Formalizing Java Exceptions

Position Paper

Tatyana Valkevych, Sophia Drossopoulou
Department of Computing, Imperial College of Science, Technology and Medicine
email: {tanya,sd}@doc.ic.ac.uk

Abstract

We present a summary of our formalization of the static and dynamic semantics of Java related to exceptions. We distinguish between normal execution, where if an exception is thrown then it is handled, and abnormal execution, where an exception is thrown and not handled. The type system distinguishes normal types which describe the possible outcomes of normal execution, and abnormal types which describe the possible outcomes of abnormal execution. The type of a term consists of its normal type and its abnormal type.

The meaning of our subject reduction theorem we proof with this set-up is stronger than usual: it guarantees that normal execution returns a value of a type compatible with the normal type of the term, and that abnormal execution throws an exception compatible with the abnormal type of the term.

1 Introduction

Exceptions and exception handling aim to support the development of robust programs with reliable error detection, and fast error handling [7]. Although integration of exception handling mechanisms and object-oriented languages is often considered as conflicting [1, 10], exceptions have been incorporated in most larger programming languages, *e.g.* Ada, Smalltalk, C++, SML, *etc.* On the other hand exceptions are not widely used by programmers because either they are not duly presented in language descriptions, or, on contrary, their description tends to be too long and complex. Many formal descriptions of exceptions have a very high abstract level [12, 14, 16]. We suggest formalization of Java exception which is both precise and near to Java programmer's intuition and can help bridge the gap in the use of this crucial feature. In this paper we present a summary of our formalization of the static and dynamic semantics of Java related to exceptions.[1]

1.1 Exceptions in Java

In Java [8] exceptions are objects, and behave as such (*i.e.* they can be assigned, perform a method, passed as parameters, *etc.*) until they are raised.

Exceptions are objects of the predefined class `Throwable` or its subclasses. Methods have a **throws** clause to declare exceptions potentially escaping methods execution. Java distinguishes between checked and unchecked exceptions. *Unchecked* exceptions are the subclasses of class `RuntimeException` describing run-time errors, *e.g.* dereferencing `null` or division by zero, *etc.*, and the subclasses of `Error` describing linkage and virtual machine errors, *e.g.* the absence of a `*.class` file or verification errors. Unchecked exceptions are exempt from the requirement of being declared. All the other exceptions are *checked* exceptions. A method *must* declare the class or superclass of any checked exception that might be raised during evaluation of its body. With this the compiler ensures that callers of a method either handle this method's potential exceptions, or they themselves explicitly mention these classes or the superclasses in *their* **throws** clauses.

[1] The complete formalization of the Java subset we have considered so far can be found in [6]; a version of this paper with examples and formal part there is in [4].

Java requires exceptions declared in overriding methods to conform [10] those in the overridden one, *i.e.* the exception classes declared in an overriding method must be subclasses of one of those declared in overridden ones. Therefore, **throws** clause *must also* mention the class or superclass of any checked exception that might be raised during evaluation of any overriding method.

Exceptions are raised – *thrown* in Java terminology – by **throw** statements. Once an exception is thrown, control is transferred to the nearest handler – **catch** clause – for that exception found in the dynamic call stack, and the exception is *propagated* through the *dynamic chain* of method calls, until its handler is reached.

Of course, it is unpredictable whether a term will evaluate with or without exceptions.[2]

1.2 Challenges in modelling Java exceptions

Exceptions in Java have dual nature. Namely, an object of exception class acts as any other object. However, after an exception is thrown, the exception object only serves to pass information to the most enclosing applicable exception handler. When such a handler has been found, the exception object "turns back" to being a usual object. To model this we need to distinguish between objects of exception class which have been thrown, and those that have not. We achieve this by representing the first by **throw** ι (where ι is the address of the exception object), and the latter – by ι (where ι is the address of any object) which is not enclosed in the **throw** statement.

A fine point is related to the exact moment when the exception is thrown and the search for the handler (exception propagation) starts. Namely, **throw** `e` requires complete evaluation of the expression `e` first: a term **throw** `e` can throw another exception of a type different than that of `e`.

So, firstly, we need to distinguish *normal execution*, where if exception is thrown then it is handled, and *abnormal execution*, where an exception is thrown and not handled. Secondly, we need to integrate these as a term has only one execution: either normal, or abnormal. For the description of abnormal execution we introduce a notion of *context* – a term that contains a **throw** ι term which is not included inside an exception handler. A thrown exception (normally evaluated) is considered in its context and propagated through it to the next most enclosing context.

To reflect this in the type system, we first introduce the (complete) type of a term which consists of two parts: the *normal type* describing the possible outcome under normal execution, and the *abnormal type* describing the possible outcomes under abnormal execution. This is unusual: a term has only one execution, and its type should reflect this fact.[3] Secondly, (to integrate the types and to describe how abnormal execution originates from normal execution), we introduce the *least* normal type for terms which definitely *will* throw exceptions. Terms which contain a thrown exception have the least normal type $\perp$ and the abnormal type $\{\texttt{E}\}$, where $\perp$ indicates that no normal outcome is possible, and `E` is the type of the thrown exception.[4]

Rewriting a term preserves both the normal type and the abnormal type; any exception it *may* raise is described by the abnormal type. Our definition of types as normal-abnormal types makes the meaning of the subject reduction theorem [17] stronger, namely that normal execution will return a value of a type compatible with the normal type, and abnormal execution will raise an exception whose class is a subclass of one of the classes in its abnormal type.

2 Java$^{\mathsf{r}}$, the run-time language

As in earlier works [2, 3] we define Java$^{\mathsf{s}}$, a provably *safe* subset of Java containing some primitive types, interfaces, classes with instance variables and instance methods, inheritance, hiding of

[2] C++ and SML exceptions are similar to Java exceptions in the way they are propagated and caught, and that exceptions behave as ordinary objects/values unless explicitly thrown. Unlike Java, C++ exceptions do not need to be objects of a special exception class; SML exceptions are special values of any type which have been annotated as `exception`. Neither C++, nor SML programmers are expected to declare the exceptions which escape a function.

[3] To describe properties different from the "classical" type system, effect systems [9, 15] are usually employed.

[4] Originally we had given such terms the type `E-Thrn` [5]. However, the current representation, suggested by an anonymous FOSSACS referee where **throw** `e` has $\perp$ type, fits seamlessly to the general case and correctly denotes the moment when we already can predict abnormal execution of the term.

instance variables, overloading and overriding of instance methods, arrays, the `null` value, object and array creation, field and array access, method call and dynamic method binding, assignment, and exception treatment.

The execution of Java programs requires some type information at run-time, *e.g.* field and method descriptors. For this reason, we defined Java$^{\mathtt{b}}$, an *enriched* version of Java$^{\mathtt{s}}$ containing compile-time type information necessary for the execution of method call and field access. Compilation maps Java$^{\mathtt{s}}$ terms to Java$^{\mathtt{b}}$ terms preserving their types. To describes *run-time* terms including addresses, we extended Java$^{\mathtt{b}}$ obtaining Java$^{\mathtt{r}}$.

Because exceptions are run-time events we concentrate our discussion on the run-time language Java$^{\mathtt{r}}$ defined in appendix A1. Java$^{\mathtt{r}}$ terms correspond to Java terms, except that Java$^{\mathtt{r}}$ is enriched with type information for execution (which is part of Java$^{\mathtt{b}}$) and also allows for run-time artifacts, namely, addresses of the form ι representing references to objects, the `null` value in field access and method call, and statements as expressions. The type information necessary for execution consists of field and method descriptors. Therefore, Java$^{\mathtt{r}}$ variables may have the form ι.[`ClassName`]`VarName` or `null`[*Expr*], expressions may have the form ι or **throw** ι**;return** `m()`**;**.

In order to give a succinct description of the operational semantics for propagation of and typing a thrown exception, we introduce the notion of context defined in appendix A2 through which an exception thrown is propagated. The *context* of an exception encompasses all enclosing terms up to the nearest enclosing **try-catch** or **try-catch-finally** clause, *i.e.* up to the first possible position at which the exception can be handled.

3 The operational semantics

Evaluation of Java$^{\mathtt{r}}$ terms takes place in the context of a state and a program. We define a small steps operational semantics as following:

$\leadsto$: Java$^{\mathtt{b}}$ program $\longrightarrow$ Java$^{\mathtt{r}}$-term $\times$ state $\longrightarrow$ ((Java$^{\mathtt{r}}$-term $\times$ state) $\cup$ (state))

where for a given program `P` the small rewriting steps are defined as:

$\underset{\mathtt{P}}{\leadsto}$: Java$^{\mathtt{r}}$-term $\times$ state $\longrightarrow$ ((Java$^{\mathtt{r}}$-term $\times$ state) $\cup$ (state)).

States, designated as σ, map identifiers onto primitive values or addresses, and addresses onto objects. Objects are annotated by their classes, *i.e.* $\sigma(\iota) = \ll ... \gg^{\mathtt{C}}$.

The operational semantics related to exceptions is given in appendix A3. The NE, AE abbreviation in the titles of inference rules means normal execution and abnormal execution correspondingly.

Execution of **try-catch** and **try-catch-finally** statements proceeds by execution of the **try** part. If no exception is thrown in the **try** part, then the most enclosing **try-catch** statement terminates, whereas the most enclosing **try-catch-finally** statement proceeds to execute the **finally** part.

A **throw** statement evaluates the corresponding expression – note that evaluation of that expression might itself throw a further expression. If the expression evaluates to `null`, then a `NullPE` will be thrown. Once the expression following **throw** is ground, *i.e.* once it reaches the form ι, the exception is propagated via its context.

Thus, the exception reaches either the outer level of the program, or it reaches an enclosing **try-catch** or **try-catch-finally** statement. An exception of a class which is not a subclass of any of the classes in the **catch** clauses is propagated out of an enclosing **try-catch** statement. Also, an exception of a class which is not a subclass of any of the classes in the **catch** clauses causes execution of the **finally** part of an enclosing **try-catch-finally** statement followed by renewed throwing of the original exception.

An exception is handled by the first handler whose class is a superclass of the exception's class: the statements of the corresponding **catch** clause are executed with the **catch** clause variable pointing to the exception object. The **finally** clause, if any, will follow execution of the handler independently of whether the handler executed normally or abnormally.

Thus, our formal description contains eleven rules and reflects the content of six pages of the Java specification [8] (p. 287-288, 291-294).

4 The Java^r type system

A Java^r term has a *normal type* which characterizes its normal execution, and an *abnormal type* which characterizes its abnormal execution.

Normal types are either $\bot$, or primitive types, like **int**, **char**, **bool**, or interfaces, or classes, or **void**; these are denoted by $\mathtt{T}$, $\mathtt{T}'$, *etc.* The least normal type, $\bot$, describes terms which definitely will throw exceptions. To describe normal types for statement sequence and conditional, we define the operations $\sqcap$ and $\sqcup$ giving *pessimistic union* and *optimistic intersection* of normal types:

- $\mathtt{T} \sqcap \mathtt{T}' = \begin{cases} \bot & \text{iff} \quad \mathtt{T} = \bot \\ \mathtt{T}' & \textit{otherwise} \end{cases}$

- $\mathtt{T} \sqcup \mathtt{T}' = \begin{cases} \bot & \text{iff} \quad \mathtt{T} = \bot \ \text{ and } \ \mathtt{T}' = \bot \\ \mathbf{void} & \text{iff} \quad (\mathtt{T} = \mathbf{void} \ \text{ or } \ \mathtt{T}' = \mathbf{void}) \ \text{ and } \ \mathtt{T}, \mathtt{T}' \in \{\mathbf{void}, \bot\} \\ \textit{Undef} & \textit{otherwise} \end{cases}$

Abnormal types are sets of subclasses of the predefined class `Throwable`; those are denoted by $\mathtt{ET}$, $\mathtt{ET}'$, *etc.*, or sets like $\{\mathtt{E}_1, \ldots, \mathtt{E}_\mathtt{q}\}$. In order to express conformance and difference for exceptions we introduce the relation $\subseteq_e$ and operation $\setminus^e_\mathtt{P}$ over abnormal types which are the set-theoretic subset and difference, taking subclasses into account, *i.e.*

- $\mathtt{P} \vdash \mathtt{ET} \ \subseteq_e \ \mathtt{ET}' \quad \text{iff} \quad \forall \mathtt{E} \in \mathtt{ET} : \ \mathtt{P} \vdash \mathtt{E} \sqsubseteq \mathtt{Error}, \ \text{ or } \ \mathtt{P} \vdash \mathtt{E} \sqsubseteq \mathtt{RuntimeException},$
  $\text{or } \ \exists \mathtt{E}' \in \mathtt{ET}' : \mathtt{P} \vdash \mathtt{E} \sqsubseteq \mathtt{E}'$

- $\mathtt{ET}' \setminus^e_\mathtt{P} \mathtt{ET} = \{ \ \mathtt{E}' \mid \ \mathtt{E}' \in \mathtt{ET}' \ \text{ and } \ \forall \mathtt{E} \in \mathtt{ET} : \ \neg \, \mathtt{P} \vdash \mathtt{E}' \sqsubseteq \mathtt{E} \ \}$

The type of an address ι depends on the object or array it points at in the current state σ. Therefore, Java^r type judgements take states σ into account, and have the form
$\mathtt{P}, \mathtt{V}, \sigma \Vdash^r \mathtt{t} \ : \ \mathtt{T} \,\|\, \mathtt{ET}$
for Java^r term $\mathtt{t}$, normal type $\mathtt{T}$, abnormal type $\mathtt{ET}$, program $\mathtt{P}$, and environment $\mathtt{V}$ mapping variables to their types. Some type rules can be found in appendix A4.

We first examine the normal types of the terms. If an object is stored at address ι, *i.e.* $\sigma(\iota) = \ll ... \gg^\mathtt{C}$ then its class $\mathtt{C}$ is the normal type of ι. The normal type of a method call is the return type of the method that fits the method descriptor $\mathtt{T}_1 \times ... \times \mathtt{T}_\mathtt{n}$ stored with the method call, provided that the arguments have types that can be widened to (are subclasses of) the corresponding argument types in the method descriptor. A **throw** statement has normal type $\bot$, provided that $\mathtt{E}$, the normal type of the expression `e`, is an exception class (*i.e.* a subclass of `Throwable`). The context of a term with $\bot$ normal type has also $\bot$ normal type. A **try**-**catch** statement has normal type **void**, provided that all classes mentioned in the **catch** clauses are exception classes, and that the **catch** clauses are well-typed. With the same conditions, a **try**-**catch**-**finally** statement has the type of its **finally** block's statements. Finally, a statement sequence has $\bot$ if the first statements has $\bot$; otherwise it has the same type as its last statement. If either branch of a conditional has type **void**, then the conditional has the type **void**; otherwise it has $\bot$.

We next examine the abnormal types of the terms. The abnormal type of an address ι is empty – no consistency can be violated. In most cases, the abnormal type of a term is the union of the abnormal types of its subterms (e.g. statement sequence, conditional statement). The abnormal type of a method call consists of the union of the abnormal types of the receiver, the arguments, and the abnormal type declared in the **throws** clause of the method header ($\mathit{ExcT}(\mathtt{methH})$). The abnormal type of a **throw** statement is the union of the normal type ($\mathtt{E}$) and the abnormal type ($\mathtt{ET}$) of the expression `e` that follows it, if `e` is different from `null`. Namely, if evaluation of `e` succeeds, then the **throw** statement will throw an exception compatible with $\mathtt{E}$; if evaluation of `e` throws an exception compatible with $\mathtt{ET}$, then the **throw** statement will throw an exception compatible with $\mathtt{ET}$. The abnormal type of a **throw** statement with `null` expression is set of `NullPE`. The context of a term with $\bot$ normal type has the same abnormal type as the term. Finally, the abnormal type of a **try**-**catch** statement is the abnormal type of the **try** part from which we exclude the exception classes of the **catch** clauses, united with the abnormal types of the **catch** clauses. Similar requirements for **try**-**catch**-**finally** statements.

5 Subject reduction

In order to express subject reduction we use widening, $\leq_w$, over the type $\mathsf{T} = \mathtt{T} \,\|\, \mathtt{ET}$. A type *widens* to another type if *the normal type of the first is either* $\perp$*, or identical to or a subclass of or a subinterface of or a subclass of a class that implements interface of, the normal type of the latter*, and if *every class in the abnormal type of the first is either an unchecked exception (i.e. subclass of* `Error` *or* `RunTimeException`*) or has a superclass in the abnormal type of the latter*:

$$\frac{\begin{array}{l}(\mathtt{T} = \perp \ \text{ and } \ \mathtt{P} \vdash \mathtt{T}' \diamond_{NorType}) \ \text{ or } \ \mathtt{T} = \mathtt{T}' \ \text{ or} \\ \quad \mathtt{P} \vdash \mathtt{T} \ \mathit{subclass} \ \mathtt{T}' \ \text{ or } \ \mathtt{P} \vdash \mathtt{T} \ \mathit{subinterface} \ \mathtt{T}' \ \text{ or } \ \mathtt{P} \vdash \mathtt{T} \ \mathit{implements} \ \mathtt{T}' \\ \mathtt{P} \vdash \mathtt{ET} \subseteq_e \mathtt{ET}'\end{array}}{\mathtt{P} \vdash \mathtt{T} \,\|\, \mathtt{ET} \ \leq_w \ \mathtt{T}' \,\|\, \mathtt{ET}'}$$

Execution is well-defined only if some conformance requirements are satisfied. We outline these requirements here, more can be found in [6]. A state σ *conforms* to `P` and `V`, *i.e.* $\mathtt{P},\mathtt{V} \vdash \sigma \diamond$, if all variables and addresses of σ contain values which *conform* to the variable type expected in `V` and `P`, *e.g.* objects are required to be constructed according to their classes, *etc.* An environment `V` *conforms* to an environment $\mathtt{V}'$, *i.e.* $\mathtt{V} \trianglelefteq \mathtt{V}'$, if `V` contains all variable declarations from $\mathtt{V}'$. Finally, a configuration $\mathtt{t},\sigma$ *conforms* to a type T, *i.e.* $\mathtt{P},\mathtt{V} \vdash \mathtt{t},\sigma \lhd \mathsf{T}$, if $\mathtt{P},\mathtt{V} \vdash \sigma \diamond$ and $\mathtt{P},\mathtt{V},\sigma \Vdash^{\mathsf{r}} \mathtt{t} \ : \ \mathsf{T}$.

For subject reduction we also need the notion of complete program, $\Vdash^{\mathsf{b}} \mathtt{P} \diamond$, which guarantees that the Java$^{\mathsf{b}}$ program `P` satisfies all the well-formedness requirements of Java and that it is complete, *i.e.* all the necessary classes have been linked and checked (see [6] for the details).

The subject reduction theorem states that execution of a term either terminates in a new state, or it produces a new term whose type widens to that of the original. Therefore, the theorem guarantees that normal execution returns a value compatible with the normal type of the original term, and abnormal execution terminates with an exception compatible with the abnormal type of the term. The proof can be found in [6].

Theorem 1 Subject Reduction *For Java*$^{\mathsf{b}}$ *program* `P` *with* $\Vdash^{\mathsf{b}} \mathtt{P} \diamond$*, environment* `V`*, state* σ*, non-ground Java*$^{\mathsf{r}}$ *term* `t`*, type* T *with* $\mathtt{P},\mathtt{V} \vdash \mathtt{t},\sigma \lhd \mathsf{T}$
either

- $\exists\ \sigma'$, $\mathtt{V}'$, $\mathtt{t}'$, T' *such that*:
  - $\mathtt{t},\sigma \leadsto_{\mathtt{P}} \mathtt{t}',\sigma'$ *and*
  - $\mathtt{V}' \trianglelefteq \mathtt{V}$, *and*
  - $\mathtt{P},\mathtt{V}' \vdash \mathtt{t}',\sigma' \lhd \mathsf{T}'$ *and* $\mathtt{P} \vdash \mathsf{T}' \ \leq_w \ \mathsf{T}$,

or

- $\exists \sigma'$, $\mathtt{ET}$: $\mathtt{t},\sigma \leadsto_{\mathtt{P}} \sigma'$ *and* $\mathtt{P},\mathtt{V} \vdash \sigma' \diamond$ *and* $\mathsf{T} = \mathbf{void} \,\|\, \mathtt{ET}$.

6 Conclusions

To our knowledge, our work is the first to model exceptions for such a large subset of Java, and demonstrate the soundness of the Java approach. In [13, 2, 11] operational semantics and type systems for Java and SML exceptions have been developed where method types do not mention the exceptions potentially escaping from their bodies.

Modelling Java exceptions and, in particular, the dual role they play whereby they may be used as any object in normal execution until explicitly thrown was quite challenging and required the distinction of normal and abnormal execution and composition of types from normal and abnormal type. The least normal type, $\perp$, succinctly characterizes terms whose execution cannot be finished normally and will throw an exception. The formal system we have developed is very close to Java and to programmers' intuitive ideas about program compilation and execution, and considerably more concise than [8]. So, the system itself and also a description based on it can give programmers a clear understanding of the behaviour of Java exceptions.

References

[1] Christophe Dony. Exception handling and object-oriented programming: Towards a synthesis. In *OOPSLA'90 Proceedings*, pages 322–330, 1990.

[2] 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*, LNCS. Springer, 1998. Chapter 3.

[3] Sophia Drossopoulou, Susan Eisenbach, and Sarfraz Khurshid. Is the Java Type System Sound? *Theory And Practice of Object Systems*, 5(1):3–24, 1999.

[4] Sophia Drossopoulou and Tanya Valkevych. Java Exceptions Throw no Surprises. Available from: http://www-doc.ic.ac.uk/project/slurp/, March 2000.

[5] Sophia Drossopoulou, Tanya Valkevych, and Susan Eisenbach. Java Type System with Exceptions. In Carlos Baquero, editor, *ECOOP'99 Reader - Poster Session*, June 1999.

[6] Sophia Drossopoulou, Tanya Valkevych, and Susan Eisenbach. Java Type Soundness Revisited. Technical report, Imperial College, April 2000. Also available from: http://www-doc.ic.ac.uk/project/slurp/.

[7] John B. Goodenough. Exception handling: Issues and proposed notation. *Communications of the ACM*, 18(12), December 1975.

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

[9] John M. Lucassen and David Gifford. Polymorphic Effect Systems. In *POPL'88 Proceedings*, January 1988.

[10] Robert Miller and Anand Tripathi. Issues with exception handling in object-oriented systems. In S. Matsuoka M Aksit, editor, *ECOOP'97 Proceedings*, volume 1241 of *Lecture Notes in Computer Science*. Springer Verlag, 1997.

[11] Robin Milner, Mads Tofte, and David MacQueen. *The Definition of Standard ML (Revised)*. MIT Press, 1997.

[12] H. Nakano. A Constructive Formalization of the Catch and Throw Mechanism. In *Proc. 7th Annual IEEE Symp. on Logic in Computer Science (LICS), Santa Cruz, California*, pages 82–89. IEEE, IEEE Computer Society Press, June 1992.

[13] Tobias Nipkow and David von Oheimb. Java$_{\ell ight}$ is Type-Safe — Definitely. In *POPL Proceedings*, 1998.

[14] Masahiko Sato. Intuitionistic and classical natural deduction systems with the catch and the throw rules. *Theoretical Computer Science*, 175:75–92, 1997.

[15] Jean-Pierre Talpin and Pierre Jouvelot. Polymorphic Type, Region and Effect Inference. *Functional Programming*, 2(3):245–271, July 1992.

[16] Hayo Thielecke. On Exceptions versus Continuations in the Presence of State. In *ESOP 2000*, LNCS. Springler-Verlag, 2000.

[17] Andrew Wright and Matthias Felleisen. A Syntactic Approach to Type Soundness. *Information and Computation*, 115(1), 1994.

Appendix

A1. Java$^{\mathrm{r}}$ syntax

$$
\begin{array}{lll}
\mathit{Stmts} & ::= & (\mathit{Stmt}\ \mathbf{;})^* \\
\mathit{Stmt} & ::= & \mathbf{if}\ \mathit{Expr}\ \ \mathbf{then}\ \mathit{Stmts}\ \ \mathbf{else}\ \mathit{Stmts}\ \mid\ \mathit{Var} = \mathit{Expr} \\
 & \mid & \mathit{Expr}.[\mathit{ArgType}]\mathtt{MethName}(\mathit{Expr}^*)\ \mid\ \mathbf{throw}\ \mathit{Expr} \\
 & \mid & \mathbf{try}\ \mathit{Stmts}\ (\mathbf{catch}\ \mathtt{ClassName\ Id}\ \mathit{Stmts})^*\ \mathbf{finally}\ \mathit{Stmts} \\
 & \mid & \mathbf{try}\ \mathit{Stmts}\ (\mathbf{catch}\ \mathtt{ClassName\ Id}\ \mathit{Stmts})^+ \\
\mathit{Expr} & ::= & \mathit{Value}\ \mid\ \mathit{Var}\ \mid \mathit{Expr}.[\mathit{ArgType}]\mathtt{MethName}\ (\mathit{Expr}^*) \\
 & \mid & \mathbf{new}\ \mathtt{ClassName()}\ \mid\ \mathbf{new}\ \mathit{SimpleType}\ ([\mathit{Expr}])^+([])^* \\
 & \mid & \mathit{Stmts}\ [\mathbf{return}\ \mathit{Expr}]\ \mid\ \mathbf{this} \\
\mathit{Var} & ::= & \mathtt{Name}\ \mid\ \mathit{Expr}.[\mathtt{ClassName}]\mathtt{VarName}\ \mid\ \mathit{Expr}[\mathit{Expr}] \\
 & \mid & \mathit{RefValue}.[\mathtt{ClassName}]\mathtt{VarName} \mid \mathit{RefValue}[\mathit{Expr}] \\
\mathit{Value} & ::= & \mathit{PrimValue}\ \mid\ \mathit{RefValue} \\
\mathit{RefValue} & ::= & \iota\ \mid\ \mathtt{null} \\
\mathit{PrimValue} & ::= & \mathit{intValue}\ \mid \mathit{charValue}\ \mid \mathit{boolValue}\ \mid \ldots \\
\mathit{SimpleType} & ::= & \mathit{PrimType}\ \mid \mathtt{ClassName}\ \mid \mathtt{InterfaceName} \\
\mathit{PrimType} & ::= & \mathbf{bool}\ \mid \mathbf{char}\ \mid \mathbf{int}\ \mid \ldots
\end{array}
$$

A2. Exception context

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

A3. Exception throwing, propagation and handling

$$try - catch - finally : NE : tryStmtsEvaluation$$

$$\frac{\mathtt{stmts},\sigma \underset{\mathtt{P}}{\leadsto} \mathtt{stmts}',\sigma'}{\begin{array}{l} \mathbf{try}\ \mathtt{stmts}\ \mathbf{catch}\ \mathtt{E_1}\ \mathtt{v_1}\ \mathtt{stmts_1}\ ...\ \mathbf{catch}\ \mathtt{E_n}\ \mathtt{v_n}\ \mathtt{stmts_n},\sigma \\ \quad \underset{\mathtt{P}}{\leadsto} \mathbf{try}\ \mathtt{stmts}'\ \mathbf{catch}\ \mathtt{E_1}\ \mathtt{v_1}\ \mathtt{stmts_1}\ ...\ \mathbf{catch}\ \mathtt{E_n}\ \mathtt{v_n}\ \mathtt{stmts_n},\sigma' \\ \mathbf{try}\ \mathtt{stmts}\ \mathbf{catch}\ \mathtt{E_1}\ \mathtt{v_1}\ \mathtt{stmts_1}\ ...\ \mathbf{catch}\ \mathtt{E_n}\ \mathtt{v_n}\ \mathtt{stmts_n}\ \mathbf{finally}\ \ \mathtt{stmts_{n+1}},\sigma \\ \quad \underset{\mathtt{P}}{\leadsto} \mathbf{try}\ \mathtt{stmts}'\ \mathbf{catch}\ \mathtt{E_1}\ \mathtt{v_1}\ \mathtt{stmts_1}\ ...\ \mathbf{catch}\ \mathtt{E_n}\ \mathtt{v_n}\ \mathtt{stmts_n}\ \mathbf{finally}\ \ \mathtt{stmts_{n+1}},\sigma' \end{array}}$$

$$try - catch - finally : NE : noXthrowing$$

$$\frac{\mathtt{stmts},\sigma \underset{\mathtt{P}}{\leadsto} \sigma'}{\begin{array}{l} \mathbf{try}\ \mathtt{stmts}\ \mathbf{catch}\ \mathtt{E_1}\ \mathtt{v_1}\ \mathtt{stmts_1}\ ...\ \mathbf{catch}\ \mathtt{E_n}\ \mathtt{v_n}\ \mathtt{stmts_n},\sigma \underset{\mathtt{P}}{\leadsto} \sigma' \\ \mathbf{try}\ \mathtt{stmts}\ \mathbf{catch}\ \mathtt{E_1}\ \mathtt{v_1}\ \mathtt{stmts_1}\ ...\ \mathbf{catch}\ \mathtt{E_n}\ \mathtt{v_n}\ \mathtt{stmts_n}\ \mathbf{finally}\ \ \mathtt{stmts_{n+1}},\sigma \\ \quad \underset{\mathtt{P}}{\leadsto} \mathtt{stmts_{n+1}},\sigma' \end{array}}$$

$$throw : NE : Xevaluation$$

$$\frac{\mathtt{e},\sigma \underset{\mathtt{P}}{\leadsto} \mathtt{e}',\sigma'}{\mathbf{throw}\ \mathtt{e},\sigma \underset{\mathtt{P}}{\leadsto} \mathbf{throw}\ \mathtt{e}',\sigma'}$$

$$throw : NE : NullPXgenerating$$

$$\frac{}{\mathbf{throw}\ \mathtt{null},\sigma \underset{\mathtt{P}}{\leadsto} \mathbf{throw}\ \mathbf{new}\ \mathtt{NullPE}(),\sigma}$$

$$throw : AE : Xpropagation$$

$$\frac{\mathtt{cont}\sqsubset\cdot\sqsupset \text{ a context}}{\mathtt{cont}\sqsubset\mathbf{throw}\ \iota\sqsupset,\sigma \underset{\mathtt{P}}{\leadsto} \mathbf{throw}\ \iota,\sigma}$$

$$try - catch - finally : AE : Xthrowing$$

$$\frac{\begin{array}{l} \sigma(\iota) = \ll ... \gg^{\mathtt{E}} \\ \forall \mathtt{k} \in \{1...\mathtt{n}\}\ \ \mathtt{P} \not\vdash \mathtt{E} \sqsubseteq \mathtt{E_k} \end{array}}{\begin{array}{l} \mathbf{try}\ \mathbf{throw}\ \iota\ \mathbf{catch}\ \mathtt{E_1}\ \mathtt{v_1}\ \mathtt{stmts_1}\ ...\ \ \mathbf{catch}\ \mathtt{E_n}\ \mathtt{v_n}\ \mathtt{stmts_n},\sigma \underset{\mathtt{P}}{\leadsto} \mathbf{throw}\ \iota,\sigma' \\ \mathbf{try}\ \mathbf{throw}\ \iota\ \mathbf{catch}\ \mathtt{E_1}\ \mathtt{v_1}\ \mathtt{stmts_1}\ ...\ \ \mathbf{catch}\ \mathtt{E_n}\ \mathtt{v_n}\ \mathtt{stmts_n}\ \mathbf{finally}\ \ \mathtt{stmts_{n+1}},\sigma \\ \quad \underset{\mathtt{P}}{\leadsto} \mathtt{stmts_{n+1}};\ \mathbf{throw}\ \iota,\sigma \end{array}}$$

$$try - catch - finally : NE : Xcatching$$

$$\frac{\begin{array}{l} \sigma(\iota) = \ll ... \gg^{\mathtt{E}} \\ \exists \mathtt{i} \in \{1...\mathtt{n}\}\ \ :\ \ \mathtt{P} \vdash \mathtt{E} \sqsubseteq \mathtt{E_i}\ \ \ AND\ \ \ \forall \mathtt{k} \in \{1...\mathtt{i}-1\}\ \ \mathtt{P} \not\vdash \mathtt{E} \sqsubseteq \mathtt{E_k} \\ \mathtt{stmts}' = \mathtt{stmts_i}[\mathtt{z}/\mathtt{v_i}], \mathtt{z} \text{ new in } \mathtt{stmts_i} \text{ and in } \sigma \\ \sigma' = \sigma[\mathtt{z} \mapsto \iota] \end{array}}{\begin{array}{l} \mathbf{try}\ \mathbf{throw}\ \iota\ \mathbf{catch}\ \mathtt{E_1}\ \mathtt{v_1}\ \mathtt{stmts_1}\ ...\ \ \mathbf{catch}\ \mathtt{E_n}\ \mathtt{v_n}\ \mathtt{stmts_n},\sigma \underset{\mathtt{P}}{\leadsto} \mathtt{stmts}',\sigma' \\ \mathbf{try}\ \mathbf{throw}\ \iota\ \mathbf{catch}\ \mathtt{E_1}\ \mathtt{v_1}\ \mathtt{stmts_1}\ ...\ \ \mathbf{catch}\ \mathtt{E_n}\ \mathtt{v_n}\ \mathtt{stmts_n}\ \mathbf{finally}\ \ \mathtt{stmts_{n+1}},\sigma \\ \quad \underset{\mathtt{P}}{\leadsto} \mathbf{try}\ \mathtt{stmts}'\ \mathbf{finally}\ \ \mathtt{stmts_{n+1}},\sigma' \end{array}}$$

A4. Types for some Java$^{\mathtt{r}}$ terms

...

$$\frac{\sigma(\iota) = \ll ... \gg^{\mathtt{C}}}{\mathtt{P},\mathtt{V},\sigma \vdash^{\mathtt{r}} \iota \ :\ \mathtt{C} \parallel \emptyset}$$

$$\frac{\begin{array}{l}\mathtt{P} \vdash \mathtt{T}'_0 \diamond_a \\ \mathtt{P},\mathtt{V},\sigma \vdash^{\mathtt{r}} \mathtt{e_i} \ :\ \mathtt{T}'_{\mathtt{i}} \parallel \mathtt{ET_i} \quad \mathtt{i} \in \{0...\mathtt{n}\}, \mathtt{n} \geq 0 \\ \mathtt{P} \vdash \mathtt{T}'_{\mathtt{i}} \ \leq_w \ \mathtt{T_i} \quad \mathtt{i} \in \{1...\mathtt{n}\} \\ \mathit{FirstFit}(\mathtt{P},\mathtt{m},\mathtt{T}'_0,\mathtt{T_1} \times ... \times \mathtt{T_n}) = \{\mathtt{methH}\}\end{array}}{\mathtt{P},\mathtt{V},\sigma \vdash^{\mathtt{r}} \mathtt{e_0}.[\mathtt{T_1} \times ... \times \mathtt{T_n}]\mathtt{m}(\mathtt{e_1}...\mathtt{e_n}) \ :\ \mathit{RetT}(\mathtt{methH}) \parallel (\cup\mathtt{ETi})^{\mathtt{n}}_{\mathtt{i}=0} \cup \mathit{ExcT}(\mathtt{methH})}$$

$$\frac{\begin{array}{l}\mathtt{e} \neq \mathtt{null} \\ \mathtt{P} \vdash \mathtt{E} \sqsubseteq \mathtt{Throwable} \\ \mathtt{P},\mathtt{V},\sigma \vdash^{\mathtt{r}} \mathtt{e} \ :\ \mathtt{E} \parallel \mathtt{ET}\end{array}}{\mathtt{P},\mathtt{V},\sigma \vdash^{\mathtt{r}} \textbf{throw } \mathtt{e} \ :\ \perp \parallel \{\mathtt{E}\} \cup \mathtt{ET}} \qquad \frac{}{\mathtt{P},\mathtt{V},\sigma \vdash^{\mathtt{r}} \textbf{throw } \mathtt{null} \ :\ \perp \parallel \{\mathtt{NullPE}\}}$$

$$\frac{\begin{array}{l}\mathtt{cont} \sqsubset \cdot \sqsupset \ \text{ is a context} \\ \mathtt{P},\mathtt{V},\sigma \vdash^{\mathtt{r}} \mathtt{t} \ :\ \perp \parallel \mathtt{ET}\end{array}}{\mathtt{P},\mathtt{V},\sigma \vdash^{\mathtt{r}} \mathtt{cont} \sqsubset \mathtt{t} \sqsupset \ :\ \perp \parallel \mathtt{ET}}$$

$$\frac{\begin{array}{l}\mathtt{n} \geq 0 \\ \mathtt{P} \vdash \mathtt{E_i} \ \sqsubseteq \mathtt{Throwable}, \quad \mathtt{i} \in \{1...\mathtt{n}\} \\ \mathtt{V}(\mathtt{v_i}) = \mathit{Undef} \quad \mathtt{i} \in \{1...\mathtt{n}\} \\ \mathtt{P},\mathtt{V};\ \mathtt{E_i}\ \mathtt{v_i},\sigma \vdash^{\mathtt{r}} \mathtt{stmts_i} \ :\ \mathtt{T_i} \parallel \mathtt{ET_i}, \quad \mathtt{i} \in \{1...\mathtt{n}\} \\ \mathtt{P},\mathtt{V},\sigma \vdash^{\mathtt{r}} \mathtt{stmts_j} \ :\ \mathtt{T_j} \parallel \mathtt{ET_j}, \quad \mathtt{j} \in \{0, \mathtt{n}+1\}\end{array}}{\begin{array}{ll}\mathtt{P},\mathtt{V},\sigma \vdash^{\mathtt{r}}_a & \textbf{try } \mathtt{stmts_0} \textbf{ catch } \mathtt{E_1}\ \mathtt{v_1}\ \mathtt{stmts_1}... \textbf{ catch } \mathtt{E_n}\ \mathtt{v_n}\ \mathtt{stmts_n} \ : \\ & \textbf{void} \parallel (\mathtt{ET_0} \setminus^{\mathtt{e}}_{\mathtt{P}} \{\mathtt{E_1},\dots,\mathtt{E_n}\}) \cup (\cup \mathtt{ET_i})^{\mathtt{n}}_{\mathtt{i}=1} \\ \mathtt{P},\mathtt{V},\sigma \vdash^{\mathtt{r}}_a & \textbf{try } \mathtt{stmts_0} \textbf{ catch } \mathtt{E_1}\ \mathtt{v_1}\ \mathtt{stmts_1}... \textbf{ catch } \mathtt{E_n}\ \mathtt{v_n}\ \mathtt{stmts_n} \textbf{ finally } \ \mathtt{stmts_{n+1}} \ : \\ & \textbf{void} \parallel (\mathtt{ET_0} \setminus^{\mathtt{e}}_{\mathtt{P}} \{\mathtt{E_1},\dots,\mathtt{E_n}\}) \cup (\cup \mathtt{ET_i})^{\mathtt{n}+1}_{\mathtt{i}=1}\end{array}}$$

$$\frac{\begin{array}{l}\mathtt{P},\mathtt{V},\sigma \vdash^{\mathtt{r}} \mathtt{e} \ :\ \textbf{bool} \parallel \mathtt{ET} \\ \mathtt{P},\mathtt{V},\sigma \vdash^{\mathtt{r}} \mathtt{stmts} \ :\ \mathtt{T}' \parallel \mathtt{ET}', \ \ \mathtt{P},\mathtt{V},\sigma \vdash^{\mathtt{r}} \mathtt{stmts}' \ :\ \mathtt{T}'' \parallel \mathtt{ET}'', \ \ \mathtt{P},\mathtt{V},\sigma \vdash^{\mathtt{r}} \mathtt{stmt} \ :\ \mathtt{T}''' \parallel \mathtt{ET}'''\end{array}}{\begin{array}{l}\mathtt{P},\mathtt{V},\sigma \vdash^{\mathtt{r}} \mathtt{stmts};\mathtt{stmt} \ :\ \mathtt{T}' \sqcap \mathtt{T}''' \parallel \mathtt{ET}' \cup \mathtt{ET}''' \\ \mathtt{P},\mathtt{V},\sigma \vdash^{\mathtt{r}} \textbf{if } \mathtt{e} \textbf{ then } \ \mathtt{stmts} \ \textbf{ else } \mathtt{stmts}' \ :\ \mathtt{T}' \sqcup \mathtt{T}'' \parallel \mathtt{ET} \cup \mathtt{ET}' \cup \mathtt{ET}''\end{array}}$$