Science of Computer Programming 87 (2014) 44–61

Contents lists available at ScienceDirect

Science of Computer Programming

www.elsevier.com/locate/scico

Modular type checking of anchored exception declarations

Marko van Dooren [*,1], Bart Jacobs, Wouter Joosen

DistriNet, Department of Computer Science, K.U. Leuven, Celestijnenlaan 200A, 3001 Leuven, Belgium

HIGHLIGHTS

- We present modular algorithms for type checking anchored exception declarations.
- The modular algorithms provide additional opportunities for using anchored exception declarations.
- We prove that the modular algorithms preserve exception safety.

ARTICLE INFO

Article history:
Received 4 November 2010
Received in revised form 8 March 2013
Accepted 28 October 2013
Available online 18 November 2013

Keywords:
Exception
Anchored exception declaration
Modular
Soundness

ABSTRACT

Checked exceptions improve the robustness of software, but they also decrease its adaptability because they must be propagated explicitly, and because they must often be handled even if they cannot be thrown. Anchored exception declarations solve both problems by allowing a method to declare its exceptional behavior in terms of other methods.

The original type checking analyses for anchored exception declarations, however, are not modular. In this paper, we present algorithms for modular verification of soundness in an object-oriented language without parametric polymorphism.

1. Introduction

Anchored exception declarations offer a solution to two important problems with checked exceptions [1]. With an anchored exception declaration **like** $e.m(\bar{e})$, a method n indicates that it can throw the exceptions that can be thrown by a call $e.m(\bar{e})$. The exceptions thrown by the referenced method m are thus propagated instead of being copy & pasted into the list of exceptions of method n. As a result, changes in the exceptional behavior of method m automatically apply to method n. In addition, the compiler can reduce the set of possible exceptions for a specific invocation of n if the additional type information about the target and the arguments at the call-site reveals that the method call will select a version of m that throws fewer exceptions than the method referenced at the declaration site.

To ensure that no unexpected checked exceptions can occur at run-time, the type checker uses two type checks. First, the *allowed exception* analysis computes which exceptions are allowed to be thrown by a particular method call. Second, the *conformance* analysis checks that a method does not throw an exception that is not allowed by its exception clause, and that this exception clause does not allow more exceptions than the exception clauses of the overridden methods.

In the original type checker [1], however, neither algorithm could deal with loops in the graph defined by the anchored exception declarations. Therefore, the original type checker did not allow such loops. Because this restriction requires a

* Corresponding author. Tel.: +32 16 32 70 59; fax: +32 16 32 79 96.
E-mail addresses: Marko.vanDooren@cs.kuleuven.be (M. van Dooren), Bart.Jacobs@cs.kuleuven.be (B. Jacobs), Wouter.Joosen@cs.kuleuven.be (W. Joosen).

[1] Marko van Dooren is a Postdoctoral researcher of the Fund for Scientific Research – Flanders (F.W.O. Vlaanderen).

0167-6423/$ – see front matter
http://dx.doi.org/10.1016/j.scico.2013.10.007

```
void first() {
  try {second();}
  catch(E1 e) {...}
  catch(E2 e) {...}                           // Good change
}
void second() throws E1 ,E2 {third();}        // Bad change
void third() throws E1 ,E2 {fourth();}        // Bad change
void fourth() throws E1 ,E2                   // Good change
  {... throw new E1(); ...
  ... throw new E2(); ...}                    // New exception
```

Fig. 1. Checked exception reduce the adaptability of software.

```
abstract class A {
  void template() throws Exception; {...hook()...}
  abstract void hook() throws Exception;
}

class B extends A {
  void hook() throws E1 {... throw new E1(); ...}
  void doSomething() throws E1 {
    try {...template()...}
    catch(RuntimeException exc) {throw exc;}
    catch(E1 exc) {throw exc;}
    catch(Exception exc) {throw new Error();}
  }
}
```

Fig. 2. Exceptions must be handled even if the cannot be thrown.

whole-program analysis, the original type checking algorithms were not modular. In addition, loops are required for mutually recursive methods.

The contribution of this paper is the presentation of modular algorithms for allowed exception analysis and exception conformance analysis that allow loops in an object-oriented language without parametric polymorphism.

1.1. Overview

Section 2 briefly explains anchored exception declarations. Section 3 defines an algorithm for allowed exception analysis for anchored exception declarations without bounded parametric polymorphism. Section 4 shows how modular exception conformance analysis can be decided. Section 5 discusses related work, and we conclude in Section 6.

2. Overview of anchored exception declarations

We now give a brief introduction to anchored exception declarations. The details and motivation are presented in the original paper [1].

2.1. Problem statement

While checked exceptions improve the robustness of software, they also decrease its flexibility. On the one hand, they increase the robustness of software by ensuring that no unexpected checked exceptions can be thrown at run-time. On the other hand, they decrease the adaptability of software because they must be propagated explicitly, and must often be handled even if they cannot be signaled.

Fig. 1 illustrates the reduced adaptability. Method `fourth` can throw checked exceptions of type `E1` or ones of its subtypes. Methods `second` and `third` propagate the exception. while method `first` handles the exception. When a new type of exception `E2` can be thrown by `fourth`, every method that invokes `fourth` must either handle the exception or explicitly propagate it by adding it to its exception clause. As a result, a wave of changes propagate along every call chain that includes `fourth` until the exception is handled. But in most cases, methods that simply propagate all exceptions will also propagate the newly added exception. In the example, it is most likely that `first` will handle the exception. The fact that `first` and `fourth` must be adapted means that the approach works since the exceptional behavior of these methods has changed. But the methods `second` and `third` must also be modified, even though their exceptional behavior has not changed. They propagated all exceptions before, and they still propagate all exceptions after the change.

Fig. 2 illustrates why exceptions must be handled even if the programmer knows that they cannot be thrown. Method `template` in class `A` calls method `hook` and must thus add `Exception` to its exception clause. Method `hook` in class `B` can only throw `E1` Even though the programmer knows that the call to `template` in `doSomething` can only throw `E1`, he must write useless exception handlers to block `Exception` and propagate `RuntimeException` and `E1`. The compiler