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 [1]

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.

We present a lightweight type system for Java-like languages that, by introducing a simple modifier to types, enforces programs to satisfy the strong guarantee.

Categories and Subject Descriptors
D3.3 [Software]: Programming languages—Language Constructs and Features

Keywords
Java, Exception handling, Type systems

1. INTRODUCTION

We illustrate the problem with an example in Java: suppose we need to model a pub serving beers to customers; the following method serve takes a Customer c, as argument, and performs this task.

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

The method first takes the money from c, and then it serves a beer. In the case the are no more beers to serve, the exception NoBeersException is thrown. When the customer does not have enough money, NotEnoughMoneyException is thrown by method getMoneyFrom. Because the availability of beer is checked (lines 6–7) after the money has been taken (line 5), this implementation actually steals money when there is no beer left.

In this simple example the problem is easy to spot and fix: moving the check (lines 6–7) at the beginning of the method would fix the issue. 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.

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 database world. However, such an approach looks impracticable for general purpose programming languages, because of the implied computational costs and the impossibility of rolling back most of I/O operations. Moreover, we do not want to alter the well-known semantics of Java, but simply ruling out suspicious behaviours.

Our idea is to allow a method to throw any 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-swap1, 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 [9]).

We present a type system, which can be added on top of any Java-like language, that enforces programs to 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

1See, for instance, http://en.wikibooks.org/wiki/More_C++_Idioms/Copy-and-swap
any 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 an exception.

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.

2. OUR APPROACH

We have designed a simple type system, which can be added on the top of any Java-like language, and instantiated it over a language inspired by Featherweight Java [5] and similar calculi. The formalized language, shown in Figure 1, is a minimal imperative class-based language without inheritance and casts but including some two 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 an 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, $D = \bar{D}$.

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 exceptions that can be thrown by the method. In this language the class Throwable is not modelled, so any object can be thrown as an exception. Furthermore, all exceptions are checked; Section 3 speculates on how unchecked exceptions could be handled as well.

Method bodies consist of a single expression\(^2\). Expressions e can be: arguments x, object identifiers o\(^3\), field accesses, \(^\text{throw} e\) expressions or \text{try-catch} expressions.

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 [11], that is, the read-only property of a reference r propagates to the whole object graph reachable by r. This semantics is called transitivity of constness in [3]. 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 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.

During method invocations we logically split the heap into two parts: the client connected heap, which consists in 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.

### Typing

Types and typing environments are shown in Figure 2. A variable environment $\Gamma$ maps variables x to types T, while a memory environment $\Sigma$ maps each object identifier to its corresponding class name and a flag indicating whether the object belongs to the client connected heap.

Memory environments are only needed for the proof of subject-reduction, 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:

$$\Gamma; \Sigma; Tr \vdash e : T E$$

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 $Tr$\(^\dagger\).

The evaluation effect mod indicates that the evaluation

\(^1\)Although sequences of expressions are not directly supported, the same expressive power (and typing problems) can be obtained by using methods with many parameters, since the sequence of arguments, in a method invocation, is guaranteed to be evaluated from left to right.

\(^2\)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.

\(^3\)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.
Γ; Σ; Tr ⊢ x : T clean

(NEW-T)
Γ; Σ; Tr ⊢ new C() : rw C unconn
with p(C) = \{meth\}

(NEW-K-T)
Γ; Σ; Tr ⊢ e_k : T_k E_k \forall k \in [0..n]

with
MT = (T_0, T_1, \ldots, T_n) \rightarrow (T_{n+1}, Tr')
i \in [0..n + 1]
Tr_j = Tr if j \in [0..i]
E_j \neq mod if j \in [0..i]
Tr_j = \emptyset if j \in (i..n + 1)

Γ; Σ; Tr ⊢ e_0.m(e_1 \ldots e_n) : T_{n+1}E_{n+1}

Figure 3: Typing rules (1)
may produce (visible) side effects, while an effect $P$ indicates that the client connected heap is preserved. The effect $P$ is split into two sub-cases: unconn when the result (being a value or a thrown exception) is an object identifier belonging to the unconnected heap, and clean otherwise.

The evaluation of an expression, regardless of its effect, can always throw an exception; if the effect is mod, then the type system ensures that the exception has been thrown before any client connected heap has changed.

Method types $MT$ have the following form: $(T_0, T_1 \ldots T_n) \rightarrow (T_{n+1}, Tr)$, where $T_0$ is the type of the (the implicit parameter) this (with the $mod$ modifier), $T_1 \ldots T_n$ are the type of the parameters, $T_{n+1}$ is the return type and $Tr$ are the exceptions the method may throw.

Typing and reduction rules refer implicitly to a program $p$; reduction rules also assume $p$ to be well-typed. 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 $C$.

Typing rules are shown in Figure 3 and Figure 4; because of lack of space, we detail some selected rules only.

Rule (SUB-T) models subsumption, assuming the following order relations for effects and types: unconn $\leq$ clean $\leq$ mod$^4$ and, for any $C$, $\mathbf{rw}$ $C$ $\leq$ $\mathbf{ro}$ $C$.

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 if the $i^{th}$ argument has the effect mod, which indicates that the (client connected) heap may be not preserved, then the evaluation of following arguments is forbidden to throw any exception, see the definition of $Tr_i$ in the side condition.

The rules (M-C-T-UNCONN), (M-C-T-CLEAN) and (M-C-T-MOD), are used, by (INVK-T), in order to infer the effect for the whole method invocation, considering the list of all effects of the receiver and the arguments, and the list of allowed exceptions. The resulting effect is unconn if no effect is mod and for all parameters declared rw, the corresponding effects are unconn; mod otherwise.

The rule (TRY-T) typechecks a try block; in this case, both the expression $e$ and the catch expression $e'$ need to be well typed with some effect $E$. Intuitively, $e'$ is executed only if $e$ throws an exception and, since $e$ is well-typed, the (client connected) heap is still unmodified when $e'$ is executed. Then, $e'$ can have any effect.

An assignment expression $e.f = e'$ is well typed only if $e$ and $e'$ are of types $\mathbf{rw}$ $C$ and $\mathbf{rw}$ $C'$, respectively, and the field $f$ in $C$ has type $\mathbf{rw}$ $C'$. When $e$ has effect clean (rule (ASSIGN-T1)), the evaluation of $e'$ is allowed to throw. When $e$ has effect mod (rule (ASSIGN-T2)), the evaluation of $e'$ is not allowed to throw. Finally, when both $e$ and $e'$ are unconnected (rule (ASSIGN-T3)), then the result is unconnected too.

Note that rule (ASSIGN-T1) yields a mod effect when expression $e'$, whose effect is clean, is assigned to a field $f$ of $e$, which has effect unconn. This is needed to avoid that modifications of objects in the client connected heap, made through references to objects originally belonging in the

$^4$Indeed, unconn is stronger than clean since unconn means that the visible state has not been changed (as in clean) and the result is unconnected.
connected heap, pass unnoticed.

**Example revisited**

Now that we have discussed the type system, we explain why the correct implementation of the example `Pub`, given in the introduction, would be typeable (and why the original example would not).

We assume to have an extension handling statements, where each type or method without an access modifier is implicitly considered `rw`.

Under the assumptions that method `isEmpty` does not modify the state and method `serveBeerTo` do not throw any exception (but modify the state), the method `serve` would be well typed. Indeed, the `if` statement\(^5\) is allowed to throw the exception `NoBeerException`. Then, method `getMoneyFrom` has no restrictions (the simple fact that it is well typed guarantees that if it throws the exception `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.

In the original example, instead, the initial invocation of method `getMoneyFrom` would make the following `if` statement, which contains a `throw` expression, ill typed.

**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 of 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 (s) e else e'` can be obtained by considering the super-effect (that is, the less specific) 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 `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 allows 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.

To clarify these sketched ideas, let us introduce, into class `Pub`, the new method `process` that takes a meal order `order`, consisting of a sequence of `MenuItems`, 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).

```java
rw DishList process(ro MealOrder order) rw
    throws MissingIngredientException {
        rw Ingredients ingr=new Ingredients();
        for (ro MenuItem mi : order) {
            ingr.addAll(mi.getIngredients());
            this.checkAvailability(ingr);
            this.kitchen.removeAll(ingr);
        }
        return prepare(order);
    }
```

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, obviously, `unconn`.

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 `clean` (see rule `(µ-C-T-CLEAN)` in Figure 4). Ignoring, for simplicity, the effects of the iteration variable `mi`, the global effect of the `for`-loop is `clean`.

The auxiliary method `checkAvailability`, which has already changed the state.

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

Finally, the method returns the requested dishes by invoking the auxiliary method `prepare`. This method invocation is not allowed to throw any exception, since the invocation of `removeAll` has changed the (client visible) state.

**Reduction**

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

\[
\mu [e \rightarrow \mu' | e']
\]

has the meaning “the reduction of expression `e`, in a heap `\mu`, produces an expression `e'` and a (possibly) updated heap `\mu'`\(^6\).

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

\[
\mu :: = \sigma \rightarrow o\sigma \quad os :: = \text{new } C(\pi)
\]

The reduction rules are shown in Figure 5.

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

\(^5\)That would be at the beginning in the correct implementation.

\(^6\)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.
Exception-safety strong guarantee

Theorem 1 (Exception-safety strong guarantee).

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

$$\mu, \mu' \vdash e \rightarrow \mu'' \mid \text{throw} \emptyset \implies \mu \subseteq \mu'$$

where the arrow $\rightarrow$ is the symmetric and transitive closure of the reduction arrow.

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 the thesis follows from Lemma 2.
- otherwise, the property holds for the first step for Lemma 1, and for the $n-1$ following ones for the inductive hypothesis.

□

Lemma 1 (Subject-reduction).

If $\Sigma \vdash \mu_1 ; \mu_2$, $\emptyset; \Sigma; \text{Tr} \vdash e : T E$, $\emptyset; \Sigma; [] \not\vdash e : T E$ and $\mu_1, \mu_2 \vdash e \rightarrow \mu' \mid e'$

then $\mu' = \mu_1, \mu_2$, $\Sigma' \vdash e_1 : T E$ and $\emptyset; \Sigma'; \text{Tr} \vdash e' : T E$

Proof. Sketch: By induction over reduction rules. We shows only one case.

ctx From (ctx) $e$ is of the form $\mathcal{E}\{e_1\}$. $e'$ is of the form $\mathcal{E}\{e_2\}$ and $\mu_1, \mu_2 \vdash e_1 \rightarrow \mu' \mid e_2$. Depending on the shape of the context, we shows only one case:

$$\text{[new } C(\pi, \mathcal{E}, \tau) \text{]} \text{ From (new-t1) we have three subcases depending on i:}$$

- $e_1, e \in \pi$; then $\emptyset; \Sigma; [] \not\vdash e : T E$ and $\forall e' \in \pi$; $\emptyset; \Sigma; [] \not\vdash e' : T E$ This is impossible since $\emptyset; \Sigma; [] \not\vdash e : T E$.
- $e_1 = e_2$: then the thesis holds for inductive hypothesis.
- $e_1, e \in \pi$: then $\emptyset; \Sigma; \text{Tr} \vdash e_1 : T' P$. We close by Lemma 3.

□

Lemma 2 (Throw-runs).

If $\Sigma \vdash \mu ; \mu'$, $\emptyset; \Sigma; \text{Tr} \vdash e : \text{mod}$ and $\mu, \mu' \mid e \rightarrow o$ then $\mu(\emptyset) = P \text{ C(.) and } C \in \text{Tr}$.

Proof. Trivial. □

Lemma 3 (Pure-preservation).

If $\Sigma \vdash \mu ; \mu'$ and $\emptyset; \Sigma; \text{Tr} \vdash e : \text{clean}$, then the expression $e$ preserves the heap $\mu$, that is,

$$\mu, \mu' \vdash e \rightarrow \mu'' \mid o \implies \mu \subseteq \mu''$$

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.

Figure 5: Reduction rules.
Proof. In the case $e$ is a method invocation, then all the parameters and this are either declared ro, i.e., the access of the pre-existing object-graph is read-only, or produced by unconn expressions, that is, have no access to the objects of $\mu$. Other cases of reduction consist only of sequences of method invocations, unconn expressions and operations that do not modify the state. \hfill \qed

3. CONCLUSIONS AND FURTHER WORK

The main inspiration of this work is [1], where the concept of strong safety w.r.t exceptions has been defined. In the C++ community this concept is very popular [10], but neglected in other language communities. As a notable exception, Spec# offers instruments to grant some variations of the strong safety [7]. While their work make heavy use of Spec#-specifics concepts, ours is applicable to any Java-like language.

Both the exception mechanism [2] and the introduction of a const/readonly modifier in Java-like languages has already been studied; see, for instance, the language Javari [11], [4], or Boyland’s work [3] 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 programs to satisfy the exception-safety strong guarantee.

Li et al. [8] describe an approach, which combines static analysis and model checking, to ensure that no resources are leaked even in presence of exceptions.

Jacobs and Piessens [6] 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, the 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.

We are studying how to extend our type system for handling also unchecked exceptions, like OutOfMemoryError or AssertionError. Such an extension would allow unchecked exceptions to be thrown at any point, so ensuring that the strong-safety continues to hold is challenging. The idea is to allow to catch unchecked exceptions only when the heap is still preserved (for instance, referring to rule (Invk-T), before the expression $e_i$).

One of the main problems is that a catch block, capturing an unchecked exception, could access inconsistent data structures, so we need to introduce strong restrictions on which variables and fields can be accessed in such situations. Indeed, if an unchecked exception is thrown (and caught) before the client visible state has been changed, then only the unconnected heap might contain objects in an inconsistent state. So, forbidding the catch block to access the unconnected heap would suffice to guarantee that no inconsistent objects are accessed or leaked to clients. Flow-analysis techniques could be employed to be less restrictive.

Because catch blocks, for unchecked exceptions, would be restricted to be in the non-mod part of any method, a thrown (unchecked) exception would climb up the calling stack until it finds a catch block of a client whose state has not been changed. If no such block is found, then the program would be terminated, consistently with the idea that no caller is left in a consistent situation and so no recovery is possible.

4. REFERENCES