consist of field, method and constructor declarations, while interface components consist of field and method declarations. A field declaration is a map that associates a type and a set of modifiers to the field name. A method declaration is a map that associates a set of modifiers, a result type and a set of declared exceptions to the method's signature, i.e. the name and the type of its parameter. A constructor declaration is a map that associates a set of modifiers and a set of declared exceptions to the constructor's signature. As in [7], method and constructor bodies are not included in the environment, and they will be introduced later. The reason for this is simple: When we type check a Java program, we create and update a map of local variables declarations and a set of exceptions thrown, and we also keep information about `return` statements and abrupt termination of statements. Thus, by not including method and constructor bodies, we can consider the scope of local variables and their initialization, and type check `return`, `throw` and unreachable statements. We assume that the sets of modifiers in the environment include the default modifiers of classes, interfaces, fields and methods. As an example, if a method m is declared in an interface body, then its modifiers set in the environment will contain `abstract` and `public` even if they are not explicitly declared. Finally, we suppose all that classes, interfaces, methods and fields are named uniquely.

Type Abbreviations

To increase the readability of our formalization, we define in Figure 9 some type abbreviations that will be used in both typing rules and semantics functions.

Well Formation of Types

Well formedness of types is expressed as predicates defined in Figure 10. The predicate validIfaces yields true only if all interfaces in a given set are valid.

Type Relations

We distinguish between five kinds of type relations in Java: subclass relation $\sqsubset_{class}$ and $\sqsubseteq_{class}$, subinterface relation $\sqsubset_{interface}$ and $\sqsubseteq_{interface}$, implementation relation $\sqsubset_{implements}$, widening conversion relation $\sqsubseteq_{widen}$, cast conversion relation $\sqsubseteq_{cast}$.

The judgment $\Gamma \vdash \varsigma_1 \sqsubset_{class} \varsigma_2$ means that ς_1 is a subclass of the class named ς_2 in Γ, while judgment $\Gamma \vdash \varsigma_1 \sqsubseteq_{class} \varsigma_2$ means that under the environment Γ, ς_1 is either a subclass of ς_2 or ς_2 itself. Subclass relation is given by the inference rules in Figure 11. Similarly, the subinterface judgment $\Gamma \vdash \iota_1 \sqsubset_{interface} \iota_2$ means that ι_1 is a subinterface of the interface ι_2 in Γ, while judgment $\Gamma \vdash \iota_1 \sqsubseteq_{interface} \iota_2$ means that under the environment Γ, ι_1 is either a subinterface of ι_2 or ι_2 itself. Subinterface relation is given by the inference rules in Figure 12. The judgment $\Gamma \vdash \varsigma \sqsubset_{implements} \iota$ means that the class ς provides or inherits an implementation for the interface ι in Γ. Implementation relation is given by the inference rules in Figure 13.

Widening conversions specify implicit conversions that do not require any special run-time action and never result in a run-time exception. We distinguish two types of widening conversions: widening primitive and widening reference conversions. The judgment $\Gamma \vdash \pi_1 \sqsubseteq_{widenP} \pi_2$ means that the primitive type π_1 can replace the primitive type π_2 in Γ, without any explicit conversion (cast), while the judgment $\Gamma \vdash \rho_1 \sqsubseteq_{widenR} \rho_2$ means that the reference type ρ_1 can replace the reference type ρ_2 in Γ, without any explicit conversion. Widening primitive conversions and widening reference conversions include identity conversions and are given in Figures 14 and 15 respectively. We also define, in Figure 16, a more general widening conversions, used as $\Gamma \vdash \tau_1 \sqsubseteq_{widen} \tau_2$, that include identity conversions and widening primitive conversions. Widening conversions will be used especially to type check assignment expressions and method invocation expressions.

$$
\begin{array}{lcl}
\mathit{Environment} & = & \langle \; \mathit{classMap}\text{:} \quad \mathit{ClassMap}, \\
 & & \quad \mathit{interfaceMap}\text{:} \; \mathit{InterfaceMap} \; \rangle \\[1ex]
\mathit{ClassMap} & = & \mathit{ClassType} \xrightarrow[m]{} \mathit{ClassDecl} \\[1ex]
\mathit{InterfaceMap} & = & \mathit{InterfaceType} \xrightarrow[m]{} \mathit{InterfaceDecl} \\[1ex]
\mathit{ClassDecl} & = & \langle \; \begin{array}{ll}
\mathit{modifiers}\text{:} & (\mathit{ModifierName})\text{-set}, \\
\mathit{super}\text{:} & (\mathit{ClassType})\text{-set}, \\
\mathit{interfaces}\text{:} & (\mathit{InterfaceType})\text{-set}, \\
\mathit{fields}\text{:} & \mathit{Identifier} \xrightarrow[m]{} \mathit{FieldInfo}, \\
\mathit{methods}\text{:} & \mathit{Sig} \xrightarrow[m]{} \mathit{MethodInfo}, \\
\mathit{constructors}\text{:} & \mathit{Sig} \xrightarrow[m]{} \mathit{ConstructorInfo} \; \rangle
\end{array} \\[1ex]
\mathit{InterfaceDecl} & = & \langle \; \begin{array}{ll}
\mathit{modifiers}\text{:} & (\mathit{ModifierName})\text{-set}, \\
\mathit{interfaces}\text{:} & (\mathit{InterfaceType})\text{-set}, \\
\mathit{fields}\text{:} & \mathit{Identifier} \xrightarrow[m]{} \mathit{FieldInfo}, \\
\mathit{methods}\text{:} & \mathit{Sig} \xrightarrow[m]{} \mathit{MethodInfo} \; \rangle
\end{array} \\[1ex]
\mathit{Sig} & = & \mathit{Identifier} \times \mathit{ParameterType} \\[1ex]
\mathit{ModifierName} & = & \{\texttt{public},\texttt{static},\texttt{abstract},\texttt{final},\texttt{private}, \\
 & & \;\texttt{synchronized},\texttt{native},\texttt{transient},\texttt{volatile}\} \\[1ex]
\mathit{FieldInfo} & = & \langle \; \begin{array}{ll}
\mathit{fieldType}\text{:} & \mathit{Type}, \\
\mathit{modifiers}\text{:} & (\mathit{ModifierName})\text{-set} \; \rangle
\end{array} \\[1ex]
\mathit{MethodInfo} & = & \langle \; \begin{array}{ll}
\mathit{resultType}\text{:} & \mathit{ResultType}, \\
\mathit{modifiers}\text{:} & (\mathit{ModifierName})\text{set}, \\
\mathit{throws}\text{:} & (\mathit{ClassType})\text{-set} \; \rangle
\end{array} \\[1ex]
\mathit{ConstructorInfo} & = & \langle \; \begin{array}{ll}
\mathit{modifiers}\text{:} & (\mathit{ModifierName})\text{-set}, \\
\mathit{throws}\text{:} & (\mathit{ClassType})\text{-set} \; \rangle
\end{array}
\end{array}
$$

Figure 8: Type checking environments.

$$
\begin{array}{lcl}
\mathit{MethodMap} & = & \mathit{Sig} \xrightarrow[m]{} \mathit{MethodInfo} + \\
 & & \mathit{Sig} \xrightarrow[m]{} (\mathit{MethodInfo} \times \mathit{ReferenceType}) - \textsf{set} \\
\mathit{ConstructorMap} & = & \mathit{Sig} \xrightarrow[m]{} \mathit{ConstructorInfo} + \\
 & & \mathit{Sig} \xrightarrow[m]{} (\mathit{ConstructorInfo} \times \mathit{ReferenceType}) \\
\mathit{FieldMap} & = & \mathit{Identifier} \xrightarrow[m]{} \mathit{FieldInfo} + \\
 & & \mathit{Identifier} \xrightarrow[m]{} (\mathit{FieldInfo} \times \mathit{ReferenceType}) - \textsf{set}
\end{array}
$$

Γ	$\in$	$\mathit{Environment}$	static environment
cs or tcs	$\in$	$(\mathit{ClassType})$-set	classes set
is	$\in$	$(\mathit{InterfaceType})$-set	interfaces set
ms	$\in$	$(\mathit{ModifierName})$-set	modifiers set
n	$\in$	$\mathit{ClassOrIfaceType}$	class or interface name
mn	$\in$	$\mathit{Identifier}$	method name
pn	$\in$	$\mathit{Identifier}$	parameter name
sig	$\in$	Sig	method signature
fn	$\in$	$\mathit{Identifier}$	field name
mm	$\in$	$\mathit{MethodMap}$	map of methods
cm	$\in$	$\mathit{ConstructorMap}$	map of constructors
fm	$\in$	$\mathit{FieldMap}$	map of fields
clm	$\in$	$\mathit{ClassMap}$	map of classes
ifm	$\in$	$\mathit{InterfaceMap}$	map of interfaces

Figure 9: Type abbreviations.

$$
\begin{aligned}
&\mathsf{validType} \;:\; \mathit{Environment} \times \mathit{Type} \rightarrow \mathsf{bool}\\[1ex]
&\mathsf{validType}(\Gamma, \pi) \;=\; \mathsf{true}\\[1ex]
&\mathsf{validType}(\Gamma, \varsigma) \;=\; \mathsf{validClass}(\Gamma, \varsigma)\\[1ex]
&\mathsf{validType}(\Gamma, \iota) \;=\; \mathsf{validIface}(\Gamma, \iota)\\[1ex]
&\mathsf{validType}(\Gamma, \sigma * []) \;=\; \mathsf{validType}(\Gamma, \sigma *)\\[1ex]
&\mathsf{validIface}(\Gamma, \tau) \;=\; \Gamma.\mathit{interfaceMap}(\tau) \;\neq\; \bot\\[1ex]
&\mathsf{validClass}(\Gamma, \tau) \;=\; \Gamma.\mathit{classMap}(\tau) \;\neq\; \bot\\[1ex]
&\mathsf{validIfaces}(\Gamma, \emptyset) \;=\; \mathsf{true}\\
&\mathsf{validIfaces}(\Gamma, \{\iota\} \cup \mathit{is}) \;=\; \mathsf{validIface}(\Gamma, \iota) \;\wedge\; \mathsf{validIfaces}(\Gamma, \mathit{is})
\end{aligned}
$$

Figure 10: Well-formed types.

$$\frac{\mathsf{validType}(\Gamma, \varsigma_1) \quad \Gamma.\mathit{classMap}(\varsigma_1).\mathit{super} = \varsigma_2}{\Gamma \vdash \varsigma_1 \sqsubset_{\mathit{class}} \varsigma_2}$$

$$\frac{\Gamma \vdash \varsigma_1 \sqsubset_{\mathit{class}} \varsigma_2 \quad \Gamma \vdash \varsigma_2 \sqsubset_{\mathit{class}} \varsigma_3}{\Gamma \vdash \varsigma_1 \sqsubset_{\mathit{class}} \varsigma_3}$$

$$\frac{\Gamma \vdash \varsigma_1 \sqsubset_{\mathit{class}} \varsigma_2}{\Gamma \vdash \varsigma_1 \sqsubseteq_{\mathit{class}} \varsigma_2} \qquad \frac{\mathsf{validType}(\Gamma, \varsigma)}{\Gamma \vdash \varsigma \sqsubseteq_{\mathit{class}} \varsigma}$$

Figure 11: Subclass relation.

$$\frac{\mathsf{validType}(\mathit{\Gamma}, \iota_1) \quad \iota_2 \in \mathit{\Gamma}.\mathit{interfaceMap}(\iota_1).\mathit{interfaces}}{\mathit{\Gamma} \vdash \iota_1 \sqsubset_{\mathit{interface}} \iota_2}$$

$$\frac{\mathit{\Gamma} \vdash \iota_1 \sqsubset_{\mathit{interface}} \iota_2 \quad \mathit{\Gamma} \vdash \iota_2 \sqsubset_{\mathit{interface}} \iota_3}{\mathit{\Gamma} \vdash \iota_1 \sqsubset_{\mathit{interface}} \iota_3}$$

$$\frac{\mathit{\Gamma} \vdash \iota_1 \sqsubset_{\mathit{interface}} \iota_2}{\mathit{\Gamma} \vdash \iota_1 \sqsubseteq_{\mathit{interface}} \iota_2} \qquad \frac{\mathsf{validType}(\Gamma, \iota)}{\mathit{\Gamma} \vdash \iota \sqsubseteq_{\mathit{interface}} \iota}$$

Figure 12: Subinterface relation.

$$\frac{\mathsf{validType}(\mathit{\Gamma}, \varsigma) \quad \iota \in \mathit{\Gamma}.\mathit{classMap}(\varsigma).\mathit{interfaces}}{\mathit{\Gamma} \vdash \varsigma \sqsubset_{\mathit{implements}} \iota}$$

$$\frac{\mathit{\Gamma} \vdash \varsigma \sqsubset_{\mathit{implements}} \iota_1 \quad \mathit{\Gamma} \vdash \iota_1 \sqsubset_{\mathit{interface}} \iota_2}{\mathit{\Gamma} \vdash \varsigma \sqsubset_{\mathit{implements}} \iota_2}$$

$$\frac{\mathit{\Gamma} \vdash \varsigma_1 \sqsubset_{\mathit{class}} \varsigma_2 \quad \mathit{\Gamma} \vdash \varsigma_2 \sqsubset_{\mathit{implements}} \iota}{\mathit{\Gamma} \vdash \varsigma_1 \sqsubset_{\mathit{implements}} \iota}$$

Figure 13: Implementation Relation.

$$\frac{\mathsf{validType}(\mathit{\Gamma}, \pi)}{\mathit{\Gamma} \vdash \pi \sqsubseteq_{\mathit{widenP}} \pi}$$

$$\frac{\Box}{\mathit{\Gamma} \vdash \texttt{byte} \sqsubseteq_{\mathit{widenP}} \texttt{short}} \qquad \frac{\Box}{\mathit{\Gamma} \vdash \texttt{byte} \sqsubseteq_{\mathit{widenP}} \texttt{int}} \qquad \frac{\Box}{\mathit{\Gamma} \vdash \texttt{byte} \sqsubseteq_{\mathit{widenP}} \texttt{long}}$$

$$\frac{\Box}{\mathit{\Gamma} \vdash \texttt{byte} \sqsubseteq_{\mathit{widenP}} \texttt{float}} \qquad \frac{\Box}{\mathit{\Gamma} \vdash \texttt{byte} \sqsubseteq_{\mathit{widenP}} \texttt{double}} \qquad \frac{\Box}{\mathit{\Gamma} \vdash \texttt{short} \sqsubseteq_{\mathit{widenP}} \texttt{int}}$$

$$\frac{\Box}{\mathit{\Gamma} \vdash \texttt{short} \sqsubseteq_{\mathit{widenP}} \texttt{long}} \qquad \frac{\Box}{\mathit{\Gamma} \vdash \texttt{short} \sqsubseteq_{\mathit{widenP}} \texttt{float}} \qquad \frac{\Box}{\mathit{\Gamma} \vdash \texttt{short} \sqsubseteq_{\mathit{widenP}} \texttt{double}}$$

$$\frac{\Box}{\mathit{\Gamma} \vdash \texttt{int} \sqsubseteq_{\mathit{widenP}} \texttt{long}} \qquad \frac{\Box}{\mathit{\Gamma} \vdash \texttt{int} \sqsubseteq_{\mathit{widenP}} \texttt{float}} \qquad \frac{\Box}{\mathit{\Gamma} \vdash \texttt{int} \sqsubseteq_{\mathit{widenP}} \texttt{double}}$$

$$\frac{\Box}{\mathit{\Gamma} \vdash \texttt{long} \sqsubseteq_{\mathit{widenP}} \texttt{float}} \qquad \frac{\Box}{\mathit{\Gamma} \vdash \texttt{long} \sqsubseteq_{\mathit{widenP}} \texttt{double}} \qquad \frac{\Box}{\mathit{\Gamma} \vdash \texttt{float} \sqsubseteq_{\mathit{widenP}} \texttt{double}}$$

Figure 14: Widen primitive conversion.

$$\frac{\mathsf{validType}(\Gamma, \rho)}{\Gamma \vdash \rho \sqsubseteq_{widenR} \rho} \qquad \frac{\Gamma \vdash \varsigma_1 \sqsubset_{class} \varsigma_2}{\Gamma \vdash \varsigma_1 \sqsubseteq_{widenR} \varsigma_2} \qquad \frac{\Gamma \vdash \varsigma \sqsubset_{implements} \iota}{\Gamma \vdash \varsigma \sqsubseteq_{widenR} \iota}$$

$$\frac{\mathsf{validType}(\Gamma, \rho)}{\Gamma \vdash \mathsf{Null} \sqsubseteq_{widenR} \rho} \qquad \frac{\Gamma \vdash \iota_1 \sqsubset_{interface} \iota_2}{\Gamma \vdash \iota_1 \sqsubseteq_{widenR} \iota_2}$$

$$\frac{\mathsf{validType}(\Gamma, \iota) \quad \mathsf{validClass}(\Gamma, \texttt{Object})}{\Gamma \vdash \iota \sqsubseteq_{widenR} \texttt{Object}} \qquad \frac{\mathsf{validType}(\Gamma, \alpha) \quad \mathsf{validClass}(\Gamma, \texttt{Object})}{\Gamma \vdash \alpha \sqsubseteq_{widenR} \texttt{Object}}$$

$$\frac{\mathsf{validType}(\Gamma, \alpha) \quad \mathsf{validClass}(\Gamma, \texttt{Cloneable})}{\Gamma \vdash \alpha \sqsubseteq_{widenR} \texttt{Cloneable}}$$

$$\frac{\mathsf{validType}(\Gamma, \mu_1) \quad \mathsf{validType}(\Gamma, \mu_2)}{\Gamma \vdash \mu_1\texttt{[]} \sqsubseteq_{widenR} \mu_2\texttt{[]}}$$

Figure 15: Widen reference conversion.

$$\frac{\Gamma \vdash \pi_1 \sqsubseteq_{widenP} \pi_2}{\Gamma \vdash \pi_1 \sqsubseteq_{impl} \pi_2} \qquad \frac{\Gamma \vdash \rho_1 \sqsubseteq_{widenR} \rho_2}{\Gamma \vdash \rho_1 \sqsubseteq_{widen} \rho_2}$$

Figure 16: Widen conversion.

Finally, we define the cast relation, where the judgment $\varGamma \vdash \tau_1 \sqsubseteq_{cast} \tau_2$ means that under the environment $\varGamma$, a value of type τ_1 can be cast to a value of type τ_2. The cast relation is defined by inference rules in Figure 17. A cast between two reference types is type-correct if the two types may hold a reference to objects of the same class at run-time. Thus, a cast between a class and an interface types is legal if the class or one of its subclasses implements the interface. Since a `final` class cannot have any subclasses, a cast between a such class and interface type is type-incorrect, except for the class that implements the interface. In the same way, the cast between two `abstract` reference types (an interface type or an `abstract` class type) is type-correct only if all methods with the same signature that they declare have the same result type and compatible `throws` clauses. These constraints are expressed formally as the predicate castOk defined below; the predicate checkedException checks that a given exception is a checked exception:

$$\begin{array}{lll}
\mathsf{castOk} & : & \mathit{Environment} \times \mathit{InterfaceMap} \times \mathit{InterfaceMap} \rightarrow \mathsf{bool} \\
\mathsf{notConflictThrows_1} & : & \mathit{Environment} \times (\mathit{ClassType}) - \mathsf{set} \times (\mathit{ClassType}) - \mathsf{set} \rightarrow \mathsf{bool} \\
\mathsf{checkedException} & : & \mathit{Environment} \times \mathit{ClassType} \rightarrow \mathsf{bool}
\end{array}$$

$$\begin{array}{l}
\mathsf{castOk}(\Gamma, [], \mathit{ifm}) \quad = \quad \mathsf{true} \\
\mathsf{castOk}(\Gamma, [a_1 \mapsto b_1], [a_2 \mapsto b_2] \dagger \mathit{ifm}) \quad = \\
\quad \mathsf{if} \quad a_1 = a_2 \wedge \texttt{abstract} \in b_1.\mathit{modifiers} \\
\quad \mathsf{then} \quad b_1.\mathit{resultType} = b_2.\mathit{resultType} \wedge \mathsf{notConflictThrows_1}(\Gamma, b_1.\mathit{throws}, b_2.\mathit{throws}) \\
\quad \mathsf{else} \quad \mathsf{castOk}(\varGamma, [a_1 \mapsto b_1], \mathit{ifm}) \\
\quad \mathsf{endif} \\
\mathsf{castOk}(\varGamma, [a_1 \mapsto b_1] \dagger \mathit{ifm}_1, [a_2 \mapsto b_2] \dagger \mathit{ifm}_2) \quad = \\
\quad \mathsf{castOk}(\varGamma, [a_1 \mapsto b_1], [a_2 \mapsto b_2] \dagger \mathit{ifm}_2) \wedge \\
\quad \mathsf{castOk}(\varGamma, \mathit{ifm}_1, [a_2 \mapsto b_2] \dagger \mathit{ifm}_2)
\end{array}$$

$$\begin{array}{lll}
\mathsf{notConflictThrows_1}(\varGamma, \emptyset, \mathit{tcs}) & = & \mathsf{true} \\
\mathsf{notConflictThrows_1}(\varGamma, \mathit{tcs}_1, \mathit{tcs}_2) & = & \forall \varsigma \in \mathit{tcs}_2.\ \begin{array}[t]{ll} \mathsf{if} & \mathsf{checkedException}(\varGamma, \varsigma) \\ \mathsf{then} & \mathsf{subClass}(\varGamma, \varsigma, \mathit{tcs}_2) \\ \mathsf{endif} & \end{array}
\end{array}$$

$$\begin{array}{lll}
\mathsf{checkedException}(\varGamma, \varsigma) & = & \varGamma \vdash \varsigma \sqsubseteq_{class} \texttt{Throwable} \wedge \neg(\ \varGamma \vdash \varsigma \sqsubseteq_{class} \texttt{Error}\ ) \wedge \\
 & & \neg(\ \varGamma \vdash \varsigma \sqsubseteq_{class} \texttt{RuntimeException}\ )
\end{array}$$

For the sake of the formalization, we introduce two predicates: a predicate that checks that a class is a subclass of at least another class in a given classes set, and a predicate that checks that each class in a given classes set has a superclass in another classes set. These predicates are defined formally as following:

$$\begin{array}{lll}
\mathsf{subClass} & : & \mathit{Environment} \times \mathit{ClassType} \times (\mathit{ClassType}) - \mathsf{set} \rightarrow \mathsf{bool} \\
\mathsf{subClasses} & : & \mathit{Environement} \times (\mathit{ClassType}) - \mathsf{set} \times (\mathit{ClassType}) - \mathsf{set} \rightarrow \mathsf{bool}
\end{array}$$

$$\begin{array}{lll}
\mathsf{subClass}(\varGamma, \varsigma, \emptyset) & = & \mathsf{false} \\
\mathsf{subClass}(\varGamma, \varsigma_1, \{\varsigma_2\} \cup \mathit{cs}) & = & \varGamma \vdash \varsigma_1 \sqsubseteq_{class} \varsigma_2 \vee \mathsf{subClass}(\varGamma, \varsigma, \mathit{cs})
\end{array}$$

$$\frac{\mathit{\Gamma} \vdash \tau_e \sqsubseteq_{impl} \tau}{\mathit{\Gamma} \vdash \tau_e \sqsubseteq_{cast} \tau} \qquad \frac{\square}{\mathit{\Gamma} \vdash \texttt{byte} \sqsubseteq_{cast} \texttt{char}} \qquad \frac{\square}{\mathit{\Gamma} \vdash \texttt{short} \sqsubseteq_{cast} \texttt{byte}}$$

$$\frac{\square}{\mathit{\Gamma} \vdash \texttt{short} \sqsubseteq_{cast} \texttt{char}} \qquad \frac{\square}{\mathit{\Gamma} \vdash \texttt{char} \sqsubseteq_{cast} \texttt{byte}} \qquad \frac{\square}{\mathit{\Gamma} \vdash \texttt{char} \sqsubseteq_{cast} \texttt{short}}$$

$$\frac{\square}{\mathit{\Gamma} \vdash \texttt{int} \sqsubseteq_{cast} \texttt{byte}} \qquad \frac{\square}{\mathit{\Gamma} \vdash \texttt{int} \sqsubseteq_{cast} \texttt{short}} \qquad \frac{\square}{\mathit{\Gamma} \vdash \texttt{int} \sqsubseteq_{cast} \texttt{char}}$$

$$\frac{\square}{\mathit{\Gamma} \vdash \texttt{long} \sqsubseteq_{cast} \texttt{byte}} \qquad \frac{\square}{\mathit{\Gamma} \vdash \texttt{long} \sqsubseteq_{cast} \texttt{short}} \qquad \frac{\square}{\mathit{\Gamma} \vdash \texttt{long} \sqsubseteq_{cast} \texttt{char}}$$

$$\frac{\square}{\mathit{\Gamma} \vdash \texttt{long} \sqsubseteq_{cast} \texttt{int}} \qquad \frac{\square}{\mathit{\Gamma} \vdash \texttt{float} \sqsubseteq_{cast} \texttt{byte}} \qquad \frac{\square}{\mathit{\Gamma} \vdash \texttt{float} \sqsubseteq_{cast} \texttt{short}}$$

$$\frac{\square}{\mathit{\Gamma} \vdash \texttt{float} \sqsubseteq_{cast} \texttt{char}} \qquad \frac{\square}{\mathit{\Gamma} \vdash \texttt{float} \sqsubseteq_{cast} \texttt{int}} \qquad \frac{\square}{\mathit{\Gamma} \vdash \texttt{float} \sqsubseteq_{cast} \texttt{long}}$$

$$\frac{\square}{\mathit{\Gamma} \vdash \texttt{double} \sqsubseteq_{cast} \texttt{byte}} \qquad \frac{\square}{\mathit{\Gamma} \vdash \texttt{double} \sqsubseteq_{cast} \texttt{short}} \qquad \frac{\square}{\mathit{\Gamma} \vdash \texttt{double} \sqsubseteq_{cast} \texttt{char}}$$

$$\frac{\square}{\mathit{\Gamma} \vdash \texttt{double} \sqsubseteq_{cast} \texttt{int}} \qquad \frac{\square}{\mathit{\Gamma} \vdash \texttt{double} \sqsubseteq_{cast} \texttt{long}} \qquad \frac{\square}{\mathit{\Gamma} \vdash \texttt{double} \sqsubseteq_{cast} \texttt{float}}$$

$$\frac{\mathit{\Gamma} \vdash \varsigma_1 \sqsubseteq_{class} \varsigma_2}{\mathit{\Gamma} \vdash \varsigma_2 \sqsubseteq_{cast} \varsigma_1} \qquad \frac{\mathsf{validClass}(\mathit{\Gamma}, \texttt{Object}) \quad \mathsf{validType}(\mathit{\Gamma}, \alpha)}{\mathit{\Gamma} \vdash \texttt{Object} \sqsubseteq_{cast} \alpha}$$

$$\frac{\begin{array}{c}\mathsf{validClass}(\mathit{\Gamma}, \varsigma) \quad \mathsf{validIface}(\mathit{\Gamma}, \iota) \\ \texttt{final} \notin \mathit{\Gamma}.\mathit{classMap}(\varsigma).\mathit{modifiers} \quad \neg(\mathit{\Gamma} \vdash \varsigma \sqsubseteq_{implements} \iota) \\ \texttt{abstract} \in \mathit{\Gamma}.\mathit{classMap}(\varsigma) \Rightarrow \mathsf{castOk}(\mathit{\Gamma}, \mathit{\Gamma}.\mathit{classMap}(\varsigma), \mathit{\Gamma}.\mathit{interfaceMap}(\iota))\end{array}}{\mathit{\Gamma} \vdash \varsigma \sqsubseteq_{cast} \iota}$$

$$\frac{\begin{array}{c}\mathsf{validIface}(\mathit{\Gamma}, \iota) \quad \mathsf{validClass}(\mathit{\Gamma}, \varsigma) \\ \texttt{final} \notin \mathit{\Gamma}.\mathit{classMap}(\varsigma).\mathit{modifiers} \quad \neg(\mathit{\Gamma} \vdash \varsigma \sqsubseteq_{implements} \iota) \\ \texttt{abstract} \in \mathit{\Gamma}.\mathit{classMap}(\varsigma) \Rightarrow \mathsf{castOk}(\mathit{\Gamma}, \mathit{\Gamma}.\mathit{classMap}(\varsigma), \mathit{\Gamma}.\mathit{interfaceMap}(\iota))\end{array}}{\mathit{\Gamma} \vdash \iota \sqsubseteq_{cast} \varsigma}$$

$$\frac{\begin{array}{c}\mathsf{validIface}(\mathit{\Gamma}, \iota_1) \quad \mathsf{validIface}(\mathit{\Gamma}, \iota_2) \quad \neg(\mathit{\Gamma} \vdash \iota_1 \sqsubseteq_{interface} \iota_2) \\ \mathsf{castOk}(\mathit{\Gamma}, \mathit{\Gamma}.\mathit{interfaceMap}(\iota_1), \mathit{\Gamma}.\mathit{interfaceMap}(\iota_2))\end{array}}{\mathit{\Gamma} \vdash \iota_1 \sqsubseteq_{cast} \iota_2}$$

$$\frac{\mathit{\Gamma} \vdash \mu_1 \sqsubseteq_{cast} \mu_2}{\mathit{\Gamma} \vdash \mu_1 \texttt{[]} \sqsubseteq_{cast} \mu_2 \texttt{[]}}$$

Figure 17: Cast conversion.

$$\begin{aligned}
\mathsf{subClasses}(\Gamma, \emptyset, cs) &= \mathsf{true} \\
\mathsf{subClasses}(\Gamma, \{\varsigma\} \cup cs_1, cs_2) &= \mathsf{subClass}(\Gamma, \varsigma, cs_2) \wedge \mathsf{subClasses}(\Gamma, cs_1, cs_2)
\end{aligned}$$

Lookup for Fields, Methods and Constructors Accessible from Classes and Interfaces

In this section, we will define some functions that extract fields, methods and constructors accessible from a given class or interface. Members accessible from a class are those inherited from its direct superclass, those inherited from any direct superinterfaces and those declared in the body of the class. Notice that private members are not inherited just as constructors, since the latter are not regarded as members. Members of an interface are those inherited from its superinterfaces and those declared in the body of the interface. Moreover, interfaces inherit members from the class `Object`. Notice that hidden fields and overridden methods are never inherited. Finally, members accessible from an array are those of the class `Object`, except of the method named clone that is overridden. Furthermore, arrays have a public and final field named `length`.

Lookup for Methods

Classes and interfaces might inherit several abstract methods with the same signature. In order to represent this kind of inheritance, we introduce a new type of maps $Sig \xrightarrow[m]{} (MethodInfo \times ReferenceType) - \mathsf{set}$ mapping signature values into a set of values. The type of the set is a pair of method's information, i.e. a set of modifiers, a result type and a set of declared exceptions, and a class, an interface or an array type that contains the method declaration. For brevity of notation, we use the meta-variable B_1 for the co-domain of maps $(MethodInfo \times ReferenceType) - \mathsf{set}$. We define, below, a recursive function that transform maps of method declarations (introduced in section 6) into previously defined maps.

$$\mathsf{transform_m} \quad : \quad ReferenceType \times (Sig \xrightarrow[m]{} MethodInfo) \longrightarrow (Sig \xrightarrow[m]{} B_1)$$

$$\begin{aligned}
\mathsf{transform_m}(\rho, []) &= [] \\
\mathsf{transform_m}(\rho, [sig \mapsto \langle rt, ms, tcs \rangle] \dagger m) &= [sig \mapsto \{(\langle rt, ms, tcs \rangle, \rho)\}] \dagger \\
&\quad\ \mathsf{transform_m}(\rho, m)
\end{aligned}$$

The functions $\mathsf{methods_i}$, $\mathsf{methods_c}$ and $\mathsf{methods_a}$, defined below, collect methods accessible from a given interface, a class and an array, respectively, into maps:

$$\begin{aligned}
\mathsf{methods_i} &\quad : \quad Environment \times InterfaceType \longrightarrow (Sig \xrightarrow[m]{} B_1) \\
\mathsf{methods_c} &\quad : \quad Environment \times ClassType \longrightarrow (Sig \xrightarrow[m]{} B_1) \\
\mathsf{methods_a} &\quad : \quad Environment \times ArrayType \longrightarrow (Sig \xrightarrow[m]{} B_1)
\end{aligned}$$

$$
\begin{array}{lll}
\mathsf{methodsIfaceSet} & : & \mathit{Environment} \times (\mathit{InterfaceType}) - \mathsf{set} \rightarrow (\mathit{Sig} \overrightarrow{m} B_1) \\
\mathsf{privateMethods} & : & (\mathit{Sig} \overrightarrow{m} B_1) \rightarrow (\mathit{Sig} \overrightarrow{m} B_1) \\
\_ \ddagger_m \_ & : & (\mathit{Sig} \overrightarrow{m} B_1) \times (\mathit{Sig} \overrightarrow{m} B_1) \rightarrow (\mathit{Sig} \overrightarrow{m} B_1) \\
\mathsf{modifiers} & : & (\mathit{MethodInfo} \times \mathit{ClassOrIfaceType}) - \mathsf{set} \rightarrow (\mathit{ModifierName}) - \mathsf{set}
\end{array}
$$

$$
\begin{array}{lll}
\mathsf{methods_i}(\Gamma, \iota) & = & (\ \mathsf{transform_m}(\iota, \Gamma.\mathit{interfaceMap}(\iota).\mathit{methods})\ \dagger \\
 & & \quad \mathsf{transform_m}(\mathtt{Object}, \Gamma.\mathit{classMap}(\mathtt{Object}).\mathit{methods}))\ \dagger \\
 & & \mathsf{methodsIfaceSet}(\Gamma, \Gamma.\mathit{interfaceMap}(\iota).\mathit{interfaces})
\end{array}
$$

```
methods_c($\Gamma, \varsigma_1$) =
    if    $\varsigma_1$ = Object
    then  transform_m($\varsigma_1, \Gamma.\mathit{classMap}(\varsigma_1).\mathit{methods}$)
    else  let $\langle ms, \{\varsigma_2\}, is, fm, mm, cm\rangle = \Gamma.\mathit{classMap}(\varsigma_1)$
          in  transform_m($\varsigma_1, mm$)   $\dagger$
              ( ( methods_c($\Gamma, \varsigma_2$) $\setminus$ privateMethods(
                                   transform_m($\varsigma_1, \Gamma.\mathit{classMap}(\varsigma_2).\mathit{methods}$)) ) $\ddagger_m$
                methodsIfaceSet($\Gamma, is$) )
          end let
    end if
```

$$
\begin{array}{lll}
\mathsf{methods_a}(\Gamma, \alpha) & = & [(\mathtt{clone}, \mathsf{Unit}) \mapsto \{(\langle \mathtt{Object}, \{\mathtt{public}\}, \emptyset\rangle, \alpha)\}]\ \ \dagger \\
 & & \mathsf{transform_m}(\mathtt{Object}, \Gamma.\mathit{classMap}(\mathtt{Object}).\mathit{methods})
\end{array}
$$

$$
\begin{array}{lll}
\mathsf{methodsIfaceSet}(\Gamma, \emptyset) & = & [] \\
\mathsf{methodsIfaceSet}(\Gamma, \{\iota\} \cup is) & = & \mathsf{methods_i}(\Gamma, \iota)\ \ddagger_m\ \mathsf{methodsIfaceSet}(\Gamma, is)
\end{array}
$$

```
privateMethods([])  =  []
privateMethods([$sig \mapsto \{(\langle rt, ms, tcs, \rangle, \rho)\}$] $\dagger$ $mm$)  =
  ( if    private $\in ms$
    then  [$sig \mapsto \{(\langle rt, ms, tcs, \rangle, \rho)\}$]
    else  []
    endif ) $\dagger$
  privateMethods($mm$)
```

```
[] $\ddagger_m$ $mm$  =  $mm$
([$sig \mapsto b$] $\dagger$ $mm_1$) $\ddagger_m$ $mm_2$ = if    $mm_2(sig) = \emptyset$
                              then  [$sig \mapsto b$] $\dagger$ ($mm_1 \ddagger_m mm_2$)
                              else  if    abstract $\in$ modifiers($b$)
                                    then  [$sig \mapsto b \cup mm_2(sig)$] $\dagger$ ($mm_1 \ddagger_m mm_2$)
                                    else  [$sig \mapsto b$] $\dagger$ ($mm_1 \ddagger_m mm_2$)
                                    endif
                              endif
```

$$
\begin{array}{lll}
\mathsf{modifiers}(\emptyset) & = & \emptyset \\
\mathsf{modifiers}(\{(\langle rt, ms, tcs\rangle, \mu)\} \cup s) & = & ms
\end{array}
$$

The † operator, used in functions above, overrides the map of inherited methods, with the map of methods declared in the class or the interface. This turned out to be necessary to represent methods' overriding. The function methodsIfaceSet returns a map containing all methods inherited by an interface from all its superinterfaces. Since private methods are not inherited, we do not include them in the map of superclass's methods. The map of `private` methods is computed by the recursive function privateMethods. Since classes and interfaces might inherit more than one `abstract` method with the same signature, we introduce a new operator $\ddagger_m$ that computes the map of inherited methods. If methods having the same signature are abstract, then the computed map will contain all these methods. Otherwise, if at least one of them is not abstract, then the computed map will contain only that one (of all the methods with the same signature). That is, a non-`abstract` method overrides all `abstract` methods with the same signature.

Lookup for Fields

As in the case of methods, we introduce a new type of maps $Identifier \xrightarrow[m]{} (FieldInfo \times ReferenceType) - \mathsf{set}$ to represent the inheritance of several fields with the same name. For brevity of notation, we use the meta-variable B_2 for the co-domain of those maps $(FieldInfo \times ReferenceType) - \mathsf{set}$. In order to have the same type of maps, we introduce a new function, which applied to a value of type $Identifier \xrightarrow[m]{} FieldInfo$ will yield a value of type $Identifier \xrightarrow[m]{} (FieldInfo \times ReferenceType) - \mathsf{set}$:

$$\mathsf{transform_f} : ReferenceType \times (Identifier \xrightarrow[m]{} FieldInfo) \rightarrow (Identifier \xrightarrow[m]{} B_2)$$

$$\begin{aligned}
\mathsf{transform_f}(\rho, []) &= [] \\
\mathsf{transform_f}(\rho, [fn \mapsto \langle ft, ms\rangle] \dagger fm) &= [fn \mapsto \{(\langle ft, ms\rangle, \rho)\}] \dagger \\
&\quad\ \mathsf{transform_f}(\rho, fm)
\end{aligned}$$

Function $\mathsf{fields_i}$, $\mathsf{fields_c}$ and $\mathsf{fields_a}$, defined below, collect fields accessible from a given interface, class and array, respectively, into maps, and are very similar to those of methods' lookup:

$$\begin{aligned}
\mathsf{fields_i} &: Environment \times InterfaceType \rightarrow (Identifier \xrightarrow[m]{} B_2) \\
\mathsf{fields_c} &: Environment \times ClassType \rightarrow (Identifier \xrightarrow[m]{} B_2) \\
\mathsf{fields_a} &: Environment \times ArrayType \rightarrow (Identifier \xrightarrow[m]{} B_2) \\
\mathsf{fieldsIfaceSet} &: Environment \times (InterfaceType) - \mathsf{set} \rightarrow (Identifier \xrightarrow[m]{} B_2) \\
\mathsf{privateFields} &: (Identifier \xrightarrow[m]{} B_2) \rightarrow (Identifier \xrightarrow[m]{} B_2) \\
\_ \ddagger_f \_ &: (Identifier \xrightarrow[m]{} B_2) \times (Identifier \xrightarrow[m]{} B_2) \rightarrow (Identifier \xrightarrow[m]{} B_2)
\end{aligned}$$

$$\begin{aligned}
\mathsf{fields_i}(\Gamma, \iota) = &\ \mathsf{transform_f}(\iota, \Gamma.interfaceMap(\iota).fields) \dagger \\
&\ \mathsf{fieldsIfaceSet}(\Gamma, \Gamma.interfaceMap(\iota).interfaces)
\end{aligned}$$

```
fields_c(Γ, ς1) =
    if   $\varsigma_1 = \mathsf{Object}$
    then $\mathsf{transform_f}(\varsigma_1, \mathit{\Gamma}.\mathit{classMap}(\varsigma_1).\mathit{fields})$
    else let $\langle ms, \{\varsigma_2\}, is, fm, mm, cm\rangle = \mathit{\Gamma}.\mathit{classMap}(\varsigma_1)$ in
          $\mathsf{transform_f}(\varsigma_1, fm)\ \dagger$
          $(\ (\ \mathsf{fields_c}(\mathit{\Gamma}, \varsigma_2)\ \setminus\ \mathsf{privateFields}(\mathsf{transform_f}(\varsigma_1, \mathit{\Gamma}.\mathit{classMap}(\varsigma_2).\mathit{fields})\ )\ \ddagger_f$
            $\mathsf{fieldsIfaceSet}(\mathit{\Gamma}, is)\ )$
         endlet
    endif
```

$$\mathsf{fields_a}(\mathit{\Gamma}, \alpha) = [\texttt{length} \mapsto \{(\langle \texttt{int}, \{\texttt{public}, \texttt{final}\}\rangle, \alpha)\}]$$

$$\mathsf{fieldsIfaceSet}(\mathit{\Gamma}, \emptyset) = []$$
$$\mathsf{methodsIfaceSet}(\mathit{\Gamma}, \{\iota\} \cup is) = \mathsf{fields_i}(\mathit{\Gamma}, \iota)\ \ddagger_f\ \mathsf{fieldsIfaceSet}(\Gamma, is)$$

$$\mathsf{privateFields}([]) = []$$

```
$\mathsf{privateFields}([fn \mapsto \{(\langle ft, ms\rangle, \rho)\}] \dagger fm)$ =
   ( if private $\in ms$ then $[fn \mapsto \{(\langle ft, ms\rangle, \rho)\}]$
                        else  []
                        endif ) $\dagger$
   $\mathsf{privateFields}(fm)$
```

```
$[]\ \ddagger_f\ fm = fm$
$([fn \mapsto b] \dagger fm_1)\ \ddagger_f\ fm_2$ =  if   $fm_2(fn) = \emptyset$
                                  then $[fn \mapsto b]\ \dagger\ (fm_1\ \ddagger_f\ fm_2)$
                                  else $[fn \mapsto b \cup fm_2(fn)]\ \dagger\ (fm_1\ \ddagger_f\ fm_2)$
                                  endif
```

In some cases, when we type-check method and field invocation, we do not know if the caller is a class, an interface or an array type, and therefore we cannot decide which lookup function, defined above, must be applied. For instance, in order to type-check the field access *Primary*.*Identifier*, we have to find the map of fields accessible from the *Primary* type. To do so, we must call one of the following functions: $\mathsf{fields_c}$, $\mathsf{fields_i}$, $\mathsf{fields_a}$, according to whether the type of *Primary* is a class type, an interface type or an array type, respectively. Thus, we introduce new functions that invoke the adequate function according to the caller's type:

$$\mathsf{methodsVar} : \mathit{Environment} \times \mathit{ReferenceType} \rightarrow (\mathit{Sig} \underset{m}{\overset{}{\rightarrow}} B_1)$$
$$\mathsf{fieldsVar} : \mathit{Environment} \times \mathit{ReferenceType} \rightarrow (\mathit{Identifier} \underset{m}{\overset{}{\rightarrow}} B_2)$$

```
$\mathsf{methodsVar}(\mathit{\Gamma}, \rho)$ = case $\rho$ of $\varsigma \Rightarrow \mathsf{methods_c}(\mathit{\Gamma}, \varsigma)$
                               $\iota \Rightarrow \mathsf{methods_i}(\mathit{\Gamma}, \iota)$
                               $\alpha \Rightarrow \mathsf{methods_a}(\mathit{\Gamma}, \alpha)$
                      endcase

$\mathsf{fieldsVar}(\mathit{\Gamma}, \rho)$ = case $\rho$ of $\varsigma \Rightarrow \mathsf{fields_c}(\mathit{\Gamma}, \varsigma_1)$
```

```
            $\iota \Rightarrow \mathsf{fields_i}(\Gamma, \iota)$
            $\alpha \Rightarrow \mathsf{fields_a}(\Gamma, \alpha)$
      endcase
```

Lookup for Constructors

If a class declares constructors, then they are the only constructors accessible from this class, since constructors are not inherited. Otherwise, if the class contains no constructor declarations, then a default constructor is automatically provided to it. The default constructor of the class `Object` is the one that takes no parameters and has an empty body. For all other classes, the default constructor takes no parameters and invokes the constructor of the superclass with no arguments. Thus, a map of constructors accessible from a given class is yielded by the function constructors defined as follows:

$$\mathsf{constructors} \;:\; \mathit{Environment} \times \mathit{ClassType} \longrightarrow (\mathit{Sig} \underset{m}{\rightarrow} (\mathit{ConstructorInfo} \times \mathit{ClassType}))$$

$$\mathsf{defaultConstructorInvocation} \;:\; \mathit{Environment} \times \mathit{ClassType} \longrightarrow \mathsf{bool} \times (\mathit{ClassType}) - \mathsf{set} \times (\mathit{ClassType}) - \mathsf{set}$$

$$\mathsf{transform_c} \;:\; \mathit{ClassType} \times (\mathit{Sig} \underset{m}{\rightarrow} \mathit{ConstructorInfo}) \longrightarrow (\mathit{Sig} \underset{m}{\rightarrow} (\mathit{ConstructorInfo} \times \mathit{ClassType}))$$

```
constructors$(\Gamma, \varsigma_1)$ =
    let $\Gamma.\mathit{classMap}(\varsigma_1) = \langle ms, is, cs, fm, mm, cm\rangle$ in
      if    $cm = []$
      then  if    defaultConstructorInvocation$(\Gamma, \varsigma_1) = (\mathsf{true}, tcs, \{\varsigma_2\})$
            then  $[(\varsigma_1, \mathsf{Unit}) \mapsto (\langle\{public\}, tcs\rangle, \varsigma_2)]$
            else  $[]$
            endif
      else transform$_c(\varsigma_1, cm)$
      endif
    endlet
```

```
defaultConstructorInvocation$(\Gamma, \varsigma_1)$ =
  if    $\varsigma_1 \neq$ Object
  then  let $\{\varsigma_2\} = \Gamma.\mathit{classMap}(\varsigma_1).\mathit{super}$ in
          if    $\Gamma.\mathit{classMap}(\varsigma_2).\mathit{constructors} \neq []$
          then  case $\Gamma.\mathit{classMap}(\varsigma_2).\mathit{constructors}(\varsigma_2, \mathsf{Unit})$
                of  $\langle ms, tcs\rangle \Rightarrow$ (private $\notin ms, tcs, \{\varsigma_2\})$
                    $\perp \Rightarrow (\mathsf{false}, \emptyset, \emptyset)$
                endcase
          else  defaultConstructorInvocation$(\Gamma, \varsigma_2)$
          endif
        endlet
  else  $(\mathsf{true}, \emptyset, \emptyset)$
  endif
```

$$\begin{aligned}
\mathsf{transform_c}(\varsigma, []) &= [] \\
\mathsf{transform_c}(\varsigma, [sig \mapsto \langle ms, tcs\rangle] \dagger cm) &= [sig \mapsto (\langle ms, tcs\rangle, \varsigma)] \dagger \\
&\quad \mathsf{transform_c}(\varsigma, cm)
\end{aligned}$$

The function **defaultConstructorInvocation**, used above, finds the default constructor of a class. It yields a triplet: a boolean variable that takes the **true** value if the constructor is found and the **false** value otherwise, a set of exceptions declared in the constructor and a class type that declares the constructor. To each constructor entry (modifiers and declared exceptions) in the constructors' map, the function $\mathsf{transform_c}$ adds the declaring class.

Well-formed Environments

Environments must be well-formed, i.e. their composing declarations must satisfy some important properties of the Java language. These properties are expressed as predicates on maps.

Well-formed Class Fields

A field declared in a class is well-formed if its type is valid and its modifiers set is a subset of {`public`, `private`, `final`, `static`, `volatile`, `transient`}. Moreover, a `public` field cannot also be `private`, and a `final` field cannot be `volatile`. These constraints are expressed formally as the predicate defined bellow:

$$\mathsf{wellFormedField_c} \;:\; \mathit{Environment} \times (\texttt{Identifier} \underset{m}{\twoheadrightarrow} \mathit{FieldInfo}) \rightarrow \mathsf{bool}$$

$$\begin{aligned}
&\mathsf{wellFormedField_c}(\Gamma, []) = \mathsf{true} \\
&\mathsf{wellFormedField_c}(\Gamma, [fn \mapsto \langle ft, ms\rangle] \dagger fm) = \\
&\quad \mathsf{validType}(\Gamma, ft) \wedge \\
&\quad ms \subseteq \{\texttt{public}, \texttt{private}, \texttt{final}, \texttt{static}, \texttt{volatile}, \texttt{transient}\} \wedge \\
&\quad \{\texttt{public}, \texttt{private}\} \not\subseteq ms \wedge \\
&\quad \{\texttt{volatile}, \texttt{final}\} \not\subseteq ms \wedge \\
&\quad \mathsf{wellFormedField_c}(\Gamma, fm)
\end{aligned}$$

Well-formed Class Methods

A method declared in a class is well-formed if and only if the following conditions are met:

- Its parameter and result types are valid;
- Its modifiers set is a subset of {`public`, `private`, `final`, `static`, `abstract`, `native`, `synchronized`};

- It cannot be declared `public` and `private` at the same time;
- If it is `abstract`, then it cannot be `private`, `final`, `static`, `native` or `synchronized`;
- Its declared exceptions must be subclasses of the class `Throwable`.

The above-mentioned conditions are expressed more formally by the following recursive predicate:

$$\mathsf{wellFormedMethod_c} \quad : \quad \mathit{Environment} \times (\mathit{Sig} \underset{m}{\twoheadrightarrow} \mathit{MethodInfo}) \to \mathsf{bool}$$

$$\begin{array}{l}
\mathsf{wellFormedMethod_c}(\Gamma, []) \quad = \quad \mathsf{true} \\
\mathsf{wellFormedMethod_c}(\Gamma, [(mn, \tau_p) \mapsto \langle rt, ms, tcs\rangle] \dagger mm) \quad = \\
\quad \mathsf{validType}(\Gamma, \tau_p) \wedge \mathsf{validType}(\Gamma, rt) \wedge \\
\quad ms \subseteq \{\texttt{public}, \texttt{private}, \texttt{final}, \texttt{static}, \texttt{abstract}, \texttt{native}, \texttt{synchronized}\} \wedge \\
\quad \{\texttt{public}, \texttt{private}\} \not\subseteq ms \wedge \\
\quad (\ \mathsf{if} \quad \texttt{abstract} \in ms \\
\quad \ \ \mathsf{then} \ \ \{\texttt{private}, \texttt{final}, \texttt{static}, \texttt{native}, \texttt{synchronized}\} \cap ms = \emptyset \\
\quad \ \ \mathsf{endif}\ ) \wedge \\
\quad \mathsf{subClasses}(\Gamma, tcs, \{\texttt{Throwable}\}) \wedge \\
\quad \mathsf{wellFormedMethod_c}(\Gamma, mm)
\end{array}$$

Well-formed Constructors

A constructor declaration is well-formed if and only if the following constraints hold:

- It has the same name as its declaring class;
- Its modifiers set is a subset of {`public`,`private`};
- It cannot be declared `public` and `private` at the same time;
- Its declared exceptions must be subclasses of the class `Throwable`.

These conditions are checked by the predicate defined below:

$$\mathsf{wellFormedConstructor_c} \quad : \quad \mathit{Environment} \times \mathit{ClassType} \times (\mathit{Sig} \underset{m}{\twoheadrightarrow} \mathit{ConstructorInfo}) \to \mathsf{bool}$$

$$\begin{array}{l}
\mathsf{wellFormedConstructors_c}(\Gamma, \varsigma, []) \quad = \quad \mathsf{true} \\
\mathsf{wellFormedConstructors_c}(\Gamma, \varsigma_1, [(\varsigma_2, \tau_a) \mapsto \langle ms, tcs\rangle] \dagger cm) \quad = \\
\quad \varsigma_1 = \varsigma_2 \\
\quad \mathsf{validType}(\Gamma, \tau_a) \wedge \\
\quad ms \subseteq \{\texttt{public}, \texttt{private}\} \wedge \\
\quad \{\texttt{public}, \texttt{private}\} \not\subseteq ms \wedge \\
\quad \mathsf{subClasses}(\Gamma, tcs, \{\texttt{Throwable}\}) \wedge \\
\quad \mathsf{wellFormedConstructors_c}(\Gamma, \varsigma_1, cm)
\end{array}$$

Well-formed Classes

A class declaration is considered well-formed if and only if the following conditions are satisfied:

- It is named uniquely, that is there is no interface that has the same name.
- Its modifiers set is a subset of {`public`, `abstract`,`final`}.
- It cannot be declared `final` and `abstract` at the same time, since `abstract` classes are incomplete and cannot be `final`.
- Its implemented interfaces, if any, exist.
- If it is not the class `Object`, then it must have a valid superclass that is not `final`.
- If it declares `abstract` methods, then it must be declared as an `abstract` method.
- All declared fields, if any, are well-formed.
- All declared methods, if any, are well-formed.
- If it contains constructor declarations, then they must be well-formed, otherwise the default constructor must exist, must be well-formed and cannot be `private`. Moreover, the default constructor, if any, cannot declare any checked exception, since its call may throw an exception.
- Each method declared in this class and overriding or hiding a set of methods declared in the superclasses and superinterfaces, must meet the following conditions:
  - it cannot declare more checked exceptions than the overridden or hidden methods;
  - for each checked exception type declared in the clause `throws` of this method, the same exception type or one of its supertypes must appear in the `throws` clause of each overridden or hidden method;
  - it cannot be `private`, since it cannot be more private than methods that it overrides and hides;
  - it cannot override or hide `final` methods;
  - if it is a non-`static` method, than it cannot override a `static` one, and if it is `static`, than it cannot override a non-`static` one.

However, since overridden and hidden methods are not inherited, we have to ensure that these constraints are met in the following cases: When methods of the current class override or hide methods inherited from its superclass, when methods of the current class implement `abstract` methods inherited form its superinterfaces and when non-abstract methods inherited from superclass of the current class implement methods of its superinterfaces. To do so, we introduce a new recursive predicate wellOverriddenHidden and we call it three times with different arguments for each case. Moreover, when the current class inherits more than one method with the same signature, we have to check that all these methods can be implemented by a single method in the subclass of the class, i.e. that they have the same result type and the checked exceptions that they declare are compatible. The set of checked exceptions is yielded by the function named getCheckedExceptions.

wellFormedClass	:	$Environment \times ClassType \rightarrow \mathsf{bool}$
abstractMethods	:	$(Sig \underset{m}{\rightarrow} B_1) \rightarrow (Sig \underset{m}{\rightarrow} B_1)$
wellOverriddenHidden	:	$(Sig \underset{m}{\rightarrow} B_1) \times (Sig \underset{m}{\rightarrow} B_1) \rightarrow \mathsf{bool}$
wellInherited	:	$(Sig \underset{m}{\rightarrow} B_1) \rightarrow \mathsf{bool}$
$\mathsf{notConflictThrows}_2$	:	$(ClassType) - \mathsf{set} \times (ClassType) - \mathsf{set} \rightarrow \mathsf{bool}$
getCheckedExceptions	:	$Environment \times (ClassType) - \mathsf{set} \rightarrow (ClassType) - \mathsf{set}$

```
wellFormedClass($\Gamma, \varsigma_1$)  =
   let  $\Gamma.classMap(\varsigma_1) = \langle ms, cs, is, fm, mm, cm \rangle$  in
     $\neg$(validIface($\Gamma, \varsigma_1$))  $\wedge$
     $ms \subseteq \{$public, abstract, final$\}$  $\wedge$
     $\{$final, abstract$\} \not\subseteq ms$  $\wedge$
     validIfaces($\Gamma, is$)  $\wedge$
     (case $cs$ of
        $\emptyset$      $\Rightarrow \varsigma_1 =$ Object
        $\{\varsigma_2\}$  $\Rightarrow$ validClass($\Gamma, \varsigma_2$)  $\wedge$
                 final $\notin \Gamma.classMap(\varsigma_2).modifiers$  $\wedge$
                 (if    abstractMethods(methods$_c$($\Gamma, \varsigma_1$)) $\neq$ []
                  then  abstract $\in ms$
                  endif)  $\wedge$
                 wellOverriddenHidden(transform$_m$($mm$), methods$_c$($\Gamma, \varsigma_2$)$\backslash$
                                  privateMethods(methods$_c$($\Gamma, \varsigma_2$))  $\wedge$
                 wellOverriddenHidden(transform$_m$($mm$), methodsIfaceSet($is$))  $\wedge$
                 wellOverriddenHidden(methods$_c$($\Gamma, \varsigma_2$)$\backslash$
                                  (privateMethods(methods$_c$($\Gamma, \varsigma_2$))),
                                  methodsIfaceSet($\Gamma, is$))  $\wedge$
                 wellInherited(methods$_c$($\Gamma, \varsigma_1$))
     endcase)  $\wedge$
     wellFormedField$_c$($\Gamma, fm$)  $\wedge$
     wellFormedMethod$_c$($\Gamma, mm$)  $\wedge$
     if    $cm =$ []
     then  defaultConstructorInvocation($\Gamma, \varsigma_1$) = (true, $tcs$, $\{\varsigma_3\}$) $\wedge$
           $\neg$checkedException($\Gamma, tcs$)
```

```
    else   wellFormedConstructor$_c(\Gamma, \varsigma_1, cm)$
    endif
  endlet

abstractMethods$([]) = []$
abstractMethods$([sig \mapsto \{(\langle rt, ms, tcs, \rangle, \mu)\}] \dagger mm) =$
    (if    abstract $\in ms$
     then  $[sig \mapsto \{(\langle rt, ms, tcs, \rangle, \mu)\}]$
     else  $[]$
     endif) $\dagger$
     abstractMethods$(mm)$

wellOverriddenHidden$([], mm) =$ true
wellOverriddenHidden$([sig \mapsto \{(\langle rt_1, ms_1, tcs_1\rangle, \mu_1)\}] \dagger mm_1, mm_2) =$
    (if $mm_2(sig) \neq \{\}$ then $\forall \langle rt_2, ms_2, tcs_2), \mu_2\rangle \in mm_2(sig).$
                              $rt_1 = rt_2 \ \wedge$ notConflictThrows$_1(tcs_1, tcs_2) \wedge$
                              private $\notin ms_1 \ \wedge$ final $\notin ms_2 \wedge$
                              static $\in ms_1 \leftrightarrow$ static $\in ms_2$
     else true
     endif) $\wedge$
     wellOverriddenHidden$(mm_1, mm_2)$

wellInherited$([]) =$ true
wellInherited$([sig \mapsto b] \dagger mm) =$ ( $\forall (\langle rt_1, ms_1, tcs_1\rangle, \mu_1) \in b \wedge$
                                   $\forall (\langle rt_2, ms_2, tcs_2\rangle, \mu_2) \in b.$
                                     $rt_1 = rt_2 \wedge$
                                     notConflictThrows$_2(tcs_1, tcs_2)$ ) $\wedge$
                                 wellInherited$(mm)$

notConflictThrows$_2(tcs_1, tcs_2) =$
        $tcs_1 = \emptyset \vee tcs_2 = \emptyset \vee$
        (let $ce_1$ = getCheckedExceptions$(tcs_1)$ ;
             $ce_2$ = getCheckedExceptions$(tcs_2)$
         in  $\forall e \in ce_1.$ subClass$(e, ce_2) \vee$
             $\forall e \in ce_2.$subClass$(e, ce_1)$
         endlet)

getCheckedExceptions$(\Gamma, \emptyset)$            $= \emptyset$
getCheckedExceptions$(\Gamma, \{\varsigma\} \cup cs)$  $=$ (if    checkedException$(\Gamma, \varsigma)$
                                             then  $\{\varsigma\}$
                                             else  $\emptyset$
                                             endif) $\cup$
                                             getCheckedExceptions$(\Gamma, cs)$
```

Well-formed Interface Fields

A field declared in an interface is well-formed if its type is valid and its modifiers set is a subset of {`public`, `final`, `static`}. These constraints are expressed formally as the predicate defined bellow:

$$\mathsf{wellFormedField_i} \quad : \quad \mathit{Environment} \times (\mathit{Identifer} \underset{m}{\rightarrow} \mathit{FieldInfo}) \rightarrow \mathsf{bool}$$

$$\begin{aligned}
&\mathsf{wellFormedField_i}(\Gamma, []) \quad = \quad \mathsf{true} \\
&\mathsf{wellFormedField_i}(\Gamma, [fn \mapsto \langle ft, ms \rangle] \dagger fm) \quad = \\
&\quad \mathsf{validType}(\Gamma, ft) \quad \wedge \\
&\quad ms \subseteq \{\texttt{public}, \texttt{final}, \texttt{static}\} \quad \wedge \\
&\quad \mathsf{wellFormedField_i}(\Gamma, fm)
\end{aligned}$$

Well-formed Interface Methods

A method declared in an interface is well-formed if and only if the following conditions are met:

- Its parameter and result types are valid;
- Its modifiers set is a subset of {`public`, `abstract`};
- It cannot be declared `public` and `private` at the same time;
- Its declared exceptions have to be subclasses of the class `Throwable`.

The above-named conditions are expressed more formally by the following recursive predicate:

$$\mathsf{wellFormedMethod_i} \quad : \quad \mathit{Environment} \times (\mathit{Sig} \underset{m}{\rightarrow} \mathit{MethodInfo}) \rightarrow \mathsf{bool}$$

$$\begin{aligned}
&\mathsf{wellFormedMethod_i}(\Gamma, []) \quad = \quad \mathsf{true} \\
&\mathsf{wellFormedMethod_i}(\Gamma, [(mn, \tau_p) \mapsto \langle rt, ms, tcs \rangle] \dagger mm) \quad = \\
&\quad \mathsf{validType}(\Gamma, \tau_p) \ \wedge \ \mathsf{validType}(\Gamma, rt) \quad \wedge \\
&\quad ms \subseteq \{\texttt{public}, \texttt{abstract}\} \quad \wedge \\
&\quad \mathsf{subClasses}(\Gamma, tcs, \texttt{Throwable}) \quad \wedge \\
&\quad \mathsf{wellFormedMethod_i}(\Gamma, mm)
\end{aligned}$$

Well-formed Interfaces

An interface declaration is considered well-formed if and only if the following conditions is satisfied:

- It is named uniquely, that is there is no class that has the same name.
- Its modifiers set is a subset of {`public`, `abstract`}.
- Its superinterfaces, if any, exist.
- All declared fields, if any, are well-formed.
- All declared methods, if any, are well-formed.
- Each method declared in this interface and overriding a set of methods declared in the superinterfaces must meet the conditions specified by the wellOverriddenHidden predicate.
- If it inherits more than one method with same signature, all these methods must respect some conditions expressed by the wellInherited predicate.

These constraints are expressed formally as the following predicate:

```
wellFormedIface   :   $Environment \times InterfaceType \to \mathsf{bool}$

wellFormedIface($\Gamma, \iota$)   =
    let   $\Gamma.interfaceMap(\iota) = \langle ms, is, fm, mm \rangle$
    in    $\neg$(validClass($\Gamma, \iota$))   $\wedge$
          validIfaces($\Gamma, is$)   $\wedge$
          $ms \subseteq \{$public, abstract$\}$   $\wedge$
          wellOverriddenHidden(transform$_\mathsf{m}$($mm$), methodsIfaceSet($is$))   $\wedge$
          wellInherited(methods$_\mathsf{i}$($\Gamma, \iota$))   $\wedge$
          wellFormedFields$_\mathsf{i}$($\Gamma, fm$)   $\wedge$
          wellFormedMethods$_\mathsf{i}$($\Gamma, mm$)
    end let
```

Well-formed Environments

An environment is well-formed if and only if all classes and interfaces that it declares are well-formed. We assume that the predefined classes as `Object` and `Throwable` are well-formed and that they have already been added to the environment. Since the recursive lookup functions, defined in 6, cannot be called when the class or interface hierarchy is cyclic, before we call them we ensure that no circularities can occur in the class or interface hierarchy. The predicate that checks if an environment is well-formed is defined bellow:

$$\begin{aligned}
&\mathsf{wellFormedEnv} &&: \quad Environment \to \mathsf{bool}\\
&\mathsf{wellFormedMap_c} &&: \quad Environment \times ClassDecl \to \mathsf{bool}\\
&\mathsf{wellFormedMap_i} &&: \quad Environment \times InterfaceDecl \to \mathsf{bool}
\end{aligned}$$

```
wellFormedEnv($\Gamma$)   =
    let $\Gamma = \langle clm, ifm \rangle$
    in  wellFormedMap$_{\mathsf{c}}$($\Gamma, clm$)
        wellFormedMap$_{\mathsf{i}}$($\Gamma, ifm$)
    endlet

wellFormedMap$_{\mathsf{c}}$($\Gamma$, [])   =   true
wellFormedMap$_{\mathsf{c}}$($\Gamma$, [$\varsigma_1 \mapsto \langle ms, cs, is, fm, mm, cm \rangle$] † $clm$)   =
    (case $cs$ of $\emptyset$      $\Rightarrow$ true
                  $\{\varsigma_2\}\Rightarrow$ (if     $\neg(\Gamma \vdash \varsigma_2 \sqsubseteq_{class} \varsigma_1)$
                            then   wellFormedClass($\Gamma, \varsigma_2$)
                            else   false
                            endif)
     endcase) $\wedge$
     wellFormedMap$_{\mathsf{c}}$($clm$)

wellFormedMap$_{\mathsf{i}}$($\Gamma$, [])   =   true
wellFormedMap$_{\mathsf{i}}$($\Gamma$, [$\iota_1 \mapsto \langle ms, is, fm, mm \rangle$] † $ifm$)   =
    (if      $\forall \iota_1 \in is.\ \neg(\Gamma \vdash \iota_2 \sqsubseteq_{interface} \iota_1)$
     then   wellFormedIface($\Gamma, \iota_1$)
     else   false
     endif) $\wedge$
     wellFormedMap$_{\mathsf{i}}$($ifm$)
```

Semantics Objects

In this section, we introduce some semantics objects that will be used in the typing rules:

$\mathcal{LV}$	$\in$	$\mathit{LocalVarMap}$	$=$	$\mathit{Identifier} \underset{m}{\to} (\mathit{Type}, \mathsf{bool})$
$\mathcal{E}$,$\mathcal{UE}$	$\in$	$\mathit{ExceptSet}$	$=$	$(\mathit{ClassType})$-multiset
$\mathcal{PN}$	$\in$	$\mathit{PosName}$	$=$	{‘f’,‘m’,‘c’,‘a’}
$\mathcal{PI}$	$\in$	$\mathit{PosInfo}$	$=$	$\mathit{Type} + \mathit{Sig} + \mathsf{Unit}$
$\mathcal{P}$ ou $(\mathcal{PN}, \mathcal{PI})$	$\in$	$\mathit{Position}$	$=$	$\mathit{PosName} \times \mathit{PosInfo}$
$\mathcal{B}$	$\in$	$\mathit{BoolCouple}$	$=$	bool $\times$ bool
$\mathcal{C}$	$\in$	bool		

Map of local variables $\mathcal{LV}$ The map of local variables maps each local variable name (*Identifier*) to a pair consisting of the local variable type and a boolean variable. The latter is a flag that have the true value if the variable is initialized and the false value, otherwise. This boolean variable is necessary to deal with the flow analysis done by the Java compiler to make sure that each local variable has a *definitely assigned* value before it is being used [9, chapter 16]. The map $\mathcal{LV}$ is very useful to determine the scopes of local variables. When a new variable is introduced, we add a new association in the map, and when a variable is no more visible, we delete its corresponding association from the map. The two operators $\wedge$ and $\vee$ make a special compositions of two maps. When the former is applied to two maps of local variables, it returns a new map that is similar to its left-hand parameter, except that a variable in this map is considered initialized only if it is initialized in both maps. In the other hand, when the operator $\vee$ is applied to two maps of local variables, it returns a new map that is similar to its left-hand parameter, except that a variable in this map is considered initialized only if it is initialized in at least one of these maps.

$$
\begin{array}{lcl}
\_ \wedge \_ & : & \mathit{LocalVarMap} \times \mathit{LocalVarMap} \rightarrow \mathit{LocalVarMap} \\
\_ \vee \_ & : & \mathit{LocalVarMap} \times \mathit{LocalVarMap} \rightarrow \mathit{LocalVarMap}
\end{array}
$$

$$
\begin{array}{lcl}
m \wedge [] & = & m \\
[] \wedge m & = & [] \\
[v_1 \mapsto (t_1, b_1)] \wedge ([v_2 \mapsto (t_2, b_2)] \dagger m_2) & = & \textsf{if}\ \ v_1 = v_2\ \ \textsf{then}\ \ [v_1 \mapsto (t_1, b_1 \wedge b_2)] \\
 & & \qquad\qquad\quad \textsf{else}\ \ [v_1 \mapsto (t_1, b_1)] \wedge m_2 \\
 & & \textsf{endif}
\end{array}
$$

$$
\begin{array}{l}
([v_1 \mapsto (t_1, b_1)] \dagger m_1) \wedge ([v_2 \mapsto (t_2, b_2)] \dagger m_2) \quad = \\
\qquad ([v_1 \mapsto (t_1, b_1)] \wedge ([v_2 \mapsto (t_2, b_2)] \dagger m_2)) \dagger (m_1 \wedge ([v_2 \mapsto (t_2, b_2)] \dagger m_2))
\end{array}
$$

$$
\begin{array}{lcl}
m \vee [] & = & m \\
[] \vee m & = & []
\end{array}
$$

$$
\begin{array}{lcl}
[v_1 \mapsto (t_1, b_1)] \vee ([v_2 \mapsto (t_2, b_2)] \dagger m_2) & = & \textsf{if}\ \ v_1 = v_2\ \ \textsf{then}\ \ [v_1 \mapsto (t_1, b_1 \vee b_2)] \\
 & & \qquad\qquad\quad \textsf{else}\ \ [v_1 \mapsto (t_1, b_1)] \vee m_2 \\
 & & \textsf{endif}
\end{array}
$$

$$
\begin{array}{l}
([v_1 \mapsto (t_1, b_1)] \dagger m_1) \vee ([v_2 \mapsto (t_2, b_2)] \dagger m_2) \quad = \\
\qquad ([v_1 \mapsto (t_1, b_1)] \vee ([v_2 \mapsto (t_2, b_2)] \dagger m_2)) \dagger (m_1 \vee ([v_2 \mapsto (t_2, b_2)] \dagger m_2))
\end{array}
$$

Flag variable $\mathcal{B}$ To deal with unreachable statements and the `return` statement, we introduce the flag variable $\mathcal{B}$ that consists of a couple of boolean variables. The first one will have the true value as soon as a statement completes abruptly. In this case, all following statements will be considered unreachable. The second variable will have the true value when a `return` statement is reached. The latter variable is used later in the typing rules to verify that a non-`void` method contains a reachable `return` statement. In the typing rules, to be sure that the statement to be typed is reachable, we have to verify that the first condition in $\mathcal{B}$ is `false`. The flag variable $\mathcal{B}$ is given with two predicates $\wedge$ and $\vee$:

$$
\begin{array}{lcl}
\_ \wedge \_ & : & (\textsf{bool} \times \textsf{bool}) \times (\textsf{bool} \times \textsf{bool}) \rightarrow (\textsf{bool} \times \textsf{bool}) \\
\_ \vee \_ & : & (\textsf{bool} \times \textsf{bool}) \times (\textsf{bool} \times \textsf{bool}) \rightarrow (\textsf{bool} \times \textsf{bool})
\end{array}
$$

$$\begin{aligned}(b_1, c_1) \wedge (b_2, c_2) &= (b_1 \wedge b_2, c_1 \wedge c_2)\\ (b_1, c_1) \vee (b_2, c_2) &= (b_1 \vee b_2, c_1 \vee c_2)\end{aligned}$$

Multiset of exceptions $\mathcal{E}$ The multiset of exceptions consists of all checked exceptions that can be raised in the current method or constructor body. It will be useful to ensure that this thrown exceptions will be either handled or declared.

Position $\mathcal{P}$ The variable $\mathcal{P}$ consists of a character $\mathcal{PN}$ and an information $\mathcal{PI}$. The character has the 'f' value when the expression to be typed is an initialization expression of a field, the 'm' value when the expression to be typed appears in a method body, the 'c' value when it appears in a constructor body and the 'a' value when the expression is an argument used in an explicit constructor invocation. The information $\mathcal{PI}$ is the type of the field, the method or constructor signature, or the Unit type

Flag variable $\mathcal{C}$ Since it is not legal in Java to assign a value to a `final` field, we introduce the flag variable $\mathcal{C}$ that will have the true value when an access to a `final` field is done. The value of the flag will be examined in the typing rule for a field assignment expression.

Typing Rules

Our static semantics is represented by a set of inference rules that express the well-typedness of declarations, blocks, statements and expressions. We use the function classOrIfaceDecl in order to find a record that corresponds to the current class or interface. Super-class part of an interface declaration is presented by an empty map "[]". In order to improve typing rules readability, only the parts of the record that are actually used in the rule are specified; the others are shown as "_".

Typing Rules for Declarations

Our static semantics for declarations contains several kind of judgment. Below, we introduce typing rules for declarations according to the kind of used judgment.

Judgments of the form $\Gamma \vdash Declaration$: $\diamond$ This kind of judgment is used to type check global declarations of programs, classes and interfaces, and it means that under some typing environment Γ, the phrase *Declaration* of the grammar is well-formed. Typing rules using this kind of judgments are quite simple and are given by the Figure 18.

$\boxed{\textsc{Program}}$ .... $\boxed{\Gamma \vdash \mathit{Program} : \diamond}$

$$\frac{\square}{\Gamma \vdash \varepsilon : \diamond}$$

$$\frac{\Gamma \vdash \mathit{ClassDeclaration} : \diamond \quad \Gamma \vdash \mathit{Program} : \diamond}{\Gamma \vdash \mathit{ClassDeclaration}\ \mathit{Program} : \diamond}$$

$$\frac{\Gamma \vdash \mathit{InterfaceDeclaration} : \diamond \quad \Gamma \vdash \mathit{Program} : \diamond}{\Gamma \vdash \mathit{InterfaceDeclaration}\ \mathit{Program} : \diamond}$$

$\boxed{\textsc{Class-Decl}}$ .... $\boxed{\Gamma \vdash \mathit{ClassDeclaration} : \diamond}$

$$\frac{\begin{array}{c} \Gamma.\mathit{classMap}(\mathit{ClassType}) = \langle ms, \{\varsigma\}, is, fm, mm, cm \rangle \\ \Gamma \vdash \mathit{Modifiers} : ms \quad \Gamma, \mathit{ClassType} \vdash \mathit{Extends} : \{\varsigma\} \quad \Gamma \vdash \mathit{Implements} : is \\ \Gamma, \mathit{ClassType} \vdash \mathit{ClassBodyDeclaration} : \diamond \end{array}}{\begin{array}{c} \Gamma \vdash \mathit{Modifiers}\ \texttt{class}\ \mathit{ClassType}\ \mathit{Extends}\ \mathit{Implements} \\ \texttt{\{}\ \mathit{ClassBodyDeclaration}\ \texttt{\}} : \diamond \end{array}}$$

$\boxed{\textsc{Iface-Decl}}$ .... $\boxed{\Gamma \vdash \mathit{InterfaceDeclaration} : \diamond}$

$$\frac{\begin{array}{c} \Gamma.\mathit{interfaceMap}(\mathit{InterfaceType}) = \langle ms, is, fm, mm \rangle \\ \Gamma \vdash \mathit{Modifiers} : ms \quad \Gamma \vdash \mathit{ExtendsInterfaces} : is \\ \Gamma, \mathit{InterfaceType} \vdash \mathit{InterfaceBodyDeclaration} : \diamond \end{array}}{\begin{array}{c} \Gamma \vdash \mathit{Modifiers}\ \texttt{interface}\ \mathit{InterfaceType}\ \mathit{ExtendsInterfaces} \\ \texttt{\{}\ \mathit{InterfaceBodyDeclaration}\ \texttt{\}} : \diamond \end{array}}$$

Figure 18: Typing rules for declarations: part 1.

$\boxed{\textsc{Extends}}$ .... $\boxed{\Gamma, \varsigma \vdash \mathit{Extends} : \mathit{cs}}$

$$\frac{\square}{\Gamma, \varsigma \vdash \varepsilon : \{\texttt{Object}\}}$$

$$\frac{\square}{\Gamma, \varsigma \vdash \texttt{extends}\ \mathit{ClassType} : \{\mathit{ClassType}\}}$$

Figure 19: Typing rules for declarations: part 2.

Judgments of the form $\Gamma, \varsigma \vdash \mathit{Extends} : \mathit{cs}$ This kind of judgment means that under some typing environment Γ and the current class ς, *Extends* is well-formed and elaborates to a set of classes. The latter contains the superclasse type declared in the clause `extends` of the class ς if it exists, otherwise it contains the class `Object`. Typing rules for *Extends* are given in Figure 19.

Judgments of the form $\Gamma \vdash \mathit{Modifiers} : \mathit{ms}$ This kind of judgment means that under some typing environment Γ, *Modifiers* is well-formed and elaborates to a set of modifiers. Typing rules for *Modifiers* are given in Figure 20.

Judgments of the form $\Gamma \vdash \mathit{Declaration}: (\mu) - \mathsf{set}$ This kind of judgment means that under some typing environment Γ, the *Declaration* of the grammar is well-formed and elaborates to a set of classes or interfaces. Typing rules that use this kind of judgment are introduced in Figure 21.

Judgments of the form $\Gamma \vdash \mathit{Parameter} : \mathit{pn}, \tau_p$ This kind of judgment is used to type-check the parameter of a method or a constructor and it means that under some typing environment Γ, the parameter declaration *Parameter* is well-formed and elaborates to the parameter name and type. Typing rules for *Parameter* are given in Figure 22. When the method or the constructor do not take parameter, the evaluation of *Parameter* must elaborate to the couple $(\varepsilon, \mathsf{Unit})$.

Judgments of the form $\Gamma, \mu \vdash \mathit{Declaration}: \diamond$ This kind of judgment means that under some typing environment Γ and the current class or interface μ, i.e. the one that we are currently typing, the phrase *Declaration* of the grammar is well-formed. Typing rules for class and interface bodies including global declarations of their components, use this kind of judgment and they are defined in Figures 23 and 24. Most of those rules are straightforward. Below we comment some of them:

- Field-Decl: Since `final` fields must be initialized when they are declared, a field declaration that does not include an initialization expression cannot be

$\boxed{\textsc{Modifiers}}$ .... $\boxed{\Gamma \vdash \mathit{Modifiers} : \mathit{ms}}$

$$\frac{\square}{\Gamma \vdash \varepsilon : \emptyset}$$

$$\frac{\Gamma \vdash \mathit{Modifiers} : \mathit{ms}}{\Gamma \vdash \texttt{public}\ \mathit{Modifiers} : \{\texttt{public}\} \cup \mathit{ms}}$$

$$\frac{\Gamma \vdash \mathit{Modifiers} : \mathit{ms}}{\Gamma \vdash \texttt{private}\ \mathit{Modifiers} : \{\texttt{private}\} \cup \mathit{ms}}$$

$$\frac{\Gamma \vdash \mathit{Modifiers} : \mathit{ms}}{\Gamma \vdash \texttt{static}\ \mathit{Modifiers} : \{\texttt{static}\} \cup \mathit{ms}}$$

$$\frac{\Gamma \vdash \mathit{Modifiers} : \mathit{ms}}{\Gamma \vdash \texttt{abstract}\ \mathit{Modifiers} : \{\texttt{abstract}\} \cup \mathit{ms}}$$

$$\frac{\Gamma \vdash \mathit{Modifiers} : \mathit{ms}}{\Gamma \vdash \texttt{final}\ \mathit{Modifiers} : \{\texttt{final}\} \cup \mathit{ms}}$$

$$\frac{\Gamma \vdash \mathit{Modifiers} : \mathit{ms}}{\Gamma \vdash \texttt{synchronized}\ \mathit{Modifiers} : \{\texttt{synchronized}\} \cup \mathit{ms}}$$

$$\frac{\Gamma \vdash \mathit{Modifiers} : \mathit{ms}}{\Gamma \vdash \texttt{native}\ \mathit{Modifiers} : \{\texttt{native}\} \cup \mathit{ms}}$$

$$\frac{\Gamma \vdash \mathit{Modifiers} : \mathit{ms}}{\Gamma \vdash \texttt{volatile}\ \mathit{Modifiers} : \{\texttt{volatile}\} \cup \mathit{ms}}$$

$$\frac{\Gamma \vdash \mathit{Modifiers} : \mathit{ms}}{\Gamma \vdash \texttt{transient}\ \mathit{Modifiers} : \{\texttt{transient}\} \cup \mathit{ms}}$$

Figure 20: Typing rules for declarations: part 3.

$\boxed{\text{SUPER-INTERFACES}}$ .... $\boxed{\Gamma \vdash \mathit{InterfaceTypeList} : \mathit{is}}$

$$\frac{\square}{\Gamma \vdash \mathit{InterfaceType} : \{\mathit{InterfaceType}\}}$$

$$\frac{\Gamma \vdash \mathit{InterfaceTypeList} : \mathit{is}}{\Gamma \vdash \mathit{IntrefaceType}\ \texttt{,}\ \mathit{InterfaceTypeList} : \{\mathit{InterfaceType}\} \cup \mathit{is}}$$

$\boxed{\text{IMPLEMENTS}}$ .... $\boxed{\Gamma \vdash \mathit{Implements} : \mathit{is}}$

$$\frac{\square}{\Gamma \vdash \varepsilon : \emptyset}$$

$$\frac{\Gamma \vdash \mathit{InterfaceTypeList} : \mathit{is}}{\Gamma \vdash \texttt{implements}\ \mathit{InterfaceTypeList} : \mathit{is}}$$

$\boxed{\text{EXTENDS-INTERFACES}}$ .... $\boxed{\Gamma \vdash \mathit{ExtendsInterfaces} : \mathit{is}}$

$$\frac{\square}{\Gamma \vdash \varepsilon : \emptyset}$$

$$\frac{\Gamma \vdash \mathit{InterfaceTypeList} : \mathit{is}}{\Gamma \vdash \texttt{extends}\ \mathit{InterfaceTypeList} : \mathit{is}}$$

$\boxed{\text{THROWS}}$ .... $\boxed{\Gamma \vdash \mathit{Throws} : \mathit{cs}}$

$$\frac{\square}{\Gamma \vdash \varepsilon : \emptyset}$$

$$\frac{\Gamma \vdash \mathit{ClassTypeList} : \mathit{cs}}{\Gamma \vdash \texttt{throws}\ \mathit{ClassTypeList} : \mathit{cs}}$$

$$\frac{\square}{\Gamma \vdash \mathit{ClassType} : \{\mathit{ClassType}\}}$$

$$\frac{\Gamma \vdash \mathit{ClassType} : \{\mathit{ClassType}\} \quad \mathit{ClassTypeList} : \mathit{cs}}{\Gamma \vdash \mathit{ClassType}\ \mathit{ClassTypeList} : \{\mathit{ClassType}\} \cup \mathit{cs}}$$

Figure 21: Typing rules for declarations: part 4.

$$\boxed{\textsc{Parameter}} \quad \dots\dots\dots\dots\dots\dots\dots\dots\dots\dots\dots \quad \boxed{\Gamma \vdash \mathit{Parameter} : pn, \tau_p}$$

$$\frac{\square}{\Gamma \vdash \varepsilon : \varepsilon, \mathsf{Unit}}$$

$$\frac{\square}{\Gamma \vdash \mathit{Type}\ \mathit{Identifier} : \mathit{Identifier}, \mathit{Type}}$$

Figure 22: Typing rules for declarations: part 5.

declared `final`. Also for this reason, when a field declaration do not include an initialization expression, the field must be declared in a class ς and not in an interface, since interface fields are implicitly `final`. Moreover, the evaluation of the expression used in the field initialization must yield an $\emptyset$ of checked exceptions. Indeed, according to the Java Language Specification [9, chapter 8], it is a compile-time error if the evaluation of the variable initializer for a field of a class or an interface completes abruptly raising a checked exception.

- Method-Decl: If a method has a body, then it cannot be declared `abstract` nor `native`. This constraint is expressed in the typing rule by $\{$`abstract`$,$`native`$\} \cap ms = \emptyset$. A method body is a block. When we type-check this block, we create a new local variables map that must contain an entry for the parameter of the method that is considered as a special local variable. Since the parameter is initialized when the method is called, its initialization flag in the map must be true. The scope of the local variables declared in a method body is the method body itself. Thus, once the latter is evaluated, the map of local variables must be empty. If some checked exceptions result from this evaluation, then for each thrown checked exception that has not been handled in the method body, the `throws` clause of the method must contain the class of this exception or one of its superclasses. To do so, we use the function $\mathsf{unCaughtExceptions}$, defined below, to compute the set of checked exceptions that may be raised in the method body and that were not declared in the method's `throws` clause. Then, we require the computed set to be the empty set $\emptyset$.

$$\begin{array}{lcl}
\mathsf{unCaughtExceptions} & : & \mathit{Environment} \times \mathit{ExceptSet} \times (\mathit{ClassType}) - \mathsf{set} \rightarrow \\
 & & (\mathit{ClassType}) - \mathsf{set}
\end{array}$$

$$\begin{array}{lcl}
\mathsf{unCaughtExceptions}(\Gamma, \emptyset, tcs) & = & \emptyset \\
\mathsf{unCaughtExceptions}(\Gamma, \{\!|\varsigma|\!\} \cup \mathcal{E}, tcs) & = & (\mathsf{if}\quad \mathsf{subClass}(\Gamma, \varsigma, tcs) \\
 & & \ \ \mathsf{then}\quad \emptyset \\
 & & \ \ \mathsf{else}\quad \{\varsigma\}
\end{array}$$

endif) $\cup$
unCaughtExceptions($\Gamma, \mathcal{E}, tcs$)

Finally, a method declared to have a return type, i.e. not declared `void`, must contain in its body an accessible `return` statement. This is expressed as the logical formula $\mathit{ResultType} \neq$ `void` $\Rightarrow b_2 =$ true.

- Abstract-Method-Decl: This rule is similar to the previous one, but the method to be type-checked has no body, and then it must be declared `abstract` or `native`.

- Constructor-Decl This rule is also similar to Method-Decl. A constructor declaration can be considered as a non-`abstract` method that is declared `void`.

Judgments of the form: $\Gamma, \mu, \mathcal{LV}_1, \mathcal{E}_1, \mathcal{B}_1, \mathcal{P} \vdash \mathit{ConstructorBody}\colon \mathcal{LV}_2, \mathcal{E}_2, \mathcal{B}_2$ **and of the form** $\Gamma, \mu, \mathcal{LV}_1, \mathcal{E}_1, \mathcal{P} \vdash \mathit{ExplicitConsInvocation}\colon \mathcal{LV}_2, \mathcal{E}_2$ The first kind of judgment means that under the static context that consists of the static environment Γ, the current class or interface μ, the local variables map $\mathcal{LV}_1$, the set of checked exceptions $\mathcal{E}_1$, the flag variable $\mathcal{B}_1$ and the position $\mathcal{P}$, a declaration is well-formed and it may change the components $\mathcal{LV}_1$, $\mathcal{E}_1$ and $\mathcal{B}_1$ of the context. The second kind of judgment is quite similar to the first one, except that the flag variable does not form part of the context. Typing rules for constructor bodies and explicit constructor invocations use this kind of judgment and they are defined in Figure 25. Below we comment some of them.

- Default-Cons-Invocation: When the constructor body does not begin with an explicit constructor invocation (this is represented by ε in the grammar), the constructor of the direct superclass that takes no arguments is implicitly invoked, i.e. the call `super ();` is automatically done. To type-check this implicit invocation, we call the function defaultConstructorInvocation, defined in section 6, that must find the superclass's constructor with no arguments. If the latter declares checked exceptions in its clause `throws`, then for each checked exception type, the same exception type or one of its supertypes must appear in the `throws` clause of the calling constructor, i.e. the one that we are currently typing. Thus, we can be ensured that all checked exceptions that may be raised from the implicit constructor call, will be handled later.

- This-Cons-Invocation: When the constructor begins with an explicit constructor invocation, we have to find the invoked constructor and then check, as for previous rule, that all checked exceptions that it declares are specified in the clause `throws` of the caller constructor. In addition, we have to ensure that the constructor does not invoke itself. This is expressed in the typing rule by the logical formula snd(snd($\mathcal{P}$)) $\neq \tau_a$.

$$\boxed{\textsc{Class-Body}} \quad \dots\dots\dots\dots\dots\dots \quad \boxed{\Gamma, \varsigma \vdash \mathit{ClassBodyDeclaration} : \diamond}$$

$$\frac{\square}{\Gamma, \varsigma \vdash \varepsilon : \diamond}$$

$$\frac{\Gamma, \varsigma \vdash \mathit{FieldDeclaration} : \diamond \quad \Gamma, \varsigma \vdash \mathit{ClassBodyDeclaration} : \diamond}{\Gamma, \varsigma \vdash \mathit{FieldDeclaration}\ \mathit{ClassBodyDeclaration} : \diamond}$$

$$\frac{\Gamma, \varsigma \vdash \mathit{MethodDeclaration} : \diamond \quad \Gamma, \varsigma \vdash \mathit{ClassBodyDeclaration} : \diamond}{\Gamma, \varsigma \vdash \mathit{MethodDeclaration}\ \mathit{ClassBodyDeclaration} : \diamond}$$

$$\frac{\Gamma, \varsigma \vdash \mathit{AbstractMethodDeclaration} : \diamond \quad \Gamma, \varsigma \vdash \mathit{ClassBodyDeclaration} : \diamond}{\Gamma, \varsigma \vdash \mathit{AbstractMethodDeclaration}\ \mathit{ClassBodyDeclaration} : \diamond}$$

$$\frac{\Gamma, \varsigma \vdash \mathit{ConstructorDeclaration} : \diamond \quad \Gamma, \varsigma \vdash \mathit{ClassBodyDeclaration} : \diamond}{\Gamma, \varsigma \vdash \mathit{ConstructorDeclaration}\ \mathit{ClassBodyDeclaration} : \diamond}$$

$$\boxed{\textsc{Iface-Body}} \quad \dots\dots\dots\dots\dots\dots \quad \boxed{\Gamma, \iota \vdash \mathit{InterfaceBodyDeclaration} : \diamond}$$

$$\frac{\square}{\Gamma, \iota \vdash \varepsilon : \diamond}$$

$$\frac{\Gamma, \iota \vdash \mathit{FieldDeclaration} : \diamond \quad \Gamma, \iota \vdash \mathit{InterfaceBodyDeclaration} : \diamond}{\Gamma, \iota \vdash \mathit{FieldDeclaration}\ \mathit{InterfaceBodyDeclaration} : \diamond}$$

$$\frac{\Gamma, \iota \vdash \mathit{AbstractMethodDeclaration} : \diamond \quad \Gamma, \iota \vdash \mathit{InterfaceBodyDeclaration} : \diamond}{\Gamma, \iota \vdash \mathit{AbstractMethodDeclaration}\ \mathit{ClassBodyDeclaration} : \diamond}$$

$$\boxed{\textsc{Field-Decl}} \quad \dots\dots\dots\dots\dots\dots \quad \boxed{\Gamma, \mu \vdash \mathit{FieldDeclaration} : \diamond}$$

$$\frac{\begin{array}{c}\Gamma.\mathit{classMap}(\varsigma) = \langle \_, \_, \_, fm, \_, \_ \rangle \\ fm(\mathit{Identifier}) = \langle \mathit{Type}, ms \rangle \quad \Gamma \vdash \mathit{Modifiers} : ms \quad \texttt{final} \notin ms\end{array}}{\Gamma, \varsigma \vdash \mathit{Modifiers}\ \mathit{Type}\ \mathit{Identifier}\ \texttt{;} : \diamond}$$

$$\frac{\begin{array}{c}\mathsf{classOrIfaceDecl}(\Gamma, \mu) = \langle \_, \_, \_, fm, \_, \_ \rangle \\ fm(\texttt{Identifier}) = \langle \mathit{Type}, ms \rangle \quad \Gamma \vdash \mathit{Modifiers} : ms \\ \Gamma, \mu, [], \emptyset, (\text{‘f’}, \mathit{Identifier}) \vdash \mathit{Expression} : \tau_e, [], \emptyset, \mathcal{C} \quad \Gamma \vdash \tau_e \sqsubseteq_{impl} \mathit{Type}\end{array}}{\Gamma, \mu \vdash \mathit{Modifiers}\ \mathit{Type}\ \mathit{Identifier}\ \texttt{=}\ \mathit{Expression}\ \texttt{;} : \diamond}$$

$$\frac{\begin{array}{c}\mathsf{classOrIfaceDecl}(\Gamma, \mu) = \langle \_, \_, \_, fm, \_, \_ \rangle \\ fm(\mathit{Identifier}) = \langle \mathit{SimpleType}\,\texttt{[]}, ms \rangle \quad \Gamma \vdash \mathit{Modifiers} : ms \\ \Gamma, \mu, [], \emptyset, (\text{‘f’}, \mathit{Identifier}) \vdash \mathit{ArrayInitializer} : \sigma, [], \emptyset \\ \Gamma \vdash \sigma \sqsubseteq_{impl} \mathit{SimpleType}\end{array}}{\Gamma, \mu \vdash \mathit{Modifiers}\ \mathit{SimpleType}\,\texttt{[]}\ \mathit{Identifier}\ \texttt{=}\ \mathit{ArrayInitializer}\ \texttt{;} : \diamond}$$

Figure 23: Typing rules for declarations: part 6.

$$\boxed{\textsc{Method-Decl}} \; \cdots\cdots\cdots\cdots \; \boxed{\Gamma, \varsigma \vdash \mathit{MethodDeclaration} : \diamond}$$

$$\frac{\begin{array}{c}\Gamma.\mathit{classMap}(\varsigma) = \langle \_, \_, \_, \_, \mathit{mm}, \_\rangle \quad \Gamma \vdash \mathit{Parameter} : \mathit{pn}, \tau_p \\ \mathit{mm}(\mathit{Identifier}, \tau_p) = \langle \mathit{ResultType}, \mathit{ms}, \mathit{tcs}\rangle \quad \Gamma \vdash \mathit{Modifiers} : \mathit{ms} \\ \Gamma \vdash \mathit{Throws} : \mathit{tcs} \quad \{\texttt{abstract}, \texttt{native}\} \cap \mathit{ms} = \emptyset \quad \mathcal{B} = (\mathsf{false}, \mathsf{false}) \\ \Gamma, \varsigma, [\mathit{pn} \mapsto (\tau_p, \mathsf{true})], \emptyset, \mathcal{B}, (\text{`m'}, (\mathit{Identifier}, \tau_p)) \vdash \mathit{Block} : [], \mathcal{E}, (b_1, b_2) \\ \mathsf{unCaughtExceptions}(\Gamma, \mathcal{E}, \mathit{tcs}) = \emptyset \quad \mathit{ResultType} \neq \texttt{void} \Rightarrow b_2 = \mathsf{true}\end{array}}{\Gamma, \varsigma \vdash \mathit{Modifiers}\ \mathit{ResultType}\ \mathit{Identifier}\ \texttt{(}\ \mathit{Parameter}\ \texttt{)}\ \mathit{Throws}\ \mathit{Block} : \diamond}$$

$$\boxed{\textsc{Abstract-Method-Decl}} \; \cdots\cdots\cdots \; \boxed{\Gamma, \mu \vdash \mathit{AbstractMethodDeclaration} : \diamond}$$

$$\frac{\begin{array}{c}\mathsf{classOrIfaceDecl}(\Gamma, \mu) = \langle \_, \_, \_, \_, \mathit{mm}, \_\rangle \quad \Gamma \vdash \mathit{Parameter} : \mathit{pn}, \tau_p \\ \mathit{mm}(\mathit{Identifier}, \tau_p) = \langle \mathit{ResultType}, \mathit{ms}, \mathit{tcs}\rangle \quad \Gamma \vdash \mathit{Modifiers} : \mathit{ms} \\ \Gamma \vdash \mathit{Throws} : \mathit{tcs} \quad \{\texttt{abstract}, \texttt{native}\} \cap \mathit{ms} \neq \emptyset\end{array}}{\Gamma, \mu \vdash \mathit{Modifiers}\ \mathit{ResultType}\ \mathit{Identifier}\ \texttt{(}\ \mathit{Parameter}\ \texttt{)}\ \mathit{Throws}\ \texttt{;} : \diamond}$$

$$\boxed{\textsc{Constructor-Decl}} \; \cdots\cdots\cdots\cdots \; \boxed{\Gamma, \varsigma \vdash \mathit{ConstructorDeclaration} : \diamond}$$

$$\frac{\begin{array}{c}\Gamma.\mathit{classMap}(\varsigma) = \langle \_, \_, \_, \_, \_, \mathit{cm}\rangle \quad \Gamma \vdash \mathit{Parameter} : \mathit{pn}, \tau_p \\ \mathit{cm}(\mathit{ClassType}, \tau_p) = \langle \mathit{ms}, \mathit{tcs}\rangle \quad \Gamma \vdash \mathit{Modifiers} : \mathit{ms} \\ \Gamma \vdash \mathit{Throws} : \mathit{tcs} \quad \mathcal{B} = (\mathsf{false}, \mathsf{false}) \\ \Gamma, \mathit{ClassType}, [\mathit{pn} \mapsto \tau_p], \emptyset, \mathcal{B}, (\text{`c'}, (\mathit{ClassType}, \tau_p)) \vdash \mathit{ConstructorBody} : [], \mathcal{E}, \mathcal{B} \\ \mathsf{unCaughtExceptions}(\Gamma, \mathcal{E}, \mathit{tcs}) = \emptyset\end{array}}{\Gamma, \varsigma \vdash \mathit{Modifiers}\ \mathit{ClassType}\ \texttt{(}\ \mathit{Parameter}\ \texttt{)}\ \mathit{Throws}\ \mathit{ConstructorBody} : \diamond}$$

Figure 24: Typing rules for declarations: part 7.

- SUPER-CONS-INVOCATION: This rule is quite similar to the previous one, but instead of to check that the constructor does not invoke itself, we must verify that the invoked constructor is not `private`.

Judgments of the form: $\Gamma, \mu, \mathcal{LV}_1, \mathcal{E}_1, \mathcal{P} \vdash \mathit{Declaration}: \sigma, \mathcal{LV}_2, \mathcal{E}_2$ This kind of judgment means that under a static context consisting of Γ, μ, $\mathcal{LV}_1$, $\mathcal{E}_1$, $\mathcal{B}_1$ and $\mathcal{P}$, the declaration *Declaration* is well-formed and it may change the components $\mathcal{LV}_1$ and $\mathcal{E}_1$.Typing rules using this kind of judgment are defined in Figure 26. An array initializer is a list of expressions separated by a comma and delimited by `{` and `}`. It provides some initial values to an array. All the expressions must have a common super-type noted by σ_3 in the rule named INIT-EXPR. When the array initializer is empty (`{ }`), it can be used to initialize different array types.

Typing Rules for blocks

Our static semantics for Java blocks uses judgments of the form $\Gamma, \varsigma, \mathcal{LV}_1, \mathcal{E}_1, \mathcal{B}_1, \mathcal{P} \vdash \mathit{Block} : \mathcal{LV}_2, \mathcal{E}_2, \mathcal{B}_2$, which mean that under the static context $\Gamma, \varsigma, \mathcal{LV}_1, \mathcal{E}_1, \mathcal{B}_1, \mathcal{P}$, the block *Block* is well-formed and its evaluation may modify the components $\mathcal{LV}_1$, $\mathcal{E}_1$ and $\mathcal{B}_1$ of the context. For the sake of readability, we present the static semantics for the Java blocks in three steps. First, we introduce typing rules for global declarations of blocks, then typing rules for local variable declarations, and finally those for statements.

Typing rules for block declarations

Typing rules for block declarations are given in Figure 27. The scope of local variables declared in a block is the block itself. Thus, all variables declared in a block must be removed from the map of local variables after the block evaluation. However, if a variable, which is declared out of the block and is accessible from it, has been modified in the block, then this modification is visible after the evaluation of the block. To express this constraint in the typing rules, we compute the map of local variables after the evaluation of the block by $\mathcal{LV}_2/\mathcal{LV}_1$.

Typing rules for local variable declarations

Figure 28 introduces the typing rules for local variable declarations. When a new local variable is declared, we create an entry in the map of local variables. If the variable declaration statement includes an initialization expression, then the variable is considered initialized, otherwise it is not. In addition, a local variable is visible from its initialization expression, i.e. the expression that appears in the right-hand side of the local variable assignment, but it is considered not initialized

$\boxed{\textsc{Constructor-Body}}\ \ldots\ldots\ \boxed{\Gamma, \varsigma, \mathcal{LV}_1, \mathcal{E}_1, \mathcal{B}_1, \mathcal{P} \vdash \mathit{ConstructorBody} : \mathcal{LV}_2, \mathcal{E}_2, \mathcal{B}_2}$

$$\frac{\begin{array}{c}\Gamma, \varsigma, \mathcal{LV}_1, \mathcal{E}_1, \mathcal{P} \vdash \mathit{ExplicitConsInvocation} : \mathcal{E}_2, \mathcal{LV}_2 \\ \Gamma, \varsigma, \mathcal{LV}_1, \mathcal{E}_2, \mathcal{P}, \mathcal{B}_1 \vdash \mathit{BlockStatementsOrEmpty} : \mathcal{LV}_2, \mathcal{E}_2, \mathcal{B}_2\end{array}}{\begin{array}{c}\Gamma, \varsigma, \mathcal{LV}_1, \mathcal{E}_1, \mathcal{B}_1, \mathcal{P} \vdash \texttt{\{}\ \mathit{ExplicitConsInvocation}\ \mathit{BlockStatementsOrEmpty}\ \texttt{\}} : \\ \mathcal{LV}_2, \mathcal{E}_2, \mathcal{B}_2\end{array}}$$

$\boxed{\textsc{Explicit-Cons-Inv}}\ \ldots\ldots\ldots\ \boxed{\Gamma, \varsigma, \mathcal{LV}_1, \mathcal{E}_1, \mathcal{P} \vdash \mathit{ExplicitConsInvocation} : \mathcal{LV}_2, \mathcal{E}_2}$

$\boxed{\textsc{Default-Cons-Invocation}}$

$$\frac{\begin{array}{c}\mathsf{defaultConstructorInvocation}(\Gamma, \varsigma_1) = (\mathsf{true}, \mathit{tcs}_1, \{\varsigma_2\}) \\ \Gamma.\mathit{classMap}(\varsigma_1).\mathit{constructors}(\mathsf{snd}(\mathcal{P})) = \langle \mathit{ms}, \mathit{tcs}_2 \rangle \\ \mathsf{getCheckedExceptions}(\Gamma, \mathit{tcs}_1) = \mathit{tcs}_3 \quad \mathsf{subClasses}(\Gamma, \mathit{tcs}_3, \mathit{tcs}_2)\end{array}}{\Gamma, \varsigma_1, \mathcal{LV}_1, \mathcal{E}_1, \mathcal{P} \vdash \varepsilon : \mathcal{LV}_1, \mathcal{E}_1}$$

$\boxed{\textsc{This-Cons-Invocation}}$

$$\frac{\begin{array}{c}\Gamma, \varsigma, \mathcal{LV}_1, \mathcal{E}_1, (\text{‘}a\text{’}, \mathsf{Unit}) \vdash \mathit{Argument} : \tau_a, \mathcal{E}_2, \mathcal{LV}_2, \mathcal{C} \quad \mathsf{snd}(\mathsf{snd}(\mathcal{P})) \neq \tau_a \\ \Gamma.\mathit{classMap}(\varsigma).\mathit{constructors}(\mathsf{snd}(\mathcal{P})) = \langle \mathit{ms}_1, \mathit{tcs}_1 \rangle \\ \Gamma.\mathit{classMap}(\varsigma).\mathit{constructors}(\varsigma, \tau_a) = \langle \mathit{ms}_2, \mathit{tcs}_2 \rangle \\ \mathsf{getCheckedExceptions}(\Gamma, \mathit{tcs}_2) = \mathit{tcs}_3 \quad \mathsf{subClasses}(\Gamma, \mathit{tcs}_3, \mathit{tcs}_1)\end{array}}{\Gamma, \varsigma, \mathcal{LV}_1, \mathcal{E}_1, \mathcal{P} \vdash \texttt{this (} \mathit{Argument} \texttt{) ;} : \mathcal{LV}_2, \mathcal{E}_2}$$

$\boxed{\textsc{Super-Cons-Invocation}}$

$$\frac{\begin{array}{c}\Gamma, \varsigma, \mathcal{LV}_1, \mathcal{E}_1, (\text{‘}a\text{’}, \mathsf{Unit}) \vdash \mathit{Argument} : \tau_a, \mathcal{E}_2, \mathcal{LV}_2, \mathcal{C} \\ \Gamma.\mathit{classMap}(\varsigma_1) = \langle \_, \{\varsigma_2\}, \_, \_, \_, \mathit{cm} \rangle \\ \mathit{cm}(\mathsf{snd}(\mathcal{P})) = \langle \mathit{ms}_1, \mathit{tcs}_1 \rangle \quad \mathsf{constructors}(\Gamma, \varsigma_2)(\varsigma_2, \tau_a) = \langle \mathit{ms}_2, \mathit{tcs}_2 \rangle \\ \texttt{private} \notin \mathit{ms}_2 \quad \mathsf{getCheckedExceptions}(\Gamma, \mathit{tcs}_2) = \mathit{tcs}_3 \\ \mathsf{subClasses}(\Gamma, \mathit{tcs}_3, \mathit{tcs}_1)\end{array}}{\Gamma, \varsigma_1, \mathcal{LV}_1, \mathcal{E}_1, \mathcal{P} \vdash \texttt{super (} \mathit{Argument} \texttt{) ;} : \mathcal{LV}_2, \mathcal{E}_2}$$

Figure 25: Typing rules for declarations: part 8.

$$\boxed{\textsc{Init-Tab}} \;\dotfill\; \boxed{\Gamma, \mu, \mathcal{LV}_1, \mathcal{E}_1, \mathcal{P} \vdash \mathit{ArrayInitializer} : \sigma, \mathcal{LV}_2, \mathcal{E}_2}$$

$$\frac{\square}{\Gamma, \mu, \mathcal{LV}, \mathcal{E}, \mathcal{P} \vdash \{\ \} : \sigma, \mathcal{LV}, \mathcal{E}}$$

$$\frac{\Gamma, \mu, \mathcal{LV}_1, \mathcal{E}_1, \mathcal{P} \vdash \mathit{ExpressionInitializer} : \sigma, \mathcal{LV}_2, \mathcal{E}_2}{\Gamma, \mu, \mathcal{LV}_1, \mathcal{E}_1, \mathcal{P} \vdash \{\ \mathit{ExpressionInitializer}\ \} : \sigma, \mathcal{LV}_2, \mathcal{E}_2}$$

$$\boxed{\textsc{Expr-Init}} \;\dotfill\; \boxed{\Gamma, \mu, \mathcal{LV}_1, \mathcal{E}_1, \mathcal{P} \vdash \mathit{ExpressionInitializer} : \sigma, \mathcal{LV}_2, \mathcal{E}_2}$$

$$\frac{\begin{array}{c}\Gamma, \mu, \mathcal{LV}_1, \mathcal{E}_1, \mathcal{P} \vdash \mathit{Expression} : \sigma_1, \mathcal{LV}_2, \mathcal{E}_2, \mathcal{C} \\ \Gamma, \mu, \mathcal{LV}_2, \mathcal{E}_2, \mathcal{P} \vdash \mathit{ExpressionInitializer} : \sigma_2, \mathcal{LV}_3, \mathcal{E}_3 \\ \Gamma \vdash \sigma_1 \sqsubseteq_{impl} \sigma_3 \quad \Gamma \vdash \sigma_2 \sqsubseteq_{impl} \sigma_3\end{array}}{\Gamma, \mu, \mathcal{LV}_1, \mathcal{E}_1, \mathcal{P} \vdash \mathit{Expression\ ExpressionInitializer} : \sigma_3, \mathcal{LV}_3, \mathcal{E}_3}$$

Figure 26: Typing rules for declarations: part 9.

$$\boxed{\textsc{Block}} \;\dotfill\; \boxed{\Gamma, \varsigma, \mathcal{LV}_1, \mathcal{E}_1, \mathcal{B}_1, \mathcal{P} \vdash \mathit{Block} : \mathcal{LV}_2, \mathcal{E}_2, \mathcal{B}_2}$$

$$\frac{\Gamma, \varsigma, \mathcal{LV}_1, \mathcal{E}_1, (\mathsf{false}, b), \mathcal{P} \vdash \mathit{BlockStatementsOrEmpty} : \mathcal{LV}_2, \mathcal{E}_2, \mathcal{B}}{\Gamma, \varsigma, \mathcal{LV}_1, \mathcal{E}_1, (\mathsf{false}, b), \mathcal{P} \vdash \{\ \mathit{BlockStatementsOrEmpty}\ \} : \mathcal{LV}_2/\mathcal{LV}_1, \mathcal{E}_2, \mathcal{B}}$$

$$\boxed{\textsc{Empty-Block}} \;.\; \boxed{\Gamma, \varsigma, \mathcal{LV}_1, \mathcal{E}_1, \mathcal{B}_1, \mathcal{P} \vdash \mathit{BlockStatementsOrEmpty} : \mathcal{LV}_2, \mathcal{E}_2, \mathcal{B}_2}$$

$$\frac{\square}{\Gamma, \varsigma, \mathcal{LV}, \mathcal{E}, (\mathsf{false}, b), \mathcal{P} \vdash \varepsilon : \mathcal{LV}, \mathcal{E}, (\mathsf{false}, b)}$$

$$\boxed{\textsc{Block-Stmt}} \;\dotfill\; \boxed{\Gamma, \varsigma, \mathcal{LV}_1, \mathcal{E}_1, \mathcal{B}_1, \mathcal{P} \vdash \mathit{BlockStatements} : \mathcal{LV}_2, \mathcal{E}_2, \mathcal{B}_2}$$

$$\frac{\begin{array}{c}\Gamma, \varsigma, \mathcal{LV}_1, \mathcal{E}_2, (\mathsf{false}, b), \mathcal{P} \vdash \mathit{BlockStatement} : \mathcal{LV}_2, \mathcal{E}_2, \mathcal{B}_1 \\ \Gamma, \varsigma, \mathcal{LV}_2, \mathcal{E}_2, \mathcal{B}_1, \mathcal{P} \vdash \mathit{BlockStatements} : \mathcal{LV}_3, \mathcal{E}_3, \mathcal{B}_2\end{array}}{\Gamma, \varsigma, \mathcal{LV}_1, \mathcal{E}_2, (\mathsf{false}, b), \mathcal{P} \vdash \mathit{BlockStatement\ BlockStatements} : \mathcal{LV}_3, \mathcal{E}_3, \mathcal{B}_2}$$

Figure 27: Typing rules for Blocks.

$\boxed{\textsc{Local-Var}}$ .... $\boxed{\Gamma, \varsigma, \mathcal{LV}_1, \mathcal{E}_1, \mathcal{B}_1, \mathcal{P} \vdash \mathit{LocalVariable} : \mathcal{LV}_2, \mathcal{E}_2, \mathcal{B}_2}$

$$\frac{\mathsf{validType}(\Gamma, \mathit{Type}) \quad \mathcal{LV}(\mathit{Identifier}) = \bot}{\begin{array}{l}\Gamma, \varsigma, \mathcal{LV}, \mathcal{E}, (\mathsf{false}, b), \mathcal{P} \vdash \mathit{Type}\ \mathit{Identifier}\ \texttt{;} : \\ \qquad \mathcal{LV} \dagger [\mathit{Identifier} \mapsto (\mathit{Type}, \mathsf{false})], \mathcal{E}, (\mathsf{false}, b)\end{array}}$$

$$\frac{\begin{array}{c}\mathsf{validType}(\Gamma, \mathit{Type}) \quad \mathcal{LV}(\mathit{Identifier}) = \bot \\ \Gamma, \varsigma, \mathcal{LV}_1 \dagger [\texttt{Identifier} \mapsto (\mathit{Type}, \mathsf{false})], \mathcal{E}_1, \mathcal{P} \vdash \mathit{Expression} : \tau_e, \mathcal{LV}_2, \mathcal{E}_2, \mathcal{C} \\ \Gamma \vdash \tau_e \sqsubseteq_{impl} \mathit{Type}\end{array}}{\begin{array}{l}\Gamma, \varsigma, \mathcal{LV}_1, \mathcal{E}_1, (\mathsf{false}, b), \mathcal{P} \vdash \mathit{Type}\ \mathit{Identifier}\ \texttt{=}\ \mathit{Expression}\ \texttt{;} : \\ \qquad \mathcal{LV}_2 \dagger [\mathit{Identifier} \mapsto (\mathit{Type}, \mathsf{true})], \mathcal{E}_2, (\mathsf{false}, b)\end{array}}$$

$$\frac{\begin{array}{c}\mathsf{validType}(\Gamma, \mathit{SimpleType}) \quad \mathcal{LV}(\mathit{Identifier}) = \bot \\ \Gamma, \varsigma, \mathcal{LV}_1 \dagger [\texttt{Identifier} \mapsto (\mathit{SimpleType}\,\texttt{[]}, \mathsf{false})], \mathcal{E}_1, \mathcal{P} \vdash \mathit{ArrayInitializer} : \\ \sigma, \mathcal{LV}_2, \mathcal{E}_2 \\ \Gamma \vdash \sigma \sqsubseteq_{impl} \mathit{SimpleType}\end{array}}{\begin{array}{l}\Gamma, \varsigma, \mathcal{LV}_1, \mathcal{E}_1, (\mathsf{false}, b), \mathcal{P} \vdash \mathit{SimpleType}\,\texttt{[]}\ \mathit{Identifier}\ \texttt{=}\ \mathit{ArrayInitializer}\ \texttt{;} : \\ \qquad \mathcal{LV}_2 \dagger [\mathit{Identifier} \mapsto (\mathit{SimpleType}\,\texttt{[]}, \mathsf{true})], \mathcal{E}_2, (\mathsf{false}, b)\end{array}}$$

Figure 28: Typing rules for local variable declarations.

as long as its initialization is not completed. Below, we introduce an example to illustrate this idea:

```
int m(int p) { return  p;}
int v = m(v = 3) ;
```

The compilation of this example passes without errors. However, the variable v is considered initialized only after the evaluation of the statement `int v = m(v = 3)`. Thus, we evaluate the expression *Expression* under the current map of local variables overridden by the association $[\texttt{Identifier} \mapsto (\mathit{Type}, \mathsf{false})]$ or $[\texttt{Identifier} \mapsto (\mathit{SimpleType}\,\texttt{[]}, \mathsf{false})]$ according to whether we type-check a simple local variable declaration or an array declaration, respectively.

Typing rules for statements

Typing rules for statements are given in Figures 29, 30 and 31. Some explanations concerning these rules are given below:

- Expr-Stmt: An expression statement is evaluated by evaluating the corresponding expression. If the expression has a value, then we discard its type.

- If-Then: The execution of the `then`-statement depends on the value of the conditional expression *Expression*. Thus, we do not know, at compile-time, if this statement will be executed, and then if the static context must be really modified after the evaluation of the `if`-`then` statement. On the other hand, if the conditional expression is the literal `true`, we can be sure that the `then`-statement will be executed. In this case, we can be sure that if a local variable has been initialized in the `then`-statement, then it considered initialized. In the typing rule, the map of local variables yielded by the evaluation of the `if`-`then` statement is computed by the following function:

  $$\begin{array}{lcl} \mathsf{getLocalVarMap}_1 & : & \mathit{Expression} \times \mathit{LocalVarMap} \times \mathit{LocalVarMap} \rightarrow \\ & & \mathit{LocalVarMap} \end{array}$$

  $$\begin{array}{lcll} \mathsf{getLocalVarMap}_1(e, m_1, m_2) & = & \mathsf{if} & e = \texttt{true} \\ & & \mathsf{then} & m_2 \\ & & \mathsf{else} & m_1 \\ & & \mathsf{endif} & \end{array}$$

  On the other hand, according to the Java Language Specification [9], even if the conditional expression is the literal `false`, the `then`-statement is not considered unreachable. So, the flag variable $\mathcal{B}_1$ must remain unchanged after the evaluation of the `if`-`then` statement.

- If-Then-Else: After the evaluation of the `if`-`then`-`else` statement, only the modifications made in both `then` and `else` statements really affect the static context. So, in the typing rule, the flag variable is computed by $\mathcal{B}_2 \wedge \mathcal{B}_3$. A local variable is considered initialized after the evaluation of the `if`-`then`-`else` statement, in the three following cases:

  - It is initialized in both `then` and `else` statements.
  - It is initialized in the `then`-statement and the conditional expression is the literal `true`.
  - It is initialized in the `else`-statement and the conditional expression is the literal `false`.

  The map of local variables yielded by the evaluation of the `if`-`then`-`else` statement is computed by the following function:

  $$\begin{array}{lcl} \mathsf{getLocalVarMap}_2 & : & \mathit{Expression} \times \mathit{LocalVarMap} \times \mathit{LocalVarMap} \rightarrow \\ & & \mathit{LocalVarMap} \end{array}$$

  $$\begin{array}{lcll} \mathsf{getLocalVarMap}_2(e, m_1, m_2) & = & \mathsf{if} & e = \texttt{true} \end{array}$$

```
then  $m_1$
else  if    $e = \texttt{false}$
      then  $m_2$
      else  $m_1 \wedge m_2$
      endif
endif
```

- WHILE: The `while`-statement is considered unreachable when the conditional expression is the literal `false`. This constraint is expressed in the typing rules by the logical formulas *Expression* $\neq$ `false`. On the other hand, if the conditional expression is the literal `true`, then the program cannot complete normally. In this case, after the evaluation of the `while` statement, the boolean variable that indicates the abnormal termination of the program must have the true value. Otherwise, if we do not know the value of the conditional expression, then we cannot be sure that the `while`-statement will be executed, and then we cannot change the static context after the evaluation of this statement.

- SYNCHRONIZED: In this rule, we must simply check that the type of *Expression* is a reference type ρ.

- THROW: Each checked exception thrown by a `throw` statement, has to be either caught in a `try-catch` statement or declared in the `throws` clause of the current method, i.e. the one containing the `throw` statement. To be able to handle exceptions, we add each thrown checked exception to the exception set $\mathcal{E}$. In the typing rule, the latter is updated by $\mathcal{E}_2 \cup \{\!|\varsigma|\!\}$. Since the execution of the `throw` statement cannot complete normally, the boolean variable that indicates the abrupt termination of the current program must have the true value.

- RETURN: A `return` statement with an *Expression* must appear only in the body of a method that declares a return type. Moreover, the type of the *Expression* must be assignable to this declared return type. On the other hand, a `return` statement with no *Expression* must appear only in a constructor body or in a body of a method declared `void`. The function that finds the result type of a method or a constructor is defined as follows:

$$\mathsf{getResultType} \;:\; \mathit{Environment} \times \mathit{ClassType} \times \mathit{Position} \rightarrow \mathit{ResultType}$$

```
getResultType($\Gamma, \varsigma, \mathcal{P}$)  =  if    fst($\mathcal{P}$) = 'm'
                            then  $\Gamma.\mathit{classMap}(\varsigma).\mathit{methods}(\mathsf{snd}(\mathcal{P})).\mathit{resultType}$
                            else  if    fst($\mathcal{P}$) = 'c'
                                  then  void
                                  endif
                            endif
```

- Try-Catches: When an exception is thrown in a `try` block having one or more `catch` clauses, the first one capable of catching this exception is executed. A `catch` clause can catch an exception if it has a parameter that is either of the same type as this exception or one of its supertypes. To type-check the `catch` clauses (*Catches*), we introduce a new kind of judgment $\Gamma, \varsigma, \mathcal{LV}_1, \mathcal{E}_1, \mathcal{UE}_1, \mathcal{B}_1 \vdash Catches : \mathcal{LV}_2, \mathcal{E}_2, \mathcal{UE}_2, \mathcal{B}_2$. The multiset $\mathcal{UE}_1$ contains all checked exceptions raised in the `try` block, while $\mathcal{UE}_2$ contains just the ones among them that have not been caught by any clause `catch`. In the typing rule, the multiset $\mathcal{UE}_1$ is computed by $\mathcal{E}_2 - \mathcal{E}_1$. If we use a simple set for exceptions, we cannot compute correctly the set of exceptions raised in the block `try` when some of them have also been raised before the block. For instance, if the initial exception set $\mathcal{E}_1$ contains the checked exception types `classNotFoundException` and `IllegalAccessException`, and the exception `classNotFoundException` is also thrown in the block `try`, then the new exceptions set $\mathcal{E}_2$ yielded by the evaluation of the block will be the same as $\mathcal{E}_1$. So, $\mathcal{E}_2 - \mathcal{E}_1$ will yield an empty set which is incorrect. Hence, we use multiset. The multiset of exceptions yielded by the evaluation of the `try` statement must contain all the exceptions raised before this statement and those thrown in it and not yet handled. Finally, a local variable is considered to be initialized after the evaluation of the `try` statement only when it is initialized in the `try` block and in all the blocks of the `catch` clauses.

- Try-Catches-Finally: This rule is quite similar to the previous one, except that here there exists a `finally` block that is always executed. When the execution of the `finally` block completes abruptly, none of the exceptions thrown in the `try` statement need to be handled, since these exceptions cannot propagate out of the statement. Thus, we call the function remainingExceptions, defined below, which returns the multiset of exceptions that must be handled.

$$\mathsf{remainingExceptions} \; : \; \mathsf{bool} \times \mathit{ExceptSet} \longrightarrow \mathit{ExceptSet}$$

```
remainingExceptions($b, \mathcal{E}_1, \mathcal{E}_2$)  =  if    $b = \mathsf{true}$
                                                then  $\mathcal{E}_1 - \mathcal{E}_2$
                                                else  $\mathcal{E}_1$
                                                end if
```

The `try`-`finally` statement completes abruptly if either the block `try` completes abruptly and all the blocks of the `catch` clauses complete abruptly, or the block `finally` completes abruptly. This constraint is expressed as $(\mathcal{B}_1 \wedge \mathcal{B}_2) \vee \mathcal{B}_3$. In the same way, the map of local vairables is updated by $(\mathcal{LV}_2 \wedge \mathcal{LV}_3) \vee \mathcal{LV}_4$.

- Catches: A `catch` clause must have exactly one parameter, called an *exception parameter*, that must be of the class type `Throwable` or one of its subclasses. The name of the exception parameter cannot hide a name of a local variable or a parameter of the current method or constructor $\mathcal{LV}_1(\texttt{Identifier}) = \bot$. The scope of the exception parameter is the block of the `catch` clause. When the latter is executed, the exception parameter is initialized by the exception object handled. For this reason, the boolean variable indicating the initialization of the exception parmeter in the map of lacal variables have the true value. The set of the exceptions not handled by any `catch` clause is computed by the function unCaughtExceptions.

Typing rules for expressions

This section is devoted to the static semantics for expressions. Typing rules for expressions are introduced in Figures 32, 33, 34 and 35 and they use judgments of the form $\Gamma, \mu, \mathcal{LV}_1, \mathcal{E}_1, \mathcal{P} \vdash \mathit{Expression} : \tau_e, \mathcal{LV}_2, \mathcal{E}_2, \mathcal{C}$. We comment below some of these rules.

- Literal: The evaluation of a literal never raises an exception. The function typeOf computes the type of a literal. As this function is simple, we deliberately omit its definition. Note that the type of the `null` literal is Null.
- This: The reserved word `this` can only appear in the body of an instance method, a constructor or in the initialization expression of a non-`static` field. This restriction is verified by the following predicate:

```
licitInvokeThis   :   $\mathit{Environment} \times \mathit{ClassType} \times \mathit{Position} \rightarrow \mathsf{bool}$

licitInvokeThis($\Gamma, \varsigma, \mathcal{P}$)   =
          validClass($\Gamma, \varsigma$) $\wedge$ fst($\mathcal{P}$) $\neq$ ‘a’ $\wedge$
          if     fst($\mathcal{P}$) = ‘m’
          then   static $\notin \Gamma.\mathit{classMap}(\varsigma).\mathit{methods}(\mathsf{snd}(\mathcal{P})).\mathit{modifiers}$
          else   if     fst($\mathcal{P}$) = ‘f’
                 then   static $\notin \Gamma.\mathit{classMap}(\varsigma).\mathit{fields}(\mathsf{snd}(\mathcal{P})).\mathit{modifiers}$
                 else   true
                 endif
          endif
```

- New-Object: Since `abstract` methods are incomplete, we cannot create an instance of an `abstract` class. The argument specified in the class instance creation expression is used to invoke a constructor that is declared in the body of the class. The function maxSpecConstructor, defined below, returns the *maximally specific* [9, section 15.1] constructor for the constructor invocation.

```
maxSpecConstructors   :   $\mathit{Environment} \times \mathit{ClassType} \times \mathit{ArgumentType} \rightarrow$
                          $(\mathit{ConstructorInfo} \times \mathit{ClassType}) - \mathsf{set}$

maxSpecConstructors($\Gamma, \varsigma, \tau_a$)   =
```

$\boxed{\textsc{Empty}}$ .... $\boxed{\Gamma, \varsigma, \mathcal{LV}, \mathcal{E}, \mathcal{P}, \mathcal{B} \vdash \textit{Statement} : \mathcal{LV}, \mathcal{E}, \mathcal{B}}$

$$\frac{\square}{\Gamma, \varsigma, \mathcal{LV}, \mathcal{E}, (\mathsf{false}, b), \mathcal{P} \vdash \texttt{;} : \mathcal{LV}, \mathcal{E}, (\mathsf{false}, b)}$$

$\boxed{\textsc{Expr-Stmt}}$ .... $\boxed{\Gamma, \varsigma, \mathcal{LV}_1, \mathcal{E}_1, \mathcal{B}_1, \mathcal{P} \vdash \textit{ExpressionStatement} : \mathcal{LV}_2, \mathcal{E}_2, \mathcal{B}_2}$

$$\frac{\Gamma, \varsigma, \mathcal{LV}_1, \mathcal{E}_1, \mathcal{P} \vdash \textit{StatementExpression} : \tau_e, \mathcal{LV}_2, \mathcal{E}_2, \mathcal{C}}{\Gamma, \varsigma, \mathcal{LV}_1, \mathcal{E}_1, (\mathsf{false}, b), \mathcal{P} \vdash \textit{StatementExpression}\ \texttt{;} : \mathcal{LV}_2, \mathcal{E}_2, (\mathsf{false}, b)}$$

$\boxed{\textsc{If}}$ .... $\boxed{\Gamma, \varsigma, \mathcal{LV}_1, \mathcal{E}_1, \mathcal{B}_1, \mathcal{P} \vdash \textit{IfStatement} : \mathcal{LV}_2, \mathcal{E}_2, \mathcal{B}_2}$

$\boxed{\textsc{If-Then}}$

$$\frac{\begin{array}{c}\mathcal{B}_1 = (\mathsf{false}, b) \quad \Gamma, \varsigma, \mathcal{LV}_1, \mathcal{E}_1, \mathcal{P} \vdash \textit{Expression} : \texttt{boolean}, \mathcal{LV}_2, \mathcal{E}_2, \mathcal{C} \\ \Gamma, \varsigma, \mathcal{LV}_2, \mathcal{E}_2, \mathcal{B}_1, \mathcal{P} \vdash \textit{Statement} : \mathcal{LV}_3, \mathcal{E}_3, \mathcal{B}_2 \\ \mathcal{LV}_4 = \mathsf{getLocalVarMap}_1(\textit{Expression}, \mathcal{LV}_2, \mathcal{LV}_3)\end{array}}{\Gamma, \varsigma, \mathcal{LV}_1, \mathcal{E}_1, \mathcal{B}_1, \mathcal{P} \vdash \texttt{if (}\ \textit{Expression}\ \texttt{)}\ \textit{Statement} : \mathcal{LV}_4, \mathcal{E}_3, \mathcal{B}_1}$$

$\boxed{\textsc{If-Then-Else}}$

$$\frac{\begin{array}{c}\mathcal{B}_1 = (\mathsf{false}, b) \quad \Gamma, \varsigma, \mathcal{LV}_1, \mathcal{E}_1, \mathcal{P} \vdash \textit{Expression} : \texttt{boolean}, \mathcal{LV}_2, \mathcal{E}_2, \mathcal{C} \\ \Gamma, \varsigma, \mathcal{LV}_2, \mathcal{E}_2, \mathcal{B}_1, \mathcal{P} \vdash \textit{Statement}_1 : \mathcal{LV}_3, \mathcal{E}_3, \mathcal{B}_2 \\ \Gamma, \varsigma, \mathcal{LV}_2, \mathcal{E}_2, \mathcal{B}_1, \mathcal{P} \vdash \textit{Statement}_2 : \mathcal{LV}_4, \mathcal{E}_4, \mathcal{B}_3 \\ \mathcal{LV}_5 = \mathsf{getLocalVarMap}_2(\textit{Expression}, \mathcal{LV}_3, \mathcal{LV}_4)\end{array}}{\begin{array}{c}\Gamma, \varsigma, \mathcal{LV}_1, \mathcal{E}_1, \mathcal{B}_1, \mathcal{P} \vdash \texttt{if (}\ \textit{Expression}\ \texttt{)}\ \textit{Statement}_1\ \texttt{else}\ \textit{Statement}_2 : \\ \mathcal{LV}_5, \mathcal{E}_4, \mathcal{B}_2 \wedge \mathcal{B}_3\end{array}}$$

$\boxed{\textsc{While}}$ .... $\boxed{\Gamma, \varsigma, \mathcal{LV}_1, \mathcal{E}_1, \mathcal{B}_1, \mathcal{P} \vdash \textit{WhileStatement} : \mathcal{LV}_2, \mathcal{B}_2, \mathcal{E}_2}$

$$\frac{\begin{array}{c}\mathcal{B}_1 = (\mathsf{false}, b) \quad \textit{Expression} \neq \texttt{false} \quad \textit{Expression} \neq \texttt{true} \\ \Gamma, \varsigma, \mathcal{LV}_1, \mathcal{E}_1, \mathcal{P} \vdash \textit{Expression} : \texttt{boolean}, \mathcal{LV}_2, \mathcal{E}_2, \mathcal{C} \\ \Gamma, \varsigma, \mathcal{LV}_2, \mathcal{E}_2, \mathcal{B}_1, \mathcal{P} \vdash \textit{Statement} : \mathcal{LV}_3, \mathcal{E}_3, \mathcal{B}_2 \\ \mathcal{LV}_4 = \mathsf{getLocalVarMap}_1(\textit{Expression}, \mathcal{LV}_2, \mathcal{LV}_3)\end{array}}{\Gamma, \varsigma, \mathcal{LV}_1, \mathcal{E}_1, \mathcal{B}_1, \mathcal{P} \vdash \texttt{while (}\ \textit{Expression}\ \texttt{)}\ \textit{Statement} : \mathcal{LV}_4, \mathcal{E}_3, \mathcal{B}_1}$$

$$\frac{\begin{array}{c}\mathcal{B}_1 = (\mathsf{false}, b) \quad \textit{Expression} \neq \texttt{false} \\ \Gamma, \varsigma, \mathcal{LV}_1, \mathcal{E}_1, \mathcal{P} \vdash \textit{Expression} : \texttt{boolean}, \mathcal{LV}_2, \mathcal{E}_2, \mathcal{C} \\ \Gamma, \varsigma, \mathcal{LV}_2, \mathcal{E}_2, \mathcal{B}_1, \mathcal{P} \vdash \textit{Statement} : \mathcal{LV}_3, \mathcal{E}_3, \mathcal{B}_2 \\ \mathcal{LV}_4 = \mathsf{getLocalVarMap}_1(\textit{Expression}, \mathcal{LV}_2, \mathcal{LV}_3)\end{array}}{\Gamma, \varsigma, \mathcal{LV}_1, \mathcal{E}_1, \mathcal{B}_1, \mathcal{P} \vdash \texttt{while (true)}\ \textit{Statement} : \mathcal{LV}_4, \mathcal{E}_3, (\mathsf{true}, b)}$$

Figure 29: Typing rules for statements: part 1.

$\boxed{\textsc{Synchronized}}$ .... $\boxed{\mathit{\Gamma}, \varsigma, \mathcal{LV}_1, \mathcal{E}_1, \mathcal{B}_1, \mathcal{P} \vdash \mathit{Synchronized} : \mathcal{LV}_2, \mathcal{E}_2, \mathcal{B}_2}$

$$\frac{\begin{array}{c}\mathcal{B}_1 = (\mathsf{false}, b) \quad \mathit{\Gamma}, \varsigma, \mathcal{LV}_1, \mathcal{E}_1, \mathcal{P} \vdash \mathit{Expression} : \rho, \mathcal{LV}_2, \mathcal{E}_2, \mathcal{C} \\ \mathit{\Gamma}, \varsigma, \mathcal{LV}_2, \mathcal{E}_2, \mathcal{B}_1, \mathcal{P} \vdash \mathit{Block} : \mathcal{E}_3, \mathcal{LV}_3, \mathcal{B}_2\end{array}}{\mathit{\Gamma}, \varsigma, \mathcal{LV}_1, \mathcal{E}_1, \mathcal{B}_1, \mathcal{P} \vdash \texttt{synchronized (} \mathit{Expression} \texttt{) } \mathit{Block} : \mathcal{LV}_3, \mathcal{E}_3, \mathcal{B}_2}$$

$\boxed{\textsc{Throw}}$ .... $\boxed{\mathit{\Gamma}, \varsigma, \mathcal{LV}_1, \mathcal{E}_1, \mathcal{B}_1, \mathcal{P} \vdash \mathit{ThrowStatement} : \mathcal{LV}_2, \mathcal{E}_2, \mathcal{B}_2}$

$$\frac{\mathit{\Gamma}, \varsigma, \mathcal{LV}_1, \mathcal{E}_1, \mathcal{P} \vdash \mathit{Expression} : \varsigma, \mathcal{LV}_2, \mathcal{E}_2, \mathcal{C} \quad \mathit{\Gamma} \vdash \varsigma \sqsubseteq_{class} \texttt{Error}}{\mathit{\Gamma}, \varsigma, \mathcal{LV}_1, \mathcal{E}_1, (\mathsf{false}, b), \mathcal{P} \vdash \texttt{throw } \mathit{Expression} \texttt{ ;} : \mathcal{LV}_2, \mathcal{E}_2, (\mathsf{true}, b)}$$

$$\frac{\mathit{\Gamma}, \varsigma, \mathcal{LV}_1, \mathcal{E}_1, \mathcal{P} \vdash \mathit{Expression} : \varsigma, \mathcal{LV}_2, \mathcal{E}_2, \mathcal{C} \quad \mathit{\Gamma} \vdash \varsigma \sqsubseteq_{class} \texttt{RuntimeException}}{\mathit{\Gamma}, \varsigma, \mathcal{LV}_1, \mathcal{E}_1, (\mathsf{false}, b), \mathcal{P} \vdash \texttt{throw } \mathit{Expression} \texttt{ ;} : \mathcal{LV}_2, \mathcal{E}_2, (\mathsf{true}, b)}$$

$$\frac{\begin{array}{c}\mathit{\Gamma}, \varsigma, \mathcal{LV}_1, \mathcal{E}_1, \mathcal{P} \vdash \mathit{Expression} : \varsigma, \mathcal{LV}_2, \mathcal{E}_2, \mathcal{C} \\ \mathsf{checkedException}(\mathit{\Gamma}, \varsigma)\end{array}}{\mathit{\Gamma}, \varsigma, \mathcal{LV}_1, \mathcal{E}_1, (\mathsf{false}, b), \mathcal{P} \vdash \texttt{throw } \mathit{Expression} \texttt{ ;} : \mathcal{LV}_2, \mathcal{E}_2 \cup \{\!|\varsigma|\!\}, (\mathsf{true}, b)}$$

$\boxed{\textsc{Return}}$ .... $\boxed{\mathit{\Gamma}, \varsigma, \mathcal{LV}_1, \mathcal{E}_1, \mathcal{B}_1, \mathcal{P} \vdash \mathit{ReturnStatement} : \mathcal{LV}_2, \mathcal{E}_2, \mathcal{B}_2}$

$$\frac{\begin{array}{c}\mathsf{getResultType}(\mathit{\Gamma}, \varsigma, \mathcal{P}) \neq \texttt{void} \\ \mathit{\Gamma}, \varsigma, \mathcal{LV}_1, \mathcal{E}_1, \mathcal{P}_1 \vdash \mathit{Expression} : \tau, \mathcal{LV}_2, \mathcal{E}_2, \mathcal{C} \quad \tau \sqsubseteq_{impl} \mathsf{getResultType}(\mathit{\Gamma}, \varsigma, \mathcal{P})\end{array}}{\mathit{\Gamma}, \varsigma, \mathcal{LV}_1, \mathcal{E}_1, (\mathsf{false}, b), \mathcal{P} \vdash \texttt{return } \mathit{Expression} \texttt{ ;} : \mathcal{LV}_2, \mathcal{E}_2, (\mathsf{true}, \mathsf{true})}$$

$$\frac{\mathsf{getResultType}(\mathit{\Gamma}, \varsigma, \mathcal{P}) = \texttt{void}}{\mathit{\Gamma}, \varsigma, \mathcal{LV}_1, \mathcal{E}_1, (\mathsf{false}, b), \mathcal{P} \vdash \texttt{return ;} : \mathcal{LV}_2, \mathcal{E}_2, (\mathsf{true}, \mathsf{true})}$$

Figure 30: Typing rules for statements: part 2.

$$\boxed{\textsc{Try}} \quad \ldots\ldots\ldots\ldots\ldots\ldots\ldots\ldots \quad \boxed{\mathit{\Gamma}, \varsigma, \mathcal{LV}_1, \mathcal{E}_1, \mathcal{B}_1, \mathcal{P} \vdash \mathit{TryStatement} : \mathcal{LV}_2, \mathcal{E}_2, \mathcal{B}_2}$$

$$\boxed{\textsc{Try-Catches}}$$

$$\frac{\begin{array}{c}\mathit{\Gamma}, \varsigma, \mathcal{LV}_1, \mathcal{E}_1, (\mathsf{false}, b), \mathcal{P} \vdash \mathit{Block} : \mathcal{LV}_2, \mathcal{E}_2, \mathcal{B}_1 \\ \mathit{\Gamma}, \varsigma, \mathcal{LV}_1, \mathcal{E}_1, \mathcal{E}_2 - \mathcal{E}_1, (\mathsf{false}, b), \mathcal{P} \vdash \mathit{Catches} : \mathcal{LV}_3, \mathcal{E}_3, \mathcal{UE}, \mathcal{B}_2\end{array}}{\mathit{\Gamma}, \varsigma, \mathcal{LV}_1, \mathcal{E}_1, (\mathsf{false}, b), \mathcal{P} \vdash \texttt{try}\ \mathit{Block}\ \mathit{Catches} : \mathcal{LV}_2 \wedge \mathcal{LV}_3, \mathcal{E}_3 \cup \mathcal{UE}, \mathcal{B}_1 \wedge \mathcal{B}_2}$$

$$\boxed{\textsc{Try-Catches-Finally}}$$

$$\frac{\begin{array}{c}\mathit{\Gamma}, \varsigma, \mathcal{LV}_1, \mathcal{E}_1, (\mathsf{false}, b), \mathcal{P} \vdash \mathit{Block}_1 : \mathcal{LV}_2, \mathcal{E}_2, \mathcal{B}_1 \\ \mathit{\Gamma}, \varsigma, \mathcal{LV}_1, \mathcal{E}_1, \mathcal{E}_2 - \mathcal{E}_1, (\mathsf{false}, b), \mathcal{P} \vdash \mathit{Catches} : \mathcal{LV}_3, \mathcal{E}_3, \mathcal{UE}, \mathcal{B}_2 \\ \mathit{\Gamma}, \varsigma, \mathcal{LV}_1, \mathcal{E}_3 \cup \mathcal{UE}, (\mathsf{false}, b) \vdash \mathit{Block}_2 : \mathcal{LV}_4, \mathcal{E}_4, \mathcal{B}_3 \\ \mathcal{E}_5 = \mathsf{remainingExceptions}(\mathsf{fst}(\mathcal{B}_3), \mathcal{E}_4, (\mathcal{E}_3 \cup \mathcal{UE}) - \mathcal{E}_1)\end{array}}{\begin{array}{r}\mathit{\Gamma}, \varsigma, \mathcal{LV}_1, \mathcal{E}_1, (\mathsf{false}, b), \mathcal{P} \vdash \texttt{try}\ \mathit{Block}_1\ \mathit{Catches}\ \texttt{finally}\ \mathit{Block}_2 : \\ (\mathcal{LV}_2 \wedge \mathcal{LV}_3) \vee \mathcal{LV}_4, \mathcal{E}_5, (\mathcal{B}_1 \wedge \mathcal{B}_2) \vee \mathcal{B}_3\end{array}}$$

$$\boxed{\textsc{Try-Finally}}$$

$$\frac{\begin{array}{c}\mathit{\Gamma}, \varsigma, \mathcal{LV}_1, \mathcal{E}_1, (\mathsf{false}, b), \mathcal{P} \vdash \mathit{Block}_1 : \mathcal{LV}_2, \mathcal{E}_2, \mathcal{B}_1 \\ \mathit{\Gamma}, \varsigma, \mathcal{LV}_1, \mathcal{E}_2, (\mathsf{false}, b), \mathcal{P}, \vdash \mathit{Block}_2 : \mathcal{LV}_3, \mathcal{E}_3, \mathcal{B}_2 \\ \mathcal{E}_4 = \mathsf{remainingExceptions}(\mathsf{fst}(\mathcal{B}_2), \mathcal{E}_3, \mathcal{E}_2 - \mathcal{E}_1)\end{array}}{\mathit{\Gamma}, \varsigma, \mathcal{LV}_1, \mathcal{E}_1, (\mathsf{false}, b), \mathcal{P} \vdash \texttt{try}\ \mathit{Block}_1\ \texttt{finally}\ \mathit{Block}_2 : \mathcal{LV}_3 \vee \mathcal{LV}_4, \mathcal{E}_2, \mathcal{B}_1 \vee \mathcal{B}_2}$$

$$\boxed{\textsc{Catches}}$$

$$\frac{\begin{array}{c}\mathit{\Gamma}, \varsigma, \mathcal{LV}_1, \mathcal{E}_1, \mathcal{UE}_1, \mathcal{B}_1, \mathcal{P} \vdash \mathit{CatchClause} : \mathcal{LV}_2, \mathcal{E}_2, \mathcal{UE}_2, \mathcal{B}_2 \\ \mathit{\Gamma}, \varsigma, \mathcal{LV}_1, \mathcal{E}_2, \mathcal{UE}_2, \mathcal{B}_1, \mathcal{P} \vdash \mathit{Catches} : \mathcal{LV}_3, \mathcal{E}_3, \mathcal{UE}_3, \mathcal{B}_3\end{array}}{\mathit{\Gamma}, \varsigma, \mathcal{LV}_1, \mathcal{E}_1, \mathcal{UE}_1, \mathcal{B}_1, \mathcal{P} \vdash \mathit{CatchClause}\ \mathit{Catches} : \mathcal{LV}_2 \wedge \mathcal{LV}_3, \mathcal{E}_3, \mathcal{UE}_3, \mathcal{B}_2 \wedge \mathcal{B}_3}$$

$$\frac{\begin{array}{c}\mathit{\Gamma} \vdash \mathit{ClassType} \sqsubseteq_{class} \texttt{Throwable} \\ \mathcal{LV}_1(\mathit{Identifier}) = \bot \\ \mathit{\Gamma}, \varsigma, \mathcal{LV}_1 \dagger [\mathit{Identifier} \mapsto (\mathit{ClassType}, \mathsf{true})], \mathcal{E}_1, \mathcal{B}_1, \mathcal{P} \vdash \mathit{Block} : \\ \mathcal{LV}_2 \dagger [\texttt{Idntifier} \mapsto (\mathit{ClassType}, \mathsf{true})], \mathcal{E}_2, \mathcal{B}_2 \\ \mathsf{unCaughtExceptions}(\mathit{\Gamma}, \mathcal{UE}_1, \mathit{ClassType}) = \mathcal{UE}_2\end{array}}{\begin{array}{c}\mathit{\Gamma}, \varsigma, \mathcal{LV}_1, \mathcal{E}_1, \mathcal{UE}_1, \mathcal{B}_1, \mathcal{P} \vdash \texttt{catch (}\ \mathit{ClassType}\ \mathit{Identifier}\ \texttt{)}\ \mathit{Block} : \\ \mathcal{LV}_2, \mathcal{E}_2, \mathcal{UE}_2, \mathcal{B}_2\end{array}}$$

Figure 31: Typing rules for statements: part 3.

$$
\begin{array}{l}
\{ (\langle ms', tcs'\rangle, \varsigma') \mid ((\langle ms', tcs'\rangle, \varsigma'), \tau'_a) \in \mathsf{applConstructors}(\Gamma, \varsigma, \tau_a) \quad \wedge \\
\qquad \forall\ ((\langle ms'', tcs''\rangle, \varsigma'), \tau''_a) \in \mathsf{applConstructors}(\Gamma, \varsigma, \tau_a). \\
\qquad\quad \mathsf{if} \quad \Gamma \vdash \tau''_a \sqsubseteq_{impl} \tau'_a \\
\qquad\quad \mathsf{then} \quad (\langle ms', tcs'\rangle, \varsigma') = (\langle ms'', tcs''\rangle, \varsigma') \\
\qquad\quad \mathsf{endif}\ \}
\end{array}
$$

This function must find exactly one constructor. The declaring class of the latter is named ς. Since, `private` constructors can only be invoked in their declaring class, we add the logical formula `private` $\in ms \Rightarrow \mu = \varsigma$ in the typing rule. when the invloked constructor declares some checked exceptions, we add them to the multiset of the thrown exceptions. Thus, we can handle them later. This is done by the following function:

$$\mathsf{throwsExceptions} \quad : \quad \mathit{Environment} \times (\mathit{ClassType}) - \mathsf{set} \rightarrow \mathit{ExceptSet}$$

$$
\begin{array}{lcl}
\mathsf{throwsExceptions}(\Gamma, \emptyset, \mathcal{E}) & = & \mathcal{E} \\
\mathsf{throwsExceptions}(\Gamma, \{\varsigma\} \cup tcs, \mathcal{E}) & = & (\mathsf{if} \quad \mathsf{checkedException}(\Gamma, \varsigma) \\
 & & \ \mathsf{then} \quad \{\!|\varsigma|\!\} \\
 & & \ \mathsf{else} \quad \emptyset \\
 & & \ \mathsf{endif}) \cup \\
 & & \ \mathsf{throwsExceptions}(\Gamma, tcs, \mathcal{E})
\end{array}
$$

- SIMPLE-FIELD-ACCESS: The fucntion fieldsVar must find exactly one accessible field for the filed access. As for the previous rule, if the invoked field is `private`, then the class that contains its invocation must be its declaring class. If the invoked filed is `final`, then the variable $\mathcal{C}$ receives the true value. When the field access expression is `super.`*Identifier*, we have to check, that the reserved word `super` is used correctly. This is done by the following function:

$$\mathsf{licitInvokeSuper} \quad : \quad \mathit{Environment} \times \mathit{ClassOrIfaceType} \times \mathit{Position} \rightarrow \mathsf{bool}$$

```
licitInvokeSuper($\Gamma, \mu, \mathcal{P}$)   =
    validClass($\Gamma, \mu$) $\wedge$ $\mu \neq$ Object $\wedge$ fst($\mathcal{P}$) $\neq$ 'a' $\wedge$
    if     fst($\mathcal{P}$) = 'm'
    then   static $\notin \Gamma.classMap(\mu).methods($snd$(\mathcal{P})).modifiers$
    else   if     fst($\mathcal{P}$) = 'f'
           then   static $\notin \Gamma.classMap(\mu).fields($snd$(\mathcal{P}).modifiers$
           else   true
           endif
    endif
```

- SIMPLE-NAME-FIELD-ACCESS: These rules are used to type-check a field access expression that is a single identifier or several identifiers separated by a "." token. These rules are quite similar to previous ones, except for the first rule. When the expression is a simple identifier, no local variable or parameter can have the same name as the field, since local variables and parameters hide fields with same names in their scopes. This is expressed as

$\mathcal{LV}(\mathit{Identifier}) = \bot$. Moreover, since we lookup for the invoked field in the map of fields accessible from the current class or interface μ, we do not need to check that `private` fields must only be invoked in their declaring classes. The predicate licitAppear defined below checks that a non-`static` field cannot be invoked in a `static` method or field:

$$\begin{aligned}\mathsf{licitAppear} \quad : \quad & \mathit{Environment} \times \mathit{ClassOrIfaceType} \times \\ & (\mathit{ModifierName}) - \mathsf{set} \times \mathit{Position} \rightarrow \mathsf{bool}\end{aligned}$$

```
licitAppear($\Gamma, \mu, ms, \mathcal{P}$) =
     if     $\texttt{static} \notin ms$
     then   ($\mathsf{fst}(\mathcal{P}) =$ ‘m’ $\wedge$
             $\texttt{static} \notin \Gamma.classMap(\mu).methods(\mathsf{snd}(\mathcal{P})).modifiers$ ) $\vee$
            $\mathsf{fst}(\mathcal{P}) =$ ‘c’ $\vee$
            ($\mathsf{fst}(\mathcal{P}) =$ ‘f’ $\wedge$ validClass($\Gamma, \mu$) $\wedge$
             $\texttt{static} \notin \Gamma.classMap(\mu).fields(\mathsf{snd}(\mathcal{P})).modifiers$)
     else   true
     endif
```

If the field access expression that is a simple identifier is used in a field initialization expression, then we have to ensure that there is no circularity [9, section 8.3.2] . This is checked by the following predicate:

$$\begin{aligned}\mathsf{isDeclared} \quad : \quad & \mathit{Environment} \times \mathit{ClassOrIfaceType} \times \mathit{Identifier} \times \\ & (\mathit{ModifierName}) - \mathsf{set} \times \mathit{Position} \rightarrow \mathsf{bool}\end{aligned}$$

```
isDeclared($\Gamma, \mu, fn, ms, \mathcal{P}$) =
     if     $\mathsf{fst}(\mathcal{P}) =$ ‘f’
     then   if     ($\texttt{static} \in ms \leftrightarrow$
                    $\texttt{static} \in$ classOrIfaceDecl($\Gamma, \mu$)$.fields(\mathsf{snd}(\mathcal{P})).modifiers$)
            then textuallyBefore($\Gamma, \mu, fn, \mathsf{fst}(\mathcal{P})$)
            else  true
            endif
     else   true
     endif
```

We assume that the fields are sorted by their declaration order in the map. Thus, the predicate textuallyBefore can verify that a field declaration occurs to the left of, i.e. textually before, another field declaration in the same class.

- SIMPLE-VAR-ACCESS: The accessed variable is looked-up in the map of the local variables $\mathcal{LV}$. It must exist and must have an initial value.
- ARRAY-FIELD-ACCESS: This rule is simple. The type of the field access expression is the one of the array elements. The index of the array, i.e. the expression that appears between the brackets, must be of the `int` type or of a type that can be widened to this type. Note that an array elements cannot be `final` even if the array is.

- ARRAY-VAR-ACCESS: This rule is very similar to the previous one, except that the type of the array access is looked-up in the map of the local variables. The array access is type-correct only if the array is found and it was initialized.

- METHOD-NAME: These rules use another kind of judgment, different from the one used in the typing rules for expressions, since a method name is not an expression. They elaborate to an identifier (the method name), a class or an interface type from which we must look for the method declaration and a character (*whichName*). This one has the 's' value if the method invocation expression is a simple identifier, the 'c' value if it is a *ClassType*. *Identifier* and 'e' if it is an *ExpressionName*. *Identifier*.

- METHOD-INVOCATION: These rules type-check a method call. When the function maxSpecMethods, determining the set of *maximally specific* methods for a method invocation [9, 15.11.2], returns several methods with the same signaute, we call the function unAmbiguousMaxSpec to check that they have the same signature. The function applAccessMethods finds methods that are applicable and accessible to the method invocation. All these functions are defined as following:

$$\begin{aligned}
\mathsf{unAmbiguousMaxSpec} \;&:\; \mathit{Environment} \times \mathit{ReferenceType} \times \mathit{Sig} \times \\
&\quad \mathit{ClassOrIfaceType} \rightarrow \\
&\quad (\mathit{MethodInfo} \times \mathit{RefernceType}) - \mathsf{set}
\end{aligned}$$

$$\begin{aligned}
\mathsf{maxSpecMethods} \;&:\; \mathit{Environment} \times \mathit{ReferenceType} \times \mathit{Sig} \times \mathit{ClassOrIfaceType} \\
&\rightarrow ((\mathit{MethodInfo} \times \mathit{RefernceType}) \times \mathit{ArgumentType}) - \mathsf{set}
\end{aligned}$$

$$\begin{aligned}
\mathsf{applAccessMethods} \;&:\; \mathit{Environment} \times \mathit{ReferenceType} \times \mathit{Sig} \times \mathit{Sig} \times \\
&\quad \mathit{ClassOrIfaceType} \rightarrow \\
&\quad ((\mathit{MethodInfo} \times \mathit{RefernceType}) \times \mathit{ArgumentType}) - \mathsf{set}
\end{aligned}$$

$$\begin{aligned}
&\mathsf{unAmbiguousMaxSpec}(\Gamma, \rho, sig, \mu) \quad = \\
&\qquad \{(m', \rho') \mid ((m', \rho'), \tau'_a) \in \mathsf{maxSpecMethods}(\Gamma, \rho, sig, \mu) \quad \wedge \\
&\qquad\qquad \forall \{(m'', \rho''), \tau''_a)\} \in \mathsf{maxSpecMethods}(\Gamma, \rho, sig, \mu).\ \tau'_a = \tau''_a
\end{aligned}$$

$$\begin{aligned}
&\mathsf{maxSpecMethods}(\Gamma, \rho, sig, \mu) \quad = \\
&\qquad \{((m', \rho'), \tau'_a) \mid ((m', \rho'), \tau'_a) \in \mathsf{applAccessMethods}(\Gamma, \rho, sig, \mu) \quad \wedge \\
&\qquad\qquad \forall\ ((m'', \rho''), \tau''_a) \in \mathsf{applAccessMethods}(\Gamma, \rho, sig, \mu). \\
&\qquad\qquad\quad \mathsf{if} \quad \Gamma \vdash \rho'' \sqsubseteq_{impl} \rho' \ \wedge\ \Gamma \vdash \tau''_a \sqsubseteq_{impl} \tau'_a \\
&\qquad\qquad\quad \mathsf{then} \ \ ((m', \rho'), \tau'_a) = ((m'', \rho''), \tau''_a) \\
&\qquad\qquad\quad \mathsf{end\ if}
\end{aligned}$$

$$\begin{aligned}
&\mathsf{applAccessMethods}(\Gamma, \rho, (mn, \tau_a), \mu) \quad = \\
&\qquad \{(((\langle ms, rt, tcs\rangle, \rho'), \tau'_a) \mid ((\langle ms, rt, tcs\rangle, \rho') \in \mathsf{methodsVar}(\Gamma, \rho)(mn, \tau'_a) \wedge \\
&\qquad\qquad \Gamma \vdash \tau_a \sqsubseteq_{impl} \tau'_a\} \wedge
\end{aligned}$$

$$(\ \text{if } \texttt{private} \in ms \text{ then } \rho' = \mu\ )\ \}$$

When the function unAmbiguousMaxSpec yields more than one method, we can be sure that the found methods are all `abstract` and have the same result type. This is checked by the well-formed predicates. Thus, the function methodDeclarationInfo, used in the typing rules, looks for the modifiers set and the return type declared in these methods. As for the set of delcared exceptions, the function computes the common subtype of all set of exceptions declared in these methods. This is necessary to handle the exceptions that can be raised by the method invocation. The function methodDeclarationInfo is defined bellow:

$$\begin{array}{lll}
\mathsf{methodDeclInfo} & : & (\mathit{MethodInfo} \times \mathit{ReferenceType}) - \mathsf{set} \rightarrow \mathit{MethodInfo} \\
\mathsf{narrower} & : & (\mathit{ClassType}) - \mathsf{set} \times (\mathit{MethodInfo} \times \mathit{ReferenceType}) - \mathsf{set} \rightarrow \\
 & & (\mathit{ClassType}) - \mathsf{set}
\end{array}$$

$$\begin{array}{lll}
\mathsf{methodDeclInfo}(\{(\langle ms, rt, tcs\rangle, \rho)\}) & = & \langle ms, rt, tcs\rangle \\
\mathsf{methodDeclInfo}(\{(\langle ms, rt, tcs\rangle, \rho)\} \cup s) & = & \langle ms, rt, \mathsf{narrower}(tcs, s)\rangle
\end{array}$$

```
narrower($tcs, \emptyset$)   =   $tcs$
narrower($tcs_1, \{(\langle ms, rt, tcs_2\rangle, \rho)\} \cup s$)   =
    let $ce_1$ = getCheckedExceptions($tcs_1$) ;
        $ce_2$ = getCheckedExceptions($tcs_2$)
    in  if     $\forall e \in ce_1$. subClass($e, ce_2$)
        then   narrower($ce_1, m$)
        else   if     $\forall e \in ce_2$. subClass($e, ce_1$)
               then   narrower($ce_2, s$)
               else   $\emptyset$
               endif
        endif
    endlet
```

- ASSIGNMENT: Typing rules for a field assignment (the two first rules) and local variable assignment (the two second rules) are quite simple. A simple field assignment is type-correct when it is a variable, i.e. it is not a `final` field. Therefore, in this case the boolean variable $\mathcal{C}$ must have the false value. The evaluation of a local variable assignment expression updates the map of local variables to indicate that the variable is henceforth initialized.

$\boxed{\textsc{Argument}}$ .... $\boxed{\mathit{\Gamma}, \varsigma, \mathcal{LV}_1, \mathcal{E}_1, \mathcal{P} \vdash \mathit{Argument} : \tau_a, \mathcal{LV}_2, \mathcal{E}_2, \mathcal{C}}$

$$\frac{\square}{\mathit{\Gamma}, \varsigma, \mathcal{LV}, \mathcal{E}, \mathcal{P} \vdash \varepsilon : \mathsf{Unit}, \mathcal{LV}, \mathcal{E}, \mathsf{false}}$$

$\boxed{\textsc{New-Array}}$ .... $\boxed{\mathit{\Gamma}, \mu, \mathcal{LV}_1, \mathcal{E}_1, \mathcal{P} \vdash \mathit{ArrayCreation} : \alpha, \mathcal{LV}_2, \mathcal{E}_2, \mathcal{C}}$

$$\frac{\mathsf{validType}(\mathit{\Gamma}, \mathit{SimpleType}) \quad \mathit{\Gamma}, \mu, \mathcal{LV}_1, \mathcal{E}_1, \mathcal{P} \vdash \mathit{Expression} : \pi, \mathcal{LV}_2, \mathcal{E}_2, \mathcal{C} \qquad \pi \sqsubseteq_{\mathit{implP}} \texttt{int}}{\mathit{\Gamma}, \mu, \mathcal{LV}_1, \mathcal{E}_1, \mathcal{P} \vdash \texttt{new}\ \mathit{SimpleType}\ \texttt{[}\ \mathit{Expression}\ \texttt{]} : \mathit{SimpleType}\texttt{[ ]}, \mathcal{LV}_2, \mathcal{E}_2, \mathsf{false}}$$

$\boxed{\textsc{Prim-Not-Array}}$ ...... $\boxed{\mathit{\Gamma}, \mu, \mathcal{LV}_1, \mathcal{E}_1, \mathcal{P} \vdash \mathit{PrimaryNoNewArray} : \tau_e, \mathcal{LV}_2, \mathcal{E}_2, \mathcal{C}}$

$\boxed{\textsc{Literal}}$

$$\frac{\mathsf{typeOf}(\texttt{literal}) = \tau_e}{\mathit{\Gamma}, \mu, \mathcal{LV}, \mathcal{E}, \mathcal{P} \vdash \texttt{literal} : \tau_e, \mathcal{LV}, \mathcal{E}, \mathsf{false}}$$

$\boxed{\textsc{This}}$

$$\frac{\mathsf{licitInvokeThis}(\mathit{\Gamma}, \varsigma, \mathcal{P})}{\mathit{\Gamma}, \varsigma, \mathcal{LV}, \mathcal{E}, \mathcal{P} \vdash \texttt{this} : \varsigma, \mathcal{LV}, \mathcal{E}, \mathsf{false}}$$

$\boxed{\textsc{Parentheses}}$

$$\frac{\mathit{\Gamma}, \mu, \mathcal{LV}_1, \mathcal{E}_1, \mathcal{P} \vdash \mathit{Expression} : \tau_e, \mathcal{LV}_2, \mathcal{E}_2, \mathcal{C}}{\mathit{\Gamma}, \mu, \mathcal{LV}_1, \mathcal{E}_1, \mathcal{P} \vdash \texttt{(}\ \mathit{Expression}\ \texttt{)} : \tau_e, \mathcal{LV}_2, \mathcal{E}_2, \mathsf{false}}$$

$\boxed{\textsc{New-Object}}$ .... $\boxed{\mathit{\Gamma}, \mu, \mathcal{LV}_1, \mathcal{E}_1, \mathcal{P} \vdash \mathit{ClassInstanceCreation} : \varsigma, \mathcal{LV}_2, \mathcal{E}_2, \mathcal{C}}$

$$\frac{\begin{array}{c}\mathsf{validClass}(\mathit{\Gamma}, \mathit{ClassType}) \\ \texttt{abstract} \notin \mathit{\Gamma}.\mathit{classMap}(\mathit{ClassType}).\mathit{modifiers} \\ \mathit{\Gamma}, \mu, \mathcal{LV}_1, \mathcal{E}_1, \mathcal{P} \vdash \mathit{Argument} : \tau_a, \mathcal{LV}_2, \mathcal{E}_2, \mathcal{C} \\ \mathsf{maxSpecConstructor}(\mathit{\Gamma}, \mathit{ClassType}, \tau_a) = \{(\langle ms, tcs \rangle, \varsigma)\} \\ \texttt{private} \in ms \Rightarrow \mu = \varsigma \quad \mathsf{throwsExceptions}(\mathit{\Gamma}, tcs, \mathcal{E}_2) = \mathcal{E}_3\end{array}}{\mathit{\Gamma}, \mu, \mathcal{LV}_1, \mathcal{E}_1, \mathcal{P} \vdash \texttt{new}\ \mathit{ClassType}\ \texttt{(}\ \mathit{Argument}\ \texttt{)} : \mathit{ClassType}, \mathcal{LV}_2, \mathcal{E}_3, \mathsf{false}}$$

Figure 32: Typing rules for expressions: part 1.

$$\boxed{\textsc{Simple-Field-Access}} \;\ldots\ldots\ldots\; \boxed{\Gamma, \mu, \mathcal{LV}_1, \mathcal{E}_1, \mathcal{P} \vdash \mathit{SimpleFieldAcess} : \tau, \mathcal{LV}_2, \mathcal{E}_2, \mathcal{C}}$$

$$\frac{\begin{array}{c}\Gamma, \mu, \mathcal{LV}, \mathcal{E}, \mathcal{P} \vdash \mathit{Primary} : \rho_1, \mathcal{LV}_2, \mathcal{E}_2 \\ \mathsf{fieldsVar}(\Gamma, \rho_1)(\mathit{Identifier}) = \{(\langle \tau, ms \rangle, \rho_2)\} \\ \texttt{private} \in ms \Rightarrow \rho_2 = \mu \quad \mathcal{C} = \texttt{final} \in ms\end{array}}{\Gamma, \mu, \mathcal{LV}_1, \mathcal{E}_1, \mathcal{P} \vdash \mathit{Primary}.\mathit{Identifier} : \tau, \mathcal{E}_2, \mathcal{LV}_2, \mathcal{C}}$$

$$\frac{\begin{array}{c}\mathsf{licitInvokeSuper}(\Gamma, \varsigma_1, \mathcal{P}) \quad \Gamma.\mathit{classMap}(\varsigma_1).\mathit{super} = \varsigma_2 \\ \mathsf{fields_c}(\Gamma, \varsigma_2)(\mathit{Identifier}) = \{(\langle \tau, ms \rangle, \rho)\} \\ \texttt{private} \in ms \Rightarrow \rho = \varsigma_1 \quad \mathcal{C} = \texttt{final} \in ms\end{array}}{\Gamma, \varsigma_1, \mathcal{LV}, \mathcal{E}, \mathcal{P} \vdash \texttt{super}.\mathit{Identifier} : \tau, \mathcal{E}, \mathcal{LV}, \mathcal{C}}$$

$$\boxed{\textsc{Simple-Name-Field-Access}} \;\ldots\ldots\ldots\ldots\; \boxed{\Gamma, \mu, \mathcal{LV}, \mathcal{E}, \mathcal{P} \vdash \mathit{FieldName} : \tau, \mathcal{LV}, \mathcal{E}, \mathcal{C}}$$

$$\frac{\begin{array}{c}\mathcal{LV}(\mathit{Identifier}) = \bot \quad \mathsf{fieldsVar}(\Gamma, \mu)(\mathit{Identifier}) = \{(\langle \tau, ms \rangle, \rho)\} \\ \mathcal{C} = \texttt{final} \in ms \quad \mathsf{licitAppear}(\Gamma, \mu, ms, \mathcal{P}) \\ \mathsf{isDeclared}(\Gamma, \mu, \mathit{Identifier}, ms, \mathcal{P})\end{array}}{\Gamma, \mu, \mathcal{LV}, \mathcal{E}, \mathcal{P} \vdash \mathit{Identifier} : \tau, \mathcal{LV}, \mathcal{E}, \mathcal{C}}$$

$$\frac{\begin{array}{c}\mathsf{validClass}(\Gamma, \mathit{ClassOrInterfaceType}) \vee \mathsf{validIface}(\Gamma, \mathit{ClassOrInterfaceType}) \\ \mathsf{fieldsVar}(\Gamma, \mathit{ClassOrInterfaceType})(\mathit{Identifier}) = \{(\langle \tau, ms \rangle, \rho)\} \\ \texttt{static} \in ms \quad \texttt{private} \in ms \Rightarrow \rho = \mu \quad \mathcal{C} = \texttt{final} \in ms\end{array}}{\Gamma, \mu, \mathcal{LV}, \mathcal{E}, \mathcal{P} \vdash \mathit{ClassOrInterfaceType}.\mathit{Identifier} : \tau, \mathcal{LV}, \mathcal{E}, \mathcal{C}}$$

$$\frac{\begin{array}{c}\Gamma, \mu, \mathcal{LV}, \mathcal{E}, \mathcal{P} \vdash \mathit{ExpressionName} : \rho_1, \mathcal{LV}, \mathcal{E}, \mathcal{C}_1 \\ \mathsf{fieldsVar}(\Gamma, \rho_1)(\mathit{Identifier}) = \{(\langle \tau, ms \rangle, \rho_2)\} \\ \texttt{private} \in ms \Rightarrow \rho_2 = \mu \quad \mathcal{C}_2 = \texttt{final} \in ms\end{array}}{\Gamma, \mu, \mathcal{LV}, \mathcal{E}, \mathcal{P} \vdash \mathit{ExpressionName}.\mathit{Identifier} : \tau, \mathcal{LV}, \mathcal{E}, \mathcal{C}_2}$$

$$\boxed{\textsc{Simple-Var-Access}} \;\ldots\ldots\ldots\ldots\ldots\ldots\; \boxed{\Gamma, \mu, \mathcal{LV}, \mathcal{E}, \mathcal{P} \vdash \mathit{LocalVarName} : \tau, \mathcal{LV}, \mathcal{E}, \mathcal{C}}$$

$$\frac{\mathcal{LV}(\mathit{Identifier}) = (\tau, \mathsf{true})}{\Gamma, \mu, \mathcal{LV}, \mathcal{E}, \mathcal{P} \vdash \mathit{Identifier} : \tau, \mathcal{E}, \mathcal{LV}, \mathsf{false}}$$

$$\boxed{\textsc{Array-Field-Access}} \;\ldots\ldots\ldots\ldots\; \boxed{\Gamma, \mu, \mathcal{LV}_1, \mathcal{E}_1, \mathcal{P} \vdash \mathit{ArrayAccess} : \sigma*, \mathcal{LV}_2, \mathcal{E}_2, \mathcal{C}}$$

$$\frac{\begin{array}{c}\Gamma, \mu, \mathcal{LV}_1, \mathcal{E}_1, \mathcal{P} \vdash \mathit{PrimaryNoNewArray} : \sigma * \texttt{[ ]}, \mathcal{LV}_2, \mathcal{E}_2 \\ \Gamma, \mu, \mathcal{LV}_2, \mathcal{E}_2, \mathcal{P} \vdash \mathit{Expression} : \pi, \mathcal{LV}_3, \mathcal{E}_3 \quad \Gamma \vdash \pi \sqsubseteq_{\mathit{implP}} \texttt{int}\end{array}}{\Gamma, \mu, \mathcal{LV}_1, \mathcal{E}_1, \mathcal{P} \vdash \mathit{PrimaryNoNewArray}\ \texttt{[}\ \mathit{Expression}\ \texttt{]} : \sigma*, \mathcal{LV}_3, \mathcal{E}_3, \mathsf{false}}$$

Figure 33: Typing rules for expressions: part 2.

$\boxed{\textsc{Array-Var-Access}}$ .... $\boxed{\mathit{\Gamma}, \mu, \mathcal{LV}_1, \mathcal{E}_1, \mathcal{P} \vdash \mathit{ArrayAccess} : \sigma *, \mathcal{LV}_2, \mathcal{E}_2.\mathcal{C}}$

$$\frac{\begin{array}{c}\mathsf{fst}(\mathcal{LV}_1(\texttt{Identifier})) = \sigma * \texttt{[ ]} \quad \mathsf{snd}(\mathcal{LV}_1(\mathit{Identifier})) = \mathsf{true} \\ \mathit{\Gamma}, \mu, \mathcal{LV}_1, \mathcal{E}_1, \mathcal{P} \vdash \mathit{Expression} : \pi, \mathcal{LV}_2, \mathcal{E}_2 \quad \mathit{\Gamma} \vdash \pi \sqsubseteq_{\mathit{implP}} \texttt{int}\end{array}}{\mathit{\Gamma}, \mu, \mathcal{LV}_1, \mathcal{E}_1, \mathcal{P} \vdash \mathit{Identifier}\ \texttt{[}\ \mathit{Expression}\ \texttt{]} : \sigma *, \mathcal{LV}_2, \mathcal{E}_2, \mathsf{false}}$$

$\boxed{\textsc{Cast}}$ .... $\boxed{\mathit{\Gamma}, \mu, \mathcal{LV}_1, \mathcal{E}_1, \mathcal{P} \vdash \mathit{CastExpression} : \tau, \mathcal{LV}_2, \mathcal{E}_2, \mathcal{C}}$

$$\frac{\mathit{\Gamma}, \mu, \mathcal{LV}_1, \mathcal{E}_1, \mathcal{P} \vdash \mathit{Expression} : \tau_e, \mathcal{LV}_2, \mathcal{E}_2 \quad \mathit{\Gamma} \vdash \tau_e \sqsubseteq_{\mathit{cast}} \mathit{Type}}{\mathit{\Gamma}, \mu, \mathcal{LV}_1, \mathcal{E}_1, \mathcal{P} \vdash \texttt{(}\ \mathit{Type}\ \texttt{)}\ \mathit{Expression} : \mathit{Type}, \mathcal{LV}_2, \mathcal{E}_2, \mathsf{false}}$$

$\boxed{\textsc{Method-Name}}$ ..... $\boxed{\mathit{\Gamma}, \mu, \mathcal{LV}, \mathcal{E}, \mathcal{P} \vdash \mathit{MethodName} : \mathit{Identifier}, \rho, \mathit{wichName}}$

$$\frac{\square}{\mathit{\Gamma}, \mu, \mathcal{LV}, \mathcal{E}, \mathcal{P} \vdash \texttt{Identifier} : \mathit{Identifier}, \mu, \text{‘s’}}$$

$$\frac{\mathsf{validType}(\mathit{\Gamma}, \mathit{ClassType})}{\mathit{\Gamma}, \mu, \mathcal{LV}, \mathcal{E}, \mathcal{P} \vdash \mathit{ClassType.Identifier} : \mathit{Identifier}, \mathit{ClassType}, \text{‘c’}}$$

$$\frac{\mathit{\Gamma}, \mu, \mathcal{LV}, \mathcal{E}, \mathcal{P} \vdash \mathit{ExpressionName} : \rho, \mathcal{E}, \mathcal{LV}, \mathcal{C}}{\mathit{\Gamma}, \mu, \mathcal{LV}, \mathcal{E}, \mathcal{P} \vdash \mathit{ExpressionName.Identifier} : \mathit{Identifier}, \rho, \text{‘e’}}$$

$\boxed{\textsc{Method-Invocation}}$ ... $\boxed{\mathit{\Gamma}, \mu, \mathcal{LV}_1, \mathcal{E}_1, \mathcal{P} \vdash \mathit{MethodInvocation} : \tau_e, \mathcal{LV}_2, \mathcal{E}_2, \mathcal{C}}$

$$\frac{\begin{array}{c}\mathit{\Gamma}, \mu, \mathcal{LV}_1, \mathcal{P} \vdash \mathit{MethodName} : \mathit{Identifier}, \rho, \mathit{wichName} \\ \mathit{\Gamma}, \mu, \mathcal{LV}_{,}\mathcal{E}_1, \mathcal{P} \vdash \mathit{Argument} : \tau_a, \mathcal{LV}_1, \mathcal{E}_2, \mathcal{C} \\ \mathsf{unAmbiguousMaxSpec}(\mathit{\Gamma}, \rho, (\texttt{Identifier}, \tau_a), \mu) = \mathit{mths} \\ \mathsf{methodDeclInfo}(\mathit{mths}) = \langle \mathit{ms}, \tau_e, \mathit{tcs} \rangle \\ \mathit{wichName} = \text{‘s’} \Rightarrow \mathsf{licitAppear}(\mathit{\Gamma}, \mu, \mathit{ms}, \mathcal{P}) \\ \mathit{wichName} = \text{‘c’} \Rightarrow \texttt{static} \in \mathit{ms} \quad \mathsf{throwsExceptions}(\mathit{\Gamma}, \mathit{tcs}, \mathcal{E}_2) = \mathcal{E}_3\end{array}}{\mathit{\Gamma}, \mu, \mathcal{LV}, \mathcal{E}_1, \mathcal{P} \vdash \mathit{MethodName}\ \texttt{(}\ \mathit{Argument}\ \texttt{)} : \tau_e, \mathcal{LV}_2, \mathcal{E}_3, \mathsf{false}}$$

$$\frac{\begin{array}{c}\mathit{\Gamma}, \mu, \mathcal{LV}_1, \mathcal{E}_1, \mathcal{P} \vdash \mathit{Primary} : \rho, \mathcal{LV}_2, \mathcal{E}_2 \\ \mathit{\Gamma}, \mu, \mathcal{LV}'_1, \mathcal{E}_2, \mathcal{P} \vdash \mathit{Argument} : \tau_a, \mathcal{LV}_3, \mathcal{E}_3, \mathcal{C} \\ \mathsf{unAmbiguousMaxSpec}(\mathit{\Gamma}, \rho, (\mathit{Identifier}, \tau_a)) = \mathit{mths} \\ \mathsf{methodDeclInfo}(\mathit{mths}) = \langle \mathit{ms}, \tau_e, \mathit{tcs} \rangle \quad \mathsf{throwsExceptions}(\mathit{\Gamma}, \mathit{tcs}, \mathcal{E}_3) = \mathcal{E}_4\end{array}}{\mathit{\Gamma}, \mu, \mathcal{LV}_1, \mathcal{E}_1, \mathcal{P} \vdash \mathit{Primary.Identifier}\ \texttt{(}\ \mathit{Argument}\ \texttt{)} : \tau_e, \mathcal{LV}_3, \mathcal{E}_4, \mathsf{false}}$$

Figure 34: Typing rules for expressions: part 3.

$$\frac{\begin{array}{c} \Gamma,\mu,\mathcal{LV}_1,\mathcal{E}_1,\mathcal{P} \vdash \mathit{Argument} : \tau_a,\mathcal{LV}_2,\mathcal{E}_2,\mathcal{C} \quad \mathsf{licitInvokeSuper}(\Gamma,\mu,\mathcal{P}) \\ \Gamma.\mathit{classMap}(\mu).\mathit{super} = \varsigma \\ \mathsf{unAmbiguousMaxSpec}(\Gamma,\varsigma,(\texttt{Identifier},\tau_a)) = \mathit{mths} \\ \mathsf{methodDeclInfo}(\mathit{mths}) = \langle \mathit{ms},\tau_e,\mathit{tcs}\rangle \quad \texttt{abstract} \notin \mathit{ms} \\ \mathsf{throwsExceptions}(\Gamma,\mathit{tcs},\mathcal{E}_2) = \mathcal{E}_3 \end{array}}{\Gamma,\mu,\mathcal{LV}_1,\mathcal{E}_1,\mathcal{P} \vdash \texttt{super}.\mathit{Identifier}\ \texttt{(}\ \mathit{Argument}\ \texttt{)} : \tau_e,\mathcal{LV}_2,\mathcal{E}_3,\mathsf{false}}$$

$$\boxed{\textsc{Assignment}} \quad \ldots \quad \boxed{\Gamma,\mu,\mathcal{LV}_1,\mathcal{E}_1,\mathcal{P} \vdash \mathit{AssignmentExpression} : \tau,\mathcal{LV}_2,\mathcal{E}_2,\mathcal{C}}$$

$$\frac{\begin{array}{c} \Gamma,\mu,\mathcal{LV}_1,\mathcal{E}_1,\mathcal{P} \vdash \mathit{SimpleFieldAccess} : \tau,\mathcal{LV}_2,\mathcal{E}_2,\mathsf{false} \\ \Gamma,\mu,\mathcal{LV}_2,\mathcal{E}_2,\mathcal{P} \vdash \mathit{Expression} : \tau_e,\mathcal{LV}_3,\mathcal{E}_3,\mathcal{C} \quad \tau_e \sqsubseteq_{\mathit{impl}} \tau \end{array}}{\Gamma,\mu,\mathcal{LV}_1,\mathcal{E}_1,\mathcal{P} \vdash \mathit{SimpleFieldAccess}\ \texttt{=}\ \mathit{Expression} : \tau,\mathcal{LV}_3,\mathcal{E}_3,\mathsf{false}}$$

$$\frac{\begin{array}{c} \Gamma,\mu,\mathcal{LV}_1,\mathcal{E}_1,\mathcal{P} \vdash \mathit{ArrayFieldAccess} : \sigma*,\mathcal{LV}_2,\mathcal{E}_2,\mathcal{C}_1 \\ \Gamma,\mu,\mathcal{LV}_2,\mathcal{E}_2,\mathcal{P} \vdash \mathit{Expression} : \tau_e,\mathcal{LV}_3,\mathcal{E}_3,\mathcal{C}_2 \quad \tau_e \sqsubseteq_{\mathit{impl}} \sigma* \end{array}}{\Gamma,\mu,\mathcal{LV}_1,\mathcal{E}_1,\mathcal{P} \vdash \mathit{ArrayFieldAccess}\ \texttt{=}\ \mathit{Expression} : \sigma*,\mathcal{LV}_3,\mathcal{E}_3,\mathsf{false}}$$

$$\frac{\begin{array}{c} \mathsf{fst}(\mathcal{LV}_1(\mathit{Identifier})) = \tau \\ \Gamma,\mu,\mathcal{LV}_1,\mathcal{E}_1,\mathcal{P} \vdash \mathit{Expression} : \tau_e,\mathcal{LV}_2,\mathcal{E}_2,\mathcal{C} \quad \tau_e \sqsubseteq_{\mathit{impl}} \tau \end{array}}{\begin{array}{c} \Gamma,\mu,\mathcal{LV}_1,\mathcal{E}_1,\mathcal{P} \vdash \mathit{Identifier}\ \texttt{=}\ \mathit{Expression} : \\ \tau,\mathcal{LV}_2[\mathit{Identifier} \mapsto (\tau,\mathsf{true})],\mathcal{E}_2,\mathsf{false} \end{array}}$$

$$\frac{\begin{array}{c} \mathsf{fst}(\mathit{LV}(\mathit{Identifier})) = \varsigma*\ \texttt{[]} \\ \Gamma,\mu,\mathcal{LV}_1,\mathcal{E}_1,\mathcal{P} \vdash \mathit{Expression}_1 : \pi,\mathcal{LV}_2,\mathcal{E}_2,\mathcal{C}_1 \quad \pi \sqsubseteq_{\mathit{impl}} \texttt{int} \\ \Gamma,\mu,\mathcal{LV}_2,\mathcal{E}_2,\mathcal{P} \vdash \mathit{Expression}_2 : \tau_e,\mathcal{LV}_3,\mathcal{E}_3,\mathcal{C}_2 \\ \tau_e \sqsubseteq_{\mathit{impl}} \sigma* \end{array}}{\begin{array}{c} \Gamma,\mu,\mathcal{LV}_1,\mathcal{E}_1,\mathcal{P} \vdash \mathit{Identifier}\ \texttt{[}\ \mathit{Expression}_1\ \texttt{]}\ \texttt{=}\ \mathit{Expression}_2 : \\ \sigma*,\mathcal{LV}_2[\mathit{Identifier} \mapsto (\sigma*,\mathsf{true})],\mathcal{E}_3,\mathsf{false} \end{array}}$$

Figure 35: Typing rules for expressions: part 4.

7 CONCLUSION

We have defined along this paper a formal static semantics for almost all of the Java language, even the flow analysis that is carried out by every Java compiler has been considered. Among other important features of the Java language, the formal system that we have developed covers modifiers, constructors, initialization and scoping rules. During the course of this work, we have exhaustively studied the Java specification and tested several versions of the JDK Java compiler of Sun. We were surprised by the subtlety and the non-uniformity of the language semantics, especially since its description is rather colossal, ambiguous, and erroneous. We believe that our work is very useful for the Java language understanding, it sheds some light on both its specification and semantics. Besides being the first work encompassing the static semantics of almost the whole of Java, our work gives a formal description that is rather clear and concise when compared with the official Java Language Specification. Indeed, our static semantics extends over around 50 pages, which is about sixth the size of the informal description of type-checking in Java. We do estimate that our work will serve as the solid basis for the development of a well-established semantics theory. We are currently working on the extension of our type-system to do a variety of robust static analysis techniques for security purposes such as self-certified Java code.

Mourad Debbabi is a Full Professor at the Concordia Institute for Information Systems Engineering of Concordia University. He is also a Concordia Research Chair Tier I in Information Systems Security. He holds Ph.D. and M.Sc. degrees in computer science from Paris-XI Orsay, University, France. He published more than 120 research papers in international journals and conferences on computer security, formal semantics, mobile and embedded platforms, Java technology security and acceleration, cryptographic protocol specification, design and analysis, malicious code detection, programming languages, type theory and specification and verification of safety-critical systems. In the past, he served as Senior Scientist at the Panasonic Information and Network Technologies Laboratory, Princeton, New Jersey, USA; Associate Professor at the Computer Science Department of Laval University, Quebec, Canada; Senior Scientist at General Electric Corporate 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. See also http://www.ciise.concordia.ca/~debbabi.

Myriam Fourati is a Ph.D. student at the Computer Science and Software Engineering Department of Laval University, Sainte-Foy, Quebec, Canada. She is also affiliated with the Computer Security Laboratory of Concordia University. She is finishing a Ph.D. thesis under the supervision of Dr. M. Debbabi on trusted self-certified code in Java technology. She has been awarded the President Prize of Tunisia for her research on Java security and semantics. She is also a lecturer at the Faculté des Sciences de Tunis.

REFERENCES

[1] Davide Ancona and Elena Zucca. Principal typings for Java-like languages. In *POPL '04: Proceedings of the 31st ACM SIGPLAN-SIGACT symposium on Principles of programming languages*, pages 306–317, New York, NY, USA, 2004. ACM Press.

[2] Isabelle Attali, Denis Caromel, and Marjorie Russo. Graphical Visualization of Java Objects, Threads, and Locks. *IEEE Distributed Systems Online*, 2(1), 2001.

[3] Anindya Banerjee and David A. Naumann. Using Access Control for Secure Information Flow in a Java-like Language. In *CSFW*, pages 155–169, 2003.

[4] Adrian Birka and Michael D. Ernst. A practical type system and language for reference immutability. In *Object-Oriented Programming Systems, Languages, and Applications (OOPSLA 2004)*, pages 35–49, Vancouver, BC, Canada, October 26–28, 2004.

[5] Sophia Drossopolou and Susan Eisenbach. Java is Type-Safe –Probably. In *Proceeedings of the 11th European Conference on Object-Oriented Programming*, Jyvskyl, Finland, 1997. LNCS.

[6] Sophia Drossopoulou and Susan Eisenbach. Is the Java Type System Sound? In *Fourth International Workshop on Foundations of Object-Oriented Languages*, page 22, January 1997.

[7] Sophia Drossopoulou and Susan Eisenbach. Towards an Operational Semantics and Proof of Type Soundness for Java. In Jim Alves-Foss, editor, *Formal Syntax and Semantics of Java*, volume 1523 of *LNCS*. Springer-Verlag, 1998.

[8] Stephen N. Freund and John C. Mitchell. A Type System for Object Initialization in the Java Bytecode Language. In *Proceedings OOPSLA '98*, pages 310–327, 1998.

[9] Bill Joy Gilad Bracha, James Gosling and Guy Steele. *The Java Language Specification, Second Edition.* The Java Series. Addison-Wesley, 2000.

[10] Bill Joy Guy Steele and James Gosling. Errors and Omissions in the Java Language Specification 1.0. http://www.dina.kvl.dk/~jsr/java-jls-errors.html, 1998.

[11] Denis Caromel Isabelle Attali and Marjorie Russo. A Formal Executable Semantics for Java. In *Proceedings of Formal Underpinnings of Java*, October 1998.

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

[13] Sun Microsystems. Clarifications and Amendments to the Java Language Specification. http://java.sun.com/docs/books/jls/clarify.html, 1998.

[14] Tobias Nipkow. Jinja: Towards a Comprehensive Formal Semantics for a Java-like Language. In H. Schwichtenberg and K. Spies, editors, *Proc. Marktobderdorf Summer School 2003*. IOS Press, 2003.

[15] Tobias Nipkow and David von Oheimb. Java$_{light}$ is Type-Safe –Definitely. In *Proceeedings of the 25th ACM Symp. Principles of Programming Languages*, pages 61–170. ACM Press, New York, 1998.

[16] Tobias Nipkow and David von Oheimb. Machine-checking the Java Specification: Proving Type-Safety. In Jim Alves-Foss, editor, *Formal Syntax and Semantics of Java*, volume 1523 of *LNCS*, chapter 5. Springer-Verlag, 1998.

[17] Isabelle Pollet and Baudouin Le Charlier. Towards a Complete Static Analyser for Java: an Abstract Interpretation Framework and its Implementation. *Electr. Notes Theor. Comput. Sci.*, 131:85–98, 2005.

[18] Vijay Saraswat. Java is not Type-Safe. Technical report, AT&T Research, 180 Park Avenue, Florham Park NJ 07932, 1997.

[19] Matthew Smith and Sophia Drossopoulou. Inner Classes visit Aliasing. In *ECOOP Workshop on Formal Techniques for Java Programs (FTfJP 2003)*, 2003.

[20] Tanya Valkevych Sophia Drossopoulou and Susan Eisenbach. Java Type Soundness Revisited. Technical report, Imperial College, November 1999.

[21] Donald Syme. Proving Java Type Sound. In Jim Alves-Foss, editor, *Formal Syntax and Semantics of Java*, volume 1523 of *LNCS*. Springer-Verlag, 1998.