

Modular Anchored Exception Declarations

Marko van Dooren

Wouter Joosen

Report CW 544, August 2009



Katholieke Universiteit Leuven
Department of Computer Science

Celestijnenlaan 200A – B-3001 Heverlee (Belgium)

Modular Anchored Exception Declarations

Marko van Dooren
Wouter Joosen

Report CW 544, August 2009

Department of Computer Science, K.U.Leuven

Abstract

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

The original type checking algorithms for anchored exception declarations, however, are not modular. In this paper, we present algorithms that allow complete, modular, and decidable verification of exception safety in a language without parametric polymorphism. In addition, we show that both complete exception flow analysis and complete exception safety analysis based on type information are undecidable in a language with subtyping and parametric polymorphism.

Keywords : Exceptions, decidability, modularity, analysis.

Modular Anchored Exception Declarations

MARKO VAN DOOREN, BART JACOBS, and WOUTER JOOSEN

Abstract

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

The original type checking algorithms for anchored exception declarations, however, are not modular. In this paper, we present algorithms that allow complete and modular verification of exception safety in a language without parametric polymorphism. In addition, we show that both complete exception flow analysis and complete exception safety analysis based on type information are undecidable in a language with subtyping and parametric polymorphism.

1 Introduction

Checked exceptions improve the robustness of software [1, 2, 3]. Because every checked exception that can be signalled by a method must either be listed in the *exception clause* – the `throws` clause in Java – of that method or handled in its body, it is impossible to encounter an unanticipated checked exception at run-time.

Unfortunately, checked exceptions also cause problems. First, they decrease the adaptability of software. Modifying the exception clause of a method triggers modifications along every call chain encountering that method. Second, exception handling mechanisms do not take context information into account. Often, a programmer knows that a certain checked exception cannot be signalled by a method call, but the exception handling analysis does not. This leads to dummy exception handlers that clutter the code and are dangerous for the evolution of the program.

Anchored exception declarations offer a solution to both problems [4]. With an anchored exception declaration **like** `t.m(args)`, a programmer indicates that exceptions from the referenced method `m` are propagated instead of copy & pasting the list of exceptions. As a result, changes in the exceptional behavior of the referenced method automatically apply to the method using the anchored declaration. In addition, the compiler can reduce the set of possible exceptions for a specific method call if the additional context information reveals that a

more specific method will be invoked. The latter method may signal fewer exceptions than the method that was originally referenced.

The type checker uses two algorithms to enforce exception safety. The *exception flow analysis* computes which exceptions can be signalled by a method call. The *conformance* analysis ensures that a method cannot throw an exception if that is not allowed by its super methods. In the original type checker, however, both algorithms cannot deal with loops in the anchor graph. Therefore, the original type checker forbids loops in the anchor graph.

Because loops in the anchor graph can only be detected by a whole-program analysis, the original type checking algorithms are not modular. This is an important problem for the construction of large-scale software systems, where modularity is a crucial property.

Existing exception flow analyses [5, 6, 7, 8, 9, 10, 11, 12, 13, 14, 15, 16] are modular, but they do not give modular guarantees about exception safety. These analyses generate an exception clause to represent the exceptional behavior of a method, but they do not enforce that subclasses respect this exception clause for two reasons. First, a programmer cannot explicitly write these exception clauses. Second, the inferred exception clauses cannot be used as contracts because they are based on implementation details, and are thus fragile. As a result, if new code is added, existing code can encounter unexpected exceptions at run-time.

The contribution of this paper is twofold. First, we present complete modular type checking algorithms that guarantee exception safety in a language without parametric polymorphism. Second, we show that both complete exception flow analysis and complete exception safety analysis based on type information are undecidable in a language with subtyping and parametric polymorphism.

Overview

Section 2 briefly explains anchored exception declarations. Section 3 explains the link between anchored exception declarations and exception flow analysis. Section 4 presents the different kinds of loops that occur in the anchor graph, and Section 5 briefly presents the core language and formal notation. Sections 8 and 9 show how the type checker can be modified to deal with these loops. Section 11 shows that both safety and flow analysis are undecidable in a language with subtyping and parametric polymorphism. Section 13 discusses related work, and we conclude in Section 14.

2 Anchored Exception Declarations

We now give a brief introduction to anchored exception declarations. The details and motivation are presented in the original paper [4].

2.1 Copy & Paste for Checked Exceptions

Fig. 1 shows the interface of a strategy pattern [17] with a general purpose method that declares that it can signal a `StrategyException`. The `template` method uses a `Strategy` object to perform a more complex computation. Both the specification, written in JML [18], and the implementation of `template` delegate a part of their work to the strategy object. The exception clause of `template`, however, cannot do this; it must copy the exception clause of `compute` and thus declare that it can always signal `StrategyException`.

```
public interface Strategy {
    public Result compute() throws StrategyException;
}

/*@ post \result == ...strategy.compute()...; @*/
public Result template(Strategy strategy)
    throws StrategyException {
    ...
    ...strategy.compute()...;
    ...
}
```

Figure 1: Copy & paste for checked exceptions.

The first problem caused by the copy & paste approach for checked exceptions is the reduced adaptability of the program. If the exception clause of a method is changed, that change will ripple through all call chains that involve the modified method. In Fig. 1, we can for example let `compute` signal `Exception` to make it more general-purpose. Most of the methods in these call chains, however, simply propagate these exceptions. This means that although these methods have been modified, their behavior has not really changed. They propagated all exceptions before, and they still do after the modification.

The second problem is the lack of context information available to the exception handling mechanism. Fig. 2 shows a part of a client application that uses the template method of Fig. 1. The application contains a specific implementation of the strategy, `MyStrategy`, that can signal only the checked exception `MyException`. But even though the programmer knows that the concrete strategy signals only `MyException`, he must provide an exception handler for `Exception` in the client method. He cannot use the context information about the concrete strategy to exclude `StrategyException`. If `MyException` is handled, only the handlers for `RuntimeException` and `StrategyException` are useless. But if `MyException` is propagated, as in the example, all handlers are useless. In both cases, this is a very verbose way to write `template(myStrategy)`. In addition, if the `compute` method of `MyStrategy` is modified such that it can signal `StrategyException`, then the code will still compile, and the exception will be ignored at run-time.

```

//client code:
public class MyStrategy implements Strategy{
    public Result compute() throws MyException {...}
}

public Result client(MyStrategy myStrategy)
    throws MyException {
    try {
        ...
        ...template(myStrategy)...;
        ...
    }
    catch(RuntimeException exc) { throw exc; }
    catch(MyException exc) { throw exc; }
    catch(StrategyException exc) {
        throw new Error(); // this cannot happen
    }
}

```

Figure 2: A useless exception handler in client code.

2.2 Anchored Exception Declarations

Anchored exception declarations offer a solution by allowing a method to specify not only *which* exceptions can be signalled, but also *where* they come from.

An exception clause is no longer a list of exception types, but a list of *exception declarations*. Each exception declaration declares that certain checked exceptions can be signalled under certain circumstances. The exception types in traditional exception clauses will be called *absolute exception declarations* from now on. They declare that a certain type of exceptions can always be signalled.

An anchored exception declaration declares that a method propagates exceptions from another method, and automatically reflects changes in the exception clause of the referenced method. An anchored exception declaration consists of the keyword `like`, followed by a method expression and optionally a filter clause, as shown by the grammar in Fig. 9. The method expression determines to which method the anchored exception declaration is anchored, and thus the set of exceptions that can be signalled as a result of that exception declaration. The filter clause can narrow this set by allowing only a fixed set of exceptions to be propagated using a `propagating` filter, or by allowing everything to be propagated except for a fixed set of exceptions using a `blocking` filter. The default filter clause – no filter clause – allows all exceptions of the anchor to be propagated.

A method expression can be any method invocation that is valid in the context of the method header, including the formal parameters of the method. On top of that, type names can be used as expressions to hide implementation details or replace subexpressions that are invisible outside the method body.

2.3 Information Hiding

To ensure compile-time safety, the type checker enforces that if a method signals an exception, either it is a subtype of an absolute declaration, or it is caused by a method invocation that matches ¹ to one of the anchored exception declarations in the exception clause of that method. Otherwise, the type checker cannot know if call-site information inserted into an anchored exception declaration is also reflected in the implementation, which is required for exception safety.

As a result, an anchored exception declaration reveals that a certain method is directly or indirectly invoked by the implementation. Therefore, anchored exception declarations must be used carefully. One scenario where they should be used, is if that information is already known to the clients, such as in the example in Fig. 1. Method `template` cannot be implemented without invoking `compute`, nor can it be specified without revealing that `compute` will be invoked. Therefore, no extra information is revealed by using an anchored exception declaration to specify the exceptional behavior of `template`. This is the case in many delegation scenarios, which includes many design patterns such as Strategy, Command, Template Method, Visitor, Decorator, and many others.

2.4 Example

We illustrate the use of anchored exception declarations for the example of Fig. 1, which had two problems. We had to modify the `template` method when adding exceptions to `MyStrategy.compute`, and we had to provide a dummy exception handler for `StrategyException`. Fig. 4 shows the same example

¹If a type name is used in the method expression, then the implementation can use any expression in that position as long as the type conforms.

```
ExceptionClause:
  throws ExceptionDeclaration ( , ExceptionDeclaration)*
ExceptionDeclaration:
  AbsoluteExceptionDeclaration
  AnchoredExceptionDeclaration
AbsoluteExceptionDeclaration:
  Type
AnchoredExceptionDeclaration:
  like MethodExpression FilterClause?
FilterClause:
  (propagating ( ExceptionList ))?
  (blocking ( ExceptionList ))?
ExceptionList:
  Type ( , ExceptionList)*
MethodExpression:
  MethodCall in which type names can be used as expressions
```

Figure 3: A grammar for exception clauses.

```

public Result template(Strategy strategy)
    throws like strategy.compute() {
    ... strategy.compute() ...
}

public Result client(MyStrategy myStrategy)
    throws like myStrategy.compute() {
    ... template(myStrategy)...
}

```

Figure 4: The example using anchored exception declarations.

using anchored exception declarations. We have now expressed that changes in the exceptional behavior of `strategy.compute` and `myStrategy.compute` will always be reflected in the set of exceptions signalled by the methods `template` and `client` respectively. Consequently, the addition of exceptions to `MyStrategy.compute` will not require the modification of the `client` method.

We illustrate how context information is exploited for the method invocation in the body of the `client` method. The invocation of `template` has `myStrategy` as its actual argument, and the type of `myStrategy` is `MyStrategy`. If we insert this call-site information in the exception clause of `template` by substituting the formal parameters and the `this` variable, we get the following exception clause:

```
like myStrategy.compute()
```

To obtain the set of exceptions that is declared by this calculated exception clause, we repeat the same process, which is called *expansion*, until only absolute exception declarations remain. The formal definition of expansion is given in Section 8. In this case, there is one more step. The exception clause of `MyStrategy.compute` contains only an absolute declaration:

```
MyException
```

By inserting the static type information of the call-site, we made the method expression in the exception clause of the `template` method select a more specific method than it referenced originally. As a result, we limited the set of possible checked exceptions from `StrategyException` to `MyException`.

3 Linking Anchored Exception Declarations To Exception Flow Analysis

Anchored exception declarations are very similar to the exceptional return types used in different exception flow analyses. For example, Pessaux and Leroy [10] use an effect system to track the exceptions that a method can signal. Type

$\alpha \xrightarrow{\phi} \beta$ represent a function from α to β that can signal exceptions that are in set ϕ . Call-site information is used in the exceptional type of a function by using the variables that represent the exception signalled by the arguments. The `map` function, for example, can be typed as follows:

$$\text{map} : \forall \alpha, \beta, \phi. (\alpha \xrightarrow{\phi} \beta) \xrightarrow{\emptyset} \alpha \text{ list} \xrightarrow{\phi} \beta \text{ list}$$

The type of the corresponding Java method with anchored exception declarations looks as follows:

```
<A,B> List<B> map(F<A,B> f, List<A> l)
    throws like f.apply(A);
```

Other exception flow analyses use different notations, but the expressiveness is almost always the same. Some analyses have no equivalent for filter clauses, but that is not a fundamental problem.

The algorithms used by exception flow analyses to infer the exception clauses are very similar. For anchored exception declarations, a similar algorithm is used to infer the *implementation exception clause*, which represents the exceptional behavior of the implementation of a method. The implementation exception clause is used to verify if the implementation of a method respects the exception clause of that method.

The only real difference between anchored exception declarations and exception flow analyses is the fact that anchored exception declarations use both explicitly written and inferred exception clauses, whereas exception flow analyses use only inferred exception clauses. This has three consequences for the results of this paper.

First, making the recursive expansion algorithm modular (Section 8) closes the gap with existing exception flow analyses, but it does not improve upon them. Second, if the inferred exception clauses of existing exception flow analyses could be written explicitly by the programmer, the modular conformance verification algorithm for anchored exception declarations (Section 9) could be used to verify conformance. Third, it allows us to apply our results about the undecidability of type checking anchored exception declarations in a language with parametric polymorphism and subtyping (Section 11) to exception flow analysis in general.

3.1 Completeness

We now define a notation of *completeness* for exception flow analysis and exception safety analysis. In this definition, *analyzed* exceptions, are those exceptions that are being tracked by the analysis. For anchored exception declarations, only checked exceptions are analyzed.

Definition 1 (Completeness) *An exception flow analysis based on static type information is complete if it never determines that an analyzed exception can be signalled at run-time when it can be shown using only static type information that the exception cannot be signalled.*

```

class A {void m(B b) throws E1 {...}}

class B {void n(A a) throws E1 {...}}

// extension 1
class C extends A {
    void m(B b) throws like b.n() {...}
}

// extension 2
class D extends B {
    void n(A a) throws like a.m() {...}
}

```

Figure 5: Independent extensions can introduce loops.

4 Loops In The Anchor Graph

As illustrated in Section 2.4, the type checker recursively resolves the methods referenced by anchored exception declarations, which means that it can encounter a method that has already been processed. The original type checker did not allow such loops because it could not determine when to stop the analysis.

Because call-site information is inserted in anchored exception declarations during the expansion, the type checker might process any method that overrides the method referenced by an anchored exception declaration. As a result, the loop check must do a whole-program analysis, which makes type checker non-modular.

4.1 Intentional and Unintentional Loops

We now present examples of unintentional and intentional loops. The first kind is rare but unavoidable. The second kind is required to prevent useless exception handlers when mutually recursive delegation is used.

Unintentional loops can occur is when two extensions of a program are combined. An example is shown in Fig. 5. Methods `A.m` and `B.n` of the original program state that they can throw exceptions of type `E1`. The extensions each override one of these methods and replace the absolute declaration with an anchored exception declaration that points to original definition of the other method. Both extensions are valid because they do not introduce cycles, or allow additional exceptions. The combination of both extensions, however, introduces a loop in the anchor graph involving `C.m` and `D.n`. As a result, if the programmer does not have control over all involved source code, which can happen in an industrial setting, it is impossible to correct the problem.

The second scenario involves loops that are written on purpose to avoid

useless exception handlers. Fig. 6 shows two classes that each implement a Strategy pattern. The classes have mutually recursive methods `m` and `n` that respectively throw exceptions of types `E1` and `E2`, in addition to the exceptions originating from the sibling method of the actual argument. For both classes, we create a safe subclass where the methods no longer throw exceptions themselves, but only propagate exceptions originating from the sibling method of the actual argument. Therefore, if we use different combinations of both strategies, the invocations can throw different exceptions in each case, as shown in the example. Without a loop in the anchor chain, the analysis cannot reduce the set of possible exceptions, as at least one of both methods would have to mention both `E1` and `E2` in its exception clause in order to break the loop.

These examples show that the no-loops condition of the original type checker for anchored exception declarations is problematic. A whole-program analysis can be acceptable to let a compiler make more optimizations, but it is not acceptable for a core language construct. In addition, because of the no-loops condition, useless exception handlers are still required for collaborations in which call-backs can cause checked exceptions.

4.2 Manifestations of Loops

Recursive expansion of an anchored exception declaration that is part of a loop can result in two kinds of loops in the type check.

In the first kind of loop, the size of the method expressions remains finite. An example is shown in Fig. 7. The expansion of `like this.f()` is again `like this.f()`. Upon the second encounter of `like this.f()`, the analysis could stop because any error that can be found by further expansion will also have been found by an analysis branch that resulted from the previous expansion of `like this.f()`.

In the second kind of loop, the size of the method expressions keeps getting bigger. An example is shown in Fig. 8. The expansion of `like f()` is `like f().f()`, which in turn expands to `like f().f().f()`, and so forth. The expansion of `like g(g(a))` is `like g(g(g(a)))`, which in turn expands to `like g(g(g(g(a))))`, and so forth. In these cases, deciding when to stop the analysis is no longer trivial.

In Sections 8 and 9, we show how the type checker can compress infinitely growing expressions to expressions that contain only the information that can still influence the outcome of the analysis. The size of these compressed expressions is bounded by a finite number that depends on the particular analysis, and will force the analysis into a fixed point.

5 The Core Language

Before we explain the type checking algorithms, we describe the core language to which we add anchored exception declarations, and introduce the formal notation used in the rest of the paper.

To simplify the formal semantics and the proof of correctness, we put some restrictions on the underlying programming language. We limit expressions

```

class Strategy1 {
    void m(Strategy2 s2) throws E1, like s2.n(this) {
        ... throw new E1() ...
        ... s2.n(this) ...
    }
}
class Strategy2 {
    void n(Strategy1 s1) throws E2, like s1.m(this) {
        ... throw new E2() ...
        ... s1.m(this) ...
    }
}
// subclasses that throw no exceptions themselves
class SafeStrategy1 extends Strategy1{
    void m(Strategy2 s2) like s2.n(this) {...}
}
class SafeStrategy2 extends Strategy2{
    void n(Strategy1 s1) like s1.m(this) {...}
}
// different combinations throw different exceptions
Strategy1 s1 = ...; Strategy2 s2 = ...
SafeStrategy1 safe1 = ...; SafeStrategy2 safe2 = ...

// invocations can each throw E1 and E2
s1.m(s2);
s2.n(s1);
// invocations can each throw E2
safe1.m(s2);
s2.n(safe1);
// invocations can each throw E1
s1.m(safe2);
safe2.n(s1);
// neither invocation can throw a checked exception
safe1.m(safe2);
safe2.n(safe1);

```

Figure 6: Cycles in mutually recursive methods.

```

class A {
    void f() throws like this.f();
}

```

Figure 7: The size of the method expression is constant.

```

class A {
    A f() throws like f().f();
    A g(A a) throws like g(g(a));
}

```

Figure 8: The method expression keeps growing.

to **this**, references to formal parameters and fields, type names, constructor invocations, and method invocations. A BNF grammar for the expressions is shown in Fig. 9. In an anchored exception declaration, a type name can be used as an expression. Other expressions have been omitted for reasons of brevity, but can easily be added.

A class may not introduce fields with the same name as a field or method of one of its superclasses. In addition, syntactic overloading of methods is forbidden. Without these restrictions, type elaboration as used by ClassicJava [19] is needed to keep the fields and methods in a class hierarchy apart.

Parametric polymorphism is not supported in this part of the paper because it makes complete type checking undecidable. This is discussed further in Section 11.

Assignments to formal parameter is not allowed in the formalization. If the value of a formal parameter that is used in an anchored exception declaration can be modified, then it is impossible to exploit call-site information about that parameter. The modification might lead to the invocation of a super method of the method that was computed by the call-site analysis. In a language with mutable local variables, formal parameters referenced in an anchored exception declaration would have to be final.

5.1 Formal notation

Exception lists are represented by sets of types. The \leq operator denotes that a type is a subtype of an element of a such a set, and can be thought of as the \in operator for normal sets. The $\sqcap, \sqcup, \sqsubseteq,$ and $-$ operators correspond to the $\cap, \cup, \subseteq,$ and \setminus operators for sets, except that the former take subtyping into account. The symbol \top represents a set containing every type (or equivalently the top-level type such as `Object` in Java). The definitions of the operations are shown in Fig. 10. Note that the definition of \cap is more strict than necessary in case of multiple inheritance, where \emptyset is returned in case of two independent types, while there may be types that are subtypes of both T_a and T_b .

$e ::= var$	local variable
$e.m(\bar{e})$	method call
$e.f$	field read
$\mathbf{new} T(\bar{e})$	constructor call
$var ::= formal$	formal parameter
\mathbf{this}	

Figure 9: Grammar for expressions.

$$\begin{aligned}
S \trianglelefteq \{\bar{T}\} &\Leftrightarrow \exists T_i \in \bar{T} : S <: T_i \\
S \not\trianglelefteq \{\bar{T}\} &\Leftrightarrow \neg T_a \trianglelefteq \{\bar{T}\} \\
\{\bar{T}\} \sqcup \{\bar{S}\} &= \{\bar{T}, \bar{S}\} \\
\{\bar{T}\} \cap \{\bar{S}\} &= \{(T_1 \cap S_1) \cup \dots \cup (T_1 \cap S_m) \cup \dots \\
&\quad \cup (T_n \cap S_1) \cup \dots \cup (T_n \cap S_m)\} \setminus \{\emptyset\} \\
T_a \cap T_b &= \begin{cases} T_a & \text{if } T_a <: T_b \\ T_b & \text{if } T_b <: T_a \\ \emptyset & \text{otherwise} \end{cases} \\
\{\bar{T}\} - \{\bar{S}\} &= \{T_i \mid T_i \in \bar{T} \wedge T_i \not\trianglelefteq \{\bar{S}\}\} \\
\{\bar{T}\} \sqsubseteq \{\bar{S}\} &\Leftrightarrow \forall T_i \in \bar{T} : T_i \trianglelefteq \{\bar{S}\}
\end{aligned}$$

Figure 10: Operations on sets of types.

An absolute exception declaration is represented by a pair of sets of types: (P, B) . The first set contains the types of exceptions that can be signalled, while the second set contains the types that are blocked. An absolute exception declaration E in a program is then represented by $(\{E\}, \emptyset)$. The second element of the pair is non-empty for intermediate results during the expansion process.

We denote an anchored exception declaration `like t.m(args) propagating (P) blocking (B)`, where P and B are exception lists, as *like* $t.m(\overline{arg}) \trianglelefteq P \not\trianglelefteq B$, where P and B are the corresponding sets of types. The default values for P and B are respectively \top and \emptyset .

An exception clause is denoted as a set of exception declarations.

6 Formal Semantics of Anchored Exception Declarations

We now define the semantics of anchored exception declarations by introducing the boolean δ and ω functions. The δ function determines the exceptional behav-

ior of a particular invocation of a method, while the ω function determines the worst-case exceptional behavior of a method. The Υ and Ω functions, which are used to expand anchored exception declarations and insert call-site type information into the δ function, are defined in Section 9.1. We assume that a global class table is available for performing lookups, and omit it in the definitions to improve readability.

Definition 2 *The δ function determines whether or not an exception clause or declaration allows a checked exception E to be signalled when the parent method of the exception declaration is invoked by the given method invocation. It adds context awareness to exception declarations.*

- A method, when invoked as $t.m(\overline{arg})$, is allowed to signal a checked exception E if at least one of its exception declarations allows E to be signalled.

$$\delta(\{\overline{ED}\}, t.m(\overline{arg}), E) \Leftrightarrow \bigvee_{i=1}^{i=n} \delta(ED_i, t.m(\overline{arg}), E)$$

- An absolute exception declaration allows a checked exception E to be signalled if it is explicitly propagated and is not blocked.

$$\delta((P, B), t.m(\overline{arg}), E) \Leftrightarrow E \sqsubseteq (P - B)$$

- An anchored exception declaration allows a checked exception E to be signalled if the exception clause resulting from its expansion after inserting context information allows E to be signalled.

$$\begin{aligned} & \delta(\text{like } t_a.m_a(\overline{arg}_a) \sqsubseteq P_a \not\sqsubseteq B_a, t.m(\overline{arg}), E) \Leftrightarrow \\ & \omega(\Upsilon(\Omega(\text{like } t_a.m_a(\overline{arg}_a) \sqsubseteq P_a \not\sqsubseteq B_a, t, \overline{arg})), E) \end{aligned}$$

Note that the rule for anchored exception declarations is defined in terms of the ω function. We first insert the call-site type information using Υ and Ω , and then compute the worst-case behavior for the more specific type information using ω .

Definition 3 *The ω function determines the worst-case behavior of an exception clause or declaration by specifying when an exception E is allowed. It is a shorthand form for the δ function when the target is the parent type of the method and the actual arguments are references to the formal parameters of the method. The ω function is defined in two forms. The elaborate form keeps a trace of methods that have been processed in order to deal with loops. The simple form simply delegates the work to the elaborate form and provides an empty trace.*

1. Simple form

- $\omega(\{\overline{ED}\}, E) \Leftrightarrow \omega(\{\overline{ED}\}, E, \emptyset)$
- $\omega((P, B), E) \Leftrightarrow \omega((P, B), E, \emptyset)$
- $\omega(\text{like } t.m(\overline{arg}) \sqsubseteq P \not\sqsubseteq B, E) \Leftrightarrow \omega(\text{like } t.m(\overline{arg}) \sqsubseteq P \not\sqsubseteq B, E, \emptyset)$

2. Elaborate form

- $\omega(\{\overline{ED}\}, E, trace) \Leftrightarrow \bigvee_{i=1}^{i=n} \omega(ED_i, E, trace)$
- $\omega((P, B), E, trace) \Leftrightarrow E \preceq (P - B)$
- *For an anchored exception declaration, check if the method has already been processed with the same type of implicit and explicit arguments. If that is the case, the function is stuck in an infinite loop since method selection is based only on the types of these arguments. Any absolute declaration that would be processed by continuing – only absolute declarations actually allow exceptions to be signalled – has already been processed, so the current path of the analysis can be stopped by returning false. If the function is not stuck in a loop, expand the anchored exception declaration, update the trace, and continue.*

$$\omega(\text{like } t.m(\overline{arg}) \preceq P \not\preceq B, E, trace) \Leftrightarrow$$

$$\Gamma(t).m(\overline{\Gamma(arg)}) \notin trace \wedge$$

$$\omega(\Upsilon(\text{like } t.m(\overline{arg}) \preceq P \not\preceq B), E, \{\Gamma(t).m(\overline{\Gamma(arg)})\} \cup trace)$$

The Υ and the Ω functions insert more specific type information into the method expression of an anchored exception declaration. As a result, it can select a more specific method and thus reduce the set of exceptions that can be signalled.

7 Validity of Anchored Exception Declarations

In this section, we will discuss the validity rules for anchored exception declarations.

7.0.1 Accessibility Rule

The client of a method must have access to every element of an anchored exception declaration in order to determine which exceptions to expect when invoking the method. This is similar to the precondition availability rule of Eiffel [20] and the accessibility constraints imposed on types used in method signatures in C# [21]. Note, however, that the formalisation does not model access modifiers, because it is not required for compile-time safety. We still include the rule since it would be required in a real programming language.

Rule 1 *All elements of an anchored exception declaration must have at least the level of accessibility that the declaring method has.*

7.0.2 Conformance Rules

To ensure compile-time safety, we must impose a few rules such that no exception can ever be thrown if the caller could not expect it. Otherwise, the whole purpose of using checked exceptions is defeated.

An exception clause EC_a conforms to another exception clause EC_b , denoted as $EC_a \preceq EC_b$, when EC_a never allows a checked exception that EC_b does not

allow. For a valid program, the following conformance relations must hold. The functions and relations used in these rules will be explained further on.

Rule 2 *A method may not signal a checked exception when one of the methods it overrides does not allow it.*

$$m_a <: m_b \Rightarrow \varepsilon(m_a) \preceq \varepsilon(m_b)$$

Rule 3 *The implementation of a method may not signal a checked exception when the exception clause does not allow it.*

$$\neg m \text{ abstract} \Rightarrow IEC(m) \preceq \varepsilon(m)$$

As a result of these rules, the exception clauses of the overridden methods act as *upper bounds*, while the exception clause defined by the implementation of a method acts as a *lower bound*.

8 Exceptions Thrown by a Method Call

As illustrated by the example in Section 2.4, the type checker traverses the graph formed by the anchored exception declarations to determine which exceptions can be signalled by a method invocation. During this traversal, the type checker can encounter a loop. In Section 8.1, we explain how expansion works, and show why the original algorithm could not deal with loops. In Section 8.2, we solve the problem by introducing a stopping condition.

8.1 Expansion

The set of exceptions that can be signalled by a method invocation is computed by recursively invoking a process called *expansion*, denoted by Υ . The analysis is performed *at compile time*. Expanding an anchored exception declaration is the process of cloning the exception clause of the referenced method and substituting call-site arguments.

The power of expansion depends on the programming language. The more information can be specialized in subtypes or at a call site, the more powerful the expansion process is. In Section 11, we will show that adding parametric polymorphism makes the expansion of anchored exception declarations non-computable. Features that increase the power of expansion include covariant return types, type parameters, and type anchors.

8.1.1 Substitution

The Ω function inserts context information into an exception clause. It substitutes the formal parameters and the implicit argument *this* with call-site information.

Under the assumptions made in this paper, applying the Ω function to expression e is equal to the substitution of actual arguments: $\{e/par,$

$target/this\}e$. If static methods, syntactic overloading, and overloading of instance variables are allowed, this is no longer the case because lookups of instance variables and signatures are influenced by the insertion of more specific type information. In this case, type elaboration must be used to insert static type information, as done in ClassicJava [19]. Of course, Ω may only be applied when $ok_{\Omega}((target, this), (e, par))$ holds. This precondition demands that the target and actual arguments have the correct type, no parameter is substituted twice, and all references to $this$ in \bar{e} have the same type.

The definitions for expressions, exception declarations, and exception clauses are shown in Fig. 11. In these definitions, a is an actual argument, and p is the name of the corresponding formal parameter. The $<$: relation is used to denote subtyping for types and overriding for methods, the Γ function returns the type of an expression.

$$\begin{aligned}
\Omega(e, t, \overline{(a, p)}) &= \overline{[a/p, t/this]e} \\
\Omega((P, B), t, \overline{(a, p)}) &= (P, B) \\
\Omega(\text{like } t'.m'(\overline{a'}) \leq P' \not\leq B', t, \overline{(a, p)}) &= \\
&\quad \text{like } \Omega(t'.m'(\overline{a'}), t, \overline{(a, p)}) \leq P' \not\leq B' \\
\Omega(\{\overline{ED}\}, t, \overline{(a, p)}) &= \{\Omega(ED, t, \overline{(a, p)})\}
\end{aligned}$$

Figure 11: Substitution.

8.1.2 Filtering

The Φ function applies the filter clauses P and B of an anchored exception declaration to an exception clause. The propagated exceptions of an exception declaration are combined with P using an intersection. The blocked exceptions are combined with B using a union. The function is shown in Fig. 12.

$$\begin{aligned}
\Phi((P', B'), P, B) &= (P' \sqcap P, B' \sqcup B) \\
\Phi(\text{like } t.m(\overline{arg}) \leq P' \not\leq B', P, B) &= \\
&\quad \text{like } t.m(\overline{arg}) \leq (P' \sqcap P) \not\leq (B' \sqcup B) \\
\Phi(\{\overline{ED}\}, P, B) &= \{\Phi(ED, P, B)\}
\end{aligned}$$

Figure 12: Filtering.

8.1.3 Expansion

The expansion of an anchored exception declaration, performed by the Υ function, selects the exception clause of the invoked method using the ε function, and applies the Φ and Ω functions to the result. Because the static types of the actual arguments and the target are subtypes of the formal parameters and the parent type of the invoked method, a more specific method may be selected. As a result, a number of checked exceptions may be eliminated. The definition of

the function is shown in Fig. 13. In the definition, p_i is the formal parameter corresponding to actual argument a_i .

$$\begin{aligned} \varepsilon(t.m(\bar{a})) &= \text{exception clause of referenced method} \\ \Upsilon(\text{like } t.m(\bar{a}) \trianglelefteq P \not\trianglelefteq B) &= \Omega(\Phi(\varepsilon(t.m(\bar{a})), P, B), t, \overline{(a, p)}) \\ \Upsilon(t.m(\bar{a})) &= \Upsilon(\text{like } t.m(\bar{a}) \trianglelefteq \top \not\trianglelefteq \emptyset) \end{aligned}$$

Figure 13: Expansion.

8.1.4 Recursive Expansion

The Υ_{rec} function recursively applies the Υ function to exception clauses, exception declarations, and method invocations. If the function terminates, the result is an upper bound for the types of checked exceptions that can be signalled by a method invocation or allowed by an exception declaration or an exception clause. The Υ_{rec} function uses the regular expansion function Υ to traverse the anchor graph, and collects the absolute exception declarations that it encounters. For an absolute exception declaration (P, B) , Υ_{rec} uses the $-$ operator to calculate the worst case exception types by removing exception types that are completely blocked. The definition of the Υ_{rec} function is shown in Fig. 14. The recursion is naive in the sense that it cannot detect when further expansion cannot add new absolute declarations.

$$\begin{aligned} \Upsilon_{rec}((P, B)) &= P - B \\ \Upsilon_{rec}(\text{like } t.m(\bar{a}) \trianglelefteq P \not\trianglelefteq B) &= \\ &\quad \Upsilon_{rec}(\Upsilon(\text{like } t.m(\bar{a}) \trianglelefteq P \not\trianglelefteq B)) \\ \Upsilon_{rec}(\{\overline{ED}\}) &= \bigcup_{i=1}^{i=n} \Upsilon_{rec}(ED_i) \\ \Upsilon_{rec}(t.m(\bar{a})) &= \Upsilon_{rec}(\text{like } t.m(\bar{a}) \trianglelefteq \top \not\trianglelefteq \emptyset, \emptyset) \end{aligned}$$

Figure 14: Naive recursive expansion.

8.2 Halting Condition

To stop the recursive expansion process, the type checker must determine if the expansion of an anchored exception declaration can lead to absolute exception declarations that may not have been encountered before.

The set of absolute exception declarations that can be encountered after an expansion step is determined by the set all methods that will be analysed after that expansion step. The latter set is in turn defined in terms of the *types* of targets t of the method expressions in the anchored exception declarations **like** $t.m(\overline{arg})$ that results from *inserting context information* into the analyzed methods. As a result, if an anchored exception declaration **like** $t.m(\overline{arg})$ is expanded recursively, then the set of encountered absolute exception declarations

$$\begin{aligned}
\Upsilon_{rec}((P, B), trace) &= P - B \\
\Upsilon_{rec}(like\ t.m(\bar{a}) \leq P \not\leq B, trace) &= \\
&\quad \mathbf{if}\ \Gamma(t).m(\overline{\Gamma(\bar{a})}) \in trace\ \mathbf{then} \\
&\quad \emptyset \\
&\quad \mathbf{else} \\
&\quad \Upsilon_{rec}(\Upsilon(like\ t.m(\bar{a}) \leq P \not\leq B), \\
&\quad \quad \{\Gamma(t).m(\overline{\Gamma(\bar{a})})\} \cup trace) \\
\Upsilon_{rec}(\{\overline{ED}\}, trace) &= \bigcup_{i=1}^{i=n} \Upsilon_{rec}(ED_i, trace) \\
\Upsilon_{rec}(t.m(\bar{a})) &= \Upsilon_{rec}(like\ t.m(\bar{a}) \leq \top \not\leq \emptyset, \emptyset)
\end{aligned}$$

Figure 15: Recursive expansion.

is determined by the types of t and \overline{arg} . Therefore, if the types of the targets and actual arguments in anchored exception declarations **like** $t_1.m(\overline{arg}_1)$ and **like** $t_2.m(\overline{arg}_2)$ are equal, then the recursive expansion of both anchored exception declarations yields the same result.

The new recursive expansion function, which is shown in Fig. 15, uses this information to stop the analysis. The Γ function returns the type of an expression. The type checker stores every anchored exception declaration that it expands in a trace, after replacing t and \overline{arg} by their types. This trace is consulted before an anchored exception declaration is expanded to verify if it has already been expanded with the same types of t and \overline{arg} . If this is the case, the analysis stops because further analysis cannot return absolute exception declarations that are not also returned by the analysis of the anchored exception declaration that was responsible for the trace element that stopped the analysis. The branches of both analyses are identical.

Because parametric polymorphism is not supported, the number of types that is valid in a program is finite. Therefore, the trace kept by recursive expansion will always reach a fixed point, ensuring that the analysis will end.

8.3 Example

Fig. 16 shows the analysis for method call `safe1.m(safe2)` of Fig. 6. For each step, the figure shows the method call or anchored exception declaration that is currently being analyzed, along with the current trace, and the result of the analysis. Because the involved exception clauses have only one exception declaration, the analysis consists of a single branch. Therefore, in each step except the last one, the result equals the result of the next step. The analysis stops when `safe1.m(safe2)` is about to be expanded a second time.

In this example, the last anchored exception declaration is syntactically equal to an anchored exception declaration that has been processed before, but that is not always the case. It is for example possible that the method expression keeps growing because of delegation ($f().f().f().f().\dots$) or recursive calls ($f(f(f(f(\dots))))$). Mapping such expressions onto their types keeps the number of possible elements in the trace finite.

```

0. safe1.m(safe2)
   result0 = result1
1. like safe1.m(safe2) ≤ ⊤ ↯ ∅
   trace = ∅
   result1 = result2
2. like safe2.n(safe1) ≤ ⊤ ↯ ∅
   trace = {SafeStrategy1.m(SafeStrategy2)}
   result2 = result3
3. like safe1.m(safe2) ≤ ⊤ ↯ ∅
   trace = {SafeStrategy1.m(SafeStrategy2),
            SafeStrategy2.n(SafeStrategy1)}
   result3 = ∅

```

Figure 16: Recursive expansion of `safe1.m(safe2)`.

9 Conformance Verification

The second type checker algorithm is the conformance check. The conformance check $EC_a \preceq EC_b$ returns true if exception clause EC_a does not allow an exception unless EC_b allows that exception as well if the same call-site information is used for both exception clauses. This type check is used for two purposes. First, the exception clause of a method must conform to the exception clauses of all methods that it overrides. This check ensures that a method cannot specify that it throws an exception when the overridden methods would not do so. Second, the *implementation exception clause* of a method must conform to the exception clause of that method. The implementation exception clause is an exception clause that is computed at compile time, and represents an upper bound for the exceptional behavior of the method body. This check ensures that the implementation of a method respects the exception clause.

We first explain the old conformance algorithm and the auxiliary functions it depends on in Section 9.1. We then present the new algorithm in Section 9.3.

9.1 The Original Algorithm

Fig. 17 shows the conformance check of the original type checker. The algorithm checks if every exception declaration of EC_a conforms to EC_b . If that is the case, then EC_a as a whole also conforms to EC_b .

The first condition (1) is equivalent to the traditional exception conformance rule for checked exceptions. It ensures that every checked exception allowed by an absolute declaration of EC_a is also allowed by an absolute declaration of EC_b . Note that this rule forbids transforming anchored exception declarations into absolute declarations since an anchored declaration promises that an exception can be signalled only by the referenced method, which is not the case for an absolute declaration.

For an anchored exception declaration $anchor_a$, there are two cases in which

$$\begin{array}{c}
EC_a \preceq_{old} EC_b \\
\Downarrow \\
\forall (P_a, B_a) \in EC_a, \forall E \in P_a - P_b : \quad (1) \\
\quad \exists (P_b, B_b) \in EC_b : \Phi((P_a, B_a), E, \emptyset) \preceq (P_b, B_b) \\
\wedge \forall anchor_a \in EC_a, \forall E \in \Upsilon_{rec}(anchor_a) : \quad (2) \\
\quad \exists anchor_b \in EC_b : \Phi(anchor_a, E, \emptyset) \preceq anchor_b \quad (2.a) \\
\quad \vee \Phi(\Upsilon(anchor_a), E, \emptyset) \preceq_{old} EC_b \quad (2.b)
\end{array}$$

Figure 17: The old conformance verification algorithm.

it is valid. Both cases are quantified over the checked exceptions that are allowed by $anchor_a$ ($\Upsilon_{rec}(anchor_a, \emptyset)$). Without this quantification, it would not be allowed to replace `like m() propagating (E1)`, `like m() propagating (E2)` by `like m() propagating (E1,E2)`, which is equivalent.

First, $anchor_a$ is valid if there exists an $anchor_b$ in EC_b that always references the same method as $anchor_a$ or one of its supermethods when the same context information is inserted in both declarations, and if $anchor_b$ also allows the exception (case 2.a). The actual check is done by the conformance check \preceq for anchored exception declarations, which we discuss below. Second, an anchored exception declaration is valid if the exception clause that results from its expansion conforms to EC_b (case 2.b). This case allows a method to delegate the exceptional behavior to any method, as long as that method does not allow more exceptions than EC_b if the same context information is inserted. For example, this allows a method to replace an absolute exception declaration by an anchored exception declaration that signals the same set of exceptions or a subset. The absolute declaration states that the exception can always be signalled, while the anchored declarations states that they can only be signalled if a particular method signals them. Rule 2.b makes anchored exception declarations flexible to use, but because it performs an expansion step, it can make the algorithm get stuck in an infinite loop.

The algorithm is similar to that of recursive expansion in the sense that it traverses the anchor graph, but instead of collecting exception types, it searches for conformance of exception declarations (cases 2.a and 2.b). Therefore, we will also introduce a halting condition based on a similar technique. Before we can explain how this is done for the conformance check, however, we must explain the conformance checks for exception declarations.

9.1.1 Conformance of Exception Declarations

Fig. 18 shows the conformance checks for absolute exception declarations and anchored exception declarations. In case of absolute exception declarations, the type checker verifies that the set of exceptions allowed by the left-hand declaration is a subset of that of the right-hand side. For anchored exception declarations, the type checker verifies that the filter clause of $anchor_a$ is stronger

than that of $anchor_b$, and that the method expression of $anchor_a$ conforms to that of $anchor_b$. The latter check ensures that $anchor_b$ references the same method as $anchor_a$, or one of its supermethods.

$$\begin{aligned}
(P_a, B_a) \preceq (P_b, B_b) &\Leftrightarrow (P_a - B_a) \sqsubseteq (P_b - B_b) \\
\text{like } t_a.m_a(\bar{e}_a) \preceq P_a \not\triangleleft B_a \preceq \text{like } t_b.m_b(\bar{e}_b) \preceq P_b \not\triangleleft B_b & \\
&\Downarrow \\
t_a.m_a(\bar{e}_a) \preceq t_b.m_b(\bar{e}_b) \wedge (P_a - B_a) \sqsubseteq (P_b - B_b) &
\end{aligned}$$

Figure 18: Conformance of exception declarations.

Fig. 19 shows the conformance check for method expressions. Basically, the expression must be syntactically equal, except for the places where $expr_b$ contains a type name. As those places, $expr_a$ may have any expression, as long as its type is a subtype of the corresponding type name in $expr_b$. The \cong relation denotes that both formal parameters are corresponding formal parameters of overriding or equal methods.

$$\begin{aligned}
this_a \preceq this_b &\Leftrightarrow \Gamma(this_a) <: \Gamma(this_b) \\
e \preceq T &\Leftrightarrow \Gamma(e) <: T \\
formal_a \preceq formal_b &\Leftrightarrow formal_a \cong formal_b \\
new A(\bar{a}) \preceq new B(\bar{b}) &\Leftrightarrow A = B \wedge \overline{a \preceq b} \\
t_a.f_a \preceq t_b.f_b &\Leftrightarrow t_a \preceq t_b \wedge f_a = f_b \\
t_a.m_a(\bar{a}) \preceq t_b.m_b(\bar{b}) &\Leftrightarrow m_a = m_b \wedge t_a \preceq t_b \wedge \overline{a \preceq b}
\end{aligned}$$

Figure 19: Conformance of method expression.

9.2 The logical conformance relation: \preceq_{log}

We introduce the logical \preceq_{log} relation in order to simplify reasoning about anchored exception declarations. For compile-time safety, it suffices to require that $\delta(EC_a, t.m(\bar{a}), E) \Rightarrow \delta(EC_b, t.m(\bar{a}), E)$ holds between a method and the methods it overrides and between a method body and the exception clause of that method. In a full-blown programming language, however, this becomes difficult to reason about because of concepts such as static and final methods. They allow EC_a to be a valid refinement of EC_b based on the knowledge that some methods cannot be overridden. Such an analysis is hard for a programmer to do and would thus cause confusion when a certain type of transition of exception clauses would be accepted in one part of a program, but rejected in another part because the modifiers of the methods involved are slightly different.

$$\begin{aligned}
EC_a \preceq_{log} EC_b &\Leftrightarrow \forall n : EC_a \preceq_{log,n} EC_b \\
&EC_a \preceq_{log,n} EC_b \\
&\Downarrow \\
\forall (P_a, B_a) \in EC_a, \forall E \mid \omega((P_a, B_a), E) : & \quad (1) \\
&\exists (P_b, B_b) \in EC_b : \Phi((P_a, B_a), E, \emptyset) \preceq (P_b, B_b) \\
\wedge \forall anchor_a \in EC_a, \forall E \mid \omega(anchor_a, E) : & \quad (2) \\
&\exists anchor_b \in EC_b : \Phi(anchor_a, E, \emptyset) \preceq anchor_b \\
\vee n > 0 \Rightarrow \Phi(\Upsilon(anchor_a), E, \emptyset) \preceq_{log,n-1} EC_b & \quad (2.b)
\end{aligned}$$

Figure 20: The logical conformance relation \preceq_{log} .

Relation \preceq_{log} , however, is not directly used during type checking, as it can get stuck in an infinite loop. We will therefore introduce the \preceq relation in the next section, which is a modified version of \preceq_{log} that correctly handles loops. We discuss them separately to simplify the explanation, and because we need both definitions to prove that they are equivalent.

The \preceq_{log} relation for exception clauses is shown in Figure 20. It is defined in terms of $\preceq_{log,n}$ which means that after n expansion steps, the analysis has not yet found a problem. This is required to cover the case of an infinite loop, where the result would be undefined if \preceq_{log} would be defined as $\preceq_{log,n}$ without using n , and thus without the $n > 0$ check. In that case, both $EC_a \preceq_{log} EC_b$ and $EC_a \not\preceq_{log} EC_b$ would hold if there is a loop in one of the anchor chains of EC_a and no problem is ever encountered. After all, the result would be defined entirely in terms of itself. It is important to note however, that the direct evaluation of \preceq_{log} still cannot always be done in a finite amount of time. That problem is solved in Section 4.

9.3 Halting Condition

Because of the syntactical equivalence in the conformance check for method expressions, type information alone does not suffice to determine when the analysis can stop. For recursive expansion, expressions are allowed to keep growing because the outcome of the analysis is only determined by their types, and the number of valid types is limited in a language without generics. For the conformance check, the type checker must additionally determine which parts of an expression can influence the outcome of the conformance check.

9.3.1 Compressing Method Expressions

Because every branch for which the analysis ends must end in either case 1 or 2.a, it suffices to compute each way in which an anchored exception declaration can influence the syntactical match of $anchor_a$, and any anchored exception declaration $anchor_b$ of EC_b .

Because $anchor_b$ is written in the program text, its size is finite. As a result, only a finite number of elements of $anchor_a$ that can influence the success of a syntactic match. Therefore, an analysis branch can stop if it is about to expand an anchored exception declaration of which the elements that can influence the analysis are the same as those of a previously expanded anchored exception declaration that references the same method. Any other differences between the previously expanded anchored exception declaration and the current one cannot influence the analysis.

To compute how $anchor_a$ can influence the outcome of the analysis, all *relevant forms* of the method expression e_a of $anchor_a$ must be computed. A relevant form of e_a with respect to matching e_b is a transformation e'_a of e_a such that e'_a matches e'_b , where e'_b is either e_b or one of its subexpressions, and such that every part of e_a that matches a type name in e'_b is replaced by its type. If no match is found, there is no relevant form, and e_a can be replaced entirely by its static type since it can only influence the selection of method, similar to the analysis for recursive expansion. Matching is done using the \preceq relation of Fig. 19, because that is the one that is used to determine the result of the analysis.

It is important that matching is also checked for subexpressions of e_b because it could be the case that e_a will match e_b if it grows bigger. If e_a cannot match any subexpression of e_b , it can never be part of an expression that matches e_b , and therefore it cannot change the outcome of the analysis. After all, expressions cannot be broken down into their subexpression by the analysis; they either remain unmodified or become part of a bigger expression.

Replacing the parts that match a type name by their type keeps the size of the relevant forms finite because the size of e_b is finite. Therefore, the set of all relevant forms for $anchor_a$ with respect to the anchored exception declarations of EC_b is finite. This ensures that if the analysis loops, it must eventually encounter an anchored exception declaration $anchor'_a$ that has the same relevant forms as $anchor_a$. As a result, further expansion is not required because the result is covered by the analysis branches that split off from the expansion of $anchor_a$.

We illustrate the computation of relevant forms in Fig. 21, where we search for the relevant forms of different expressions e_i with respect to expression e . For reasons of brevity, each variable is assumed to have a type equal to its capitalized name. For example, a has type A and so on. The irrelevant part of an expression is marked with a gray background along with the type name that it matches. The relevant part is underlined, along with the part of e that it matches. The relevant form equals the type of the irrelevant part concatenated with the relevant part. For expression e_1 , there are two relevant forms. In the left case, e_1 matches C , meaning that $a.b.c$ is irrelevant and there is no relevant part. The relevant form is C . In the right case, e_1 matches $B.c$, meaning that $a.b$ is the irrelevant part, and c is the relevant part. The relevant form is $B.c$. For e_2 , there is no irrelevant part, as the entire expression matches the prefix of the second argument of the invocation. The relevant form therefore is e_2 itself. For e_3 , no match can be found, which means that it is completely irrelevant when it comes to matching e . Therefore, the relevant form is the type of e_2 .

$\Gamma(a) = A, \Gamma(b) = B, \dots, \Gamma(z) = Z$
 $e : \text{C.d.m}(\text{B.c.d}, \text{x.y.z})$
 $e_1 : \text{a.b.c}$
 $e_2 : \text{x.y}$
 $e_3 : \text{y.z}$

irrelevant, relevant

1) Overlay: $\text{C.d.m}(\text{B.c.d}, \text{x.y.z})$
 a.b.c a.b.c

Relevant forms: C, B.c

2) Overlay: $\text{C.d.m}(\text{B.c.d}, \text{x.y.z})$
 x.y

Relevant forms: x.y

3) Overlay: $\text{C.d.m}(\text{B.c.d}, \text{x.y.z})$
 y.z

Relevant forms: Z

Figure 21: Finding relevant forms of an expression.

9.3.2 The Compression Function

We now present the compression function, which computes all relevant forms of an anchored exception declaration with method expression e_a with respect to an exception clause EC_b . The result of the compression of an anchored exception declaration $anchor_a$ ($\kappa(anchor_a)$) is the same anchored exception declaration, but in which the target and arguments have been replaced by *sets*. We call this the compression of $anchor_a$. Each set contains the relevant forms of the corresponding subexpression of $anchor_a$ with respect to matching EC_b . The set of relevant forms of $anchor_a$ with respect to EC_b can then be obtained by computing the cartesian product of the sets of target and arguments, and filling in each tuple in the anchored exception declaration. But because identical sets of sets result in the same cartesian product, we skip the last step, and directly store the compression of $anchor_a$ in the trace. The analysis can stop if the trace already contains the compressed form of $anchor_a$ because all relevant elements are the same.

The κ function compresses the anchored exception declaration by replacing its irrelevant parts with their static types, and is defined in Fig. 22.

The κ function works as follows. It tries to match the arguments of the method expression of the anchored exception declaration to be compressed (e_a) with the method expressions of all anchored exception declarations in the given exception clause and their subexpressions. Rule 1 splits up the computation for all arguments, including the target. Rule 2 splits up the computation for all anchored exception declarations in the reference exception clause, and adds the type of the expression to be compressed. This is required to ensure that the result set is not empty, and to ensure that if two anchored exception declarations have the same compressed forms, the target and all arguments have the same type. Rules 3-6 compute the relevant form for each node in the expression, then delegate to the subexpressions, and collect the results. The check for a match is done in the \mathcal{M} function. Rule 7 returns the type of e_a if there is no match. This ensures that the result of κ is always defined. Rule 8 wraps the relevant form of the expression in a set. Function \mathcal{R} computes the relevant form of e_a for matching e_b . Rules 9-13 rebuild the matching part of e_a and descend into the subexpressions. Rule 14, however, returns the type of e_a because the exact form of e_a is irrelevant for matching T_b ; only its type is relevant. This is where the actual compression is done. As a result, the size (number of nodes in the abstract syntax tree) of $\mathcal{R}(e_a, e_b)$ is equal to the size of e_b . Therefore, the biggest relevant form of an argument of an anchored exception declaration, as determined by κ is limited in size by the biggest anchored exception declaration in EC_b . Because EC_b is a textual part of the program, we know that its size is finite. As a result, for a given EC_b and method m_a , there is a finite number of possible compressed forms for all anchored exception declarations referencing m_a .

Fig. 23 shows the updated conformance verification algorithm. Before expanding an anchored exception declaration, the algorithm checks if its compressed form is already in the trace. If that is the case, the current analysis

$$\frac{}{\kappa(\mathbf{like} \ t_a.m_a(\bar{a}_i) \trianglelefteq P_a \not\trianglelefteq B_a, EC_b)} \quad (1)$$

$$= \mathbf{like} \ \kappa(t_a, EC_b).m_a(\kappa(a_i, EC_b)) \trianglelefteq P_a \not\trianglelefteq B_a$$

$$\frac{EC_b = \overline{abs}, \mathbf{like} \ t_{b,i}.m_{b,i}(\bar{b}_i) \trianglelefteq P_i \not\trianglelefteq B_i}{\kappa(e_a, EC_b) = \bigcup_{i=1}^{i=n} (\kappa(e_a, t_{b,i}.m_{b,i}(\bar{b}_i)) \cup \{\Gamma(expr_a)\})} \quad (2)$$

$$\frac{e_b = \mathit{new} \ T_b(\bar{b}_i^{i \in 1..n})}{\kappa(e_a, e_b) = \mathcal{M}(e_a, e_b) \cup \bigcup_{i=1}^{i=n} \kappa(e_a, b_i)} \quad (3)$$

$$\frac{e_b = t_b.m_b(\bar{b}_i^{i \in 1..n})}{\kappa(e_a, e_b) = \mathcal{M}(e_a, e_b) \cup \kappa(e_a, t_b) \cup \bigcup_{i=1}^{i=n} \kappa(e_a, b_i)} \quad (4)$$

$$\frac{e_b = t_b.var_b}{\kappa(e_a, e_b) = \mathcal{M}(e_a, e_b) \cup \kappa(e_a, t_b)} \quad (5)$$

$$\frac{e_b = \mathit{formal}_b \vee e_b = \mathit{this}_b \vee e_b = T_b}{\kappa(e_a, e_b) = \mathcal{M}(e_a, e_b)} \quad (6)$$

$$\frac{e_a \not\trianglelefteq e_b}{\mathcal{M}(e_a, e_b) = \{\Gamma(e_a)\}} \quad (7)$$

$$\frac{e_a \preceq e_b}{\mathcal{M}(e_a, e_b) = \{\mathcal{R}(e_a, e_b)\}} \quad (8)$$

$$\frac{\mathit{new} \ T_a(\bar{a}_i) \preceq \mathit{new} \ T_b(\bar{b}_i)}{\mathcal{R}(\mathit{new} \ T_a(\bar{a}_i), \mathit{new} \ T_b(\bar{b}_i)) = \mathit{new} \ T_a(\mathcal{R}(a_i, b_i))} \quad (9)$$

$$\frac{t_a.m_a(\bar{a}_i) \preceq t_b.m_b(\bar{b}_i)}{\mathcal{R}(t_a.m_a(\bar{a}_i), t_b.m_b(\bar{b}_i)) = \mathcal{R}(t_a, t_b).m_a(\mathcal{R}(a_i, b_i))} \quad (10)$$

$$\frac{t_a.var_a \preceq t_b.var_b}{\mathcal{R}(t_a.var_a, t_b.var_b) = \mathcal{R}(t_a, t_b).var_a} \quad (11)$$

$$\frac{\mathit{formal}_a \preceq \mathit{formal}_b}{\mathcal{R}(\mathit{formal}_a, \mathit{formal}_b) = \mathit{formal}_a} \quad (12)$$

$$\frac{\mathit{this}_a \preceq \mathit{this}_b}{\mathcal{R}(\mathit{this}_a, \mathit{this}_b) = \mathit{this}_a} \quad (13) \quad \frac{e_a \preceq T_b}{\mathcal{R}(e_a, T_b) = \Gamma(e_a)} \quad (14)$$

Figure 22: Compression of anchored exception declarations.

$$\begin{aligned}
& EC_a \preceq EC_b \Leftrightarrow \emptyset \vdash EC_a \preceq EC_b \\
& \text{trace} \vdash EC_a \preceq EC_b \\
& \quad \Downarrow \\
& \forall (P_a, B_a) \in EC_a, \forall E \mid \omega((P_a, B_a), E) : \quad (1) \\
& \quad \exists (P_b, B_b) \in EC_b : \Phi((P_a, B_a), E, \emptyset) \preceq (P_b, B_b) \\
& \wedge \forall \text{anchor}_a \in EC_a, \forall E \mid \omega(\text{anchor}_a, E) : \quad (2) \\
& \quad \exists \text{anchor}_b \in EC_b : \Phi(\text{anchor}_a, E, \emptyset) \preceq \text{anchor}_b \quad (2.a) \\
& \vee \kappa(\text{anchor}_a, EC_b) \notin \text{trace} \implies \quad (2.b) \\
& \quad \{\kappa(\text{anchor}_a, EC_b)\} \cup \text{trace} \vdash \Phi(\Upsilon(\text{anchor}_a), E, \emptyset) \preceq EC_b
\end{aligned}$$

Figure 23: The new conformance verification algorithm.

branch is stopped without indicating an error. Otherwise, the anchored exception declaration is expanded, and the analysis continues.

The new conformance algorithm is complete, safe, and modular. For the formal proof, we refer to the companion technical report [22].

9.4 Example

In Fig. 24, we show how the trace of compressed anchored exception declarations prevents the conformance analysis from entering an infinite loop. The example adds class `SuperStrategy1` which is a super class of `Strategy1`. The `m` method of `SuperStrategy1` states that it can signal `E1`, `E2`, or the exceptions signalled by invoking `n()` on the result of `this.f()`. This exception clause is called `EC_b`. The anchored exception declaration `like this.f().n()` is only added to have an actual `anchor_b` in the analysis.

The bottom of the figure shows the analysis for verifying that the exception clause of `Strategy1.m` conforms to that of `SuperStrategy1.m`. The analysis steps that verify the absolute declarations are not shown to save space. Only the analysis steps in the loop are shown. For each step, the figure shows the current exception clause and the different cases (1,2.a, and 2.b) of the conformance analysis.

Because the anchored exception declarations never match the only anchored exception declaration in the exception clause of `SuperStrategy1.m`, the compression function replaces the target of the invocation with its static type. As a result, when the exception clause of `Strategy1.m` is processed for the second time, and still no match has been found, the compressed form of `like this.s2().s1().s2().n()`, which is equal to the compressed form of `like this.s2().n()`, is found in the trace, and the analysis can stop. No problems have been found, and no problems could be found if the analysis would continue. Therefore, this path of analysis returns `true`. Since the absolute exception declarations `E1` and `E2` are also valid, the conclusion is that the exception clause of `Strategy1.m` is valid.

```

class SuperStrategy1{
  Strategy2 f() {...}
  void m() throws E1, E2, like this.f().n();
}
class Strategy1 extends SuperStrategy1{
  Strategy2 s2() {...}
  void m() throws E1, like this.s2().n() {...}
}
class Strategy2 {
  Strategy1 s1() {...}
  void n() throws E2, like this.s1().m() {...}
}

```

let $EC_b = \mathbf{throws} E1, E2, \mathbf{like} \text{ this.f().n()}$

- 1) $EC_1 = E_1, \mathbf{like} \text{ this.s2().n()}$
 $trace_1 = \emptyset$
 $abs_1 = E_1 <: E_1 \implies \mathbf{TRUE}$ (1)
 $anchor_1 = \mathbf{like} \text{ this.s2().n()}$
 $anchor_1 \not\leq \mathbf{like} \text{ this.f().n()} \implies \mathbf{FALSE}$ (2.a)
 $\kappa_1 = \kappa(\mathbf{like} \text{ this.s2().n()}, EC_b)$
 $= \{\mathit{Strategy2}\}.n()$
 $\kappa_1 \notin trace_1 \implies \text{store } \kappa_1 \text{ and expand}$ (2.b)
- 2) $EC_2 = E_2, \mathbf{like} \text{ this.s2().s1().m()}$
 $trace_2 = \{\{\mathit{Strategy2}\}.n()\}$
 $abs_2 = E_2 <: E_2 \implies \mathbf{TRUE}$ (1)
 $anchor_2 = \mathbf{like} \text{ this.s2().s1().m()}$
 $anchor_2 \not\leq \mathbf{like} \text{ this.f().n()} \implies \mathbf{FALSE}$ (2.a)
 $\kappa_2 = \kappa(\mathbf{like} \text{ this.s2().s1().m()}, EC_b)$
 $= \{\mathit{Strategy1}\}.m()$
 $\kappa_2 \notin trace_2 \implies \text{store } \kappa_2 \text{ and expand}$ (2.b)
- 3) $EC_2 = E_1, \mathbf{like} \text{ this.s2().s1().s2().n()}$
 $trace_3 = \{\{\mathit{Strategy2}\}.n(), \{\mathit{Strategy1}\}.m()\}$
 $abs_3 = E_1 <: E_1 \implies \mathbf{TRUE}$ (1)
 $anchor_3 = \mathbf{like} \text{ this.s2().s1().s2().n()}$
 $anchor_3 \not\leq \mathbf{like} \text{ this.f().n()} \implies \mathbf{FALSE}$ (2.a)
 $\kappa_3 = \kappa(\mathbf{like} \text{ this.s2().s1().s2().n()}, EC_b)$
 $= \{\mathit{Strategy2}\}.n()$
 $\kappa_3 \in trace_3 \implies \mathbf{STOP}$ (2.b)

Figure 24: Compression forces the trace into a fixed point.

10 Important Theorems

In this section we give an overview of the most interesting theorems of the type system. The full proof of compile-time safety is shown in Appendix 12.

Theorem 10.1 states that the algorithm for computing \preceq , as defined in Fig. 23 always ends.

Theorem 10.1 *The algorithm for computing \preceq always ends.*

Theorem 10.2 states that \preceq and \preceq_{log} are equivalent. This means that the stopping condition works correctly, since \preceq never stops too soon.

Theorem 10.2 *For exception clauses, \preceq is equivalent to \preceq_{log} .*

$$EC_a \preceq_{log} EC_b \Leftrightarrow EC_a \preceq EC_b$$

Theorems 10.3 and 10.4 state that the Φ and Ω functions are monotone with respect to the \preceq pre-order when the same context information is inserted in both operands or when more specific context information is inserted in the left-hand operand.

Theorem 10.3 *Φ is monotone.*

$$\begin{aligned} EC_a \preceq EC_b \wedge (P_c - B_c) \sqsubseteq (P_d - B_d) \\ \Downarrow \\ \Phi(EC_a, P_c, B_c) \preceq \Phi(EC_b, P_d, B_d) \end{aligned}$$

Theorem 10.4 *Ω is monotone.*

$$\begin{aligned} EC_a \preceq EC_b \wedge target_a \preceq target_b \wedge args_a \preceq args_b \wedge \\ ok_{\Omega}((target_a, this(EC_a)), args_a) \wedge ok_{\Omega}((target_b, this(EC_b)), args_b) \\ \Downarrow \\ \Omega(EC_a, target_a, args_a) \preceq \Omega(EC_b, target_b, args_b) \end{aligned}$$

Theorem 10.5 states that the \preceq relation is a pre-order. The $=$ relation can be chosen such that \preceq becomes a partial order by demanding that $a \preceq b \wedge b \preceq a \Rightarrow a = b$. That would mean that two exception clauses are equal when they specify the same exceptional behavior, which makes perfect sense.

Theorem 10.5 *The \preceq relation is a pre-order.*

1. \preceq is reflexive

$$a \preceq a$$

2. \preceq is transitive

$$a \preceq b \wedge b \preceq c \Rightarrow a \preceq c$$

Theorem 10.6 states that the \preceq relation between exception clauses implies that the left-hand exception clause never declares a checked exception that is not declared by the right-hand exception clause, a property we will need for ensuring compile-time safety.

Theorem 10.6

$$EC_a \preceq EC_b \Rightarrow (\omega(EC_a, E) \Rightarrow \omega(EC_b, E))$$

Theorem 10.7 states that the *IEC* function correctly computes the exceptional behavior of a method. This corresponds to stating that it follows the language specification – in this case the Java language specification.

Theorem 10.7 *The implementation exception clause of a non-abstract method is an upper bound for the exceptional behavior of the implementation of that method.*

Theorem 10.8 states that the worst-case behavior of a method body, specified by the implementation exception clause, conforms to the exception clause of that method for any specific call-site. As a result, a programmer – and a compiler – can obtain an upper bound for the exceptional behavior of a method call by inserting the context information into the exception clause of the invoked method.

Theorem 10.8 *Let $t.m(\bar{e}_i^i)$ be a method invocation in a valid program, $EC = \varepsilon(t.m(\bar{e}_i^i))$, $IEC = IEC(t.m(\bar{e}_i^i))$, and let par_i be the formal parameter corresponding to e_i , then:*

$$\omega(\Omega(IEC, t, (\overline{e_i, par_i})^i), E) \Rightarrow \omega(\Omega(EC, t, (\overline{e_i, par_i})^i), E)$$

For compile-time safety to be violated, there must be at least one method of which the implementation can signal a checked exception in a certain context that could not have been predicted by the client when inspecting the exception clause of that method in the same context. In Appendix 12, we show that this is not possible for a program satisfying all rules.

Theorem 10.9 states that the expansion of a method invocation allows less exceptions than the exception clause of the invoked method. This property is not necessary for compile-time safety, but is crucial from a methodological point of view. If it does not hold, a method invocation can allow more checked exceptions to be signalled than the invoked method declares, which would be safe but very confusing. For example, if the Υ function would simply return `throws Throwable`, compile-time safety would not be at risk, but anchored exception declarations would become useless.

Theorem 10.9

$$\begin{array}{c} \omega(\Upsilon(\text{like } t.m(\bar{e}) \trianglelefteq P \not\trianglelefteq B), E) \\ \downarrow \\ \omega(\varepsilon(\text{like } t.m(\bar{e}) \trianglelefteq P \not\trianglelefteq B), E) \end{array}$$

11 Parametric Polymorphism

The algorithms present in Section 8 and 9 make the exception safety analysis modular while retaining the safety and completeness properties of the old algorithms, but only in a language without parametric polymorphism.

In a language that supports parametric polymorphism and subtyping, however, complete exception safety analysis of anchored exception declarations is undecidable. We prove this by proving that complete exception flow analysis, which is required for exception safety analysis, is undecidable. To prove the latter, we encode a Minsky counter machine [23] with an arbitrary number of counters using anchored exception declarations as the program text, and recursive expansion as the execution engine. A counter machine uses counters to store its state as natural numbers; its instructions perform arithmetic operations on those counter. A program consists of a list of labeled instructions. A Minsky counter machine has two real instructions: $\text{INC}(r,z)$ and $\text{JZDEC}(r,z_{true},z_{false})$. The first instruction increases counter r , and then jumps to label z . The second instruction checks if counter r is zero. If that counter is zero, the program jumps to label z_{true} . If that counter is not zero, the program decrements the counter and jumps to label z_{false} . The HALT instruction stops the machine.

It has been proven that, given a proper encoding for the inputs, counter machines with two or more counters are Turing complete [23]. Therefore, the halting problem for such a counter machine is undecidable. We prove that exception flow analysis of anchored exception declarations in presence of subtyping and parametric polymorphism is undecidable by showing that it is equivalent to the halting problem for a Minsky counter machine with an arbitrary number of counters.

Fig. 25 shows the encoding. The top of the figure contains the base infrastructure for the encoding. Classes Nat , Zero , and Succ are used to encode natural numbers. Class M represents the counter machine. For each counter in the counter machine, class M has a generic parameter that is constrained to type Nat . If a counter in the counter machine has value 3, the corresponding generic parameter of M has value $\text{Succ}\langle\text{Succ}\langle\text{Succ}\langle\text{Zero}\rangle\rangle\rangle$.

The second part of the figure shows the translation rules for the instructions. The $\text{class} += \text{method}$ operator adds the right-hand side method to the left-hand side class. An instruction with label L_x is encoded as the exception clause of method L_x of class M . To keep the encoding as short as possible, type names are used in the method expressions instead of real expressions. Using real expressions requires additional inspector methods for the counters, and leads to longer method expressions because both the inspector methods and their types must be written for constructor invocations.

The encoding of the $\text{INC}(i,L_j)$ instruction is straightforward. The i -th generic parameter is wrapped in type Succ , and the method expression points to method L_j to encode the jump. The translation of $\text{JZDEC}(i,L_j,L_k)$ is more complicated. To test if the i -th counter is zero, the exception clause points to a method L_x of Nat (step 1). In Nat , method L_x is added, and its exception clause specifies that it can signal any kind of exception. If the counter is Zero , the program must jump to L_j , which is reflected in the exception clause of Zero.L_x . If the counter is $\text{Succ}\langle N \rangle$, the i -th counter is decremented by replacing it by N , and program control goes to method L_k of the updated machine. The HALT instruction is encoded as a method L_x that throws an exception X .

The program is executed by starting the analysis at method $\text{M}\langle\dots\rangle.\text{L0}$.

```

// Infrastructure
class Nat {}
class Zero extends Nat {}
class Succ<N extends Nat> extends Nat {}
class M<C1 extends Nat,...,Cn extends Nat> {}

// Translation

A) Lx: INC(i,Lj)
  1) M +=
     Lx() throws like M<C1,...Succ<Ci>,...,Cn>.Lj();

B) Lx: JZDEC(i,Lj,Lk)
  1) M += Lx() throws like Ci.Lx(M<C1,...,Cn>);

  2) Nat +=
     <C1,...,Cn> Lx(M<C1,...,Cn> x)
     throws Throwable;

  3) Zero +=
     <C1,...,Cn> Lx(M<C1,...,Cn> x)
     throws like M<C1,...,Cn>.Lj();

  4) Succ +=
     <C1,...,Cn> Lx(M<C1,...,Cn> x)
     throws like M<C1,...,Ci-1,N,Ci+1,...,Cn>.Lk();

C) HALT
  1) M += Lx() throws X;

```

Figure 25: Encoding an N-counter Minsky machine with anchored exception declarations and recursive expansion.

An invocation of `M<...>.L0` will either throw exception `X`, or no exception at all. To determine if `X` can be signalled by a particular invocation of `M<...>.L0`, the type checker must keep expanding the anchored exception declarations until it ends up at a method that represents a `HALT` instruction. The problem of determining the precise set of exceptions that can be signalled by a method invocation is thus equivalent to the halting problem of a Minsky machine with `N` counters, which is undecidable. As a result, complete type checking of anchored exception declarations in a language with subtyping and parametric polymorphism is undecidable. Therefore, it is impossible to create a halting condition for the type checker that does not reject valid programs.

Creating a type check that is decidable, sound, and precise enough to be useful remains an open research challenge. Letting any anchored exception declaration mean `throws Throwable` is decidable and sound, but completely useless. We are currently exploring algorithms that track which types can possibly be assigned to a generic parameter during the analysis. For example, in type `List<List<List<A>>>`, generic parameter `T` of `List` can only be used to invoke methods defined in `List` or in `A`. By combining this information with a constraint that forbids the use of generic parameters in absolute exception declarations, the type check can possibly be made decidable by always creating an analysis branch for each possible parent type to cover the worst-case scenario. These restrictions probably remove the possibility to encode an *if* instruction if the type checker is used as the execution engine, because the analysis will always execute every branch.

11.1 Exception Flow Analysis in General

The encoding of the counter machine can be used to show that exception flow analysis in general is undecidable in a language with parametric polymorphism and subtyping. Such an encoding can be made by rewriting the example to use real expressions, and moving those expressions to the method bodies. Determining if method `L0` throws exception `X` is again equivalent to deciding whether the counter machine halts, even if only static type information is used.

12 Proof of Exception Safety

This section contains the compile-time safety proof of anchored exception declarations. For the proof, we limit expressions to *this*, references to formal parameters and instance variables, and method invocations. Additionally, type names may be used as expressions in method expressions.

12.1 Notation

In addition to the formal notation presented in Section 5.1, we need some extra notation for the proof.

An actual argument that is used for substitution is represented by a pair containing the expression of the actual argument as the first element, and the corresponding formal parameter as the second element. $actual = (e, par)$

For the substitution of parameters in other parameters that are to be substituted, we write: $\Omega((val, par), pre, args) = (\Omega(val, pre, args), par)$

$$\begin{aligned} &\Omega((v_1, p_1) \dots (v_n, p_n), pre, args) = \\ &(\Omega(v_1, pre, args), p_1) \dots (\Omega(v_n, pre, args), p_n) \end{aligned}$$

12.2 Extension to the \preceq relation

For arguments that are to be substituted, we extend the definition of the \preceq relation.

$$\begin{aligned} (val_a, param_a) \preceq (val_b, param_b) &\Leftrightarrow val_a \preceq val_b \wedge param_a \preceq param_b \\ (v_{a,1}, p_{a,1}) \dots (v_{a,n}, p_{a,n}) \preceq (v_{b,1}, p_{b,1}) \dots (v_{b,n}, p_{b,n}) &\Leftrightarrow \\ (v_{a,1}, p_{a,1}) \preceq (v_{b,1}, p_{b,1}) \wedge \dots \wedge (v_{a,n}, p_{a,n}) \preceq (v_{b,n}, p_{b,n}) & \end{aligned}$$

12.3 Sets of types

We will need the following Lemma for sets of types. The proof is analogous to the proof for mathematical sets.

Lemma 12.1

$$\begin{aligned} (P_a - B_a) \sqsubseteq (P_b - B_b) \wedge (P_c - B_c) \sqsubseteq (P_d - B_d) \\ \Downarrow \\ ((P_a \sqcap P_c) - (B_a \sqcup B_c)) \sqsubseteq ((P_b \sqcap P_d) - (B_b \sqcup B_d)) \end{aligned}$$

Proof 1

$$\begin{aligned} (P_a - B_a) \sqsubseteq (P_b - B_b) \wedge (P_c - B_c) \sqsubseteq (P_d - B_d) \\ \Updownarrow \text{(definitions of } \sqsubseteq \text{ and } -) \\ \forall x : ((x \sqsubseteq P_a \wedge x \not\sqsubseteq B_a) \Rightarrow (x \sqsubseteq P_b \wedge x \not\sqsubseteq B_b)) \wedge \\ ((x \sqsubseteq P_c \wedge x \not\sqsubseteq B_c) \Rightarrow (x \sqsubseteq P_d \wedge x \not\sqsubseteq B_d)) \\ \Downarrow \\ \forall x : (x \sqsubseteq P_a \wedge x \not\sqsubseteq B_a \wedge x \sqsubseteq P_c \wedge x \not\sqsubseteq B_c) \Rightarrow \\ (x \sqsubseteq P_b \wedge x \not\sqsubseteq B_b \wedge x \sqsubseteq P_d \wedge x \not\sqsubseteq B_d) \\ \Updownarrow \text{(definitions of } \sqcap \text{ and } \sqcup) \\ \forall x : (x \sqsubseteq (P_a \sqcap P_c) \wedge x \not\sqsubseteq (B_a \sqcup B_c)) \Rightarrow (x \sqsubseteq (P_b \sqcap P_d) \wedge x \not\sqsubseteq (B_b \sqcup B_d)) \\ \Updownarrow \text{(definitions of } \sqsubseteq \text{ and } -) \\ ((P_a \sqcap P_c) - (B_a \sqcup B_c)) \sqsubseteq ((P_b \sqcap P_d) - (B_b \sqcup B_d)) \end{aligned}$$

$$\begin{array}{c}
S = \{e_1, \dots, e_n\} \\
\frac{\exists i \in [1 \dots n] : \forall j [1 \dots n] : i \neq j \implies (\text{len}(e_i) > \text{len}(e_j) \vee (i < j \wedge \text{len}(e_i) = \text{len}(e_j)))}{\text{max}(S) = e_i} \quad (15) \\
\frac{}{\text{len}(\text{this}) = 1} \quad (16) \quad \frac{}{\text{len}(\text{formal}) = 1} \quad (17) \quad \frac{}{\text{len}(T) = 1} \quad (18) \\
\frac{}{\text{len}(t.\text{var}) = 1 + \text{len}(t)} \quad (19) \quad \frac{}{\text{len}(t.m(\overline{\text{arg}}^{i \in 1..n})) = 1 + \text{len}(t) + \sum_{i=1}^{i=n} \text{len}(\text{arg}_i)} \quad (20) \\
\frac{}{\text{len}(\text{new } T(\overline{\text{arg}}^{i \in 1..n})) = 1 + \sum_{i=1}^{i=n} \text{len}(\text{arg}_i)} \quad (21)
\end{array}$$

Figure 26: Definition of the length of an expression.

12.4 The Algorithm for Computing \preceq Ends in Finite Time

Figure 26 shows the definition of the length of an expression, which is used to prove that there is a finite number of traces for each combination of an exception clause, and referenced method.

Lemma 12.2 *The compression of an expression is as long as the expression as a reference for compression in case of match.*

$$(\mathcal{R}(\text{expr}_a, \text{expr}_b) = \{\text{expr}_c\}) \Rightarrow \text{len}(\text{expr}_c) = \text{len}(\text{expr}_b)$$

Proof 2 *We prove this by induction on the structure of expr_b . Since there is a concrete result, we know that $\text{expr}_a \preceq \text{expr}_b$.*

$$\begin{array}{l}
1. \text{expr}_a = \text{new } T_a(\overline{\text{arg}}_{a,i}^{i \in 1..n}) \wedge \text{expr}_b = \text{new } T_b(\overline{\text{arg}}_{b,i}^{i \in 1..n}) \wedge \\
\text{expr}_c = \text{new } T_a(\overline{\mathcal{R}(\text{arg}_{a,i}, \text{arg}_{b,i})}^{i \in 1..n})
\end{array}$$

In both cases, the length is one more than the sum of the lengths of the arguments. Induction on the arguments of expr_b completes the case.

$$\begin{array}{l}
2. \text{expr}_a = t_a.m_a(\overline{\text{arg}}_{a,i}^{i \in 1..n}) \wedge \text{expr}_b = t_b.m_b(\overline{\text{arg}}_{b,i}^{i \in 1..n}) \wedge \\
\text{expr}_c = \mathcal{R}(t_a, t_b).m_a(\overline{\mathcal{R}(\text{arg}_{a,i}, \text{arg}_{b,i})}^{i \in 1..n})
\end{array}$$

In both cases, the length is one more than the sum of the lengths of the arguments added to the length of the target. Induction on the target and the arguments of expr_b completes the case.

$$3. \text{expr}_a = t_a.\text{var}_a \wedge \text{expr}_b = t_b.\text{var}_b \wedge \text{expr}_c = \mathcal{R}(t_a, t_b).\text{var}_a$$

In both cases, the length is one more than the length of the target. Induction the target of expr_b completes the case.

$$4. \text{expr}_a = \text{formal}_a \wedge \text{expr}_b = \text{formal}_b \wedge \text{expr}_c = \text{formal}_a$$

In both cases, the length is one.

5. $expr_a = this_a \wedge expr_b = this_b \wedge expr_c = this_a$

In both cases, the length is one.

6. $expr_b = T_b \wedge expr_b = T_b \wedge expr_c = \Gamma(expr_a)$

In both cases, the length is one.

Lemma 12.3 *The compression of an expression cannot be longer than the longest expression in the exception clause used as a reference for compression, or length 1.*

let $EC_b = abs_1, \dots, abs_m,$

like $t_1.m_1(args_1) \trianglelefteq P_1 \not\trianglelefteq B_1, \dots, \text{like } t_n.m_n(args_n) \trianglelefteq P_n \not\trianglelefteq B_n$

in $len(max(\kappa(expr_a, EC_b))) \leq max(len(max(\{t_i.m_i(args_i)^{i \in 1..n}\})), 1)$

Proof 3 *If $expr_a$ does not match with any subexpression, the \mathcal{R} function will always return an empty set. In that case, the only element in the result is $\Gamma(expr_a)$, which has length one. For any positive matches, Lemma 12.2 proves the case, as the κ function only collects the results.*

Lemma 12.4 *The trace containing the compressed forms of anchored exception declarations during type checking has a finite number of states.*

Proof 4 *From Lemma 12.3, it follows that the compression of both the target and the arguments of an anchored exception declaration are limited in length by the longest expression of the exception clause used as a reference for compression or by length 1. Therefore, since the program is finite, the resulting set for the target and each of the arguments has a finite number of possible elements.*

Theorem 12.5 *The algorithm for computing \preceq always ends.*

Proof 5 *This follows directly from Lemma 12.4.*

12.5 \preceq and \preceq_{log} are Equivalent

Lemma 12.6

$$\kappa(\text{like } t_a.m_a(\overline{a_i^{i \in 1..n}}) \trianglelefteq P \not\trianglelefteq B, EC) = \kappa(\text{like } t_b.m_b(\overline{b_i^{i \in 1..n}}) \trianglelefteq P \not\trianglelefteq B, EC) \wedge \text{anchor}_d \in EC$$

$$\left(\begin{array}{c} \downarrow \\ \text{like } \Omega(t_c.m_c(\overline{c_i^{i \in 1..n}}), t_a, \overline{a_i^{i \in 1..n}}) \trianglelefteq P \not\trianglelefteq B \preceq \text{anchor}_d \\ \updownarrow \\ \text{like } \Omega(t_c.m_c(\overline{c_i^{i \in 1..n}}), t_b, \overline{b_i^{i \in 1..n}}) \trianglelefteq P \not\trianglelefteq B \preceq \text{anchor}_d \end{array} \right)$$

Proof 6 *Because t_a and t_b have the same compressed form with respect to EC and thus also with respect to anchor_d , t_a will match a part of anchor_d if and only if t_b also matches that part. The same goes for the corresponding arguments a_i and b_i . Therefore, if the anchored exception declarations matches anchor_d after substitution with t_a and $\overline{a_i^{i \in 1..n}}$, then it must also match anchor_d after substitution with t_b and $\overline{b_i^{i \in 1..n}}$.*

Lemma 12.7

$$\kappa(\mathbf{like} \ t_a.m_a(\overline{a_i^{i \in 1..n}}) \preceq P \not\preceq B, EC) = \kappa(\mathbf{like} \ t_b.m_b(\overline{b_i^{i \in 1..n}}) \preceq P \not\preceq B, EC)$$

$$\begin{aligned} & \downarrow \\ \kappa(\mathbf{like} \ \Omega(t_c.m_c(\overline{c_i^{i \in 1..n}}), t_a, \overline{a_i^{i \in 1..n}}) \preceq P \not\preceq B, EC) = \\ & \kappa(\mathbf{like} \ \Omega(t_c.m_c(\overline{c_i^{i \in 1..n}}), t_b, \overline{b_i^{i \in 1..n}}) \preceq P \not\preceq B, EC) \end{aligned}$$

Proof 7 *Since the compressed forms are identical in both cases, the compressed forms of the targets and arguments, which have now become subexpressions, will again have identical compressed forms. Since the other parts of the anchored exception declarations into which they are inserted are identical, it follows from the definitions of κ and \mathcal{R} that the compressed forms of these anchored exception declarations will again be identical.*

Lemma 12.8

$$\begin{aligned} \kappa(\mathit{anchor}_a, EC) &= \kappa(\mathit{anchor}_b, EC) \\ & \downarrow \\ \mathit{anchor}_a \preceq_{log,n} EC &\Leftrightarrow \mathit{anchor}_b \preceq_{log,n} EC \end{aligned}$$

Proof 8 *We prove this by induction on n .*

1. *Base case: $n = 0$ For $n = 0$, we must prove that $\mathit{anchor}_a \preceq_{log,0} EC \Leftrightarrow \mathit{anchor}_b \preceq_{log,0} EC$. Since there are no absolute declaration in the left-hand side, and no expansion will be done, we must prove that $\mathit{anchor}_a \preceq \mathit{anchor}_c \Leftrightarrow \mathit{anchor}_b \preceq \mathit{anchor}_c$ with $\mathit{anchor}_c \in EC$. Since both anchors have the same compressed form with respect to EC , and anchor_c , it follows from the definition of \preceq for anchored exception declarations that this is true.*
2. *Induction step*

We know that the induction hypothesis holds for n . We must now prove that it holds for $n + 1$. Again, the direct conformance checks will give identical results. We must still prove, however, that this also holds after expansion. We must still prove that $\Phi(\Upsilon(\mathit{anchor}_a), E, \emptyset) \preceq_{log,n} EC \Leftrightarrow \Phi(\Upsilon(\mathit{anchor}_b), E, \emptyset) \preceq_{log,n} EC$. We now prove that all three parts of the \preceq_{log} conformance checks (1, 2.a, and 2.b) given identical results.

- (a) *Absolute declarations (1): from the definition of κ , it follows that all corresponding arguments have identical types, and that the filter clauses are identical. Therefore, the check on absolute declarations will give the same results in both cases.*
- (b) *Direct conformance for anchors (2.a): from Lemma 12.6, it follows that for each anchor in the expanded exception clauses, the direct conformance check with anchors from EC gives identical results.*

(c) *Indirect conformance for anchors (2.b): to prove this, we apply the induction hypothesis. Before, we can do that, however, we must prove that the corresponding anchors of the expanded exception clauses have the same compressed form with respect to EC. This follows from Lemma 12.7.*

12.5.1 \preceq is Correct if the Trace is not Sabotaged

Lemma 12.9 proves that for every n , if the trace of \preceq is not sabotaged, it is equivalent to \preceq_{\log_n} when \preceq_{\log} fails after n expansion steps. We define $\not\preceq_{\log,n}$ to mean that the verification fails after exactly n expansion steps:

$$EC_a \not\preceq_{\log,n} EC_b \Leftrightarrow ((n > 0 \Rightarrow EC_a \preceq_{\log,n-1} EC_b) \wedge \neg EC_a \preceq_{\log,n} EC_b)$$

The condition under which the trace is fine is $\forall anchor \in \kappa^{-1}(trace) : anchor \preceq_{\log,n} EC_b$. This means that if an anchored exception declaration $anchor$ compresses to an element of the trace, its analysis may not lead to a contradiction in n steps or less. In other words, if $anchor$ would fail in n or less steps, that would mean that \preceq could use it to stop the analysis before it performs n expansions along the path that made \preceq_{\log} fail after n expansion steps. Note that the theorem does not state that the results will always be different. If the “faulty” anchored exception declaration in the trace has nothing to do with the analysis of EC_a , it will of course not change the outcome of \preceq .

Lemma 12.9

$$\begin{aligned} \forall n : \forall trace : \\ (\forall anchor \in \kappa^{-1}(trace) : anchor \preceq_{\log,n} EC_b) \wedge EC_a \not\preceq_{\log,n} EC_b \\ \Downarrow \\ trace \vdash EC_a \not\preceq EC_b \end{aligned}$$

Proof 9 *We prove this by induction on n .*

1. *Base case: $n=0$*

Because no anchor can end in a failure after 0 expansion steps ($anchor \preceq_{\log,0} EC_b$ is trivially true), we must prove that $EC_a \not\preceq_{\log,0} EC_b \Rightarrow trace \vdash EC_a \not\preceq EC_b$. This follows directly from the definitions of \preceq and \preceq_{\log} since they are identical if no expansion is done.

2. *Induction step:*

We know that

$$\begin{aligned} (\forall anchor \in \kappa^{-1}(trace) anchor \preceq_{\log,n} EC_b) \wedge EC_a \not\preceq_{\log,n} EC_b \\ \Downarrow \\ trace \vdash EC_a \not\preceq EC_b \end{aligned}$$

We must prove that

$$\begin{aligned} & (\forall \text{anchor} \in \kappa^{-1}(\text{trace}) \text{anchor} \preceq_{\log, n+1} EC_b) \wedge EC_a \not\preceq_{\log, n+1} EC_b \\ & \quad \downarrow \\ & \text{trace} \vdash EC_a \not\preceq EC_b \end{aligned}$$

Since the conformance check does not fail immediately ($n > 0$), and \preceq and $\preceq_{\log, n}$ are identical for the direct checks, there must be at least one $\text{anchor}_a \in EC_a$ that leads to a failure after $n + 1$ expansion steps in relation \preceq_{\log} . We must now prove that \preceq will follow that path, and lead to the correct result.

First we prove that \preceq follows anchor_a . This is trivial because the precondition states that if its compressed form was in the trace, the following property should hold: $\text{anchor}_a \preceq_{\log, n+1} EC_b$, which contradicts that fact that it fails after $n + 1$ expansion steps.

Next, we must prove that the result of \preceq following anchor_a leads to a failure. This means expanding anchor_a , and adding its compressed form to the trace.

From $EC_a \not\preceq_{\log, n+1}$, it follows for every $\text{anchor}_a \in EC_a$ that leads to a failure after $n + 1$ expansion steps that: $\Phi(\Upsilon(\text{anchor}_a), E, \emptyset) \not\preceq_{\log, n} EC_b$. We can now apply the induction hypothesis to the resulting exception clause, which fails after n expansion steps, provided that we prove that the trace is still ok. In other words, we must prove that that:

$$\forall \text{anchor} \in \kappa^{-1}(\text{trace} \cup \{\kappa(\text{anchor}_a)\}) : \text{anchor} \preceq_{\log, n} EC_b$$

For the elements in the old trace, this is trivially true since their conformance check using \preceq_{\log} will not even result in a failure after $n + 1$ steps. We must still prove however, that the same holds for all anchors that have the same compressed form as anchor_a . For anchor_a itself, this is trivially true as it only leads to failure after $n + 1$ steps. From Lemma 12.8, it follows that the same holds for all other anchors with the same compressed form as anchor_a , which means that according to the induction hypothesis:

$$\text{trace} \cup \{\kappa(\text{anchor}_a)\} \vdash \Phi(\Upsilon(\text{anchor}_a), E, \emptyset) \not\preceq EC_b$$

and thus:

$$\text{trace} \vdash EC_a \not\preceq EC_b$$

12.5.2 \preceq is Equivalent to \preceq_{log}

Theorem 12.10 *The conformance algorithm \preceq for exception clauses is equivalent to \preceq_{log} .*

$$EC_a \preceq_{log} EC_b \Leftrightarrow EC_a \preceq EC_b$$

Proof 10 *For the \Leftarrow case of the equivalence, the proof is trivial because if the \preceq algorithm finds a contradiction, the \preceq_{log} relation will find it as well. The only possible mismatch between both is \preceq stopping too soon.*

To prove the \Rightarrow case of the equivalence, we use the fact that a failure is always detected by \preceq_{log} in a finite number of expansion steps if there is one. In all other cases, the relation is false. We must therefore still prove that:

$$\forall n : EC_a \not\preceq_{log,n} EC_b \Rightarrow \emptyset \vdash EC_a \not\preceq EC_b$$

This follows directly from Lemma 12.9 because the empty trace cannot contain an compressed form that will change the outcome of \preceq .

12.6 Properties of Φ and Ω

In this section we prove some properties about the Φ and Ω functions. Specifically, we will prove that under certain conditions $\Phi \circ \Omega$ is equivalent to $\Omega \circ \Phi$, possibly after modifying the arguments.

The first lemma states that Φ and Ω may always be swapped when the arguments of Ω are valid. The function *this*(x) returns the implicit parameter *this* that is in the scope of the program element x .

Lemma 12.11

$$\begin{aligned} & ok_{\Omega}(args, (pre, this(EC))) \\ & \quad \Downarrow \\ & \Phi(\Omega(EC, pre, args), P, B) = \Omega(\Phi(EC, P, B), pre, args) \end{aligned}$$

Proof 11 *Since an exception clause is a list of exception declarations, and the Φ and Ω functions respectively apply Φ and Ω to the exception declarations, it suffices to prove that:*

$$\Phi(\Omega(ED, pre, args), P, B) = \Omega(\Phi(ED, P, B), pre, args)$$

1. (P_x, B_x)

$$\begin{aligned} \Phi(\Omega((P_x, B_x), pre, args), P, B) &= \Omega(\Phi((P_x, B_x), P, B), pre, args) \\ & \quad \Updownarrow \text{(definition of } \Omega \text{ and } \Phi) \\ \Phi((P_x, B_x), P, B) &= \Omega((P_x \sqcap P, B_x \sqcup B), pre, args) \\ & \quad \Updownarrow \text{(definition of } \Omega) \\ \Phi((P_x, B_x), P, B) &= (P_x \sqcap P, B_x \sqcup B) \\ & \quad \Updownarrow \text{(definition of } \Phi) \\ & \quad \text{true} \end{aligned}$$

4. $new\ C(a_1, \dots, a_n)$

$$\begin{aligned}
& \Omega(\Omega(new\ C(a_1, \dots, a_n), pre_a, args_a), pre_b, args_b) = \\
& \Omega(new\ C(a_1, \dots, a_n), \Omega(pre_a, pre_b, args_b), \Omega(args_a, pre_b, args_b)) \\
& \quad \Downarrow (definition\ of\ \Omega) \\
& \Omega(new\ C(\Omega(a_1, pre_a, args_a), \dots, \Omega(a_n, pre_a, args_a)), pre_b, args_b) = \\
& \quad new\ C(\Omega(a_1, \Omega(pre_a, pre_b, args_b), \Omega(args_a, pre_b, args_b)), \dots, \\
& \quad \quad \Omega(a_n, \Omega(pre_a, pre_b, args_b), \Omega(args_a, pre_b, args_b))) \\
& \quad \Downarrow (definition\ of\ \Omega) \\
& \quad new\ C(\Omega(\Omega(a_1, pre_a, args_a), pre_b, args_b), \dots, \\
& \quad \quad \Omega(\Omega(a_n, pre_a, args_a), pre_b, args_b)) = \\
& \quad new\ C(\Omega(a_1, \Omega(pre_a, pre_b, args_b), \Omega(args_a, pre_b, args_b)), \dots, \\
& \quad \quad \Omega(a_n, \Omega(pre_a, pre_b, args_b), \Omega(args_a, pre_b, args_b))) \\
& \quad \Downarrow (induction\ on\ finite\ expression\ tree) \\
& \quad \quad \quad true
\end{aligned}$$

5. $t.var$

$$\begin{aligned}
& \Omega(\Omega((t.var, env_{var}), pre_a, args_a), pre_b, args_b) = \\
& \Omega((t.var, env_{var}), \Omega(pre_a, pre_b, args_b), \Omega(args_a, pre_b, args_b)) \\
& \quad \Downarrow (definition\ of\ \Omega) \\
& \Omega((\Omega(t, pre_a, args_a).var, env_{var}), pre_b, args_b) = \\
& (\Omega(t, \Omega(pre_a, pre_b, args_b), \Omega(args_a, pre_b, args_b)).var, env_{var}) \\
& \quad \Downarrow (definition\ of\ \Omega) \\
& (\Omega(\Omega(t, pre_a, args_a), pre_b, args_b).var, env_{var}) = \\
& (\Omega(t, \Omega(pre_a, pre_b, args_b), \Omega(args_a, pre_b, args_b)).var, env_{var}) \\
& \quad \Downarrow (definition\ of\ \Omega) \\
& \quad \Omega(\Omega(t, pre_a, args_a), pre_b, args_b) = \\
& \quad \Omega(t, \Omega(pre_a, pre_b, args_b), \Omega(args_a, pre_b, args_b)) \\
& \quad \Downarrow (induction\ on\ finite\ expression\ tree) \\
& \quad \quad \quad true
\end{aligned}$$

6. $t.m(a_1, \dots, a_n)$

$$\begin{aligned}
& \Omega(\Omega(t.m(a_1, \dots, a_n), pre_a, args_a), pre_b, args_b) = \\
& \Omega(t.m(a_1, \dots, a_n), \Omega(pre_a, pre_b, args_b), \Omega(args_a, pre_b, args_b)) \\
& \quad \Downarrow \text{(definition of } \Omega) \\
& \Omega(\Omega(t, pre_a, args_a).m(\Omega(a_1, pre_a, args_a), \dots, \\
& \quad \Omega(a_n, pre_a, args_a)), pre_b, args_b) = \\
& \Omega(t, \Omega(pre_a, pre_b, args_b), \Omega(args_a, pre_b, args_b)).m(\\
& \Omega(a_1, \Omega(pre_a, pre_b, args_b), \Omega(args_a, pre_b, args_b)), \dots, \\
& \Omega(a_n, \Omega(pre_a, pre_b, args_b), \Omega(args_a, pre_b, args_b))) \\
& \quad \Downarrow \text{(definition of } \Omega) \\
& \Omega(\Omega(t, pre_a, args_a), pre_b, args_b).m(\\
& \quad \Omega(\Omega(a_1, pre_a, args_a), pre_b, args_b), \dots, \\
& \quad \Omega(\Omega(a_n, pre_a, args_a), pre_b, args_b)) = \\
& \Omega(t, \Omega(pre_a, pre_b, args_b), \Omega(args_a, pre_b, args_b)).m(\\
& \Omega(a_1, \Omega(pre_a, pre_b, args_b), \Omega(args_a, pre_b, args_b)), \dots, \\
& \Omega(a_n, \Omega(pre_a, pre_b, args_b), \Omega(args_a, pre_b, args_b))) \\
& \quad \Downarrow \text{(induction on finite expression tree)} \\
& \quad \text{true}
\end{aligned}$$

The same property holds for applying two consecutive substitutions on an exception clause.

Lemma 12.13

$$\begin{aligned}
& ok_{\Omega}(args_a, (pre_a, this(EC))) \wedge ok_{\Omega}(args_b, (pre_b, this(pre_a))) \\
& \quad \forall formal \in EC : formal \in args_a \\
& \quad \Downarrow \\
& \Omega(\Omega(EC, pre_a, args_a), pre_b, args_b) = \\
& \Omega(EC, \Omega(pre_a, pre_b, args_b), \Omega(args_a, pre_b, args_b))
\end{aligned}$$

Proof 13 *Since an exception clause is a list of exception declarations, and the Ω function applies Ω to the exception declarations, it suffices to prove that:*

$$\begin{aligned}
& \Omega(\Omega(ED, pre_a, args_a), pre_b, args_b) = \\
& \Omega(ED, \Omega(pre_a, pre_b, args_b), \Omega(args_a, pre_b, args_b))
\end{aligned}$$

1. (P, B)

$$\begin{aligned}
& \Omega(\Omega((P, B), pre_a, args_a), pre_b, args_b) = \\
& \Omega((P, B), \Omega(pre_a, pre_b, args_b), \Omega(args_a, pre_b, args_b)) \\
& \quad \Downarrow \text{(definition of } \Omega) \\
& \Omega((P, B), pre_b, args_b) = (P, B) \\
& \quad \Downarrow \text{(definition of } \Omega) \\
& (P, B) = (P, B)
\end{aligned}$$

2. *like* $t.m(args) \sqsubseteq P \not\sqsubseteq B$

$$\begin{aligned}
& \Omega(\Omega(\textit{like } t.m(args) \sqsubseteq P \not\sqsubseteq B, pre_a, args_a), pre_b, args_b) = \\
& \Omega(\textit{like } t.m(args) \sqsubseteq P \not\sqsubseteq B, \Omega(pre_a, pre_b, args_b), \Omega(args_a, pre_b, args_b)) \\
& \quad \Downarrow (\textit{definition of } \Omega) \\
& \textit{like } \Omega(t.m(args), pre_a, args_a) \sqsubseteq P \not\sqsubseteq B, pre_b, args_b) = \\
& \textit{like } \Omega(t.m(args), \Omega(pre_a, pre_b, args_b), \Omega(args_a, pre_b, args_b)) \sqsubseteq P \not\sqsubseteq B \\
& \quad \Downarrow (\textit{definition of } \Omega) \\
& \textit{like } \Omega(\Omega(t.m(args), pre_a, args_a), pre_b, args_b) = \sqsubseteq P \not\sqsubseteq B \\
& \textit{like } \Omega(t.m(args), \Omega(pre_a, pre_b, args_b), \Omega(args_a, pre_b, args_b)) \sqsubseteq P \not\sqsubseteq B \\
& \quad \Downarrow (\textit{Lemma 12.12}) \\
& \quad \textit{true}
\end{aligned}$$

12.7 Properties of the ω Function

Filtering an exception declaration to only allow checked exceptions of type E to pass has no effect on whether or not E is allowed or not.

12.7.1 Exception Declarations

Lemma 12.14

$$\omega(ED, E, trace) \Leftrightarrow \omega(\Phi(ED, E, \emptyset), E, trace)$$

Proof 14

1. (P, B)

$$\begin{aligned}
& \omega((P, B), E, trace) \Leftrightarrow \omega(\Phi((P, B), E, \emptyset), E, trace) \\
& \quad \Downarrow (\textit{definition of } \omega) \\
& E \sqsubseteq (P - B) \Leftrightarrow E \sqsubseteq ((P \sqcap E) - B) \\
& \quad \Downarrow \\
& \quad \textit{true}
\end{aligned}$$

2. *anchor*: proven by induction on the first derivation of Lemma 12.15. This induction will end in absolute exception declarations for which the proof is given in the first part of this lemma. As for the definition of the ω function,

the trace will prevent the induction from getting stuck in an infinite loop.

$$\begin{aligned}
& \omega(\Phi(\text{anchor}, E, \emptyset), E, \text{trace}) \\
& \quad \Downarrow (\text{definition of } \omega) \\
& \Gamma(t).m(\Gamma(\bar{\text{args}})) \notin \text{trace} \Rightarrow \\
& \quad \omega(\Upsilon(\Phi(\text{anchor}, E, \emptyset)), E, \{\Gamma(t).m(\Gamma(\bar{\text{args}}))\} \cup \text{trace}) \\
& \quad \quad \Downarrow (\Phi \text{ and } \Upsilon \text{ can be switched}) \\
& \Gamma(t).m(\Gamma(\bar{\text{args}})) \notin \text{trace} \Rightarrow \\
& \quad \omega(\Phi(\Upsilon(\text{anchor}), E, \emptyset), E, \{\Gamma(t).m(\Gamma(\bar{\text{args}}))\} \cup \text{trace}) \\
& \quad \quad \Downarrow (\text{induction on Lemma 12.15}) \\
& \Gamma(t).m(\Gamma(\bar{\text{args}})) \notin \text{trace} \Rightarrow \\
& \quad \omega(\Upsilon(\text{anchor}), E, \{\Gamma(t).m(\Gamma(\bar{\text{args}}))\} \cup \text{trace}) \\
& \quad \quad \Downarrow (\text{definition of } \omega) \\
& \quad \quad \omega(\text{anchor}, E, \text{trace})
\end{aligned}$$

12.7.2 Exception Clauses

The same property holds for exception clauses:

Lemma 12.15

$$\omega(\text{EC}, E, \text{trace}) \Leftrightarrow \omega(\Phi(\text{EC}, E, \emptyset), E, \text{trace})$$

Proof 15

$$\begin{aligned}
& \omega(\Phi(\text{EC}, E, \emptyset), E, \text{trace}) \\
& \quad \Downarrow \\
& \omega(\Phi(\text{ED}_1, E, \emptyset), E, \text{trace}) \vee \dots \vee \omega(\Phi(\text{ED}_n, E, \emptyset), E, \text{trace}) \\
& \quad \Downarrow (\text{induction on first part of Lemma 12.14}) \\
& \quad \omega(\text{ED}_1, E, \text{trace}) \vee \dots \vee \omega(\text{ED}_n, E, \text{trace}) \\
& \quad \quad \Downarrow \omega(\text{EC}, E, \text{trace})
\end{aligned}$$

12.8 Properties of the \preceq relation

Lemma 12.16 *If expr_a conforms to expr_b , the type of expr_a conforms to the type of expr_b .*

$$\text{expr}_a \preceq \text{expr}_b \Rightarrow \Gamma(\text{expr}_a) <: \Gamma(\text{expr}_b)$$

Proof 16 *For this, constructor invocations, and type names, the lemma directly from the definition. For formal parameters, it follows from the definition because we only allow invariant formal parameters. We now prove the fifth and the sixth cases.*

(5) $t.var$

$$\begin{aligned} target_a.var_a \preceq target_b.var_b &\Leftrightarrow target_a \preceq target_b \wedge var_a = var_b \\ &\Downarrow \\ \Gamma(target_a.var_a) &= \Gamma(target_b.var_b) \end{aligned}$$

(6) $t.m(args)$

$$\begin{aligned} \Gamma(t_a.m_a(a_1, \dots, a_n)) &<: \Gamma(t_b.m_b(b_1, \dots, b_n)) \\ &\Updownarrow \\ returnType(method(t_a.m_a(a_1, \dots, a_n))) &<: \\ returnType(method(t_b.m_b(b_1, \dots, b_n))) & \\ &\Updownarrow \text{ (covariant return types)} \\ method(t_a.m_a(a_1, \dots, a_n)) &<: method(t_b.m_b(b_1, \dots, b_n)) \\ &\Updownarrow \text{ (dynamic binding and invariant argument types)} \\ \Gamma(t_a) &<: \Gamma(t_b) \wedge \Gamma(a_1) <: \Gamma(b_1) \wedge \dots \wedge \Gamma(a_n) <: \Gamma(b_n) \\ &\Uparrow \\ t_a \preceq t_b \wedge a_1 \preceq b_1 \wedge \dots \wedge a_n \preceq b_n & \end{aligned}$$

*This last case is the induction step of the proof for method invocations. Because a **Typeable** is a finite tree and a method invocation always has a target, as required by the assumptions, the other cases serve as base cases, which have been proven.*

Lemma 12.17 *If anchored exception declaration $anchor_a$ conforms to $anchor_b$, then the method referenced by $anchor_a$ will always conform to the method referenced by $anchor_b$.*

$$anchor_a \preceq anchor_b \Rightarrow method(anchor_a) <: method(anchor_b)$$

Proof 17 *Because of lemma 12.16, the types of the target and the actual arguments of $anchor_a$ will always conform to the corresponding types of $anchor_b$. Consequently, because we do not allow syntactic overloading and require parameter types to be invariant, $anchor_a$ will always reference a method conform to the method referenced by $anchor_b$.*

12.9 Overview of Dependencies

This section gives an overview of the dependencies in the proofs of Theorems 12.25, 12.30, and 12.36. From Theorem 12.5, it follows that the induction always ends up in a base case. This section merely serves to clarify the reasoning.

Each arrow represents a dependency. The solid arrows represent dependencies that apply the target lemma or theorem directly to the current exception clause or a part of it. The dotted arrows represent dependencies for which the

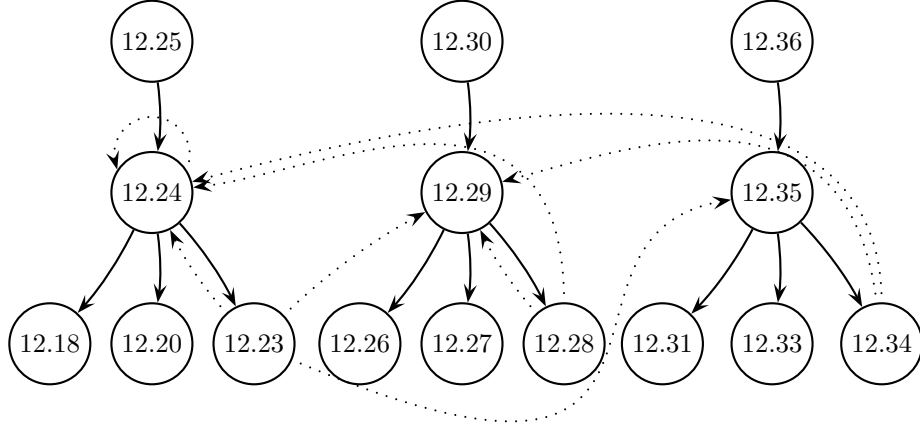


Figure 27: Dependency graph for Theorems 12.25, 12.30, and 12.36.

target lemma or theorem is applied after following an anchored exception declaration. In every lemma or theorem, the anchor that is followed is the one with index a , and it will always be handed to the next theorem or lemma with index a . Because no loop can be made in the dependency graph without using a dotted arrow, the induction process follows a path in the expansion graph of an exception clause. Because of Theorem 12.5, the induction will always end.

12.10 The \preceq relation is transitive

12.10.1 Absolute Exception Declarations

Lemma 12.18

$$\begin{aligned} \Phi((P_a, B_a), E, \emptyset) \preceq (P_b, B_b) \wedge \Phi((P_b, B_b), E, \emptyset) \preceq (P_c, B_c) \\ \Downarrow \\ \Phi((P_a, B_a), E, \emptyset) \preceq (P_c, B_c) \end{aligned}$$

Proof 18

$$\begin{aligned} \Phi((P_a, B_a), E, \emptyset) \preceq (P_b, B_b) \\ \Downarrow \\ ((P_a \sqcap E) - B_a) \sqsubseteq (P_b - B_b) \\ \Downarrow \\ ((P_a \sqcap E) - B_a) \sqsubseteq ((P_b \sqcap E) - B_b) \\ \Downarrow \\ ((P_b \sqcap E) - B_b) \sqsubseteq (P_c - B_c) \\ ((P_a \sqcap E) - B_a) \sqsubseteq (P_c - B_c) \end{aligned}$$

The transitivity of the \sqsubseteq relation follows straightforward from its definition.

12.10.2 Method expressions

Lemma 12.19

$$\begin{aligned} \text{expr}_a \preceq \text{expr}_b \wedge \text{expr}_b \preceq \text{expr}_c \\ \Downarrow \\ \text{expr}_a \preceq \text{expr}_c \end{aligned}$$

Proof 19 *From the definition of \preceq for expressions, it follows that the form of expr_c dictates the form of expr_a and expr_b . Only a type name allows a and b to be of a different form.*

1. $\text{this}_a, \text{this}_b, \text{this}_c$: follows directly from the transitivity of the subtyping ($<:$) relation.
2. $\text{expr}_a, \text{expr}_b, \text{type}_c$: follows from Lemma 12.16 and the transitivity of the subtyping ($<:$) relation.
3. $\text{formal}_a, \text{formal}_b, \text{formal}_c$: follows directly from the definition.
4. $\text{new } C(\text{args})$: follows from the definition and induction on this lemma.
5. $t_a.\text{var}_a, t_b.\text{var}_b, t_c.\text{var}_c$: follows from the definition and induction on this lemma.
6. $t_a.m(\text{args}_a), t_b.m(\text{args}_b), t_c.m(\text{args}_c)$: follows from the definition and induction on this lemma.

12.10.3 Anchored Exception Declarations

Directly conformance

Lemma 12.20

$$\begin{aligned} \Phi(\text{anchor}_a, E, \emptyset) \preceq \text{anchor}_b \wedge \Phi(\text{anchor}_b, E, \emptyset) \preceq \text{anchor}_c \\ \Downarrow \\ \Phi(\text{anchor}_a, E, \emptyset) \preceq \text{anchor}_c \end{aligned}$$

Proof 20 *This lemma follows directly from Lemma 12.19 which proves transitivity for the condition on the method expressions, and the transitivity of the \sqsubseteq relation which proves transitivity for the filter clauses.*

Both direct conformance and conformance after expansion

Lemma 12.21

$$\begin{aligned} \text{trace}_1 \vdash EC_a \preceq EC_b \wedge \text{trace}_1 \subseteq \text{trace}_2 \\ \Downarrow \\ \text{trace}_2 \vdash EC_a \preceq EC_b \end{aligned}$$

Proof 21 If $trace_2$ is a superset of $trace_1$, its analysis will return true while the analysis for $trace_1$ might give false. The reverse can never happen.

Lemma 12.22

$$\begin{aligned} trace \vdash \Phi(\Upsilon(anchor), E, \emptyset) \preceq EC \\ \Downarrow \\ \{\kappa(anchor, EC)\} \cup trace \vdash \Phi(\Upsilon(anchor), E, \emptyset) \preceq EC \end{aligned}$$

Proof 22 Because the method referenced by the anchored exception declaration is analyzed anyway in both cases, it does not matter that in the relation without a trace, the analysis is done a second time (after which it is in the trace and will not be analyzed again if there is no new relevant information).

Lemma 12.23

$$\begin{aligned} \Phi(anchor_a, E, \emptyset) \preceq anchor_b \wedge \\ \{\kappa(anchor_b, EC_c)\} \vdash \Phi(\Upsilon(anchor_b), E, \emptyset) \preceq EC_c \\ \Downarrow \\ \{\kappa(anchor_a, EC_c)\} \notin trace \Rightarrow \\ \{\kappa(anchor_a, EC_c)\} \cup trace \vdash \Phi(\Upsilon(anchor_a), E, \emptyset) \preceq EC_c \end{aligned}$$

Proof 23 Let $AED_a = \text{like } t_a.m(args_a) \preceq P_a \not\preceq B_a$ and $AED_b = \text{like } t_b.m(args_b) \preceq P_b \not\preceq B_b$.

If $\{\kappa(anchor_a, EC_c)\} \in trace$, the Lemma is trivially true. We now prove the lemma for the case $\{\kappa(anchor_a, EC_c)\} \notin trace$.

$$\begin{aligned} \Phi(anchor_a, E, \emptyset) \preceq anchor_b \\ \Downarrow \text{(Lemma 12.27)} \\ \Phi(anchor_a, E, \emptyset) \preceq \Phi(anchor_b, E, \emptyset) \\ \Downarrow \text{(Lemma 12.17 and rule 2)} \\ \varepsilon(\Phi(anchor_a, E, \emptyset)) \preceq \varepsilon(\Phi(anchor_b, E, \emptyset)) \\ \Downarrow \text{(Lemma 12.21)} \\ \{\kappa(anchor_a, EC_c)\} \cup trace \vdash \varepsilon(\Phi(anchor_a, E, \emptyset)) \preceq \varepsilon(\Phi(anchor_b, E, \emptyset)) \\ \Downarrow \text{(induction on Lemma 12.29)} \\ \{\kappa(anchor_a, EC_c)\} \cup trace \vdash \\ \Phi(\varepsilon(\Phi(anchor_a, E, \emptyset)), P_a, B_a) \preceq \Phi(\varepsilon(\Phi(anchor_b, E, \emptyset)), P_b, B_b) \\ \Downarrow \text{(induction on Lemma 12.35)} \\ \{\kappa(anchor_a, EC_c)\} \cup trace \vdash \Omega(\Phi(\varepsilon(\Phi(anchor_a, E, \emptyset)), P_a, B_a), t_a, args_a) \preceq \\ \Omega(\Phi(\varepsilon(\Phi(anchor_b, E, \emptyset)), P_b, B_b), t_b, args_b) \\ \Downarrow \text{(definition of } \Upsilon \text{ and Lemma 12.11)} \\ \{\kappa(anchor_a, EC_c)\} \cup trace \vdash \Phi(\Upsilon(anchor_a), E, \emptyset) \preceq \Phi(\Upsilon(anchor_b), E, \emptyset) \end{aligned}$$

Applying Lemma 12.22 to the second part of the precondition of this Lemma, we get $\Phi(\Upsilon(anchor_b), E, \emptyset) \preceq EC_c$. This complete the precondition for induction on Lemma 12.24. The induction to 12.24 ends either in Lemma 12.18 or 12.20. It cannot get stuck in an infinite loop because we add $\kappa(anchor_a, EC_c)$ to the trace, and we stop if $\kappa(anchor_a, EC_c)$ is in the trace (see Theorem 12.5).

$$\begin{aligned}
& \{\kappa(\text{anchor}_a, EC_c)\} \cup \text{trace} \vdash \Phi(\Upsilon(\text{anchor}_a), E, \emptyset) \preceq EC_c \\
& \quad \Downarrow (\text{assumption}) \\
& \{\kappa(\text{anchor}_a, EC)\} \notin \text{trace} \Rightarrow \\
& \quad \{\kappa(\text{anchor}_a, EC)\} \cup \text{trace} \vdash \Phi(\Upsilon(\text{anchor}_a), E, \emptyset) \preceq EC_c
\end{aligned}$$

The inductions on Lemmas 12.29, 12.35, and 12.24 will end because they all go back to Lemma 12.24 after performing an expansion and keeping a trace. Therefore, every branch eventually ends up in Lemmas 12.18 or 12.20, or at the stopping condition of this lemma.

12.10.4 Exception Clauses

Lemma 12.24

$$\text{trace} \vdash EC_a \preceq EC_b \wedge EC_b \preceq EC_c \Rightarrow \text{trace} \vdash EC_a \preceq EC_c$$

Proof 24 We must prove that:

$$\left(\begin{array}{l} \forall (P_a, B_a) \in EC_a, \forall E, \omega((P_a, B_a), E) : \\ \quad \exists (P_c, B_c) \in EC_c : \\ \quad \quad \Phi((P_a, B_a), E, \emptyset) \preceq (P_c, B_c) \end{array} \right) \wedge \\
\left(\begin{array}{l} \forall \text{anchor}_a \in EC_a, \forall E : \omega(\text{anchor}_a, E) : \\ \quad (\exists \text{anchor}_c \in EC_c : \\ \quad \quad (\Phi(\text{anchor}_a, E, \emptyset) \preceq \text{anchor}_c \vee \\ \quad \quad \quad \{\kappa(\text{anchor}_a, EC_c)\} \notin \text{trace} \Rightarrow \\ \quad \quad \quad \{\kappa(\text{anchor}_a, EC_c)\} \cup \text{trace} \vdash \Phi(\Upsilon(\text{anchor}_a), E, \emptyset) \preceq EC_c)) \end{array} \right)$$

The case for absolute exception declarations is proven by Lemma 12.18. For anchored exception declarations of EC_a that directly conform to anchored exception declaration of EC_b , the proof is given by Lemmas 12.20 and 12.23. For the case where $\Upsilon(AED_a) \preceq EC_b$, we apply induction on this lemma.

Theorem 12.25 The \preceq relation for exception clauses is transitive.

$$EC_a \preceq EC_b \wedge EC_b \preceq EC_c \Rightarrow EC_a \preceq EC_c$$

Proof 25 The proof follows directly from Lemma 12.24.

12.11 Φ is monotone

The Φ function maintains the order between two exception clauses or exception declarations when the same types are filtered from both. It also maintains the order when the smaller clause or declaration is filtered with stronger arguments (allowing less types to be propagated and blocking more types).

12.11.1 Absolute Exception Declarations

Lemma 12.26

$$\begin{aligned} (P_c - B_c) \sqsubseteq (P_d - B_d) \wedge \Phi((P_a, B_a), E, \emptyset) \preceq (P_b, B_b) \\ \Downarrow \\ \Phi(\Phi((P_a, B_a), P_c, B_c), E, \emptyset) \preceq \Phi((P_b, B_b), P_d, B_d) \end{aligned}$$

Proof 26

$$\begin{aligned} \Phi(\Phi((P_a, B_a), P_c, B_c), E, \emptyset) \preceq \Phi((P_b, B_b), P_d, B_d) \\ \Downarrow \text{(definition of } \Phi) \\ (P_a \sqcap (P_c \sqcap E), B_a \sqcup B_c) \preceq (P_b \sqcap P_d, B_b \sqcup B_d) \\ \Downarrow \text{(definition of } \preceq) \\ (((P_a \sqcap E) \sqcap P_c) - (B_a \sqcup B_c)) \sqsubseteq ((P_b \sqcap P_d) - (B_b \sqcup B_d)) \\ \Uparrow \text{(Lemma 12.1)} \\ (P_a - B_a) \sqsubseteq (P_b - B_b) \wedge (P_c - B_c) \sqsubseteq (P_d - B_d) \\ \Downarrow \text{(definition of } \preceq) \\ (P_a \sqcap E, B_a) \preceq (P_b, B_b) \wedge (P_c - B_c) \sqsubseteq (P_d - B_d) \end{aligned}$$

12.11.2 Anchored Exception Declarations

Direct compatibility

Lemma 12.27

$$\begin{aligned} \Phi(\text{anchor}_a, E, \emptyset) \preceq \text{anchor}_b \wedge (P_c - B_c) \sqsubseteq (P_d - B_d) \\ \Downarrow \\ \Phi(\Phi(\text{anchor}_a, P_c, B_c), E, \emptyset) \preceq \Phi(\text{anchor}_b, P_d, B_d) \end{aligned}$$

Proof 27

$$\begin{aligned} \text{like } t_a.m_a(\text{args}_a) \preceq (P_a \sqcap E) \not\sqsubseteq B_a \preceq \text{like } t_b.m_b(\text{args}_b) \preceq P_b \not\sqsubseteq B_b \wedge \\ (P_c - B_c) \sqsubseteq (P_d - B_d) \\ \Downarrow \text{(definition of } \preceq) \\ t_a.m_a(\text{args}_a) \preceq t_b.m_b(\text{args}_b) \wedge \\ ((P_a \sqcap E) - B_a) \sqsubseteq (P_b - B_b) \wedge (P_c - B_c) \sqsubseteq (P_d - B_d) \\ \\ \Downarrow \text{(lemma 12.1)} \\ t_a.m_a(\text{args}_a) \preceq t_b.m_b(\text{args}_b) \wedge \\ (((P_a \sqcap E) \sqcap P_c) - (B_a \sqcup B_c)) \sqsubseteq ((P_b \sqcap P_d) - (B_b \sqcup B_d)) \\ \Downarrow \text{(definitions of } \preceq \text{ and } \sqcup) \\ \text{like } t_a.m_a(\text{args}_a) \preceq (P_a \sqcap E \sqcap P_c) \not\sqsubseteq (B_a \sqcup E \sqcup B_c) \preceq \\ \text{like } t_b.m_b(\text{args}_b) \preceq (P_b \sqcap P_d) \not\sqsubseteq (B_b \sqcup B_d) \\ \Downarrow \text{(definition of } \Phi) \\ \Phi(\Phi(AED_a, P_c, B_c), E, \emptyset) \preceq \Phi(AED_b, P_d, B_d) \end{aligned}$$

Compatibility After Expansion

Lemma 12.28

$$\left(\begin{array}{c} (P_a - B_a) \sqsubseteq (P_b - B_b) \wedge \\ \kappa(\text{anchor}_a, EC_b) \notin \text{trace} \Rightarrow \\ \{\kappa(\text{anchor}_a, EC_b)\} \cup \text{trace} \vdash \Upsilon(\text{anchor}_a), E, \emptyset \preceq EC_b \end{array} \right) \Downarrow \left(\begin{array}{c} \kappa(\text{anchor}_a, EC_b) \notin \text{trace} \Rightarrow \\ \{\kappa(\text{anchor}_a, EC_b)\} \cup \text{trace} \vdash \\ \Phi(\Upsilon(\Phi(\text{anchor}_a, P_a, B_a)), E, \emptyset) \preceq \Phi(EC_b, P_b, B_b) \end{array} \right)$$

Proof 28 We prove this using induction on Lemma 12.29. We expand the anchored exception declaration one level and assume that Lemma 12.29 holds for the resulting exception clause and EC_B . The exception clause resulting from the expansion is the exception clause of the method referenced by anchor, or one of its submethods, with context information inserted. Because we keep a trace, the recursion must end in methods of which the exception clauses contain no anchored exception declarations, or if $\kappa(\text{anchor}_a, EC_b) \in \text{trace}$.

If $\kappa(\text{anchor}_a, EC_b) \in \text{trace}$, the lemma is trivially true. We now prove the lemma for $\kappa(\text{anchor}_a, EC_b) \notin \text{trace}$.

The preconditions of Theorem 12.29 follow directly from the preconditions of this lemma.

$$\begin{array}{c} \Phi(\{\kappa(\text{anchor}_a, EC_b)\} \cup \text{trace} \vdash \Upsilon(\text{anchor}_a), E, \emptyset) \preceq EC_b \\ \Downarrow (\text{induction on Lemma 12.29}) \\ \Phi(\{\kappa(\text{anchor}_a, EC_b)\} \cup \text{trace} \vdash \\ \Phi(\Upsilon(\text{anchor}_a), E, \emptyset), P_a, B_a) \preceq \Phi(EC_b, P_b, B_b) \\ \Downarrow (\text{induction on Lemma 12.24}) \\ \left(\begin{array}{c} \Phi(\{\kappa(\text{anchor}_a, EC_b)\} \cup \text{trace} \vdash \\ \Phi(\Upsilon(\Phi(\text{anchor}_a, P_a, B_a)), E, \emptyset) \preceq \Phi(\Phi(\Upsilon(\text{anchor}_a), E, \emptyset), P_a, B_a)) \\ \Downarrow \\ \Phi(\{\kappa(\text{anchor}_a, EC_b)\} \cup \text{trace} \vdash \\ \Phi(\Upsilon(\Phi(\text{anchor}_a, P_a, B_a)), E, \emptyset) \preceq \Phi(EC_b, P_b, B_b)) \end{array} \right) \end{array}$$

As explained in the proof of Theorem 12.24 the transitivity property of \preceq is indirectly based on this lemma. Because we keep add $\kappa(\text{anchor}_a, EC_b)$ to the trace, and stop if we receive it again, the induction must end (see Theorem 12.5). On this side, it will end in either Lemma 12.26 or 12.27. Now we only need to prove the left-hand side of the last implication.

$$\begin{array}{c} \text{anchor}_a = \text{like } t.m(\text{args}) \preceq P \not\triangleleft B \\ \Downarrow \\ \Phi(\text{anchor}_a, P_a, B_a) = \text{like } t.m(\text{args}) \preceq (P \sqcap P_a) \not\triangleleft (B \sqcup B_a) \end{array}$$

$$\begin{aligned}
& \Phi(\{\kappa(anchor_a, EC_b)\} \cup trace \vdash \\
& \Phi(\Upsilon(\Phi(anchor_a, P_a, B_a)), E, \emptyset) \preceq \Phi(\Phi(\Upsilon(anchor_a), E, \emptyset), P_a, B_a) \\
& \quad \Downarrow \text{(definition of } \Upsilon \text{ and } \Phi) \\
& \Phi(\{\kappa(anchor_a, EC_b)\} \cup trace \vdash \\
& \Phi(\Omega(\Phi(\varepsilon(\Phi(anchor, P_a, B_a)), (P \sqcap P_a), (B \sqcup B_a)), t, args), E, \emptyset) \preceq \\
& \Phi(\Omega(\Phi(\varepsilon(anchor), P, B), t, args), (P_a \sqcap E), B_a)
\end{aligned}$$

Because Φ does not alter the method expression, it does not have any effect on the ε function.

$$\begin{aligned}
& \quad \Downarrow \\
& \Phi(\{\kappa(anchor_a, EC_b)\} \cup trace \vdash \\
& \Phi(\Omega(\Phi(\varepsilon(anchor), (P \sqcap P_a), (B \sqcup B_a)), t, args), E, \emptyset) \preceq \\
& \Phi(\Omega(\Phi(\varepsilon(anchor), P, B), t, args), (P_a \sqcap E), B_a) \\
& \quad \Downarrow \text{(Lemma 12.11)} \\
& \Phi(\{\kappa(anchor_a, EC_b)\} \cup trace \vdash \\
& \Omega(\Phi(\Phi(\varepsilon(anchor), (P \sqcap P_a), (B \sqcup B_a)), E, \emptyset), t, args) \preceq \\
& \Omega(\Phi(\Phi(\varepsilon(anchor), P, B), (P_a \sqcap E), B_a), t, args) \\
& \quad \Downarrow \text{(definitions of } \Phi, \sqcup \text{ and } \sqcap) \\
& \Phi(\{\kappa(anchor_a, EC_b)\} \cup trace \vdash \\
& \Omega(\Phi(\varepsilon(anchor), (P \sqcap P_a \sqcap E), (B \sqcup B_a)), t, args) \preceq \\
& \Omega(\Phi(\varepsilon(anchor), (P \sqcap P_a \sqcap E), (B \sqcup B_a)), t, args) \\
& \quad \Downarrow \text{(definition of } \vdash \preceq) \\
& \text{true}
\end{aligned}$$

12.11.3 Exception Clauses

Lemma 12.29

$$\begin{aligned}
(P_c - B_c) \sqsubseteq (P_d - B_d) \wedge trace \vdash EC_a \preceq EC_b \\
\Downarrow \\
trace \vdash \Phi(EC_a, P_c, B_c) \preceq \Phi(EC_b, P_d, B_d)
\end{aligned}$$

Proof 29

$$\begin{array}{c}
\text{trace} \vdash \Phi(EC_a, P_c, B_c) \preceq \Phi(EC_b, P_d, B_d) \\
\Downarrow (\text{definition of } \preceq) \\
\left(\begin{array}{l}
\forall \Phi((P_a, B_a), P_c, B_c) \in \Phi(EC_a, P_c, B_c), \forall E, \omega(\Phi((P_a, B_a), P_c, B_c), E) : \\
\exists \Phi((P_b, B_b), P_d, B_d) \in \Phi(EC_b, P_d, B_d) : \\
\Phi(\Phi((P_a, B_a), P_c, B_c), E, \emptyset) \preceq \Phi((P_b, B_b), P_d, B_d) \\
\wedge \\
\forall \Phi(\text{anchor}_a, P_c, B_c) \in \Phi(EC_a, P_c, B_c), \\
\forall E : \omega(\Phi(\text{anchor}_a, P_c, B_c), E) : \\
(\exists \Phi(\text{anchor}_b, P_d, B_d) \in \Phi(EC_b, P_d, B_d) : \\
\Phi(\Phi(\text{anchor}_a, P_c, B_c), E, \emptyset) \preceq \Phi(\text{anchor}_b, P_d, B_d) \vee \\
\kappa(\text{anchor}_a, EC_b) \notin \text{trace} \Rightarrow \\
\{\kappa(\text{anchor}_a, EC_b)\} \cup \text{trace} \vdash \\
\Phi(\Upsilon(\Phi(\text{anchor}_a, P_c, B_c)), E, \emptyset) \preceq \Phi(EC_b, P_d, B_d))
\end{array} \right)
\end{array}$$

$$\begin{array}{c}
\Uparrow (\text{trace} \vdash EC_a \preceq EC_b) \\
\left(\begin{array}{l}
\left((P_c - B_c) \sqsubseteq (P_d - B_d) \wedge \Phi(ABS_a, E, \emptyset) \preceq ABS_b \right) \\
\Downarrow \\
\Phi(\Phi(ABS_a, P_c, B_c), E, \emptyset) \preceq \Phi(ABS_b, P_d, B_d) \\
\wedge \\
\left((P_c - B_c) \sqsubseteq (P_d - B_d) \wedge \Phi(\text{anchor}_a, E, \emptyset) \preceq \text{anchor}_b \right) \\
\Downarrow \\
\Phi(\Phi(\text{anchor}_a, P_c, B_c), E, \emptyset) \preceq \Phi(\text{anchor}_b, P_d, B_d) \\
\wedge \\
\left(\begin{array}{l}
(P_c - B_c) \sqsubseteq (P_d - B_d) \wedge \\
\kappa(\text{anchor}_a, EC_b) \notin \text{trace} \Rightarrow \\
\{\kappa(\text{anchor}_a, EC_b)\} \cup \text{trace} \vdash \Phi(\Upsilon(\text{anchor}_a), E, \emptyset) \preceq EC_b
\end{array} \right) \\
\Downarrow \\
\left(\begin{array}{l}
\kappa(\text{anchor}_a, EC_b) \notin \text{trace} \Rightarrow \\
\{\kappa(\text{anchor}_a, EC_b)\} \cup \text{trace} \vdash \\
\Phi(\Upsilon(\Phi(\text{anchor}_a, P_c, B_c)), E, \emptyset) \preceq \Phi(EC_b, P_d, B_d)
\end{array} \right)
\end{array} \right)
\end{array}$$

\Downarrow
 Lemma 12.26 \wedge Lemma 12.27 \wedge Lemma 12.28

Theorem 12.30

$$\begin{array}{c}
(P_c - B_c) \sqsubseteq (P_d - B_d) \wedge EC_a \preceq EC_b \\
\Downarrow \\
\Phi(EC_a, P_c, B_c) \preceq \Phi(EC_b, P_d, B_d)
\end{array}$$

Proof 30 *The proof follows directly from Lemma 12.29.*

12.12 Ω is monotone

In this section we prove the same property for the Ω function.

12.12.1 Absolute Exception Declarations

Lemma 12.31

$$\begin{aligned} \Phi((P_a, B_a), E, \emptyset) &\preceq (P_b, B_b) \\ &\Downarrow \\ \Phi(\Omega((P_a, B_a), pre_a, a_1 \dots a_n), E, \emptyset) &\preceq \Omega((P_b, B_b), pre_b, b_1 \dots b_n) \end{aligned}$$

Proof 31

$$\begin{aligned} \Phi(\Omega((P_a, B_a), pre_a, a_1 \dots a_n), E, \emptyset) &\preceq \Omega((P_b, B_b), pre_b, b_1 \dots b_n) \\ &\Updownarrow \text{(definition of } \Omega \text{)} \\ \Phi((P_a, B_a), E, \emptyset) &\preceq (P_b, B_b) \end{aligned}$$

12.12.2 Method Expressions

Lemma 12.32

$$\begin{aligned} &expr_a \preceq expr_b \wedge pre_a \preceq pre_b \wedge args_a \preceq args_b \wedge \\ &ok_\Omega(args_a, (pre_a, this(expr_a))) \wedge ok_\Omega(args_b, (pre_b, this(expr_b))) \\ &\Downarrow \\ &\Omega(expr_a, pre_a, args_a) \preceq \Omega(expr_b, pre_b, args_b) \end{aligned}$$

Proof 32 Let $args_a = a_1 \dots a_n$ and $args_b = b_1 \dots b_n$.

1. *this*

$$\begin{aligned} \Omega(this_a, pre_a, a_1 \dots a_n) &\preceq \Omega(this_b, pre_b, b_1 \dots b_n) \\ &\Updownarrow \text{(definition of } \Omega \text{)} \\ pre_a &\preceq pre_b \end{aligned}$$

2. *type*

$$\begin{aligned} \Omega(type_a, pre_a, a_1 \dots a_n) &\preceq \Omega(type_b, pre_b, b_1 \dots b_n) \\ &\Updownarrow \text{(definition of } \Omega \text{)} \\ type_a &\preceq type_b \end{aligned}$$

3. *formal*

$$\begin{aligned} \Omega(formal_a, pre_a, (v_{a,1}, p_{a,1}) \dots (v_{a,n}, p_{a,n})) &\preceq \\ \Omega(formal_b, pre_b, (v_{b,1}, p_{b,1}) \dots (v_{b,n}, p_{b,n})) &\end{aligned}$$

(a) $formal_a = p_{a,i}$

Because of the definition of the \preceq relation and the given assumptions,
 $formal_b = p_{b,i}$.

$$v_{a,i} \preceq v_{b,i}$$

(b) $\text{formal}_a \neq p_{a,i}$
 Because of the definition of the \preceq relation and the given assumptions,
 $\text{formal}_b \neq p_{b,i}$.
 $\text{formal}_a \preceq \text{formal}_b$

4. $\text{new } C(\text{args})$

$$\begin{aligned} \Omega(\text{new } C(\text{args}_a), \text{pre}_a, a_1 \dots a_n) &\preceq \Omega(\text{new } C(\text{args}_b), \text{pre}_b, b_1 \dots b_n) \\ &\Downarrow (\text{definition of } \Omega) \\ \Omega(\text{args}_a, \text{pre}_a, a_1 \dots a_n) &\preceq \Omega(\text{args}_b, \text{pre}_b, b_1 \dots b_n) \\ &\Downarrow (\text{induction on finite expression tree}) \\ &\text{true} \end{aligned}$$

5. $t.\text{var}$

$$\begin{aligned} \Omega(t_a.\text{var}_a, \text{pre}_a, a_1 \dots a_n) &\preceq \Omega(t_b.\text{var}_b, \text{pre}_b, b_1 \dots b_n) \\ &\Downarrow (\text{definition of } \Omega) \\ \Omega(t_a, \text{pre}_a, a_1 \dots a_n).\text{var}_a &\preceq \Omega(t_b, \text{pre}_b, b_1 \dots b_n).\text{var}_b \\ &\Downarrow (\text{definition of } \preceq) \\ \Omega(t_a, \text{pre}_a, a_1 \dots a_n) &\preceq \Omega(t_b, \text{pre}_b, b_1 \dots b_n) \wedge \text{var}_a \preceq \text{var}_b \\ &\Downarrow (\text{induction on finite expression tree}) \\ &\text{true} \end{aligned}$$

6. $t.m(\text{args})$

$$\begin{aligned} \Omega(t_a.m(\text{arg}_{a,1}, \dots, \text{arg}_{a,n}), \text{pre}_a, a_1 \dots a_n) &\preceq \\ \Omega(t_b.m(\text{arg}_{b,1}, \dots, \text{arg}_{b,n}), \text{pre}_b, b_1 \dots b_n) & \\ &\Downarrow (\text{definition of } \Omega) \\ \Omega(t_a, \text{pre}_a, a_1 \dots a_n).m(\Omega(\text{arg}_{a,1}, \text{pre}_a, a_1 \dots a_n), \dots & \\ , \Omega(\text{arg}_{a,n}, \text{pre}_a, a_1 \dots a_n)) &\preceq \\ \Omega(t_b, \text{pre}_b, b_1 \dots b_n).m(\Omega(\text{arg}_{b,1}, \text{pre}_b, b_1 \dots b_n), \dots & \\ , \Omega(\text{arg}_{b,n}, \text{pre}_b, b_1 \dots b_n)) & \\ &\Downarrow (\text{definition of } \preceq) \\ \Omega(t_a, \text{pre}_a, a_1 \dots a_n) &\preceq \Omega(t_b, \text{pre}_b, b_1 \dots b_n) \wedge \\ \Omega(\text{arg}_{a,1}, \text{pre}_a, a_1 \dots a_n) &\preceq \Omega(\text{arg}_{b,1}, \text{pre}_b, b_1 \dots b_n) \wedge \dots \wedge \\ \Omega(\text{arg}_{a,n}, \text{pre}_a, a_1 \dots a_n) &\preceq \Omega(\text{arg}_{b,n}, \text{pre}_b, b_1 \dots b_n) \\ &\Downarrow (\text{induction on finite expression tree}) \\ &\text{true} \end{aligned}$$

12.12.3 Anchored Exception Declarations

Direct Compatibility

Lemma 12.33

$$\begin{aligned} \text{pre}_a \preceq \text{pre}_b \wedge \text{args}_a \preceq \text{args}_b \wedge \Phi(\text{anchor}_a, E, \emptyset) &\preceq \text{anchor}_b \\ \text{ok}_\Omega(\text{args}_a, (\text{pre}_a, \text{this}(\text{anchor}_a))) \wedge \text{ok}_\Omega(\text{args}_b, (\text{pre}_b, \text{this}(\text{anchor}_b))) & \\ \Downarrow & \\ \Phi(\Omega(\text{anchor}_a, \text{pre}_a, \text{args}_a), E, \emptyset) &\preceq \Omega(\text{anchor}_b, \text{pre}_b, \text{args}_b) \end{aligned}$$

Proof 33 Let $anchor_a = \text{like } t_a.m_a(a_1, \dots, a_n) \trianglelefteq P_a \not\trianglelefteq B_a$, and let $anchor_b = \text{like } t_b.m_b(b_1, \dots, b_n) \trianglelefteq P_b \not\trianglelefteq B_b$.

$$\begin{aligned}
& \Phi(\Omega(anchor_a, pre_a, args_a), E, \emptyset) \preceq \Omega(anchor_b, pre_b, args_b) \\
& \quad \Downarrow \\
& \text{like } \Omega(t_a.m_a(a_1 \dots a_n), pre_a, args_a) \trianglelefteq (P_a \sqcap E) \not\trianglelefteq B_a \preceq \\
& \quad \text{like } \Omega(t_b.m_b(b_1 \dots b_n), pre_b, args_b) \trianglelefteq P_b \not\trianglelefteq B_b \\
& \quad \Downarrow (\text{definition of } \preceq) \\
& \Omega(t_a.m_a(a_1 \dots a_n), pre_a, args_a) \preceq \Omega(t_b.m_b(b_1 \dots b_n), pre_b, args_b) \wedge \\
& \quad ((P_a \sqcap E) - B_a) \sqsubseteq (P_b - B_b) \\
& \quad \Uparrow (\text{Lemma 12.32 and preconditions}) \\
& \quad ((P_a \sqcap E) - B_a) \sqsubseteq (P_b - B_b) \\
& \quad \Downarrow (\Phi(anchor_a, E, \emptyset) \preceq anchor_b) \\
& \quad \text{true}
\end{aligned}$$

Compatibility After Expansion

Lemma 12.34 Let $anchor = \text{like } t.m(arg_1, \dots, arg_m)$.

$$\begin{aligned}
& \left(\begin{array}{c} pre_a \preceq pre_b \wedge args_a \preceq args_b \wedge \\ \kappa(anchor, EC_b) \notin trace \Rightarrow \\ \left(\{ \kappa(anchor, EC_b) \} \cup trace \vdash \Phi(\Upsilon(anchor), E, \emptyset) \preceq EC_b \right) \wedge \\ ok_{\Omega}(args_a, (pre_a, this(anchor))) \wedge ok_{\Omega}(args_b, (pre_b, this(EC_b))) \end{array} \right) \\
& \quad \Downarrow \\
& \left(\begin{array}{c} \kappa(anchor, EC_b) \notin trace \Rightarrow \\ \{ \kappa(anchor, EC_b) \} \cup trace \vdash \\ \Phi(\Upsilon(\Omega(anchor, pre_a, args_a)), E, \emptyset) \preceq \Omega(EC_b, pre_b, args_b) \end{array} \right)
\end{aligned}$$

Proof 34 We prove the lemma using induction on Lemmas 12.29 and 12.35. We expand the anchored exception declaration one level, or go to the exception clause of a submethod of the method referenced by $anchor$, and assume that the lemmas hold for the resulting exception clause and EC_B . Because we add $\kappa(anchor, EC_b)$ to the trace and stop if we receive it again, and because these other lemmas themselves only perform further expansions, this induction must end in methods whose exception clauses contain no anchored exception declarations or in the trivial case where $anchor$ is not processed.

If $\kappa(anchor, EC_b) \in trace$, the lemma is trivially true. We now prove the lemma for $\kappa(anchor, EC_b) \notin trace$.

Before we perform the induction on $\Upsilon(anchor)$ and EC_b , we need to verify that the precondition of Lemma 12.35 is satisfied. The first three preconditions follow directly from the preconditions of this lemma. The fourth precondition is satisfied because the type of $this$ in $\Upsilon(anchor)$ is the same as the type of $this$ in $anchor$. The last preconditions follow directly from the preconditions of this

lemma.

$$\begin{array}{c}
\{\kappa(\text{anchor}, EC_b)\} \cup \text{trace} \vdash \Phi(\Upsilon(\text{anchor}), E, \emptyset) \preceq EC_b \\
\Downarrow (\text{induction on Theorem 12.35}) \\
\{\kappa(\text{anchor}, EC_b)\} \cup \text{trace} \vdash \\
\Omega(\Phi(\Upsilon(\text{anchor}), E, \emptyset), \text{pre}_a, \text{args}_a) \preceq \Omega(EC_b, \text{pre}_b, \text{args}_b) \\
\Downarrow (\text{induction on Theorem 12.25}) \\
\left(\begin{array}{c}
\{\kappa(\text{anchor}, EC_b)\} \cup \text{trace} \vdash \\
\Phi(\Upsilon(\Omega(\text{anchor}, \text{pre}_a, \text{args}_a)), E, \emptyset) \preceq \\
\Omega(\Phi(\Upsilon(\text{anchor}), E, \emptyset), \text{pre}_a, \text{args}_a) \\
\Downarrow \\
\{\kappa(\text{anchor}, EC_b)\} \cup \text{trace} \vdash \\
\Phi(\Upsilon(\Omega(\text{anchor}, \text{pre}_a, \text{args}_a)), E, \emptyset) \preceq \Omega(EC_b, \text{pre}_b, \text{args}_b)
\end{array} \right)
\end{array}$$

As explained in the proof of Lemma 12.24 the transitivity property of \preceq is indirectly based on this lemma. Because of the expansion done in this lemma and because we keep a trace, the induction must end (see Theorem 12.5). On this side, it will end in either Lemma 12.31 or 12.33. Now we only need to prove the left-hand side of the last implication. From the definitions of Ω and Ω , we know that:

$$\begin{aligned}
\Omega(\text{anchor}, \text{pre}_a, \text{args}_a) &= \text{like } \Omega(t, \text{pre}_a, \text{args}_a).m(\Omega(\text{arg}_1, \text{pre}_a, \text{args}_a), \\
&\dots, \Omega(\text{arg}_m, \text{pre}_a, \text{args}_a)) \preceq P \not\triangleleft B
\end{aligned}$$

The actual arguments $\text{arg}_1, \dots, \text{arg}_m$ are bound respectively to formal parameters $\text{par}_1, \dots, \text{par}_m$. As a result, we can prove the induction step as follows:

$$\begin{aligned}
&\{\kappa(\text{anchor}, EC_b)\} \cup \text{trace} \vdash \\
&\Phi(\Upsilon(\Omega(\text{anchor}, \text{pre}_a, \text{args}_a)), E, \emptyset) \preceq \\
&\Omega(\Phi(\Upsilon(\text{anchor}), E, \emptyset), \text{pre}_a, \text{args}_a) \\
&\Updownarrow (\text{definition of } \Upsilon) \\
&\{\kappa(\text{anchor}, EC_b)\} \cup \text{trace} \vdash \\
&\Phi(\Omega(\Phi(\varepsilon(\Omega(\text{anchor}, \text{pre}_a, \text{args}_a)), P, B), \Omega(t, \text{pre}_a, \text{args}_a), \\
&\Omega((\text{arg}_1, \text{par}_1), \text{pre}_a, \text{args}_a) \dots \Omega((\text{arg}_m, \text{par}_m), \text{pre}_a, \text{args}_a)), E, \emptyset) \preceq \\
&\Omega(\Phi(\Omega(\Phi(\varepsilon(\text{anchor}), P, B), t, \\
&(\text{arg}_1, \text{par}_1) \dots (\text{arg}_m, \text{par}_m)), E, \emptyset), \text{pre}_a, \text{args}_a)
\end{aligned}$$

Because $\varepsilon(\text{anchor})$ is the exception clause of a method of the program, it can only reference the formal parameters of its method, being $\text{par}_1, \dots, \text{par}_m$. As a result, Lemma 12.13 may be applied. The filter operations may be merged due to Lemma 12.11 and the definition of Φ .

$$\begin{aligned}
&\Updownarrow (\text{Lemma 12.13}) \\
&\{\kappa(\text{anchor}, EC_b)\} \cup \text{trace} \vdash \\
&\Omega(\Phi(\varepsilon(\Omega(\text{anchor}, \text{pre}_a, \text{args}_a)), P \sqcap E, B), \Omega(t, \text{pre}_a, \text{args}_a), \\
&\Omega((\text{arg}_1, \text{par}_1), \text{pre}_a, \text{args}_a) \dots \Omega((\text{arg}_m, \text{par}_m), \text{pre}_a, \text{args}_a)) \preceq \\
&\Omega(\Phi(\varepsilon(\text{anchor}), P \sqcap E, B), \Omega(t, \text{pre}_a, \text{args}_a), \\
&\Omega((\text{arg}_1, \text{par}_1), \text{pre}_a, \text{args}_a) \dots \Omega((\text{arg}_m, \text{par}_m), \text{pre}_a, \text{args}_a))
\end{aligned}$$

Because of Lemma 12.16, Lemma 12.32, and the preconditions of this lemma, we know that:

$$\text{method}(\Omega(\text{anchor}, \text{pre}_a, \text{args}_a)) <: \text{method}(\text{anchor})$$

As a result, we know that according to rule 2:

$$\begin{aligned} \varepsilon(\Omega(\text{anchor}, \text{pre}_a, \text{args}_a)) &\preceq \varepsilon(\text{anchor}) \\ &\Downarrow (\text{Lemma 12.21}) \\ \{\kappa(\text{anchor}, EC_b)\} \cup \text{trace} &\vdash \\ \varepsilon(\Omega(\text{anchor}, \text{pre}_a, \text{args}_a)) &\preceq \varepsilon(\text{anchor}) \end{aligned}$$

Now we use induction on Lemmas 12.29 and 12.35 to prove the induction step. All that is left is proving that their preconditions are satisfied.

1. For the application of Φ , the preconditions of Theorem 12.30 are met because both sides use the same sets of types and the \preceq relation above.
2. For the application of Ω , the first precondition of Lemma 12.35 follows from the application of Lemma 12.29. The second and third preconditions are satisfied because the prefixes and actual arguments are identical. The last preconditions are satisfied because of Lemmas 12.16 and 12.32.

12.12.4 Exception Clauses

Lemma 12.35

$$\begin{aligned} \text{trace} \vdash EC_a \preceq EC_b \wedge \text{pre}_a \preceq \text{pre}_b \wedge \text{args}_a \preceq \text{args}_b \wedge \\ \text{ok}_\Omega(\text{args}_s, (\text{pre}_a, \text{this}(EC_a))) \wedge \text{ok}_\Omega(\text{args}_b, (\text{pre}_b, \text{this}(EC_b))) \\ \Downarrow \\ \text{trace} \vdash \Omega(EC_a, \text{pre}_a, \text{args}_a) \preceq \Omega(EC_b, \text{pre}_b, \text{args}_b) \end{aligned}$$

Proof 35 The proof of this lemma is similar to that of Lemma 12.29. After rewriting the expression $\text{trace} \vdash \Omega(EC_a, \text{pre}_a, \text{args}_a) \preceq \Omega(EC_b, \text{pre}_b, \text{args}_b)$, we

obtain the following conditions for this lemma to be true:

$$\begin{array}{c}
\left(\begin{array}{c} \Phi(ABS_a, E, \emptyset) \preceq ABS_b \\ \downarrow \\ \Phi(\Omega(ABS_a, pre_a, args_a), E, \emptyset) \preceq \Omega(ABS_b, pre_b, args_b) \end{array} \right) \\
\wedge \\
\left(\begin{array}{c} pre_a \preceq pre_b \wedge args_a \preceq args_b \wedge \Phi(anchor_a, E, \emptyset) \preceq anchor_b \\ \wedge ok_{\Omega}(args_a, (pre_a, this(anchor_a))) \wedge ok_{\Omega}(args_b, (pre_b, this(anchor_b))) \end{array} \right) \\
\downarrow \\
\left(\begin{array}{c} \Phi(\Omega(anchor_a, pre_a, args_a), E, \emptyset) \preceq \Omega(anchor_b, pre_b, args_b) \end{array} \right) \\
\wedge \\
\left(\begin{array}{c} pre_a \preceq pre_b \wedge args_a \preceq args_b \wedge \\ \left(\begin{array}{c} \kappa(anchor, EC_b) \notin trace \Rightarrow \\ \{ \kappa(anchor, EC_b) \} \cup trace \vdash \Phi(\Upsilon(anchor), E, \emptyset) \preceq EC_b \end{array} \right) \wedge \\ ok_{\Omega}(args_a, (pre_a, this(anchor))) \wedge ok_{\Omega}(args_b, (pre_b, this(EC_b))) \end{array} \right) \\
\downarrow \\
\left(\begin{array}{c} \kappa(anchor, EC_b) \notin trace \Rightarrow \\ \{ \kappa(anchor, EC_b) \} \cup trace \vdash \\ \Phi(\Upsilon(\Omega(anchor, pre_a, args_a)), E, \emptyset) \preceq \Omega(EC_b, pre_b, args_b) \end{array} \right)
\end{array}$$

\Updownarrow

Lemma 12.31 \wedge Lemma 12.33 \wedge Lemma 12.34

Theorem 12.36

$$\begin{array}{c}
EC_a \preceq EC_b \wedge pre_a \preceq pre_b \wedge args_a \preceq args_b \wedge \\
ok_{\Omega}(args_s, (pre_a, this(EC_a))) \wedge ok_{\Omega}(args_b, (pre_b, this(EC_b))) \\
\downarrow \\
\Omega(EC_a, pre_a, args_a) \preceq \Omega(EC_b, pre_b, args_b)
\end{array}$$

Proof 36 *The proof follows directly from Lemma 12.35.*

12.13 The Implementation Exception Clause is an Upper Bound

Theorem 12.37 *The implementation exception clause of a non-abstract method is an upper bound for the exceptional behaviour of the implementation of that method.*

Proof 37 *This theorem follow obviously from the definition of the implementation exception clause and the Java Language Specification.*

12.14 Method Invocations Maintain Compatibility

Theorem 12.38 *Let $t.m(arg_1, \dots, arg_n)$ be a method invocation in a valid program, let $EC_b = \varepsilon(t.m(arg_1, \dots, arg_n))$ and let par_i be the formal parameter*

$$\begin{aligned}
& \tau(\text{like } \{T_t, e_{t,1}, \dots, e_{t,n}\}.m(\{T_{a,1}, a_{1,1}, \dots, a_{1,k_1}\}, \dots, \{T_a, e_{o,1}, \dots, e_{o,k_o}\}) \preceq P \not\preceq B) = \\
& \quad T_t.m(T_{a,1}, \dots, T_{a,n}) \\
& \tau(t_1.m_1(a_{1,1}, \dots, a_{1,n_1}), \dots, \tau(t_n.m_n(a_{1,n}, \dots, a_{1,n_n})) = \\
& \quad \tau(\Gamma(t_1).m_1(\Gamma(a_{1,1}), \dots, \Gamma(a_{1,n_1})), \dots, \tau(\Gamma(t_n).m_n(\Gamma(a_{1,n}), \dots, \Gamma(a_{1,n_n})))
\end{aligned}$$

Figure 28: Definition of τ .

corresponding to arg_i .

$$EC_a \preceq EC_b \wedge \Gamma(\text{this}(EC_b)) = \Gamma(\text{this}(EC_a))$$

$$\begin{aligned}
& \Downarrow \\
& \Omega(EC_a, t, (arg_1, par_1) \dots (arg_n, par_n)) \preceq \Upsilon(t.m(args))
\end{aligned}$$

Proof 38 *The requirements for substitution in EC_a are satisfied because the program is valid. Since $EC_a \preceq EC_b$, they are also valid if the type of this is the same. For formal parameters, the type must be invariant.*

$$\begin{aligned}
& \Omega(EC_a, t, args) \preceq \Upsilon(t.m(args)) \\
& \quad \Downarrow \\
& \quad \Omega(EC_a, t, args) \preceq \\
& \quad \Omega(\Phi(EC_b, \top, \emptyset), t, args) \\
& \quad \Downarrow \\
& \Omega(EC_a, t, args) \preceq \Omega(EC_b, t, args)
\end{aligned}$$

Because $EC_a \preceq EC_b$, it suffices to prove that the preconditions of Theorem 12.36 are satisfied. The preconditions all follow directly from the preconditions of this lemma and the fact that $EC_b = \varepsilon(t.m(arg_1, \dots, arg_n))$.

12.15 The \preceq relation implies the ω relation

In this section, we prove that when the \preceq relation holds between two exception clauses, the left-hand side cannot signal an exception that is not allowed by the right-hand side.

First, we define the τ function, which replaces the target and arguments of an anchored exception declaration by their static type. The definition is shown in Figure 28. For the compressed form of an anchored exception declaration, it uses the type of the target and the arguments. This type is always present because of the definition of κ . For a method expression, it replaces the target and the arguments by their types.

12.15.1 Absolute Exception Declarations

Lemma 12.39

$$\begin{aligned}
& \Phi((P_a, B_a), E, \emptyset) \preceq (P_b, B_b) \\
& \quad \Downarrow \\
& \omega((P_a, B_a), E, \text{trace}) \Rightarrow \omega((P_b, B_b), E)
\end{aligned}$$

Proof 39

$$\begin{aligned}
& \Phi((P_a, B_a), E, \emptyset) \preceq (P_b, B_b) \\
& \Downarrow (\text{definition of } \Phi \text{ and } \preceq) \\
& ((P_a \sqcap E) - B_a) \sqsubseteq (P_b - B_b) \\
& \Downarrow (\text{definition of } \sqsubseteq) \\
& E \leq ((P_a \sqcap E) - B_a) \Rightarrow E \leq (P_b - B_b) \\
& \Downarrow (\text{definition of } \leq \text{ and } \sqcap) \\
& E \leq (P_a - B_a) \Rightarrow E \leq (P_b - B_b) \\
& \Downarrow (\text{definition of } \omega) \\
& \omega((P_a, B_a), E) \Rightarrow \omega((P_b, B_b), E) \\
& \Downarrow (\text{definition of } \omega) \\
& \omega((P_a, B_a), E, \text{trace}) \Rightarrow \omega((P_b, B_b), E)
\end{aligned}$$

12.15.2 Anchored Exception Declarations

Lemma 12.40

$$\begin{aligned}
& \omega(EC, E, \text{trace}_1) \wedge \text{trace}_2 \subseteq \text{trace}_1 \\
& \Downarrow \\
& \omega(EC, E, \text{trace}_2)
\end{aligned}$$

Proof 40 *If trace_2 is a superset of trace_1 , its analysis will return false while the analysis for trace_1 might give true. The reverse can never happen.*

Lemma 12.41

$$\begin{aligned}
& \omega(\Phi(\Upsilon(\text{anchor}), E, \emptyset), \text{trace}) \\
& \Updownarrow \\
& \omega(\Phi(\Upsilon(\text{anchor}), E, \emptyset), \{\tau(\text{anchor})\} \cup \text{trace})
\end{aligned}$$

Proof 41 *Because the method referenced by the anchored exception declaration is analyzed anyway in both cases, it does not matter that in the relation without a trace, the analysis is done a second time (after which it is in the trace and will not be analyzed again).*

Lemma 12.42 *If trace contains compressed forms of anchored exception declarations, then:*

$$\begin{aligned}
& \Phi(\text{anchor}_a, E, \emptyset) \preceq \text{anchor}_b \\
& \Downarrow \\
& (\omega(\text{anchor}_a, E, \tau(\text{trace})) \Rightarrow \omega(\text{anchor}_b, E))
\end{aligned}$$

Proof 42 *If $\tau(\text{anchor}_a) \in \tau(\text{trace})$, the lemma is trivially true because of the definition of ω , which then returns false. It makes the left-hand side of the bottom implication false, making the bottom implication true, which in turn makes the complete implication true. This will stop the induction loop.*

$$\begin{aligned}
& \Phi(\text{anchor}_a, E, \emptyset) \preceq \text{anchor}_b \\
& \quad \Downarrow (\text{Lemma 12.17}) \\
& \text{method}(\text{anchor}_a) <: \text{method}(\text{anchor}_b) \\
& \quad \Downarrow (\text{Rule 2}) \\
& \varepsilon(\text{anchor}_a) \preceq \varepsilon(\text{anchor}_b)
\end{aligned}$$

We will now use Theorems 12.30 and 12.36.

$$\begin{aligned}
& \Upsilon(\Phi(\text{anchor}_a, E, \emptyset)) \preceq \Upsilon(\text{anchor}_b) \\
& \quad \Updownarrow \\
& \Omega(\Phi(\varepsilon(\text{anchor}_a), (P_a \sqcap E), B_a), t_a, \text{args}_a) \preceq \\
& \quad \Omega(\Phi(\varepsilon(\text{anchor}_b), P_b, B_b), t_b, \text{args}_b)
\end{aligned}$$

The preconditions of Theorems 12.30 and 12.36 are satisfied because $\text{anchor}_a \preceq \text{anchor}_b$, and thus $\Upsilon(\Phi(\text{anchor}_a, E, \emptyset)) \preceq \Upsilon(\text{anchor}_b)$.

$$\begin{aligned}
& \Upsilon(\Phi(\text{anchor}_a, E, \emptyset)) \preceq \Upsilon(\text{anchor}_b) \\
& \quad \Downarrow (\text{Lemma 12.21}) \\
& \text{trace} \vdash \Upsilon(\Phi(\text{anchor}_a, E, \emptyset)) \preceq \Upsilon(\text{anchor}_b) \\
& \quad \Downarrow (\text{Induction on Lemma 12.43}) \\
& \omega(\Upsilon(\Phi(\text{anchor}_a, E, \emptyset)), E, \tau(\text{trace})) \Rightarrow \\
& \quad \omega(\Upsilon(\text{anchor}_b), E) \\
& \quad \Downarrow (\text{definition of } \omega) \\
& \omega(\Phi(\text{anchor}_a, E, \emptyset), E, \tau(\text{trace})) \Rightarrow \omega(\text{anchor}_b, E) \\
& \quad \Updownarrow (\text{Lemma 12.14}) \\
& \omega(\text{anchor}_a, E, \tau(\text{trace})) \Rightarrow \omega(\text{anchor}_b, E)
\end{aligned}$$

For the induction, we perform a one-level expansion. Because we put $\tau(\text{anchor}_b)$, which substitutes the target and arguments of the method expression by their types, in the trace, this induction will always end (see Theorem 12.5). The base cases are exception clauses that only contain absolute exception declarations. For such exception clauses, Theorem 12.43 is proven by Lemma 12.39.

12.15.3 Exception Clauses

Lemma 12.43

$$\text{trace} \vdash EC_a \preceq EC_b \Rightarrow (\omega(EC_a, E, \tau(\text{trace})) \Rightarrow \omega(EC_b, E))$$

Proof 43

$$\begin{aligned}
& \text{trace} \vdash EC_a \preceq EC_b \\
& \quad \Updownarrow (\text{definition of } \preceq) \\
& \left(\begin{array}{l}
(\forall (P_a, B_a) \in EC_a, \forall E, \omega((P_a, B_a), E) : \\
\quad \exists (P_b, B_b) \in EC_b : \Phi((P_a, B_a), E, \emptyset) \preceq (P_b, B_b)) \wedge \\
(\forall AED_a \in EC_a, \forall E, \omega(AED_a, E) : \exists AED_b \in EC_b : \\
\quad \Phi(AED_a, E, \emptyset) \preceq AED_b \vee \\
\quad \kappa(AED_a, EC_b) \notin \text{trace} \Rightarrow \\
\quad \{\kappa(AED_a, EC_b)\} \cup \text{trace} \vdash (\Phi(\Upsilon(AED_a), E, \emptyset) \preceq EC_b))
\end{array} \right)
\end{aligned}$$

As a result, for every E , we can find ABS_{b,x_i} and AED_{b,y_i} such that:

$$\left(\begin{array}{c} \Downarrow \text{(Lemmas 12.39 and 12.42 and 12.14)} \\ \omega(ABS_{a,1}, E, \tau(\text{trace})) \Rightarrow \omega(ABS_{b,x_1}, E) \wedge \\ \dots \wedge \\ \omega(ABS_{a,n}, E, \tau(\text{trace})) \Rightarrow \omega(ABS_{b,x_n}, E) \wedge \\ \left(\begin{array}{c} \omega(AED_{a,1}, \tau(\text{trace})) \Rightarrow \omega(AED_{b,y_1}, E) \vee \\ \kappa(AED_{a,1}, EC_b) \notin \text{trace} \Rightarrow \\ \{\kappa(AED_{a,1}, EC_b)\} \cup \text{trace} \vdash (\Phi(\Upsilon(AED_{a,1}), E, \emptyset) \preceq EC_b) \end{array} \right) \wedge \\ \dots \wedge \\ \left(\begin{array}{c} \omega(AED_{a,m}, \tau(\text{trace})) \Rightarrow \omega(AED_{b,y_m}, E) \vee \\ \kappa(AED_{a,m}, EC_b) \notin \text{trace} \Rightarrow \\ \{\kappa(AED_{a,m}, EC_b)\} \cup \text{trace} \vdash (\Phi(\Upsilon(AED_{a,m}), E, \emptyset) \preceq EC_b) \end{array} \right) \end{array} \right)$$

If $\kappa(AED_{a,i}, EC_b) \in \text{trace}$, the proof ends for that anchored exception declaration. If this is the case, $\tau(AED_{a,i}) \in \tau(\text{trace})$, which means $\omega(AED_{a,i}, E, \tau(\text{trace})) = \text{false}$. If that is not the case, we use induction. As such, we assume $\kappa(AED_{a,i}, EC_b) \notin \text{trace}$ for the remainder of the proof.

$$\left(\begin{array}{c} \Downarrow \text{(Lemma 12.14 and induction on Theorem 12.44)} \\ \omega(ABS_{a,1}, E, \tau(\text{trace})) \Rightarrow \omega(ABS_{b,x_1}, E) \wedge \\ \dots \wedge \\ \omega(ABS_{a,n}, E, \tau(\text{trace})) \Rightarrow \omega(ABS_{b,x_n}, E) \wedge \\ \left(\begin{array}{c} \omega(AED_{a,1}, E, \tau(\text{trace})) \Rightarrow \omega(AED_{b,y_1}, E) \vee \\ \omega(\Upsilon(AED_{a,1}), E, \{\tau(AED_{a,1})\} \cup \tau(\text{trace})) \Rightarrow \\ \omega(EC_b, E) \end{array} \right) \wedge \\ \dots \wedge \\ \left(\begin{array}{c} \omega(AED_{a,m}, E, \tau(\text{trace})) \Rightarrow \omega(AED_{b,y_m}, E) \vee \\ \omega(\Upsilon(AED_{a,m}), E, \{\tau(AED_{a,m})\} \cup \tau(\text{trace})) \Rightarrow \\ \omega(EC_b, E) \end{array} \right) \end{array} \right)$$

$$\left(\begin{array}{c} \Downarrow \text{(Definition of } \omega \text{ and Lemma 12.41)} \\ \omega(ABS_{a,1}, E, \tau(\text{trace})) \Rightarrow \omega(ABS_{b,x_1}, E) \wedge \\ \dots \wedge \\ \omega(ABS_{a,n}, E, \tau(\text{trace})) \Rightarrow \omega(ABS_{b,x_n}, E) \wedge \\ \left(\begin{array}{c} \omega(AED_{a,1}, E, \tau(\text{trace})) \Rightarrow \omega(AED_{b,y_1}, E) \vee \\ \omega(AED_{a,1}, E, \tau(\text{trace})) \Rightarrow \omega(EC_b, E) \end{array} \right) \wedge \\ \dots \wedge \\ \left(\begin{array}{c} \omega(AED_{a,m}, E, \tau(\text{trace})) \Rightarrow \omega(AED_{b,y_m}, E) \vee \\ \omega(AED_{a,m}, E, \tau(\text{trace})) \Rightarrow \omega(EC_b, E) \end{array} \right) \\ \Downarrow \text{(definition of } \omega) \\ \omega(EC_a, E) \Rightarrow \omega(EC_b, E, \tau(\text{trace})) \end{array} \right)$$

Theorem 12.44

$$EC_a \preceq EC_b \Rightarrow (\omega(EC_a, E) \Rightarrow \omega(EC_b, E))$$

Proof 44 *The proof follows directly from Lemma 12.44*

12.16 Expansion Does Not Allow More Than the Exception Clause

In this section, we prove that the exception clause resulting from the expansion of a method invocation does not allow more exception to be signalled than the exception clause of the invoked method. This property is important from a methodological point of view. If it were allowed, a method invocation could be allowed to signal a checked exception that could not have been foreseen by looking only to the exception clause of the method. This is very confusing for a programmer. For example, the expansion function could simply return `throws Throwable`. This would not compromise compile-time safety, but it would make anchored exception declarations useless.

Lemma 12.45

$$\omega(\Phi((P, B), P_n, B_n), E) \Rightarrow \omega((P, B), E)$$

Proof 45

$$\begin{aligned} & \omega(\Phi((P, B), P_n, B_n), E) \Rightarrow \\ & \omega((P \sqcap P_n, B \sqcup B_n), E) \Rightarrow \\ & E \trianglelefteq ((P \sqcap P_n) - (B \sqcup B_n)) \Rightarrow \\ & E \trianglelefteq P \wedge E \trianglelefteq P_n \wedge E \not\trianglelefteq B \wedge E \not\trianglelefteq B_n \Rightarrow \\ & E \trianglelefteq P \wedge E \not\trianglelefteq B \Rightarrow \\ & \omega((P, B), E) \end{aligned}$$

Lemma 12.46

$$\omega(\Phi(\text{anchor}, P_n, B_n), E) \Rightarrow \omega(\text{anchor}, E)$$

Proof 46 *Let $\text{anchor} = \text{like } t.m(\text{args}) \trianglelefteq P \not\trianglelefteq B$.*

$$\begin{aligned} & \omega(\Phi(\text{anchor}, P_n, B_n), E) \Rightarrow \omega(\text{anchor}, E) \\ & \quad \Updownarrow \\ & \omega(\Upsilon(\Phi(\text{anchor}, P_n, B_n)), E) \Rightarrow \omega(\Upsilon(\text{anchor}), E) \\ & \quad \Updownarrow (\Phi \text{ does not affect the method expression}) \\ & \omega(\Omega(\Phi(\varepsilon(\text{anchor}), P \sqcap P_n, B \sqcup B_n), t, \text{args}), E) \Rightarrow \\ & \quad \omega(\Omega(\Phi(\varepsilon(\text{anchor}), P, B), t, \text{args}), E) \\ & \quad \Updownarrow (\text{Lemma 12.11 and definition of } \Phi) \\ & \omega(\Phi(\Omega(\Phi(\varepsilon(\text{anchor}), P, B), t, \text{args}), P_n, B_n), E) \Rightarrow \\ & \quad \omega(\Phi(\Omega(\Phi(\varepsilon(\text{anchor}), P, B), t, \text{args}), *, \emptyset), E) \end{aligned}$$

We now use Theorems 12.36 and 12.30.

- *The first three preconditions of 12.36 are satisfied because the corresponding elements in the equation above are identical. The last preconditions follow from the fact that the program must be valid and the fact that Φ does not affect the method expression of anchor and thus does not affect the selected method either.*

- The preconditions of Theorem 12.30 are satisfied because $(P_n - B_n) \preceq (\ast - \emptyset)$.

As a result, we know that:

$$\begin{aligned} & \Phi(\Omega(\Phi(\varepsilon(\text{anchor}), P, B), t, \text{args}), P_n, B_n) \preceq \\ & \Phi(\Omega(\Phi(\varepsilon(\text{anchor}), P, B), t, \text{args}), \ast, \emptyset) \end{aligned}$$

Applying Theorem 12.44 completes the proof.

Lemma 12.47

$$\omega(\Phi(EC, P, B), E) \Rightarrow \omega(EC, E)$$

Proof 47

$$\begin{aligned} & \omega(\Phi(EC, P, B), E) \\ & \quad \Downarrow \\ & \omega(\Phi(\{ABS_1, \dots, ABS_n, AED_1, \dots, AED_m\}, P, B), E) \\ & \quad \Downarrow \text{(definition of } \Omega) \\ & \omega(\{\Phi(ABS_1, P, B), \dots, \Phi(ABS_n, P, B) \\ & \quad \Phi(AED_1, \text{pre}, \text{args}), \dots, \Phi(AED_m, \text{pre}, \text{args})\}, E) \\ & \quad \Downarrow \text{(definition of } \omega) \\ & \omega(\Phi(ABS_1, P, B), E) \vee \dots \vee \omega(\Phi(ABS_n, P, B), E) \vee \\ & \omega(\Phi(AED_1, P, B), E) \vee \dots \vee \omega(\Phi(AED_m, P, B), E) \\ & \quad \Downarrow \text{(Lemmas 12.45 and 12.46)} \\ & \omega(ABS_1, E) \vee \dots \vee \omega(ABS_n, E) \vee \\ & \omega(AED_1, E) \vee \dots \vee \omega(AED_m, E) \\ & \quad \Downarrow \\ & \omega(EC, E) \end{aligned}$$

Lemma 12.48

$$\begin{aligned} & \text{ok}_\Omega(\text{args}, (\text{pre}, \text{this}(AED))) \\ & \quad \Downarrow \\ & \omega(\Omega(AED, \text{pre}, \text{args}), E) \Rightarrow \omega(AED, E) \end{aligned}$$

Proof 48 Let $AED = \text{like } t.m(a_1, \dots, a_n) \preceq P \not\preceq B$.

$$\begin{aligned} & \omega(\Omega(AED, \text{pre}, \text{args}), E) \Rightarrow \omega(AED, E) \\ & \quad \Downarrow \text{(definition of } \omega) \\ & \omega(\Upsilon(\Omega(AED, \text{pre}, \text{args})), E) \Rightarrow \omega(\Upsilon(AED), E) \\ & \quad \Downarrow \text{(definition of } \Upsilon) \\ & \omega(\Omega(\Phi(\varepsilon(\Omega(AED, \text{pre}, \text{args})), P, B), \Omega(t, \text{pre}, \text{args}), \\ & \quad \Omega(a_1, \text{pre}, \text{args}) \dots \Omega(a_n, \text{pre}, \text{args})), E) \Rightarrow \\ & \omega(\Omega(\Phi(\varepsilon(AED), P, B), t, \text{args}), E) \end{aligned}$$

Because $ok_{\Omega}(args, (pre, env(AED)))$, we know from Lemmas 12.16 and 12.32 that:

$$\begin{aligned} & \Gamma(\Omega(t, pre, args)) <: \Gamma(t) \wedge \\ & \Gamma(\Omega(a_1, pre, args)) <: \Gamma(a_1) \wedge \dots \wedge \Gamma(\Omega(a_n, pre, args)) <: \Gamma(a_n) \end{aligned}$$

As a result, we know that the method selected by $\Omega(AED, pre, args)$ will override or be equal to the method selected by AED . This means that rule 2 applies.

$$\varepsilon(\Omega(AED, pre, args)) \preceq \varepsilon(AED)$$

We now apply Theorems 12.30 and 12.36 to the arguments of ω in the implication above.

- The preconditions of Theorem 12.30 are satisfied because the arguments of Φ are identical.
- The first precondition of Theorem 12.36 follows from the application of Theorem 12.30. The second and third preconditions follow from Lemma 12.32. The last preconditions follow from the preconditions of this lemma, from Lemmas 12.16 and 12.32, from the fact that the types of the target of a method invocation must be conform to the type of this in the invoked method, from the fact that the type of the actual arguments must be conform to that of the invoked method, and from the requirement that types of formal parameters must be invariant.

As a result, we know that:

$$\begin{aligned} & \Omega(\Phi(\varepsilon(\Omega(AED, pre, args)), P, B), \Omega(t, pre, args), \\ & \quad \Omega(a_1, pre, args) \dots \Omega(a_n, pre, args)) \preceq \\ & \quad \Omega(\Phi(\varepsilon(AED), P, B), t, args) \end{aligned}$$

Applying Theorem 12.44 completes the proof.

Lemma 12.49

$$\begin{aligned} & ok_{\Omega}(args, (pre, env(EC))) \\ & \quad \downarrow \\ & \omega(\Omega(EC, pre, args), E) \Rightarrow \omega(EC, E) \end{aligned}$$

Proof 49 The proof of this lemma is nearly identical to that of Lemma 12.47.

Theorem 12.50

$$\omega(\Upsilon(AED), E) \Rightarrow \omega(\varepsilon(AED), E)$$

Proof 50 This theorem follows directly from Lemmas 12.49 and 12.47.

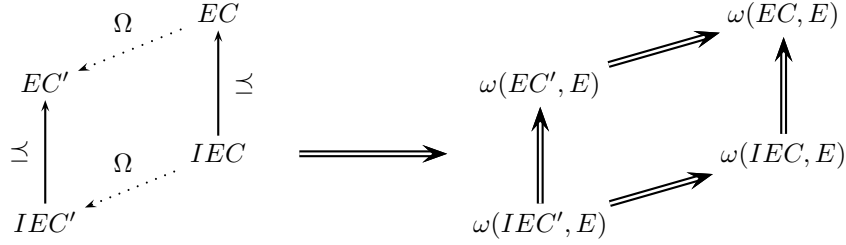


Figure 29: Schema for compile-time safety proof.

12.17 Compile-time safety

Now we can finally prove that anchored exception declarations are compile-time safe. For compile-time safety to be violated, there must be at least one method of which the implementation can signal a checked exception under a circumstance that could not have been predicted by the client when inspecting the exception clause of that method. We now show that this is not possible for a program satisfying all rules.

Theorem 12.51 (Safety) *Let $t.m(arg_1, \dots, arg_n)$ be a method invocation in a valid program, $EC = \varepsilon(t.m(arg_1, \dots, arg_n))$, $IEC = IEC(t.m(arg_1, \dots, arg_n))$, and let par_i be the formal parameter corresponding to arg_i , then:*

$$\omega(\Omega(IEC, t, arg_1, \dots, arg_n), E) \Rightarrow \omega(\Omega(EC, t, arg_1, \dots, arg_n), E)$$

Proof 51 *Figure 29 illustrates the proof. The exception clause of the method is represented by EC , its implementation exception clause by IEC . We know from rule 3 that $IEC \preceq EC$, so Theorem 12.38 ensures that after insertion of the context information of any call-site, resulting in EC' and IEC' , $EC' \preceq IEC'$ holds. Note that at run-time, the available context information is even more specific, but because the same information is inserted in both exception clauses, the relation between IEC' and EC' will still hold. Both relations are shown in the left diagram.*

Using Theorem 12.44 and Lemma 12.49, we can transform the left diagram into the right diagram. Theorem 12.44 ensures that $\omega(IEC, E) \Rightarrow \omega(EC, E)$ and $\omega(IEC', E) \Rightarrow \omega(EC', E)$. Lemma 12.49 ensures that $\omega(EC', E) \Rightarrow \omega(EC, E)$ and $\omega(IEC', E) \Rightarrow \omega(IEC, E)$. Both relations are shown in the right diagram.

From these relations, we can conclude that no method invocation can result in a checked exception that was not declared by the exception clause of the invoked method.

13 Related Work

There has been a lot of work on the analysis of the exception flow in programs. We first describe the different approaches, and then compare them to our work.

Every approach uses a different mechanism to track the dependencies between methods or functions, and automatically infers the exception clause of every method.

Many exception flow analyses augment the type system of the underlying programming language with information about the exceptions that can be signalled by a particular language construct [5, 6, 7, 8, 9, 10, 11, 12]. The analyses differ in their representation of the inferred exception clauses, and the speed and precision of the analysis, but the principle is always the same. To track the dependencies between methods or functions, they insert type variables of the invoked method or function into the exceptional type of the invoking method or function. This is illustrated by the `map` function that is discussed in Section 3. Glynn et al. augment the type system with boolean constraints in order to track these dependencies [13].

Another approach exception flow analysis based on control flow graphs [14, 15, 16]. The normal and exceptional program flow of each method is encoded in a separate graph. The interprocedural exception flow is then analysed by creating edges between the graphs of the involved methods. The analysis connects exceptional exit nodes of the invoked method to nodes representing exception handlers or exit nodes of the invoking method. These connections represent the dependencies between methods.

Robillard and Murphy [24] developed a language-independent model for analyzing the exception flow in object-oriented programs, along with a tool specifically for Java. Their analysis is similar to that of Schaefer and Bundy [25]. They use functions instead of type variables to incorporate the exceptional behavior of other methods. They also discuss the cost of modifying the exception clause of a method, and the use of unchecked exceptions as a result. In [26], they show that the difficulty in determining all exceptional conditions in advance gives rise to the need for evolution of the exceptional behavior of a method.

The expressiveness of the inferred exception clauses of the above approaches is equal to that of ours – disregarding minor details. In the actual analysis, however, there will be a significant difference. The automatically inferred exception clause of the other approaches will be as tight as possible. In our approach, the exception clause is an explicitly written upper bound for the exceptional behavior of a method. This exception clause is often less precise than an inferred exception clause, but it allows the type checker to ensure that a newly added method cannot throw exceptions that are not allowed by the overridden methods. As a result, the exception safety guaranteed by our approach is modular. For exception flow analyses, statements about exception safety can be violated by adding code.

A safety analysis based on automatically inferred exception clauses, however, remains useful for object-oriented programming languages. It can be used to obtain a more precise analysis of the exception flow of a particular program, and thus exclude some exceptions that our approach cannot exclude due to exception clauses that are too loose for that particular program.

Mikhailova and Romanovsky [3] provide support for evolution of the exceptional behavior of a method by introducing a *rescue clause*. A rescue clause is

a default exception handler that allows a method to have an exception clause that does not conform to the methods it overrides. If a client of that method provides a handler for the new exception, that handler is used, otherwise the rescue clause handles the exception. This mechanism only provides a solution when a useful default handler can be provided, which usually is not the case. Anchored exception declarations are complementary to rescue clauses. Rescue clauses allow a method to signal exceptions that would normally not be allowed, if a default handler is provided. Anchored exception declarations offer similar flexibility, but they cannot be used after the facts; they must be written in advance.

Cacho et al. [27] propose an extension of AspectJ, called EJFlow, which provides abstractions to describe global views of the exception control flow of a program. An *explicit exception channel* specifies the exception that can be signalled, the methods that raise the exceptions, methods that handle the exceptions, methods that propagate the exceptions, and the exceptions that can exit the channel. The latter set is called the *exception interface*. *Pluggable handlers* are separate pieces of code that can be attached to an exception handling context (i.e. a try-block) to handle the exceptions. This allows the extraction of exception handling code that would otherwise be duplicated. They add constructs to AspectJ to define pointcuts based on the exception control flow of other language constructs.

14 Conclusion

The original type checker for anchored exception declarations was not modular because it could not handle loops in the anchor graph. This limitation is problematic for the development of large-scale software systems.

In this paper, we improved the type checker by letting it determine which information in an anchored exception declaration can still influence the outcome of the analysis. By keeping a trace of compressed anchored exception declarations, which contain only the relevant information, the type checker can determine when the analysis can be stopped. For recursive expansion, compression is done by replacing the target and arguments by their types. For conformance verification of two exception clauses, compression is done by replacing all subexpression that cannot influence a syntactical match by their static types. These modifications make the exception safety analysis complete and decidable for language without parametric polymorphism.

We proved that both complete exception safety analysis and complete exception flow analysis based on static type information are undecidable in a language with subtyping and parametric polymorphism.

References

- [1] J. B. Goodenough, “Exception handling: issues and a proposed notation,” *Commun. ACM*, vol. 18, no. 12, pp. 683–696, 1975.
- [2] R. Miller and A. Tripathi, “Issues with exception handling in object-oriented systems,” in *Proceedings of the European Conference on Object-Oriented Programming (ECOOP ’97)*, ser. LNCS, vol. 1241, Jun. 1997, p. 85.
- [3] A. Mikhailova and A. Romanovsky, “Supporting evolution of interface exceptions,” in *Advances in exception handling techniques*, 2001, pp. 94–110.
- [4] M. van Dooren and E. Steegmans, “Combining the robustness of checked exceptions with the flexibility of unchecked exceptions using anchored exception declarations,” in *OOPSLA*, 2005, pp. 455–471.
- [5] B.-M. Chang, J. Jo, K. Yi, and K. Choe, “Interprocedural exception analysis for Java,” in *Proceedings of the 16th ACM Symposium on Applied Computing*, March 2001.
- [6] M. Fähndrich and A. Aiken, “Program analysis using mixed term and set constraints,” in *SAS ’97: Proceedings of the 4th International Symposium on Static Analysis*, pp. 114–126.
- [7] —, “Refined type inference for ML,” in *Proceedings of the 1st Workshop on Types in Compilation*, June 1997.
- [8] J. Guzmán and A. Suárez, “An Extended Type System for Exceptions,” in *Record of the fifth ACM SIGPLAN workshop on ML and its Applications*, June 1994.
- [9] J. Jo, B.-M. Chang, K. Yi, and K.-M. Choe, “An uncaught exception analysis for Java,” *Journal of Systems and Software*, vol. 72, no. 1, pp. 59–69, 2004.
- [10] F. Pessaux and X. Leroy, “Type-based analysis of uncaught exceptions,” in *POPL*, 1999, pp. 276–290.
- [11] K. Yi, “Compile-time detection of uncaught exceptions in standard ML programs,” in *The 1st International Static Analysis Symposium*, ser. Lecture Notes in Computer Science, vol. 864, September 1994, pp. 238–254.
- [12] K. Yi and S. Ryu, “Towards a cost-effective estimation of uncaught exceptions in SML programs,” in *Proceedings of the Annual International Static Analysis Symposium*, ser. Lecture Notes in Computer Science, vol. 1302, September 1997, pp. 98–113.
- [13] K. Glynn, P. J. Stuckey, M. Sulzmann, and H. Søndergaard, “Exception analysis for non-strict languages,” in *ICFP*, 2002, pp. 98–109.

- [14] S. Sinha and M. J. Harrold, “Analysis of programs with exception-handling constructs,” in *ICSM '98: Proceedings of the International Conference on Software Maintenance*, 1998, p. 348.
- [15] M. Allen and S. Horwitz, “Slicing Java programs that throw and catch exceptions,” in *PEPM '03: Proceedings of the 2003 ACM SIGPLAN workshop on Partial evaluation and semantics-based program manipulation*, 2003, pp. 44–54.
- [16] J.-D. Choi, D. Grove, M. Hind, and V. Sarkar, “Efficient and precise modeling of exceptions for the analysis of Java programs,” in *PASTE '99: Proceedings of the 1999 ACM SIGPLAN-SIGSOFT workshop on Program analysis for software tools and engineering*, 1999, pp. 21–31.
- [17] E. Gamma, R. Helm, R. Johnson, and J. Vlissides, *Design patterns: elements of reusable object-oriented software*. Addison-Wesley Longman Publishing Co., Inc., 1995.
- [18] G. T. Leavens, A. L. Baker, and C. Ruby, “Preliminary design of JML: A behavioral interface specification language for Java,” Tech. Rep. 98-06i, 2000.
- [19] M. Flatt, S. Krishnamurthi, and M. Felleisen, “Classes and mixins,” in *POPL*, 1998, pp. 171–183.
- [20] B. Meyer, *Object-oriented software construction (2nd ed.)*. Prentice-Hall, Inc., 1997.
- [21] ECMA Technical Committee 39 (TC39) Task Group 2 (TG2), *C# Language Specification*, 2nd ed., ECMA, December 2002.
- [22] M. van Dooren and W. Joosen, “Modular anchored exception declarations,” Department of Computer Science, K.U.Leuven, Leuven, Belgium, Report CW 544, 2009. [Online]. Available: <http://www.cs.kuleuven.be/~marko/tr-anchors.pdf>
- [23] M. Minsky, *Computation : finite and infinite machines / Marvin L. Minsky*. Prentice-Hall, [Hemel Hempstead], 1972.
- [24] M. P. Robillard and G. C. Murphy, “Static analysis to support the evolution of exception structure in object-oriented systems,” *ACM Trans. Softw. Eng. Methodol.*, vol. 12, no. 2, pp. 191–221, 2003.
- [25] C. F. Schaefer and G. N. Bundy, “Static analysis of exception handling in Ada.” *Softw., Pract. Exper.*, vol. 23, no. 10, pp. 1157–1174, 1993.
- [26] M. P. Robillard and G. C. Murphy, “Designing robust Java programs with exceptions,” in *Proceedings of the 8th ACM SIGSOFT international symposium on Foundations of software engineering*, 2000, pp. 2–10.

- [27] N. Cacho, F. C. Filho, A. Garcia, and E. Figueiredo, “Ejflow: taming exceptional control flows in aspect-oriented programming,” in *AOSD*, 2008, pp. 72–83.
- [28] M. Bruntink, A. van Deursen, and T. Tourwé, “Discovering faults in idiom-based exception handling,” in *ICSE*, 2006, pp. 242–251.
- [29] D. Teller, A. Spiwack, and T. Varoquaux, “Catch me if you can: Towards type-safe, hierarchical, lightweight, polymorphic and efficient error management in ocaml,” in *2008 ACM Sigplan workshop on ML*, 2008.
- [30] F. C. Filho, N. Cacho, E. Figueiredo, a. Raquel Maranh A. Garcia, and C. M. F. Rubira, “Exceptions and aspects: the devil is in the details,” in *FSE*, 2006, pp. 152–162.
- [31] B. Jacobs, P. Müller, and F. Piessens, “Sound reasoning about unchecked exceptions,” in *SEFM 2007*, 2007, pp. 113–122.
- [32] B. C. Pierce, *Types and programming languages*. MIT Press, 2002.
- [33] A. F. Garcia, C. M. F. Rubira, A. Romanovsky, and J. Xu, “A comparative study of exception handling mechanisms for building dependable object-oriented software,” *The Journal of Systems and Software*, vol. 59, no. 2, pp. 197–222, 2001.
- [34] M. Lippert and C. Lopes, “A study on exception detection and handling using aspect-oriented programming,” Xerox PARC, Tech. Rep., 1999.
- [35] M. P. Robillard and G. C. Murphy, “Analyzing exception flow in Java programs,” in *Software Engineering – ESEC/FSE’99*, ser. Lecture Notes in Computer Science, vol. 1687, September 1999, pp. 322–337.
- [36] P. A. Buhr and W. Y. R. Mok, “Advanced exception handling mechanisms,” *IEEE Trans. Softw. Eng.*, vol. 26, no. 9, pp. 820–836, 2000.
- [37] C. Dony, “Exception handling and object-oriented programming: towards a synthesis,” in *ECOOP*, 1990, pp. 322–330.
- [38] S. Yemini and D. M. Berry, “A modular verifiable exception handling mechanism,” *ACM Trans. Program. Lang. Syst.*, vol. 7, no. 2, pp. 214–243, 1985.
- [39] B. G. Ryder, D. Smith, U. Kremer, M. Gordon, and N. Shah, “A static study of Java exceptions using JESP,” in *Computational Complexity*, 2000, pp. 67–81.
- [40] S. Ryu and K. Yi, “Exception analysis for multithreaded Java programs,” in *APAQS*, p. 23.
- [41] A. Romanovsky and B. Sandén, “Except for exception handling . . .,” *Ada Lett.*, vol. XXI, no. 3, pp. 19–25, 2001.

- [42] B. Jacobs, “A formalisation of Java’s exception mechanism,” in *ESOP*, 2001, pp. 284–301.
- [43] S. Yemini and D. M. Berry, “An axiomatic treatment of exception handling in an expression-oriented language,” *ACM Trans. Program. Lang. Syst.*, vol. 9, no. 3, pp. 390–407, 1987.
- [44] K. R. M. Leino and W. Schulte, “Exception safety for C#,” in *Proceedings, Software Engineering and Formal Methods (SEFM), Beijing, China*, J. Cuellar and Z. Liu, Eds., 2004.
- [45] R. Chatterjee, B. G. Ryder, and W. A. Landi, “Complexity of points-to analysis of Java in the presence of exceptions,” *IEEE Trans. Softw. Eng.*, vol. 27, no. 6, pp. 481–512, 2001.
- [46] K. Yi, “An abstract interpretation for estimating uncaught exceptions in standard ML programs,” *Science of Computer Programming*, vol. 31, no. 1, pp. 147–173, 1998.
- [47] S. P. Jones, A. Reid, F. Henderson, T. Hoare, and S. Marlow, “A semantics for imprecise exceptions,” in *PLDI 1999*, 1999, pp. 25–36.
- [48] J. Gosling *et al.*, *The Java Language Specification, Second Edition*. Addison-Wesley Longman Publishing Co., Inc., 2000.
- [49] A. Hejlsberg, “The trouble with checked exceptions.”