Java Exceptions Throw no Surprises

Sophia Drossopoulou and Tatyana Valkevych
Department of Computing,
Imperial College of Science, Technology and Medicine
email: {sd,tanya}@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 no exception is thrown – or, more precisely, any exception thrown 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 prove 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 [3, 11], exceptions have been incorporated in most larger programming languages, *e.g.* Ada, Smalltalk, C++, SML, *etc.* In this paper we present a summary of our formalization of the static and dynamic semantics of Java related to exceptions. The complete formalization of the Java subset we have considered so far can be found in [6].

As in earlier works [4, 5] we define Java^{s}, a provably *safe* subset of Java containing some primitive types, interfaces, classes with instance variables and instance methods, inheritance, hiding of 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.

$$\begin{tikzcd}
\text{Java} \arrow[r, phantom, "\supset"] & \text{Java}^{s} \arrow[r, "\mathcal{C}"'] \arrow[d] & \text{Java}^{b} \arrow[r, phantom, "\subset"] \arrow[d] & \text{Java}^{r} \arrow[r, squiggly, "\mathsf{p}"'] \arrow[d] & \text{Java}^{r} \arrow[d] \\
 & \text{Type} \arrow[r, phantom, "="] & \text{Type} \arrow[r, phantom, "="] & \text{Type} \arrow[r, phantom, "\geq_w"] & \text{Type}
\end{tikzcd}$$

The execution of Java programs requires some type information at run-time, *e.g.* field and method descriptors. For this reason, we defined Java^{b}, an *enriched* version of Java^{s} containing compile-time type information necessary for the execution of method call and field access. Compilation $\mathcal{C}$ maps Java^{s} terms to Java^{b} terms preserving their types. Execution of run-time terms may produce terms which are not described by Java^{b}. Therefore we extended Java^{b}, obtaining Java^{r}, which describes *run-time* terms, including addresses.

Execution may proceed *normally*, where no exception is thrown – or, more precisely, any exception thrown is handled – or *abnormally* (abruptly in Java terminology), where an exception is thrown and not handled. A term therefore has a type which is a pair of a *normal type* and an *abnormal type.* The normal type describes the possible outcomes of normal execution. A normal type may be either a class, an interface, a primitive type, **void**, or $\perp$ for terms which definitely will throw exceptions. The abnormal type describes the possible outcomes of abnormal execution. An abnormal type is represented by a *set* of subclasses of the predefined `Throwable` class.

Excluding $\perp$, normal types are "usual" types, whereas abnormal types correspond to effect systems [10, 18]. Rewriting a term preserves both the normal type and the abnormal type; any exception it *may* raise is described by the abnormal type, *i.e.* by the effect system.

We define widening for types. A type T widens to another type T', if the normal part of T is $\perp$, or identical to or a subclass or subinterface of, the normal part of T', and if the abnormal part of T consists of classes which are unchecked exceptions, or identical to or subclasses of, corresponding ones in the abnormal part of T'.

We prove a subject reduction theorem [20] which states that term execution preserves types up to widening. Our definition of types as pairs of normal $-$abnormal types makes the meaning of the subject reduction theorem stronger, namely that normal execution will return a value of a type compatible with the normal type, and abnormal execution will throw an exception whose class is a subclass of one of the classes in its abnormal type.

The rest of this paper is organized as follows. In section 2 we describe Java exceptions through an example. Because exceptions are run-time events, we concentrate on the run-time language Java^{r} defined in section 3. The operational semantics for Java^{r} related to exceptions is described in section 4. In section 5 we discuss the types of Java^{r} terms. In section 6 we define widening and state the subject reduction theorem. Finally, in section 7 we draw some conclusions and describe further work. In appendix A1 we outline the requirements for complete, well-formed Java^{b} programs. In appendix A2 we demonstrate the execution of an example.

2 An example demonstrating exceptions

Exception handling in programming languages supports the development of robust programs [7]. When an exception is raised – *thrown* in Java terminology – control is transferred to the nearest handler – **catch** clause in Java – for that exception found in the dynamic call stack.

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

Exceptions are objects of the predefined class `Throwable` or its subclasses. Java distinguishes between checked and unchecked exceptions. The *unchecked* exceptions are exempt from the requirement of being declared. Unchecked exceptions are the subclasses of class `RuntimeException` describing run-time errors, *e.g.* dereferencing `null` or division by zero, and the subclasses of `Error` describing linkage and virtual machine errors, *e.g.* the absence of a `*.class` file or verification errors. All the other exceptions are checked exceptions.

Checked exceptions are intended to determine at compile-time whether such if thrown are either declared or handled. A method, therefore, must mention in its header all the *checked* exceptions that might escape from execution of its body, and the compiler ensures that callers of a method either handle this method's potential exceptions, or they themselves explicitly mention these exceptions in *their* headers.

C++ [16] and SML [12] exceptions are similar to Java exceptions in the way they are propagated and caught, and in 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.[1]

Exception throwing is subtle in Java. So, assignment to an array component
`null`[k] = 3/0; causes `ArithmeticException` whereas similar assignment
`null`[k] = 3+0; will throw `NullPointerException`. Even if array address $\iota \neq$ `null` then
ι[k] = 7; may cause `IndexOutOfBoundsException` if `k` is out of the array's bounds.

The following example $\mathtt{P}_{\mathtt{wr}}$ demonstrates some of these issues.

[1]This is why the SML formal description[12] does not have a concept corresponding to our abnormal types.

P_{wr}=
```
class Worry extends Exception {}

class Illness extends Exception {
    int severity;
    Illness treat() throws Worry {
            if(...) throw new Worry(); else return this;}
    Illness cure() {
            severity = -10;return this; }
}

class Person {
    int age;
    Illness diagnose(){return new Illness(); }
    void act() throws Worry, Illness{
            try { if (...) throw diagnose().treat(); else age=age+1; }
            catch( Illness i){ i.cure();} }
    void live() throws Worry, Illness{
            act(); Doctor d=null; d.act(); }
    void study() {
            try { act();}
            catch(Throwable x){age=age+2;} }
}

class Doctor extends Person {
    void act() throws Illness { throw diagnose(); }
    void live() { int i=diagnose().cure().severity; }
}
```

Consider the following declarations and initializations:

```
Person peter; peter = new Person();
Doctor david; david = new Doctor();
```

The above example demonstrates:

- Exception classes are those declared as subclasses of `Throwable`. The predefined class `Exception` is a subclass of `Throwable`, therefore the classes `Worry` and `Illness` are exception classes.
- Exceptions are thrown by **throw** statements, *e.g.* execution of `david.act()` throws an exception of class `Illness`.
- Unless thrown, objects of an exception class behave as normal objects. So, execution of `david.live()` will *not* throw any exception: the call `diagnose()` returns an `Illness` object, which executes the method `cure`, and then returns its field `severity`. Therefore, the method `live` in class `Doctor` has no **throws** clause.
- A method header may mention one or more exception classes in its **throws** clause.
- The **throws** clause of a method header *must* mention the class or superclass of any checked exception that might be thrown during evaluation of its body, and during evaluation of any overriding method. Therefore, the method header of `live` in class `Person`, which calls the method `act`, has to mention the exceptions `Illness` and `Worry`, because these may be thrown when executing the nested call of `act`, *i.e.* the classes `Illness` and `Worry`. Similarly, the method header for `act` in class `Person` has to mention the class `Worry`, because it might

be thrown in its method body, and it also has to mention the exception `Illness`, because `Illness` is a possible exception of the overriding method `act` from class `Doctor`.

- Unchecked exceptions need not be mentioned in **throws** clauses. That is why, although execution of `Doctor d=null; d.act();` inevitably throws a `NullPointerException`, this exception need not be mentioned in the **throws** clause of the header of the method `live`.
- An exception thrown and caught within a method need not be mentioned in the header of that method. For example, the method `study` of class `Person` does not mention any exception because any exception potentially thrown by the nested method call `act()` is caught by the **catch** clause.
- Once an exception is thrown, it is propagated through the *dynamic chain* of method calls, until a handler is reached. Thus, an exception of class `Worry` will be thrown when evaluating `peter.act()`. On the other hand, the call `peter.study()` will *not* terminate with a thrown exception, because any exception is caught inside the method body of `study`.
- Exception propagation follows the *dynamic chain* of method calls, and does *not* take overridden methods into account. Thus, `david.act()` will throw an exception of class `Illness`. This exception will *not* be caught in the **catch** clause of `act` in class `Person`; it will be propagated to the expression that contained the method call `david.act()`.
- In general, it is unpredictable whether a term will evaluate with or without exceptions. For example, depending on the outcome of the evaluation of the condition in the method `treat`, the term **new** `Illness().treat()` may throw an exception of class `Worry`, or it may return an object of class `Illness`.

3 Java^r, the run-time language

Because exceptions are run-time events we concentrate our discussion on the run-time language Java^r. Java^r is used only for execution, and therefore it contains terms only, and does not contain class, method, or interface definitions. Java^r terms correspond to Java terms, except that we sometimes omit separators like {, }, (,), , and ; when obvious. Furthermore, Java^r is enriched with type information for execution (which is part of Java^b) and also allows for run-time artifacts not possible in the source language or Java^b.

The type information necessary for execution consists of field and method descriptors. For example, the Java^s field access `peter.age` is represented in Java^b and Java^r as `peter.`[`Person`]`age`. Also the Java^s method call `peter.act()` is represented in Java^b and Java^r as `peter.`[]`act()` – the method descriptor is empty because there are no parameters.

The run-time artifacts are addresses, the `null` value in field access and method call, and statements as expressions. Addresses have the form ι, ι', ι_1, *etc.*; they represent references to objects and arrays, and may appear wherever a value is expected, as well as in array access and field access. Therefore, Java^r variables may have the form ι.[`ClassName`]`VarName` or ι[*Expr*], and expressions may have the form ι. An access to `null` may arise during evaluation of array access or field access; therefore, Java^r expressions may have the form `null.`[`ClassName`]`VarName` or `null`[*Expr*]. Furthermore, in order to describe method evaluation through in-line expansion rather than closures and stacks, we allow an expression in Java^r to consist of a sequence of statements, so that in the operational semantics a method call can be rewritten to a statement sequence.

In order to give a succinct description of the operational semantics of exception propagation (*i.e.* term's abnormal execution) and typing, we introduce the notion of context via which an exception thrown is propagated. The *context* of an exception, defined in figure 2, 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.

Stmts	::=	(*Stmt* **;**)*
Stmt	::=	**if** *Expr* **then** *Stmts* **else** *Stmts* $\mid$ *Var* = *Expr*
	$\mid$	*Expr*.[*ArgType*]`MethName`**(***Expr****)** $\mid$ **throw** *Expr*
	$\mid$	**try** *Stmts* (**catch** `ClassName Id` *Stmts*)* **finally** *Stmts*
	$\mid$	**try** *Stmts* (**catch** `ClassName Id` *Stmts*)$^+$
Expr	::=	*Value* $\mid$ *Var* $\mid$ *Expr*.[*ArgType*]`MethName` **(***Expr****)**
	$\mid$	**new** `ClassName`**()** $\mid$ **new** *SimpleType* ([*Expr*])$^+$([])*
	$\mid$	*Stmts* [**return** *Expr*] $\mid$ **this**
Var	::=	`Name` $\mid$ *Expr*.[`ClassName`]`VarName` $\mid$ *Expr*[*Expr*]
	$\mid$	*RefValue*.[`ClassName`]`VarName` $\mid$ *RefValue*[*Expr*]
Value	::=	*PrimValue* $\mid$ *RefValue*
RefValue	::=	ι $\mid$ `null`
PrimValue	::=	*intValue* $\mid$ *charValue* $\mid$ *boolValue* $\mid$ ...
SimpleType	::=	*PrimType* $\mid$ `ClassName` $\mid$ `InterfaceName`
PrimType	::=	**bool** $\mid$ **char** $\mid$ **int** $\mid$ ...

Figure 1: Java$^{\mathtt{r}}$ terms

Context	::=	$\sqsubset\cdot\sqsupset$.[`ClassName`]`VarName` $\mid$ $\sqsubset\cdot\sqsupset$[*Expr*] $\mid$ *Expr*[$\sqsubset\cdot\sqsupset$]
	$\mid$	**new** *VarType* $[Expr]_1$ ... $[\sqsubset\cdot\sqsupset]_k$... $[Expr]_n$
	$\mid$	$\sqsubset\cdot\sqsupset$.[*ArgType*]`MethName`**(***Expr****)**
	$\mid$	*Expr*.[*ArgType*]`MethName`**(**$Expr_1, \ldots \sqsubset\cdot\sqsupset_k, \ldots Expr_n$**)**
		where $n \geq 1, 1 \leq k \leq n$
	$\mid$	$\sqsubset\cdot\sqsupset$ = *Expr* $\mid$ *Var* = $\sqsubset\cdot\sqsupset$
	$\mid$	**if** $\sqsubset\cdot\sqsupset$ **then** *Stmts* **else** *Stmts*
	$\mid$	$\sqsubset\cdot\sqsupset$ **; return** *Expr* $\mid$ $\sqsubset\cdot\sqsupset$ **;** *Stmt*
	$\mid$	**return** $\sqsubset\cdot\sqsupset$ $\mid$ **throw** $\sqsubset\cdot\sqsupset$

Figure 2: Java$^{\mathtt{r}}$ exception contexts

4 The operational semantics

Evaluation of Java$^{\mathtt{r}}$ terms takes place in the context of a state and a program. Therefore, 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 map identifiers onto primitive values or addresses, and addresses onto objects or arrays. Objects are annotated by their classes. This annotation is used for example, when binding methods according to the run-time class of the receiver. The objects contain the name of their fields, the classes which contain the field declaration and the value of these fields. The explanation can be found in *e.g.* [6]. For example, σ_{10} is a possible state with:

$$\sigma_{10}(\mathtt{peter}) \quad = \quad \iota_{11}$$
$$\sigma_{10}(\iota_{11}) \quad = \quad \ll \mathtt{age}\ \mathtt{Person}:\ 0 \gg^{\mathtt{Person}}$$

We discuss some of the rules of the operational semantics. Figure 3 describes the evaluation of assignments. According to the first rule, the left-hand side is evaluated first, until it becomes l-ground (*i.e.* of the form `t = id`, or `t =` ι.[`C`]`f`, or `t = null`.[`C`]`f`, or `t =` ι[`k`], or `t = null[k]` for some identifier `id`, class `C`, field `f`, integer `k`, or address ι). Then, according to the next rule, the right-hand side of the assignment is evaluated, up to the point of obtaining a ground term.

$$\dfrac{\begin{array}{l}\mathtt{v}\text{ is not l-ground}\\ \mathtt{v},\sigma\underset{\mathtt{P}}{\leadsto}\mathtt{v}',\sigma'\end{array}}{\mathtt{v}{=}\mathtt{e},\sigma\underset{\mathtt{P}}{\leadsto}\mathtt{v}'{=}\mathtt{e},\sigma'}
\qquad
\dfrac{\begin{array}{l}\mathtt{v}\text{ is l-ground}\\ \mathtt{e},\sigma\underset{\mathtt{P}}{\leadsto}\mathtt{e}',\sigma'\end{array}}{\mathtt{v}{=}\mathtt{e},\sigma\underset{\mathtt{P}}{\leadsto}\mathtt{v}{=}\mathtt{e}',\sigma'}$$

$$\dfrac{\begin{array}{l}\mathtt{val},\mathtt{k}\text{ are ground}\\ \mathtt{id}\text{ is an identifier}\end{array}}{\begin{array}{l}\mathtt{id}{=}\mathtt{val},\sigma\underset{\mathtt{P}}{\leadsto}\sigma[\mathtt{id}\mapsto\mathtt{val}]\\ \iota.[\mathtt{C}]\mathtt{f}{=}\mathtt{val},\sigma\underset{\mathtt{P}}{\leadsto}\sigma[\iota,\mathtt{f},\mathtt{C}\mapsto\mathtt{val}]\\ \mathtt{null}.[\mathtt{C}]\mathtt{f}{=}\mathtt{val},\sigma\underset{\mathtt{P}}{\leadsto}\textbf{throw new }\mathtt{NullPE}(),\sigma\\ \mathtt{null}[\mathtt{k}]{=}\mathtt{val},\sigma\underset{\mathtt{P}}{\leadsto}\textbf{throw new }\mathtt{NullPE}(),\sigma\end{array}}
\qquad
\dfrac{\begin{array}{l}\mathtt{val},\mathtt{k}\text{ are ground}\\ \sigma(\iota)=[\![\mathtt{val}_0\ldots\mathtt{val}_{\mathtt{n-1}}]\!]^{\mathtt{T}[]_1\cdots[]_{\mathtt{m}}}\\ 0>\mathtt{k},\text{ or }\mathtt{k}>n-1\end{array}}{\begin{array}{l}\iota[\mathtt{k}]{=}\mathtt{val},\sigma\underset{\mathtt{P}}{\leadsto}\\ \quad\textbf{throw new }\mathtt{IndOutBndE}(),\sigma\end{array}}$$

$$\dfrac{\begin{array}{l}\mathtt{val},\mathtt{k}\text{ are ground}\\ \sigma(\iota)=[\![\mathtt{val}_0\ldots\mathtt{val}_{\mathtt{n-1}}]\!]^{\mathtt{T}[]_1\cdots[]_{\mathtt{m}}}\\ 0\le\mathtt{k}\le n-1\\ \mathtt{val}\text{ does not fit }\mathtt{T}[]_1\ldots[]_{\mathtt{m}}\text{ in }\mathtt{P},\sigma\\ \textbf{new }\mathtt{ArrStoreE}(),\sigma\underset{\mathtt{P}}{\leadsto}\iota',\sigma'\end{array}}{\iota[\mathtt{k}]{=}\mathtt{val},\sigma\underset{\mathtt{P}}{\leadsto}\textbf{throw new }\mathtt{ArrStoreE}(),\sigma}
\qquad
\dfrac{\begin{array}{l}\mathtt{val},\mathtt{k}\text{ are ground}\\ \sigma(\iota)=[\![\mathtt{val}_0\ldots\mathtt{val}_{\mathtt{n-1}}]\!]^{\mathtt{T}[]_1\cdots[]_{\mathtt{m}}}\\ 0\le\mathtt{k}\le n-1\\ \mathtt{val}\text{ fits }\mathtt{T}[]_1\ldots[]_{\mathtt{m}}\text{ in }\mathtt{P},\sigma\end{array}}{\iota[\mathtt{k}]{=}\mathtt{val},\sigma\underset{\mathtt{P}}{\leadsto}\sigma[\iota,\mathtt{k}\mapsto\mathtt{val}]}$$

Figure 3: Assignment execution

Assignment to variables or to object or array components modifies the state accordingly.

Assignment to array (object) components may throw various exceptions. Firstly they check whether the array (object) address is not `null`; if no, `NullPE` is generated to throw. Secondly, they check if the array index is within the array bounds; if not – `IndOutBndE` will be thrown. Thirdly, they check fitting the value to the array type; if this is not the case, `ArrStoreE` is to throw. Finally, if these requirements are satisfied, then the assignment is performed.

Figure 4 describes the operational semantics related to exceptions. NE, AE, X abbriviations in the titles of inference rules mean normal execution, abnormal execution, exception 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.

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 not covered[2] by any of the classes in the **catch** clauses is propagated out of an enclosing **try**-**catch** statement. Also, an exception of a class not covered by 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 ($\mathtt{v_i}$) 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.[3]

The last two rules in figure 4 reflect the contents of pages 291-294 of the Java specification where the evaluation of **try**-**catch** and **try**-**catch**-**finally** is described using twenty two rules written in natural language.

[2]The judgement $\mathtt{P}\vdash\mathtt{C}\sqsubseteq\mathtt{C}'$ means that $\mathtt{C}$ is a subclass of $\mathtt{C}'$.

[3]The appendix outlines possible executions of the term `peter.[]act()`

$$\dfrac{\mathtt{stmts},\sigma \leadsto_{\mathtt{P}} \mathtt{stmts}',\sigma'}{\begin{array}{l} \textbf{try}\ \mathtt{stmts}\ \textbf{catch}\ \mathtt{E_1}\ \mathtt{v_1}\ \mathtt{stmts_1}\ ...\ \textbf{catch}\ \mathtt{E_n}\ \mathtt{v_n}\ \mathtt{stmts_n},\sigma \\ \quad \leadsto_{\mathtt{P}} \textbf{try}\ \mathtt{stmts}'\ \textbf{catch}\ \mathtt{E_1}\ \mathtt{v_1}\ \mathtt{stmts_1}\ ...\ \textbf{catch}\ \mathtt{E_n}\ \mathtt{v_n}\ \mathtt{stmts_n},\sigma' \\ \textbf{try}\ \mathtt{stmts}\ \textbf{catch}\ \mathtt{E_1}\ \mathtt{v_1}\ \mathtt{stmts_1}\ ...\ \textbf{catch}\ \mathtt{E_n}\ \mathtt{v_n}\ \mathtt{stmts_n}\ \textbf{finally}\ \ \mathtt{stmts_{n+1}},\sigma \\ \quad \leadsto_{\mathtt{P}} \textbf{try}\ \mathtt{stmts}'\ \textbf{catch}\ \mathtt{E_1}\ \mathtt{v_1}\ \mathtt{stmts_1}\ ...\ \textbf{catch}\ \mathtt{E_n}\ \mathtt{v_n}\ \mathtt{stmts_n}\ \textbf{finally}\ \ \mathtt{stmts_{n+1}},\sigma' \end{array}} \quad \mathit{try : NE : stmtsEvaluation}$$

$$\dfrac{\mathtt{stmts},\sigma \leadsto_{\mathtt{P}} \sigma'}{\begin{array}{l} \textbf{try}\ \mathtt{stmts}\ \textbf{catch}\ \mathtt{E_1}\ \mathtt{v_1}\ \mathtt{stmts_1}\ ...\ \textbf{catch}\ \mathtt{E_n}\ \mathtt{v_n}\ \mathtt{stmts_n},\sigma \leadsto_{\mathtt{P}} \sigma' \\ \textbf{try}\ \mathtt{stmts}\ \textbf{catch}\ \mathtt{E_1}\ \mathtt{v_1}\ \mathtt{stmts_1}\ ...\ \textbf{catch}\ \mathtt{E_n}\ \mathtt{v_n}\ \mathtt{stmts_n}\ \textbf{finally}\ \ \mathtt{stmts_{n+1}},\sigma \\ \quad \leadsto_{\mathtt{P}} \mathtt{stmts_{n+1}},\sigma' \end{array}} \quad \mathit{try : NE : noXthrowing}$$

$$\dfrac{\mathtt{e},\sigma \leadsto_{\mathtt{P}} \mathtt{e}',\sigma'}{\textbf{throw}\ \mathtt{e},\sigma \leadsto_{\mathtt{P}} \textbf{throw}\ \mathtt{e}',\sigma'} \quad \mathit{throw : NE : Xevaluation}$$

$$\dfrac{}{\textbf{throw}\ \mathtt{null},\sigma \leadsto_{\mathtt{P}} \textbf{throw}\ \textbf{new}\ \mathtt{NullPE()},\sigma} \quad \mathit{throw : NE : NullPXgenerating}$$

$$\dfrac{\mathtt{cont}\sqsubset\cdot\sqsupset\ \text{a context}}{\mathtt{cont}\sqsubset\textbf{throw}\ \iota\sqsupset,\sigma \leadsto_{\mathtt{P}} \textbf{throw}\ \iota,\sigma} \quad \mathit{throw : AE : Xpropagation}$$

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

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

Figure 4: Exception throwing, propagation and handling

5 The Java$^{\mathtt{r}}$ type system

A Java$^{\mathtt{r}}$ 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:

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

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

Abnormal types are sets of subclasses of the predefined class `Throwable`; those are denoted by $\mathtt{ET}$, $\mathtt{ET}'$, *etc.*, or by sets like $\{\mathtt{E_1}, \ldots, \mathtt{E_q}\}$. For example, `Illness`, `Person`, **void** are normal types; {`Illness, Worry`}, {`NullPointerException, Worry`} are abnormal types.[4] 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$^{\mathtt{r}}$ type judgements take states σ into account, and have the form

$\mathtt{P}, \mathtt{V}, \sigma \Vdash^{\mathtt{r}} \mathtt{t} \;:\; \mathtt{T} \,\|\, \mathtt{ET}$

for Java$^{\mathtt{r}}$ term $\mathtt{t}$, normal type $\mathtt{T}$, abnormal type $\mathtt{ET}$, program $\mathtt{P}$, and environment $\mathtt{V}$ mapping variables to their types. In figure 5 we show the type rules for some Java$^{\mathtt{r}}$ terms.[5]

We first examine the normal types of the terms. If an object is stored at address ι, *i.e.* $\sigma(\iota) = \ll\!\ldots\!\gg^{\mathtt{C}}$, then its class $\mathtt{C}$ is the normal type of ι. An assignment is type correct, and has normal type **void**, if its right-hand side has a type which widens (is a subclass) of that of the left-hand side. The normal type of a method call is the return type of the method that fits the method descriptor $\mathtt{T_1} \times \ldots \times \mathtt{T_n}$ stored with the method call, provided that the arguments have types that can be widened to 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 $\mathtt{e}$, 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 (*i.e.* both have $\bot$) it has the type $\bot$.

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. assignment, 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** part of the method header ($\mathit{ExcT}(\mathtt{methH})$). The abnormal type of a **throw** statement is the union of the set of the normal type ($\mathtt{E}$) and the

[4]Compared with [21], normal types correspond to the sets $\mathcal{X}_e$, abnormal types correspond to the sets $\mathcal{P}_e$.

[5]In our current work, in order to model separte compilation we define typings in terms of declarations `D` derived from programs rather than the programs themselves. For simplicity of presentation we dropped the distinction between `D` and `P` in this paper

...

$$\frac{\begin{array}{l} \mathtt{v} \neq \mathtt{e}'[\mathtt{e}''] \;\; \text{for any } \mathtt{e}', \mathtt{e}'' \\ \mathtt{P}, \mathtt{V}, \sigma \Vdash^{\mathtt{r}} \mathtt{v} \;:\; \mathtt{T} \,\|\, \mathtt{ET} \\ \mathtt{P}, \mathtt{V}, \sigma \Vdash^{\mathtt{r}} \mathtt{e} \;:\; \mathtt{T}' \,\|\, \mathtt{ET}' \\ \mathtt{P} \vdash \mathtt{T}' \;\leq_w\; \mathtt{T} \end{array}}{\mathtt{P}, \mathtt{V}, \sigma \Vdash^{\mathtt{r}} \mathtt{v{=}e} \;:\; \mathbf{void} \,\|\, \mathtt{ET} \cup \mathtt{ET}'}$$

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

$$\frac{\begin{array}{l} \mathtt{P} \vdash \mathtt{T}'_0 \;\diamond_a \\ \mathtt{P}, \mathtt{V}, \sigma \Vdash^{\mathtt{r}} \mathtt{e_i} \;:\; \mathtt{T}'_\mathtt{i} \,\|\, \mathtt{ET_i} \quad \mathtt{i} \in \{0 ... \mathtt{n}\}, \mathtt{n} \geq 0 \\ \mathtt{P} \vdash \mathtt{T}'_\mathtt{i} \;\leq_w\; \mathtt{T_i} \quad \mathtt{i} \in \{1 ... \mathtt{n}\} \\ \mathit{FirstFit}(\mathtt{P}, \mathtt{m}, \mathtt{T}'_0, \mathtt{T}_1 \times ... \times \mathtt{T_n}) = \{\mathtt{methH}\} \end{array}}{\mathtt{P}, \mathtt{V}, \sigma \Vdash^{\mathtt{r}} \mathtt{e}_0.[\mathtt{T}_1 \times ... \times \mathtt{T_n}]\mathtt{m}(\mathtt{e}_1 ... \mathtt{e_n}) \;:\; \mathit{RetT}(\mathtt{methH}) \,\|\, (\cup \mathtt{ETi})^{\mathtt{n}}_{\mathtt{i}=0} \cup \mathit{ExcT}(\mathtt{methH})}$$

$$\frac{\begin{array}{l} \mathtt{e} \neq \mathtt{null} \\ \mathtt{P} \vdash \mathtt{E} \sqsubseteq \mathtt{Throwable} \\ \mathtt{P}, \mathtt{V}, \sigma \Vdash^{\mathtt{r}} \mathtt{e} \;:\; \mathtt{E} \,\|\, \mathtt{ET} \end{array}}{\mathtt{P}, \mathtt{V}, \sigma \Vdash^{\mathtt{r}} \mathbf{throw}\ \mathtt{e} \;:\; \bot \,\|\, \{\mathtt{E}\} \cup \mathtt{ET}} \qquad \frac{}{\mathtt{P}, \mathtt{V}, \sigma \Vdash^{\mathtt{r}} \mathbf{throw}\ \mathtt{null} \;:\; \bot \,\|\, \{\mathtt{NullPE}\}}$$

$$\frac{\begin{array}{l} \mathtt{cont} \sqsubset \cdot \sqsupset \;\; \text{is a context} \\ \mathtt{P}, \mathtt{V}, \sigma \Vdash^{\mathtt{r}} \mathtt{t} \;:\; \bot \,\|\, \mathtt{ET} \end{array}}{\mathtt{P}, \mathtt{V}, \sigma \Vdash^{\mathtt{r}} \mathtt{cont} \sqsubset \mathtt{t} \sqsupset \;:\; \bot \,\|\, \mathtt{ET}}$$

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

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

Figure 5: Types for some Java$^{\mathtt{r}}$ terms.

abnormal type (ET) of the expression e that follows the **throw**, if e is different from null. Namely, if evaluation of e succeeds, then the **throw** statement will throw an exception compatible with E; if evaluation of e throws an exception compatible with ET, then the **throw** statement will throw an exception compatible with 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 from the **catch** clauses, united with the abnormal types of the **catch** clauses. Similar requirements for **try**-**catch**-**finally** statements.

Concerning with our example, and taking $\mathtt{V_{wr}} = \mathtt{Person\ peter}$ and a state σ with $\sigma(\iota) = \ll ... \gg^{\mathtt{Illness}}$, we obtain the following typings:

$\mathtt{P_{wr}}, \mathtt{V_{wr}}, \sigma \Vdash^{r}$ peter.[]diagnose() : Illness $\|$ {}
$\mathtt{P_{wr}}, \mathtt{V_{wr}}, \sigma \Vdash^{r}$ **throw** peter.[]diagnose() : $\bot$ $\|$ {Illness}
$\mathtt{P_{wr}}, \mathtt{V_{wr}}, \sigma \Vdash^{r}$ peter.[]diagnose().[]treat() : Illness $\|$ {Worry}
$\mathtt{P_{wr}}, \mathtt{V_{wr}}, \sigma \Vdash^{r}$ **throw** peter.[]diagnose().[]treat() : $\bot$ $\|$ {Worry, Illness}
$\mathtt{P_{wr}}, \mathtt{V_{wr}}, \sigma \Vdash^{r}$ peter.[]act() : **void** $\|$ {Worry, Illness}
$\mathtt{P_{wr}}, \mathtt{V_{wr}}, \sigma \Vdash^{r}$ peter.[]study() : **void** $\|$ {},
$\mathtt{P_{wr}}, \mathtt{V_{wr}}, \sigma \Vdash^{r}$ **throw** ι : $\bot$ $\|$ {Illness}
$\mathtt{P_{wr}}, \mathtt{V_{wr}}, \sigma \Vdash^{r}$ (**throw** ι).[]act() : $\bot$ $\|$ {Illness}
$\mathtt{P_{wr}}, \mathtt{V_{wr}}, \sigma \Vdash^{r}$ **try** (**throw** ι).[]act()... **catch**... : **void** $\|$ {...}

6 Widening and subject reduction

Widening, $\leq_w$, is defined for all types excluding $\bot$; some rules are given in figure 6.

Widening for normal types corresponds to subtyping. A normal type widens to another normal type if they are not $\bot$, or are identical, or if the first is a subclass or subinterface or a subclass of a class that implements interface of, the latter. Widening for normal types is required for assignments and for parameter passing. For example,

$\mathtt{P_{wr}} \vdash \mathtt{Doctor} \ \leq_w \ \mathtt{Person}$

which makes the assignment peter=david; type-correct.

A normal-abnormal type *widens* to another type if the normal type of the first is either $\bot$, or widens to 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. Widening for normal-abnormal type is used to express subject reduction.[6]

$$\frac{\mathtt{P} \vdash \mathtt{T} \sqsubseteq \mathtt{T'}}{\mathtt{P} \vdash \mathtt{T} \ \leq_w \ \mathtt{T'}} \qquad \ldots \qquad \frac{\mathtt{P} \vdash \mathtt{T} \ \leq_w \ \mathtt{T'} \ \text{ or } \ \mathtt{T} = \bot \ \text{ and } \ \mathtt{P} \vdash \mathtt{T'} \diamond_{NorType} \qquad \mathtt{P} \vdash \mathtt{ET} \subseteq_e \mathtt{ET'}}{\mathtt{P} \vdash \mathtt{T} \,\|\, \mathtt{ET} \ \leq_w \ \mathtt{T'} \,\|\, \mathtt{ET'}}$$

Figure 6: Widening – some rules

Applying these rules to our example:

$\mathtt{P_{wr}} \vdash$ Illness $\|$ {} $\leq_w$ Illness $\|$ {Worry},
$\mathtt{P_{wr}} \vdash$ $\bot$ $\|$ {NullPE} $\leq_w$ Illness $\|$ {Worry},
$\mathtt{P_{wr}} \vdash$ $\bot$ $\|$ {Worry} $\leq_w$ **void** $\|$ {Worry, Illness}.

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

[6]In systems where a term is allowed to throw any exception the situation is much simpler, and abnormal types need not be considered. A raise expression may have any type τ in [12]. In [5, 4], where we had not yet considered the requirement to declare exceptions in throws clauses, the type E-Thrn widens to any normal type.

conforms to an environment $\mathtt{V}'$, *i.e.* $\mathtt{V} \trianglelefteq \mathtt{V}'$, if $\mathtt{V}$ contains all variable declarations from $\mathtt{V}'$. Finally, a configuration $\mathtt{t},\sigma$ *conforms* to type T, *i.e.* $\mathtt{P},\mathtt{V} \vdash \mathtt{t},\sigma \lhd \mathsf{T}$, if $\mathtt{P},\mathtt{V} \vdash \sigma \diamond$ and $\mathtt{P},\mathtt{V},\sigma \models^{\mathrm{r}} \mathtt{t} : \mathsf{T}$.

For subject reduction we also need the notion of complete program, $\models^{\mathrm{b}} \mathtt{P} \diamond$, which guarantees that the Java$^{\mathrm{b}}$ program $\mathtt{P}$ 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 in appendix A1).

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.

Theorem 1 Subject Reduction *For Java*$^{\mathrm{b}}$ *program* $\mathtt{P}$ *with* $\models^{\mathrm{b}} \mathtt{P} \diamond$*, environment* $\mathtt{V}$*, state* σ*, non-ground Java*$^{\mathrm{r}}$ *term* $\mathtt{t}$*, 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* $\mathsf{T} = \mathbf{void} \,||\, \mathtt{ET}$

Going again to our example, the subject reduction theorem and the typing
$\mathtt{P}_{\mathtt{wr}},\mathtt{V}_{\mathtt{wr}},\sigma_{10} \models^{\mathrm{r}} \mathtt{peter.[]act()} : \mathbf{void} \,||\, \{\mathtt{Worry},\mathtt{Illness}\}$
guarantee that execution of the term `peter.[]act()` in the state σ_{10} will either terminate successfully or throw an exception of class `Worry` or `Illness` (or a subclass), or throw an unchecked exception such as `NullPointerException` or `OutOfMemoryError`. In appendix A2 we follow three possible execution paths, one where no exception is thrown, one where an `Illness` exception thrown and caught, and one where a `Worry` exception thrown, and demonstrate that in all cases the final state conforms to the program and the environment.

7 Conclusions, Comparisons and Further Work

To our knowledge, our work is the first to model Java exceptions, and to demonstrate that the type system guarantees that the classes mentioned in the throws-clauses of methods indeed describe any potentially escaping unchecked exceptions. In [13, 4, 1, 12] operational semantics and type systems for Java and SML exceptions are developed where method types do not mention the exceptions potentially escaping from their bodies. [2] model Java exception handling and virtual machine subroutines, and prove that the compilation of exception related features from Java to JVM is meaning preserving.

The formal system we have developed is very close to Java and to programmers' intuitive ideas about program compilation and execution. The proofs of the lemmas and theorem can be found in [6]. The work would benefit from the use of a theorem prover, as *e.g.* in [19, 17].

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. Formalization of language features leads to better understanding; for example, introducing $\perp$ as the least type for a normal type, consideration type as normal -abnormal types combination, or our succinct description of the operational semantics of exception propagation and handling, *etc.* Our recent work on Java dynamic linker-loader, take the linkage exceptions `NoClassDefFoundError`, `IncompatibleClassChangeError` into account and give the complete the picture of dynamic linking. We hope that our more concise description influences the next version of the Java specification.

On a related topic, the inference of the **throws** clauses – as opposed to simple checking as in Java – has attracted great interest. The precision of that information has considerable practical implications: An over-cautious programmer, who declares too many exceptions in **throws** clauses, makes his methods more heavy-weight to call. A tool which analyzes exception flow in Java source code [15], (and, in particular, detects exceptions handled through subsumption) demonstrated that commercial packages tend to handle exceptions at too coarse a grain. A constraint system [21] gives an exception flow analysis algorithm independent of the programmer's specifications. This problem has also attracted research in languages where the counterpart of **throws** clauses does not exist. Control-flow analysis and effects systems have been applied for the estimation of uncaught exceptions in SML programs [22, 14].

Exceptions introduce the potential for non-determinism. For example, the expression $\mathtt{e}_1$ **and** $\mathtt{e}_2$ is not equivalent to $\mathtt{e}_2$ **and** $\mathtt{e}_1$, because $\mathtt{e}_1$ and $\mathtt{e}_2$ might raise different exceptions, one of which might be caught. Such considerations could restrict language implementations, *i.e.* expressions could not be re-arranged unless it can be proven that they raise no exceptions. In [9] a semantics dealing with *imprecise* exceptions based on the `IO` monad in Haskell is suggested. This approach is not directly applicable to Java because of the presence of the `IO` monad, but the issue it tackles is applicable to any language.

The question of imprecise exceptions becomes even more pertinent for Java, because the language deliberately under-specifies the time and exact nature of loading and linkage-related errors, although it does place some constraints, *e.g.* ch. 12.1.2 in [8]. A semantics for this part of the Java language, which would characterize a whole family of non-deterministic implementations and their properties, is an interesting open task.

Acknowledgments

This paper is an account of the treatment of exceptions in [6] supported by EPSRC. We are grateful to anonynous referee for apt and useful comments which allows us to simplify our type system and make our operational semantics smooth. We also received valuable feedback from Susan Eisenbach, Mark Skipper and David Wragg.

References

[1] Egon Börger and Wolfram Schulte. A Programmer Friendly Modular Definition of the Semantics of Java. In Jim Alves-Foss, editor, *Formal Syntax and Semantics of Java*, LNCS. Springer, 1998.

[2] Egon Börger and Wolfram Schulte. A Practical Method for Specification and Analysis of Exception Handling – A Java/JVM Case Study. *IEEE Transactions of Software Engineering*, 2000. To appear in the Special Issue on Exception Handling, eds. D.Perry, A.Romanovsky, A.Tripathi, TSE 2000.

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

[4] Sophia Drossopoulou and Susan Eisenbach. Describing the Semantics of Java and Proving Type Soundness. In Jim Alves-Foss, editor, *Formal Syntax and Semantics of Java*, LNCS. Springer, 1998.

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

[6] Sophia Drossopoulou, Tatyana 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] Simon Peyton Jones, Alastair Reid, Tony Hoare, Simon Marlow, and Fergus Henderson. A semantics for imprecise exceptions. In *Proceedings of the SIGPLAN Symposium on Programming Language Design and Implementation*, May 1999.

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

[11] 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.

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

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

[14] François Pessaux and Xavier Leroy. Type-based analysis of uncaught exceptions. In *Proceedings of 26th ACM Conference on Principles of Programming Languages*, January 1999.

[15] Martin P. Robillard and Gail C. Murphy. Analyzing Exception Flow in Java Programs. In *Proceedings of the European Software Engineering Conference-Foundations of Software Engineering*, September 1999.

[16] Bjarne Strousrup. *The C++ Programming Language.* Addison Wesley, 1991.

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

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

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

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

[21] Kwangkeun Yi and Byeong-Mo Chang. Exception Analysis for Java. In Bart Jacobs, Gary T. Leavens, Peter Mueller, and Arnd Poetzsch-Heffter, editors, *Proceedings of the ECOOP-Workshop on Formal Techniques for Java Programs*, June 1999.

[22] Kwangkeun Yi and Sukyoung Ryu. Towards a cost-effective estimation of uncaught exceptions in SML programs. In *Proceedings of the 4th International Static Analysis Conference*, volume 1302 of *Lecture Notes in Computer Science.* Springer Verlag, 1997.

$$\frac{\mathcal{D}(\mathtt{P}) \vdash \mathtt{P}\ \diamond}{\vdash^{b}\ \mathtt{P}\ \diamond}$$

$$\frac{\begin{array}{l}\mathtt{D} \vdash \mathcal{D}(\mathtt{P})\ \diamond \\ \forall \mathtt{C}\quad \mathit{CDef}(\mathtt{P},\mathtt{C}) \neq \mathit{Undef} \quad \Longrightarrow \quad \mathtt{D} \vdash^{b}\ \mathit{CDef}(\mathtt{P},\mathtt{C})\ \diamond\end{array}}{\mathtt{D} \vdash^{b}\ \mathtt{P}\ \diamond}$$

$$\frac{\begin{array}{l}\mathtt{meth} \in \mathit{MDef}(\mathtt{P},\mathtt{C},\mathtt{m}) \quad \mathit{with} \quad \mathtt{meth} = \mathtt{T}\ \mathtt{m}(\mathtt{T_1}\ \mathtt{p_1},\ldots,\mathtt{T_n}\ \mathtt{p_n})\ \mathbf{throws}\ \mathtt{E_1},\ldots,\mathtt{E_q}\ \{\mathtt{mBody}\} \\ \quad \Longrightarrow \quad \mathtt{D} \vdash \mathtt{C}\ \mathbf{this};\ \mathtt{T_1}\ \mathtt{p_1};\ldots;\mathtt{T_n}\ \mathtt{p_n};\ \diamond \\ \qquad\qquad \exists \mathtt{T'}, \exists \mathtt{ET'}\quad \mathtt{D},\ \mathtt{C}\ \mathbf{this};\ \mathtt{T_1}\ \mathtt{p_1};\ldots;\mathtt{T_n}\ \mathtt{p_n};\ \vdash^{b} \mathtt{mBody}\ :\ \mathtt{T'} \parallel \mathtt{ET'} \quad \text{and} \\ \qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad \mathtt{D} \vdash \mathtt{T'} \parallel \mathtt{ET'}\ \leq_w\ \mathtt{T} \parallel \{\mathtt{E_1},\ldots,\mathtt{E_q}\}\end{array}}{\mathtt{D} \vdash^{b}\ \mathit{CDef}(\mathtt{P},\mathtt{C})\ \diamond}$$

$$\frac{\begin{array}{l}\vdash\ \mathtt{D}\ \diamond_u \\ \mathtt{D'} \vdash \mathtt{D}\ \diamond_a \\ \forall \mathtt{I}\quad \mathit{IDecl}(\mathtt{D},\mathtt{I}) \neq \mathit{Undef} \quad \Longrightarrow \quad \mathtt{D'} \vdash \mathit{IDecl}(\mathtt{D},\mathtt{I})\ \diamond \\ \forall \mathtt{C}\quad \mathit{CDecl}(\mathtt{D},\mathtt{C}) \neq \mathit{Undef} \quad \Longrightarrow \quad \mathtt{D'} \vdash \mathit{CDecl}(\mathtt{D},\mathtt{C})\ \diamond\end{array}}{\mathtt{D'} \vdash \mathtt{D}\ \diamond}$$

$$\frac{\begin{array}{l}\mathtt{I'} \in \mathit{Interfs}(\mathtt{D},\mathtt{I}) \quad \Longrightarrow \quad \mathtt{D} \vdash \mathtt{I'} \leq \mathtt{I'} \\ \mathtt{methH} \in \mathit{MDecl}(\mathtt{D},\mathtt{I},\mathtt{m}) \\ \quad \Longrightarrow \quad \mathtt{D} \vdash \mathit{RetT}(\mathtt{methH})\ \diamond_{NorType}, \\ \qquad\qquad \mathtt{D} \vdash \mathit{ArgT}(\mathtt{methH})\ \diamond_{ArgType}, \\ \qquad\qquad \mathtt{D} \vdash \mathit{ExcT}(\mathtt{methH})\ \diamond_{AbnType} \\ \mathtt{methH_1} \in \mathit{MDecl}(\mathtt{D},\mathtt{I},\mathtt{m}) \quad \text{and} \quad \mathtt{methH_2} \in \mathit{MDecl}(\mathtt{D},\mathtt{I},\mathtt{m}) \quad \text{and} \quad \mathtt{methH_1} \neq \mathtt{methH_2} \\ \quad \Longrightarrow \quad \mathit{ArgT}(\mathtt{methH_1}) \neq \mathit{ArgT}(\mathtt{methH_2}) \\ \forall \mathtt{m}, \forall \mathtt{I'} \in \mathit{Interfs}(\mathtt{D},\mathtt{I}), \\ \quad \mathtt{methH} \in \mathit{MDecl}(\mathtt{D},\mathtt{I},\mathtt{m}),\ \mathtt{methH'} \in \mathit{Meths}(\mathtt{D},\mathtt{I'},\mathtt{m}),\ \mathit{ArgT}(\mathtt{methH}) = \mathit{ArgT}(\mathtt{methH'}) \\ \quad \Longrightarrow \quad \mathit{RetT}(\mathtt{methH}) = \mathit{RetT}(\mathtt{methH'}),\ \mathtt{D} \vdash \mathit{ExcT}(\mathtt{methH})\ \subseteq_e\ \mathit{ExcT}(\mathtt{methH'})\end{array}}{\mathtt{D} \vdash \mathit{IDecl}(\mathtt{D},\mathtt{I})\ \diamond}$$

$$\frac{\begin{array}{l}\mathtt{C'} = \mathit{SuperC}(\mathtt{D},\mathtt{C}) \quad \Longrightarrow \quad \mathtt{D} \vdash \mathtt{C'} \sqsubseteq \mathtt{C'} \\ \mathtt{I} \in \mathit{Interfs}(\mathtt{D},\mathtt{C}) \quad \Longrightarrow \quad \mathtt{D} \vdash \mathtt{I} \leq \mathtt{I} \\ \mathtt{T}\ \mathtt{f} = \mathit{FDecl}(\mathtt{D},\mathtt{C},\mathtt{f}) \quad \Longrightarrow \quad \mathtt{D} \vdash \mathtt{T}\ \diamond_{VarType} \\ \mathtt{methH} \in \mathit{MDecl}(\mathtt{D},\mathtt{C},\mathtt{m}) \quad \mathit{with} \quad \mathtt{methH} = \mathtt{T}\ \mathtt{m}(\mathtt{T_1}\ \mathtt{p_1},\ldots,\mathtt{T_n}\ \mathtt{p_n})\ \mathbf{throws}\ \mathtt{E_1},\ldots,\mathtt{E_s}; \\ \quad \Longrightarrow \quad \mathtt{D} \vdash \mathtt{T}\ \diamond_{NorType}, \\ \qquad\qquad \mathtt{D} \vdash \mathtt{T_1} \times \ldots \times \mathtt{T_n}\ \diamond_{ArgType}, \\ \qquad\qquad \mathtt{D} \vdash \{\mathtt{E_1},\ldots,\mathtt{E_s}\}\ \diamond_{AbnType} \\ \mathtt{methH_1} \in \mathit{MDecl}(\mathtt{D},\mathtt{C},\mathtt{m}) \quad \text{and} \quad \mathtt{methH_2} \in \mathit{MDecl}(\mathtt{D},\mathtt{C},\mathtt{m}) \quad \text{and} \quad \mathtt{methH_1} \neq \mathtt{methH_2} \\ \quad \Longrightarrow \quad \mathit{ArgT}(\mathtt{methH_1}) \neq \mathit{ArgT}(\mathtt{methH_2}) \\ \forall \mathtt{m}\quad \mathtt{methH} \in \mathit{MDecl}(\mathtt{D},\mathtt{C},\mathtt{m}),\ \mathtt{methH'} \in \mathit{Meths}(\mathtt{D},\mathtt{C'},\mathtt{m}),\ \mathit{ArgT}(\mathtt{methH}) = \mathit{ArgT}(\mathtt{methH'}) \\ \quad \Longrightarrow \quad \mathit{RetT}(\mathtt{methH}) = \mathit{RetT}(\mathtt{methH'}),\ \mathtt{D} \vdash \mathit{ExcT}(\mathtt{methH})\ \subseteq_e\ \mathit{ExcT}(\mathtt{methH'}) \\ \forall \mathtt{m}, \forall \mathtt{I} \in \mathit{Interfs}(\mathtt{D},\mathtt{C}),\ \mathtt{methH_1} \in \mathit{MDecl}(\mathtt{D},\mathtt{I},\mathtt{m}) \\ \quad \Longrightarrow \quad \exists \mathtt{methH_2} \in \mathit{Meths}(\mathtt{D},\mathtt{C},\mathtt{m}),\ \mathit{ArgT}(\mathtt{methH_1}) = \mathit{ArgT}(\mathtt{methH_2}), \\ \qquad\qquad \mathit{RetT}(\mathtt{methH_1}) = \mathit{RetT}(\mathtt{methH_2}),\ \mathtt{D} \vdash \mathit{ExcT}(\mathtt{methH_2})\ \subseteq_e\ \mathit{ExcT}(\mathtt{methH_1})\end{array}}{\mathtt{D} \vdash \mathit{CDecl}(\mathtt{D},\mathtt{C})\ \diamond}$$

Figure 7: Well-formed and complete Java^{b} programs

Appendix

A1. Well-formed and complete Java$^{\mathsf{b}}$ programs

We outline the requirements that make Java$^{\mathsf{b}}$ programs well-formed and complete, concentrating on the issues around exceptions. The requirements are formalized in figure 7.

A Java$^{\mathsf{b}}$ program is well-formed and complete, *i.e.* $\vdash^{\mathsf{b}} \mathtt{P} \diamond$, if it is well-formed in the context of *its* declarations, *i.e.* if $\mathcal{D}(\mathtt{P}) \vdash \mathtt{P} \diamond$. P is well-formed in the context of some declarations D, *i.e.* $\mathtt{D} \vdash \mathtt{P} \diamond$, if the declarations of P are well-formed, *i.e.* $\mathtt{D} \vdash \mathcal{D}(\mathtt{P}) \diamond$, and if all class definitions in P are well-formed in D, *i.e.* if $\mathtt{D} \vdash^{\mathsf{b}} \mathit{CDef}(\mathtt{P},\mathtt{C}) \diamond$ for all C defined in P.

Well-formedness of class definitions requires all methods

$$\mathtt{meth{=}T\ m(T_1\ p_1,...,T_n\ p_n)}\ \textbf{throws}\ \mathtt{E_1,...,E_q\ \{mBody\}}$$

to be well-formed, and the method body mBody to be of type $\mathtt{T}' \parallel \mathtt{ET}'$ where $\mathtt{T}'$ widens to the declared return type T, and $\mathtt{ET}'$ consists of unchecked exceptions or subclasses of those declared in the **throws** clause. These two requirements are formalized as $\mathtt{D} \vdash \mathtt{T}' \parallel \mathtt{ET}' \ \leq_w\ \mathtt{T} \parallel \{\mathtt{E_1},\ldots,\mathtt{E_q}\}$.

Well-formedness of declarations imposes several restrictions, *e.g.* unique declarations of classes, interfaces and fields, *i.e.* $\vdash\ \mathtt{D} \diamond_u$, acyclic subclass and subinterface hierarchies, *i.e.* $\mathtt{D}' \vdash \mathtt{D} \diamond_a$. With respect to exceptions it requires the **throws** clause of any method header methH to consist of unchecked exceptions or of subclasses of the **throws** clause of any method header methH′ it overrides. In other words (*c.f.* figure 7, the fifth rule, last requirement), if $\mathtt{methH} \in \mathit{MDecl}(\mathtt{D},\mathtt{I},\mathtt{m})$ – *i.e.* methH belongs to interface I – and $\mathtt{methH}' \in \mathit{Meths}(\mathtt{D},\mathtt{I}',\mathtt{m})$ with $\mathtt{I}' \in \mathit{Interfs}(\mathtt{D},\mathtt{I})$ and $\mathit{ArgT}(\mathtt{methH}) = \mathit{ArgT}(\mathtt{methH}')$ – *i.e.* methH′ is declared in a superinterface and has the same identifier and argument type as methH and therefore is overridden by methH – then $\mathit{ExcT}(\mathtt{methH}) \subseteq_e \mathit{ExcT}(\mathtt{methH}')$. Similar requirements are for classes (C) and their methods ($\mathtt{methH} \in \mathit{MDecl}(\mathtt{D},\mathtt{C},\mathtt{m})$) in the last rule of figure 7.

Appendix

A2. Example Execution

As an example of our subject reduction consider program $\mathtt{P_{wr}}$ and the term peter.[]act() in a state σ_{10}, where

$\sigma_{10}(\mathtt{peter}) \;=\; \iota_{11}$
$\sigma_{10}(\iota_{11}) \;=\; \ll \mathtt{age\ Person:}\ 0 \gg^{\mathtt{Person}}$

Disregarding the possibility for unchecked exceptions like null dereferencing, *etc.*, there are three possible outcomes, namely:

σ_{11}	–	no exception thrown,
σ_{14}	–	an Illness exception thrown and caught,
throw ι_{13}, σ_{13}	–	a Worry exception thrown.

where the states are:

$\sigma_{11} = \sigma_{10}[\iota_{11} \mapsto \ll \mathtt{age\ Person:}\ 1 \gg^{\mathtt{Person}}]$
$\sigma_{12} = \sigma_{10}[\iota_{12} \mapsto \ll \mathtt{severity\ Illness:}\ 0 \gg^{\mathtt{Illness}}]$
$\sigma_{13} = \sigma_{12}[\iota_{13} \mapsto \ll\gg^{\mathtt{Worry}}]$
$\sigma_{14} = \sigma_{12}[\iota_{12} \mapsto \ll \mathtt{severity\ Illness:}\ {-10} \gg^{\mathtt{Illness}}]$

The following table demonstrates execution of the term peter.[]act() in more detail. We group consecutive executions in marked blocks. At the end of block 1, depending on the outcome of the condition, execution may continue at block 2 or 3. Similarly, at the end of block 3 execution may continue at block 4 or 5.

Block 1 – start

peter.[]act(), σ_{10}
$\rightsquigarrow_{\mathtt{P_{wr}}}$
ι_{11}.[]act(), σ_{10}
$\rightsquigarrow_{\mathtt{P_{wr}}}$
try { if (...) **throw** diagnose().[]treat(); **else** age=age + 1; }
catch(Illness i){ i.[]cure(); }, σ_{10}
$\rightsquigarrow_{\mathtt{P_{wr}}}$ continue at 2) or 3)

Block 2 – the condition returns false

try {age=age + 1; }
catch(Illness i){ i.[]cure(); }, σ_{10}
$\rightsquigarrow^{*}_{\mathtt{P_{wr}}}$
try {age=1; }
catch(Illness i){ i.[]cure(); }, σ_{10}
$\rightsquigarrow_{\mathtt{P_{wr}}}$
σ_{11}

Block 3 – the condition returns true

try { **throw** diagnose().[]treat(); }
catch(Illness i){ i.cure(); }, σ_{10}
$\rightsquigarrow^{*}_{\mathtt{P_{wr}}}$
try { **throw** ι_{12}().[]treat(); }
catch(Illness i){ i.[]cure(); }, σ_{12}
$\rightsquigarrow^{*}_{\mathtt{P_{wr}}}$
try { **throw** (**if** (...)
    throw new Worry();
    else return this;)}
catch(Illness i){ i.[]cure(); }, σ_{12}
$\rightsquigarrow_{\mathtt{P_{wr}}}$ continue at 4) or 5)

Block 4 – the condition returns false

try { **throw this**;}
catch(Illness i){ i.[]cure(); }, σ_{12}
$\rightsquigarrow_{\mathtt{P_{wr}}}$
try { **throw** ι_{12};}
catch(Illness i){ i.[]cure(); }, σ_{12}
$\rightsquigarrow^{*}_{\mathtt{P_{wr}}}$
ι_{12}.[]cure();, σ_{12}
$\rightsquigarrow^{*}_{\mathtt{P_{wr}}}$
σ_{14}

Block 5 – the condition returns true

try { **throw** (**throw new** Worry();)}
catch(Illness i){ i.[]cure(); }, σ_{12}
$\rightsquigarrow_{\mathtt{P_{wr}}}$
try { **throw** (**throw** ι_{13};)}
catch(Illness i){ i.[]cure(); }, σ_{13}
$\rightsquigarrow_{\mathtt{P_{wr}}}$
try { **throw** ι_{13};}
catch(Illness i){ i.[]cure(); }, σ_{13}
$\rightsquigarrow_{\mathtt{P_{wr}}}$
throw ι_{13}, σ_{13}

Indeed, as stated in the subject reduction theorem (in section 6), for $\mathtt{V_{wr}}$ and $\mathtt{P_{wr}}$ the final s-tates describing the *three* possible outcomes of the term rewriting we have:

$\mathtt{P_{wr}},\mathtt{V_{wr}} \vdash \sigma_{11}\ \diamond$,
$\mathtt{P_{wr}},\mathtt{V_{wr}} \vdash \sigma_{14}\ \diamond$,
$\mathtt{P_{wr}},\mathtt{V_{wr}} \vdash$ **throw** $\iota_{13}, \sigma_{13}\ \lhd\ \bot\ ||\ \{\mathtt{Worry}\}$ and
$\mathtt{P_{wr}} \vdash \bot\ ||\ \{\mathtt{Worry}\}\ \leq_w$ **void** $||\ \{\mathtt{Worry,\ Illness}\}$.