JOURNAL OF OBJECT TECHNOLOGY
Published by AITO — Association Internationale pour les Technologies Objets, © JOT 2011
Online at `http://www.jot.fm`.

# Strong exception-safety for checked and unchecked exceptions

Giovanni Lagorio[a] Marco Servetto [a]

a. DISI - University of Genova
Via Dodecaneso, 35
16146 Genova, Italy

**Acknowledgments** This work has been partially supported by MIUR DISCO - Distribution, Interaction, Specification, Composition for Object Systems.

Abstract **"Exception-safety strong guarantee: The operation has either completed successfully or thrown an exception, leaving the program state exactly as it was before the operation started."** *David Abrahams* [Abr00]
The above definition of strong exception-safety comes from the world of C++, but it can be applied to any language.

Because the *exception-safety strong guarantee* plays a central role in easing the development of robust software, we have designed a type-system able to capture its essence. The idea is that the state of the reachable objects at the beginning of a `catch` block is the same as the beginning of the corresponding `try` block.

We present a lightweight type system for Java-like languages that, by introducing a simple modifier to types, enforces that programs satisfy the strong guarantee in the presence of *checked* and *unchecked* exceptions.

## 1 Introduction

We illustrate the problem with an example in Java: suppose we need to model a pub serving beers to customers; the method `serve`, shown in Figure 1, takes a `Customer c`, as argument, and performs this task. The method first takes the money from `c`, and then it serves a beer. In the case there are no more beers to serve, the exception `NoBeersException` is thrown. When the customer does not have enough money, the exception `NotEnoughMoneyException` is thrown by method `getMoneyFrom`. Because the availability of beer is checked (lines 5–6) after the money has been taken (line 4), this implementation actually *steals* money when there is no beer left.

```
1 class Pub {
2   void serve(Customer c) throws NoBeersException,
3                                 NotEnoughMoneyException {
4     getMoneyFrom(c);
5     if (beers.isEmpty())
6       throw new NoBeersException();
7     serveBeerTo(c, beers.pop());
8   }
9 // ...
```

Figure 1 – Class `Pub` (wrong version).

In this simple example the problem is easy to spot and fix: moving the check (lines 5–6) at the beginning of the method would fix the issue, as shown in Figure 2. However, in more complex scenarios, especially when exceptions are simply propagated, some subtle interactions can pass unnoticed.

We would like to avoid these situations without being too restrictive: throwing an exception, like `NoBeersException`, is the right thing to do when an error is encountered; the problem is *when* the exception is thrown by method `serve`. When a method execution fails by throwing an exception, its clients should see no changes in the reachable objects. This is the spirit of the exception-safety strong guarantee, quoted in the abstract. To clarify the idea we need to understand, first of all, what an *operation* is. Because the first statement actually executed after an exception has been thrown is the matching catch block, the right granularity for an operation is the try-catch block, as detailed later on.

A possible, but rather expensive, approach to solve this problem is to enhance the language with a transaction construct, following the commit/rollback pattern, as it happens in the database world. However, such an approach looks impractical for general purpose programming languages, because of the implied computational costs and the impossibility of rolling back most I/O operations. Moreover, we do not want to alter the well-known semantics of Java, but simply rule out suspicious behaviours.

Our idea is to allow a method to throw any checked exception (obviously, declared in the method `throws` clause) as long as there are no visible side effects for clients. Note that a method is allowed to create and modify as many objects as it likes, as long as these are not connected to the objects that are visible to clients.

A well-know C++ idiom, *copy-and-swap*[1], obtains this semantics by calculating the result in a temporary variable and swapping it with the actual result at the very end (this works when the swap operation is guaranteed not to throw any exception; for more details on the exception safety of the C++ standard library see the Appendix E of [Str97]).

We present a type system, which can be added on top of any Java-like

[1]See, for instance, `http://en.wikibooks.org/wiki/More_C++_Idioms/Copy-and-swap`

```
class Pub {
  void serve(Customer c) throws NoBeersException,
                                NotEnoughMoneyException {
    if (beers.isEmpty())
      throw new NoBeersException();
    getMoneyFrom(c);
    serveBeerTo(c, beers.pop());
  }
// ...
```

Figure 2 – Class `Pub` (correct version).

language, that enforces that programs satisfy the exception-safety strong guarantee by generalizing the idea of the *copy-and-swap* idiom: methods can do any operation until they modify some part of the heap reachable by their clients and, after that, they are forbidden to throw any checked exception.

This enforces methods to check for error conditions first, ensuring clients that there are no (visible) side effects when the execution of a method throws a *checked* exception. Indeed, in Java there are two kinds of exceptions: *checked* and *unchecked* exceptions.

The former represent exceptional conditions that well-written applications should anticipate and recover from. A method that may throw a checked exception $C$ must declare $C$ in the throws clause, and any client must either catch $C$, via a try statement, or, in turn, declare $C$ in its own throws clause.

The latter kind of exceptions represent exceptional conditions that applications usually cannot anticipate and recover from, these kinds of exceptions can be thought of as observable bugs. In Java these unchecked exceptions are further split into (external) *errors* and (internal) runtime exceptions, but we do not model this distinction because it is immaterial.

We expect the proposed approach to be applicable to the full Java language in single-threaded programs; however, this paper only formalizes the above idea on a minimal Java subset that includes constructs having non trivial interactions with exception safety; in particular, we do not model inheritance and casts.

The model presented in this paper is an extension of our model for checked exceptions [LS10], with the addition of unchecked exception handling and let expressions.

The paper is structured as follows: Section 2 describes our formalization and sketches the proof of soundness, while Section 3 concludes and considers some extensions that are subject of further work.

$$
\begin{array}{lll}
p & ::= & \overline{D} \\
D & ::= & \texttt{class}\ C\ \{\overline{\mathit{field}}\ \ \overline{\mathit{meth}}\} \\
\mathit{field} & ::= & T\ f; \\
\mathit{meth} & ::= & T\ m(\overline{T\ x})\ \mathit{modif}\ \texttt{throws}\ \mathit{Tr}\ \{\texttt{return}\ e;\} \\
\mathit{Tr} & ::= & \overline{C} \\
e & ::= & x \mid o \mid e.m(\overline{e}) \mid e.f \mid e_1.f = e_2 \mid T\ x = e_1; e_2 \\
 & & \mid \texttt{new}\ C(\overline{e}) \mid \texttt{throw}\ e \mid \texttt{try}\ e\ \texttt{catch}(C\ x)\ e' \\
T & ::= & \mathit{modif}\ C \\
\mathit{modif} & ::= & \texttt{rw} \mid \texttt{ro}
\end{array}
$$

Figure 3 – Syntax

## 2 Our approach

We have designed a simple type system, which can be added on top of any Java-like language, and instantiated it over a language inspired by Featherweight Java [IPW01] and similar calculi. The formalized language, shown in Figure 3, is a minimal imperative class-based language without inheritance and casts but including two simple constructs for exception handling: `throw` and `try-catch`.

The inclusion of inheritance and casts in the model should be quite natural and not particularly interesting. The only non-standard feature to be checked is that a method overriding a read-only `ro`-method (described below) must be declared `ro` as well.

For the sake of simplicity, `try` blocks are followed by exactly one `catch` clause. Programs $p$, as usual, consist of a sequence of class declaration $D$; the overbar notation indicates a (possibly empty) sequence, that is, $\overline{D}$ is an abbreviation for $D_1 \dots D_n$.

Class declarations $D$ consist of a class name $C$, followed by a sequence of field and method declarations. Field declarations *field* are standard, while method declarations *meth* include a modifier *modif*, discussed below, before the `throws`-clause. A `throws`-clause contains a sequence of class names, *Tr*, which corresponds to the checked exceptions that can be thrown by the method. In this language the class `Throwable` is not modelled, so we simply assume the set of classes to be partitioned into three kinds: non-throwable, checked and unchecked classes. Method bodies consist of a single expression. Expressions $e$ can be: arguments $x$, object identifiers o[2], method invocations, field accesses, field assignments, let expressions, instance creations, `throw` expressions or `try-catch` expressions.

A let expression $T\ x = e_1; e_2$ introduces the local variable $x$ of type $T$, initialized by the value of $e_1$, for evaluating expression $e_2$, which is the result of the whole let expression.

Types $T$ consist of an *access modifier*, *modif*, which can be either `ro` (read-only) or `rw` (read-write), and a class name $C$. The access modifier `ro` has the same meaning as `readonly` of Javari [TE05], that is, the

[2]For the sake of simplicity we do not distinguish between source expressions, that is, the ones that can appear in the source code, and runtime expressions, which are a superset of the former.

read-only property of a reference $r$ propagates to the whole object graph reachable by $r$. This semantics is called *transitivity of constness* in [Boy06]. This modifier is similar to `const` of C++, but in C++ the constness of a pointed/referenced object $o$ does not propagate to the objects pointed/referenced by $o$. In other words, C++ const is shallow.

Accessing a field $f$ through a `ro`-reference yields a `ro`-reference, regardless of the declared access modifier for $f$. Trying to modify a field through a `ro`-reference is, obviously, forbidden.

On a `ro`-reference only `ro`-methods, that is, methods annotated with a `ro` modifier, can be invoked; these methods receive the (implicit) parameter `this` as a `ro`-reference, so they cannot modify the state of the object itself or of any of the (directly or indirectly) referenced ones.

The `rw`-references correspond to usual references, as in Java: they can be written to, and the result of accessing a field on a `rw`-reference yields the type (class name and modifier) of the corresponding field declaration.

Note that modifier `ro` does not correspond to `final` of Java; indeed, `final` deals with the *assignability* concept [TE05], which is unrelated to exception handling. So, we do not model it even if doing that would not be hard.

During method invocations we logically split the heap into two parts: the *client connected heap*, which consists of all (directly or indirectly) reachable objects starting from the target of the invocation or any of its arguments, and the *unconnected heap*, which consists of all other objects, including those created during the execution of the invocation and not connected (yet?) to the first ones.

In the same way, when executing a try-catch statement, we can again split the heap into the *connected heap*, that is, the reachable heap existing before the execution of the try block, and the *unconnected heap*, which includes all other objects.

The intuitive idea is that if an exception is thrown and propagated, either outside a method body or a try block, no changes to the heap should be observable.

## Reduction

Reduction and typing rules refer implicitly to a program $p$. We indicate with $p(C)$ the declaration of class $C$ in $p$, and with $p(C.m)$ or $p(C.f)$ the declaration of the member $m$ or $f$ in $p(C)$. Moreover, the auxiliary function `mBody`$(C.m)$ retrieves the parameter list and body of method $m$ in $C$.

The reduction is standard and given by a small-step semantics, where:

$$\mu \mid e \longrightarrow \mu' \mid e'$$

has the meaning "the reduction of expression $e$, in a heap $\mu$, produces an expression $e'$ and a (possibly) updated heap $\mu'$".

In order to model unchecked exceptions, (UNCHKD) allows, non deterministically, to rewrite *any* expression into the throw of an `OutOfMemory` object, which we use to model the canonical example of an unchecked exception.

A heap $\mu$ maps object identifiers $o$ to object states $os$:

$$\text{(INVK)}\quad \frac{}{\mu \mid o.m(\overline{o}) \longrightarrow \mu \mid e'} \quad \begin{array}{l} \mu(o) = \texttt{new}\ C(\ldots) \\ \texttt{mBody}(C.m) = \overline{x}, e \\ e' = e[o/\texttt{this}, \overline{o}/\overline{x}] \end{array}$$

$$\text{(NEW)}\quad \frac{}{\mu \mid \texttt{new}\ C(\overline{o}) \longrightarrow \mu' \mid o} \quad \begin{array}{l} o \notin dom(\mu) \\ \mu' = \mu, o \mapsto \texttt{new}\ C(\overline{o}) \end{array}$$

$$\text{(FIELD)}\quad \frac{}{\mu \mid o.f_i \longrightarrow \mu \mid o_i} \quad \begin{array}{l} \mu(o) = \texttt{new}\ C(o_1 \ldots o_n) \\ p(C) = \{T_1\ f_1 \ldots T_n\ f_n \overline{meth}\} \end{array}$$

$$\text{(ASSIGN)}\quad \frac{}{\mu \mid o.f_i = o' \longrightarrow \mu' \mid o'} \quad \begin{array}{l} p(C) = \{C_1\ f_1 \ldots C_n\ f_n\ \overline{meth}\} \\ \mu = \mu'', o \mapsto \texttt{new}\ C(o_1 \ldots, o_i, \ldots o_n) \\ \mu' = \mu'', o \mapsto \texttt{new}\ C(o_1 \ldots, o', \ldots o_n) \end{array}$$

$$\text{(T-EXIT)}\quad \frac{}{\mu \mid \texttt{try}\ o\ \texttt{catch}(C\ x)\ e \longrightarrow \mu \mid o} \qquad \text{(LET-IN)}\quad \frac{}{\mu \mid T\ x = o; e \longrightarrow \mu \mid e[o/x]}$$

$$\text{(T-CATCH)}\quad \frac{}{\mu \mid \texttt{try}\ \texttt{throw}\ o\ \texttt{catch}(C\ x)\ e \longrightarrow \mu \mid e[o/x]} \quad \mu(o) = \texttt{new}\ C(\ldots)$$

$$\text{(T-MISS)}\quad \frac{}{\mu \mid \texttt{try}\ \texttt{throw}\ o\ \texttt{catch}(C\ x)\ e \longrightarrow \mu \mid \texttt{throw}\ o} \quad \begin{array}{l} \mu(o) = \texttt{new}\ C'(\ldots) \\ C \neq C' \end{array}$$

$$\text{(T-PROP)}\quad \frac{\mu \mid e_1 \longrightarrow \mu' \mid e_1'}{\mu \mid \texttt{try}\ e_1\ \texttt{catch}(C\ x)\ e_2 \longrightarrow \mu' \mid \texttt{try}\ e_1'\ \texttt{catch}(C\ x)\ e_2}$$

$$\text{(UNCHKD)}\quad \frac{}{\mu \mid e \longrightarrow \mu' \mid \texttt{throw}\ o'} \quad \begin{array}{l} o' \notin dom(\mu) \\ \mu' = \mu, o' \mapsto \texttt{new OutOfMemory}() \end{array}$$

$$\text{(CTX)}\quad \frac{\mu \mid e \longrightarrow \mu' \mid e'}{\mu \mid \mathcal{E}\{e\} \longrightarrow \mu' \mid \mathcal{E}\{e'\}} \qquad \text{(PROP)}\quad \frac{}{\mu \mid \mathcal{E}\{\texttt{throw}\ o\} \longrightarrow \mu \mid \texttt{throw}\ o}$$

$$\begin{array}{lcl} \mathcal{E} & ::= & \square \mid \mathcal{E}.m(\overline{e}) \mid o.m(\overline{o}, \mathcal{E}, \overline{e}) \mid \texttt{new}\ C(\overline{o}, \mathcal{E}, \overline{e}) \mid \mathcal{E}.f \\ & & \mid \mathcal{E}.f = e \mid o.f = \mathcal{E} \mid \texttt{throw}\ \mathcal{E} \mid T\ x = \mathcal{E}; e \end{array}$$

Figure 4 – Reduction rules.

$$\begin{array}{lcl} E & ::= & \texttt{mod} \mid P \\ P & ::= & \texttt{safe} \mid \texttt{danger} \\ \Gamma & ::= & \overline{x{:}T\ P} \\ \Sigma & ::= & \overline{o{:}C\ P} \end{array}$$

Figure 5 – Types and typing environments

$$\begin{array}{lcl} \mu & ::= & \overline{o \mapsto os} \\ os & ::= & \texttt{new}\ C(\overline{o}) \end{array}$$

The reduction rules are shown in Figure 4.

Contextual closure is standard except for the `try-catch` expressions, which are handled by their specific rules (T-PROP), (T-MISS), (T-EXIT) and (T-CATCH).

## Typing

Types and typing environments are shown in Figure 5.

Evaluation effects $E$ can be `mod` to indicate that the evaluation of an expression may produce (visible) side effects, or $P$ that indicates that the client connected heap is preserved. The effect $P$ is split into two sub-cases: `danger` indicates that the result value is a (possibly indirect) *writable* link to the connected heap, `safe` otherwise.

A variable environment $\Gamma$ maps variables $x$ to types $T$, while a memory environment $\Sigma$ maps each object identifier to its corresponding class name. Both $\Gamma$ and $\Sigma$ mark elements with a flag $P$ indicating whether an element represents a (possibly indirect) *writable* link to the connected heap.

Memory environments are only needed for the proofs, since method bodies are typechecked in an empty memory environment.

The type judgement for typing heaps has the form $\Sigma \vdash \mu\ ;\ \mu'$, with the meaning: "the connected heap $\mu$ and the unconnected heap $\mu'$ are well typed w.r.t. the memory environment $\Sigma$".

Typing of expressions is expressed by the judgement:

$$\Gamma; \Sigma; \mathit{Tr} \vdash e : T\ E$$

with the meaning "expression $e$ has type $T$ and evaluation effect $E$, in a variable environment $\Gamma$, memory environment $\Sigma$ and in a context where the list of throwable exceptions is $\mathit{Tr}$".

Obviously, unchecked exceptions can be thrown at any moment. Moreover, the evaluation of an expression, regardless of its effect, can always throw checked exceptions; if the effect is `mod`, then the type system ensures that the checked exception has been thrown before any connected heap has been changed.

Typing rules are shown in Figures 6 to 8.

Rule (SUB-T) models subsumption, assuming the following order relations for effects and types: `safe` $\leq$ `danger` $\leq$ `mod`[3] and, for any $C$, `rw` $C \leq$ `ro` $C$.

[3]Indeed, `safe` is stronger than `danger` since `safe` means that the visible state has not been changed (as in `danger`) and the result is unconnected.

$$(\text{SUB-T})\ \dfrac{\Gamma;\Sigma;\mathit{Tr}\vdash e : T\ E}{\Gamma;\Sigma;\mathit{Tr}\vdash e : T'\ E'}\ T\ E \leq T'\ E' \qquad (\text{VAR-T})\ \dfrac{}{\Gamma;\Sigma;\mathit{Tr}\vdash x : \Gamma(x)}$$

$$(\text{ADDR-T})\ \dfrac{}{\Gamma;\Sigma;\mathit{Tr}\vdash o : T\ P}\ \begin{array}{l}\Sigma(o) = C\ P\\ T = \texttt{rw}\ C\end{array}$$

$$(\text{INVK-T})\ \dfrac{\begin{array}{l}\Gamma;\Sigma;\mathit{Tr}\vdash e_k : T_k\ E_k\ \ \forall k \in [0..i]\\ \Gamma;\Sigma;[]\vdash e_k : T_k\ E_k\ \ \forall k \in (i..n]\end{array}}{\Gamma;\Sigma;\mathit{Tr}\vdash e_0.m(e_1 \ldots e_n) : T'E}\ \begin{array}{l}T_0 = \mathit{modif}\ C\\ p(C.m) =\ T'\ m(T_1\,x_1 \ldots T_n\,x_n)\ \mathit{modif}\\ \qquad\qquad\qquad \texttt{throws}\ \mathit{Tr}'\ \{\texttt{return}\ \_;\}\\ i \in [0..n]\\ E_j \neq \texttt{mod}\text{ if } j \in [0..i)\\ E = \begin{cases}\texttt{safe} & \text{if } E_i = \texttt{safe}\ \forall i \in [0..n]\\ \texttt{mod} & \text{otherwise}\end{cases}\end{array}$$

$$(\text{NEW-T})\ \dfrac{\begin{array}{l}\Gamma;\Sigma;\mathit{Tr}\vdash e_k : T_k\ E_k\ \ \forall k \in [1..i]\\ \Gamma;\Sigma;[]\vdash e_k : T_k\ E_k\ \ \forall k \in (i..n]\end{array}}{\Gamma;\Sigma;\mathit{Tr}\vdash \texttt{new}\ C(e_1 \ldots e_n) : \texttt{rw}\ C\ E}\ \begin{array}{l}p(C) = \{T_1\ f_1 \ldots T_n\ f_n\ \overline{\mathit{meth}}\}\\ i \in [1..n]\\ E_j \neq \texttt{mod}\text{ if } j \in [1..i)\\ E = \begin{cases}\texttt{safe} & \text{if } E_i = \texttt{safe}\ \forall i \in [1..n]\\ \texttt{mod} & \text{otherwise}\end{cases}\end{array}$$

$$(\text{THROW-T})\ \dfrac{\Gamma;\Sigma;\mathit{Tr}\vdash e : \mathit{modif}\ C\ E}{\Gamma;\Sigma;\mathit{Tr}\vdash \texttt{throw}\ e : T'\ \texttt{safe}}\ \begin{array}{l}E \neq \texttt{mod}\text{ if }\texttt{checked}(C)\\ (C \in \mathit{Tr}\text{ or }\texttt{unchecked}(C))\end{array}$$

$$(\text{TRY-T})\ \dfrac{\begin{array}{l}\Gamma;\Sigma;\mathit{Tr},C_2\vdash e_1 : T\ E_1\\ \Gamma[\texttt{danger}];\Sigma[\texttt{danger}];\mathit{Tr},C_2\vdash e_1 : T\ \_\\ \Gamma,x_2{:}\texttt{ro}\ C_2\ \texttt{danger};\Sigma;\mathit{Tr}\vdash e_2 : T\ E_2\end{array}}{\Gamma;\Sigma;\mathit{Tr}\vdash \texttt{try}\ e_1\ \texttt{catch}(C_2\ x_2)\ e_2 : T\ E}\ \begin{array}{l}\texttt{checked}(C_2)\\ E = \mathit{max}(E_1,E_2)\end{array}$$

$$(\text{TRY-U-T})\ \dfrac{\begin{array}{l}\Gamma;\Sigma;\mathit{Tr}\vdash e_1 : T\ P_1\\ \Gamma[\texttt{danger}];\Sigma[\texttt{danger}];\mathit{Tr}\vdash e_1 : T\ P\\ \Gamma,x_2{:}\texttt{ro}\ C_2\ \texttt{danger};\Sigma;\mathit{Tr}\vdash e_2 : T\ E_2\end{array}}{\Gamma;\Sigma;\mathit{Tr}\vdash \texttt{try}\ e_1\ \texttt{catch}(C_2\ x_2)\ e_2 : T\ E}\ \begin{array}{l}\texttt{unchecked}(C_2)\\ E = \mathit{max}(P_1,E_2)\end{array}$$

Figure 6 – Typing rules (1)

(FIELD-C-T)
$$\frac{\Gamma; \Sigma; \mathit{Tr} \vdash e : \texttt{ro}\ C\ E}{\Gamma; \Sigma; \mathit{Tr} \vdash e.f : \texttt{ro}\ C'\ E} \quad p(C.f) = \mathit{modif}\ C'\ f;$$

(FIELD-T)
$$\frac{\Gamma; \Sigma; \mathit{Tr} \vdash e : \texttt{rw}\ C\ E}{\Gamma; \Sigma; \mathit{Tr} \vdash e.f : T\ E} \quad p(C.f) = T\ f;$$

(ASSIGN-T)
$$\frac{\begin{array}{l}\Gamma; \Sigma; \mathit{Tr} \vdash e_1 : \texttt{rw}\ C\ P_1 \\ \Gamma; \Sigma; \mathit{Tr} \vdash e_2 : T\ E_2\end{array}}{\Gamma; \Sigma \vdash e_1.f = e_2 : T\ E} \quad \begin{array}{l} p(C.f) = T\ f; \\ E = \begin{cases} \texttt{safe} & \text{if } P_1 = \texttt{safe} \text{ and } E_2 = \texttt{safe} \\ \texttt{mod} & \text{otherwise} \end{cases}\end{array}$$

(ASSIGN-M-T)
$$\frac{\begin{array}{l}\Gamma; \Sigma; \mathit{Tr} \vdash e_1 : \texttt{rw}\ C\ \texttt{mod} \\ \Gamma; \Sigma; [] \vdash e_2 : T\ E\end{array}}{\Gamma; \Sigma; \mathit{Tr} \vdash e_1.f = e_2 : T\ \texttt{mod}} \quad p(C.f) = T\ f;$$

(LET-T)
$$\frac{\begin{array}{l}\Gamma; \Sigma; \mathit{Tr} \vdash e_1 : T_1\ P_1 \\ \Gamma, x_1 : T_1\ P_1; \Sigma; \mathit{Tr} \vdash e_2 : T_2\ E_2\end{array}}{\Gamma; \Sigma; \mathit{Tr} \vdash T\ x_1 = e_1; e_2 : T_2\ E} \quad E = \mathit{max}(P_1, E_2)$$

(LET-M-T)
$$\frac{\begin{array}{l}\Gamma; \Sigma; \mathit{Tr} \vdash e_1 : T_1\ \texttt{mod} \\ \Gamma, x_1 : T_1\ \texttt{danger}; \Sigma; [] \vdash e_2 : T_2\ \texttt{mod}\end{array}}{\Gamma; \Sigma; \mathit{Tr} \vdash T\ x_1 = e_1; e_2 : T_2\ \texttt{mod}}$$

Figure 7 – Typing rules (2)

(HEAP-T)
$$\frac{\begin{array}{l}\Sigma \vdash \mu(o) : \Sigma(o) \;\; \forall o \in dom(\mu)\\ \Sigma \vdash \mu'(o) : \Sigma(o) \;\; \forall o \in dom(\mu')\\ \Sigma \vdash o : \_\ \texttt{danger} \;\; \forall o \in dom(\mu)\\ \Sigma \vdash o : \_\ \texttt{safe} \;\; \forall o \in dom(\mu')\end{array}}{\Sigma \vdash \mu \;;\; \mu'} \quad \begin{array}{l} dom(\Sigma) = dom(\mu, \mu')\\ wf(\mu;\mu')\end{array}$$

(OBJ-T)
$$\frac{}{\Sigma \vdash \texttt{new}\ C(o_1 \ldots o_n) : C\ \_} \quad \begin{array}{l} p(C) = \{\_\ C_1\ f_1 \ldots \_\ C_n\ f_n\ \overline{meth}\}\\ \forall i \in [1..n]\ \Sigma(o_i) = C_i\ \_\end{array}$$

(CLASS-T)
$$\frac{C \vdash meth_i : \text{OK} \;\; \forall meth_i \in p(C)}{\vdash C : \text{OK}}$$

(METH-T)
$$\frac{\Gamma; \emptyset; Tr \vdash e : T\ \_}{C \vdash meth : \text{OK}} \quad \begin{array}{l} meth = T\ m(T_1\ x_1, \ldots, T_n\ x_n)\ modif\ \texttt{throws}\ Tr\ \{\texttt{return}\ e;\}\\ \Gamma = \texttt{this}{:}modif\ C, x_1{:}T_1\ \texttt{danger}, \ldots, x_n{:}T_n\ \texttt{danger}\end{array}$$

Figure 8 – Typing rules (3)

This intuitive notion is extended to pairs of types and effects as follows:

$$\begin{array}{ccccc}
\texttt{ro}\ C\ \texttt{safe} & = & \texttt{ro}\ C\ \texttt{danger} & \leq & \texttt{ro}\ C\ \texttt{mod} \\
\rotatebox{90}{$\leq$} & & \rotatebox{90}{$\leq$} & & \rotatebox{90}{$\leq$} \\
\texttt{rw}\ C\ \texttt{safe} & \leq & \texttt{rw}\ C\ \texttt{danger} & \leq & \texttt{rw}\ C\ \texttt{mod}
\end{array}$$

Note that `ro safe` and `ro danger` are equivalent; this means, for instance, that a `rw danger` reference can be passed as argument where a `ro safe` is required.

Rules (VAR-T) and (ADDR-T) are standard.

Rule (INVK-T) typechecks a method invocation, considering that the receiver is evaluated first, and then the arguments are evaluated from left to right.

The evaluation order is particularly important because we must find an argument that is the last allowed to throw exceptions $Tr$, and the first allowed to modify the (connected) heap.

We express this non-deterministic choice by finding one $i$ such that the $i^{\texttt{th}}$ argument meets the above conditions, see the premises of the rule.

The following arguments are typechecked in an empty $Tr$; that is, after the heap has been changed, no checked exceptions can be propagated. In order to infer the effect $E$ for the whole method invocation, the list of all effects $E_i$, $i \in [0..n]$ are considered: when all effects are `safe`, then $E = \texttt{safe}$, otherwise $E = \texttt{mod}$. Since $\texttt{ro}\ C\ \texttt{safe} = \texttt{ro}\ C\ \texttt{danger}$ this rule allows the type checker to infer the effect `safe` in many cases.

Rule (NEW-T) is analogous to the rule (INVK-T); while in the current limited model we could use a more liberal rule, we chose to use the same strategy used for method invocations because in Java constructors are allowed to execute arbitrary code.

The rule (THROW-T) typechecks a `throw` expression; unchecked exceptions can be thrown at any time; a checked exception $C$ can be thrown only if it is contained in $Tr$ and the effect is not `mod`. We use the predicates `checked` and `unchecked` to identify classes representing checked and unchecked exceptions, respectively. We assume that those two predicates are disjoint.

The rules (TRY-T) and (TRY-U-T) typecheck a `try` block for checked and unchecked exceptions, respectively. In both cases we require that the try body can be typed in a typing environment where all elements are marked with `danger`, indicated by $\Gamma[\texttt{danger}]$ and $\Sigma[\texttt{danger}]$, because this guarantees that if any exception is thrown and caught by the catch, then its body $e_2$ will see the same state originally seen by $e_1$ (indeed, since $e_1$ sees the whole heap as `danger`, it cannot change any already existing object *and* throw a checked exception afterwards).

In any case, the resulting effect for the whole expression is inferred using the original type environment. Since unchecked exceptions can be thrown at any moment, rule (TRY-U-T) requires $e_1$ to leave the state unchanged.

The rules (FIELD-C-T) and (FIELD-T) typecheck a field assignment for a `ro` target or `rw` target, respectively. Note that the result of accessing a field on a `ro` target has type `ro`, regardless the field modifier.

An assignment expression $e_1.f = e_2$ is well typed only if $e_1$ is of type `rw` $C$, and $e_2$ and the field $f$ in $C$ have type $T$. When $e_1$ has effect $P_1$ (ASSIGN-T), the evaluation of $e_2$ is allowed to throw. When $e_1$ has effect `mod` (ASSIGN-M-T), the evaluation of $e_2$ is not allowed to propagate checked exceptions or catch unchecked exceptions.

Note that rule (ASSIGN-T) yields a `mod` effect when expression $e_2$, whose effect is `danger`, is assigned to a writable field $f$ of $e$. This is needed to avoid that modifications of objects in the connected heap, made through other references, pass unnoticed.

Note that modeling the `final` modifier for fields would amount to checking, in these two rules, that $p(C.f)$ is not final.

The rules (LET-T) and (LET-M-T) typecheck a let expression; as it happens for assignments, when $e_1$ has effect `mod` (LET-M-T), the evaluation of $e_2$ is not allowed to propagate or catch unchecked exceptions.

Rule (HEAP-T) typechecks, w.r.t $\Sigma$, a well-formed pair of connected heap $\mu$ and unconnected heap $\mu'$, that is, $wf(\mu;\mu')$. This predicate holds when:

$$dom(\mu) \cap dom(\mu') = \emptyset$$
$$\mu(o) = \texttt{new } C(\ldots, o_i, \ldots) \text{ implies } o_i \in dom(\mu)$$
$$\mu'(o) = \texttt{new } C(\ldots, o_i, \ldots) \text{ implies } \begin{cases} o_i \in dom(\mu') \\ \text{or} \\ o_i \in dom(\mu) \text{ and } p(C.f_i) = \texttt{ro \_} \end{cases}$$

Rule (OBJ-T) is used by (HEAP-T) to check the consistency of a single object.

Finally, (CLASS-T) checks a single class by checking all its methods via (METH-T), that checks that the method body $e$ is well-typed in an environment when all its argument are `danger` (that is, the worst-case scenario). Note that the effect of evaluating the body is immaterial, since it will be inferred depending on the invocation context (that is, different effects can be inferred for different invocations of the same method).

## Typing other constructs

Since the formalized language does not contain statement sequences, `if` statements and loops, we would like to informally discuss how they could be handled by a simple extension, and consider an example that uses them.

Sequences of expressions/statements could be typed, as mentioned before, in the same way as argument sequences.

Since the execution of an `if` statement corresponds to the execution of its guard followed by the execution of one of its branches, the effect of `if (`$e$`)` $s$ `else` $s'$ can be obtained by considering the super-effect (that is, the less specific[4]) of the following two sequences: `{`$e$`;`$s$`}` and `{`$e$`;`$s'$`}`. This is, of course, an over-approximation, due to the fact that we cannot statically know which branch will be executed.

Analogously, the execution of a loop, say `while (`$e$`)` $s$, corresponds to the execution of the guard $e$ at least once, followed by an arbitrary number (possibly zero) of executions of the statement $s$ and the guard

[4]In the formalization we have used the function $max$ to denote the less specific effect.

$e$. So, the effect of the `while`-loop can be obtained by considering the super-effect of the sequence `{`$e$`;`$s$`;`$e$`;`$s$`}` (repeating both $e$ and $s$ twice in order to over-approximate the global effect of an arbitrary number of loop executions).

More sophisticated static analysis/verification techniques could be applied, to both branching and looping constructs, to obtain better approximations in some cases.

## Example revisited

Now that we have discussed the type system, we explain why the correct implementation of the `Pub` example, shown in Figure 2, would be typeable (and why the original version, Figure 1, would not).

We assume that there is an extension of our typesystem handling statements, where each type or method without an access modifier is implicitly considered `rw`.

Under the following assumptions:

$$
\begin{array}{l}
\texttt{Beers} \vdash \texttt{boolean isEmpty() ro \{...\}} : \text{OK} \\
\texttt{Pub} \vdash \ \texttt{void getMoneyFrom(Customer c) ro} \qquad : \text{OK} \\
\qquad\quad \texttt{throws NotEnoughMoneyException \{...\}} \\
\texttt{Beers} \vdash \texttt{Beer pop() \{...\}} : \text{OK} \\
\texttt{Pub} \vdash \texttt{void serveBeerTo(Customer c, ro Beer b) ro \{...\}} : \text{OK}
\end{array}
$$

the method `serve` would be well typed (without requiring any `ro` modifier).

$$
\dfrac{
\begin{array}{l}
\ldots;\texttt{NoBeersException} \vdash \texttt{if(beers.isEmpty())} \qquad : \texttt{safe} \\
\qquad\qquad\qquad\qquad\quad \texttt{throw new NoBeersException();} \\
\ldots;\texttt{NotEnoughMoneyException} \vdash \texttt{getMoneyFrom(c);} : \texttt{mod} \\
\ldots;\emptyset \vdash \texttt{serveBeerTo(c,beers.pop());} : \texttt{mod}
\end{array}
}{
\begin{array}{l}
\texttt{Pub} \vdash \ \texttt{void serve(Customer c) throws NoBeersException,} \ : \text{OK} \\
\qquad\quad \texttt{NotEnoughMoneyException \{...\}}
\end{array}
}
$$

Indeed, the `if` statement is allowed to throw `NoBeerException`. Then, method `getMoneyFrom` has no restrictions (the simple fact that it is well typed guarantees that if it throws `NotEnoughMoneyException`, then the state is still unchanged).

Finally, method `serveBeerTo` is forbidden to throw any exception since it is executed after method `getMoneyFrom`, which has already changed the state.

Instead, in the original example, the initial invocation of method `getMoneyFrom` would make the following `if` statement ill typed. Indeed, recall that sequences of statements would be typed as sequences of arguments, so, the statements following one having effect `mod` could not throw checked exceptions.

## Another example: how does the Pub serve meals?

Let us introduce, into class `Pub`, the new method `process`, shown in Figure 9, that takes a meal order `order`, consisting of a sequence of `MenuItems`,

```
 1 rw DishList process(ro MealOrder order) rw
 2                               throws MissingIngredientException {
 3   rw Ingredients ingr=new Ingredients();
 4   for(ro MenuItem mi : order)
 5     ingr.addAll(mi.getIngredients());
 6   this.checkAvailability(ingr);
 7   this.kitchen.removeAll(ingr);
 8   return prepare(order);
 9 }
10
11 void checkAvailability(ro Ingredients i) ro
12                               throws MissingIngredientException {
13   /* ... */
14 }
15
16 rw DishList prepare(ro MealOrder o) ro {
17   /* ... */
18 }
```

Figure 9 – Method `process`

and processes it by: finding the needed ingredients (lines 3–5), checking their availability (line 6), removing them from the kitchen (line 7) and, finally, preparing the requested dishes (line 8). The method is annotated with `rw` because the state of the pub needs to be changed in order to remove the ingredients from the kitchen. The annotation of the return type is, in this case, arbitrary.

The parameter `order` is `ro`, so it cannot be modified.

The effect of creating and initializing the local variable `ingr` is `safe`; indeed, the initialization expression is `safe` by (NEW-T), so the following statements "see" a new variable `ingr` of type `rw Ingredients safe` by (LET-T).

The method `addAll` is invoked on an unconnected target and (we assume that) its parameter is declared `ro`; in these settings the effect of the method invocation is `safe` (by rule (INVK-T) in Figure 6, with $i = 1$ because the argument is typed `safe`). Ignoring, for simplicity, the effects of the iteration variable `mi`, the global effect of the `for`-loop is `safe`.

The auxiliary method `checkAvailability`[5] ensures that all needed ingredients `ingr` are available, throwing `MissingIngredientException` when this is not the case. The effect of invoking `checkAvailability` is `safe`, since both the method and its parameter are `ro`.

The invocation of `removeAll`, on the field `this.kitchen`, has effect `mod`.

Finally, the method returns the requested dishes by invoking the aux-

[5]The implementation of methods `checkAvailability` and `prepare`, and the declaration of the field `kitchen` and its method `removeAll`, have been omitted for the sake of brevity.

```
class TestRunner {
  enum Result { OK, FAIL };

  Vector<Result> run() ro {
    Vector<Result> results = new Vector<Result>();
    for(Test test : this.allTests)
      results.add(this.runSingleTest(test));
    return results;
  }

  Result runSingleTest(ro Test test) ro {
    try {
      test.run(); // run must preserve the (visible) heap
    } catch (AssertionError ae) {
      return Result.FAIL;
    } catch (Throwable t) {
      return test.shouldThrow(t) ? Result.OK:Result.FAIL;
    }
    return test.shouldThrow() ? Result.FAIL:Result.OK;
  }
}
```

Figure 10 – Class `TestRunner`

iliary method `prepare`. This method invocation is not allowed to throw any checked exception, since the invocation of `removeAll` has changed the (client visible) state.

### Unchecked exception example: a test runner

It seldom makes sense to catch unchecked exceptions; one example where this is actually useful is when writing a test runner, that is, a program which needs to control the execution of test methods, which may contain "arbitrary" (and probably wrong) code.

In the class `TestRunner`, shown in Figure 10, an extremely simplified test runner, the method `run` runs a series of tests, contained in the instance variable `allTests`, and collects their results into the local variable `results`.

If the execution of a test throws `AssertionError`, then the test fails, otherwise we must consider different cases depending whether the test is supposed to throw some exception (this fact is encoded in the auxiliary overloaded method `shouldThrow`). Let us discuss how this class would be typed; method `run` initializes the local variable `results` with a new object, so the resulting effect would be `safe`. Then, the `for` loop updates such a variable by invoking `add`, which modifies an unconnected object. The argument expression has also effect `safe`, since the method `runSingleTest` receives two `ro` references: the implicit argument `this` and the local variable `test`. Therefore, the loop is well-typed since the effect of its body is `safe`.

The interesting method is `runSingleTest`, containing a `try` block that catches the unchecked exceptions `AssertionError` and the special exception `Throwable`. In Java `Throwable` and `Exception` are special since the roots of unchecked exceptions, `Error` and `RuntimeException`, are their direct subclasses.

To integrate those special exceptions in our model we would have to typecheck their catch with the rule (TRY-U-T). Such a rule is applicable in this case since the body of the `try` consists of an expression having effect `safe`.

## Exception-safety strong guarantee

To formally encode the Abrahams's quote (shown at the beginning of this paper), we need to understand: what an *operation* is, what it means to *complete successfully/throw a (checked or unchecked) exception* and what the *program state* is.

A single reduction step is not meaningful as an *operation*; this concept is usually mapped to method invocations. However, this intuitive solution is not adequate since there is no way to guarantee the property in the presence of unchecked exceptions. Moreover, it is irrelevant to have a corrupted state if no part of the program can observe it. So, it is important to guarantee that the state has been preserved when an exception is *caught*.

Therefore, we choose to map the *operation* concept to the `try` block, which has the right granularity. With this perspective, the focus is on *propagating* an exception instead of on *throwing* an exception. In particular, *to complete successfully* means to propagate no exceptions, and *to throw an exception* means to *propagate* an exception outside the `try` block.

So, *leaving the program state exactly as it was before the operation started* corresponds to having the state of the reachable objects at the beginning of a `catch` block as it was at the beginning of the corresponding `try` block. This is formally expressed by Theorem 2.

**Lemma 1 (substitutability)**
$\overline{x{:}T\ \texttt{danger}}; \emptyset; Tr \vdash e : T'\ \texttt{mod}$ *implies* $\emptyset; \overline{o{:}T\ \texttt{safe}}; Tr \vdash e[\overline{o}/\overline{x}] : T'\ \texttt{safe}$

**Lemma 2 (pure-preservation)**
*If* $\Sigma \vdash \mu\ ;\ \mu'$ *and* $\emptyset; \Sigma; Tr \vdash e : \_\ \texttt{danger}$*, then the expression* $e$ *preserves the heap* $\mu$*, that is,*

$$\mu, \mu' \mid e \xrightarrow{\star} \mu'' \mid e' \ \textit{implies}\ \mu \subseteq \mu''$$

*The intuition is that* $\mu'$ *can be modified (and new objects can be created and modified), but everything that was already in* $\mu$ *is not affected. We say that an expression produces side effects on a heap* $\mu$ *if it is not preserved.*

**Proof** Sketch: the statement is equivalent to proving that $\Sigma \vdash \mu \; ; \mu'$ and $\emptyset; \Sigma; \mathit{Tr} \vdash e : T$ `danger`, then $\mu, \mu' \mid e \longrightarrow \mu, \mu'' \mid e'$ implies $\Sigma' \vdash \mu \; ; \mu''$ and $\emptyset; \Sigma'; \mathit{Tr} \vdash e' : T$ `danger`.

By induction over the depth of the expression $e$. *Base (for depth* 0 *and* 1*)*; by case analysis.

One interesting case is $o.m(\overline{o})$ with $p(C.m)$; we know that the target and all arguments have been typed with effect `safe`, so the resulting expression contains no writable links to the client connected heap and is typed `safe` so the invocation can be safely replaced by the well-typed method body by Lemma 1.

*Inductive step (depth* $= k, \; k > 1$*).*

Similar to the inductive step of Lemma 3.

**Lemma 3 (subject-reduction)**

*If* $\Sigma \vdash \mu_1 \; ; \mu_2, \quad \emptyset; \Sigma; \mathit{Tr} \vdash e : T \; E, \quad \emptyset; \Sigma; [] \not\vdash e : T \; E$

*and* $\mu_1, \mu_2 \mid e \longrightarrow \mu' \mid e'$

*then* $\mu' = \mu_1, \mu_2', \quad \Sigma' \vdash \mu_1 \; ; \mu_2'$ *and* $\emptyset; \Sigma'; \mathit{Tr} \vdash e' : T \; E$

**Proof** Sketch: by induction over the depth of the expression $e$. *Base (for depth* 0 *and* 1*)*; by case analysis. Most cases are trivial, since they type with $\mathit{Tr} = []$; for instance, `new` $C(o)$.

One interesting case is $o.m(\overline{o})$ with $p(C.m)$ with a non-empty throw clause; there are two possibilities: either the target and all arguments have been typed with effect `safe`, or some of them is `mod`. In both cases, the invocation can be safely replaced by the well-typed method body. In the former case the resulting expression contains no writable links to the client connected heap and is typed `safe` by Lemma 1, in the latter case $o.m(\overline{o})$ is typed `mod`.

*Inductive step (depth* $= k, \; k > 1$*).* Cases (INVK), (NEW), (FIELD) and (ASSIGN) never occur: their depth is less or equal than 1. Cases (TRY-EXIT) , (TRY-MISS) and (PROP) are trivial. Cases (LET-IN) and (TRY-CATCH) are similar to the case of method invocation, where the expression can be safely replaced by the well-typed $e_2$ by Lemma 1.

In case (UNCHKD), $\mu_1, \mu_2$ is not modified, since a new `OutOfMemory` object is allocated into a fresh location $o$; this resulting heap is trivially well-typed. Moreover, $e' =$ `throw` $o$ is well-typed by (THROW-T) and (ADDR-T).

Cases (CTX) and (T-PROP) are analogous; we only detail (CTX). From (CTX) $e$ is of the form $\mathcal{E}\{e_1\}$, $e'$ is of the form $\mathcal{E}\{e_2\}$ and $\mu_1, \mu_2 \mid e_1 \longrightarrow \mu' \mid e_2$. By case analysis over the shape of the context; we show only one case.

**[**`new` $C(\overline{o}, e_1, \overline{e})$**]** if the whole expression can be typed with $\mathit{Tr} = []$, then the case trivially holds.

If $\overline{e}$ cannot be typed with $\mathit{Tr} = []$, then by rule (NEW-T) the expression $e_1$ has effect $P$ and we can close by Lemma 2.

Otherwise, $e_1$ itself cannot be typed with $\mathit{Tr} = []$, so if it has effect $P$ we can close by Lemma 2. If it has effect `mod`, knowing that the depth of $e_1$ is less or equal than $k$, by inductive hypothesis we know that $e_2$ is typed with the same type and effect of $e_1$ so, by rule (NEW-T), we can conclude.

**Lemma 4 (throw-runs)**

*If* $\Sigma \vdash \mu\ ;\ \mu'$, $\emptyset; \Sigma; \mathit{Tr} \vdash e : \texttt{mod}$ *and* $\mu, \mu' \mid e \xrightarrow{\star} \mu'' \mid \texttt{throw}\ o$ *then* $\mu''(o) = \texttt{new}\ C(\ldots)$ *and* $C \in \mathit{Tr}$ *or* $\texttt{unchecked}(C)$.

**Theorem 1 (Checked exception-safety)**

*If* $\Sigma \vdash \mu\ ;\ \mu'$ *and* $\emptyset; \Sigma; \mathit{Tr} \vdash e : T\ E$, *then the expression* $e$ *is safe w.r.t. the heap* $\mu$, *that is,*

$$\mu, \mu' \mid e \xrightarrow{\star} \mu'' \mid \texttt{throw}\ o \textit{ and } \mu''(o) = \texttt{new}\ C(\ldots), \texttt{checked}(C) \quad \textit{implies } \mu \subseteq \mu''$$

**Proof** Sketch:

By induction on the number of steps of the reduction.

Base: if there are zero steps, then the heap is trivially preserved.

Induction step: if there are $n$ steps, then either:

- $\emptyset; \Sigma; [] \vdash e : T\ E$ can be derived, then from Lemma 4 we know that no checked exception can be thrown.
- otherwise, the property holds for the first step by Lemma 3, and for the $n-1$ following ones for the inductive hypothesis.

**Theorem 2 (Exception-safety strong guarantee)**

*If* $e = \texttt{try}\ e_1\ \texttt{catch}(C_2\ x_2)\ e_2$, $\Sigma \vdash \mu\ ;\ \mu'$ *and* $\Gamma; \Sigma; \mathit{Tr} \vdash e : T\ E$, *then the expression* $e$ *is safe w.r.t. the heap, that is,*

$$\mu, \mu' \mid e_1 \xrightarrow{\star} \mu'' \mid \texttt{throw}\ o \textit{ and } \mu''(o) = \texttt{new}\ C_2(\ldots) \quad \textit{implies } \mu, \mu' \subseteq \mu''$$

*where the arrow* $\xrightarrow{\star}$ *is the reflexive and transitive closure of the reduction arrow.*

**Proof** Sketch: If $\texttt{unchecked}(C_2)$ then $e$ has been typed by (TRY-U-T), so $e_1$ preserves the whole heap (second premise of the rule) by Lemma 2.

Otherwise, that is, if $\texttt{checked}(C_1)$, then $e$ has been typed by (TRY-T), so $e_1$ enjoys the checked exception safety property and we conclude by Theorem 1.

## 3 Conclusions and further work

This paper presents an extension of our model for checked exceptions [LS10], that adds the support for declaring local variables and handling unchecked exceptions. The main inspiration of this work is [Abr00], where the concept of strong safety w.r.t exceptions has been defined.

In the C++ community this concept is very popular [Str01], but it is neglected in other language communities. As a notable exception, Spec# offers instruments to grant some variations of the strong safety [LS04]. While their work makes heavy use of Spec#-specific concepts, ours is applicable to any Java-like language.

Both the exception mechanism [ALZ01] and the introduction of a const/readonly modifier in Java-like languages has already been studied; see, for instance, the language Javari [TE05], [HP09], or Boyland's work [Boy06] for a survey. However, in this paper, we are not interested in the modifier per-se, but we see it as a tool for enforcing that programs satisfy the exception-safety strong guarantee.

Li et al. [LHR06] describe an approach, which combines static analysis and model checking, to ensure that no resources are leaked even in the presence of exceptions.

Jacobs and Piessens [JP09] have proposed a language extension, called *failboxes*, that facilitates writing sequential or multithreaded programs that preserve intended safety properties without leaking resources. Our model does not consider multithreading and, as it is, with multithreading our proposed type checking cannot guarantee the "standard" definition of the exception safety strong guarantee. Indeed, when an exception is thrown, our type system can only guarantee that the *current thread* has not modified the client visible state; however, other threads may have already altered it.

Further work includes the treatment of "global variables" (that is, `static` fields), which could be accessed and/or modified by any method. This could be achieved by allowing methods to declare the fact that they intend to modify the global state via a modifier, similar to `ro` modifier used to indicate that a method is read-only.

Because our proofs are not obvious, we plan to encode them in a machine-checkable form, and include the treatment of a larger Java subset.

Once gained the confidence that there are no pitfalls in our reasoning, due to some unexpected feature interactions, we are going to implement our system to evaluate its applicability on real-world open-source projects.

We are interested, first of all, in seeing how much the existing code can be typed as it is (or with little changes). Then evaluating, in the part of code that cannot be typed, what is the source of the problem; that is, whether our type system is too restrictive or there is an actual bug.

For the implementation we plan to rely on some existing tools for extending an existing typechecker for Java, like the *Checker Framework*[6].

**Acknowledgments** We would like to warmly thank our reviewers for their careful work and helpful advice.

## References

[Abr00] David Abrahams. Exception-safety in generic components. In Mehdi Jazayeri, Rüdiger Loos, and David R. Musser, editors, *International Seminar on Generic Programming,Selected Papers*, volume 1766 of *Lecture Notes in Computer Science*, pages 69–79. Springer, 2000.

[ALZ01] Davide Ancona, Giovanni Lagorio, and Elena Zucca. A core calculus for Java exceptions. In *ACM SIGPLAN Conference*

[6]`http://types.cs.washington.edu/checker-framework/`

*on Object-Oriented Programming, Systems, Languages and Applications (OOPSLA 2001)*, SIGPLAN Notices. ACM Press, October 2001.

[Boy06] John Boyland. Why we should not add readonly to Java (yet). *Journal of Object Technology*, 5(5):5–29, 2006.

[HP09] Christian Haack and Erik Poll. Type-based object immutability with flexible initialization. In *ECOOP'09 - Object-Oriented Programming*, Lecture Notes in Computer Science. Springer, 2009.

[IPW01] Atsushi Igarashi, Benjamin C. Pierce, and Philip Wadler. Featherweight Java: a minimal core calculus for Java and GJ. *ACM Transactions on Programming Languages and Systems*, 23(3):396–450, 2001.

[JP09] Bart Jacobs and Frank Piessens. Failboxes: Provably safe exception handling. In Sophia Drossopoulou, editor, *ECOOP 2009 - Object-Oriented Programming, 23rd European Conference, Genoa, Italy, July 6-10, 2009. Proceedings*, volume 5653 of *Lecture Notes in Computer Science*, pages 470–494. Springer, 2009.

[LHR06] Xin Li, H. James Hoover, and Piotr Rudnicki. Towards automatic exception safety verification. In Jayadev Misra, Tobias Nipkow, and Emil Sekerinski, editors, *FM 2006: Formal Methods, 14th International Symposium on Formal Methods, Hamilton, Canada, August 21-27, 2006, Proceedings*, volume 4085 of *Lecture Notes in Computer Science*, pages 396–411. Springer, 2006.

[LS04] K. Rustan M. Leino and Wolfram Schulte. Exception safety for C#. In *2nd International Conference on Software Engineering and Formal Methods*, pages 218–227. IEEE Computer Society, 2004.

[LS10] Giovanni Lagorio and Marco Servetto. Strong exception-safety for Java-like languages. In *12th Intl. Workshop on Formal Techniques for Java-like Programs*, ACM International Conference Proceedings Series, 2010.

[Str97] Bjarne Stroustrup. *The C++ Programming Language: Third Edition*. Addison-Wesley Publishing Co., Reading, Mass., 1997.

[Str01] Bjarne Stroustrup. Exception safety: Concepts and techniques. In *Advances in Exception Handling Techniques*, Lecture Notes in Computer Science, pages 60–76. Springer, 2001.

[TE05] Matthew S. Tschantz and Michael D. Ernst. Javari: adding reference immutability to Java. *ACM SIGPLAN Notices*, 40(10):211–230, October 2005.