Combining the Robustness of Checked Exceptions with the Flexibility of Unchecked Exceptions using Anchored Exception Declarations

Marko van Dooren *Eric Steegmans*

Report CW 407, March 2005

Katholieke Universiteit Leuven
Department of Computer Science
Celestijnenlaan 200A – B-3001 Heverlee (Belgium)

Combining the Robustness of Checked Exceptions with the Flexibility of Unchecked Exceptions using Anchored Exception Declarations

Marko van Dooren *Eric Steegmans*

Report CW 407, March 2005

Department of Computer Science, K.U.Leuven

Abstract

Ever since their invention 30 years ago, checked exceptions have been a point of much discussion. On the one hand, they increase the robustness of soft ware by preventing the manifestation of unanticipated checked exceptions at run-time. On the other hand, they decrease the adaptability of software bec ause they must be propagated explicitly, and must often be handled even if they cannot be signalled.

We show that these problems are caused by a conflict between the exceptional interface of a method and the principle of abstraction. We then solve this conflict by introducing anchored exception declarations, which allow the exceptional behaviour of a method to be declared relative to that of others. We present their formal semantics, along with the necessary rules for ensuring compile-time safety, and give a proof of correctness. We show that ancho red exception declarations do not violate the principle of information hiding when used properly, and provide a guideline for when to use them.

We have implemented anchored exception declarations as an extension to the ClassicJava programming language, called Cappuccino.

Keywords : Exception handling, anchoring
CR Subject Classification : D.2.5, D.3.1

Combining the Robustness of Checked Exceptions with the Flexibility of Unchecked Exceptions using Anchored Exception Declarations

Marko van Dooren and Eric Steegmans
Katholieke Universiteit Leuven

1. INTRODUCTION

The common way of dealing with exceptional conditions in object-oriented software is the use of an exception handling mechanism. When an exceptional condition is detected by a component, the latter raises an exception and signals it to the caller. The caller can then handle the exception in a context dependent manner. This way, the reusability of a component is improved by removing the specific logic for handling the abnormal condition from that component. Additionally, exception handling mechanisms force a separation of normal code and exception handling code, resulting in programs that are easier to understand.

Exceptions can be divided into two categories: checked exceptions and unchecked exceptions. Checked exceptions must be propagated explicitly by listing them in the method header, while unchecked exceptions are propagated implicitly.

Checked exceptions improve the robustness of software [10; 25; 24]. Because every checked exception that can be signalled during the execution of 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. The programmer is forced to take a decision for every checked exception, so he can be reasonably sure that all checked exceptions are handled properly.

As the software evolves, checked exceptions may need to be added to or removed from existing methods [24; 25]. The exception handling mechanism will reject all methods that can encounter newly added checked exceptions, but do not deal with them. Outdated exception handlers for checked exceptions that cannot be signalled anymore will also be rejected, keeping the source code clean. Consequently, all affected methods must be modified manually by the programmer.

Unfortunately, checked exceptions also complicate the adaptability of software. Because a checked exception must be explicitly listed in the method header when it can be signalled, it leaves marks along every chain of method invocations that propagates the exception. This becomes problematic when the software evolves, since a new checked exception introduced at the end of a chain will trigger changes in every link of the chain until it is handled. The exception handling mechanism does not only force a programmer to handle the exception at the appropriate places, but also demands the modification of many methods that only propagate the exception. Consequently, dummy exception handlers are added to filter out checked exceptions that cannot be signalled. These handlers not only clutter the code, but they also discard exceptions if the filtered exception can actually be signalled after the software has evolved.

Another problem with checked exceptions is the lack of context information available to the exception handling mechanism. Often, a programmer knows that a checked exception

cannot be signalled by a method invocation, but the exception handling mechanism does not.

In this paper, we track down both problems to a conflict between the exceptional interface of a method and the principle of abstraction. We then solve the conflict by introducing anchored exception declarations to provide a relative means to specify the exceptional behaviour of a method besides traditional absolute declarations. Anchored exception declarations will be presented as an extension to ClassicJava [7]. The mechanism itself, however, is not specific to the Java programming language, nor to any particular exception handling mechanism.

Overview

In section 3, we present the root of the problems with checked exceptions. We introduce anchored exception declarations in section 4, along with the rules that are necessary for compile-time safety. In section 6.1, we show that anchored exception declarations do not violate the principle of information hiding when used properly. In section 6.2, we define which modifications of source code make sense, and which do not. We present the implementation of anchored exception declarations in section 5, followed by a case study in section 9. We discuss related work and future work in sections 10 and 11, and we conclude in section 12.

2. EXCEPTION HANDLING IN CURRENT LANGUAGES

In their extensive study of exception handling mechanisms[9], Garcia, Rubira, Romanovsky, and Xu differentiate four ways of combining the method header with the exception handling mechanism:

—The first option is not to let the exception handling mechanism interfere with the method header; there is no support for enumerating the signalled exceptions. Languages taking this approach are: Ada 95, Smalltalk, Python, Eiffel, Delphi, BETA, and C#. In this approach, all exceptions are unchecked exceptions.

—A second option is to support the enumeration of exception types in the method header, but to make it optional. Lore, C++, and Arche fall into the category of languages with optional support for mentioning exception types. Again, all exceptions are unchecked exceptions.

—The third option, used in Modula-3, Guide, and Extended Ada, not only allows the programmer to put exception types in the method headers, but forces the programmer to mention every exception that can be signalled. Here, all exceptions are checked exceptions.

—A fourth approach, introduced in Java, is a hybrid form of the second and the third option. In Java, any exception that can be thrown by a method, and is not an instance of `RuntimeException` or `Error` must be mentioned in the header. All exceptions that are instances of `RuntimeException` and `Error` are unchecked exceptions, the others are checked exceptions.

3. A CONFLICT WITH ABSTRACTION

The root of the problems with checked exceptions is a conflict with the principle of abstraction. Abstraction is an essential concept of software development, regardless of the

programming paradigm. It is the process of decomposing a large problem into several smaller and easier problems [20]. Programming logic for solving the large problem is written in terms of programming logic solving smaller problems. If the latter is modified, the former will automatically reflect the changes.

In object-oriented programming, the flexibility of abstraction is increased through delegation and dynamic binding. The *delegator* delegates a task to an object, the *delegatee*, which can be replaced to obtain another behaviour. This allows many algorithms to be composed from only a few parts. Most design patterns are based on this principle [8].

Alexander Romanovsky and Bo Sandén [31] argue that "exception handling mechanisms should correspond to the features the language provides". But this is clearly not the case for the exception clause of a method. The relative nature of abstraction conflicts with the absolute nature of the exception clause. A method can delegate the detection and raising of exceptions to other methods, but not the specification of the exceptional behaviour.

In order to specify the normal behaviour of the delegator, its postconditions contain one or more expressions referencing the delegatee, just like its implementation[1]. These expressions can be used to obtain the exact contract of a composed algorithm by filling in the postconditions of the specific delegatee in the postconditions of the delegator. This is possible because of the relative nature of the abstractions used in the implementation and the specification. Without this property, a programmer cannot be sure that the specific behaviour of the delegatee will be reflected by the combined algorithm, making it useless.

Figure 2 illustrates this for class of predicates using the Strategy pattern[8]. The specifications are written in JML [17]. The `eval` method declares that it can signal any exception at any time because it is a general purpose method. Specific predicates can narrow the set of signalled exceptions. The `forAll` method calculates whether or not all objects of a given collection satisfy a given predicate; its exact location is not important for this example. The specification and the implementation of `forAll` both contain a reference to the `eval` method being applied to the given predicate, ensuring that the result will always be consistent with the given predicate. The exception clause of `forAll`, however, declares that it can signal any exception at any time, even though it will only signal an exception when the `eval` method of the given predicate does so.

Figure 1a illustrates the situation. In both figures, the *delegator* method invokes the *delegatee* method. The thin arrows represent invocations of the delegatee. The thick arrows represent the propagation of the postconditions and behaviour, which happens in the opposite direction. Figure 1b illustrates the same for the exceptional behaviour. The delegator signals `E1` directly and propagates `E2`, which is signalled by the delegatee. It is clear that the absolute nature of the exception clause hides the origin of exception `E2`; a programmer cannot be sure that `E2` is only signalled by the delegatee.

We will now discuss the problems caused by the absolute nature of exception clauses.

3.1 Reduced Adaptability

The best-known drawback of checked exceptions is their impact on the adaptability of software [10; 24; 5]. Adding new exceptions to and removing exceptions from a program is a natural consequence of software evolution, either caused by the addition of functionality [24; 5], or by the difficulties in predicting all exceptional conditions in advance [29]. Such

[1] For specifications, this is currently only done for queries/inspectors. Research is being done to allow this for commands/mutators as well, like in the Z specification language [27].

(a) Behaviour and normal specification

(b) Exceptional specification

1: The conflict between exception handling and the principle of abstraction.

changes, however, cause a rippling effect along every call chain involving the modified method. Every method in such a call chain that propagates the new exception must also be modified.

In his paper introducing exception handling [10], Goodenough already mentions this effect for adding new checked exceptions, and argues that it is "not entirely wasted effort". Indeed, it will reveal all methods requiring modification for dealing with the new exception, thus increasing robustness as argued in the previous section. But while not entirely wasted, the effort mostly is.

Usually, methods that do not handle exceptions, but only propagate them, will also propagate the newly added exception, as illustrated by our case study in section 9. Such methods provide a certain functionality, but are unable to handle exceptions. So their behaviour has not really changed by adding a new checked exception to their exception clause; they still do the same work and report all failures. While changes in the implementation and the postconditions of a method are propagated automatically, changes in the exception clause of a method must be propagated manually. The former changes take advantage of the relative nature of abstraction, while the latter change is obstructed by the absolute nature of current exception clauses. In section 6.2, we will discuss which code modifications make

```
public abstract class Predicate {
   /*@
     @ public behavior
     @ post true;
     @ signals (Exception) ! isValidElement(object);
     @*/
   public abstract boolean eval(Object subject) throws Exception;
}
...
/*@
  @ public behavior
  @ pre collection != null;
  @ pre predicate != null;
  @ post \result == (\forall Object o;
  @                      collection.contains(o);
  @                      predicate.eval(o) == true);
  @ signals (Exception) (collection != null) &&
  @                          (\exists Object o; collection.contains(o);
  @                           !predicate.isValidElement(o));
  @*/
public boolean forAll(Collection collection, Predicate predicate) throws Exception {
  ...
  result = result && predicate.eval(element);
  ...
}
```

2: Propagation of exceptions in a strategy pattern.

sense and which do not.

The same problem applies to removing an exception from an exception clause. Although not enforced by the compiler, a programmer will want to remove the exception from the exception clauses of all methods that cannot signal it anymore because this avoids the need to write dummy exception handlers when invoking these methods. The compiler will then reveal all unnecessary exception handlers. The detection and modification of outdated exception clauses, however, must be done manually in this scenario.

As a result, programmers often switch to unchecked exceptions [30], leaving room for unanticipated exceptions at run-time. This happens to such an extent that many programming languages, like Smalltalk, C#[12], C++, Python, and Eiffel, completely omit checked exceptions.

The example in Figure 3 illustrates the consequences of adding a checked exception to existing software; it does not show the need for evolution, which is presented in [25; 24].

We have a class of bank accounts with methods to withdraw and deposit money. The `transferTo` method is the combination of a withdrawal followed by a deposit. The underlined exception is not part of the first version of the application. In the next version of our banking application, a client can only withdraw a limited amount of money every week. The `withdraw` method is modified, and signals a `WeekLimitException` when the withdrawal exceeds the week limit. Not only do we need to modify the `withdraw` method, but we must also change the exception clause of the `transferTo` method al-

```
public class Account {
   public void transferTo(Account other, double amount)
                  throws SuspiciousDepositException,
                         NotEnoughMoneyException,
                         WeekLimitException {
      withdraw(amount);
      other.deposit(amount);
   }
   public void withdraw(double amount)
                  throws NotEnoughMoneyException,
                         WeekLimitException {...}
   public void deposit(double amount)
                  throws SuspiciousDepositException {...}
}
```

3: Unnecessary modification of `transferTo`.

```
try{
  delegatee.execute(args);
}
catch(E1 exc) {
  // handle or propagate E1
  ...
}
catch(E2 exc) {
  // dummy exception handler to prevent E2
  // from showing up in the exception clause
}
```

4: A longer notation for `delegatee.execute(args)`.

though it is still just a withdrawal followed by a deposit.

3.2 Loss of Context Information

A programmer often knows that certain checked exceptions cannot be signalled by a method invocation when delegation is used. If he knows the type of the concrete delegatee that will be used by the delegator, he can eliminate certain checked exceptions based on the exception clause of the delegatee. The exception handling mechanism, however, cannot make the same deduction because, as discussed above, the delegator hides the exception clause of the delegatee.

Consider again the example in Figure 1b. Even if a programmer knows that the concrete delegatee cannot signal an exception at all, he will have to provide an exception handler for `E2` when invoking the delegator, as shown in Figure 4. He cannot use the context information about the delegatee to exclude exception `E2`. If `E1` is handled, only the handler for `E2` is useless. But if exception `E1` is propagated, this is a very long way to write `delegatee.execute(args)`.

An alternative approach would be to override the delegator specifically to deal with a certain type of delegatees, and filter out the checked exceptions in the delegator. But this

defeats the purpose of the design, providing many composed algorithms without changing the delegator, and is just as inconvenient.

As software evolves, the inconvenient situation turns into a dangerous one. Suppose that the delegatee now signals exceptions of type `E2`. The assumption under which the dummy exception handler for `E2` was valid, is no longer valid. But unless the entire program is manually verified, these dummy exception handlers will not be brought to attention, and exceptions will disappear at run-time. The problem can be alleviated by raising an unchecked exception in the dummy exception handler, but this exception can only be detected at run-time, and thus slip unnoticed through the testing phase.

4. ANCHORED EXCEPTION DECLARATIONS

Eiffel has a concept called *type anchoring* [22], to declare the type of an entity *relative* to the type of another entity, the *anchor*, within its scope. If the type of the anchor is changed, the type of the other entity automatically follows the change. You can define an entity relative to another entity using the following syntax :

some_entity: **like** *anchor*

In this section, we use the anchoring technique to solve the conflict between current exception clauses and the principle of abstraction. We extend the exception clause of a method to specify not only *what* exceptions can be signalled, but also *when* they can be signalled.

We had four goals when developing anchored exception declarations. First, anchored exception declarations must solve the adaptability problem; no unnecessary modifications of source code should be required when adding or removing checked exceptions from exception clauses. Second, compile-time safety must be guaranteed. Otherwise the very essence of using checked exceptions is lost. Third, the solution must be easy to use and understand by programmers. Finally, our solution must exploit as much type information as possible. If a programmer can narrow the set of exceptions signalled by a method invocation, based on static type information, anchored exception declarations must yield the same result.

4.1 Language simplifications

In order to simplify the formal semantics and the proof of correctness, we put some restrictions on the programming language. We use a variant of ClassicJava [7] where the `throws` clause and statements are added again. We limit expressions to `this`, references to formal parameters and fields, type names, and method invocations. Type names will only be valid when used in anchored exception declarations since ClassicJava does not model static methods and fields. Additionally, a class may not introduce a field with the same name as a field of one of its superclasses in order to simplify the lookup after substituting parameters. For methods, ClassicJava already takes care of this by forbidding syntactic overloading [23].

4.2 The Elements of an Anchored Exception Declaration

The addition of a new concept for specifying the exceptional behaviour of a method requires an extension of the terminology. An exception clause will no longer be a list of exception types, but a list of *exception declarations*. Each exception declaration declares

ExceptionClause:
  `throws` *ExceptionDeclaration (, ExceptionDeclaration)**
ExceptionDeclaration:
  AbsoluteExceptionDeclaration
  AnchoredExceptionDeclaration
AbsoluteExceptionDeclaration:
  Identifier
AnchoredExceptionDeclaration:
  `like` *MethodExpression [FilterClause]*
FilterClause:
  (`propagating (` *ExceptionList* `)`)?
      (`blocking (` *ExceptionList* `)`)?
ExceptionList:
  *Identifier (, ExceptionList)**
MethodExpression:
  MethodInvocation allowing type names as expressions

5: A grammar for anchored exception declarations.

what exceptions can be signalled under what 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.

To solve the conflict between checked exceptions and the principle of abstraction, we introduce *anchored exception declarations*. Instead of always declaring signalled exceptions in an absolute manner, a programmer can also declare them relative to another method using anchored exception declarations. An anchored exception declaration automatically reflects changes in the exception clause of its anchor.

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 Figure 5. 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.

To anchor the exception clause of a method to that of another method, it is not sufficient to use only the name of the other method. In order to exploit call-site type information, as seen in sections 4.5 and 4.6, and to ensure compile-time safety, as seen in section 4.7.4, the use of a method expression is required.

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 because some subexpressions of the method invocation may not always be visible outside the method body, or the programmer may want to hide them. The type name avoids ambiguities in presence of syntactic overloading [23].

The filter clause allows the developer to propagate only a limited set of exceptions using the `propagating` keyword, to propagate everything except for a set of exceptions using the `blocking` keyword, or a combination of both. The default filter clause – no filter clause – allows all exceptions of the anchor to be propagated.

Figure 6 shows a few anchored exception declarations. The syntax is chosen such that

void f() **throws like** g(), **like** h(x);

void f(A a) **throws like** a.g() **propagating** (E1, E2);

void f() **throws like** b().g(x) **blocking** (E1, E2);

void f() **throws like** A.g(X) **propagating** (E0) **blocking** (E1, E2);

6: Anchored exception declarations read like a sentence.

$$E \trianglelefteq T \Leftrightarrow \exists X \in T : E <: X$$

$$E \not\trianglelefteq T \Leftrightarrow \neg\, E \trianglelefteq T$$

$$\begin{aligned} T \sqcup S &= \{Type\ t \mid t \trianglelefteq T \vee t \trianglelefteq S\} \\ &= \{Type\ t \mid \exists X \in T : t <: X \vee \exists Y \in S : t <: Y\} \\ &= \{Type\ t \mid \exists X \in (T \cup S) : t <: X\} \\ &= \{T_1, \ldots, T_n, S_1, \ldots, S_m\} \end{aligned}$$

$$\begin{aligned} T \sqcap S &= \{Type\ t \mid t \trianglelefteq T \wedge t \trianglelefteq S\} \\ &= \{Type\ t \mid \exists X \in T : t <: X \wedge \exists Y \in S : t <: Y\} \\ &= \{Type\ t \mid t \in ((T_1 \cup \ldots \cup T_n) \cap (S_1 \cup \ldots \cup S_m))\} \\ &= \{Type\ t \mid t \in ((T_1 \cap S_1) \cup \ldots \cup (T_1 \cap S_m) \cup \ldots \cup \\ &\qquad (T_n \cap S_1) \cup \ldots \cup (T_n \cap S_m))\} \\ &= \{(T_1 \cap S_1), \ldots, (T_1 \cap S_m), \ldots, (T_n \cap S_1), \ldots, (T_n \cap S_m)\} \end{aligned}$$

$$Type_a \cap Type_b = \begin{cases} Type_a & \text{if } Type_a <: Type_b \\ Type_b & \text{if } Type_b <: Type_a \\ \emptyset & otherwise \end{cases}$$

$$T - S = \{Type\ t \mid t \trianglelefteq T \ \wedge \ t \not\trianglelefteq S\}$$

$$T \sqsubseteq S \Leftrightarrow \forall\, x \trianglelefteq T : X \trianglelefteq S$$

7: Operations on sets of types

an anchored exception declaration reads like a sentence.

4.3 Formal notation

We now define a shorter notation for exception declarations for use in formulas. Exception lists will be represented sets of types. The $\trianglelefteq$ operator checks whether or not 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 $\sqcap$,$\sqcup$, and $-$ operator correspond to the $\cap$, $\cup$, and $\setminus$ operators on normal sets, and the $\sqsubseteq$ relation corresponds to the $\subseteq$ relation. The symbol $\divideontimes$ represents a set containing every type. The operations are shown in figure 7.

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 will be non-empty for intermediate results during the expansion process (section 4.5).

An anchored exception declaration `like t.m(args) propagating (P) blocking (B)`, where `P` and `B` are exception lists, is denoted as $like\ t.m(args) \trianglelefteq P \not\trianglelefteq B$, where P and B are sets of exception types. The default values for P and B are $*$ and $\emptyset$.

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

4.4 Semantics

We now define the semantics of anchored exception declarations by introducting the δ function, which has two forms. The Υ and Ω functions are defined in section 4.5.

Definition 4.1. The first form of 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 $t.m(args)$, is allowed to signal a checked exception E if at least one of its exception declarations allows E to be signalled.

$$\delta(\{ED_1, \ldots, ED_n\}, t.m(args), E) \Leftrightarrow \bigvee_{i=1}^{i=n} \delta(ED_i, t.m(args), E)$$

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

$$\delta((P, B), t.m(args), E) \Leftrightarrow E \trianglelefteq (P - B)$$

—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.

$$\delta(like\ t_a.m_a(args_a) \trianglelefteq P_a \not\trianglelefteq B_a, t.m(args), E) \Leftrightarrow$$
$$\delta(\Upsilon(\Omega(like\ t_a.m_a(args_a) \trianglelefteq P_a \not\trianglelefteq B_a, t, args)), E)$$

Definition 4.2. The second form of the δ function determines the worst-case behaviour of an exception clause or declaration. It is a short-hand form for the first one when the target is the parent type of the method and the actual arguments are references to the formal parameters of the method.

— $\delta(\{ED_1, \ldots, ED_n\}, E) \Leftrightarrow \bigvee_{i=1}^{i=n} \delta(ED_i, E)$

— $\delta((P, B), E) \Leftrightarrow E \trianglelefteq (P - B)$

— $\delta(like\ t.m(args) \trianglelefteq P \not\trianglelefteq B, E) \Leftrightarrow$
$\delta(\Upsilon(like\ t.m(args) \trianglelefteq P \not\trianglelefteq B), E)$

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.

4.5 Exploiting Context Information

Call-site type information is inserted in an anchored exception declaration using a process called *expansion*, denoted by Υ, which is performed *at compile-time*. Expanding an anchored exception declaration is the process of cloning the exception clause of the invoked method and adapting it to include the context information.

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. Features that increase the power of expansion include covariant return types, generic parameters, and type anchors.

4.5.1 *Substitution.* Inserting context information into an exception clause is done using the Ω function. It substitutes formal parameters and the implicit argument $this$ with call-site information.

For the assumptions made in this paper, the Ω function is equal to $\{val_1/par_1 \ldots, val_n/par_n, target/this\}T$. If static methods, syntactic overloading, and overloading of instance variables are allowed, this is no longer the case then because lookups of instance variables and singatures are influenced by insertion of more specific type information. In this case, type elaboration can be used to take the static binding into account, as done in [7]. Of course, this function may only be applied when $ok_\Omega((this, target), (val_1, par_1), \ldots, (val_n, par_n))$ 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 $val_1, \ldots, val_n$ have the same type.

The definitions for expressions, exception declarations, and exception clauses are shown in Figure 8a. The $<:$ relation is used to denote subtyping for types and overriding for methods, the Γ function returns the type of an expression.

Note that the method binding mechanism used in an anchored exception declaration must be the same as the one used by the programming language for binding method invocations in the implementation. If the method being invoked at run-time does not correspond to the method resulting from evaluating the method expression, compile-time checks are useless. Again, the exception handling mechanism must correspond to the features of the language [31].

4.5.2 *Filtering.* The Φ function applies the filter clauses, P_{new} and B_{new}, of an anchored exception declaration to an exception clause. The propagated exceptions of an exception declaration are combined with P_{new} using an intersection. The blocked exceptions are combined with B_{new} using a union. The function is shown in Figure 8b

4.5.3 *Expansion.* The expansion of an anchored exception declaration , performed by the Υ function, selects the exception clause of the invoked method, done by 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 function is shown in Figure 8c. In the definition, p_i is the formal parameter corresponding to actual argument a_i.

4.5.4 *Recursive Expansion.* The Υ_{rec} function gives an upper bound for the types of checked exceptions that can be signalled by a method invocation, an exception declaration, or an exception clause. It uses the $\ominus$ operator to determine the types of exceptions sig-

$$ok_\Omega((v_{a,1},p_{a,1})\dots(v_{a,n},p_{a,n})) = \Big(\bigwedge_{i=1}^{i=n}\Gamma(v_{a,i}) <: \Gamma(p_{a,i})\Big) \wedge \Big(\bigwedge_{i=1,j=1}^{i=n,j=n}\Gamma(this(v_{a,i})) = \Gamma(this(v_{a,j}))\Big) \wedge (p_{a,i} = p_{a,j} \Leftrightarrow i = j)$$

$$\Omega(e, target, args) = \{val_1/par_1 \dots, val_n/par_n, pre/this\}e$$
$$\Omega((P,B), target, args) = (P,B)$$
$$\Omega(like\ t_a.m_a(args_a) \trianglelefteq P_a \not\trianglelefteq B_a, target, args) = like\ \Omega(t_a.m_a(args_a), target, args) \trianglelefteq P_a \not\trianglelefteq B_a$$
$$\Omega(\{ED_1,\dots,ED_n\}, target, args) = \{\Omega(ED_1, target, args),\dots,\Omega(ED_n, target, args)\}$$

(a) Substitution

$$\Phi((P,B),P_{new},B_{new}) = (P \sqcap P_{new}, B \sqcup B_{new})$$
$$\Phi(like\ t.m(args) \trianglelefteq P \not\trianglelefteq B, P_{new}, B_{new}) = like\ t.m(args) \trianglelefteq (P \sqcap P_{new}) \not\trianglelefteq (B \sqcup B_{new})$$
$$\Phi(\{ED_1,\dots,ED_n\},P,B) = \{\Phi(ED_1,P,B),\dots,\Phi(ED_n,P,B)\}$$

(b) Filtering

$$\Upsilon(like\ t.m(a_1,\dots,a_n) \trianglelefteq P \not\trianglelefteq B) = \Omega(\Phi(\varepsilon(t.m(a_1,\dots,a_n)),P,B), t, (a_1,p_1)\dots(a_n,p_n))$$
$$\Upsilon(t.m(a_1,\dots,a_n)) = \Upsilon(like\ t.m(a_1,\dots,a_n) \trianglelefteq * \not\trianglelefteq \emptyset)$$

(c) Expansion

$$\Upsilon_{rec}((P,B)) = P \ominus B$$
$$\Upsilon_{rec}(like\ t.m(args) \trianglelefteq P \not\trianglelefteq B) = \Upsilon_{rec}(\Upsilon(like\ t.m(args) \trianglelefteq P \not\trianglelefteq B))$$
$$\Upsilon_{rec}(\{ED_1,\dots,ED_n\}) = \Upsilon_{rec}(ED_1) \cup \dots \cup \Upsilon_{rec}(ED_n)$$
$$\Upsilon_{rec}(t.m(args)) = \Upsilon_{rec}(like\ t.m(args) \trianglelefteq * \not\trianglelefteq \emptyset)$$

(d) Recursive expansion

$$\{T_1,\dots,T_n\} \ominus \{S_1,\dots,S_m\} = \bigcup_{i=1}^{i=n}(T_i \ominus \{S_1,\dots,S_m\})$$
$$Type_a \ominus \{S_1,\dots,S_n\} = \begin{cases} \emptyset & \text{if } \exists\, Type_b \in \{S_1,\dots,S_n\} : Type_a <: Type_b \\ \{Type_a\} & otherwise \end{cases}$$

(e) Upper bound of absolute exception declarations

8: Definition of the expansion function.

nalled by absolute exception declarations because they can only appear in real exception clauses in the form $(P, \emptyset)$. The $\ominus$ operator calculates the worst case exception types for an absolute exception declaration by removing propagate types that are completely blocked and ignoring blocked types that do not completely block a propagated type. To prevent infinite loops in this process, we need to apply a restriction on the exception clause of a method, which is presented in section 4.7.2. THe function is shown in Figure 8d.

```
public class Account {
   public void transferTo(Account other, double amount)
                throws like withdraw(amount),
                        like other.deposit(amount) {
      withdraw(amount);
      other.deposit(amount);
   }
   public void withdraw(double amount)
                throws NotEnoughMoneyException,
                        WeekLimitException;
   public void deposit(double amount)
                throws SuspiciousDepositException;
}
public class UnsuspiciousAccount extends Account {
   public void deposit(double amount);
}
public class TimelessAccount extends Account {
   public void withdraw(double amount)
                throws NotEnoughMoneyException;
}

...
double myAmount = ...
UnsuspiciousAccount unsuspicious = ...
TimelessAccount timeless = ...

timeless.transferTo(unsuspicious, myAmount);
...
```

9: Class `Account` using anchored exception declarations.

4.6 Examples

Before we present the rules we impose on anchored exception declarations, we give two examples to make the reader familiar with them.

4.6.1 *Bank Account.* Using anchored exception declarations, the code for the second version of the bank accounts of Figure 3 would look as shown in Figure 9. We have now expressed that changes in the exceptional behaviour of `this.withdraw` and `other.deposit` will always be reflected in the set of exceptions signalled by `transferTo`. Consequently, the addition of `WeekLimitException` to `withdraw` does not require the modification of `transferTo`.

Figure 9 also shows two special classes of bank accounts. An `UnsuspiciousAccount` will never signal an exception when money is deposited, while a `TimelessAccount` does not have a week limit. The bottom of the figure shows a transfer from a `TimelessAccount` to an `UnsuspiciousAccount`.

In order to calculate which checked exceptions can be signalled by the transfer, we will expand the invocation. After the first expansion step of Υ_{rec}, the intermediate exception clause contains two anchored exception declarations.

(1) `like timeless.withdraw(myAmount)`
(2) `like unsuspicious.deposit(myAmount)`

Evaluating the method expressions will yield the overridden methods of classes `TimelessAccount` and `UnsuspiciousAccount`. The type information known about the target of the method invocation (`timeless`) and the first actual argument (`unsuspicious`) has allowed a more specialized selection of both methods than a selection based only on the exception clause of `Account.transferTo`. The benefits become clear after applying Υ_{rec} a last time.

(1) `NotEnoughMoneyException`
(2) $\emptyset$

Expanding `like timeless.withdraw(myAmount)` only results in `NotEnoughMoneyException` since `withdraw` does not signal `WeekLimitException` in class `TimelessAccount`. Likewise, the expansion of `like unsuspicious.deposit(myAmount)` yields an empty set because an `UnsuspiciousAccount` will never signal a `SuspiciousDepositException` during a deposit. The anchored exception declarations successfully removed all exceptions that cannot be signalled in the given context.

4.6.2 *Strategy Pattern.* Figure 10 shows an implementation of a strategy pattern. The `Predicate` class has a method `eval` that verifies whether an object satisfies a certain condition. It declares that it can signal any exception because it is a general purpose method. The `forAll` method implements a universal quantifier. This method also contains `Exception` in its exception clause because it uses a predicate to perform its job. Method `compatibleWith` of class `ExceptionClause` checks whether or not the current exception clause is compatible with another one, reusing the algorithm for universal quantification. As is clear from Figure 10, about half of its implementation is useless code to prevent `Exception` from showing up in the exception clause. Additionally, the dummy exception handler introduces a problem when the `compatibleWith` method of class `ExceptionDeclaration` signals additional checked exceptions after evolution of the code.

Figure 11 shows the same code using anchored exception declarations. The template algorithm `forAll` now declares that all checked exceptions come from the `eval(Object)` method. When the method expression `like predicate.eval(Object)` is evaluated in the context of the invocation of `forAll`, it will select the `eval(Object)` method of the anonymous inner class, which only signals `NotResolvedException`. Therefore no exception handling must be inserted for filtering exceptions, resulting in more elegant and safe code.

4.7 Restrictions on Anchored Exception Declarations

In this section, we will discuss the restrictions on anchored exception declarations. They are needed to ensure compile-time safety of anchored exception declarations.

4.7.1 *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 [22] and the accessibility constraints imposed on types used in method signatures in C# [6]. For

```
public abstract class Predicate {
   public abstract boolean eval(Object o)
                                  throws Exception;
}
public boolean forAll(Collection collection,
                      Predicate predicate)
                  throws Exception{...}
}
public class ExceptionClause {
   public boolean compatibleWith(
                      final ExceptionClause other)
                      throws NotResolvedException {
      try {
        return ...forAll(getDeclarations(),
           new Predicate() {
             public boolean eval(Object o) throws
                                  NotResolvedException {
               return ((ExceptionDeclaration)o).
                 compatibleWith(other);
             }});
      }
      catch (NotResolvedException e)
        {throw e;}
      catch (Exception e)
        {throw new Error();}
   }
}
```

10: Algorithm composition, the traditional way.

```
public boolean forAll(Collection collection,
                      Predicate predicate)
                  throws like predicate.eval(Object){...}
}
public class ExceptionClause {
   public boolean compatibleWith(
                      final ExceptionClause other)
                      throws NotResolvedException {
     return ...forAll(getDeclarations(),
        new Predicate() {
          public boolean eval(Object o) throws
                                  NotResolvedException {
            return ((ExceptionDeclaration)o).
              compatibleWith(other);
          }
        });
   }
}
```

11: Algorithm composition using anchored exception declarations.

example, it is not allowed to use a protected method in an anchored exception declaration of a public method because not every client may know about the protected method.

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

4.7.2 *Acyclic Anchor Graph.* In order to ensure that we cannot encounter a loop during the expansion process, we need to apply a restriction on the anchored exception declarations of a method. Using the anchored exception declarations as edges and methods as nodes, we can define a directed graph, called an *anchor graph*. For every anchored exception declaration, an edge is added starting from the parent method to the referenced method and all methods overriding it. The latter is required because submethods can be selected due to the insertion of context information.

RULE 2. *A program must have an acyclic anchor graph.*

This rule can be relaxed at the cost of requiring a whole program analysis. Instead of constructing a graph for each method, which covers the worst-case scenario, a graph can be constructed for each method invocation. This is similar to the system-wide check for polymorphic cat-calls in Eiffel. We chose not to use this relaxation because a whole program analysis is not appealing.

4.7.3 *Useful Anchor Rule.* The exception clause of a method may not contain an anchored exception declaration that does not declare any checked exception. Such an anchored exception declaration is useless, and can only cause confusion. Suppose that after evolution of the application a checked exception `E` can be signalled by `m` . If `E` can be signalled by an absolute exception declaration of EC_b, the anchored exception declaration is redundant. Otherwise, it breaks the compatibility between EC_a and EC_b.

RULE 3.

$$\Upsilon_{rec}(anchor) \neq \emptyset$$

4.7.4 *Compatibility Rules.* An exception clause EC_a is compatible with 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 compatibility relations must hold. The functions and relations used in these rules will be explained further on.

RULE 4. *A method may not signal a checked exception when one of its supermethods does not allow it.*

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

RULE 5. *The implementation of a method may not signal a checked exception when the exception clause does not allow it.*

$$\neg\, m\ abstract \Rightarrow IEC(m) \preceq \varepsilon(m)$$

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

We will now discuss compatibility of filter clauses, method expressions, anchored exception declarations and exception clauses, followed by the definition of the exception clause defined by the implementation of a method.

The $\preceq$ relation. We introduce the $\preceq$ relation in order to simplify reasoning about anchored exception declarations. For compile-safety, it would suffice to require that $\delta(EC_a, E) \Rightarrow \delta(EC_b, E)$ holds between a method and its supermethods 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.

A method expression ME_a is compatible with ME_b, denoted as $ME_a \preceq ME_b$, when the evaluation of ME_a always results in a method that is equal to, or overrides the method resulting from the evaluation of ME_b. Consequently, if $\text{ME}_\text{a} \preceq \text{ME}_\text{b}$, the method selected by ME_a can never signal an exception that is not allowed by the method selected by ME_b because of rules 4 and 5. The relations are shown in figure 12a. The $\cong$ relation denotes that both formal parameters are corresponding formal parameters of overriding or equal methods.

For absolute exception declarations, the set of exception declared by ABS_a must be a subset of those declared by ABS_b. For anchored exception declarations, their method expressions and their filter clauses must be compatible. The filter clauses follow the same rule as absolute declarations. Both relations are shown in figure 12b.

The $\preceq$ relation for exception clauses is shown in figure 12c. 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. Removing absolutely declared exceptions is of course allowed. Note that this rule forbids transforming anchored exception declarations into absolute declarations since an anchored declaration promises that an exception can only be signalled by the anchor, which is not the case for an absolute declaration. The set of checked exceptions for which $\delta((P_a, B_a), E)$ is true is $P_a \ominus B_a$.

The second condition states that anchored exception declarations of EC_b may be removed, copied, replaced by an anchored exception declaration that is compatible with $\text{AE}_\text{b,y}$ (`2.a`), and that a part of EC_b may be replaced by an anchored declaration that expands to an exception clause that is compatible with EC_b (`2.b`). The set of checked exceptions for which $\delta(ANCHOR_a, E)$ is true is $\Upsilon_{rec}(ANCHOR_a)$. Note that rule 3 cannot be integrated into condition 2 because it only holds for exception clauses that are part of the program an not for exception clauses that are the result of an expansion.

Rule `2.b` allows replacing a part of exception clause EC_b by an anchored exception declaration if the expansion of that anchored exception declaration is compatible with EC_b. For example, `E`$_1$`, like a().g()` may be replaced by `like a().f()` when the exception clause of `f()` is `E`$_1$`, like g()`. This is a valid transformation because it adds no extra exceptions or circumstances under which exceptions can be signalled; the expansion is compatible with the original exception clause. It does however create an opportunity for reducing the circumstances under which the exception can be signalled, by handling them in method `f()`. Because of the recursion in this condition, the rule must be used in conjunction with the requirement for an acyclic anchor graph.

The compatibility rule for exception clauses is related to the rules for refinement of

$$this_a \preceq this_b \Leftrightarrow \Gamma(this_a) <: \Gamma(this_b)$$

$$expression \preceq T \Leftrightarrow \Gamma(expression) <: T$$

$$\begin{array}{c} formal_a \preceq formal_b \\ \Updownarrow \\ formal_a \cong formal_b \end{array}$$

$$\begin{array}{c} new\ A(a_1, \ldots, a_n) \preceq new\ B(b_1, \ldots, b_n) \\ \Updownarrow \\ A = B \wedge \left(\bigwedge_{i=1}^{i=n} a_i \preceq b_i \right) \end{array}$$

$$t_a.var_a \preceq t_b.var_b \Leftrightarrow t_a \preceq t_b \wedge var_a = var_b$$

$$t_a.m(a_1, \ldots, a_n) \preceq t_b.m(b_1, \ldots, b_n) \Leftrightarrow t_a \preceq t_b \wedge \left(\bigwedge_{i=1}^{i=n} a_i \preceq b_i \right)$$

(a) Method expressions

$$(P_a, B_a) \preceq (P_b, B_b) \Leftrightarrow (P_a - B_a) \sqsubseteq (P_b - B_b)$$

$$\begin{array}{c} like\ t_a.m(arg_{a,1}, \ldots, arg_{a,n}) \trianglelefteq P_a \not\trianglelefteq B_a \preceq \\ like\ t_b.m(arg_{b,1}, \ldots, arg_{b,n}) \trianglelefteq P_b \not\trianglelefteq B_b \\ \Updownarrow \\ t_a.m(arg_{a,1}, \ldots, arg_{a,n}) \preceq \\ t_b.m(arg_{b,1}, \ldots, arg_{b,n}) \ \wedge\ (P_a - B_a) \sqsubseteq (P_b - B_b) \end{array}$$

(b) Exception declarations

$$\begin{array}{c} EC_a \preceq EC_b \\ \Updownarrow \end{array}$$

$$\begin{array}{lll} \forall (P_a,B_a)\ \in EC_a, \forall E | \delta((P_a, B_a), E) : & & (1) \\ \quad \exists (P_b, B_b) \in EC_b : \Phi((P_a, B_a), E, \emptyset) \preceq (P_b, B_b)) & & \\ \wedge\ \forall anchor_a \in EC_a, \forall E | \delta(anchor_a, E) : & & (2) \\ \quad \exists anchor_b \in EC_b : \Phi(anchor_a, E, \emptyset) \preceq anchor_b & & (2.a) \\ \quad \vee\ \Phi(\Upsilon(anchor_a), E, \emptyset) \preceq EC_b) & & (2.b) \end{array}$$

(c) Exception clauses

12: The compatibility relation $\preceq$.

$$\Psi([[throw(e)]]) = \{(\Gamma(e), (\Gamma(e), \emptyset))\} \cup \Psi([[e]])$$

$$\Psi([[t.m(args)]]) = \{(E, like\ t.m(args) \trianglelefteq E) | E \in \Upsilon_{rec}(t.m(args))\} \cup \Psi([[t]]) \cup \Psi([[args]])$$

$$\Psi([[try\{tb\}catch(E_1\ e_1)\{h_1\} \ldots catch(E_n\ e_n)\{h_n\}finally\{fin\}]]) = \{(E, ED) | (E, ED) \in \Psi([[tb]]) \wedge \nexists x \in [1, n] : E <: E_n\} \cup \left(\bigcup_{i=1}^{i=n} \Psi([[h_i]])\right) \cup \Psi([[fin]])$$

$$strip(\{(E_1, ED_1), \ldots, (E_n, ED_n)\}) = \{ED_1, \ldots, ED_n\}$$

$$IEC(method) = strip(\Psi(body(method)))$$

13: Calculation of the implementation exception clause.

reuse contracts [35], and the rules for conformance declarations of *Contracts* [16]. These rules enforce the substitution principle with respect to the specification of dependencies between methods. They involve either direct compatibility of elements, like rules `1` and `2.a`, or compatibility when taking the transitive closure of dependencies into account, like rule `2.b`.

The Implementation Exception Clause. The *implementation exception clause* (`IEC`) of a method is a calculated exception clause the declares what checked exception can be signalled by its implementation, and when they can be signalled. It is derived from a set of pairs containing the type of a checked exception and an exception declaration. The exception declaration is a declaration that represents a part of the exceptional behaviour of the implementation, while the exception type is used to filter pairs when an exception handler is encountered.

The algorithm to compute the IEC is similar to the *encounters* function presented by Robillard and Murphy [30], and is shown in Figure 13a. We do not give the definitions for every statement and expression, but only for the elements that are interesting with respect to the exception flow. For a checked exception that is raised directly, a pair is added that contains the static type of the exception as its first and second element. For a checked exception originating from a method invocation, the first element is the static type of the exception, and the second element is an anchored exception declaration containing the method invocation and a filter clause propagating only that type of exception. Adding multiple pairs that propagate only a single exception when encountering a method invocation simplifies the formula for exception handlers. A `try-catch-finally` block removes all exception pairs for which the exception type can be caught by one of its `catch` blocks. After that, exception pairs are added based on the code in the `catch` blocks and the `finally` block. Once the set is constructed for the method body, the implementation exception clause can be obtained by contructing an exception clause that contains the exception declaration of each pair.

The implementation exception clause specifies the worst-case run-time behaviour of a method body with respect to checked exceptions. In section 7, we will use this property to show that a method body will never signal an exception that was not declared by the exception clause of that method.

Note that the implementation exception clause is not always a valid exception clause

```
public class Account {
   public void transferTo(Account other, double amount)
                throws like withdraw(amount),
                       like other.deposit(amount) {...}

   public void withdraw(double amount)
                throws NotEnoughMoneyException,
                       WeekLimitException {...}

   public void deposit(double amount)
                throws SuspiciousDepositException {...}

   public void myTransaction() throws MyException,
                like Account.withdraw(double)
                     blocking (WeekLimitException),
                like Account.deposit(double) {...}
}
public class SpecialAccount extends Account {
   public Account someAccount() {...}
   public double someAmount() {...}
   public void myTransaction() throws MyException,
                like transferTo(Account, double)
                     blocking (WeekLimitException) {
      try {
       transferTo(someAccount(), someAmount());
       //{(NotEnoughMoneyException,
       //   like transferTo(someAccount(), someAmount())
       //     propagating (NotEnoughMoneyException))
       //  (WeekLimitException,
       //   like transferTo(someAccount(), someAmount())
       //     propagating (WeekLimitException))
       //  (SuspiciousDepositException,
       //   like transferTo(someAccount(), someAmount())
       //     propagating (SuspiciousDepositException))}
       if(...) {
         throw new MyException();
         // (MyException, MyException)
       }
      }
      catch(WeekLimitException exc)
        {... // no exceptions signalled here}
   }
}
```

14: Calculating the implementation exception clause.

since it may violate the accessibility rule.

Figure 14 illustrates the algorithm. The exception pairs are written in the comments after the corresponding statements. Exception `WeekLimitException` is caught by a `catch` clause that does not signal an exception. Therefore, all pairs within the body of

the `try` statement that have `WeekLimitException` as exception type can be removed. The resulting set of pairs is:

{(MyException, MyException),
 (NotEnoughMoneyException,
  like transferTo(someAccount(), someAmount())
   propagating (NotEnoughMoneyException)),
 (SuspiciousDepositException,
  like transferTo(someAccount(), someAmount())
   propagating (SuspiciousDepositException))}

The resulting implementation exception clause is:

MyException,
like transferTo(someAccount(), someAmount())
    propagating (NotEnoughMoneyException)
like transferTo(someAccount(), someAmount())
    propagating (SuspiciousDepositException)

4.8 Generic Parameters

Some of the effect of anchored exception declarations can be obtained by using generic parameters as exception types. Instead of using an anchored exception declaration, a programmer could use a generic parameter that is restricted to exception types, e.g. `PARAM extends Exception` in Java. This approach, however, is not nearly as elegant and flexible as using anchored exception declarations. The addition of generic parameters for exception handling clutters the code since they will appear everywhere in the static typing of the program. On top of that, the number of types of checked exceptions that a method can signal cannot exceed the number of generic parameters in its exception clause. As a result, the programmer could be forced to introduce new abstract exception types and provide wrappers for existing checked exceptions in order to get his code to compile. Finally, using this approach, the exceptional behaviour of a method is fixed at the construction time of an object, whereas an anchored exception declaration can exploit all static type information of every method invocation.

5. TRANSLATING CAPPUCCINO TO JAVA

We have implemented anchored exception declarations as an extension of ClassicJava, called *Cappuccino*. We have done this by adding elements representing anchored exception declarations to Jnome[34], a metamodel for Java, along with the algorithms necessary for validation. The extended metamodel reads ClassicJava files containing anchored exception declarations and checks all the rules they must adhere to.

A translator is provided to transform Cappuccino programs into plain Java programs. It replaces anchored exception declarations by absolute exception declarations and, if necessary, adds dummy exception handlers for checked exceptions that cannot be signalled. This is done by performing the following steps for each method.

(1) Transform anchored exception declarations into absolute exception declarations, which are calculated by the Υ_{rec} function.

(2) Remove redundant exception types from the new exception clause. That way, we can add exception handlers in any order.

```
public void transaction() throws NotEnoughMoneyException {
   try {
     new TimelessAccount().transferTo(
                new UnsuspiciousAccount(), 3.0);
   }
   catch (java.lang.RuntimeException Z)
     {throw Z;}
   catch (java.lang.Error Z)
     {throw Z;}
   catch (account.NotEnoughMoneyException Z)
     {throw Z;}
   catch (java.lang.Throwable Z)
     {throw new java.lang.Error();}
}
```

15: Generated code.

(3) Generate a unique name for the parameter of the `catch` clauses.

(4) Surround the body of the method with a `try` block.

(5) Add `catch` clauses for `Error` and `RuntimeException` that propagate the exception.

(6) For each checked exception declared by the new exception clause, calculate if it can be signalled by the method body by applying Υ_{rec} to the implementation exception clause. If that is the case, add a `catch` clause that propagates the exception.

(7) Finally, if `Throwable` is not already propagated, add a `catch` clause for `Throwable` that raises an `Error`. For a correct program this code will never be executed. Raising an `Error` in this handler can reveal some version conflicts between two parts of generated code.

Note that this algorithm is not ideal in terms of performance. By adding the exception handlers this way, every signalled exception will be caught and re-raised for every stack frame until the relevant handler is encountered.

Figure 15 contains the generated code for a method performing the transfer of money from a timeless account to an unsuspicious account from section 4.6.1.

6. METHODOLOGICAL DISCUSSION

6.1 Information Hiding

A consequence of the compatibility rule is that for a single anchored exception declaration $like\, t.m(args) \trianglelefteq P \not\trianglelefteq B$, the implementation may only signal checked exceptions caused by a compatible expression. But this is a violation of the principle of *information hiding* [26]. The anchored exception declaration reveals information about the implementation of the method, which must directly or indirectly execute `t.m(args)`. So how can anchored exception declarations fit in the object-oriented programming paradigm, where information hiding is a crucial concept?

The answer to this question has been given by Helm, Holland, and Gangopadhyay in [13] and by Steyaert, Lucas, Mens, and D'Hondt in [35]. In order to specify the behaviour of composable software elements, and thus allow a client to reuse them, it can be necessary to reveal some of the dependencies between the methods of these elements.

Remember that for queries, the contract of a delegator often contains expressions referencing the delegatee to allow the derivation of the full contract when the concrete delegatee is known. If the contract of the interface of the delegatee introduces indeterminism in the contract of the delegator, and the indeterministic part is relevant for a client of the delegator, the link between the delegator and the delegatee cannot be hidden. The contract of the delegator promises that the postconditions of the delegatee will be part of the result. But because the indeterminism prevents the exact postconditions from being known at compile-time, it is impossible for the delegator to satisfy its own contract without evaluating the expressions that reference the delegatee.

Consider for example the universal quantification of Figure 10. It is clear that the implementation of `forAll` must evaluate `predicate.eval(Object)` either directly or indirectly for every element of the collection in order to fulfill its contract because it cannot know the precise postconditions of the `eval` method of the given predicate in advance. So by using an anchored exception declaration, no extra information has been revealed.

For the examples like the `transferTo` method, it is not clear whether or not anchored exception declarations should be used. If a transfer of money must always have the same effect as a combination of a withdrawal and a deposit, the dependencies between the methods must be made explicit. If on the other hand, `withdraw` and `deposit` are just methods that are reused to implement the desired behaviour of `transferTo`, and there is no semantic connection, it is best to hide the dependencies.

From these arguments, we can derive a rule of thumb concerning the use of anchored exception declarations:

GUIDELINE 1. *Use an anchored exception declaration if the link between the delegator and the delegatee must be known by a client in order to use the delegator. Do not use an anchored exception declaration if that link must remain hidden for clients.*

6.2 Usefulness of Source Code Modifications

As mentioned in section 3.1, some modifications triggered by the addition of a checked exception make sense, while others do not.

If the modification concerns a method that handles at least one exception, the modification makes sense. For such a method, an active decision is taken to propagate some exceptions, but handle others, and so it is normal that this decision must be repeated when the exceptional behaviour has changed.

For methods that do not handle exceptions, it depends on whether or not the method already propagated checked exceptions. If the method did not propagate checked exceptions before, the modification makes sense. It is not realistic to expect that the exceptional specification of a method changes from “no checked exceptions” to “some checked exceptions” automatically. But if the method did signal checked exceptions before and the new checked exception is simply propagated, the modification is unnecessary. In this case, the method already propagated all checked exceptions coming from certain method invocations, so it should not be modified if such an invocation can result in a new checked exception. If the method must handle the exception, the modification makes sense.

Figure 16 illustrates the addition of a new checked exception for both absolute and anchored exception declarations. The new exception is not always propagated to the end of the chain of anchored exception declarations, such that the situation using anchored exception declarations will also require the modification of some methods. The anchored

16: Adding a new checked exception.

exception declarations always reference the next method in the chain. The circles represent methods, the lines represent chains of method invocations. The big circle in the middle is the method where the new exception was added. A circle is white if it is not modified, gray if it is modified and that modification makes sense, and black if it was modified unnecessarily. We assume that all anchored exception declarations are in place.

If only absolute exception declarations are used, as in the left figure, the exception must be propagated manually along the invocation chains until it is handled. The methods that handle the exception are colored gray; these changes are useful. The modifications that merely serve to propagate the exception until it can be handled are unnecessary, and thus colored black. If anchored exception declarations are used, as in the right figure, the exception automatically propagates to the end of the chain of anchored exception declarations. For the exceptions that should not reach that point, the programmer can backtrack along the invocation chain until he arrives at the method that should handle the exception. No unnecessary modifications must be performed.

7. PROOF OF CORRECTNESS

This section contains the soundness proof of anchored exception declarations. For the proof, we limit expressions to $this$, references to formal parameters and class variables, and method invocations. Additionally, type names may be used as expressions in method expressions.

7.1 Notation

In addition to the formal notation presented in section 4.3, we will need some extra notation during the proof.

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

For the substitution of parametrs in other parameters that are to be substituted, we write:

$$\Omega((val, par), pre, args) = (\Omega(val, pre, args), par)$$
$$\Omega((v_1, p_1) \ldots (v_n, p_n), pre, args) =$$
$$(\Omega(v_1, pre, args), p_1) \ldots (\Omega(v_n, pre, args), p_n)$$

7.2 Extension to the $\preceq$ relation

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

$$(val_a, param_a) \preceq (val_b, param_b) \Leftrightarrow$$
$$val_a \preceq val_b \ \wedge \ param_a \preceq param_b$$

$$(v_{a,1}, p_{a,1}) \ldots (v_{a,n}, p_{a,n}) \preceq (v_{b,1}, p_{b,1}) \ldots (v_{b,n}, p_{b,n}) \Leftrightarrow$$
$$(v_{a,1}, p_{a,1}) \preceq (v_{b,1}, p_{b,1}) \wedge \ldots \wedge (v_{a,n}, p_{a,n}) \preceq (v_{b,n}, p_{b,n})$$

7.3 Sets of types

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

LEMMA 7.1.

$$(P_a - B_a) \sqsubseteq (P_b - B_b) \ \wedge \ (P_c - B_c) \sqsubseteq (P_d - B_d)$$
$$\Downarrow$$
$$((P_a \sqcap P_c) - (B_a \sqcup B_c)) \sqsubseteq ((P_b \sqcap P_d) - (B_b \sqcup B_d))$$

PROOF.

$$(P_a - B_a) \sqsubseteq (P_b - B_b) \ \wedge \ (P_c - B_c) \sqsubseteq (P_d - B_d)$$
$$\Updownarrow (definitions\ of\ \sqsubseteq\ and\ -)$$
$$\forall x : ((x \trianglelefteq P_a \wedge x \not\trianglelefteq B_a) \Rightarrow (x \trianglelefteq P_b \wedge x \not\trianglelefteq B_b)) \wedge$$
$$((x \trianglelefteq P_c \wedge x \not\trianglelefteq B_c) \Rightarrow (x \trianglelefteq P_d \wedge x \not\trianglelefteq B_d))$$

$$\Downarrow$$
$$\forall x : (x \trianglelefteq P_a \wedge x \not\trianglelefteq B_a \wedge x \trianglelefteq P_c \wedge x \not\trianglelefteq B_c) \Rightarrow$$
$$(x \trianglelefteq P_b \wedge x \not\trianglelefteq B_b \wedge x \trianglelefteq P_d \wedge x \not\trianglelefteq B_d)$$
$$\Updownarrow (definitions\ of\ \sqcap\ and\ \sqcup)$$
$$\forall x : (x \trianglelefteq (P_a \sqcap P_c) \wedge x \not\trianglelefteq (B_a \sqcup B_c)) \Rightarrow (x \trianglelefteq (P_b \sqcap P_d) \wedge x \not\trianglelefteq (B_c \sqcup B_d))$$
$$\Updownarrow (definitions\ of\ \sqsubseteq\ and\ -)$$
$$((P_a \sqcap P_c) - (B_a \sqcup B_c)) \sqsubseteq ((P_b \sqcap P_d) - (B_b \sqcup B_d))$$

□

7.4 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 $f \circ g$ is equivalent to $g \circ f$, 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 7.2.

$$ok_\Omega(args, (pre, this(EC)))$$
$$\Downarrow$$
$$\Phi(\Omega(EC, pre, args), P, B) = \Omega(\Phi(EC, P, B), pre, args)$$

PROOF. 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)

$$\Phi(\Omega((P_x, B_x), pre, args), P, B) = \Omega(\Phi((P_x, B_x), P, B), pre, args)$$
$$\Updownarrow (definition\ of\ \Omega\ and\ \Phi)$$
$$\Phi((P_x, B_x), P, B) = \Omega((P_x \sqcap P, B_x \sqcup B), pre, args)$$
$$\Updownarrow (definition\ of\ \Omega)$$
$$\Phi((P_x, B_x), P, B) = (P_x \sqcap P, B_x \sqcup B)$$
$$\Updownarrow (definition\ of\ \Phi)$$
$$true$$

(2) $like\ t_x.m_x(args_x) \trianglelefteq P_x \not\trianglelefteq B_x$

$$\Phi(\Omega(like\ t_x.m_x(args_x) \trianglelefteq P_x \not\trianglelefteq B_x, pre, args), P, B) =$$
$$\Omega(\Phi(like\ t_x.m_x(args_x) \trianglelefteq P_x \not\trianglelefteq B_x, P, B), pre, args)$$
$$\Updownarrow (definition\ of\ \Omega\ and\ \Phi)$$
$$\Phi(like\ \Omega(t_x.m_x(args_x), pre, args) \trianglelefteq P_x \not\trianglelefteq B_x, P, B) =$$
$$\Omega(like\ t_x.m_x(args_x) \trianglelefteq (P_x \sqcap P) \not\trianglelefteq (B_x \sqcup B), pre, args)$$
$$\Updownarrow (definition\ of\ \Omega\ and\ \Phi)$$
$$like\ \Omega(t_x.m_x(args_x), pre, args) \trianglelefteq (P_a \sqcap P) \not\trianglelefteq (B_a \sqcup B) =$$
$$like\ \Omega(t_x.m_x(args_x), pre, args) \trianglelefteq (P_a \sqcap P) \not\trianglelefteq (B_a \sqcup B)$$

□

The second lemma states that if you perform two consecutive substitutions on an expression, that is equivalent to performing the last substitution on that actual arguments of the first substitution, and then applying the first substitution.

LEMMA 7.3.

$$ok_\Omega(args_a, (pre_a, this(expr))) \wedge ok_\Omega(args_b, (pre_b, this(pre_a))))$$
$$\forall formal \in expr : formal \in args_a$$
$$\Downarrow$$
$$\Omega(\Omega(expr, pre_a, args_a), pre_b, args_b) =$$
$$\Omega(expr, \Omega(pre_a, pre_b, args_b), \Omega(args_a, pre_b, args_b))$$

PROOF.

(1) $this$

$$\Omega(\Omega(this, pre_a, args_a), pre_b, args_b) =$$
$$\Omega(this, \Omega(pre_a, pre_b, args_b), \Omega(args_a, pre_b, args_b))$$
$$\Updownarrow (definition\ of\ \Omega)$$
$$\Omega(pre_a, pre_b, args_b) = \Omega(pre_a, pre_b, args_b)$$

(2) $typeName$

$$\begin{gathered}\Omega(\Omega(typeName, pre_a, args_a), pre_b, args_b) = \\ \Omega(typeName, \Omega(pre_a, pre_b, args_b), \Omega(args_a, pre_b, args_b)) \\ \Updownarrow (definition\ of\ \Omega) \\ \Omega(typeName, pre_b, args_b) = typeName \\ \Updownarrow (definition\ of\ \Omega) \\ typeName = typeName\end{gathered}$$

(3) $formal$: because of the precondition, $formal = par_i$ for exactly one (val_i, par_i) in $args_a$.

$$\begin{gathered}\Omega(\Omega(formal, pre_a, args_a), pre_b, args_b) = \\ \Omega(formal, \Omega(pre_a, pre_b, args_b), \Omega(args_a, pre_b, args_b)) \\ \Updownarrow (definition\ of\ \Omega) \\ \Omega(val_i, pre_b, args_b) = \Omega(val_i, pre_b, args_b)\end{gathered}$$

(4) $new\ C(a_1, \ldots, a_n)$

$$\begin{gathered}\Omega(\Omega(new\ C(a_1, \ldots, a_n), pre_a, args_a), pre_b, args_b) = \\ \Omega(new\ C(a_1, \ldots, a_n), \Omega(pre_a, pre_b, args_b), \Omega(args_a, pre_b, args_b)) \\ \Updownarrow (definition\ of\ \Omega) \\ \Omega(new\ C(\Omega(a_1, pre_a, args_a), \ldots, \Omega(a_n, pre_a, args_a)), pre_b, args_b) = \\ new\ C(\Omega(a_1, \Omega(pre_a, pre_b, args_b), \Omega(args_a, pre_b, args_b)), \ldots, \\ \Omega(a_n, \Omega(pre_a, pre_b, args_b), \Omega(args_a, pre_b, args_b))) \\ \Updownarrow (definition\ of\ \Omega) \\ new\ C(\Omega(\Omega(a_1, pre_a, args_a), pre_b, args_b), \ldots, \\ \Omega(\Omega(a_n, pre_a, args_a), pre_b, args_b)) = \\ new\ C(\Omega(a_1, \Omega(pre_a, pre_b, args_b), \Omega(args_a, pre_b, args_b)), \ldots, \\ \Omega(a_n, \Omega(pre_a, pre_b, args_b), \Omega(args_a, pre_b, args_b))) \\ \Updownarrow (induction\ on\ finite\ expression\ tree) \\ true\end{gathered}$$

(5) $t.var$

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

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

$$\begin{array}{c}
\Omega(\Omega(t.m(a_1,\ldots,a_n), pre_a, args_a), pre_b, args_b) = \\
\Omega(t.m(a_1,\ldots,a_n), \Omega(pre_a, pre_b, args_b), \Omega(args_a, pre_b, args_b)) \\
\Updownarrow (definition\ of\ \Omega) \\
\Omega(\Omega(t, pre_a, args_a).m(\Omega(a_1, pre_a, args_a), \ldots, \\
\Omega(a_n, pre_a, args_a)), pre_b, args_b) = \\
\Omega(t, \Omega(pre_a, pre_b, args_b), \Omega(args_a, pre_b, args_b)).m(\\
\Omega(a_1, \Omega(pre_a, pre_b, args_b), \Omega(args_a, pre_b, args_b)), \ldots, \\
\Omega(a_n, \Omega(pre_a, pre_b, args_b), \Omega(args_a, pre_b, args_b))) \\
\Updownarrow (definition\ of\ \Omega) \\
\Omega(\Omega(t, pre_a, args_a), pre_b, args_b).m(\\
\Omega(\Omega(a_1, pre_a, args_a), pre_b, args_b), \ldots, \\
\Omega(\Omega(a_n, pre_a, args_a), pre_b, args_b)) = \\
\Omega(t, \Omega(pre_a, pre_b, args_b), \Omega(args_a, pre_b, args_b)).m(\\
\Omega(a_1, \Omega(pre_a, pre_b, args_b), \Omega(args_a, pre_b, args_b)), \ldots, \\
\Omega(a_n, \Omega(pre_a, pre_b, args_b), \Omega(args_a, pre_b, args_b))) \\
\Updownarrow (induction\ on\ finite\ expression\ tree) \\
true
\end{array}$$

□

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

LEMMA 7.4.

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

PROOF. 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:

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

(1) (P, B)

$$\begin{array}{c}
\Omega(\Omega((P,B), pre_a, args_a), pre_b, args_b) = \\
\Omega((P,B), \Omega(pre_a, pre_b, args_b), \Omega(args_a, pre_b, args_b)) \\
\Updownarrow (definition\ of\ \Omega) \\
\Omega((P,B), pre_b, args_b) = (P,B) \\
\Updownarrow (definition\ of\ \Omega) \\
(P,B) = (P,B)
\end{array}$$

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

$$\begin{array}{c}
\Omega(\Omega(like\ t.m(args) \trianglelefteq P \not\trianglelefteq B, pre_a, args_a), pre_b, args_b) = \\
\Omega(like\ t.m(args) \trianglelefteq P \not\trianglelefteq B, \Omega(pre_a, pre_b, args_b), \Omega(args_a, pre_b, args_b)) \\
\Updownarrow (definition\ of\ \Omega) \\
\Omega(like\ \Omega(t.m(args), pre_a, args_a) \trianglelefteq P \not\trianglelefteq B, pre_b, args_b) = \\
like\ \Omega(t.m(args), \Omega(pre_a, pre_b, args_b), \Omega(args_a, pre_b, args_b)) \trianglelefteq P \not\trianglelefteq B \\
\Updownarrow (definition\ of\ \Omega) \\
like\ \Omega(\Omega(t.m(args), pre_a, args_a), pre_b, args_b) = \trianglelefteq P \not\trianglelefteq B \\
like\ \Omega(t.m(args), \Omega(pre_a, pre_b, args_b), \Omega(args_a, pre_b, args_b)) \trianglelefteq P \not\trianglelefteq B \\
\Updownarrow (Lemma 7.3) \\
true
\end{array}$$

□

7.5 Properties of the δ 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.

LEMMA 7.5.

$$\delta(ED, E) \Leftrightarrow \delta(\Phi(ED, E, \emptyset), E)$$

PROOF.

(1) (P, B)

$$\begin{array}{c}
\delta((P, B), E) \Leftrightarrow \delta(\Phi((P, B), E, \emptyset), E) \\
\Updownarrow (definition\ of\ \delta) \\
E \trianglelefteq (P - B) \Leftrightarrow E \trianglelefteq ((P \sqcap E) - B) \\
\Updownarrow \\
true
\end{array}$$

(2) $ANCHOR$: proven by induction on Lemma 7.6. This induction will end in absolute exception declarations for which the proof is given in the first part of this lemma.

$$\begin{array}{rl}
\delta(\Phi(ANCHOR, E, \emptyset), E) & \Leftrightarrow \delta(\Upsilon(\Phi(ANCHOR, E, \emptyset)), E) \\
& \Leftrightarrow \delta(\Phi(\Upsilon(ANCHOR), E, \emptyset), E) \\
& \Leftrightarrow \delta(\Upsilon(ANCHOR, E)) \\
& \Leftrightarrow \delta(ANCHOR, E)
\end{array}$$

□

The same property holds for exception clauses.

LEMMA 7.6.

$$\delta(EC, E) \Leftrightarrow \delta(\Phi(EC, E, \emptyset), E)$$

PROOF.

$$\begin{array}{c}
\delta(\Phi(EC, E, \emptyset), E) \Leftrightarrow \delta(\Phi(ED_1, E, \emptyset), E) \vee \ldots \vee \delta(\Phi(ED_n, E, \emptyset), E) \\
\Updownarrow (induction\ on\ Lemma\ 7.5) \\
\delta(\Phi(EC, E, \emptyset), E) \Leftrightarrow \delta(ED_1, E) \vee \ldots \vee \delta(ED_n, E) \\
\Updownarrow \delta(\Phi(EC, E, \emptyset), E) \Leftrightarrow \delta(EC, E)
\end{array}$$

□

7.6 Properties of the $\preceq$ relation

LEMMA 7.7. *If $expr_a$ is compatible with $expr_b$, the type of $expr_a$ is conform to the type of $expr_b$.*

$$expr_a \preceq expr_b \Rightarrow \Gamma(expr_a) <: \Gamma(expr_b)$$

PROOF. 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) $t.var$

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

(6) $t.m(args)$

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

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 7.8. *If anchored exception declaration $anchor_a$ is compatible with $anchor_b$, then the method referenced by $anchor_a$ will always be conform to the method referenced by $anchor_b$.*

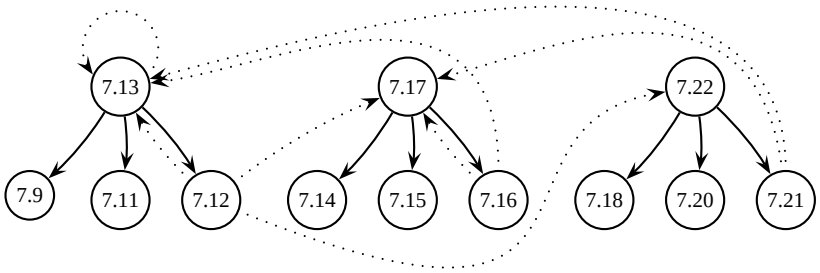
$$anchor_a \preceq anchor_b \Rightarrow method(anchor_a) <: method(anchor_b)$$

PROOF. Because of lemma 7.7, the types of the target and the actual arguments of $anchor_a$ will always be conform to the corresponding types of $anchor_b$. Consequently, because we do no allow syntactic overloading and require parameter types to be invariant, $anchor_a$ will always reference a method conform to the method referenced by $anchor_b$.

□

7.7 Overview of Dependencies

This section gives an overview of the dependencies in the proofs of Theorems 7.13, 7.17, and 7.22, and explains why the inductions that are used in their proofs will always end. This is also explained in the proofs themselves. This section merely serves to clarify the reasoning.



17: Dependency graph for Theorems 7.13, 7.17, and 7.22.

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 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. Since this graph contains no cycles, the induction will always end.

7.8 The $\preceq$ relation is transitive

7.8.1 *Absolute Exception Declarations*

LEMMA 7.9.

$$\Phi((P_a, B_a), E, \emptyset) \preceq (P_b, B_b) \wedge \Phi((P_b, B_b), E, \emptyset) \preceq (P_c, B_c)$$
$$\Downarrow$$
$$\Phi((P_a, B_a), E, \emptyset) \preceq (P_c, B_c)$$

PROOF.

$$\Phi((P_a, B_a), E, \emptyset) \preceq (P_b, B_b)$$
$$\Downarrow$$
$$((P_a \sqcap E) - B_a) \sqsubseteq (P_b - B_b)$$
$$\Downarrow$$
$$((P_a \sqcap E) - B_a) \sqsubseteq ((P_b \sqcap E) - B_b)$$
$$\Downarrow ((P_b \sqcap E) - B_b) \sqsubseteq (P_c - B_c)$$
$$((P_a \sqcap E) - B_a) \sqsubseteq (P_c - B_c)$$

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

7.8.2 *Method expressions*

LEMMA 7.10.

$$expr_a \preceq expr_b \wedge expr_b \preceq expr_c$$
$$\Downarrow$$
$$expr_a \preceq expr_c$$

PROOF. 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) $this_a, this_b, this_c$: follows directly from the transitivity of the subtyping ($<:$) relation.

(2) $expr_a, expr_b, type_c$: follows from Lemma 7.7 and the transitivity of the subtyping ($<:$) relation.

(3) $formal_a, formal_b, formal_c$: follows directly from the definition.

(4) $new\ C(args)$: follows from the definition and induction on this lemma.

(5) $t_a.var_a, t_b.var_b, t_c.var_c$: follows from the definition and induction on this lemma.

(6) $t_a.m(args_a), t_b.m(args_b), t_c.m(args_c)$: follows from the definition and induction on this lemma.

□

7.8.3 *Anchored Exception Declarations.*

Directly compatibility

LEMMA 7.11.

$$\begin{array}{c}\Phi(anchor_a, E, \emptyset) \preceq anchor_b \wedge \Phi(anchor_b, E, \emptyset) \preceq anchor_c \\ \Downarrow \\ \Phi(anchor_a, E, \emptyset) \preceq anchor_c\end{array}$$

PROOF. This lemma follows directly from Lemma 7.10 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 compatibility and compatibility after expansion

LEMMA 7.12.

$$\begin{array}{c}\Phi(anchor_a, E, \emptyset) \preceq anchor_b \wedge \Phi(\Upsilon(anchor_b), E, \emptyset) \preceq EC_c \\ \Downarrow \\ \Phi(\Upsilon(anchor_a), E, \emptyset) \preceq EC_c\end{array}$$

PROOF. Let $AED_a = like\ t_a.m(args_a) \trianglelefteq P_a \not\trianglelefteq B_a$ and $AED_b = like\ t_b.m(args_b) \trianglelefteq P_b \not\trianglelefteq B_b$.

$$\begin{array}{c}\Phi(anchor_a, E, \emptyset) \preceq anchor_b \\ \Downarrow (Lemma\ 7.15) \\ \Phi(anchor_a, E, \emptyset) \preceq \Phi(anchor_b, E, \emptyset) \\ \Downarrow (Lemma\ 7.8\ and\ rule\ @@@@@@@) \\ \varepsilon(\Phi(anchor_a, E, \emptyset)) \preceq \varepsilon(\Phi(anchor_b, E, \emptyset)) \\ \Downarrow (induction\ on\ Theorem\ 7.17) \\ \Phi(\varepsilon(\Phi(anchor_a, E, \emptyset)), P_a, B_a) \preceq \Phi(\varepsilon(\Phi(anchor_b, E, \emptyset)), P_b, B_b) \\ \Downarrow (induction\ on\ Theorem\ 7.22) \\ \Omega(\Phi(\varepsilon(\Phi(anchor_a, E, \emptyset)), P_a, B_a), t_a, args_a) \preceq \\ \Omega(\Phi(\varepsilon(\Phi(anchor_b, E, \emptyset)), P_b, B_b), t_b, args_b) \\ \Downarrow (definition\ of\ \Upsilon\ and\ Lemma\ 7.2) \\ \Phi(\Upsilon(anchor_a), E, \emptyset) \preceq \Phi(\Upsilon(anchor_b), E, \emptyset) \\ (induction\ on\ Theorem\ 7.13) \\ \Phi(\Upsilon(anchor_a), E, \emptyset) \preceq EC_c\end{array}$$

The inductions on Theorems 7.17, 7.22, and 7.13 will end because they all go back to Theorem 7.13 after performing an expansion. The **no-loops** rule ensures that every branch eventually ends up in Lemmas 7.9 or 7.11. □

7.8.4 *Exception Clauses*

THEOREM 7.13. *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. We must prove that:

$$\begin{pmatrix} \forall (P_a, B_a) \in EC_a, \forall E, \delta((P_a, B_a), E) : \\ \exists (P_c, B_c) \in EC_c : \\ \Phi((P_a, B_a), E, \emptyset) \preceq (P_c, B_c) \end{pmatrix} \wedge$$
$$\begin{pmatrix} \forall ANCHOR_a \in EC_a, \forall E : \delta(ANCHOR_a, E) : \\ (\exists ANCHOR_c \in EC_c : (\Phi(ANCHOR_a, E, \emptyset) \preceq ANCHOR_c \vee \\ \Phi(\Upsilon(ANCHOR_a), E, \emptyset) \preceq EC_c)) \end{pmatrix}$$

The case for absolute exception declarations is proven by Lemma 7.9. For anchored exception declarations of EC_a that are directly compatible with an anchored exception declaration of EC_b, the proof is given by Lemmas 7.11 and 7.12. For the case where $\Upsilon(AED_a) \preceq EC_b$, we apply induction on this theorem. Because of the **no-loops** rule and because an expansion is done between every two consecutive encounters of this theorem or the anchor is followed without insering context information, this induction will end in the first case (Lemma 7.9) or the second case (Lemma 7.12). □

7.9 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).

7.9.1 *Absolute Exception Declarations*

LEMMA 7.14.

$$(P_c - B_c) \sqsubseteq (P_d - B_d) \wedge \Phi((P_a, B_a), E, \emptyset) \preceq (P_b, B_b)$$
$$\Updownarrow$$
$$\Phi(\Phi((P_a, B_a), P_c, B_c), E, \emptyset) \preceq \Phi((P_b, B_b), P_d, B_d)$$

PROOF.

$$\Phi(\Phi((P_a, B_a), P_c, B_c), E, \emptyset) \preceq \Phi((P_b, B_b), P_d, B_d)$$
$$\Updownarrow (definition\ of\ \Phi)$$
$$(P_a \sqcap (P_c \sqcap E), B_a \sqcup B_c) \preceq (P_b \sqcap P_d, B_b \sqcup B_d)$$
$$\Updownarrow (definition\ of\ \preceq)$$
$$(((P_a \sqcap E) \sqcap P_c) - (B_a \sqcup B_c)) \sqsubseteq ((P_b \sqcap P_d) - (B_b \sqcup B_d))$$
$$\Uparrow (Lemma\ 7.1)$$
$$(P_a - B_a) \sqsubseteq (P_b - B_b) \wedge\ (P_c - B_c) \sqsubseteq (P_d - B_d)$$
$$\Updownarrow (definition\ of\ \preceq)$$
$$(P_a \sqcap E, B_a) \preceq (P_b, B_b)\ \wedge\ (P_c - B_c) \sqsubseteq (P_d - B_d)$$

□

7.9.2 *Anchored Exception Declarations.*

Direct compatibility

LEMMA 7.15.

$$\Phi(AED_a, E, \emptyset) \preceq AED_b \ \wedge \ (P_c - B_c) \sqsubseteq (P_d - B_d)$$
$$\Downarrow$$
$$\Phi(\Phi(AED_a, P_c, B_c), E, \emptyset) \preceq \Phi(AED_b, P_d, B_d)$$

PROOF.

$$like\ t_a.m_a(args_a) \trianglelefteq (P_a \sqcap E) \not\trianglelefteq B_a \preceq like\ t_b.m_b(args_b) \trianglelefteq P_b \not\trianglelefteq B_b) \wedge$$
$$(P_c - B_c) \sqsubseteq (P_d - B_d)$$
$$\Updownarrow (definition\ of\ \preceq)$$
$$t_a.m_a(args_a) \preceq t_b.m_b(args_b) \wedge$$
$$((P_a \sqcap E) - B_a) \sqsubseteq (P_b - B_b) \wedge \ (P_c - B_c) \sqsubseteq (P_d - B_d)$$

$$\Downarrow (lemma\ 7.1)$$
$$t_a.m_a(args_a) \preceq t_b.m_b(args_b) \wedge$$
$$(((P_a \sqcap E) \sqcap P_c) - (B_a \sqcup B_c)) \sqsubseteq ((P_b \sqcap P_d) - (B_b \sqcup B_d))$$
$$\Updownarrow (definitions\ of\ \preceq\ and\ \sqcup)$$
$$like\ t_a.m_a(args_a) \trianglelefteq (P_a \sqcap E \sqcap P_c) \not\trianglelefteq (B_a \sqcup E \sqcup B_c) \preceq$$
$$like\ t_b.m_b(args_b) \trianglelefteq (P_b \sqcap P_d) \not\trianglelefteq (B_b \sqcup B_d)$$
$$\Updownarrow (definition\ of\ \Phi)$$
$$\Phi(\Phi(AED_a, P_c, B_c), E, \emptyset) \preceq \Phi(AED_b, P_d, B_d)$$

□

Compatibility After Expansion

LEMMA 7.16.

$$\Phi(\Upsilon(anchor), E, \emptyset) \preceq EC_b \ \wedge \ (P_a - B_a) \sqsubseteq (P_b - B_b)$$
$$\Downarrow$$
$$\Phi(\Upsilon(\Phi(anchor, P_a, B_a)), E, \emptyset) \preceq \Phi(EC_b, P_b, B_b)$$

PROOF. We prove this using induction on Theorem 7.17. We expand the anchored exception declaration one level and assume that Theorem 7.17 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 AED, or one of its submethods, with context information inserted. Because of the **no-loops** rule, the recursion must end in methods of which the exception clauses contain no anchored exception declarations. These will provide the base case.

—Induction step

The preconditions of Theorem 7.17 follow directly from the preconditions of this

lemma.

$$\begin{array}{c}\Phi(\Upsilon(anchor), E, \emptyset) \preceq EC_b \\ \Downarrow (induction\ on\ Theorem\ 7.17) \\ \Phi(\Phi(\Upsilon(anchor), E, \emptyset), P_a, B_a) \preceq \Phi(EC_b, P_b, B_b) \\ \Downarrow (induction\ on\ Theorem\ 7.13) \\ \left(\begin{array}{c}\Phi(\Upsilon(\Phi(anchor, P_a, B_a)), E, \emptyset) \preceq \Phi(\Phi(\Upsilon(anchor), E, \emptyset), P_a, B_a) \\ \Downarrow \\ \Phi(\Upsilon(\Phi(anchor, P_a, B_a)), E, \emptyset) \preceq \Phi(EC_b, P_b, B_b)\end{array}\right)\end{array}$$

As explained in the proof of Theorem 7.13 the transitivity property of $\preceq$ is indirectly based on this lemma. Because of the expansion done in this lemma and the **no-loops** rule, the induction must end. On this side, it will end in either Lemma 7.14 or 7.15. Now we only need to prove the left-hand side of the last implication.

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

$$\begin{array}{c}\Phi(\Upsilon(\Phi(anchor, P_a, B_a)), E, \emptyset) \preceq \Phi(\Phi(\Upsilon(anchor), E, \emptyset), P_a, B_a) \\ \Updownarrow (definition\ of\ \Upsilon\ and\ \Phi) \\ \Phi(\Omega(\Phi(\varepsilon(\Phi(anchor, P_a, B_a)), (P \sqcap P_a), (B \sqcup B_a)), t, args), E, \emptyset) \preceq \\ \Phi(\Omega(\Phi(\varepsilon(anchor), P, B), t, args), (P_a \sqcap E), B_a)\end{array}$$

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

$$\begin{array}{c}\Updownarrow \\ \Phi(\Omega(\Phi(\varepsilon(anchor), (P \sqcap P_a), (B \sqcup B_a)), t, args), E, \emptyset) \preceq \\ \Phi(\Omega(\Phi(\varepsilon(anchor), P, B), t, args), (P_a \sqcap E), B_a) \\ \Updownarrow (Lemma\ 7.2) \\ \Omega(\Phi(\Phi(\varepsilon(anchor), (P \sqcap P_a), (B \sqcup B_a)), E, \emptyset), t, args) \preceq \\ \Omega(\Phi(\Phi(\varepsilon(anchor), P, B), (P_a \sqcap E), B_a), t, args) \\ \Updownarrow (definitions\ of\ \Phi, \sqcup\ and\ \sqcap) \\ \Omega(\Phi(\varepsilon(anchor), (P \sqcap P_a \sqcap E), (B \sqcup B_a)), t, args) \preceq \\ \Omega(\Phi(\varepsilon(anchor), (P \sqcap P_a \sqcap E), (B \sqcup B_a)), t, args) \\ \Updownarrow (definition\ of\ \preceq) \\ true\end{array}$$

—For the base case, we need to prove Theorem 7.17 when $\varepsilon(anchor)$ contains no anchored exception declarations. Since $\varepsilon(anchor)$ only contains absolute exception declarations, the last two conditions in the proof of Theorem 7.17 disappear. The remaining condition is proven by Lemma 7.14.

□

7.9.3 *Exception Clauses*

THEOREM 7.17.

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

PROOF.

$$
\begin{array}{c}
\Phi(EC_a, P_c, B_c) \preceq \Phi(EC_b, P_d, B_d) \\
\Updownarrow (definition\ of\ \preceq) \\
\left(\begin{array}{c}
\left(\begin{array}{c}
\forall \Phi((P_a, B_a), P_c, B_c) \in \Phi(EC_a, P_c, B_c), \forall E, \delta(\Phi((P_a, B_a), P_c, B_c), E) : \\
\exists \Phi((P_b, B_b), P_d, B_d) \in \Phi(EC_b, P_d, B_d) : \\
\Phi(\Phi((P_a, B_a), P_c, B_c), E, \emptyset) \preceq \Phi((P_b, B_b), P_d, B_d)
\end{array}\right) \\
\wedge \\
\left(\begin{array}{c}
\forall \Phi(ANCHOR_a, P_c, B_c) \in \Phi(EC_a, P_c, B_c), \forall E : \delta(\Phi(ANCHOR_a, P_c, B_c), E) : \\
(\exists \Phi(ANCHOR_b, P_d, B_d) \in \Phi(EC_b, P_d, B_d) : \\
(\Phi(\Phi(ANCHOR_a, P_c, B_c), E, \emptyset) \preceq \Phi(ANCHOR_b, P_d, B_d) \vee \\
\Phi(\Upsilon(\Phi(ANCHOR_a, P_c, B_c)), E, \emptyset) \preceq \Phi(EC_b, P_d, B_d)))
\end{array}\right)
\end{array}\right)
\end{array}
$$

Let $ABS_a, AED_a \in EC_a$ and $ABS_b, AED_b \in EC_b$. From Lemmas 7.14, 7.15, 7.25, and 7.26, we know that $\delta(\Phi(ED, P_c, B_c), E) \Rightarrow \delta(ED, E)$, so if the left-hand side is true, we know that the corresponding condition in $EC_a \preceq EC_b$ must be true and can be used as a precondition.

$$
\begin{array}{c}
\Uparrow (EC_a \preceq EC_b) \\
\left(\begin{array}{c}
\left(\begin{array}{c}
(P_c - B_c) \sqsubseteq (P_d - B_d) \ \wedge\ \Phi(ABS_a, E, \emptyset) \preceq ABS_b \\
\Downarrow \\
\Phi(\Phi(ABS_a, P_c, B_c), E, \emptyset) \preceq \Phi(ABS_b, P_d, B_d)
\end{array}\right) \\
\wedge \\
\left(\begin{array}{c}
(P_c - B_c) \sqsubseteq (P_d - B_d) \ \wedge\ \Phi(AED_a, E, \emptyset) \preceq AED_b \\
\Downarrow \\
\Phi(\Phi(AED_a, P_c, B_c), E, \emptyset) \preceq \Phi(AED_b, P_d, B_d)
\end{array}\right) \\
\wedge \\
\left(\begin{array}{c}
(P_c - B_c) \sqsubseteq (P_d - B_d) \ \wedge\ \Phi(\Upsilon(AED_a), E, \emptyset) \preceq EC_b \\
\Downarrow \\
\Phi(\Upsilon(\Phi(AED_a, P_c, B_c)), E, \emptyset) \preceq \Phi(EC_b, P_d, B_d)
\end{array}\right)
\end{array}\right) \\
\\
\Updownarrow \\
Lemma\ 7.14 \wedge Lemma\ 7.15 \wedge Lemma\ 7.16
\end{array}
$$

□

7.10 Ω is monotone

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

7.10.1 *Absolute Exception Declarations*

LEMMA 7.18.

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

PROOF.

$$\begin{array}{c}\Phi(\Omega((P_a, B_a), pre_a, a_1 \ldots a_n), E, \emptyset) \preceq \Omega((P_b, B_b), pre_b, b_1 \ldots b_n) \\ \Updownarrow (definition\ of\ \Omega) \\ \Phi((P_a, B_a), E, \emptyset) \preceq (P_b, B_b)\end{array}$$

□

7.10.2 *Method Expressions*

LEMMA 7.19.

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

PROOF. Let $args_a = a_1 \ldots a_n$ and $args_b = b_1 \ldots b_n$.

(1) $this$

$$\begin{array}{c}\Omega(this_a, pre_a, a_1 \ldots a_n) \preceq \Omega(this_b, pre_b, b_1 \ldots b_n) \\ \Updownarrow (definition\ of\ \Omega) \\ pre_a \preceq pre_b\end{array}$$

(2) $type$

$$\begin{array}{c}\Omega(type_a, pre_a, a_1 \ldots a_n) \preceq \Omega(type_b, pre_b, b_1 \ldots b_n) \\ \Updownarrow (definition\ of\ \Omega) \\ type_a \preceq type_b\end{array}$$

(3) $formal$

$$\begin{array}{c}\Omega(formal_a, pre_a, (v_{a,1}, p_{a,1}) \ldots (v_{a,n}, p_{a,n})) \preceq \\ \Omega(formal_b, pre_b, (v_{b,1}, p_{b,1}) \ldots (v_{b,n}, p_{b,n}))\end{array}$$

(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) $formal_a \neq p_{a,i}$
Because of the definition of the $\preceq$ relation and the given assumptions, $formal_b \neq p_{b,i}$.

$$formal_a \preceq formal_b$$

(4) $new\ C(args)$

$$\begin{array}{c}\Omega(new\ C(args_a), pre_a, a_1 \ldots a_n) \preceq \Omega(new\ C(args_b), pre_b, b_1 \ldots b_n) \\ \Updownarrow (definition\ of\ \Omega) \\ \Omega(args_a, pre_a, a_1 \ldots a_n) \preceq \Omega(args_b, pre_b, b_1 \ldots b_n) \\ \Updownarrow (induction\ on\ finite\ expression\ tree) \\ true\end{array}$$

(5) $t.var$

$$\begin{array}{c}\Omega(t_a.var_a, pre_a, a_1 \ldots a_n) \preceq \Omega(t_b.var_b, pre_b, b_1 \ldots b_n)\\ \Updownarrow (definition\ of\ \Omega)\\ \Omega(t_a, pre_a, a_1 \ldots a_n).var_a \preceq \Omega(t_b, pre_b, b_1 \ldots b_n).var_b\\ \Updownarrow (definition\ of\ \preceq)\\ \Omega(t_a, pre_a, a_1 \ldots a_n) \preceq \Omega(t_b, pre_b, b_1 \ldots b_n)\ \wedge var_a \preceq var_b\\ \Updownarrow (induction\ on\ finite\ expression\ tree)\\ true\end{array}$$

(6) $t.m(args)$

$$\begin{array}{c}\Omega(t_a.m(arg_{a,1}, \ldots, arg_{a,n}), pre_a, a_1 \ldots a_n) \preceq\\ \Omega(t_b.m(arg_{b,1}, \ldots, arg_{b,n}), pre_b, b_1 \ldots b_n)\\ \Updownarrow (definition\ of\ \Omega)\\ \Omega(t_a, pre_a, a_1 \ldots a_n).m(\Omega(arg_{a,1}, pre_a, a_1 \ldots a_n), \ldots\\ , \Omega(arg_{a,n}, pre_a, a_1 \ldots a_n)) \preceq\\ \Omega(t_b, pre_b, b_1 \ldots b_n).m(\Omega(arg_{b,1}, pre_b, b_1 \ldots b_n), \ldots\\ , \Omega(arg_{b,n}, pre_b, b_1 \ldots b_n))\\ \Updownarrow (definition\ of\ \preceq)\\ \Omega(t_a, pre_a, a_1 \ldots a_n) \preceq \Omega(t_b, pre_b, b_1 \ldots b_n) \wedge\\ \Omega(arg_{a,1}, pre_a, a_1 \ldots a_n) \preceq \Omega(arg_{b,1}, pre_b, b_1 \ldots b_n) \wedge \ldots \wedge\\ \Omega(arg_{a,n}, pre_a, a_1 \ldots a_n) \preceq \Omega(arg_{b,n}, pre_b, b_1 \ldots b_n)\\ \Updownarrow (induction\ on\ finite\ expression\ tree)\\ true\end{array}$$

□

7.10.3 *Anchored Exception Declarations.*

Direct Compatibility

LEMMA 7.20.

$$\begin{array}{c}pre_a \preceq pre_b \wedge args_a \preceq args_b \wedge \Phi(anchor_a, E, \emptyset) \preceq anchor_b\\ ok_\Omega(args_a, (pre_a, this(anchor_a))) \wedge ok_\Omega(args_b, (pre_b, this(anchor_b)))\\ \Downarrow\\ \Phi(\Omega(anchor_a, pre_a, args_a), E, \emptyset) \preceq \Omega(anchor_b, pre_b, args_b)\end{array}$$

PROOF. Let $anchor_a = like\ t_a.m_a(a_1, \ldots, a_n) \trianglelefteq P_a \not\trianglelefteq B_a$, and let $anchor_b = like\ t_b.m_b(b_1, \ldots, b_n) \trianglelefteq P_b \not\trianglelefteq B_b$.

$$\begin{array}{c}\Phi(\Omega(anchor_a, pre_a, args_a), E, \emptyset) \preceq \Omega(anchor_b, pre_b, args_b)\\ \Updownarrow\\ like\ \Omega(t_a.m_a(a_1 \ldots a_n), pre_a, args_a) \trianglelefteq (P_a \sqcap E) \not\trianglelefteq B_a \preceq\\ like\ \Omega(t_b.m_a(b_1 \ldots b_n), pre_b, args_b) \trianglelefteq P_b \not\trianglelefteq B_b\\ \Updownarrow (definition\ of\ \preceq)\\ \Omega(t_a.m_a(a_1 \ldots a_n), pre_a, args_a) \preceq \Omega(t_b.m_b(b_1 \ldots b_n), pre_b, args_b) \wedge\\ ((P_a \sqcap E) - B_a) \sqsubseteq (P_b - B_b)\\ \Uparrow (Lemma\ 7.19\ and\ preconditions)\\ ((P_a \sqcap E) - B_a) \sqsubseteq (P_b - B_b)\\ \Updownarrow (\Phi(anchor_a, E, \emptyset) \preceq anchor_b)\\ true\end{array}$$

□

Compatibility After Expansion

LEMMA 7.21. *Let* $anchor = like\ t.m(arg_1, \ldots, arg_m)$.

$$\begin{array}{c} pre_a \preceq pre_b \wedge args_a \preceq args_b \wedge \Phi(\Upsilon(anchor), E, \emptyset) \preceq EC_b \wedge \\ ok_\Omega(args_a, (pre_a, this(anchor))) \wedge ok_\Omega(args_b, (pre_b, this(EC_b))) \\ \Downarrow \\ \Phi(\Upsilon(\Omega(anchor, pre_a, args_a)), E, \emptyset) \preceq \Omega(EC_b, pre_b, args_b) \end{array}$$

PROOF. We prove the lemma using induction on Theorems 7.17 and 7.22. 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 of the **no-loops** rule, the restriction to finite programs, and because these lemmas themselves only perform further expansions, this induction must end in methods whose exception clauses contain no anchored exception declarations. These will provide the base case.

—Induction step

Before we perform the induction on $\Upsilon(anchor)$ and EC_b, we need to verify that the precondition of Theorem 7.22 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.

$$\begin{array}{c} \Phi(\Upsilon(anchor), E, \emptyset) \preceq EC_b \\ \Downarrow (induction\ on\ Theorem\ 7.22) \\ \Omega(\Phi(\Upsilon(anchor), E, \emptyset), pre_a, args_a) \preceq \Omega(EC_b, pre_b, args_b) \\ \Downarrow (induction\ on\ Theorem\ 7.13) \\ \left(\begin{array}{c} \Phi(\Upsilon(\Omega(anchor, pre_a, args_a)), E, \emptyset) \preceq \\ \Omega(\Phi(\Upsilon(anchor), E, \emptyset), pre_a, args_a) \\ \Downarrow \\ \Phi(\Upsilon(\Omega(anchor, pre_a, args_a)), E, \emptyset) \preceq \Omega(EC_b, pre_b, args_b) \end{array} \right) \end{array}$$

As explained in the proof of Theorem 7.13 the transitivity property of $\preceq$ is indirectly based on this lemma. Because of the expansion done in this lemma and the **no-loops** rule, the induction must end. On this side, it will end in either Lemma 7.18 or 7.20. Now we only need to prove the left-hand side of the last implication. From the definitions of Ω and Ω, we know that:

$$\begin{array}{c} \Omega(anchor, pre_a, args_a) = like\ \Omega(t, pre_a, args_a).m(\Omega(arg_1, pre_a, args_a), \\ \ldots, \Omega(arg_m, pre_a, args_a)) \trianglelefteq P \not\trianglelefteq B \end{array}$$

The actual arguments $arg_1, \ldots, arg_m$ are bound respectively to formal parameters

$par_1, \ldots par_m$. As a result, we can prove the induction step as follows:

$$\begin{gathered}\Phi(\Upsilon(\Omega(anchor, pre_a, args_a)), E, \emptyset) \preceq \\ \Omega(\Phi(\Upsilon(anchor), E, \emptyset), pre_a, args_a) \\ \Updownarrow (definition\ of\ \Upsilon) \\ \Phi(\Omega(\Phi(\varepsilon(\Omega(anchor, pre_a, args_a)), P, B), \Omega(t, pre_a, args_a), \\ \Omega((arg_1, par_1), pre_a, args_a) \ldots \Omega((arg_m, par_m), pre_a, args_a)), E, \emptyset) \preceq \\ \Omega(\Phi(\Omega(\Phi(\varepsilon(anchor), P, B), t, \\ (arg_1, par_1) \ldots (arg_m, par_m)), E, \emptyset), pre_a, args_a)\end{gathered}$$

Because $\varepsilon(anchor)$ is the exception clause of a method of the program, it can only reference the formal parameters of its method, being $par_1, \ldots, par_m$. As a result, Lemma 7.4 may be applied. The filter operations may be merged due to Lemma 7.2 and the definition of Φ.

$$\begin{gathered}\Updownarrow (Lemma\ 7.4) \\ \Omega(\Phi(\varepsilon(\Omega(anchor, pre_a, args_a)), P \sqcap E, B), \Omega(t, pre_a, args_a), \\ \Omega((arg_1, par_1), pre_a, args_a) \ldots \Omega((arg_m, par_m), pre_a, args_a)) \preceq \\ \Omega(\Phi(\varepsilon(anchor), P \sqcap E, B), \Omega(t, pre_a, args_a), \\ \Omega((arg_1, par_1), pre_a, args_a) \ldots \Omega((arg_m, par_m), pre_a, args_a))\end{gathered}$$

Because of Lemma 7.7, Lemma 7.19, and the preconditions of this lemma, we know that:

$$method(\Omega(anchor, pre_a, args_a)) <: method(anchor)$$

As a result, we know that according to rule **@#@#@#@#@**:

$$\varepsilon(\Omega(anchor, pre_a, args_a)) \preceq \varepsilon(anchor)$$

Now we use induction on Theorems 7.17 and 7.22 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 7.17 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 Theorem 7.22 follows from the application of Theorem 7.17. The second and third preconditions are satisfied because the prefixes and actual arguments are identical. The last preconditions are satisfied because of Lemmas 7.7 and 7.19.

—For the base case, we need to prove Theorems 7.17 and 7.22 when $\varepsilon(anchor)$ contains no anchored exception declarations, and the method referenced by $anchor$ has no submethods. For Theorem 7.17 the proof is included in its own proof. We will now prove the base case for Theorem 7.22. Since $\varepsilon(anchor)$ only contains absolute exception declarations, the last two conditions in the proof of Theorem 7.22 disappear. The remaining condition is proven by Lemma 7.18.

□

7.10.4 *Exception Clauses*

THEOREM 7.22.

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

PROOF. The proof of this lemma is similar to that of Theorem 7.17. After rewriting the expression $\Omega(EC_a, pre_a, args_a) \preceq \Omega(EC_b, pre_b, args_b)$, we obtain the following conditions for this lemma to be true:

$$\begin{array}{c} \left(\begin{array}{c} \Phi(ABS_a, E, \emptyset) \preceq ABS_b \\ \Downarrow \\ \Phi(\Omega(ABS_a, pre_a, args_a), E, \emptyset) \preceq \Omega(ABS_b, pre_b, args_b) \end{array} \right) \\ \wedge \\ \left(\begin{array}{c} pre_a \preceq pre_b \wedge args_a \preceq args_b \wedge \Phi(anchor_a, E, \emptyset) \preceq anchor_b \\ \wedge ok_\Omega(args_a, (pre_a, this(anchor_a))) \wedge ok_\Omega(args_b(pre_b, this(anchor_b))) \\ \Downarrow \\ \Phi(\Omega(anchor_a, pre_a, args_a), E, \emptyset) \preceq \Omega(anchor_b, pre_b, args_b) \end{array} \right) \\ \wedge \\ \left(\begin{array}{c} pre_a \preceq pre_b \wedge args_a \preceq args_b \wedge \Phi(\Upsilon(anchor), E, \emptyset) \preceq EC_b \wedge \\ ok_\Omega(args_a, (pre_a, this(anchor))) \wedge ok_\Omega(args_b, (pre_b, this(EC_b))) \\ \Downarrow \\ \Phi(\Upsilon(\Omega(anchor, pre_a, args_a)), E, \emptyset) \preceq \Omega(EC_b, pre_b, args_b) \end{array} \right) \\ \\ \Updownarrow \\ Lemma\ 7.18 \wedge Lemma\ 7.20 \wedge Lemma\ 7.21 \end{array}$$

□

7.11 The Implementation Exception Clause is an Upper Bound

THEOREM 7.23. *The implementation exception clause of a non-abstract method is an upper bound for the exceptional behaviour of the implementation of that method.*

PROOF. This theorem follow obviously from the definition of the implementation exception clause and the Java Language Specification. □

7.12 Method Invocations Maintain Compatibility

THEOREM 7.24. *Let $t.m(arg_1, \ldots, arg_n)$ be a method invocation in a valid program, let $EC_b = \varepsilon(t.m(arg_1, \ldots, arg_n))$ and let par_i be the formal parameter corresponding to arg_i.*

$$\begin{array}{c} EC_a \preceq EC_b \wedge \Gamma(this(EC_b)) = \Gamma(this(EC_a)) \\ \\ \Downarrow \\ \Omega(EC_a, t, (arg_1, par_1) \ldots (arg_n, par_n)) \preceq \Upsilon(t.m(args)) \end{array}$$

PROOF. The requirements for substitution in EC are satisfied because the program is valid. Since $EC \preceq IEC$, they are also valid if the type of $this$ is the same. For formal

parameters, the type must be invariant.

$$\begin{array}{c}\Omega(EC_a, t, args) \preceq \Upsilon(t.m(args)) \\ \Updownarrow \\ \Omega(EC_a, t, args) \preceq \\ \Omega(\Phi(EC_b, \divideontimes, \emptyset), t, args) \\ \Updownarrow \\ \Omega(EC_a, t, args) \preceq \Omega(EC_b, t, args)\end{array}$$

Because $EC_a \preceq EC_b$, it suffices to prove that the preconditions of Theorem 7.22 are satisfied. The preconditions all follow directly from the preconditions of this lemma and the fact that $EC_b = \varepsilon(t.m(arg_1, \ldots, arg_n))$. □

7.13 The $\preceq$ relation implies the δ 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.

7.13.1 *Absolute Exception Declarations*

LEMMA 7.25.

$$\begin{array}{c}\Phi((P_a, B_a), E, \emptyset) \preceq (P_b, B_b) \\ \Downarrow \\ \delta((P_a, B_a), E) \Rightarrow \delta((P_b, B_b), E)\end{array}$$

PROOF.

$$\begin{array}{c}\Phi((P_a, B_a), E, \emptyset) \preceq (P_b, B_b) \\ \Downarrow (definition\ of\ \Phi\ and\ \preceq) \\ ((P_a \sqcap E) - B_a) \sqsubseteq (P_b - B_b) \\ \Downarrow (definition\ of\ \subseteq) \\ E \trianglelefteq ((P_a \sqcap E) - B_a) \Rightarrow E \trianglelefteq (P_b - B_b) \\ \Downarrow (definition\ of\ \trianglelefteq\ and \sqcap) \\ E \trianglelefteq (P_a - B_a) \Rightarrow E \trianglelefteq (P_b - B_b) \\ \Downarrow (definition\ of\ \delta) \\ \delta((P_a, B_a), E) \Rightarrow \delta((P_b, B_b), E)\end{array}$$

□

7.13.2 *Anchored Exception Declarations*

LEMMA 7.26.

$$\begin{array}{c}\Phi(AED_a, E, \emptyset) \preceq AED_b \\ \Downarrow \\ (\delta(AED_a, E) \Rightarrow \delta(AED_b, E))\end{array}$$

PROOF.

$$\begin{array}{c}\Phi(AED_a, E, \emptyset) \preceq AED_b \\ \Downarrow (Lemma\ 7.8) \\ method(AED_a) <: method(AED_b) \\ \Downarrow (Rule\ \textbf{@\#@\#@@}) \\ \varepsilon(AED_a) \preceq \varepsilon(AED_b)\end{array}$$

We will now use Theorems 7.17 and 7.22.

$$\begin{array}{c}\Upsilon(\Phi(AED_a, E, \emptyset)) \preceq \Upsilon(AED_b) \\ \Updownarrow \\ \Omega(\Phi(\varepsilon(AED_a), (P_a \sqcap E), B_a), t_a, args_a) \preceq \\ \Omega(\Phi(\varepsilon(AED_b), P_b, B_b), t_b, args_b)\end{array}$$

The preconditions of Theorems 7.17 and 7.22 are satisfied because $AED_a \preceq AED_b$, and thus $\Upsilon(\Phi(AED_a, E)) \preceq \Upsilon(AED_b)$.

$$\begin{array}{c}\Upsilon(\Phi(AED_a, E, \emptyset)) \preceq \Upsilon(AED_b) \\ \Downarrow (Induction\ on\ Theorem\ 7.27) \\ \delta(\Upsilon(\Phi(AED_a, E, \emptyset)), E) \Rightarrow \delta(\Upsilon(AED_b), E) \\ \Downarrow (definition\ of\ \delta) \\ \delta(\Phi(AED_a, E, \emptyset), E) \Rightarrow \delta(AED_b, E) \\ \Updownarrow (Lemma\ 7.5) \\ \delta(AED_a, E) \Rightarrow \delta(AED_b, E)\end{array}$$

For the induction, we perform a one-level expansion. Because of the no-loops rule, this induction will always end. The base cases are exception clauses that only contain absolute exception declarations. For such exception clauses, Theorem 7.27 is proven by Lemma 7.25. □

7.13.3 *Exception Clauses*

THEOREM 7.27.

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

PROOF.

$$\begin{array}{c}EC_a \preceq EC_b \\ \Updownarrow (definition\ of\ \preceq) \\ \left(\begin{array}{c}(\forall (P_a, B_a) \in EC_a, \forall E, \delta((P_a, B_a), E) : \\ \exists (P_b, B_b) \in EC_b : \Phi((P_a, B_a), E) \preceq (P_b, B_b)) \wedge \\ (\forall AED_a \in EC_a, \forall E, \delta(AED_a, E) : \exists AED_b \in EC_b : \\ \Phi(AED_a, E, \emptyset) \preceq AED_b \vee \\ (\Phi(\Upsilon(AED_a), E, \emptyset) \preceq EC_b))\end{array}\right)\end{array}$$

As a result, for every E, we can find ABS_{b,x_i} and AED_{b,y_1} such that:

$$\begin{array}{c}\Downarrow (Lemmas\ 7.25\ and\ 7.26\ and\ 7.5) \\ \left(\begin{array}{c}\delta(ABS_{a,1}, E) \Rightarrow \delta(ABS_{b,x_1}, E) \wedge \\ \ldots \wedge \\ \delta(ABS_{a,n}, E) \Rightarrow \delta(ABS_{b,x_n}, E) \wedge \\ \left(\begin{array}{c}(\delta(AED_{a,1}) \Rightarrow \delta(AED_{b,y_1}) \vee \\ \forall E, \delta(AED_{a,1}, E) : \Phi(\Upsilon(AED_{a,1}), E, \emptyset) \preceq EC_b)\end{array}\right) \wedge \\ \ldots \wedge \\ \left(\begin{array}{c}(\delta(AED_{a,m}) \Rightarrow \delta(AED_{b,y_m}) \vee \\ \forall E, \delta(AED_{a,1}, E) : \Phi(\Upsilon(AED_{a,m}), E, \emptyset) \preceq EC_b)\end{array}\right)\end{array}\right)\end{array}$$

For the induction below, we perform a one-level expansion. Because of the no-loops rule, this induction will always end. The base cases are exception clauses that only contain

absolute exception declarations. For such exception clauses, theorem 7.27 is proven by Lemma 7.25.

$$\Downarrow (Lemma\ 7.5\ and\ induction\ on\ Theorem\ 7.27)$$
$$\left(\begin{array}{c}
\delta(ABS_{a,1},E) \Rightarrow \delta(ABS_{b,x_1},E) \wedge \\
\ldots \wedge \\
\delta(ABS_{a,n},E) \Rightarrow \delta(ABS_{b,x_n},E) \wedge \\
\left(\begin{array}{c}(\delta(AED_{a,1},E) \Rightarrow \delta(AED_{b,y_1},E) \vee \\ \delta(\Upsilon(AED_{a,1}),E) \Rightarrow \delta(EC_b,E))\end{array}\right) \wedge \\
\ldots \wedge \\
\left(\begin{array}{c}(\delta(AED_{a,m},E) \Rightarrow \delta(AED_{b,y_m},E) \vee \\ \delta(\Upsilon(AED_{a,m}),E) \Rightarrow \delta(EC_b,E))\end{array}\right)
\end{array}\right)$$

$$\Downarrow (Definition\ of\ \delta)$$
$$\left(\begin{array}{c}
\delta(ABS_{a,1},E) \Rightarrow \delta(ABS_{b,x_1},E) \wedge \\
\ldots \wedge \\
\delta(ABS_{a,n},E) \Rightarrow \delta(ABS_{b,x_n},E) \wedge \\
\left(\begin{array}{c}(\delta(AED_{a,1},E) \Rightarrow \delta(AED_{b,y_1},E) \vee \\ \delta(AED_{a,1},E) \Rightarrow \delta(EC_b,E))\end{array}\right) \wedge \\
\ldots \wedge \\
\left(\begin{array}{c}(\delta(AED_{a,m},E) \Rightarrow \delta(AED_{b,y_m},E) \vee \\ \delta(AED_{a,m},E) \Rightarrow \delta(EC_b,E))\end{array}\right)
\end{array}\right)$$
$$\Downarrow (definition\ of\ \delta)$$
$$\delta(EC_a,E) \Rightarrow \delta(EC_b,E)$$

□

7.14 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 7.28.

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

PROOF.

$$\begin{array}{c}
\delta(\Phi((P,B),P_n,B_n),E) \Rightarrow \\
\delta((P \sqcap P_n, B \sqcup B_n),E) \Rightarrow \\
E \preceq ((P \sqcap P_n) - (B \sqcup B_n)) \Rightarrow \\
E \preceq P \wedge E \preceq P_n \wedge E \not\preceq B \wedge E \not\preceq B_n \Rightarrow \\
E \preceq P \wedge E \not\preceq B \Rightarrow \\
\delta((P,B),E)
\end{array}$$

□

LEMMA 7.29.

$$\delta(\Phi(anchor), P_n, B_n), E) \Rightarrow \delta(anchor, E)$$

PROOF. Let $anchor = like\ t.m(args) \trianglelefteq P \not\trianglelefteq B$.

$$\begin{array}{c}
\delta(\Phi(anchor, P_n, B_n), E) \Rightarrow \delta(anchor, E) \\
\Updownarrow \\
\delta(\Upsilon(\Phi(anchor, P_n, B_n)), E) \Rightarrow \delta(\Upsilon(anchor), E) \\
\Updownarrow (\Phi\ does\ not\ affect\ the\ method\ expression) \\
\delta(\Omega(\Phi(\varepsilon(anchor), P \sqcap P_n, B \sqcup B_n), t, args), E) \Rightarrow \\
\delta(\Omega(\Phi(\varepsilon(anchor), P, B), t, args), E) \\
\Updownarrow (Lemma\ 7.2\ and\ definition\ of\ \Phi) \\
\delta(\Phi(\Omega(\Phi(\varepsilon(anchor), P, B), t, args), P_n, B_n), E) \Rightarrow \\
\delta(\Phi(\Omega(\Phi(\varepsilon(anchor), P, B), t, args), \divideontimes, \emptyset), E)
\end{array}$$

We now use Theorems 7.22 and 7.17.

—The first three preconditions of 7.22 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 7.17 are satisfied because $(P_n - B_n)) \preceq (\divideontimes - \emptyset)$.

As a result, we know that:

$$\begin{array}{c}
\Phi(\Omega(\Phi(\varepsilon(anchor), P, B), t, args), P_n, B_n) \preceq \\
\Phi(\Omega(\Phi(\varepsilon(anchor), P, B), t, args), \divideontimes, \emptyset)
\end{array}$$

Applying Theorem 7.27 completes the proof. □

LEMMA 7.30.

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

PROOF.

$$\begin{array}{c}
\delta(\Phi(EC, P, B), E) \\
\Updownarrow \\
\delta(\Phi(\{ABS_1, \ldots, ABS_n\ AED_1, \ldots, AED_m\}, P, B), E) \\
\Updownarrow (definition\ of\ \Omega) \\
\delta(\{\Phi(ABS_1, P, B), \ldots, \Phi(ABS_n, P, B) \\
\Phi(AED_1, pre, args), \ldots, \Phi(AED_m, pre, args)\}, E) \\
\Updownarrow (definition\ of\ \delta) \\
\delta(\Phi(ABS_1, P, B)E) \vee \ldots \vee \delta(\Phi(ABS_n, P, B), E) \vee \\
\delta(\Phi(AED_1, P, B), E) \vee \ldots \vee \delta(\Phi(AED_m, P, B), E) \\
\Downarrow (Lemmas\ 7.28\ and\ 7.29)
\end{array}$$

$$\begin{array}{c}
\delta(ABS_1, E) \vee \ldots \vee \delta(ABS_n, E) \vee \\
\delta(AED_1, E) \vee \ldots \vee \delta(AED_m, E) \\
\Updownarrow \\
\delta(EC, E)
\end{array}$$

□

LEMMA 7.31.

$$ok_\Omega(args, (pre, this(AED)))$$
$$\Downarrow$$
$$\delta(\Omega(AED, pre, args), E) \Rightarrow \delta(AED, E)$$

PROOF. Let $AED = like\ t.m(a_1, \ldots, a_n) \trianglelefteq P \not\trianglelefteq B$.

$$\begin{array}{c}\delta(\Omega(AED, pre, args), E) \Rightarrow \delta(AED, E)\\ \Updownarrow (definition\ of\ \delta)\\ \delta(\Upsilon(\Omega(AED, pre, args)), E) \Rightarrow \delta(\Upsilon(AED), E)\\ \Updownarrow (definition\ of\ \Upsilon)\\ \delta(\Omega(\Phi(\varepsilon(\Omega(AED, pre, args)), P, B), \Omega(t, pre, args),\\ \Omega(a_1, pre, args) \ldots \Omega(a_n, pre, args)), E) \Rightarrow\\ \delta(\Omega(\Phi(\varepsilon(AED), P, B), t, args), E)\end{array}$$

Because $ok_\Omega(args, (pre, env(AED)))$, we know from Lemmas 7.7 and 7.19 that:

$$\begin{array}{c}\Gamma(\Omega(t, pre, args)) <: \Gamma(t) \wedge\\ \Gamma(\Omega(a_1, pre, args)) <: \Gamma(a_1) \wedge \ldots \Gamma(\Omega(a_n, pre, args)) <: \Gamma(a_n)\end{array}$$

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 **@@@@@@** applies

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

We now apply Theorems 7.17 and 7.22 to the arguments of δ in the implication above.

—The preconditions of Theorem 7.17 are satisfied because the arguments of Φ are identical.

—The first precondition of Theorem 7.22 follows from the application of Theorem 7.17. The second and third preconditions follow from Lemma 7.19. The last preconditions follow from the preconditions of this lemma, from Lemmas 7.7 and 7.19, 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:

$$\begin{array}{c}\Omega(\Phi(\varepsilon(\Omega(AED, pre, args)), P, B), \Omega(t, pre, args),\\ \Omega(a_1, pre, args) \ldots \Omega(a_n, pre, args)) \preceq\\ \Omega(\Phi(\varepsilon(AED), P, B), t, args)\end{array}$$

Applying Theorem 7.27 completes the proof. □

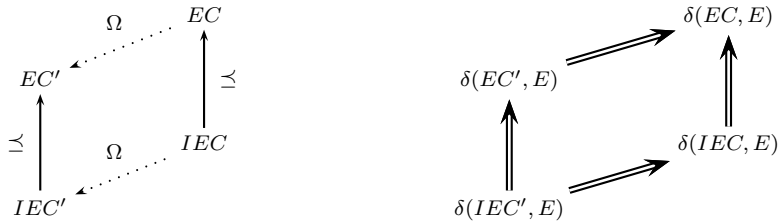
LEMMA 7.32.

$$ok_\Omega(args, (pre, env(EC)))$$
$$\Downarrow$$
$$\delta(\Omega(EC, pre, args), E) \Rightarrow \delta(EC, E)$$

PROOF. The proof of this lemma is nearly identical to that of Lemma 7.30. □

THEOREM 7.33.

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



18: bla

PROOF. This theorem follows directly from Lemmas 7.32 and 7.30. □

7.15 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.

Figure 18 illustrates the proof. The exception clause of the method is represented by EC, its implementation exception clause by IEC. We know from rule **@@@@@**that $IEC \preceq EC$, so Theorem 7.24 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 7.27 and Lemma 7.32, we can transform the left diagram into the right diagram. Theorem 7.27 ensures that $\delta(IEC, E) \Rightarrow \delta(EC, E)$ and $\delta(IEC', E) \Rightarrow \delta(EC', E)$. Lemma 7.32 ensures that $\delta(EC', E) \Rightarrow \delta(EC, E)$ and $\delta(IEC', E) \Rightarrow \delta(IEC, E)$. Both relation 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.

8. COMPARISON WITH TYPE ANCHORS

The anchoring technique has more impact on the exceptional return type than on the normal return type of a method. The reason for this is that a normal return value can be used through subsumption. The most specific type information is often not needed. For an exception, however, the general type is usually not sufficient [25]. In this case we need as much information as possible because, by the very nature of an exception handling mechanism, the signaller is not supposed to know how to handle it. Consequently, he cannot provide an exception that will handle itself, prohibiting the use of subsumption.

The conformance rule for anchored exception declarations is more flexible than the corresponding *conformance* rule for Eiffel type anchors. In Eiffel, the only type conform to `like anchor` is itself. The rule for anchored exception declarations leaves the opportunity to redefine a part of an exception clause by one or more stronger anchored declarations. The need for this is caused by the difference between the normal and exceptional behaviour of a method. Adding an extra layer of indirection (rule `2.b`) is useful for exceptions because some of them may be handled in the extra layer. For example, a redefined version

of the extra layer may declare that it cannot signal any checked exception at all although the method referenced by the original anchored declaration can. This is not possible for the normal return type since there must always be exactly one return type and that type has already been fixed. Both anchored declarations have a slightly different meaning. An Eiffel type anchor declares that the type is always the same as the type of the anchor, while an anchored exception declaration declares that it cannot signal exception when the anchor cannot.

The difference in the conformance rules results in a difference between the rules to prevents loops while following anchored declarations. The rule of Eiffel type anchors is weaker than the rule for anchored exception declarations because it is not allowed to redefine the type of an anchored declaration in Eiffel. As a result, it suffices to demand that there is no loop in the anchor chain.

9. CASE STUDY

An analysis of the core of Jnome [34] showed that while only 46 methods directly raise a `NotResolvedException`, there are more than 400 methods that only propagate this exception. A `NotResolvedException` signals an unexpected failure while looking up a named element, such as encountering multiple matches for an element when only a single element should match – the metamodel does not enforce validity at every moment for reasons of flexibility. When the element simply cannot be found, a null reference is returned.

Suppose that instead of returning a null reference, a checked exception must be signalled when an element cannot be found. This exception cannot be a subclass of `NotResolvedException` since it does not signal a failure caused by an 'invalid' instance of the metamodel. Just like `NotResolvedException`, the new exception cannot be handled by the metamodel itself, since it is up to the client of the metamodel to decide what to do when an element cannot be found. This means that all methods that previously propagated `NotResolvedException` must now also propagate `ElementNotFoundException`, resulting in modifying over 400 other methods. With anchored exception declarations, they would not have to be modified.

From the 110 `try-catch` statements in the code, only 30 actually handle exceptions. The other 80 `try-catch` statements are dummy constructions to filter out `Exception`.

10. RELATED WORK

Java was the first programming language seeking a compromise between robustness and flexibility by providing both checked and unchecked exceptions [9]. But as we showed in section 3, this solution is not sufficient.

Mikhailova and Romanovsky [24] provide support for evolution of the exceptional behaviour 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 is not compatible with its supermethods. 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 the rescue clause. The rescue clause allows a programmer to signal new exceptions for which a default handler can be provided, while anchored exception declarations can be used when such a handler cannot

be provided.

Romanovsky and Sandén [31] show that an exception handling mechanism should correspond to the features of the language. We have shown that the exception clause of object-oriented programming languages conflicts with the principle of abstraction, which is a fundamental concept of object-oriented programming. By solving this conflict, many problems with checked exceptions are solved.

Miller and Tripathi [25] analyse the conflicts between exception handling and object-oriented programming. Our paper is related to these conflicts in several ways. We showed there is a conflict between the principle of abstraction and the exception clauses of object-oriented programming languages. This conflict however is not a conflict with object-oriented programming itself, but with the incarnation of the exception clause in existing languages. Furthermore, by bringing context information into the exception clause, anchored exception declarations reduce – but do not eliminate – the conflict between *exception conformance* and *complete exception specification*. Specific information about the exception behaviour of an overriding method can still be used when the interface of its supermethod has a general exception for conformance reasons. The authors also argue that exception handling increases coupling in object-oriented programs. Anchored exception declarations increase coupling, but in a way that is beneficial for the programmer. With respect to adaptability, they decrease coupling. Last, Miller and Tripathi discuss the need for evolution of the exceptional behaviour of a method. They briefly suggest that a language should allow exception non-conformance and the ability to add exception handlers to existing code.

Lippert and Lopes [19] simplify exception handling by using aspect-oriented programming. Their approach focuses on removing redundant exception handlers, and can be used for adding the dummy exception handlers and propagating exceptions. Using aspect-oriented programming can be very useful when the exception handlers are meaningful, but for checked exceptions it does not solve the adaptability problem and the program still suffers from hazardous situations under evolution. Anchored exception declarations solve these aspects of exception handling in a better way.

Robillard and Murphy [30] developed a language-independent model for analysing the exception flow in object-oriented programs, along with a tool specifically for Java. Their model is based on the analysis of source code, and also takes unchecked exceptions into account. This paper also discusses the cost of modifying the exception clause of a method, and the use of unchecked exceptions as a result. In [29], they show that the difficulty in determining all exceptional conditions in advance gives rise to the need for evolution of the exceptional behaviour of a method.

Specification of the dependencies between methods has been presented by Helm et al. [13], by Lamping [16], and by Steyaert et al. [35]. They present their work using the normal behaviour of a method, but their techniques also apply to the exceptional behaviour of method. Anchored exception declarations provide these dependencies for the exceptional behaviour in a way that is verifiable by a compiler.

11. FUTURE WORK

While anchored exception declarations solve most of the problems of using checked exceptions, there is still room for improvement.

At this moment, an anchored exception declaration can limit the set of exceptions that are

propagated, but it cannot express the transformation of one type of exceptions into another, which can be necessary when crossing the boundaries of a component [29]. A construct to express this would allow for a more fine grained specification of the exceptional behaviour of a method, and could look like this:

NewException **like** *MethodExpression*
signals (*OldException*)

Additionally, the algorithm for calculating the implementation exception clause can be improved. For example, at this moment it does not retain information about the origin of a checked exception if it is caught, and then raised again. It will treat the raised exception as if it can be signalled at any time and discard possible anchor relations.

The expressiveness of anchored exception declarations is limited in the sense that they only take static type information into account. Information about the exact conditions under which certain exceptions can be signalled still have to be provided by specifications. Anchored exception declarations are complementary to traditional specifications, and can be added to existing specification languages, such as JML and Spec# [], in order to enrich their expressiveness regarding exception handling.

12. CONCLUSION

We have shown that problems with checked exceptions, like reduced adaptability and loss of context information, are caused by a conflict with the principle of abstraction. The relative nature of abstraction improves the adaptability of the implementation and post-conditions, while the absolute nature of traditional exception clauses does not offer any of those benefits.

By introducing anchored exception declarations, we have opened the road for a broader acceptance of checked exceptions. They bring the benefits of abstraction to the exception clause by allowing the exceptional behaviour of a method to be declared relative to other methods. This results in better adaptability of software, more elegant code, and eliminates most of the dangerous exception handlers.

We have defined the formal semantics of anchored exception declarations, and the rules they must adhere to in order to ensure compile-time safety, which we have proved. We have shown that anchored exception declarations do not violate the principle of information hiding when used properly, and have presented a guideline for when to use them, and when not to use them.

Finally, we have implemented anchored exception declarations in Cappuccino, an extension of ClassicJava. A translator validates Cappuccino programs and transforms them into plain Java programs.

13. ACKNOWLEDGEMENTS

We want to thank Nele Smeets and Tom Schrijvers for their invaluable feedback. We would also like to thank Bart Jacobs of the K.U.Leuven for pointing out the approach using generic parameters.

REFERENCES

ALDRICH, J., AND DONNELLY, K. Selective open recursion: Modular reasoning about components and inheritance. In *Proc. FSE 2004 Workshop on Specification and Verification of Component-Based Systems.*

BARNETT, M., LEINO, K. R. M., AND SCHULTE, W. The spec# programming system: An overview. In *CASSIS 2004 proceedings* (2004).

BUHR, P. A., AND MOK, W. Y. R. Advanced exception handling mechanisms. *IEEE Trans. Softw. Eng. 26*, 9 (2000), 820–836.

CLIFTON, C., LEAVENS, G. T., CHAMBERS, C., AND MILLSTEIN, T. MultiJava: Modular open classes and symmetric multiple dispatch for Java. In *OOPSLA 2000 Conference on Object-Oriented Programming, Systems, Languages, and Applications, Minneapolis, Minnesota* (2000), vol. 35(10), pp. 130–145.

DONY, C. Exception handling and object-oriented programming: towards a synthesis. In *Proceedings of the European conference on object-oriented programming on Object-oriented programming systems, languages, and applications* (1990), ACM Press, pp. 322–330.

ECMA TECHNICAL COMMITTEE 39 (TC39) TASK GROUP 2 (TG2). *C# Language Specification*, 2 ed. ECMA, December 2002.

FLATT, M., KRISHNAMURTHI, S., AND FELLEISEN, M. Classes and mixins. In *POPL '98: Proceedings of the 25th ACM SIGPLAN-SIGACT symposium on Principles of programming languages* (1998), ACM Press, pp. 171–183.

GAMMA, E., HELM, R., JOHNSON, R., AND VLISSIDES, J. *Design patterns: elements of reusable object-oriented software*. Addison-Wesley Longman Publishing Co., Inc., 1995.

GARCIA, A. F., RUBIRA, C. M. F., ROMANOVSKY, A., AND XU, J. A comparative study of exception handling mechanisms for building dependable object-oriented software. *The Journal of Systems and Software 59*, 2 (2001), 197–222.

GOODENOUGH, J. B. Exception handling: issues and a proposed notation. *Commun. ACM 18*, 12 (1975), 683–696.

GOSLING, J., ET AL. *The Java Language Specification, Second Edition*. Addison-Wesley Longman Publishing Co., Inc., 2000.

HEJLSBERG, A. The trouble with checked exceptions. http://www.artima.com/intv/handcuffs.html.

HELM, R., HOLLAND, I. M., AND GANGOPADHYAY, D. Contracts: specifying behavioral compositions in object-oriented systems. In *Proceedings of the European conference on object-oriented programming on Object-oriented programming systems, languages, and applications* (1990), ACM Press, pp. 169–180.

IGARASHI, A., PIERCE, B. C., AND WADLER, P. Featherweight java: a minimal core calculus for java and gj. *ACM Trans. Program. Lang. Syst. 23*, 3 (2001), 396–450.

JACOBS, B. A formalisation of java's exception mechanism. In *ESOP '01: Proceedings of the 10th European Symposium on Programming Languages and Systems* (2001), Springer-Verlag, pp. 284–301.

LAMPING, J. Typing the specialization interface. In *Proceedings of the eighth annual conference on Object-oriented programming systems, languages, and applications* (1993), ACM Press, pp. 201–214.

LEAVENS, G. T., BAKER, A. L., AND RUBY, C. Preliminary design of JML: A behavioral interface specification language for Java. Tech. Rep. 98-06i, 2000.

LEINO, K. R. M., AND SCHULTE, W. Exception safety for C#. In *Proceedings, Software Engineering and Formal Methods (SEFM), Beijing, China* (2004), J. Cuellar and Z. Liu, Eds., IEEE Press.

LIPPERT, M., AND LOPES, C. A study on exception detection and handling using aspect-oriented programming. Tech. rep., Xerox PARC, 1999.

LISKOV, B. *Abstraction and specification in program development*. MIT Press, 1986.

LISKOV, B., AND WING, J. Family values: A behavioral notion of subtyping. Tech. Rep. MIT/LCS/TR-562b, 1993.

MEYER, B. *Object-oriented software construction (2nd ed.)*. Prentice-Hall, Inc., 1997.

MEYER, B. Overloading vs. object technology. *Journal of Object-Oriented Programming* (October 2001).

MIKHAILOVA, A., AND ROMANOVSKY, A. Supporting evolution of interface exceptions. 94–110.

MILLER, R., AND TRIPATHI, A. Issues with exception handling in object-oriented systems. In *Proceedings of the European Conference on Object-Oriented Programming (ECOOP '97)* (Jyväskylä, Finland, June 1997), vol. 1241 of *LNCS*, Springer, p. 85.

PARNAS, D. L. On the criteria to be used in decomposing systems into modules. 411–427.

POTTER, B., SINCLAIR, J., AND TILL, D. *An introduction to formal specification and Z*. Prentice-Hall, Inc., 1991.

ROBILLARD, M. P., AND MURPHY, G. C. Analyzing exception flow in Java programs. In *Software Engineering – ESEC/FSE'99* (September 1999), vol. 1687 of *Lecture Notes in Computer Science*, Springer-Verlag, pp. 322–337.

ROBILLARD, M. P., AND MURPHY, G. C. Designing robust java programs with exceptions. In *Proceedings of the 8th ACM SIGSOFT international symposium on Foundations of software engineering* (2000), ACM Press, pp. 2–10.

ROBILLARD, M. P., AND MURPHY, G. C. Static analysis to support the evolution of exception structure in object-oriented systems. *ACM Trans. Softw. Eng. Methodol. 12*, 2 (2003), 191–221.

ROMANOVSKY, A., AND SANDÉN, B. Except for exception handling .... *Ada Lett. XXI*, 3 (2001), 19–25.

RYDER, B. G., SMITH, D., KREMER, U., GORDON, M., AND SHAH, N. A static study of java exceptions using JESP. In *Computational Complexity* (2000), pp. 67–81.

RYU, S. Exception analysis for multithreaded java programs.

SMEETS, N., AND VAN DOOREN, M. Jnome, 2004. http://www.jnome.org.

STEYAERT, P., LUCAS, C., MENS, K., AND D'HONDT, T. Reuse contracts: managing the evolution of reusable assets. In *Proceedings of the 11th ACM SIGPLAN conference on Object-oriented programming, systems, languages, and applications* (1996), ACM Press, pp. 268–285.

VAN DOOREN, M., AND STEEGMANS, E. Combining the robustness of checked exceptions with the flexibility of unchecked exceptions using anchored exception declarations. Tech. rep., Katholieke Universiteit Leuven, 2004.

VAN ROSSUM, G. Python reference manual. Report CS-R9525, Apr. 1995.

WINSTON, W. *Operations Research: Applications and Algorithms*. Duxbury, 2003.

YEMINI, S., AND BERRY, D. M. A modular verifiable exception handling mechanism. *ACM Trans. Program. Lang. Syst. 7*, 2 (1985), 214–243.

YEMINI, S., AND BERRY, D. M. An axiomatic treatment of exception handling in an expression-oriented language. *ACM Trans. Program. Lang. Syst. 9*, 3 (1987), 390–407.